The Python Library Reference

发行版本 *3.13.3*

Guido van Rossum and the Python development team

四月 27, 2025

Python Software Foundation
Email: docs@python.org

Contents

1 概述 **3**
- 1.1 可用性注释 . . . 3
  - 1.1.1 WebAssembly 平台 . . . 4
  - 1.1.2 移动平台 . . . 4

2 内置函数 **7**

3 内置常量 **33**
- 3.1 由 site 模块添加的常量 . . . 34

4 内置类型 **35**
- 4.1 逻辑值检测 . . . 35
- 4.2 布尔运算 --- and, or, not . . . 35
- 4.3 比较运算 . . . 36
- 4.4 数字类型 --- int, float, complex . . . 36
  - 4.4.1 整数类型的按位运算 . . . 37
  - 4.4.2 整数类型的附加方法 . . . 38
  - 4.4.3 浮点类型的附加方法 . . . 40
  - 4.4.4 数字类型的哈希运算 . . . 41
- 4.5 布尔类型 - bool . . . 42
- 4.6 迭代器类型 . . . 42
  - 4.6.1 生成器类型 . . . 43
- 4.7 序列类型 --- list, tuple, range . . . 43
  - 4.7.1 通用序列操作 . . . 43
  - 4.7.2 不可变序列类型 . . . 45
  - 4.7.3 可变序列类型 . . . 45
  - 4.7.4 列表 . . . 45
  - 4.7.5 元组 . . . 46
  - 4.7.6 range 对象 . . . 47
- 4.8 文本序列类型 --- str . . . 48
  - 4.8.1 字符串的方法 . . . 49
  - 4.8.2 格式化字符串字面值（f-字符串） . . . 57
  - 4.8.3 printf 风格的字符串格式化 . . . 58
- 4.9 二进制序列类型 --- bytes, bytearray, memoryview . . . 60
  - 4.9.1 bytes 对象 . . . 60
  - 4.9.2 bytearray 对象 . . . 61
  - 4.9.3 bytes 和 bytearray 操作 . . . 62
  - 4.9.4 printf 风格的字节串格式化 . . . 72
  - 4.9.5 内存视图 . . . 75
- 4.10 集合类型 --- set, frozenset . . . 81
- 4.11 映射类型 --- dict . . . 83

4.11.1 字典视图对象 . . . 87
4.12 上下文管理器类型 . . . 88
4.13 类型注解的类型 --- Generic Alias 、Union . . . 89
4.13.1 GenericAlias 类型 . . . 89
4.13.2 union 类型 . . . 92
4.14 其他内置类型 . . . 94
4.14.1 模块 . . . 94
4.14.2 类与类实例 . . . 94
4.14.3 函数 . . . 94
4.14.4 方法 . . . 95
4.14.5 代码对象 . . . 95
4.14.6 类型对象 . . . 95
4.14.7 空对象 . . . 95
4.14.8 省略符对象 . . . 95
4.14.9 未实现对象 . . . 96
4.14.10 内部对象 . . . 96
4.15 特殊属性 . . . 96
4.16 整数字符串转换长度限制 . . . 96
4.16.1 受影响的 API . . . 97
4.16.2 配置限制值 . . . 97
4.16.3 推荐配置 . . . 98

5 内置异常 **99**
5.1 异常上下文 . . . 99
5.2 从内置异常继承 . . . 100
5.3 基类 . . . 100
5.4 具体异常 . . . 101
5.4.1 OS 异常 . . . 106
5.5 警告 . . . 107
5.6 异常组 . . . 108
5.7 异常层次结构 . . . 109

6 文本处理服务 **111**
6.1 `string` --- 常见的字符串操作 . . . 111
6.1.1 字符串常量 . . . 111
6.1.2 自定义字符串格式化 . . . 112
6.1.3 格式字符串语法 . . . 113
6.1.4 模板字符串 . . . 120
6.1.5 辅助函数 . . . 121
6.2 `re` --- 正则表达式操作 . . . 122
6.2.1 正则表达式语法 . . . 122
6.2.2 模块内容 . . . 128
6.2.3 正则表达式对象（正则对象） . . . 134
6.2.4 匹配对象 . . . 135
6.2.5 正则表达式例子 . . . 138
6.3 `difflib` --- 计算差异的辅助工具 . . . 143
6.3.1 SequenceMatcher 对象 . . . 147
6.3.2 SequenceMatcher 的示例 . . . 149
6.3.3 Differ 对象 . . . 150
6.3.4 Differ 示例 . . . 150
6.3.5 difflib 的命令行接口 . . . 151
6.3.6 ndiff 示例 . . . 152
6.4 `textwrap` --- 文本自动换行与填充 . . . 154
6.5 `unicodedata` --- Unicode 数据库 . . . 158
6.6 `stringprep` --- 因特网字符串预处理 . . . 159
6.7 `readline` --- GNU readline 接口 . . . 161
6.7.1 初始化文件 . . . 161
6.7.2 行缓冲区 . . . 161

6.7.3 历史文件 . . . 162
6.7.4 历史列表 . . . 162
6.7.5 启动钩子 . . . 163
6.7.6 Completion . . . 163
6.7.7 示例 . . . 163
6.8 rlcompleter --- 用于 GNU readline 的补全函数 . . . 165

7 二进制数据服务 167
7.1 struct --- 将字节串解读为打包的二进制数据 . . . 167
7.1.1 函数和异常 . . . 167
7.1.2 格式字符串 . . . 168
7.1.3 应用 . . . 172
7.1.4 类 . . . 173
7.2 codecs --- 编解码器注册和相关基类 . . . 173
7.2.1 编解码器基类 . . . 176
7.2.2 编码格式与 Unicode . . . 182
7.2.3 标准编码 . . . 183
7.2.4 Python 专属的编码格式 . . . 186
7.2.5 encodings.idna --- 应用程序中的国际化域名 . . . 188
7.2.6 encodings.mbcs --- Windows ANSI 代码页 . . . 188
7.2.7 encodings.utf_8_sig --- 带 BOM 签名的 UTF-8 编解码器 . . . 189

8 数据类型 191
8.1 datetime --- 基本日期和时间类型 . . . 191
8.1.1 感知型对象和简单型对象 . . . 192
8.1.2 常量 . . . 192
8.1.3 有效的类型 . . . 192
8.1.4 timedelta 类对象 . . . 193
8.1.5 date 对象 . . . 196
8.1.6 datetime 对象 . . . 201
8.1.7 time 对象 . . . 212
8.1.8 tzinfo 对象 . . . 215
8.1.9 timezone 对象 . . . 221
8.1.10 strftime() 和 strptime() 的行为 . . . 222
8.2 zoneinfo --- IANA 时区支持 . . . 226
8.2.1 使用 ZoneInfo . . . 226
8.2.2 数据源 . . . 227
8.2.3 ZoneInfo 类 . . . 228
8.2.4 函数 . . . 230
8.2.5 全局变量 . . . 230
8.2.6 异常与警告 . . . 230
8.3 calendar --- 通用日历相关函数 . . . 230
8.3.1 命令行用法 . . . 236
8.4 collections --- 容器数据类型 . . . 238
8.4.1 ChainMap 对象 . . . 238
8.4.2 Counter 对象 . . . 241
8.4.3 deque 对象 . . . 243
8.4.4 defaultdict 对象 . . . 247
8.4.5 namedtuple() 命名元组的工厂函数 . . . 248
8.4.6 OrderedDict 对象 . . . 251
8.4.7 UserDict 对象 . . . 253
8.4.8 UserList 对象 . . . 253
8.4.9 UserString 对象 . . . 254
8.5 collections.abc --- 容器的抽象基类 . . . 254
8.5.1 容器抽象基类 . . . 255
8.5.2 多项集抽象基类 -- 详细描述 . . . 257
8.5.3 例子和配方 . . . 259
8.6 heapq --- 堆队列算法 . . . 260

8.6.1 基本示例 . . . 261
8.6.2 优先队列实现说明 . . . 261
8.6.3 理论 . . . 262
8.7 bisect --- 数组二分算法 . . . 263
8.7.1 性能说明 . . . 264
8.7.2 搜索有序列表 . . . 264
8.7.3 例子 . . . 265
8.8 array --- 高效的数字值数组 . . . 266
8.9 weakref --- 弱引用 . . . 269
8.9.1 弱引用对象 . . . 272
8.9.2 示例 . . . 274
8.9.3 终结器对象 . . . 274
8.9.4 比较终结器与 __del__() 方法 . . . 275
8.10 types --- 动态类型创建和内置类型名称 . . . 276
8.10.1 动态类型创建 . . . 276
8.10.2 标准解释器类型 . . . 277
8.10.3 附加工具类和函数 . . . 281
8.10.4 协程工具函数 . . . 281
8.11 copy --- 浅层及深层拷贝操作 . . . 282
8.12 pprint --- 数据美化输出 . . . 283
8.12.1 函数 . . . 283
8.12.2 PrettyPrinter 对象 . . . 284
8.12.3 示例 . . . 285
8.13 reprlib --- 替代性 repr() 实现 . . . 288
8.13.1 Repr 对象 . . . 289
8.13.2 子类化 Repr 对象 . . . 291
8.14 enum --- 对枚举的支持 . . . 291
8.14.1 模块内容 . . . 292
8.14.2 数据类型 . . . 293
8.14.3 工具与装饰器 . . . 304
8.14.4 备注 . . . 305
8.15 graphlib --- 操作类似图的结构的功能 . . . 305
8.15.1 异常 . . . 308

9 数字和数学模块 309
9.1 numbers --- 数字抽象基类 . . . 309
9.1.1 数字的层次 . . . 309
9.1.2 有关类型实现的注释 . . . 310
9.2 math --- 数学函数 . . . 312
9.2.1 数论函数 . . . 313
9.2.2 浮点算数 . . . 314
9.2.3 浮点操作函数 . . . 315
9.2.4 幂、指数和对数函数 . . . 317
9.2.5 加总和乘积函数 . . . 318
9.2.6 角度转换 . . . 318
9.2.7 三角函数 . . . 319
9.2.8 双曲函数 . . . 319
9.2.9 特殊函数 . . . 319
9.2.10 常量 . . . 320
9.3 cmath --- 针对复数的数学函数 . . . 321
9.3.1 到极坐标和从极坐标的转换 . . . 322
9.3.2 幂函数与对数函数 . . . 322
9.3.3 三角函数 . . . 323
9.3.4 双曲函数 . . . 323
9.3.5 分类函数 . . . 323
9.3.6 常量 . . . 324
9.4 decimal --- 十进制定点和浮点算术 . . . 325
9.4.1 快速入门教程 . . . 326

9.4.2 Decimal 对象 . . . 329
9.4.3 上下文对象 . . . 335
9.4.4 常量 . . . 341
9.4.5 舍入模式 . . . 342
9.4.6 信号 . . . 342
9.4.7 浮点数说明 . . . 344
9.4.8 使用线程 . . . 345
9.4.9 例程 . . . 345
9.4.10 Decimal 常见问题 . . . 348
9.5 fractions --- 有理数 . . . 351
9.6 random --- 生成伪随机数 . . . 354
9.6.1 簿记功能 . . . 355
9.6.2 用于字节数据的函数 . . . 355
9.6.3 整数用函数 . . . 355
9.6.4 序列用函数 . . . 356
9.6.5 离散分布 . . . 357
9.6.6 实值分布 . . . 357
9.6.7 替代生成器 . . . 358
9.6.8 关于再现性的说明 . . . 359
9.6.9 例子 . . . 359
9.6.10 例程 . . . 361
9.6.11 命令行用法 . . . 362
9.6.12 命令行示例 . . . 363
9.7 statistics --- 数字统计函数 . . . 363
9.7.1 平均值以及对中心位置的评估 . . . 364
9.7.2 对分散程度的评估 . . . 364
9.7.3 对两个输入之间关系的统计 . . . 364
9.7.4 函数细节 . . . 364
9.7.5 异常 . . . 373
9.7.6 NormalDist 对象 . . . 373
9.7.7 例子和配方 . . . 374

10 函数式编程模块 377
10.1 itertools --- 为高效循环创建迭代器的函数 . . . 377
10.1.1 Itertool 函数 . . . 378
10.1.2 itertools 配方 . . . 388
10.2 functools ——高阶函数，以及可调用对象上的操作 . . . 393
10.2.1 partial 对象 . . . 402
10.3 operator --- 标准运算符对应函数 . . . 402
10.3.1 将运算符映射到函数 . . . 407
10.3.2 原地运算符 . . . 407

11 文件和目录访问 411
11.1 pathlib --- 面向对象的文件系统路径 . . . 411
11.1.1 基础使用 . . . 412
11.1.2 异常 . . . 413
11.1.3 纯路径 . . . 413
11.1.4 具体路径 . . . 421
11.1.5 模式语言 . . . 432
11.1.6 与 glob 模块的比较 . . . 433
11.1.7 与 os 和 os.path 模块的比较 . . . 433
11.2 os.path --- 常用的路径操作 . . . 434
11.3 stat --- 解释 stat() 的结果 . . . 440
11.4 filecmp --- 文件和目录比较 . . . 446
11.4.1 dircmp 类 . . . 447
11.5 tempfile --- 生成临时文件和目录 . . . 448
11.5.1 例子 . . . 452
11.5.2 已弃用的函数和变量 . . . 453

11.6 glob --- Unix 风格的路径名模式扩展 . . . 453
11.6.1 例子 . . . 455
11.7 fnmatch --- Unix 文件名模式匹配 . . . 456
11.8 linecache --- 随机访问文本行 . . . 457
11.9 shutil --- 高层级文件操作 . . . 458
11.9.1 目录和文件操作 . . . 458
11.9.2 归档操作 . . . 463
11.9.3 查询输出终端的尺寸 . . . 467

12 数据持久化 **469**
12.1 pickle --- Python 对象序列化 . . . 469
12.1.1 与其他 Python 模块间的关系 . . . 469
12.1.2 数据流格式 . . . 470
12.1.3 模块接口 . . . 471
12.1.4 可以被封存/解封的对象 . . . 474
12.1.5 封存类实例 . . . 475
12.1.6 类型，函数和其他对象的自定义归约 . . . 480
12.1.7 外部缓冲区 . . . 480
12.1.8 限制全局变量 . . . 482
12.1.9 性能 . . . 483
12.1.10 例子 . . . 483
12.2 copyreg --- 注册 pickle 支持函数 . . . 484
12.2.1 示例 . . . 484
12.3 shelve --- Python 对象持久化 . . . 484
12.3.1 限制 . . . 485
12.3.2 示例 . . . 486
12.4 marshal --- 内部 Python 对象序列化 . . . 487
12.5 dbm --- Unix ” 数据库” 接口 . . . 488
12.5.1 dbm.sqlite3 --- 针对 dbm 的 SQLite 后端 . . . 490
12.5.2 dbm.gnu --- GNU 数据库管理器 . . . 490
12.5.3 dbm.ndbm --- 新数据库管理器 . . . 492
12.5.4 dbm.dumb --- 便携式 DBM 实现 . . . 493
12.6 sqlite3 --- SQLite 数据库的 DB-API 2.0 接口 . . . 494
12.6.1 教程 . . . 494
12.6.2 参考 . . . 496
12.6.3 常用方案指引 . . . 516
12.6.4 说明 . . . 523

13 数据压缩和存档 **525**
13.1 zlib --- 与 **gzip** 兼容的压缩 . . . 525
13.2 gzip --- 对 **gzip** 文件的支持 . . . 528
13.2.1 用法示例 . . . 531
13.2.2 命令行界面 . . . 531
13.3 bz2 --- 对 **bzip2** 压缩算法的支持 . . . 532
13.3.1 文件压缩和解压 . . . 532
13.3.2 增量压缩和解压 . . . 534
13.3.3 一次性压缩或解压缩 . . . 535
13.3.4 用法示例 . . . 535
13.4 lzma --- 使用 LZMA 算法进行压缩 . . . 536
13.4.1 读写压缩文件 . . . 536
13.4.2 在内存中压缩和解压缩数据 . . . 537
13.4.3 杂项 . . . 539
13.4.4 指定自定义的过滤器链 . . . 540
13.4.5 例子 . . . 540
13.5 zipfile --- 操作 ZIP 归档文件 . . . 541
13.5.1 ZipFile 对象 . . . 543
13.5.2 Path 对象 . . . 547
13.5.3 PyZipFile 对象 . . . 548

13.5.4 ZipInfo 对象 . . . 549
13.5.5 命令行接口 . . . 551
13.5.6 解压缩的障碍 . . . 551
13.6 tarfile --- 读写 tar 归档文件 . . . 552
13.6.1 TarFile 对象 . . . 555
13.6.2 TarInfo 对象 . . . 559
13.6.3 解压缩过滤器 . . . 561
13.6.4 命令行接口 . . . 564
13.6.5 例子 . . . 565
13.6.6 受支持的 tar 格式 . . . 566
13.6.7 Unicode 问题 . . . 566

14 文件格式 **569**
14.1 csv --- CSV 文件读写 . . . 569
14.1.1 模块内容 . . . 569
14.1.2 变种与格式参数 . . . 573
14.1.3 Reader 对象 . . . 574
14.1.4 Writer 对象 . . . 574
14.1.5 例子 . . . 575
14.2 configparser --- 配置文件解析器 . . . 576
14.2.1 快速起步 . . . 576
14.2.2 支持的数据类型 . . . 578
14.2.3 回退值 . . . 578
14.2.4 受支持的 INI 文件结构 . . . 579
14.2.5 未命名小节 . . . 580
14.2.6 值的插值 . . . 580
14.2.7 映射协议访问 . . . 581
14.2.8 定制解析器行为 . . . 582
14.2.9 旧式 API 示例 . . . 586
14.2.10 ConfigParser 对象 . . . 587
14.2.11 RawConfigParser 对象 . . . 591
14.2.12 异常 . . . 592
14.3 tomllib --- 解析 TOML 文件 . . . 592
14.3.1 例子 . . . 593
14.3.2 转换表 . . . 594
14.4 netrc --- netrc 文件处理 . . . 594
14.4.1 netrc 对象 . . . 595
14.5 plistlib --- 生成与解析 Apple .plist 文件 . . . 595
14.5.1 例子 . . . 597

15 加密服务 **599**
15.1 hashlib --- 安全哈希与消息摘要 . . . 599
15.1.1 哈希算法 . . . 599
15.1.2 用法 . . . 600
15.1.3 构造器 . . . 600
15.1.4 属性 . . . 601
15.1.5 哈希对象 . . . 601
15.1.6 SHAKE 可变长度摘要 . . . 602
15.1.7 文件哈希 . . . 602
15.1.8 密钥派生 . . . 603
15.1.9 BLAKE2 . . . 603
15.2 hmac --- 用于消息验证的密钥哈希 . . . 610
15.3 secrets --- 生成管理密码的安全随机数 . . . 611
15.3.1 随机数 . . . 612
15.3.2 生成 Token . . . 612
15.3.3 其他功能 . . . 613
15.3.4 应用技巧与最佳实践 . . . 613

16 通用操作系统服务 **615**
16.1 os --- 多种操作系统接口 . . . 615
16.1.1 文件名，命令行参数，以及环境变量。 . . . 616
16.1.2 Python UTF-8 模式 . . . 616
16.1.3 进程参数 . . . 617
16.1.4 创建文件对象 . . . 624
16.1.5 文件描述符操作 . . . 624
16.1.6 文件和目录 . . . 636
16.1.7 进程管理 . . . 661
16.1.8 调度器接口 . . . 673
16.1.9 其他系统信息 . . . 674
16.1.10 随机数 . . . 676
16.2 io --- 处理流的核心工具 . . . 677
16.2.1 概述 . . . 677
16.2.2 文本编码格式 . . . 678
16.2.3 高阶模块接口 . . . 679
16.2.4 类的层次结构 . . . 680
16.2.5 性能 . . . 689
16.3 time --- 时间的访问和转换 . . . 690
16.3.1 函数 . . . 691
16.3.2 Clock ID 常量 . . . 699
16.3.3 时区常量 . . . 701
16.4 logging --- Python 的日志记录工具 . . . 701
16.4.1 记录器对象 . . . 702
16.4.2 日志级别 . . . 706
16.4.3 处理器对象 . . . 707
16.4.4 格式器对象 . . . 708
16.4.5 过滤器对象 . . . 710
16.4.6 LogRecord 属性 . . . 710
16.4.7 LogRecord 属性 . . . 711
16.4.8 LoggerAdapter 对象 . . . 712
16.4.9 线程安全 . . . 713
16.4.10 模块级函数 . . . 713
16.4.11 模块级属性 . . . 717
16.4.12 与警告模块集成 . . . 717
16.5 logging.config --- 日志记录配置 . . . 718
16.5.1 配置函数 . . . 718
16.5.2 安全考量 . . . 720
16.5.3 配置字典架构 . . . 720
16.5.4 配置文件格式 . . . 726
16.6 logging.handlers --- 日志处理器 . . . 729
16.6.1 StreamHandler . . . 729
16.6.2 FileHandler . . . 730
16.6.3 NullHandler . . . 730
16.6.4 WatchedFileHandler . . . 730
16.6.5 BaseRotatingHandler . . . 731
16.6.6 RotatingFileHandler . . . 732
16.6.7 TimedRotatingFileHandler . . . 733
16.6.8 SocketHandler . . . 734
16.6.9 DatagramHandler . . . 735
16.6.10 SysLogHandler . . . 735
16.6.11 NTEventLogHandler . . . 737
16.6.12 SMTPHandler . . . 738
16.6.13 MemoryHandler . . . 738
16.6.14 HTTPHandler . . . 739
16.6.15 QueueHandler . . . 739
16.6.16 QueueListener . . . 740
16.7 platform --- 访问底层平台的标识数据 . . . 742

16.7.1 跨平台 . . . 742
16.7.2 Java 平台 . . . 743
16.7.3 Windows 平台 . . . 744
16.7.4 macOS 平台 . . . 744
16.7.5 iOS 平台 . . . 744
16.7.6 Unix 平台 . . . 744
16.7.7 Linux 平台 . . . 745
16.7.8 Android 平台 . . . 745
16.8 errno --- 标准 errno 系统符号 . . . 745
16.9 ctypes --- Python 的外部函数库 . . . 754
16.9.1 ctypes 教程 . . . 754
16.9.2 ctypes 参考手册 . . . 770

17 Command Line Interface Libraries **785**
17.1 argparse --- 用于命令行选项、参数和子命令的解析器 . . . 785
17.1.1 ArgumentParser 对象 . . . 786
17.1.2 add_argument() 方法 . . . 792
17.1.3 parse_args() 方法 . . . 801
17.1.4 其它实用工具 . . . 804
17.1.5 异常 . . . 812
17.2 optparse --- 命令行选项的解析器 . . . 825
17.2.1 选择参数解析库 . . . 825
17.2.2 概述 . . . 826
17.2.3 背景 . . . 827
17.2.4 教程 . . . 829
17.2.5 参考指南 . . . 836
17.2.6 选项回调 . . . 845
17.2.7 扩展 optparse . . . 848
17.2.8 异常 . . . 850
17.3 getpass --- 可移植的密码输入 . . . 851
17.4 fileinput --- 迭代来自多个输入流的行 . . . 851
17.5 curses --- 字符单元显示的终端处理 . . . 854
17.5.1 函数 . . . 854
17.5.2 Window 对象 . . . 861
17.5.3 常量 . . . 866
17.6 curses.textpad --- 用于 curses 程序的文本输入控件 . . . 878
17.6.1 文本框对象 . . . 878
17.7 curses.ascii --- 用于 ASCII 字符的工具 . . . 879
17.8 curses.panel --- 针对 curses 的面板栈扩展 . . . 883
17.8.1 函数 . . . 883
17.8.2 Panel 对象 . . . 883

18 并发执行 **885**
18.1 threading --- 基于线程的并行 . . . 885
18.1.1 线程本地数据 . . . 888
18.1.2 线程对象 . . . 888
18.1.3 锁对象 . . . 890
18.1.4 递归锁对象 . . . 891
18.1.5 条件对象 . . . 892
18.1.6 信号量对象 . . . 894
18.1.7 事件对象 . . . 895
18.1.8 定时器对象 . . . 896
18.1.9 栅栏对象 . . . 896
18.1.10 在 with 语句中使用锁、条件和信号量 . . . 897
18.2 multiprocessing --- 基于进程的并行 . . . 898
18.2.1 概述 . . . 898
18.2.2 参考 . . . 904
18.2.3 编程指导 . . . 931

18.2.4 例子 . . . 934
18.3 multiprocessing.shared_memory --- 可跨进程直接访问的共享内存 . . . 939
18.4 concurrent 包 . . . 944
18.5 concurrent.futures --- 启动并行任务 . . . 944
18.5.1 Executor 对象 . . . 945
18.5.2 ThreadPoolExecutor . . . 945
18.5.3 ProcessPoolExecutor . . . 947
18.5.4 Future 对象 . . . 948
18.5.5 模块函数 . . . 950
18.5.6 Exception 类 . . . 950
18.6 subprocess --- 子进程管理 . . . 951
18.6.1 使用 subprocess 模块 . . . 951
18.6.2 安全考量 . . . 959
18.6.3 Popen 对象 . . . 959
18.6.4 Windows Popen 助手 . . . 961
18.6.5 较旧的高阶 API . . . 964
18.6.6 使用 subprocess 模块替换旧函数 . . . 965
18.6.7 旧式的 Shell 发起函数 . . . 968
18.6.8 备注 . . . 969
18.7 sched --- 事件调度器 . . . 969
18.7.1 调度器对象 . . . 970
18.8 queue --- 同步队列类 . . . 971
18.8.1 Queue 对象 . . . 972
18.8.2 SimpleQueue 对象 . . . 974
18.9 contextvars --- 上下文变量 . . . 974
18.9.1 上下文变量 . . . 974
18.9.2 手动上下文管理 . . . 975
18.9.3 asyncio 支持 . . . 977
18.10 _thread --- 低层级多线程 API . . . 978

19 网络和进程间通信 981
19.1 asyncio --- 异步 I/O . . . 981
19.1.1 运行器 . . . 982
19.1.2 协程与任务 . . . 984
19.1.3 流 . . . 1002
19.1.4 同步原语 . . . 1009
19.1.5 子进程集 . . . 1015
19.1.6 队列集 . . . 1019
19.1.7 异常 . . . 1022
19.1.8 事件循环 . . . 1023
19.1.9 Futures . . . 1045
19.1.10 传输和协议 . . . 1048
19.1.11 策略 . . . 1060
19.1.12 平台支持 . . . 1064
19.1.13 扩展 . . . 1065
19.1.14 高层级 API 索引 . . . 1066
19.1.15 低层级 API 索引 . . . 1068
19.1.16 用 asyncio 开发 . . . 1072
19.2 socket --- 低层级的网络接口 . . . 1075
19.2.1 套接字协议族 . . . 1076
19.2.2 模块内容 . . . 1079
19.2.3 套接字对象 . . . 1091
19.2.4 关于套接字超时的说明 . . . 1097
19.2.5 示例 . . . 1098
19.3 ssl --- 套接字对象的 TLS/SSL 包装器 . . . 1101
19.3.1 方法、常量和异常处理 . . . 1102
19.3.2 SSL 套接字 . . . 1113
19.3.3 SSL 上下文 . . . 1117

19.3.4 证书 . . . 1125
19.3.5 例子 . . . 1127
19.3.6 关于非阻塞套接字的说明 . . . 1129
19.3.7 内存 BIO 支持 . . . 1130
19.3.8 SSL 会话 . . . 1132
19.3.9 安全考量 . . . 1132
19.3.10 TLS 1.3 . . . 1133
19.4 select --- 等待 I/O 完成 . . . 1134
19.4.1 /dev/poll 轮询对象 . . . 1136
19.4.2 边缘触发和水平触发的轮询 (epoll) 对象 . . . 1137
19.4.3 Poll 对象 . . . 1137
19.4.4 Kqueue 对象 . . . 1138
19.4.5 Kevent 对象 . . . 1139
19.5 selectors --- 高层级 I/O 复用 . . . 1140
19.5.1 概述 . . . 1140
19.5.2 类 . . . 1141
19.5.3 例子 . . . 1143
19.6 signal --- 设置异步事件处理器 . . . 1144
19.6.1 一般规则 . . . 1144
19.6.2 模块内容 . . . 1144
19.6.3 例子 . . . 1151
19.6.4 对于 SIGPIPE 的说明 . . . 1151
19.6.5 有关信号处理器和异常的注释 . . . 1151
19.7 mmap --- 内存映射文件支持 . . . 1152
19.7.1 MADV_* 常量 . . . 1156
19.7.2 MAP_* 常量 . . . 1157

20 互联网数据处理 1159
20.1 email --- 电子邮件与 MIME 处理包 . . . 1159
20.1.1 email.message: 表示电子邮件消息 . . . 1160
20.1.2 email.parser: 解析电子邮件消息 . . . 1167
20.1.3 email.generator: 生成 MIME 文档 . . . 1170
20.1.4 email.policy: 策略对象 . . . 1173
20.1.5 email.errors: 异常和缺陷类 . . . 1179
20.1.6 email.headerregistry: 自定义标头对象 . . . 1180
20.1.7 email.contentmanager: 管理 MIME 内容 . . . 1185
20.1.8 email: 示例 . . . 1188
20.1.9 email.message.Message: 使用 compat32 API 来表示电子邮件消息 . . . 1194
20.1.10 email.mime: 从头创建电子邮件和 MIME 对象 . . . 1201
20.1.11 email.header: 国际化标头 . . . 1204
20.1.12 email.charset: 表示字符集 . . . 1206
20.1.13 email.encoders: 编码器 . . . 1208
20.1.14 email.utils: 杂项工具 . . . 1208
20.1.15 email.iterators: 迭代器 . . . 1211
20.2 json --- JSON 编码器和解码器 . . . 1212
20.2.1 基本使用 . . . 1214
20.2.2 编码器和解码器 . . . 1216
20.2.3 异常 . . . 1218
20.2.4 标准符合性和互操作性 . . . 1219
20.2.5 CLI . . . 1220
20.3 mailbox --- 操纵多种格式的邮箱 . . . 1221
20.3.1 Mailbox 对象 . . . 1221
20.3.2 Message 对象 . . . 1230
20.3.3 异常 . . . 1237
20.3.4 例子 . . . 1238
20.4 mimetypes --- 将文件名映射到 MIME 类型 . . . 1239
20.4.1 MimeTypes 对象 . . . 1240
20.5 base64 --- Base16, Base32, Base64, Base85 数据编码 . . . 1241

20.5.1 安全考量 . . . 1244
20.6 binascii --- 在二进制数据和 ASCII 之间进行转换 . . . 1244
20.7 quopri --- 编码与解码 MIME 转码的可打印数据 . . . 1246

21 结构化标记处理工具 1249
21.1 html --- 超文本标记语言支持 . . . 1249
21.2 html.parser --- 简单的 HTML 和 XHTML 解析器 . . . 1249
21.2.1 HTML 解析器的示例程序 . . . 1250
21.2.2 HTMLParser 方法 . . . 1250
21.2.3 例子 . . . 1252
21.3 html.entities --- HTML 一般实体的定义 . . . 1254
21.4 XML 处理模块 . . . 1254
21.4.1 XML 漏洞 . . . 1255
21.4.2 defusedxml 包 . . . 1256
21.5 xml.etree.ElementTree --- ElementTree XML API . . . 1256
21.5.1 教程 . . . 1256
21.5.2 XPath 支持 . . . 1261
21.5.3 参考 . . . 1262
21.5.4 XInclude 支持 . . . 1265
21.5.5 参考 . . . 1266
21.6 xml.dom --- 文档对象模型 API . . . 1273
21.6.1 模块内容 . . . 1274
21.6.2 DOM 中的对象 . . . 1275
21.6.3 一致性 . . . 1282
21.7 xml.dom.minidom --- 最小化的 DOM 实现 . . . 1283
21.7.1 DOM 对象 . . . 1284
21.7.2 DOM 示例 . . . 1285
21.7.3 minidom 和 DOM 标准 . . . 1286
21.8 xml.dom.pulldom --- 对构建部分 DOM 树的支持 . . . 1286
21.8.1 DOMEventStream 对象 . . . 1288
21.9 xml.sax --- SAX2 解析器支持 . . . 1288
21.9.1 SAXException 对象 . . . 1290
21.10 xml.sax.handler --- SAX 处理器的基类 . . . 1290
21.10.1 ContentHandler 对象 . . . 1292
21.10.2 DTDHandler 对象 . . . 1294
21.10.3 EntityResolver 对象 . . . 1294
21.10.4 ErrorHandler 对象 . . . 1294
21.10.5 LexicalHandler 对象 . . . 1294
21.11 xml.sax.saxutils --- SAX 工具集 . . . 1295
21.12 xml.sax.xmlreader --- 用于 XML 解析器的接口 . . . 1296
21.12.1 XMLReader 对象 . . . 1297
21.12.2 IncrementalParser 对象 . . . 1298
21.12.3 Locator 对象 . . . 1298
21.12.4 InputSource 对象 . . . 1298
21.12.5 Attributes 接口 . . . 1299
21.12.6 AttributesNS 接口 . . . 1299
21.13 xml.parsers.expat --- 使用 Expat 进行快速 XML 解析 . . . 1299
21.13.1 XMLParser 对象 . . . 1301
21.13.2 ExpatError 异常 . . . 1304
21.13.3 示例 . . . 1305
21.13.4 内容模型描述 . . . 1305
21.13.5 Expat 错误常量 . . . 1306

22 互联网协议和支持 1309
22.1 webbrowser --- 方便的 Web 浏览器控制工具 . . . 1309
22.1.1 浏览器控制器对象 . . . 1311
22.2 wsgiref --- WSGI 工具和参考实现 . . . 1312
22.2.1 wsgiref.util -- WSGI 环境工具 . . . 1312

22.2.2 wsgiref.headers -- WSGI 响应标头工具 . . . 1314
22.2.3 wsgiref.simple_server -- 一个简单的 WSGI HTTP 服务器 . . . 1315
22.2.4 wsgiref.validate --- WSGI 一致性检查器 . . . 1316
22.2.5 wsgiref.handlers -- 服务器/网关基类 . . . 1316
22.2.6 wsgiref.types -- 用于静态类型检查的 WSGI 类型 . . . 1320
22.2.7 例子 . . . 1320
22.3 urllib --- URL 处理模块 . . . 1321
22.4 urllib.request --- 用于打开 URL 的可扩展库 . . . 1321
22.4.1 Request 对象 . . . 1326
22.4.2 OpenerDirector 对象 . . . 1327
22.4.3 BaseHandler 对象 . . . 1328
22.4.4 HTTPRedirectHandler 对象 . . . 1330
22.4.5 HTTPCookieProcessor 对象 . . . 1330
22.4.6 ProxyHandler 对象 . . . 1330
22.4.7 HTTPPasswordMgr 对象 . . . 1331
22.4.8 HTTPPasswordMgrWithPriorAuth 对象 . . . 1331
22.4.9 AbstractBasicAuthHandler 对象 . . . 1331
22.4.10 HTTPBasicAuthHandler 对象 . . . 1331
22.4.11 ProxyBasicAuthHandler 对象 . . . 1331
22.4.12 AbstractDigestAuthHandler 对象 . . . 1331
22.4.13 HTTPDigestAuthHandler 对象 . . . 1332
22.4.14 ProxyDigestAuthHandler 对象 . . . 1332
22.4.15 HTTPHandler 对象 . . . 1332
22.4.16 HTTPSHandler 对象 . . . 1332
22.4.17 FileHandler 对象 . . . 1332
22.4.18 DataHandler 对象 . . . 1332
22.4.19 FTPHandler 对象 . . . 1332
22.4.20 CacheFTPHandler 对象 . . . 1332
22.4.21 UnknownHandler 对象 . . . 1332
22.4.22 HTTPErrorProcessor 对象 . . . 1333
22.4.23 例子 . . . 1333
22.4.24 已停用的接口 . . . 1335
22.4.25 urllib.request 的限制 . . . 1337
22.5 urllib.response --- urllib 使用的 Response 类 . . . 1338
22.6 urllib.parse --- 将 URL 解析为组件 . . . 1338
22.6.1 URL 解析 . . . 1339
22.6.2 URL 解析安全 . . . 1343
22.6.3 解析 ASCII 编码字节 . . . 1343
22.6.4 结构化解析结果 . . . 1344
22.6.5 URL 转码 . . . 1345
22.7 urllib.error --- 由 urllib.request 引发的异常类 . . . 1347
22.8 urllib.robotparser --- 用于 robots.txt 的解析器 . . . 1347
22.9 http --- HTTP 模块 . . . 1348
22.9.1 HTTP 状态码 . . . 1349
22.9.2 HTTP 状态类别 . . . 1350
22.9.3 HTTP 方法 . . . 1351
22.10 http.client --- HTTP 协议客户端 . . . 1352
22.10.1 HTTPConnection 对象 . . . 1354
22.10.2 HTTPResponse 对象 . . . 1356
22.10.3 例子 . . . 1357
22.10.4 HTTPMessage 对象 . . . 1359
22.11 ftplib --- FTP 协议客户端 . . . 1359
22.11.1 参考 . . . 1359
22.12 poplib --- POP3 协议客户端 . . . 1365
22.12.1 POP3 对象 . . . 1366
22.12.2 POP3 示例 . . . 1367
22.13 imaplib --- IMAP4 协议客户端 . . . 1368
22.13.1 IMAP4 对象 . . . 1369

22.13.2 IMAP4 示例 . . . 1373
22.14 smtplib --- SMTP 协议客户端 . . . 1374
22.14.1 SMTP 对象 . . . 1376
22.14.2 SMTP 示例 . . . 1379
22.15 uuid --- 根据 **RFC 4122** 定义的 UUID 对象 . . . 1380
22.15.1 命令行用法 . . . 1383
22.15.2 示例 . . . 1383
22.15.3 命令行示例 . . . 1384
22.16 socketserver --- 用于网络服务器的框架 . . . 1384
22.16.1 服务器创建的说明 . . . 1385
22.16.2 Server 对象 . . . 1386
22.16.3 请求处理器对象 . . . 1388
22.16.4 例子 . . . 1389
22.17 http.server --- HTTP 服务器 . . . 1392
22.17.1 命令行接口 . . . 1397
22.17.2 安全考量 . . . 1398
22.18 http.cookies --- HTTP 状态管理 . . . 1398
22.18.1 Cookie 对象 . . . 1399
22.18.2 Morsel 对象 . . . 1399
22.18.3 示例 . . . 1400
22.19 http.cookiejar --- HTTP 客户端的 Cookie 处理 . . . 1401
22.19.1 CookieJar 和 FileCookieJar 对象 . . . 1403
22.19.2 FileCookieJar 的子类及其与 Web 浏览器的协同 . . . 1405
22.19.3 CookiePolicy 对象 . . . 1405
22.19.4 DefaultCookiePolicy 对象 . . . 1406
22.19.5 Cookie 对象 . . . 1408
22.19.6 例子 . . . 1409
22.20 xmlrpc --- XMLRPC 服务端与客户端模块 . . . 1409
22.21 xmlrpc.client --- XML-RPC 客户端访问 . . . 1409
22.21.1 ServerProxy 对象 . . . 1411
22.21.2 DateTime 对象 . . . 1412
22.21.3 Binary 对象 . . . 1413
22.21.4 Fault 对象 . . . 1414
22.21.5 ProtocolError 对象 . . . 1414
22.21.6 MultiCall 对象 . . . 1415
22.21.7 便捷函数 . . . 1416
22.21.8 客户端用法的示例 . . . 1416
22.21.9 客户端与服务器用法的示例 . . . 1417
22.22 xmlrpc.server --- 基本 XML-RPC 服务器 . . . 1417
22.22.1 SimpleXMLRPCServer 对象 . . . 1418
22.22.2 CGIXMLRPCRequestHandler . . . 1421
22.22.3 文档 XMLRPC 服务器 . . . 1421
22.22.4 DocXMLRPCServer 对象 . . . 1422
22.22.5 DocCGIXMLRPCRequestHandler . . . 1422
22.23 ipaddress --- IPv4/IPv6 操作库 . . . 1422
22.23.1 方便的工厂函数 . . . 1422
22.23.2 IP 地址 . . . 1423
22.23.3 IP 网络的定义 . . . 1427
22.23.4 接口对象 . . . 1433
22.23.5 其他模块级别函数 . . . 1434
22.23.6 自定义异常 . . . 1435

23 多媒体服务 **1437**
23.1 wave --- 读写 WAV 文件 . . . 1437
23.1.1 Wave_read 对象 . . . 1438
23.1.2 Wave_write 对象 . . . 1439
23.2 colorsys --- 颜色系统间的转换 . . . 1439

24 国际化 **1441**
24.1 gettext --- 多语种国际化服务 . . . 1441
24.1.1 GNU gettext API . . . 1441
24.1.2 基于类的 API . . . 1442
24.1.3 国际化 (I18N) 你的程序和模块 . . . 1445
24.1.4 致谢 . . . 1448
24.2 locale --- 国际化服务 . . . 1448
24.2.1 背景、细节、提示、技巧和注意事项 . . . 1454
24.2.2 针对扩展程序编写人员和嵌入 Python 运行的程序 . . . 1454
24.2.3 访问消息目录 . . . 1454

25 程序框架 **1457**
25.1 turtle --- 海龟绘图 . . . 1457
25.1.1 概述 . . . 1457
25.1.2 入门 . . . 1457
25.1.3 教程 . . . 1458
25.1.4 如何... . . . 1460
25.1.5 海龟绘图参考 . . . 1461
25.1.6 RawTurtle/Turtle 方法和对应函数 . . . 1463
25.1.7 TurtleScreen/Screen 方法及对应函数 . . . 1479
25.1.8 公共类 . . . 1486
25.1.9 说明 . . . 1487
25.1.10 帮助与配置 . . . 1487
25.1.11 turtledemo --- 演示脚本集 . . . 1490
25.1.12 Python 2.6 之后的变化 . . . 1491
25.1.13 Python 3.0 之后的变化 . . . 1491
25.2 cmd --- 对面向行的命令解释器的支持 . . . 1491
25.2.1 Cmd 对象 . . . 1491
25.2.2 Cmd 例子 . . . 1493
25.3 shlex --- 简单词法分析 . . . 1496
25.3.1 shlex 对象 . . . 1497
25.3.2 解析规则 . . . 1499
25.3.3 改进的 shell 兼容性 . . . 1500

26 Tk 图形用户界面 (GUI) **1501**
26.1 tkinter --- Tcl/Tk 的 Python 接口 . . . 1501
26.1.1 架构 . . . 1502
26.1.2 Tkinter 模块 . . . 1503
26.1.3 Tkinter 拾遗 . . . 1504
26.1.4 线程模型 . . . 1507
26.1.5 快速参考 . . . 1507
26.1.6 文件处理程序 . . . 1512
26.2 tkinter.colorchooser --- 颜色选择对话框 . . . 1513
26.3 tkinter.font --- Tkinter 字体包装器 . . . 1513
26.4 Tkinter 对话框 . . . 1514
26.4.1 tkinter.simpledialog --- 标准 Tkinter 输入对话框 . . . 1514
26.4.2 tkinter.filedialog --- 文件选择对话框. . . . 1515
26.4.3 tkinter.commondialog --- 对话窗口模板 . . . 1517
26.5 tkinter.messagebox --- Tkinter 消息提示 . . . 1517
26.6 tkinter.scrolledtext --- 流动文本控件 . . . 1519
26.7 tkinter.dnd --- 拖放操作支持 . . . 1520
26.8 tkinter.ttk --- Tk 带主题的控件 . . . 1520
26.8.1 ttk 的用法 . . . 1521
26.8.2 ttk 控件 . . . 1521
26.8.3 控件 . . . 1521
26.8.4 Combobox . . . 1523
26.8.5 Spinbox . . . 1524
26.8.6 Notebook . . . 1525

26.8.7 Progressbar . . . 1527
26.8.8 Separator . . . 1528
26.8.9 Sizegrip . . . 1528
26.8.10 Treeview . . . 1528
26.8.11 Ttk 样式 . . . 1533
26.9 IDLE --- Python 编辑器和 shell . . . 1537
26.9.1 目录 . . . 1537
26.9.2 编辑和导航 . . . 1542
26.9.3 启动和代码执行 . . . 1544
26.9.4 帮助和首选项 Help and Preferences . . . 1547
26.9.5 idlelib --- IDLE 应用程序的实现 . . . 1548

27 开发工具 1549
27.1 typing ——对类型提示的支持 . . . 1549
27.1.1 有关 Python 类型系统的规范说明 . . . 1550
27.1.2 类型别名 . . . 1550
27.1.3 NewType . . . 1551
27.1.4 标注可调用对象 . . . 1552
27.1.5 泛型（Generic） . . . 1553
27.1.6 标注元组 . . . 1553
27.1.7 类对象的类型 . . . 1554
27.1.8 标注生成器和协程 . . . 1555
27.1.9 用户定义的泛型类型 . . . 1556
27.1.10 Any 类型 . . . 1558
27.1.11 名义子类型 vs 结构子类型 . . . 1560
27.1.12 模块内容 . . . 1560
27.1.13 主要特性的弃用时间线 . . . 1597
27.2 pydoc --- 文档生成器和在线帮助系统 . . . 1597
27.3 Python 开发模式 . . . 1598
27.3.1 Python 开发模式的效果 . . . 1599
27.3.2 ResourceWarning 示例 . . . 1600
27.3.3 文件描述符错误示例 . . . 1600
27.4 doctest --- 测试交互式的 Python 示例 . . . 1601
27.4.1 简单用法：检查 Docstrings 中的示例 . . . 1603
27.4.2 简单的用法：检查文本文件中的例子 . . . 1603
27.4.3 命令行用法 . . . 1604
27.4.4 它是如何工作的 . . . 1605
27.4.5 基本 API . . . 1611
27.4.6 Unittest API . . . 1612
27.4.7 高级 API . . . 1614
27.4.8 调试 . . . 1619
27.4.9 肥皂盒 . . . 1621
27.5 unittest --- 单元测试框架 . . . 1622
27.5.1 基本实例 . . . 1623
27.5.2 命令行接口 . . . 1624
27.5.3 探索性测试 . . . 1625
27.5.4 组织你的测试代码 . . . 1626
27.5.5 复用已有的测试代码 . . . 1628
27.5.6 跳过测试与预计的失败 . . . 1628
27.5.7 使用子测试区分测试迭代 . . . 1630
27.5.8 类与函数 . . . 1631
27.5.9 类与模块设定 . . . 1648
27.5.10 信号处理 . . . 1650
27.6 unittest.mock --- 模拟对象库 . . . 1650
27.6.1 快速上手 . . . 1651
27.6.2 Mock 类 . . . 1653
27.6.3 patch 装饰器 . . . 1669
27.6.4 MagicMock 与魔术方法支持 . . . 1677

27.6.5 辅助对象 . . . 1680
27.6.6 side_effect, return_value 和 *wraps* 的优先顺序 . . . 1687
27.7 unittest.mock --- 新手入门 . . . 1689
27.7.1 使用 mock . . . 1689
27.7.2 补丁装饰器 . . . 1694
27.7.3 更多示例 . . . 1696
27.8 test --- Python 回归测试包 . . . 1707
27.8.1 为 test 包编写单元测试 . . . 1708
27.8.2 使用命令行界面运行测试 . . . 1709
27.9 test.support --- 针对 Python 测试套件的工具 . . . 1710
27.10 test.support.socket_helper --- 用于套接字测试的工具 . . . 1718
27.11 test.support.script_helper --- 用于 Python 执行测试工具 . . . 1719
27.12 test.support.bytecode_helper --- 用于测试正确字节码生成的支持工具 . . . 1720
27.13 test.support.threading_helper --- 用于线程测试的工具 . . . 1720
27.14 test.support.os_helper --- 用于操作系统测试的工具 . . . 1721
27.15 test.support.import_helper --- 用于导入测试的工具 . . . 1723
27.16 test.support.warnings_helper --- 用于警告测试的工具 . . . 1724

28 调试和分析 **1727**
28.1 审计事件表 . . . 1727
28.2 bdb --- 调试器框架 . . . 1731
28.3 faulthandler --- 转储 Python 回溯信息 . . . 1736
28.3.1 转储跟踪信息 . . . 1736
28.3.2 故障处理程序的状态 . . . 1736
28.3.3 一定时间后转储跟踪数据。 . . . 1737
28.3.4 转储用户信号的跟踪信息。 . . . 1737
28.3.5 文件描述符相关话题 . . . 1737
28.3.6 示例 . . . 1737
28.4 pdb --- Python 的调试器 . . . 1738
28.4.1 调试器命令 . . . 1740
28.5 Python 性能分析器 . . . 1746
28.5.1 性能分析器简介 . . . 1746
28.5.2 实时用户手册 . . . 1746
28.5.3 profile 和 cProfile 模块参考 . . . 1749
28.5.4 Stats 类 . . . 1750
28.5.5 什么是确定性性能分析? . . . 1752
28.5.6 局限性 . . . 1752
28.5.7 准确估量 . . . 1752
28.5.8 使用自定义计时器 . . . 1753
28.6 timeit --- 测量小代码片段的执行时间 . . . 1754
28.6.1 基本示例 . . . 1754
28.6.2 Python 接口 . . . 1754
28.6.3 命令行接口 . . . 1756
28.6.4 例子 . . . 1757
28.7 trace --- 跟踪或记录 Python 语句的执行 . . . 1758
28.7.1 命令行用法 . . . 1758
28.7.2 编程接口 . . . 1759
28.8 tracemalloc --- 跟踪内存分配 . . . 1760
28.8.1 例子 . . . 1761
28.8.2 API . . . 1765

29 软件打包和分发 **1771**
29.1 ensurepip --- 初始设置 pip 安装器 . . . 1771
29.1.1 命令行界面 . . . 1772
29.1.2 模块 API . . . 1772
29.2 venv --- 虚拟环境的创建 . . . 1773
29.2.1 创建虚拟环境 . . . 1773
29.2.2 虚拟环境是如何实现的 . . . 1775

29.2.3 API . . . 1776
29.2.4 一个扩展 EnvBuilder 的例子 . . . 1778
29.3 zipapp --- 管理可执行的 Python zip 归档文件 . . . 1782
29.3.1 简单示例 . . . 1782
29.3.2 命令行接口 . . . 1782
29.3.3 Python API . . . 1783
29.3.4 例子 . . . 1784
29.3.5 指定解释器程序 . . . 1784
29.3.6 用 zipapp 创建独立运行的应用程序 . . . 1784
29.3.7 Python 打包应用程序的格式 . . . 1785

30 Python 运行时服务 1787
30.1 sys --- 系统相关的形参和函数 . . . 1787
30.2 sys.monitoring --- 执行事件监测 . . . 1810
30.2.1 工具标识符 . . . 1811
30.2.2 事件 . . . 1811
30.2.3 开启和关闭事件 . . . 1813
30.2.4 注册回调函数 . . . 1814
30.3 sysconfig --- 提供对 Python 配置信息的访问 . . . 1815
30.3.1 配置变量 . . . 1815
30.3.2 安装路径 . . . 1815
30.3.3 用户方案 . . . 1816
30.3.4 主方案 . . . 1817
30.3.5 前缀方案 . . . 1817
30.3.6 安装路径函数 . . . 1818
30.3.7 其他功能 . . . 1819
30.3.8 将 sysconfig 作为脚本使用 . . . 1820
30.4 builtins --- 内置对象 . . . 1820
30.5 __main__ --- 最高层级代码环境 . . . 1821
30.5.1 __name__ == '__main__' . . . 1821
30.5.2 Python 包中的 __main__.py . . . 1823
30.5.3 import __main__ . . . 1824
30.6 warnings --- 警告信息控制 . . . 1825
30.6.1 警告类别 . . . 1826
30.6.2 警告过滤器 . . . 1826
30.6.3 暂时禁止警告 . . . 1828
30.6.4 测试警告 . . . 1829
30.6.5 为新版本的依赖关系更新代码 . . . 1829
30.6.6 可用的函数 . . . 1830
30.6.7 可用的上下文管理器 . . . 1831
30.7 dataclasses --- 数据类 . . . 1832
30.7.1 模块内容 . . . 1832
30.7.2 初始化后处理 . . . 1838
30.7.3 类变量 . . . 1839
30.7.4 仅初始化变量 . . . 1839
30.7.5 冻结的实例 . . . 1839
30.7.6 继承 . . . 1840
30.7.7 __init__() 中仅限关键字形参的重新排序 . . . 1840
30.7.8 默认工厂函数 . . . 1840
30.7.9 可变的默认值 . . . 1841
30.7.10 描述器类型的字段 . . . 1841
30.8 contextlib --- 为 with 语句上下文提供的工具 . . . 1842
30.8.1 工具 . . . 1842
30.8.2 例子和配方 . . . 1850
30.8.3 单独使用，可重用并可重进入的上下文管理器 . . . 1854
30.9 abc --- 抽象基类 . . . 1855
30.10 atexit --- 退出处理器 . . . 1860
30.10.1 atexit 示例 . . . 1861

30.11 traceback --- 打印或读取栈回溯信息 . . . 1861
30.11.1 模块级函数 . . . 1862
30.11.2 TracebackException 对象 . . . 1864
30.11.3 StackSummary 对象 . . . 1866
30.11.4 FrameSummary 对象 . . . 1866
30.11.5 使用模块级函数的例子 . . . 1867
30.11.6 使用 TracebackException 的示例 . . . 1869
30.12 __future__ --- Future 语句定义 . . . 1871
30.12.1 模块内容 . . . 1871
30.13 gc --- 垃圾回收器接口 . . . 1872
30.14 inspect --- 检查当前对象 . . . 1875
30.14.1 类型和成员 . . . 1876
30.14.2 获取源代码 . . . 1880
30.14.3 使用 Signature 对象对可调用对象进行内省 . . . 1881
30.14.4 类与函数 . . . 1885
30.14.5 解释器栈 . . . 1888
30.14.6 静态地获取属性 . . . 1890
30.14.7 生成器、协程和异步生成器的当前状态 . . . 1890
30.14.8 代码对象位标志 . . . 1891
30.14.9 缓冲区旗标 . . . 1892
30.14.10命令行界面 . . . 1893
30.15 site --- 站点专属的配置钩子 . . . 1893
30.15.1 sitecustomize . . . 1894
30.15.2 usercustomize . . . 1894
30.15.3 Readline 配置 . . . 1895
30.15.4 模块内容 . . . 1895
30.15.5 命令行界面 . . . 1896

31 自定义 Python 解释器 1897
31.1 code --- 解释器基类 . . . 1897
31.1.1 交互解释器对象 . . . 1898
31.1.2 交互式控制台对象 . . . 1898
31.2 codeop --- 编译 Python 代码 . . . 1899

32 导入模块 1901
32.1 zipimport --- 从 Zip 归档导入模块 . . . 1901
32.1.1 zipimporter 对象 . . . 1902
32.1.2 例子 . . . 1903
32.2 pkgutil --- 包扩展工具 . . . 1903
32.3 modulefinder --- 查找脚本使用的模块 . . . 1906
32.3.1 ModuleFinder 的示例用法 . . . 1906
32.4 runpy --- 查找并执行 Python 模块 . . . 1907
32.5 importlib --- import 的实现 . . . 1909
32.5.1 概述 . . . 1909
32.5.2 函数 . . . 1910
32.5.3 importlib.abc ——关于导入的抽象基类 . . . 1912
32.5.4 importlib.machinery ——导入器和路径钩子函数。 . . . 1918
32.5.5 importlib.util ——导入器的工具程序代码 . . . 1923
32.5.6 例子 . . . 1925
32.6 importlib.resources -- 包资源的读取、打开和访问 . . . 1928
32.6.1 函数式 API . . . 1929
32.7 importlib.resources.abc -- 资源的抽象基类 . . . 1931
32.8 importlib.metadata -- 访问软件包元数据 . . . 1932
32.8.1 概述 . . . 1933
32.8.2 函数式 API . . . 1933
32.8.3 分发包对象 . . . 1937
32.8.4 分发包的发现 . . . 1937
32.8.5 扩展搜索算法 . . . 1937

32.9 sys.path 模块搜索路径的初始化 . . . 1939
32.9.1 从虚拟环境 . . . 1940
32.9.2 _pth 文件 . . . 1940
32.9.3 嵌入式 Python . . . 1940

33 Python 语言服务 1941
33.1 ast --- 抽象语法树 . . . 1941
33.1.1 抽象文法 . . . 1941
33.1.2 节点类 . . . 1944
33.1.3 ast 中的辅助函数 . . . 1971
33.1.4 编译器旗标 . . . 1975
33.1.5 命令行用法 . . . 1975
33.2 symtable --- 访问编译器的符号表 . . . 1976
33.2.1 符号表的生成 . . . 1976
33.2.2 符号表的查看 . . . 1976
33.2.3 命令行用法 . . . 1979
33.3 token --- 用于 Python 解析树的常量 . . . 1979
33.4 keyword --- 检验 Python 关键字 . . . 1984
33.5 tokenize --- Python 源代码的分词器 . . . 1985
33.5.1 对输入进行解析标记 . . . 1985
33.5.2 命令行用法 . . . 1986
33.5.3 例子 . . . 1987
33.6 tabnanny --- 检测有歧义的缩进 . . . 1988
33.7 pyclbr --- Python 模块浏览器支持 . . . 1989
33.7.1 Function 对象 . . . 1990
33.7.2 Class 对象 . . . 1990
33.8 py_compile --- 编译 Python 源文件 . . . 1991
33.8.1 命令行接口 . . . 1992
33.9 compileall --- 字节编译 Python 库 . . . 1992
33.9.1 使用命令行 . . . 1993
33.9.2 公有函数 . . . 1994
33.10 dis --- Python 字节码反汇编器 . . . 1996
33.10.1 命令行接口 . . . 1997
33.10.2 字节码分析 . . . 1997
33.10.3 分析函数 . . . 1998
33.10.4 Python 字节码说明 . . . 2000
33.10.5 操作码集合 . . . 2016
33.11 pickletools --- pickle 开发者工具 . . . 2017
33.11.1 命令行语法 . . . 2017
33.11.2 编程接口 . . . 2018

34 Windows 系统相关模块 2019
34.1 msvcrt --- 来自 MS VC++ 运行时的有用例程 . . . 2019
34.1.1 文件操作 . . . 2019
34.1.2 控制台 I/O . . . 2020
34.1.3 其他函数 . . . 2020
34.2 winreg --- Windows 注册表访问 . . . 2021
34.2.1 函数 . . . 2022
34.2.2 常量 . . . 2027
34.2.3 注册表句柄对象 . . . 2029
34.3 winsound --- 针对 Windows 的声音播放接口 . . . 2030

35 Unix 专有服务 2033
35.1 posix --- 最常见的 POSIX 系统调用 . . . 2033
35.1.1 大文件支持 . . . 2033
35.1.2 重要的模块内容 . . . 2034
35.2 pwd --- 密码数据库 . . . 2034
35.3 grp --- 组数据库 . . . 2035
35.4 termios --- POSIX 风格的 tty 控制 . . . 2035

35.4.1 示例 . . . 2037
35.5 tty --- 终端控制函数 . . . 2037
35.6 pty --- 伪终端工具 . . . 2038
35.6.1 示例 . . . 2039
35.7 fcntl --- fcntl 和 ioctl 系统调用 . . . 2039
35.8 resource --- 资源使用信息 . . . 2042
35.8.1 资源限制 . . . 2042
35.8.2 资源用量 . . . 2045
35.9 syslog --- Unix syslog 库例程 . . . 2046
35.9.1 例子 . . . 2048

36 模块命令行界面（CLI） 2049

37 被取代的模块 2051
37.1 getopt --- C 风格的命令行选项解析器 . . . 2051

38 Removed Modules 2055
38.1 aifc --- Read and write AIFF and AIFC files . . . 2055
38.2 asynchat --- Asynchronous socket command/response handler . . . 2055
38.3 asyncore --- Asynchronous socket handler . . . 2055
38.4 audioop --- Manipulate raw audio data . . . 2056
38.5 cgi --- Common Gateway Interface support . . . 2056
38.6 cgitb --- Traceback manager for CGI scripts . . . 2056
38.7 chunk --- Read IFF chunked data . . . 2056
38.8 crypt --- Function to check Unix passwords . . . 2056
38.9 distutils --- Building and installing Python modules . . . 2056
38.10 imghdr --- Determine the type of an image . . . 2057
38.11 imp --- Access the import internals . . . 2057
38.12 mailcap --- Mailcap file handling . . . 2057
38.13 msilib --- Read and write Microsoft Installer files . . . 2057
38.14 nis --- Interface to Sun’s NIS (Yellow Pages) . . . 2057
38.15 nntplib --- NNTP protocol client . . . 2057
38.16 ossaudiodev --- Access to OSS-compatible audio devices . . . 2058
38.17 pipes --- Interface to shell pipelines . . . 2058
38.18 smtpd --- SMTP Server . . . 2058
38.19 sndhdr --- Determine type of sound file . . . 2058
38.20 spwd --- The shadow password database . . . 2058
38.21 sunau --- Read and write Sun AU files . . . 2058
38.22 telnetlib --- Telnet client . . . 2059
38.23 uu --- Encode and decode uuencode files . . . 2059
38.24 xdrlib --- Encode and decode XDR data . . . 2059

39 安全考量 2061

A 术语对照表 2063

B 关于本文档 2079
B.1 Python 文档的贡献者 . . . 2079

C 历史和许可证 2081
C.1 该软件的历史 . . . 2081
C.2 获取或以其他方式使用 Python 的条款和条件 . . . 2082
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 2082
C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议 . . . 2083
C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议 . . . 2083
C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议 . . . 2084
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . . . 2085
C.3 收录软件的许可与鸣谢 . . . 2085
C.3.1 Mersenne Twister . . . 2085

C.3.2 套接字 . . . 2086
C.3.3 异步套接字服务 . . . 2086
C.3.4 Cookie 管理 . . . 2087
C.3.5 执行追踪 . . . 2087
C.3.6 UUencode 与 UUdecode 函数 . . . 2088
C.3.7 XML 远程过程调用 . . . 2088
C.3.8 test_epoll . . . 2089
C.3.9 Select kqueue . . . 2089
C.3.10 SipHash24 . . . 2090
C.3.11 strtod 和 dtoa . . . 2090
C.3.12 OpenSSL . . . 2090
C.3.13 expat . . . 2093
C.3.14 libffi . . . 2094
C.3.15 zlib . . . 2094
C.3.16 cfuhash . . . 2095
C.3.17 libmpdec . . . 2095
C.3.18 W3C C14N 测试套件 . . . 2096
C.3.19 mimalloc . . . 2096
C.3.20 asyncio . . . 2097
C.3.21 Global Unbounded Sequences (GUS) . . . 2097

D 版权所有 **2099**

Bibliography **2101**

Python 模块索引 **2103**

索引 **2107**

reference-index 描述了 Python 语言的具体语法和语义，这份库参考则介绍了与 Python 一同发行的标准库。它还描述了通常包含在 Python 发行版中的一些可选组件。

Python 标准库非常庞大，所提供的组件涉及范围十分广泛，正如以下内容目录所显示的。这个库包含了多个内置模块 (以 C 编写)，Python 程序员必须依靠它们来实现系统级功能，例如文件 I/O，此外还有大量以 Python 编写的模块，提供了日常编程中许多问题的标准解决方案。其中有些模块经过专门设计，通过将特定平台功能抽象化为平台中立的 API 来鼓励和加强 Python 程序的可移植性。

Windows 版本的 Python 安装程序通常包含整个标准库，往往还包含许多额外组件。对于类 Unix 操作系统，Python 通常会分成一系列的软件包，因此可能需要使用操作系统所提供的包管理工具来获取部分或全部可选组件。

在标准库以外，还存在成千上万并且不断增加的其他组件集（从单独的程序和模块到软件包以及完整的应用程序开发框架），这些组件集可以从 Python 包索引 获取。

CHAPTER 1

概述

”Python 库” 中包含了几种不同的组件。

它包含通常被视为语言“核心”中的一部分的数据类型，例如数字和列表。对于这些类型，Python 语言核心定义了文字的形式，并对它们的语义设置了一些约束，但没有完全定义语义。（另一方面，语言核心确实定义了语法属性，如操作符的拼写和优先级。）

这个库也包含了内置函数和异常 --- 不需要 `import` 语句就可以在所有 Python 代码中使用的对象。有一些是由语言核心定义的，但是许多对于核心语义不是必需的，并且仅在这里描述。

不过这个库主要是由一系列的模块组成。这些模块集可以不同方式分类。有些模块是用 C 编写并内置于 Python 解释器中；另一些模块则是用 Python 编写并以源码形式导入。有些模块提供专用于 Python 的接口，例如打印栈追踪信息；有些模块提供专用于特定操作系统的接口，例如操作特定的硬件；另一些模块则提供针对特定应用领域的接口，例如万维网。有些模块在所有更新和移植版本的 Python 中可用；另一些模块仅在底层系统支持或要求时可用；还有些模块则仅当编译和安装 Python 时选择了特定配置选项时才可用。

本手册以” 从内到外” 的顺序组织：首先描述内置函数、数据类型和异常，最后是根据相关性进行分组的各种模块。

这意味着如果你从头开始阅读本手册，并在感到厌烦时跳到下一章，你仍能对 Python 库的可用模块和所支持的应用领域有个大致了解。当然，你并非 *必须* 如同读小说一样从头读到尾 --- 你也可以先浏览内容目录 (在手册开头)，或在索引 (在手册末尾) 中查找某个特定函数、模块或条目。最后，如果你喜欢随意学习某个主题，你可以选择一个随机页码 (参见 *random* 模块) 并读上一两小节。无论你想以怎样的顺序阅读本手册，还是建议先从 内置函数 这一章开始，因为本手册的其余内容都需要你熟悉其中的基本概念。

让我们开始吧！

1.1 可用性注释

- 如果出现“适用：Unix”注释，意味着相应函数通常存在于 Unix 系统中。但这并不保证其存在于某个特定的操作系统中。
- 如果没有单独注明，所有声称” 可用性：Unix” 的函数都支持 macOS、iOS 和 Android ，所有这些构建都基于 Unix 内核。
- 如果一条可用性注释同时包含最低 Kernel 版本和最低 libc 版本，则两个条件都必须满足。例如当某个特性带有注释 *可用性: Linux >= 3.17 且 glibc >= 2.27* 则表示同时要求 Linux 3.17 以上版本和 glibc 2.27 以上版本。

1.1.1 WebAssembly 平台

WebAssembly 平台 `wasm32-emscripten` (Emscripten) 和 `wasm32-wasi` (WASI) 分别提供了 POSIX API 的一个子集。WebAssembly 运行时和浏览器都处于沙盒模式中并具有对主机和外部资源的受限访问权。任何使用了进程、线程、网络、信号或其他形式的进程间通信 (IPC) 的 Python 标准库模块都或者不可用，或者其作用方式与在其他类 Unix 系统上不同。文件 I/O, 文件系统和 Unix 权限相关的函数也同样会受限。Emscripten 不允许阻塞式 I/O。其他阻塞式操作如*sleep()* 则会阻塞浏览器的事件循环。

Python 在 WebAssembly 平台上的特性与行为依赖于 Emscripten-SDK 或 WASI-SDK 的版本, WASM 运行时 (浏览器, NodeJS, wasmtime) 以及 Python 编译时旗标。WebAssembly, Emscripten 和 WASI 都是尚在不断演化中的标准；某些特性例如网络可能会在未来被支持。

对于在浏览器上运行的 Python，用户可以考虑 Pyodide 或 PyScript。PyScript 是在 Pyodide 之上构建的，后者本身则是在 CPython 和 Emscripten 之上构建的。Pyodide 提供了对浏览器的 JavaScript 和 DOM API 的访问并通过 JavaScript 的 `XMLHttpRequest` 和 `Fetch` API 提供了受限的网络功能。

- 进程相关的 API 或者不可用或者将始终报错失败。这包括生成新进程 (*fork()*, *execve()*), 等待进程 (*waitpid()*), 发送信号 (*kill()*) 或者以其他方式与进程交互的 API。*subprocess* 可以被导入但将没有任何作用。
- *socket* 模块可以使用，但将会受限而使其行为与在其他平台上不一致。在 Emscripten 上，套接字将始终为非阻塞式的并且要求额外的 JavaScript 代码和服务器上的辅助工具来代理通过 WebSockets 的 TCP；请参阅 Emscripten Networking 了解详情。WASI snapshot preview 1 只允许来自现有文件描述符的套接字。
- 某些函数是不执行任何操作的空壳或是始终返回硬编码的值。
- 有关文件描述符、文件访问权、文件所有权和链接的函数均受到限制并且不支持某些操作。例如，WASI 不允许具有绝对文件名的符号链接。

1.1.2 移动平台

Android 和 iOS 在大多数方面都是 POSIX 操作系统。文件 I/O、套接字处理和线程的行为都与在任何 POSIX 操作系统上的行为相同。但是，有几个主要区别：

- 移动平台只能在” 嵌入” 模式下使用 Python。没有 Python REPL，也不能使用单独的可执行文件，如:program:*python* 或 **pip** 。要在移动应用程序中添加 Python 代码，必须使用 Python 嵌入式 API 。更多详情，请参阅 using-android 和 using-ios 。
- 子进程：
  - 在 Android 上，创建子进程是可能的，但” 非官方支持 <https://issuetracker.google.com/issues/128554619#comment4>‘__”。尤其是，Android 没有支持 System V IPC 的任何部分 API ，因此*multiprocessing* 不可用。
  - iOS 应用程序不能使用任何形式的子进程、多进程或进程间通信。如果 iOS 应用程序尝试创建子进程，创建子进程的进程要么会上锁，要么会崩溃。iOS 应用程序无法看到正在运行的其他应用，也无法与运行中的其他应用通信，除非使用为此目的而存在的 iOS 特定 API 。
- 移动应用程序修改系统资源（如系统时钟）的权限有限。这些资源通常 * 可读 *，但试图修改这些资源通常会失败。
- 控制台输入与输出：
  - 在 Android 系统上，原生的 `stdout` 和 “stderr“ 没有连接到任何设备，因此 Python 安装了自己的流，将信息重定向到系统日志。这些信息可分别在 `python.stdout` 和 `python.stderr` 标签下看到。
  - iOS 应用程序对控制台输出的概念有限。`stdout` 和 `stderr` 存在，在 Xcode 中运行时，写入 “stdout“ 和 `stderr` 的内容在日志中可见，但这些内容 * 不会 * 记录在系统日志中。如果安装了您的应用程序的用户提供其应用程序日志作为诊断辅助工具，这些日志将不包括写入 `stdout` 或 `stderr` 的任何细节。
  - 移动应用程序根本没有可用的 `stdin` 。虽然应用程序可以显示屏幕键盘，但这是软件特性，而不是连接在 `stdin` 上的。

因此，涉及控制台操作的 Python 模块（如*curses* 和*readline* ）无法在移动平台上使用。

CHAPTER 2

内置函数

Python 解释器内置了很多函数和类型，任何时候都能使用。以下按字母顺序给出列表。

内置函数

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

abs(*x*)

返回一个数字的绝对值。参数可以是整数、浮点数或任何实现了 `__abs__()` 的对象。如果参数是一个复数，则返回它的模。

aiter(*async_iterable*)

返回*asynchronous iterable* 的*asynchronous iterator* 。相当于调用 `x.__aiter__()`。

注意：与 *iter()* 不同，*aiter()* 没有两个参数的版本。

Added in version 3.10.

all(*iterable*)

如果 *iterable* 的所有元素均为真值（或可迭代对象为空）则返回 `True` 。等价于：

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

awaitable anext(*async_iterator*)

awaitable anext(*async_iterator*, *default*)

当进入 await 状态时，从给定*asynchronous iterator* 返回下一数据项，迭代完毕则返回 *default*。

这是内置函数*next()* 的异步版本，类似于：

调用 *async_iterator* 的 __anext__() 方法，返回一个*awaitable*。等待返回迭代器的下一个值。若有给出 *default*，则在迭代完毕后会返回给出的值，否则会触发*StopAsyncIteration*。

Added in version 3.10.

any(*iterable*)

如果 *iterable* 的任一元素为真值则返回 True。如果可迭代对象为空，返回 False。等价于：

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*)

与*repr()* 类似，返回一个包含对象的可打印表示形式的字符串，但是使用 \x、\u 和 \U 对*repr()* 返回的字符串中非 ASCII 编码的字符进行转义。生成的字符串和 Python 2 的*repr()* 返回的结果相似。

bin(*x*)

将一个整数转换为带前缀"0b" 的二进制数字符串。结果是一个合法的 Python 表达式。如果 *x* 不是一个 Python *int* 对象，则它必须定义返回一个整数的 __index__() 方法。下面是一些例子：

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

若要控制是否显示前缀 "0b"，可以采用以下两种方案：

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

另见*format()* 获取更多信息。

class bool(*object=False*, */ (Positional-only parameter separator (PEP 570))*)

返回布尔值，即 True 或 False 中的一个。其参数将使用标准的真值测试过程 来转换。如果该参数为假值或被省略，则返回 False；在其他情况下，将返回 True。*bool* 类是*int* 的子类 (参见数字类型 --- *int, float, complex*)。它不能被继续子类化。它只有 False 和 True 这两个实例 (参见布尔类型 - *bool*)。

在 3.7 版本发生变更: 该形参现在为仅限位置形参。

breakpoint(**args*, ***kws*)

此函数会在调用位置进入调试器。具体来说，它将调用*sys.breakpointhook()*，直接传递 args 和 kws。在默认情况下，sys.breakpointhook() 将不带参数地调用*pdb.set_trace()*。在此情况下，它纯粹是一个便捷函数让你不必显式地导入*pdb* 或键入过多代码即可进入调试器。不过，*sys.breakpointhook()* 也可被设置为某些其他函数并被*breakpoint()* 自动调用，允许你进入选定的调试器。如果*sys.breakpointhook()* 不可用，此函数将引发*RuntimeError*。

在默认情况下，*breakpoint()* 的行为可使用 PYTHONBREAKPOINT 环境变量来改变。请参阅*sys.breakpointhook()* 了解详细用法。

请注意这并不保证*sys.breakpointhook()* 会被替换。

引发一个审计事件 builtins.breakpoint 并附带参数 breakpointhook。

Added in version 3.7.

class bytearray(*source=b''*)

class bytearray(*source*, *encoding*)

class bytearray(*source*, *encoding*, *errors*)

返回一个新的 bytes 数组。*bytearray* 类是一个可变序列，包含范围为 0 <= x < 256 的整数。它有可变序列大部分常见的方法，见*可变序列类型* 的描述；同时有*bytes* 类型的大部分方法，参见*bytes 和 bytearray 操作*。

可选形参 *source* 可以用不同的方式来初始化数组：

- 如果是一个 *string*，您必须提供 *encoding* 参数（*errors* 参数仍是可选的）；*bytearray()* 会使用*str.encode()* 方法来将 string 转变成 bytes。
- 如果是一个 *integer*，会初始化大小为该数字的数组，并使用 null 字节填充。
- 如果是一个遵循 缓冲区接口的对象，该对象的只读缓冲区将被用来初始化字节数组。
- 如果是一个 *iterable* 可迭代对象，它的元素的范围必须是 `0 <= x < 256` 的整数，它会被用作数组的初始内容。

如果没有实参，则创建大小为 0 的数组。

另见*二进制序列类型 --- bytes, bytearray, memoryview* 和*bytearray 对象*。

class bytes(*source=b''*)

class bytes(*source*, *encoding*)

class bytes(*source*, *encoding*, *errors*)

返回一个新的“bytes”对象，这是一个不可变序列，包含范围为 `0 <= x < 256` 的整数。*bytes* 是*bytearray* 的不可变版本——带有同样不改变序列的方法，支持同样的索引、切片操作。

因此，构造函数的实参和*bytearray()* 相同。

字节对象还可以用字面值创建，参见 strings。

另见*二进制序列类型 --- bytes, bytearray, memoryview*，*bytes 对象* 和*bytes 和 bytearray 操作*。

callable(*object*)

如果 *object* 参数是可调用的则返回*True*，否则返回*False*。如果返回 `True`，调用仍可能失败，但如果返回 `False`，则调用 *object* 肯定不会成功。请注意类是可调用的（调用类将返回一个新的实例）；如果实例所属的类有 `__call__()` 方法则它就是可调用的。

Added in version 3.2: 这个函数一开始在 Python 3.0 被移除了，但在 Python 3.2 被重新加入。

chr(*i*)

返回 Unicode 码位为整数 *i* 的字符的字符串格式。例如，`chr(97)` 返回字符串 `'a'`，`chr(8364)` 返回字符串 `'€'`。这是*ord()* 的逆函数。

实参的合法范围是 0 到 1,114,111（16 进制表示是 0x10FFFF）。如果 *i* 超过这个范围，会触发*ValueError* 异常。

@classmethod

把一个方法封装成类方法。

类方法隐含的第一个参数就是类，就像实例方法接收实例作为参数一样。要声明一个类方法，按惯例请使用以下方案：

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

`@classmethod` 这样的形式称为函数的*decorator* -- 详情参阅 function。

类方法的调用可以在类上进行 (例如 `C.f()`) 也可以在实例上进行 (例如 `C().f()`)。其所属类以外的类实例会被忽略。如果类方法在其所属类的派生类上调用，则该派生类对象会被作为隐含的第一个参数被传入。

类方法与 C++ 或 Java 中的静态方法不同。如果你需要后者，请参阅本节中的*staticmethod()*。有关类方法的更多信息，请参阅 types。

在 3.9 版本发生变更: 类方法现在可以包装其他描述器 例如*property()*。

在 3.10 版本发生变更: 类方法现在继承了方法的属性 (__module__, __name__, __qualname__, __doc__ 和 __annotations__) 并具有新的 __wrapped__ 属性。

Deprecated since version 3.11, removed in version 3.13: 类方法不再可以包装其他*descriptors* 例如*property()*。

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

将 *source* 编译成代码或 AST 对象。代码对象可以被*exec()* 或*eval()* 执行。*source* 可以是常规的字符串、字节字符串，或者 AST 对象。参见*ast* 模块的文档了解如何使用 AST 对象。

filename 实参需要是代码读取的文件名；如果代码不需要从文件中读取，可以传入一些可辨识的值 (经常会使用 '<string>')。

mode 实参指定了编译代码必须用的模式。如果 *source* 是语句序列，可以是 'exec'；如果是单一表达式，可以是 'eval'；如果是单个交互式语句，可以是 'single'。(在最后一种情况下，如果表达式执行结果不是 None 将会被打印出来。)

可选参数 *flags* 和 *dont_inherit* 控制应当激活哪个编译器选项 以及应当允许哪个 future 特性。如果两者都未提供 (或都为零) 则代码会应用与调用*compile()* 的代码相同的旗标来编译。如果给出了 *flags* 参数而未给出 *dont_inherit* (或者为零) 则会在无论如何都将被使用的旗标之外还会额外使用 *flags* 参数所指定的编译器选项和 future 语句。如果 *dont_inherit* 为非零整数，则只使用 *flags* 参数 -- 外围代码中的旗标 (future 特性和编译器选项) 会被忽略。

编译器选项和 future 语句是由比特位来指明的。比特位可以通过一起按位 OR 来指明多个选项。指明特定 future 特性所需的比特位可以在*__future__* 模块的*_Feature* 实例的*compiler_flag* 属性中找到。编译器旗标 可以在*ast* 模块中查找带有 PyCF_ 前缀的名称。

optimize 实参指定编译器的优化级别；默认值 -1 选择与解释器的 -O 选项相同的优化级别。显式级别为 0 (没有优化；__debug__ 为真)、1 (断言被删除，__debug__ 为假) 或 2 (文档字符串也被删除)。

如果编译的源码不合法，此函数会触发*SyntaxError* 异常；如果源码包含 null 字节，则会触发*ValueError* 异常。

如果您想分析 Python 代码的 AST 表示，请参阅*ast.parse()*。

引发一个审计事件 compile 附带参数 source 和 filename。此事件也可通过隐式编译来引发。

> **备注**
>
> 在 'single' 或 'eval' 模式编译多行代码字符串时，输入必须以至少一个换行符结尾。这使*code* 模块更容易检测语句的完整性。

> **警告**
>
> 在将足够大或者足够复杂的字符串编译成 AST 对象时，Python 解释器有可能因为 Python AST 编译器的栈深度限制而崩溃。

在 3.2 版本发生变更: Windows 和 Mac 的换行符均可使用。而且在 'exec' 模式下的输入不必再以换行符结尾了。另增加了 *optimize* 参数。

在 3.5 版本发生变更: 之前 *source* 中包含 null 字节的话会触发*TypeError* 异常。

Added in version 3.8: ast.PyCF_ALLOW_TOP_LEVEL_AWAIT 现在可在旗标中传入以启用对最高层级 await, async for 和 async with 的支持。

class complex(*number=0*, */*)

class complex(*string*, */*)

class complex(*real=0*, *imag=0*)

将特定的字符串或数字转换为一个复数，或基于特定的实部和虚部创建一个复数。

示例：

```
>>> complex('+1.23')
(1.23+0j)
>>> complex('-4.5j')
-4.5j
>>> complex('-1.23+4.5j')
(-1.23+4.5j)
>>> complex('\t( -1.23+4.5J )\n')
(-1.23+4.5j)
>>> complex('-Infinity+NaNj')
(-inf+nanj)
>>> complex(1.23)
(1.23+0j)
>>> complex(imag=-4.5)
-4.5j
>>> complex(-1.23, 4.5)
(-1.23+4.5j)
```

如果该参数为字符串，则它必须包含一个实部（使用与*float()* 相同的格式）或一个虚部（使用相同的格式但带有 `'j'` 或 `'J'` 后缀），或者同时包含实部和虚部（在此情况下虚部必须加上正负号）。该字符串首尾可以选择加上空格和圆括号 `'('` and `')'`，它们将会被忽略。该字符串的，which are ignored. The string must not contain whitespace between `'+'`, `'-'`, `'j'` 或 `'J'` 后缀以及十进制数字之间不可包含空格。例如，`complex('1+2j')` 是可以的，但 `complex('1 + 2j')` 则会引发*ValueError*。更准确地说，输入在移除圆括号以及开头和末尾的空格符之后，必须符合使用以下语法的*complexvalue* 产生规则：

```
complexvalue ::= floatvalue |
                 floatvalue ("j" | "J") |
                 floatvalue sign absfloatvalue ("j" | "J")
```

如果该参数为数字，则此构造器将进行与*int* 和*float* 类似的数值转换。对于一个普通的 Python 对象 `x`，`complex(x)` 会委托给 `x.__complex__()`。如果未定义 `__complex__()` 则它将回退至 `__float__()`。如果未定义 `__float__()` 则它将回退至 `__index__()`。

如果提供了两个参数或是使用了关键字参数，则每个参数可以为任意数字类型（包括复数）。如果两个参数均为实数值，则会返回一个实部为 *real* 而虚部为 *imag* 的复数。如果两个参数均为复数值，则会返回一个实部为 `real.real-imag.imag` 而虚部为 `real.imag+imag.real` 的复数。如果有一个参数为实数值，则上面的表达式中将只用到实部。

如果省略所有参数，则返回 `0j`。

数字类型 --- int, float, complex 描述了复数类型。

在 3.6 版本发生变更: 您可以使用下划线将代码文字中的数字进行分组。

在 3.8 版本发生变更: 如果 `__complex__()` 和 `__float__()` 均未定义则回退至 `__index__()`。

delattr(*object*, *name*)

这是*setattr()* 的相关函数。其参数是一个对象和一个字符串。其中字符串必须是对象的某个属性的名称。该函数会删除指定的属性，如果对象允许这样做的话。例如，`delattr(x, 'foobar')` 等价于 `del x.foobar`。*name* 不要求必须是 Python 标识符 (参见*setattr()*)。

class dict(***kwarg*)

class dict(*mapping*, ***kwarg*)

class dict(*iterable*, ***kwarg*)

创建一个新的字典。*dict* 对象是一个字典类。参见*dict* 和*映射类型 --- dict* 了解这个类。

其他容器类型，请参见内置的*list*、*set* 和*tuple* 类，以及*collections* 模块。

dir()

dir(*object*)

如果没有实参，则返回当前本地作用域中的名称列表。如果有实参，它会尝试返回该对象的有效属性列表。

如果对象有一个名为 `__dir__()` 的方法，则该方法将被调用并且必须返回由属列组成的列表。这允许实现自定义 This allows objects that implement a custom `__getattr__()` 或 `__getattribute__()` 函数的对象能够定制*dir()* 报告其属性的方式。

如果对象未提供 `__dir__()`，该函数会尽量从对象所定义的 `__dict__` 属性和其类型对象中收集信息。结果列表不一定是完整的，并且当对象具有自定义的 `__getattr__()` 时还可能是不准确的。

默认的*dir()* 机制对不同类型的对象行为不同，它会试图返回最相关而不是最全的信息：

- 如果对象是模块对象，则列表包含模块的属性名称。
- 如果对象是类型或类对象，则列表包含它们的属性名称，并且递归查找所有基类的属性。
- 否则，列表包含对象的属性名称，它的类属性名称，并且递归查找它的类的所有基类的属性。

返回的列表按字母表排序。例如：

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
...
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

> **备注**
>
> 因为*dir()* 主要是为了便于在交互式时使用，所以它会试图返回人们感兴趣的名字集合，而不是试图保证结果的严格性或一致性，它具体的行为也可能在不同版本之间改变。例如，当实参是一个类时，metaclass 的属性不包含在结果列表中。

divmod(*a*, *b*)

接受两个（非复数）数字作为参数并返回由当对其使用整数除法时的商和余数组成的数字对。在混用不同的操作数类型时，则会应用二元算术运算符的规则。对于整数来说，结果与 `(a // b, a % b)` 相同。对于浮点数来说则结果为 `(q, a % b)`，其中 *q* 通常为 `math.floor(a / b)` 但可能会比它小 1。在任何情况下 `q * b + a % b` 都非常接近 *a*，如果 `a % b` 为非零值则它将具有与 *b* 相同的正负号，并且 `0 <= abs(a % b) < abs(b)`。

enumerate(*iterable*, *start=0*)

返回一个枚举对象。*iterable* 必须是一个序列，或*iterator*，或其他支持迭代的对象。*enumerate()* 返回的迭代器的*__next__()* 方法返回一个元组，里面包含一个计数值（从 *start* 开始，默认为 0）和通过迭代 *iterable* 获得的值。

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

等价于:

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

eval(*source*, */*, *globals=None*, *locals=None*)

参数

- **source** (*str* | code object) -- 一个 Python 表达式。
- **globals** (*dict* | None) -- 全局命名空间 (默认值: None)。
- **locals** (*mapping* | None) -- 局部命名空间 (默认值: None)。

返回

被求值表达式的求值结果。

引发

语法错误将作为异常被报告。

警告

此函数可执行任意代码。调用它时附带用户提供的输入可能导致安全弱点。

expression 参数将作为一个 Python 表达式 (从技术上说，是一个条件列表) 使用 *globals* 和 *locals* 映射作为全局和局部命名空间被解析并求值。如果 *globals* 字典存在并且不包含 __builtins__ 键对应的值，则在 *expression* 被解析之前会插入该键对应的指向内置模块*builtins* 的字典的引用。这样你就可以在将 *globals* 传给*eval()* 之前通过向其传入你自己的 __builtins__ 字典来控制被执行代码可以使用哪些内置对象。如果 *locals* 映射被省略则它将默认为 *globals* 字典。如果两个映射都被省略，则将使用调用*eval()* 所在环境中的 *globals* 和 *locals* 来执行该表达式。请注意，*eval()* 将只能访问所在环境中的嵌套作用域 (非局部作用域)，如果它们已经在调用*eval()* 的作用域中被引用的话 (例如通过 nonlocal 语句)。

示例:

```
>>> x = 1
>>> eval('x+1')
2
```

该函数还可用于执行任意代码对象（比如由*compile()* 创建的对象）。这时传入的是代码对象，而非一个字符串了。如果代码对象已用参数为 *mode* 的 'exec' 进行了编译，那么*eval()* 的返回值将为 None。

提示：*exec()* 函数支持语句的动态执行。*globals()* 和*locals()* 函数分别返回当前的全局和本地字典，可供传给*eval()* 或*exec()* 使用。

如果给出的源数据是个字符串，那么其前后的空格和制表符将被剔除。

另外可以参阅*ast.literal_eval()*，该函数可以安全执行仅包含文字的表达式字符串。

引发一个审计事件 exec 附带代码对象作为参数。代码编译事件也可能被引发。

在 3.13 版本发生变更: 现在可以将 *globals* 和 *locals* 作为关键字参数传入。

在 3.13 版本发生变更: 默认 *locals* 命名空间的语义已被调整为与*locals()* 内置函数的描述一致。

exec(*source*, */*, *globals=None*, *locals=None*, ** (Keyword-only parameters separator (PEP 3102))*, *closure=None*)

警告

此函数可执行任意代码。调用它时附带用户提供的输入可能导致安全弱点。

这个函数支持动态执行 Python 代码。*source* 必须是字符串或代码对象。如果是字符串，那么该字符串将被解析为一组 Python 语句并随即被执行（除非发生语法错误）。[1] 如果是代码对象，那么它将被直接执行。在所有情况下，被执行的代码都应当是有效的文件输入（见参考手册中的 file-input 一节）。请注意即使是在传递给 `exec()` 函数的代码的上下文中 `nonlocal`, `yield` 和 `return` 语句也不可在函数定义以外使用。函数的返回值为 `None`。

在所有情况下，如果省略了可选部分，代码将在当前作用域中执行。如果只提供了 *globals*，则它必须是一个字典（并且不能是字典的子类），它将被同时用于全局和局部变量。如果给出了 *globals* 和 *locals*，它们将被分别用于全局和局部变量。如果提供了 *locals*，它可以是任何映射对象。请记住在模块层级上，globals 和 locals 是同一个字典。

备注

当 `exec` 获得两个不同的对象作为 *globals* 和 *locals* 时，代码被执行时就会像是嵌套在一个类定义中那样。这意味着在被执行代码中定义的函数和类将无法访问在最高层级上赋值的变量（因为“最高层级”变量会被当作是类定义中的类变量来对待）。

如果 *globals* 字典不包含 `__builtins__` 键值，则将为该键插入对内建 `builtins` 模块字典的引用。因此，在将执行的代码传递给 `exec()` 之前，可以通过将自己的 `__builtins__` 字典插入到 *globals* 中来控制可以使用哪些内置代码。

closure 参数指定了一个闭包——一个单元变量的元组。它只有有 *object* 是一个包含自由（闭包）变量 的代码对象时才有效。元组的长度必须与代码对象的 `co_freevars` 属性的长度完全匹配。

引发一个审计事件 `exec` 附带代码对象作为参数。代码编译事件也可能被引发。

备注

内置函数 `globals()` 和 `locals()` 分别返回当前的全局和局部字典，这在用作 `exec()` 的第二个和第三个参数进行传递时会很有用处。

备注

默认的 *locals* 行为与下面 `locals()` 函数所描述的一样。如果你需要在 `exec()` 返回之后查看代码对 *locals* 的影响可以显式地传入一个 *locals* 字典。

在 3.11 版本发生变更: 添加了 *closure* 参数。

在 3.13 版本发生变更: 现在可以将 *globals* 和 *locals* 作为关键字参数传入。

在 3.13 版本发生变更: 默认 *locals* 命名空间的语义已被调整为与 `locals()` 内置函数的描述一致。

filter(*function*, *iterable*)

使用 *iterable* 中 *function* 返回真值的元素构造一个迭代器。*iterable* 可以是一个序列，一个支持迭代的容器或者一个迭代器。如果 *function* 为 `None`，则会使用标识号函数，也就是说，*iterable* 中所有具有假值的元素都将被移除。

请注意，`filter(function, iterable)` 相当于一个生成器表达式，当 function 不是 `None` 的时候为 `(item for item in iterable if function(item))`；function 是 `None` 的时候为 `(item for item in iterable if item)` 。

[1] 解析器只接受 Unix 风格的行结束符。如果您从文件中读取代码，请确保用换行符转换模式转换 Windows 或 Mac 风格的换行符。

请参阅*itertools.filterfalse()* 来了解返回 *iterable* 中 *function* 返回假值的元素的补充函数。

class float(*number=0.0*, */*)

class float(*string*, */*)

返回基于一个数字或字符串构建的浮点数。

示例：

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

如果该参数是一个字符串，则它应当包含一个十进制数字，前面可以选择带一个符号，也可以选择嵌入空格。可选的符号有 `'+'` 或 `'-'`；`'+'` 符号对所产生的值没有影响。该参数还可以是一个代表 NaN (not-a-number) 或者正负无穷大的字符串。更确切地说，在移除前导和尾随的空格之后，输入必须为符合以下语法的*floatvalue* 产生规则：

```
sign           ::= "+" | "-"
infinity       ::= "Infinity" | "inf"
nan            ::= "nan"
digit          ::= <a Unicode decimal digit, i.e. characters in Unicode general category Nd>
digitpart      ::= digit (["_"] digit)*
number         ::= [digitpart] "." digitpart | digitpart ["."]
exponent       ::= ("e" | "E") [sign] digitpart
floatnumber    ::= number [exponent]
absfloatvalue ::= floatnumber | infinity | nan
floatvalue     ::= [sign] absfloatvalue
```

大小写是无影响的，因此举例来说，”inf”, ”Inf”, ”INFINITY” 和”iNfINity” 都是正无穷可接受的拼写形式。

另一方面，如果参数是整数或浮点数，则返回一个具有相同值（在 Python 浮点精度范围内）的浮点数。如果参数超出了 Python 浮点数的取值范围，则会引发*OverflowError*。

对于一个普通 Python 对象 `x`，`float(x)` 会委托给 `x.__float__()`。如果 `__float__()` 未定义则将回退至 `__index__()`。

如果没有实参，则返回 `0.0` 。

数字类型 --- int, float, complex 描述了浮点类型。

在 3.6 版本发生变更: 您可以使用下划线将代码文字中的数字进行分组。

在 3.7 版本发生变更: 该形参现在为仅限位置形参。

在 3.8 版本发生变更: 如果 `__float__()` 未定义则回退至 `__index__()`。

format(*value*, *format_spec=''*)

将 *value* 转换为 “格式化后” 的形式，格式由 *format_spec* 进行控制。*format_spec* 的解释方式取决于 *value* 参数的类型；但大多数内置类型使用一种标准的格式化语法：*格式规格迷你语言*。

默认的 *format_spec* 是一个空字符串，它通常给出与调用*str(value)* 相同的结果。

对 `format(value, format_spec)` 的调用会转写为 `type(value).__format__(value, format_spec)`，这样在搜索值的 `__format__()` 方法时将绕过实例字典。如果方法搜索到达*object* 并且 *format_spec* 不为空，或者如果 *format_spec* 或返回值不为字符串则会引发*TypeError* 异常。

在 3.4 版本发生变更: 当 *format_spec* 不是空字符串时，`object().__format__(format_spec)` 会触发 *TypeError*。

class frozenset(*iterable=set()*)

返回一个新的 *frozenset* 对象，它包含可选参数 *iterable* 中的元素。`frozenset` 是一个内置的类。有关此类的文档，请参阅 *frozenset* 和 *集合类型 --- set, frozenset*。

请参阅内建的 *set*、*list*、*tuple* 和 *dict* 类，以及 *collections* 模块来了解其它的容器。

getattr(*object*, *name*)

getattr(*object*, *name*, *default*)

object 中指定名称的属性的值。*name* 必须是字符串。如果该字符串是对象的某一属性的名称，则结果将为该属性的值。例如，`getattr(x, 'foobar')` 等同于 `x.foobar`。如果指定名称的属性不存在，则如果提供了 *default* 则返回该值，否则将引发 *AttributeError*。*name* 不必是一个 Python 标识符 (参见 *setattr()*)。

> **备注**
>
> 由于 私有名称混合发生在编译时，因此必须手动混合私有属性（以两个下划线打头的属性）名称以使用 *getattr()* 来提取它。

globals()

返回实现当前模块命名空间的字典。对于函数内的代码，这是在定义函数时设置的，无论函数在哪里被调用都保持不变。

hasattr(*object*, *name*)

该实参是一个对象和一个字符串。如果字符串是对象的属性之一的名称，则返回 `True`，否则返回 `False`。(此功能是通过调用 `getattr(object, name)` 看是否有 *AttributeError* 异常来实现的。)

hash(*object*)

返回该对象的哈希值（如果它有的话）。哈希值是整数。它们在字典查找元素时用来快速比较字典的键。相同大小的数字变量有相同的哈希值（即使它们类型不同，如 1 和 1.0）。

> **备注**
>
> 对于具有自定义 `__hash__()` 方法的对象，请注意 *hash()* 会根据宿主机的字长来截断返回值。

help()

help(*request*)

启动内置的帮助系统（此函数主要在交互式中使用）。如果没有实参，解释器控制台里会启动交互式帮助系统。如果实参是一个字符串，则在模块、函数、类、方法、关键字或文档主题中搜索该字符串，并在控制台上打印帮助信息。如果实参是其他任意对象，则会生成该对象的帮助页。

请注意，如果在调用 *help()* 时，目标函数的形参列表中存在斜杠 (/)，则意味着斜杠之前的参数只能是位置参数。详情请参阅 有关仅限位置形参的 FAQ 条目。

该函数通过 *site* 模块加入到内置命名空间。

在 3.4 版本发生变更: *pydoc* 和 *inspect* 的变更使得可调用对象的签名信息更加全面和一致。

hex(*x*)

将整数转换为带前缀"0x" 前缀的小写十六进制数字符串。如果 *x* 不是一个 Python *int* 对象，则它必须定义返回一个整数的 `__index__()` 方法。下面是一些例子:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

如果要将整数转换为大写或小写的十六进制字符串，并可选择有无“0x”前缀，则可以使用如下方法：

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

另见*format()* 获取更多信息。

另请参阅*int()* 将十六进制字符串转换为以 16 为基数的整数。

备注

如果要获取浮点数的十六进制字符串形式，请使用*float.hex()* 方法。

id(*object*)

返回对象的“标识值”。该值是一个整数，在此对象的生命周期中保证是唯一且恒定的。两个生命期不重叠的对象可能具有相同的*id()* 值。

这是对象在内存中的地址。

引发一个审计事件 `builtins.id` 并附带参数 `id`。

input()

input(*prompt*)

如果存在 *prompt* 实参，则将其写入标准输出，末尾不带换行符。接下来，该函数从输入中读取一行，将其转换为字符串（除了末尾的换行符）并返回。当读取到 EOF 时，则触发*EOFError*。例如:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

如果加载了*readline* 模块，*input()* 将使用它来提供复杂的行编辑和历史记录功能。

在读取输入前引发一个审计事件 `builtins.input` 附带参数 `prompt`

在成功读取输入之后引发一个审计事件 `builtins.input/result` 附带结果。

class int(*number=0*, */*)

class int(*string*, */*, *base=10*)

返回从一个数字或字符串构建的整数对象，或者如果未给出参数则返回 `0`。

示例：

```
>>> int(123.45)
123
>>> int('123')
123
>>> int('   -12_345\n')
-12345
>>> int('FACE', 16)
64206
>>> int('0xface', 0)
64206
>>> int('01110011', base=2)
115
```

如果参数定义了 `__int__()`，`int(x)` 将返回 `x.__int__()`。如果参数定义了 `__index__()`，它将返回 `x.__index__()`。如果参数定义了 `__trunc__()`，它将返回 `x.__trunc__()`。对于浮点数，这将向零方向截断。

如果参数不是数字或者如果给定了 *base*，则它必须是表示一个以 *base* 为基数的整数的字符串、*bytes* 或*bytearray* 实例。字符串前面还可选择加上 + 或 - (中间没有空格)，带有前导的零，带有两侧的空格，以及带有数位之间的单个下划线。

一个以 n 为基数的整数字符串包含多个数位，每个数位代表从 0 到 n-1 范围内的值。0--9 的值可以用任何 Unicode 十进制数码来表示。10--35 的值可以用 `a` 到 `z` (或 `A` 到 `Z`) 来表示。默认的 *base* 为 10。允许的基数为 0 和 2--36。对于基数 2, -8 和 -16 来说字符串前面还能加上可选的 `0b`/`0B`, `0o`/`0O` 或 `0x`/`0X` 前缀，就像代码中的整数字面值那样。对于基数 0 来说，字符串会以与 代码中的整数字面值类似的方式来解读，即实际的基数将由前缀确定为 2, 8, 10 或 16。基数为 0 还会禁用前导的零: `int('010', 0)` 将是无效的，而 `int('010')` 和 `int('010', 8)` 则是有效的。

整数类型定义请参阅数字类型 --- *int, float, complex* 。

在 3.4 版本发生变更: 如果 *base* 不是*int* 的实例，但 *base* 对象有 `base.__index__` 方法，则会调用该方法来获取进制数。以前的版本使用 `base.__int__` 而不是 `base.__index__`。

在 3.6 版本发生变更: 您可以使用下划线将代码文字中的数字进行分组。

在 3.7 版本发生变更: 第一个形参现在是仅限位置形参。

在 3.8 版本发生变更: 如果 `__int__()` 未定义则回退至 `__index__()`。

在 3.11 版本发生变更: 委托给 `__trunc__()` 的做法已被弃用。

在 3.11 版本发生变更: *int* 字符串输入和字符串表示形式可受到限制以帮助避免拒绝服务攻击。当将一个字符串转换为*int* 或者将一个*int* 转换为字符串的操作走出限制时会引发*ValueError*。请参阅整数字符串转换长度限制 文档。

isinstance(*object*, *classinfo*)

如果 *object* 参数是 *classinfo* 参数的实例，或者是其 (直接、间接或虚拟) 子类的实例则返回 `True`。如果 *object* 不是给定类型的对象，则该函数总是返回 `False`。如果 *classinfo* 是由类型对象结成的元组 (或是由其他此类元组递归生成) 或者是多个类型的*union 类型*，则如果 *object* 是其中任一类型的实例时将会返回 `True`。如果 *classinfo* 不是一个类型或类型元组及此类元组，则会引发*TypeError* 异常。如果之前的检查成功执行则可以不会为无效的类型引发*TypeError*。

在 3.10 版本发生变更: *classinfo* 可以是一个*union 类型*。

issubclass(*class*, *classinfo*)

如果 *class* 是 *classinfo* 的子类（直接、间接或虚的 ），则返回 `True`。类将视为自己的子类。*classinfo* 可为类对象的元组（或递归地，其他这样的元组）或*union 类型*，这时如果 *class* 是 *classinfo* 中任何条目的子类，则返回 `True` 。任何其他情况都会触发*TypeError* 异常。

在 3.10 版本发生变更: *classinfo* 可以是一个*union 类型*。

iter(*object*)

iter(*object*, *sentinel*)

返回一个*iterator* 对象。根据是否存在第二个参数，对第一个参数的解读会有很大的不同。如果没有第二个参数，*object* 必须是一个支持*iterable* 协议 (有 `__iter__()` 方法) 的多项集对象，或者必须支持序列协议 (有 `__getitem__()` 方法并使用从 `0` 开始的整数参数)。如果它不支持这些协议，则会引发*TypeError*。如果给出了第二个参数 *sentinel*，则 *object* 必须是一个可调用对象。在这种情况下创建的迭代器将针对每次调用其*__next__()* 方法不带参数地调用 *object*；如果返回的值等于 *sentinel*，则会引发*StopIteration*，否则将返回该值。

另请参阅迭代器类型。

适合*iter()* 的第二种形式的应用之一是构建块读取器。例如，从二进制数据库文件中读取固定宽度的块，直至到达文件的末尾:

```
from functools import partial
with open('mydata.db', 'rb') as f:
```

(续下页)

(接上页)

```
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*s*)

返回对象的长度（元素个数）。实参可以是序列（如 string、bytes、tuple、list 或 range 等）或集合（如 dictionary、set 或 frozen set 等）。

`len` 对于大于 `sys.maxsize` 的长度如 `range(2 ** 100)` 会引发 `OverflowError`。

class list

class list(*iterable*)

虽然被称为函数，`list` 实际上是一种可变序列类型，详情请参阅列表 和序列类型 --- *list, tuple, range*。

locals()

返回一个代表当前局部符号表的映射对象，以变量名称作为键，而以其当前绑定的引用作为值。

在模块作用域上，以及当附带单个命名空间使用 `exec()` 或 `eval()` 时，此函数将返回与 `globals()` 相同的命名空间。

在类作用域上，它会返回将被传给元类构造器的命名空间。

当附带不同的 local 和 global 参数使用 `exec()` 或 `eval()` 时，它将返回传入函数调用的 local 命名空间。

在上述所有情况下，在一个给定的执行帧中对 `locals()` 的每次调用都将返回 同一个映射对象。通过从 `locals()` 返回的映射对象所做的修改都将如局部变量的赋值、重新赋值或删除一样可见，而局部变量的赋值、重新赋值或删除都将立即影响所返回映射对象的内容。

在一个*optimized scope* 中（包括函数、生成器和协程），每个对 `locals()` 的调用将改为返回一个新字典，其中包含函数的局部变量及任何非局部单元引用的当前绑定。在此情况下，通过所返回字典对名称绑定的改变将 不会写回到对应的局部变量或非局部单元引用，并且赋值、重新赋值或删除局部变量和非局部单元引用也 不会影响之前返回的字典的内容。affect the contents of previously returned dictionaries.

将 `locals()` 作为函数、生成器或协程中的一个推导式的组成部分来调用相当于在外层作用域中调用它，不同之处在于推导式所初始化的迭代变量将被包括在内。在其他作用域下，其行为与将推导式作为嵌套函数来运行类似。

将 `locals()` 作为生成器表达式的组成部分来调用相当于在嵌套的生成器函数中调用它。

在 3.12 版本发生变更: 在推导式中的 `locals()` 的行为已被更新为符合 **PEP 709** 中的描述。

在 3.13 版本发生变更: 作为 **PEP 667** 的组成部分，改变从此函数返回的映射对象的语义现在已获得定义。在已优化作用域 中的行为现在如上所述。除了已获得定义，在其他作用域中的行为相比之前的版本仍然保持不变。

map(*function*, *iterable*, **iterables*)

返回一个将 *function* 应用于 *iterable* 的每一项，并产生其结果的迭代器。如果传入了额外的 *iterables* 参数，则 *function* 必须接受相同个数的参数并被用于到从所有可迭代对象中并行获取的项。当有多个可迭代对象时，当最短的可迭代对象耗尽则整个迭代将会停止。对于函数的输入已经是参数元组的情况，请参阅 `itertools.starmap()`。

max(*iterable*, *, *key=None*)

max(*iterable*, *, *default*, *key=None*)

max(*arg1*, *arg2*, **args*, *key=None*)

返回可迭代对象中最大的元素，或者返回两个及以上实参中最大的。

如果只提供了一个位置参数，它必须是非空*iterable*，返回可迭代对象中最大的元素；如果提供了两个及以上的位置参数，则返回最大的位置参数。

有两个可选只能用关键字的实参。*key* 实参指定排序函数用的参数，如传给*list.sort()* 的。*default* 实参是当可迭代对象为空时返回的值。如果可迭代对象为空，并且没有给 *default* ，则会触发*ValueError*。

如果有多个最大元素，则此函数将返回第一个找到的。这和其他稳定排序工具如 `sorted(iterable, key=keyfunc, reverse=True)[0]` 和 `heapq.nlargest(1, iterable, key=keyfunc)` 保持一致。

在 3.4 版本发生变更: 增加了 *default* 仅限关键字形参。

在 3.8 版本发生变更: *key* 可以为 `None`。

class memoryview(*object*)

返回由给定实参创建的“内存视图”对象。有关详细信息，请参阅内存视图。

min(*iterable*, *, *key=None*)

min(*iterable*, *, *default*, *key=None*)

min(*arg1*, *arg2*, **args*, *key=None*)

返回可迭代对象中最小的元素，或者返回两个及以上实参中最小的。

如果只提供了一个位置参数，它必须是*iterable*，返回可迭代对象中最小的元素；如果提供了两个及以上的位置参数，则返回最小的位置参数。

有两个可选只能用关键字的实参。*key* 实参指定排序函数用的参数，如传给*list.sort()* 的。*default* 实参是当可迭代对象为空时返回的值。如果可迭代对象为空，并且没有给 *default* ，则会触发*ValueError*。

如果有多个最小元素，则此函数将返回第一个找到的。这和其他稳定排序工具如 `sorted(iterable, key=keyfunc)[0]` 和 `heapq.nsmallest(1, iterable, key=keyfunc)` 保持一致。

在 3.4 版本发生变更: 增加了 *default* 仅限关键字形参。

在 3.8 版本发生变更: *key* 可以为 `None`。

next(*iterator*)

next(*iterator*, *default*)

通过调用*iterator* 的*__next__()* 方法获取下一个元素。如果迭代器耗尽，则返回给定的 *default*，如果没有默认值则触发*StopIteration*。

class object

这是所有其他类的终极基类。它提供了所有 Python 类实例均具有的方法。当其构造器被调用时，它将返回一个新的基本对象。该构造器不接受任何参数。

> **备注**
>
> *object* 实例没有 `__dict__` 属性，因此你无法将任意属性赋给*object* 的实例。

oct(*x*)

将整数转换为带前缀”0o” 的八进制数字符串。结果是一个合法的 Python 表达式。如果 *x* 不是一个 Python *int* 对象，则它必须定义返回一个整数的 `__index__()` 方法。例如:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

若要将整数转换为八进制字符串，并可选择是否带有“0o” 前缀，可采用如下方法:

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

另见*format()* 获取更多信息。

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

打开 *file* 并返回对应的*file object*。如果该文件不能被打开，则引发*OSError*。请参阅 tut-files 获取此函数的更多用法示例。

file 是一个*path-like object*，表示将要打开的文件的路径（绝对路径或者相对当前工作目录的路径），也可以是要封装文件对应的整数类型文件描述符。（如果给出的是文件描述符，则当返回的 I/O 对象关闭时它也会关闭，除非将 *closefd* 设为 `False` 。）

mode 是一个指明文件打开模式的可选字符串。它默认为 `'r'` 表示以文本模式读取。其他常见模式有表示写入的 `'w'` (若文件已存在则将其清空)，表示独占创建的 `'x'`，以及表示追加写入的 `'a'` (在某些 Unix 系统上，这意味着无论当前查找位置在哪里 所有写入操作都将追加到文件末尾)。在文本模式下，如果未指定 *encoding* 则所使用的编码格式将依赖于具体平台: *locale.getencoding()* 会被调用以获取当前语言区域的编码格式。(对于读取和写入原始字节数据请使用二进制模式并且不要指定 *encoding*。) 可用的模式有:

字符	含意
`'r'`	读取（默认）
`'w'`	写入，并先截断文件
`'x'`	排它性创建，如果文件已存在则失败
`'a'`	打开文件用于写入，如果文件存在则在末尾追加
`'b'`	二进制模式
`'t'`	文本模式（默认）
`'+'`	打开用于更新（读取与写入）

默认模式为 `'r'` （打开文件用于读取文本，与 `'rt'` 同义）。`'w+'` 和 `'w+b'` 模式将打开文件并清空内容。而 `'r+'` 和 `'r+b'` 模式将打开文件但不清空内容。

正如在概述 中提到的，Python 区分二进制和文本 I/O。以二进制模式打开的文件（包括 *mode* 参数中的 `'b'` ）返回的内容为*bytes* 对象，不进行任何解码。在文本模式下（默认情况下，或者在 *mode* 参数中包含 `'t'` ）时，文件内容返回为*str*，首先使用指定的 *encoding* （如果给定）或者使用平台默认的的字节编码解码。

备注

Python 不依赖于底层操作系统的文本文件概念; 所有处理都由 Python 本身完成，因此与平台无关。

buffering 是一个可选的整数，用于设置缓冲策略。传入 0 来关闭缓冲（仅在二进制模式下允许），传入 1 来选择行缓冲（仅在文本模式下写入时可用），传一个整数 > 1 来表示固定大小的块缓冲区的字节大小。注意这样指定缓冲区的大小适用于二进制缓冲的 I/O, 但 `TextIOWrapper` (即用 `mode='r+'` 打开的文件) 会有另一种缓冲。要禁用 `TextIOWrapper` 中的缓冲，请考虑为*io.TextIOWrapper.reconfigure()* 使用 `write_through` 旗标。当没有给出 *buffering* 参数时，默认的缓冲策略规则如下:

- 二进制文件以固定大小的块进行缓冲；缓冲区的大小是使用启发方式来尝试确定底层设备的“块大小”并会回退至*io.DEFAULT_BUFFER_SIZE*。在许多系统上，缓冲区的长度通常为 4096 或 8192 字节。

- “交互式”文本文件（*isatty()* 返回 True 的文件）使用行缓冲。其他文本文件使用上述策略用于二进制文件。

encoding 是用于编码或编码文件的编码格式名称。这应当只有文本模式下使用。默认的编码格式依赖于具体平台 (即*locale.getencoding()* 所返回的值)，但是任何 Python 支持的*text encoding* 都可以被使用。请参阅*codecs* 模块获取受支持的编码格式列表。

errors 是一个可选的字符串参数，用于指定如何处理编码和解码错误 - 这不能在二进制模式下使用。可以使用各种标准错误处理程序（列在*错误处理方案*），但是使用*codecs.register_error()* 注册的任何错误处理名称也是有效的。标准名称包括:

- 如果存在编码错误，'strict' 会引发*ValueError* 异常。默认值 None 具有相同的效果。
- 'ignore' 忽略错误。请注意，忽略编码错误可能会导致数据丢失。
- 'replace' 会将替换标记（例如 '?'）插入有错误数据的地方。
- 'surrogateescape' 将把任何不正确的字节表示为 U+DC80 至 U+DCFF 范围内的下方替代码位。当在写入数据时使用 surrogateescape 错误处理器时这些替代码位会被转回到相同的字节。这适用于处理具有未知编码格式的文件。
- 'xmlcharrefreplace' 仅在写入文件时才受到支持。编码格式不支持的字符将被替换为相应的 XML 字符引用 &#*nnn*;。
- 'backslashreplace' 用 Python 的反向转义序列替换格式错误的数据。
- 'namereplace' （也只在编写时支持）用 \N{...} 转义序列替换不支持的字符。

newline 决定如何解析来自流的换行符。它可以为 None, '', '\n', '\r' 和 '\r\n'。它的工作原理如下:

- 从流中读取输入时，如果 *newline* 为 None，则启用通用换行模式。输入中的行可以以 '\n', '\r' 或 '\r\n' 结尾，这些行被翻译成 '\n' 在返回呼叫者之前。如果它是 ''，则启用通用换行模式，但行结尾将返回给调用者未翻译。如果它具有任何其他合法值，则输入行仅由给定字符串终止，并且返回给调用者时行结尾不会被转换。
- 将输出写入流时，如果 *newline* 为 None，则写入的任何 '\n' 字符都将转换为系统默认行分隔符*os.linesep*。如果 *newline* 是 '' 或 '\n'，则不进行翻译。如果 *newline* 是任何其他合法值，则写入的任何 '\n' 字符将被转换为给定的字符串。

如果 *closefd* 为 False 且给出的不是文件名而是文件描述符，那么当文件关闭时，底层文件描述符将保持打开状态。如果给出的是文件名，则 *closefd* 必须为 True （默认值），否则将触发错误。

可以通过传递可调用的 *opener* 来使用自定义开启器。然后通过使用参数（*file*，*flags*）调用 *opener* 获得文件对象的基础文件描述符。*opener* 必须返回一个打开的文件描述符（使用*os.open* as *opener* 时与传递 None 的效果相同）。

新创建的文件是*不可继承的*。

下面的示例使用*os.open()* 函数的*dir_fd* 的形参，从给定的目录中用相对路径打开文件:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # 不要泄漏文件描述符
```

open() 函数所返回的*file object* 类型取决于所用模式。当使用*open()* 以文本模式 ('w', 'r', 'wt', 'rt' 等) 打开文件时，它将返回*io.TextIOBase* (具体为*io.TextIOWrapper*) 的一个子类。当使用缓冲以二进制模式打开文件时，返回的类是*io.BufferedIOBase* 的一个子类。具体的类会有多种: 在只读的二进制模式下，它将返回*io.BufferedReader*; 在写入二进制和追加二进制模式下，它将返回*io.BufferedWriter*，而在读/写模式下，它将返回*io.BufferedRandom*。当禁用缓冲时，则会返回原始流，即*io.RawIOBase* 的一个子类*io.FileIO*。

另请参阅文件操作模块，如*fileinput*、*io*（声明了*open()*）、*os*、*os.path*、*tempfile* 和*shutil*。

引发一个审计事件 `open` 并附带参数 `path`, `mode`, `flags`。

`mode` 与 `flags` 参数可以在原始调用的基础上被修改或传递。

在 3.3 版本发生变更:

- 增加了 *opener* 形参。
- 增加了 `'x'` 模式。
- 过去触发的*IOError*，现在是*OSError* 的别名。
- 如果文件已存在但使用了排它性创建模式（`'x'`），现在会触发*FileExistsError*。

在 3.4 版本发生变更:

- 文件现在禁止继承。

在 3.5 版本发生变更:

- 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。
- 增加了 `'namereplace'` 错误处理接口。

在 3.6 版本发生变更:

- 增加对实现了*os.PathLike* 对象的支持。
- 在 Windows 上，打开一个控制台缓冲区将返回*io.RawIOBase* 的子类，而不是*io.FileIO*。

在 3.11 版本发生变更: `'U'` 模式已被移除。

ord(*c*)

对表示单个 Unicode 字符的字符串，返回代表它 Unicode 码点的整数。例如 `ord('a')` 返回整数 `97`，`ord('€')`（欧元符号）返回 `8364` 。这是*chr()* 的逆函数。

pow(*base*, *exp*, *mod=None*)

返回 *base* 的 *exp* 次幂；如果 *mod* 存在，则返回 *base* 的 *exp* 次幂对 *mod* 取余（比 `pow(base, exp) % mod` 更高效）。两参数形式 `pow(base, exp)` 等价于乘方运算符: `base**exp`。

这些参数必须为数字类型。对于混用的操作数类型，将应用二元算术运算的强制转换规则。对于*int* 操作数，结果具有与操作数相同的类型（转换之后）除非第二个参数为负值；在那种情况下，所有参数将被转换为浮点数并输出浮点数的结果。例如，`pow(10, 2)` 返回 `100`，而 `pow(10, -2)` 返回 `0.01`。对于*int* 或*float* 类型的基数为负值而幂为非整数的情况，将产生一个复数的结果。例如，`pow(-9, 0.5)` 将返回一个接近 `3j` 的值。最后，对于*int* 或*float* 类型的基数为负值而幂为整数的情况，将产生一个浮点数的结果。例如，`pow(-9, 2.0)` 将返回 `81.0`。

对于*int* 操作数 *base* 和 *exp*，如果给出 *mod*，则 *mod* 必须为整数类型并且 *mod* 必须不为零。如果给出 *mod* 并且 *exp* 为负值，则 *base* 必须相对于 *mod* 不可整除。在这种情况下，将会返回 `pow(inv_base, -exp, mod)`，其中 *inv_base* 为 *base* 的倒数对 *mod* 取余。

下面的例子是 `38` 的倒数对 `97` 取余:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

在 3.8 版本发生变更: 对于*int* 操作数，三参数形式的 `pow` 现在允许第二个参数为负值，即可以计算倒数的余数。

在 3.8 版本发生变更: 允许关键字参数。之前只支持位置参数。

print(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)

将 *objects* 打印输出至 *file* 指定的文本流，以 *sep* 分隔并在末尾加上 *end*。*sep* 、*end* 、*file* 和 *flush* 必须以关键字参数的形式给出。

所有非关键字参数都会被转换为字符串，就像是执行了*str()* 一样，并会被写入到流，以 *sep* 分隔并在末尾加上 *end*。*sep* 和 *end* 都必须为字符串；它们也可以为 `None`，这意味着使用默认值。如果没有给出 *objects*，则*print()* 将只写入 *end*。

file 参数必须是一个具有 `write(string)` 方法的对象；如果参数不存在或为 `None`，则将使用*sys.stdout*。由于要打印的参数会被转换为文本字符串，因此*print()* 不能用于二进制模式的文件对象。对于这些对象，应改用 `file.write(...)`。

输出缓冲通常由 *file* 确定。但是，如果 *flush* 为真值，流将被强制刷新。

在 3.3 版本发生变更: 增加了 *flush* 关键字参数。

class property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

返回 property 属性。

fget 是获取属性值的函数。*fset* 是用于设置属性值的函数。*fdel* 是用于删除属性值的函数。并且 *doc* 为属性对象创建文档字符串。

一个典型的用法是定义一个托管属性 `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

如果 *c* 为 *C* 的实例，`c.x` 将调用 getter，`c.x = value` 将调用 setter，`del c.x` 将调用 deleter。

如果给出，*doc* 将成为该 property 属性的文档字符串。否则该 property 将拷贝 *fget* 的文档字符串（如果存在）。这令使用*property()* 作为*decorator* 来创建只读的特征属性可以很容易地实现:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

`@property` 装饰器会将 `voltage()` 方法转化为一个具有相同名称的只读属性"getter"，并将 *voltage* 的文档字符串设为"Get the current voltage."

@getter

@setter

@deleter

特征属性对象具有 `getter`, `setter` 和 `deleter` 方法，它们可用作装饰器来创建该特征属性的副本，并将相应的访问函数设为所装饰的函数。这最好是用一个例子来说明:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

上述代码与第一个例子完全等价。注意一定要给附加函数与原始的特征属性相同的名称 (在本例中为 `x`。)

返回的特征属性对象同样具有与构造器参数相对应的属性 `fget`, `fset` 和 `fdel`。

在 3.5 版本发生变更: 特征属性对象的文档字符串现在是可写的。

__name__

保存特征属性名称的属性。特性属性名称可在运行时被修改。

Added in version 3.13.

class range(*stop*)

class range(*start*, *stop*, *step=1*)

虽然被称为函数，但*range* 实际上是一个不可变的序列类型，参见在*range 对象* 与*序列类型 --- list, tuple, range* 中的文档说明。

repr(*object*)

返回包含一个对象的可打印表示形式的字符串。对于许多类型而言，此函数会尝试返回一个具有与传给*eval()* 时相同的值的字符串；在其他情况下，其表示形式将为一个包含对象类型名称和通常包括对象名称和地址的额外信息的用尖括号括起来的字符串。一个类可以通过定义 `__repr__()` 方法来控制此函数为其实例所返回的内容。如果*sys.displayhook()* 不可访问，则此函数将会引发*RuntimeError*。

该类具有自定义的表示形式，它可被求值为:

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age

   def __repr__(self):
      return f"Person('{self.name}', {self.age})"
```

reversed(*seq*)

返回一个反向的*iterator*。*seq* 必须是一个具有 `__reversed__()` 方法或是支持序列协议（具有 `__len__()` 方法和从 `0` 开始的整数参数的 `__getitem__()` 方法）的对象。

round(*number*, *ndigits=None*)

返回 *number* 舍入到小数点后 *ndigits* 位精度的值。如果 *ndigits* 被省略或为 `None`，则返回最接近输入值的整数。

对于支持*round()* 方法的内置类型，结果值会舍入至最接近的 10 的负 *ndigits* 次幂的倍数；如果与两个倍数同样接近，则选用偶数。因此，`round(0.5)` 和 `round(-0.5)` 均得出 `0` 而 `round(1.5)` 则为 `2`。*ndigits* 可为任意整数值（正数、零或负数）。如果省略了 *ndigits* 或为 `None` ，则返回值将为整数。否则返回值与 *number* 的类型相同。

对于一般的 Python 对象 `number`, `round` 将委托给 `number.__round__`。

> **备注**
>
> 对浮点数执行*round()* 的行为可能会令人惊讶：例如，`round(2.675, 2)` 将给出 `2.67` 而不是期望的 `2.68`。这不算是程序错误：这一结果是由于大多数十进制小数实际上都不能以浮点数精确地表示。请参阅 tut-fp-issues 了解更多信息。

class set

class set(*iterable*)

返回一个新的*set* 对象，可以选择带有从 *iterable* 获取的元素。`set` 是一个内置类型。请查看*set* 和 集合类型 --- *set, frozenset* 获取关于这个类的文档。

有关其他容器请参看内置的*frozenset*, *list*, *tuple* 和*dict* 类，以及*collections* 模块。

setattr(*object*, *name*, *value*)

本函数与*getattr()* 相对应。其参数为一个对象、一个字符串和一个任意值。字符串可以为某现有属性的名称，或为新属性。只要对象允许，函数会将值赋给属性。如 `setattr(x, 'foobar', 123)` 等价于 `x.foobar = 123`。

name 无需为在 identifiers 中定义的 Python 标识符除非对象选择强制这样做，例如在一个自定义的 `__getattribute__()` 中或是通过 `__slots__`。一个名称不为标识符的属性将不可使用点号标记来访问，但是可以通过*getattr()* 等来访问。

> **备注**
>
> 由于 私有名称混合发生在编译时，因此必须手动混合私有属性（以两个下划线打头的属性）名称以便使用*setattr()* 来设置它。

class slice(*stop*)

class slice(*start*, *stop*, *step=None*)

返回一个表示由 `range(start, stop, step)` 指定的索引集的*slice* 对象。*start* 和 *step* 参数默认为 `None`。

start

stop

step

切片对象具有只读的数据属性 `start`, `stop` 和 `step`，它们将简单地返回相应的参数值（或其默认值）。它们没有其他显式的功能；但是，它们会被 NumPy 和其他第三方包所使用。

当使用扩展索引语法时也会生成切片对象。例如: `a[start:stop:step]` 或 `a[start:stop, i]`。请参阅*itertools.islice()* 了解返回*iterator* 的替代版本。

在 3.12 版本发生变更: Slice 对象现在将为*hashable* (如果*start*, *stop* 和*step* 均为可哈希对象)。

sorted(*iterable*, */*, *, *key=None*, *reverse=False*)

根据 *iterable* 中的项返回一个新的已排序列表。

具有两个可选参数，它们都必须指定为关键字参数。

key 指定带有单个参数的函数，用于从 *iterable* 的每个元素中提取用于比较的键 (例如 `key=str.lower`)。默认值为 `None` (直接比较元素)。

reverse 为一个布尔值。如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

使用*functools.cmp_to_key()* 可将老式的 *cmp* 函数转换为 *key* 函数。

内置的*sorted()* 确保是稳定的。如果一个排序确保不会改变比较结果相等的元素的相对顺序就称其为稳定的 --- 这有利于进行多重排序（例如先按部门、再按薪级排序）。

排序算法只使用 < 在项目之间比较。虽然定义一个 `__lt__()` 方法就足以进行排序，但 **PEP 8** 建议实现所有六个 富比较。这将有助于避免在与其他排序工具（如*max()*）使用相同的数据时出现错误，这些工具依赖于不同的底层方法。实现所有六个比较也有助于避免混合类型比较的混乱，因为混合类型比较可以调用反射到 `__gt__()` 的方法。

有关排序示例和简要排序教程，请参阅 sortinghowto 。

@staticmethod

将方法转换为静态方法。

静态方法不会接收隐式的第一个参数。要声明一个静态方法，请使用此语法

```
class C:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

`@staticmethod` 这样的形式称为函数的*decorator* -- 详情参阅 function。

静态方式既可以在类上调用 (如 `C.f()`)，也可以在实例上调用 (如 `C().f()`)。此外，静态方法*descriptor* 也属于可调用对象，因而它们可以在类定义中使用 (如 `f()`)。

Python 的静态方法与 Java 或 C++ 类似。另请参阅*classmethod()* ，可用于创建另一种类构造函数。

像所有装饰器一样，也可以像常规函数一样调用 `staticmethod` ，并对其结果执行某些操作。比如某些情况下需要从类主体引用函数并且您希望避免自动转换为实例方法。对于这些情况，请使用此语法:

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

想了解更多有关静态方法的信息，请参阅 types 。

在 3.10 版本发生变更: 静态方法现在继承了方法的属性 (`__module__`, `__name__`, `__qualname__`, `__doc__` 和 `__annotations__`)，并具有新的 `__wrapped__` 属性，现在是属于与常规函数类似的可调用对象。

class str(*object=''*)

class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

返回一个*str* 版本的 *object* 。有关详细信息，请参阅*str()* 。

`str` 是内置字符串*class* 。更多关于字符串的信息查看文本序列类型 --- *str*。

sum(*iterable*, */*, *start=0*)

从 *start* 开始自左向右对 *iterable* 的项求和并返回总计值。*iterable* 的项通常为数字，而 start 值则不允许为字符串。

对于某些用例，存在*sum()* 的更好替代。拼接字符串序列的更好、更快的方式是调用 `''.join(sequence)`。要以扩展的精度执行浮点数值的求和，请参阅*math.fsum()*。要拼接一系列可迭代对象，请考虑使用*itertools.chain()*。

在 3.8 版本发生变更: *start* 形参可用关键字参数形式来指定。

在 3.12 版本发生变更: 浮点数的求和已切换为一种可在大多数构建版本中给出更高精确度和更好适应性的算法。

class super

class super(*type*, *object_or_type=None*)

返回一个代理对象，它会将方法调用委托给 *type* 的父类或兄弟类。这对于访问已在类中被重写的继承方法很有用。

object_or_type 确定要用于搜索的*method resolution order*。搜索会从 *type* 之后的类开始。

举例来说，如果 *object_or_type* 的 `__mro__` 为 `D -> B -> C -> A -> object` 并且 *type* 的值为 `B`，则*super()* 将会搜索 `C -> A -> object`。

对应于 *object_or_type* 的类的 `__mro__` 属性列出了*getattr()* 和*super()* 所共同使用的方法解析搜索顺序。该属性是动态的并可在任何继承层级结构更新时被改变。

如果省略第二个参数，则返回的超类对象是未绑定的。如果第二个参数为一个对象，则 `isinstance(obj, type)` 必须为真值。如果第二个参数为一个类型，则 `issubclass(type2, type)` 必须为真值（这适用于类方法）。

当在普通方法或类中直接调用时，这两个参数均可被省略 (即" 零参数 `super()`")。在此情况下，*type* 将为其外层的类，而 *obj* 将为其所在函数的第一个参数 (通常为 `self`)。(这意味着零参数 `super()` 在嵌套的函数内的行为将不会如预期那样，这也包括生成器表达式，因为它会隐式地创建嵌套的函数。)

super 有两个典型用例。在具有单继承的类层级结构中，*super* 可用来引用父类而不必显式地指定它们的名称，从而令代码更易维护。这种用法与其他编程语言中 *super* 的用法非常相似。

第二个用例是在动态执行环境中支持协作多重继承。此用例为 Python 所独有而不存在于静态编码语言或仅支持单继承的语言当中。这使用实现 "菱形图" 成为可能，即有多个基类实现相同的方法。好的设计强制要求这样的方法在每个情况下都具有相同的调用签名（因为调用顺序是在运行时确定的，也因为这个顺序要适应类层级结构的更改，还因为这个顺序可能包括在运行时之前未知的兄弟类）。

对于以上两个用例，典型的超类调用看起来是这样的:

```
class C(B):
    def method(self, arg):
        super().method(arg)    # 它的作用像：
                               # super(C, self).method(arg)
```

除了方法查找之外，*super()* 也可用于属性查找。一个可能的应用场合是在上级或同级类中调用描述器。

请注意*super()* 被实现为为显式的带点号属性查找的绑定过程的组成部分，例如 `super().__getitem__(name)`。它做到这一点是通过实现自己的 `__getattribute__()` 方法以便能够按支持协作多重继承的可预测的顺序来搜索类。相应地，*super()* 在像 `super()[name]` 这样使用语句或运算符进行隐式查找时则是未定义的。

还要注意的是，除了零个参数的形式以外，*super()* 并不限于在方法内部使用。两个参数的形式明确指定参数并进行相应的引用。零个参数的形式仅适用于类定义内部，因为编译器需要填入必要的细节以正确地检索到被定义的类，还需要让普通方法访问当前实例。

对于有关如何使用*super()* 来如何设计协作类的实用建议，请参阅 使用 super() 的指南。

class tuple

class tuple(*iterable*)

虽然被称为函数，但*tuple* 实际上是一个不可变的序列类型，参见在元组 与*序列类型 --- list, tuple, range* 中的文档说明。

class type(*object*)

class type(*name*, *bases*, *dict*, ***kwds*)

传入一个参数时，返回 *object* 的类型。返回值是一个 type 对象并且通常与 `object.__class__` 所返回的对象相同。

推荐使用*isinstance()* 内置函数来检测对象的类型，因为它会考虑子类的情况。

传入三个参数时，返回一个新的 type 对象。这在本质上是 `class` 语句的一种动态形式。*name* 字符串即类名并会成为 `__name__` 属性；*bases* 元组包含基类并会成为 `__bases__` 属性；如果为空，则会添加所有类的终极基类，即*object*。*dict* 字典包含类体的属性和方法定义；它在成为 `__dict__` 属性之前可能会被拷贝或包装。下面两条语句会创建同样的 `type` 对象:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (), dict(a=1))
```

另请参阅：

- 有关类的属性和方法的文档。
- 类型对象

提供给三参数形式的关键字参数会被传递给适当的元类机制 (通常为 `__init_subclass__()`)，相当于类定义中关键字 (除了 *metaclass*) 的行为方式。

另请参阅 class-customization。

在 3.6 版本发生变更: `type` 的子类如果未重写 `type.__new__` 将不再能使用一个参数的形式来获取对象的类型。

vars()

vars(*object*)

返回模块、类、实例或任何其他具有 `__dict__` 属性的对象的 `__dict__` 属性。

模块和实例这样的对象具有可更新的 `__dict__` 属性；但是，其他对象的 `__dict__` 属性可能会设置写入限制（例如，类会使用*types.MappingProxyType* 来防止直接更新字典）。

不带参数时，*vars()* 的行为将类似于*locals()*。

如果指定了一个对象但它没有 `__dict__` 属性（例如，当它所属的类定义了 `__slots__` 属性时）则会引发*TypeError* 异常。

在 3.13 版本发生变更: 不带参数调用此函数的结果已被更新为与*locals()* 内置函数的描述类似。

zip(**iterables*, *strict=False*)

在多个迭代器上并行迭代，从每个迭代器返回一个数据项组成元组。

示例:

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

更正式的说法：*zip()* 返回元组的迭代器，其中第 *i* 个元组包含的是每个参数迭代器的第 *i* 个元素。

不妨换一种方式认识*zip()*：它会把行变成列，把列变成行。这类似于 矩阵转置 。

zip() 是延迟执行的：直至迭代时才会对元素进行处理，比如 `for` 循环或放入*list* 中。

值得考虑的是，传给*zip()* 的可迭代对象可能长度不同；有时是有意为之，有时是因为准备这些对象的代码存在错误。Python 提供了三种不同的处理方案：

- 默认情况下，*zip()* 在最短的迭代完成后停止。较长可迭代对象中的剩余项将被忽略，结果会裁切至最短可迭代对象的长度：

  ```
  >>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
  [(0, 'fee'), (1, 'fi'), (2, 'fo')]
  ```

- 通常*zip()* 用于可迭代对象等长的情况下。这时建议用 `strict=True` 的选项。输出与普通的*zip()* 相同:。

  ```
  >>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
  [('a', 1), ('b', 2), ('c', 3)]
  ```

  与默认行为不同，如果一个可迭代对象在其他几个之前被耗尽则会引发*ValueError*:

```
>>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'], strict=True):
...     print(item)
...
(0, 'fee')
(1, 'fi')
(2, 'fo')
Traceback (most recent call last):
  ...
ValueError: zip() argument 2 is longer than argument 1
```

如果未指定 `strict=True` 参数，所有导致可迭代对象长度不同的错误都会被抑制，这可能会在程序的其他地方表现为难以发现的错误。

- 为了让所有的可迭代对象具有相同的长度，长度较短的可用常量进行填充。这可由 *itertools.zip_longest()* 来完成。

极端例子是只有一个可迭代对象参数，*zip()* 会返回一个一元组的迭代器。如果未给出参数，则返回一个空的迭代器。

小技巧：

- 可确保迭代器的求值顺序是从左到右的。这样就能用 `zip(*[iter(s)]*n, strict=True)` 将数据列表按长度 n 进行分组。这将重复 *相同的* 迭代器 `n` 次，输出的每个元组都包含 `n` 次调用迭代器的结果。这样做的效果是把输入拆分为长度为 n 的块。
- *zip()* 与 `*` 运算符相结合可以用来拆解一个列表：

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> list(zip(x, y))
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

在 3.10 版本发生变更: 增加了 `strict` 参数。

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

> **备注**
>
> 与 *importlib.import_module()* 不同，这是一个日常 Python 编程中不需要用到的高级函数。

此函数会由 `import` 语句唤起。它可以被替换 (通过导入 *builtins* 模块并赋值给 `builtins.__import__`) 以便修改 `import` 语句的语义，但是 **强烈**不建议这样做，因为使用导入钩子 (参见 **PEP 302**) 通常更容易实现同样的目标，并且不会导致代码问题，因为许多代码都会假定所用的是默认实现。同样也不建议直接使用 *__import__()* 而应该用 *importlib.import_module()*。

本函数会导入模块 *name*，利用 *globals* 和 *locals* 来决定如何在包的上下文中解释该名称。*fromlist* 给出了应从 *name* 模块中导入的对象或子模块的名称。标准的实现代码完全不会用到 *locals* 参数，只用到了 *globals* 用于确定 `import` 语句所在的包上下文。

level 指定是使用绝对还是相对导入。`0` (默认值) 意味着仅执行绝对导入。*level* 为正数值表示相对于模块调用 *__import__()* 的目录，将要搜索的父目录层数 (详情参见 **PEP 328**)。

当 *name* 变量的形式为 `package.module` 时，通常将会返回最高层级的包（第一个点号之前的名称），而 不是 以 *name* 命名的模块。但是，当给出了非空的 *fromlist* 参数时，则将返回以 *name* 命名的模块。

例如，语句 `import spam` 的结果将为与以下代码作用相同的字节码：

```
spam = __import__('spam', globals(), locals(), [], 0)
```

语句 `import spam.ham` 的结果将为以下调用:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

请注意在这里*__import__()* 是如何返回顶层模块的，因为这是通过 `import` 语句被绑定到特定名称的对象。

另一方面，语句 `from spam.ham import eggs, sausage as saus` 的结果将为

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

在这里，`spam.ham` 模块会由*__import__()* 返回。要导入的对象将从此对象中提取并赋值给它们对应的名称。

如果您只想按名称导入模块（可能在包中），请使用*importlib.import_module()*

在 3.3 版本发生变更: *level* 的值不再支持负数（默认值也修改为 0）。

在 3.9 版本发生变更: 当使用了命令行参数 `-E` 或 `-I` 时，环境变量 `PYTHONCASEOK` 现在将被忽略。

备注

CHAPTER 3

内置常量

有少数的常量存在于内置命名空间中。它们是:

False

bool 类型的假值。给 False 赋值是非法的并会引发*SyntaxError*。

True

bool 类型的真值。给 True 赋值是非法的并会引发*SyntaxError*。

None

通常被用来代表空值的对象，例如未向某个函数传入默认参数时。向 None 赋值是非法的并会引发*SyntaxError*。None 是*NoneType* 类型的唯一实例。

NotImplemented

一个应当由双目运算特殊方法（如 __eq__(), __lt__(), __add__(), __rsub__() 等）返回的特殊值，用来表明该运算没有针对其他类型的实现；也可由原地双目运算特殊方法（如 __imul__(), __iand__() 等）出于同样的目的而返回。它不应在布尔上下文中被求值。NotImplemented 是*types.NotImplementedType* 类型的唯一实例。

> **备注**
>
> 当一个双目（或原地）方法返回 NotImplemented 时解释器将尝试对另一种类型（或其他回退操作，具体取决于所用的运算符）的反射操作。如果所有尝试都返回 NotImplemented，解释器将引发适当的异常。错误地返回 NotImplemented 将导致误导性的错误消息或 NotImplemented 值被返回给 Python 代码。
>
> 参见实现算术运算 为例。

> **小心**
>
> NotImplemented 和 NotImplementedError 不能互相替代。此常量应当仅以上文所描述的方式使用；请参阅*NotImplementedError* 了解正确使用该异常的相关细节。

在 3.9 版本发生变更: 在布尔上下文件中对 NotImplemented 求值的操作已被弃用。虽然它目前会被求解为真值，但将同时发出*DeprecationWarning*。它将在未来的 Python 版本中引发*TypeError*。

Ellipsis

与省略号字面值"..." 相同。该特殊值主要是与用户定义的容器数据类型的扩展切片语法结合使用。`Ellipsis` 是*types.EllipsisType* 类型的唯一实例。

__debug__

如果 Python 没有以 `-O` 选项启动，则此常量为真值。另请参见 `assert` 语句。

备注

变量名*None*，*False*，*True* 和 `__debug__` 无法重新赋值（赋值给它们，即使是属性名，将引发*SyntaxError*），所以它们可以被认为是"真正的"常数。

3.1 由 site 模块添加的常量

site 模块（在启动期间自动导入，除非给出 `-S` 命令行选项）将几个常量添加到内置命名空间。它们对交互式解释器 shell 很有用，并且不应在程序中使用。

quit(*code=None*)

exit(*code=None*)

当打印此对象时，会打印出一条消息，例如 "Use quit() or Ctrl-D (i.e. EOF) to exit"，当调用此对象时，将使用指定的退出代码来引发*SystemExit*。

help

该对象被打印时，将打印消息"Type help() for interactive help, or help(object) for help about object."，并且当被调用时，其行为将如别处 所描述的一样。

copyright

credits

打印或调用的对象分别打印版权或作者的文本。

license

当打印此对象时，会打印出一条消息 "Type license() to see the full license text"，当调用此对象时，将以分页形式显示完整的许可证文本（每次显示一屏）。

CHAPTER 4

内置类型

以下部分描述了解释器中内置的标准类型。

主要内置类型有数字、序列、映射、类、实例和异常。

有些多项集类是可变的。它们用于添加、移除或重排其成员的方法将原地执行，并不返回特定的项，绝对不会返回多项集实例自身而是返回 None。

有些操作受多种对象类型的支持；特别地，实际上所有对象都可以比较是否相等、检测逻辑值，以及转换为字符串（使用*repr()* 函数或略有差异的*str()* 函数）。后一个函数是在对象由*print()* 函数输出时被隐式地调用的。

4.1 逻辑值检测

任何对象都可以进行逻辑值的检测，以便在 if 或 while 作为条件或是作为下文所述布尔运算的操作数来使用。

在默认情况下，一个对象会被视为具有真值，除非其所属的类定义了在对象上调用时返回 False 的 __bool__() 方法或者返回零的 __len__() 方法。[1] 以下基本完整地列出了具有假值的内置对象：

- 被定义为假值的常量: None 和 False
- 任何数值类型的零: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- 空的序列和多项集: '', (), [], {}, set(), range(0)

产生布尔值结果的运算和内置函数总是返回 0 或 False 作为假值，1 或 True 作为真值，除非另行说明。（重要例外：布尔运算 or 和 and 总是返回其中一个操作数。）

4.2 布尔运算 --- and, or, not

这些属于布尔运算，按优先级升序排列:

运算	结果：	备注
x or y	如果 *x* 为真值，则 *x*，否则 *y*	(1)
x and y	if *x* is false, then *x*, else *y*	(2)
not x	if *x* is false, then True, else False	(3)

[1] 有关这些特殊方法的额外信息可参看 Python 参考指南 (customization)。

注释:

(1) 这是个短路运算符，因此只有在第一个参数为假值时才会对第二个参数求值。

(2) 这是个短路运算符，因此只有在第一个参数为真值时才会对第二个参数求值。

(3) `not` 的优先级比非布尔运算符低，因此 `not a == b` 会被解读为 `not (a == b)` 而 `a == not b` 会引发语法错误。

4.3 比较运算

在 Python 中有八种比较运算符。它们的优先级相同（比布尔运算的优先级高）。比较运算可以任意串连；例如，`x < y <= z` 等价于 `x < y and y <= z`，前者的不同之处在于 *y* 只被求值一次（但在两种情况下当 `x < y` 结果为假值时 *z* 都不会被求值）。

此表格汇总了比较运算:

运算	含意
`<`	严格小于
`<=`	小于或等于
`>`	严格大于
`>=`	大于或等于
`==`	等于
`!=`	不等于
`is`	对象标识
`is not`	否定的对象标识

除不同的数字类型外，不同类型的对象不能进行相等比较。`==` 运算符总有定义，但对于某些对象类型（例如，类对象），它等于 `is` 。其他 `<`、`<=`、`>` 和 `>=` 运算符仅在有意义的地方定义。例如，当参与比较的参数之一为复数时，它们会抛出 *TypeError* 异常。

具有不同标识的类的实例比较结果通常为不相等，除非类定义了 `__eq__()` 方法。

一个类的实例不能与相同类的其他实例或其他类型的对象进行排序，除非定义该类定义了足够多的方法，包括 `__lt__()`, `__le__()`, `__gt__()` 以及 `__ge__()` (而如果你想实现常规意义上的比较操作，通常只要有 `__lt__()` 和 `__eq__()` 就可以了)。

`is` 和 `is not` 运算符无法自定义；并且它们可以被应用于任意两个对象而不会引发异常。

还有两种具有相同语法优先级的运算 `in` 和 `not in`，它们被*iterable* 或实现了 `__contains__()` 方法的类型所支持。

4.4 数字类型 --- int, float, complex

存在三种不同的数字类型: *整数*, *浮点数* 和 *复数*。此外，布尔值属于整数的子类型。整数具有无限的精度。浮点数通常使用 C 中的 `double` 来实现；有关你的程序运行所在机器上浮点数的精度和内部表示法可在 *sys.float_info* 中查看。复数包含实部和虚部，分别以一个浮点数表示。要从一个复数 *z* 中提取这两个部分，可使用 `z.real` 和 `z.imag`。(标准库包含附加的数字类型，如表示有理数的 *fractions.Fraction* 以及以用户定制精度表示浮点数的 *decimal.Decimal*。)

数字是由数字字面值或内置函数与运算符的结果来创建的。不带修饰的整数字面值（包括十六进制、八进制和二进制数）会生成整数。包含小数点或幂运算符的数字字面值会生成浮点数。在数字字面值末尾加上 `'j'` 或 `'J'` 会生成虚数（实部为零的复数），你可以将其与整数或浮点数相加来得到具有实部和虚部的复数。

Python 完全支持混合运算：当一个二元算术运算符的操作数有不同数值类型时，" 较窄" 类型的操作数会拓宽到另一个操作数的类型，其中整数比浮点数窄，浮点数比复数窄。不同类型的数字之间的比较，同比较这些数字的精确值一样。[2]

[2] 作为结果，列表 `[1, 2]` 与 `[1.0, 2.0]` 是相等的，元组的情况也类似。

构造函数*int()*、*float()* 和*complex()* 可以用来构造特定类型的数字。

所有数字类型（复数除外）都支持下列运算（有关运算优先级，请参阅：operator-summary）：

运算	结果：	备注	完整文档
`x + y`	*x* 和 *y* 的和		
`x - y`	*x* 和 *y* 的差		
`x * y`	*x* 和 *y* 的乘积		
`x / y`	*x* 和 *y* 的商		
`x // y`	*x* 和 *y* 的商数	(1)(2)	
`x % y`	`x / y` 的余数	(2)	
`-x`	*x* 取反		
`+x`	*x* 不变		
`abs(x)`	*x* 的绝对值或大小		*abs()*
`int(x)`	将 *x* 转换为整数	(3)(6)	*int()*
`float(x)`	将 *x* 转换为浮点数	(4)(6)	*float()*
`complex(re, im)`	一个带有实部 *re* 和虚部 *im* 的复数。*im* 默认为 0。	(6)	*complex()*
`c.conjugate()`	复数 *c* 的共轭		
`divmod(x, y)`	`(x // y, x % y)`	(2)	*divmod()*
`pow(x, y)`	*x* 的 *y* 次幂	(5)	*pow()*
`x ** y`	*x* 的 *y* 次幂	(5)	

注释：

(1) 也称为整数除法。对于*int* 类型的操作数，结果的类型为*int*。对于*float* 类型的操作数，结果的类型为*float*。总的说来，结果是一个整数，但结果的类型不一定为*int*。结果总是向负无穷的方向舍入：`1//2` 为 “0“，`(-1)//2` 为 `-1`，`1//(-2)` 为 `-1`，`(-1)//(-2)` 为 `0`。

(2) 不可用于复数。而应在适当条件下使用*abs()* 转换为浮点数。

(3) 从*float* 转换为*int* 将会执行截断，丢弃掉小数部分。请参阅*math.floor()* 和*math.ceil()* 函数了解替代的转换方式。

(4) float 也接受字符串”nan” 和附带可选前缀”+” 或”-” 的”inf” 分别表示非数字 (NaN) 以及正或负无穷。

(5) Python 将 `pow(0, 0)` 和 `0 ** 0` 定义为 `1`，这是编程语言的普遍做法。

(6) 接受的数字字面值包括数码 `0` 到 `9` 或任何等效的 Unicode 字符（具有 `Nd` 特征属性的代码点）。

请参阅 Unicode 标准 了解具有 `Nd` 特征属性的码位完整列表。

所有*numbers.Real* 类型 (*int* 和*float*) 还包括下列运算:

运算	结果：
math.trunc(x)	*x* 截断为*Integral*
round(x[, n])	*x* 舍入到 *n* 位小数，半数值会舍入到偶数。如果省略 *n*，则默认为 0。
math.floor(x)	<= *x* 的最大*Integral*
math.ceil(x)	>= *x* 的最小*Integral*

有关更多的数字运算请参阅*math* 和*cmath* 模块。

4.4.1 整数类型的按位运算

按位运算只对整数有意义。计算按位运算的结果，就相当于使用无穷多个二进制符号位对二的补码执行操作。

二进制按位运算的优先级全都低于数字运算，但又高于比较运算；一元运算 ~ 具有与其他一元算术运算 (+ and −) 相同的优先级。

此表格是以优先级升序排序的按位运算列表:

运算	结果:	备注
`x | y`	*x* 和 *y* 按位 或	(4)
`x ^ y`	*x* 和 *y* 按位 异或	(4)
`x & y`	*x* 和 *y* 按位 与	(4)
`x << n`	*x* 左移 *n* 位	(1)(2)
`x >> n`	*x* 右移 *n* 位	(1)(3)
`~x`	*x* 逐位取反	

注释:

(1) 负的移位数是非法的，会导致引发*ValueError*。

(2) 左移 *n* 位等价于乘以 `pow(2, n)` 。

(3) 右移 *n* 位等价于除以 `pow(2, n)` ，作向下取整除法。

(4) 使用带有至少一个额外符号扩展位的有限个二进制补码表示（有效位宽度为 `1 + max(x.bit_length(), y.bit_length())` 或以上）执行这些计算就足以获得相当于有无数个符号位时的同样结果。

4.4.2 整数类型的附加方法

int 类型实现了*numbers.Integral* *abstract base class*。此外，它还提供了其他几个方法:

`int.`**`bit_length`**`()`

返回以二进制表示一个整数所需要的位数，不包括符号位和前面的零:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

更准确地说，如果 `x` 非零，则 `x.bit_length()` 是使得 `2**(k-1) <= abs(x) < 2**k` 的唯一正整数 `k`。同样地，当 `abs(x)` 小到足以具有正确的舍入对数时，则 `k = 1 + int(log(abs(x), 2))`。如果 `x` 为零，则 `x.bit_length()` 返回 `0`。

等价于:

```
def bit_length(self):
    s = bin(self)       # 二进制表示形式:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # 移除开头的零和负号
    return len(s)       # len('100101') --> 6
```

Added in version 3.1.

`int.`**`bit_count`**`()`

返回整数的绝对值的二进制表示中 1 的个数。也被称为 population count。示例:

```
>>> n = 19
>>> bin(n)
'0b10011'
>>> n.bit_count()
3
>>> (-n).bit_count()
3
```

等价于:

```
def bit_count(self):
    return bin(self).count("1")
```

Added in version 3.10.

int.**to_bytes**(*length=1*, *byteorder='big'*, ***, *signed=False*)

返回表示一个整数的字节数组。

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

整数会使用 *length* 个字节来表示，默认为 1。如果整数不能用给定的字节数来表示则会引发*OverflowError*。

byteorder 参数确定用于表示整数的字节顺序，默认为 `"big"`。如果 *byteorder* 为 `"big"`，则最高位字节放在字节数组的开头。如果 *byteorder* 为 `"little"`，则最高位字节放在字节数组的末尾。

signed 参数确定是否使用二的补码来表示整数。如果 *signed* 为 `False` 并且给出的是负整数，则会引发*OverflowError*。*signed* 的默认值为 `False`。

默认值可用于方便地将整数转为一个单字节对象:

```
>>> (65).to_bytes()
b'A'
```

但是，当使用默认参数时，请不要试图转换大于 255 的值否则会引发*OverflowError*。

等价于:

```
def to_bytes(n, length=1, byteorder='big', signed=False):
    if byteorder == 'little':
        order = range(length)
    elif byteorder == 'big':
        order = reversed(range(length))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    return bytes((n >> i*8) & 0xff for i in order)
```

Added in version 3.2.

在 3.11 版本发生变更: 添加了 `length` 和 `byteorder` 的默认参数值。

classmethod int.**from_bytes**(*bytes*, *byteorder='big'*, ***, *signed=False*)

返回由给定字节数组所表示的整数。

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

bytes 参数必须为一个*bytes-like object* 或是生成字节的可迭代对象。

byteorder 参数确定用于表示整数的字节顺序，默认为 `"big"`。如果 *byteorder* 为 `"big"`，则最高位字节放在字节数组的开头。如果 *byteorder* 为 `"little"`，则最高位字节放在字节数组的末尾。要请求

主机系统上的原生字节顺序，请使用*sys.byteorder* 作为字节顺序值。

signed 参数指明是否使用二的补码来表示整数。

等价于:

```
def from_bytes(bytes, byteorder='big', signed=False):
    if byteorder == 'little':
        little_ordered = list(bytes)
    elif byteorder == 'big':
        little_ordered = list(reversed(bytes))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    n = sum(b << i*8 for i, b in enumerate(little_ordered))
    if signed and little_ordered and (little_ordered[-1] & 0x80):
        n -= 1 << 8*len(little_ordered)

    return n
```

Added in version 3.2.

在 3.11 版本发生变更: 添加了 `byteorder` 的默认参数值。

`int.`**`as_integer_ratio`**`()`

返回一对整数，其比率正好等于原整数并且分母为正数。整数的比率总是用这个整数本身作为分子并以 `1` 作为分母。

Added in version 3.8.

`int.`**`is_integer`**`()`

返回 `True`。存在于兼容*float.is_integer()* 的鸭子类型。

Added in version 3.12.

4.4.3 浮点类型的附加方法

float 类型实现了*numbers.Real* *abstract base class*。float 还具有以下附加方法。

`float.`**`as_integer_ratio`**`()`

返回一对整数，其比率正好等于原浮点数。该比率为最简形式且分母为正值。无穷大会引发*OverflowError* 而 NaN 则会引发*ValueError*。

`float.`**`is_integer`**`()`

如果 float 实例可用有限位整数表示则返回 `True`，否则返回 `False`:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

两个方法均支持与十六进制数字符串之间的转换。由于 Python 浮点数在内部存储为二进制数，因此浮点数与十进制数字符串之间的转换往往会导致微小的舍入错误。而十六进制数字符串却允许精确地表示和描述浮点数。这在进行调试和数值工作时非常有用。

`float.`**`hex`**`()`

以十六进制字符串的形式返回一个浮点数表示。对于有限浮点数，这种表示法将总是包含前导的 `0x` 和尾随的 `p` 加指数。

`classmethod` `float.`**`fromhex`**`(`*s*`)`

返回以十六进制字符串 *s* 表示的浮点数的类方法。字符串 *s* 可以带有前导和尾随的空格。

请注意*float.hex()* 是实例方法，而*float.fromhex()* 是类方法。

十六进制字符串采用的形式为:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

可选的 sign 可以是 + 或 -，integer 和 fraction 是十六进制数码组成的字符串，exponent 是带有可选前导符的十进制整数。大小写没有影响，在 integer 或 fraction 中必须至少有一个十六进制数码。此语法类似于 C99 标准的 6.4.4.2 小节中所描述的语法，也是 Java 1.5 以上所使用的语法。特别地，*float.hex()* 的输出可以用作 C 或 Java 代码中的十六进制浮点数字面值，而由 C 的 %a 格式字符或 Java 的 Double.toHexString 所生成的十六进制数字符串由为*float.fromhex()* 所接受。

请注意 exponent 是十进制数而非十六进制数，它给出要与系数相乘的 2 的幂次。例如，十六进制数字符串 0x3.a7p10 表示浮点数 (3 + 10./16 + 7./16**2) * 2.0**10 即 3740.0:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

对 3740.0 应用反向转换会得到另一个代表相同数值的十六进制数字符串:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

4.4.4 数字类型的哈希运算

对于可能为不同类型的数字 x 和 y，要求当 x == y 时必定有 hash(x) == hash(y) (详情参见 __hash__() 方法的文档)。为了便于在各种数字类型 (包括*int*, *float*, *decimal.Decimal* 和*fractions.Fraction*) 上实现并保证效率，Python 对数字类型的哈希运算是基于为任意有理数定义统一的数学函数，因此该运算对*int* 和*fractions.Fraction* 的全部实例，以及*float* 和*decimal.Decimal* 的全部有限实例均可用。从本质上说，此函数是通过以一个固定质数 P 进行 P 降模给出的。P 的值在 Python 中可以*sys.hash_info* 的*modulus* 属性的形式被访问。

目前所用的质数设定，在 C long 为 32 位的机器上 P = 2**31 - 1 而在 C long 为 64 位的机器上 P = 2**61 - 1。

详细规则如下所述:

- 如果 x = m / n 是一个非负的有理数且 n 不可被 P 整除，则定义 hash(x) 为 m * invmod(n, P) % P，其中 invmod(n, P) 是对 n 模 P 取反。
- 如果 x = m / n 是一个非负的有理数且 n 可被 P 整除（但 m 不能）则 n 不能对 P 降模，以上规则不适用；在此情况下则定义 hash(x) 为常数值 sys.hash_info.inf。
- 如果 x = m / n 是一个负的有理数则定义 hash(x) 为 -hash(-x)。如果结果哈希值为 -1 则将其替换为 -2。
- 特殊值 sys.hash_info.inf 和 -sys.hash_info.inf 分别用于正无穷或负无穷的哈希值。
- 对于一个*complex* 值 z，会通过计算 hash(z.real) + sys.hash_info.imag * hash(z.imag) 将实部和虚部的哈希值结合起来，并进行降模 2**sys.hash_info.width 以使其处于 range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1)) 范围之内。同样地，如果结果为 -1 则将其替换为 -2。

为了阐明上述规则，这里有一些等价于内置哈希算法的 Python 代码示例，可用于计算有理数、*float* 或*complex* 的哈希值:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # 移除 P 的公因数。 (如果 m 和 n 互质则不需要。)
```

(续下页)

(接上页)

```
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # 费马小定理：pow(n, P-1, P) 等于 1，
        # 则 pow(n, P-2, P) 等于 n 除以 P 的余数的倒数。
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return object.__hash__(x)
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # 带正负号的约减求余运算 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

4.5 布尔类型 - bool

代表真值的布尔对象。*bool* 类型只有两个常量实例: `True` 和 `False`。

内置函数*bool()* 可将任意值转换为布尔值，如果该值可以被解读为逻辑值的话（参见上面的逻辑值检测小节）。

对于逻辑运算，请使用布尔运算符 `and`, `or` 和 `not`。当于两个布尔值应用按位运算符 `&`, `|`, `^` 时，它们将返回一个等价于逻辑运算" 与", " 或", " 异或" 的布尔值。但是，更推荐使用逻辑运算符 `and`, `or` 和 `!=` 而不是 `&`, `|` 和 `^`。

自 3.12 版本弃用: 使用按位取反运算符 ~ 已被弃用并将在 Python 3.16 中引发错误。

bool 是*int* 的子类 (参见*数字类型 --- int, float, complex*)。在许多数字场景下，`False` 和 `True` 的行为分别与整数 0 和 1 类似。但是，不建议这样使用；请使用*int()* 显式地执行转换。

4.6 迭代器类型

Python 支持在容器中进行迭代的概念。这是通过使用两个单独方法来实现的；它们被用于允许用户自定义类对迭代的支持。将在下文中详细描述的序列总是支持迭代方法。

容器对象要提供*iterable* 支持，必须定义一个方法:

container.**__iter__**()

返回一个*iterator* 对象。该对象需要支持下文所述的迭代器协议。如果容器支持不同的迭代类型，则可以提供额外的方法来专门地请求不同迭代类型的迭代器。（支持多种迭代形式的对象的例子有同时支持广度优先和深度优先遍历的树结果。）此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

迭代器对象自身需要支持以下两个方法，它们共同组成了 *迭代器协议*:

iterator.**__iter__**()

返回*iterator* 对象本身。这是同时允许容器和迭代器配合 `for` 和 `in` 语句使用所必须的。此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

iterator.**__next__**()

iterator 中返回下一项。如果已经没有可返回的项，则会引发*StopIteration* 异常。此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iternext` 槽位。

Python 定义了几种迭代器对象以支持对一般和特定序列类型、字典和其他更特别的形式进行迭代。除了迭代器协议的实现，特定类型的其他性质对迭代操作来说都不重要。

一旦迭代器的*__next__()* 方法引发了*StopIteration*，它必须一直对后续调用引发同样的异常。不遵循此行为特性的实现将无法正常使用。

4.6.1 生成器类型

Python 的*generator* 提供了一种实现迭代器协议的便捷方式。如果一个容器对象的 `__iter__()` 方法以生成器的形式实现，它将自动返回一个提供*__iter__()* 和 `__next__()` 方法的迭代器对象（从技术上说，是一个生成器对象）。有关生成器的更多信息可参阅 yield 表达式的文档。

4.7 序列类型 --- `list`, `tuple`, `range`

有三种基本序列类型：list, tuple 和 range 对象。为处理二进制数据 和文本字符串 而特别定制的附加序列类型会在专门的小节中描述。

4.7.1 通用序列操作

大多数序列类型，包括可变类型和不可变类型都支持下表中的操作。*collections.abc.Sequence* ABC 被提供用来更容易地在自定义序列类型上正确地实现这些操作。

此表按优先级升序列出了序列操作。在表格中，*s* 和 *t* 是具有相同类型的序列，*n*, *i*, *j* 和 *k* 是整数而 *x* 是任何满足 *s* 所规定的类型和值限制的任意对象。

`in` 和 `not in` 操作具有与比较操作相同的优先级。`+` (拼接) 和 `*` (重复) 操作具有与对应数值运算相同的优先级。[3]

运算	结果:	备注
`x in s`	如果 *s* 中的某项等于 *x* 则结果为 `True`，否则为 `False`	(1)
`x not in s`	如果 *s* 中的某项等于 *x* 则结果为 `False`，否则为 `True`	(1)
`s + t`	*s* 与 *t* 相拼接	(6)(7)
`s * n` 或 `n * s`	相当于 *s* 与自身进行 *n* 次拼接	(2)(7)
`s[i]`	*s* 的第 *i* 项，起始为 0	(3)
`s[i:j]`	*s* 从 *i* 到 *j* 的切片	(3)(4)
`s[i:j:k]`	*s* 从 *i* 到 *j* 步长为 *k* 的切片	(3)(5)
`len(s)`	*s* 的长度	
`min(s)`	*s* 的最小项	
`max(s)`	*s* 的最大项	
`s.index(x[, i[, j]])`	*x* 在 *s* 中首次出现项的索引号（索引号在 *i* 或其后且在 *j* 之前）	(8)
`s.count(x)`	*x* 在 *s* 中出现的总次数	

[3] 必须如此，因为解析器无法判断操作数的类型。

相同类型的序列也支持比较。特别地，tuple 和 list 的比较是通过比较对应元素的字典顺序。这意味着想要比较结果相等，则每个元素比较结果都必须相等，并且两个序列长度必须相同。(完整细节请参阅语言参考的 comparisons 部分。)

可变序列的正向和逆向迭代器使用一个索引来访问值。即使底层序列被改变该索引也将持续向前（或向后）步进。迭代器只有在遇到*IndexError* 或 a *StopIteration* 时才会终结（或是当索引降至零以下）。

注释:

(1) 虽然 `in` 和 `not in` 操作在通常情况下仅被用于简单的成员检测，某些专门化序列 (例如*str*, *bytes* 和*bytearray*) 也使用它们进行子序列检测:

```
>>> "gg" in "eggs"
True
```

(2) 小于 `0` 的 *n* 值会被当作 `0` 来处理 (生成一个与 *s* 同类型的空序列)。请注意序列 *s* 中的项并不会被拷贝；它们会被多次引用。这一点经常会令 Python 编程新手感到困扰；例如:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

具体的原因在于 `[[]]` 是一个包含了一个空列表的单元素列表，所以 `[[]] * 3` 结果中的三个元素都是对这一个空列表的引用。修改 `lists` 中的任何一个元素实际上都是对这一个空列表的修改。你可以用以下方式创建以不同列表为元素的列表:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

进一步的解释可以在 FAQ 条目 faq-multidimensional-list 中查看。

(3) 如果 *i* 或 *j* 为负值，则索引顺序是相对于序列 *s* 的末尾: 索引号会被替换为 `len(s) + i` 或 `len(s) + j`。但要注意 `-0` 仍然为 `0`。

(4) *s* 从 *i* 到 *j* 的切片被定义为所有满足 `i <= k < j` 的索引号 *k* 的项组成的序列。如果 *i* 或 *j* 大于 `len(s)`，则使用 `len(s)`。如果 *i* 被省略或为 `None`，则使用 `0`。如果 *j* 被省略或为 `None`，则使用 `len(s)`。如果 *i* 大于等于 *j*，则切片为空。

(5) *s* 从 *i* 到 *j* 步长为 *k* 的切片被定义为所有满足 `0 <= n < (j-i)/k` 的索引号 `x = i + n*k` 的项组成的序列。换句话说，索引号为 `i`, `i+k`, `i+2*k`, `i+3*k`，以此类推，当达到 *j* 时停止 (但一定不包括 *j*)。当 *k* 为正值时，*i* 和 *j* 会被减至不大于 `len(s)`。当 *k* 为负值时，*i* 和 *j* 会被减至不大于 `len(s) - 1`。如果 *i* 或 *j* 被省略或为 `None`，它们会成为“终止”值 (是哪一端的终止值则取决于 *k* 的符号)。请注意，*k* 不可为零。如果 *k* 为 `None`，则当作 `1` 处理。

(6) 拼接不可变序列总是会生成新的对象。这意味着通过重复拼接来构建序列的运行时开销将会基于序列总长度的乘方。想要获得线性的运行时开销，你必须改用下列替代方案之一:

- 如果拼接*str* 对象，你可以构建一个列表并在最后使用*str.join()* 或是写入一个*io.StringIO* 实例并在结束时获取它的值
- 如果拼接*bytes* 对象，你可以类似地使用*bytes.join()* 或*io.BytesIO*，或者你也可以使用*bytearray* 对象进行原地拼接。*bytearray* 对象是可变的，并且具有高效的重分配机制
- 如果拼接*tuple* 对象，请改为扩展*list* 类
- 对于其它类型，请查看相应的文档

(7) 某些序列类型 (例如*range*) 仅支持遵循特定模式的项序列，因此并不支持序列拼接或重复。

(8) 当 *x* 在 *s* 中找不到时 index 会引发*ValueError*。不是所有实现都支持传入额外参数 *i* 和 *j*。这两个参数允许高效地搜索序列的子序列。传入这两个额外参数大致相当于使用 `s[i:j].index(x)`，但是不会复制任何数据，并且返回的索引是相对于序列的开头而非切片的开头。

4.7.2 不可变序列类型

不可变序列类型普遍实现而可变序列类型未实现的唯一操作就是对*hash()* 内置函数的支持。

这种支持允许不可变类型，例如*tuple* 实例被用作*dict* 键，以及存储在*set* 和*frozenset* 实例中。

尝试对包含有不可哈希值的不可变序列进行哈希运算将会导致*TypeError*。

4.7.3 可变序列类型

以下表格中的操作是在可变序列类型上定义的。*collections.abc.MutableSequence* ABC 被提供用来更容易地在自定义序列类型上正确实现这些操作。

表格中的 *s* 是可变序列类型的实例，*t* 是任意可迭代对象，而 *x* 是符合对 *s* 所规定类型与值限制的任何对象 (例如，*bytearray* 仅接受满足 `0 <= x <= 255` 值限制的整数)。

运算	结果:	备注
`s[i] = x`	将 *s* 的第 *i* 项替换为 *x*	
`s[i:j] = t`	将 *s* 从 *i* 到 *j* 的切片替换为可迭代对象 *t* 的内容	
`del s[i:j]`	等同于 `s[i:j] = []`	
`s[i:j:k] = t`	将 `s[i:j:k]` 的元素替换为 *t* 的元素	(1)
`del s[i:j:k]`	从列表中移除 `s[i:j:k]` 的元素	
`s.append(x)`	将 *x* 添加到序列的末尾 (等同于 `s[len(s):len(s)] = [x]`)	
`s.clear()`	从 *s* 中移除所有项 (等同于 `del s[:]`)	(5)
`s.copy()`	创建 *s* 的浅拷贝 (等同于 `s[:]`)	(5)
`s.extend(t)` 或 `s += t`	用 *t* 的内容扩展 *s* (基本上等同于 `s[len(s):len(s)] = t`)	
`s *= n`	使用 *s* 的内容重复 *n* 次来对其进行更新	(6)
`s.insert(i, x)`	在由 *i* 给出的索引位置将 *x* 插入 *s* (等同于 `s[i:i] = [x]`)	
`s.pop()` 或 `s.pop(i)`	提取在 *i* 位置上的项，并将其从 *s* 中移除	(2)
`s.remove(x)`	从 *s* 中移除第一个 `s[i]` 等于 *x* 的条目	(3)
`s.reverse()`	就地将列表中的元素逆序。	(4)

注释:

(1) 如果 *k* 不等于 `1`，则 *t* 必须与它所替换的切片具有相同的长度。

(2) 可选参数 *i* 默认为 `-1`，因此在默认情况下会移除并返回最后一项。

(3) 当在 *s* 中找不到 *x* 时 `remove()` 操作会引发*ValueError*。

(4) 当反转大尺寸序列时 `reverse()` 方法会原地修改该序列以保证空间经济性。为提醒用户此操作是通过间接影响进行的，它并不会返回反转后的序列。

(5) 包括 `clear()` 和 `copy()` 是为了与不支持切片操作的可变容器 (例如*dict* 和*set*) 的接口保持一致。`copy()` 不是*collections.abc.MutableSequence* ABC 的一部分，但大多数具体的可变序列类都提供了它。

Added in version 3.3: `clear()` 和 `copy()` 方法。

(6) *n* 值为一个整数，或是一个实现了 `__index__()` 的对象。*n* 值为零或负数将清空序列。序列中的项不会被拷贝；它们会被多次引用，正如通用序列操作 中有关 `s * n` 的说明。

4.7.4 列表

列表是可变序列，通常用于存放同类项目的集合（其中精确的相似程度将根据应用而变化）。

class list([*iterable*])

可以用多种方式构建列表：

- 使用一对方括号来表示空列表: `[]`
- 使用方括号，其中的项以逗号分隔: `[a]`, `[a, b, c]`
- 使用列表推导式: `[x for x in iterable]`
- 使用类型的构造器: `list()` 或 `list(iterable)`

构造器将构造一个列表，其中的项与 *iterable* 中的项具有相同的的值与顺序。*iterable* 可以是序列、支持迭代的容器或其它可迭代对象。如果 *iterable* 已经是一个列表，将创建并返回其副本，类似于 `iterable[:]`。例如，`list('abc')` 返回 `['a', 'b', 'c']` 而 `list( (1, 2, 3) )` 返回 `[1, 2, 3]`。如果没有给出参数，构造器将创建一个空列表 `[]`。

其它许多操作也会产生列表，包括*sorted()* 内置函数。

列表实现了所有一般 和可变 序列的操作。列表还额外提供了以下方法：

sort(*, *key=None*, *reverse=False*)

此方法会对列表进行原地排序，只使用 < 来进行各项间比较。异常不会被屏蔽——如果有任何比较操作失败，整个排序操作将失败（而列表可能会处于被部分修改的状态）。

sort() 接受两个仅限以关键字形式传入的参数 (仅限关键字参数):

key 指定带有一个参数的函数，用于从每个列表元素中提取比较键 (例如 `key=str.lower`)。对应于列表中每一项的键会被计算一次，然后在整个排序过程中使用。默认值 `None` 表示直接对列表项排序而不计算一个单独的键值。

可以使用*functools.cmp_to_key()* 将 2.x 风格的 *cmp* 函数转换为 *key* 函数。

reverse 为一个布尔值。如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

当顺序大尺寸序列时此方法会原地修改该序列以保证空间经济性。为提醒用户此操作是通过间接影响进行的，它并不会返回排序后的序列（请使用*sorted()* 显示地请求一个新的已排序列表实例）。

sort() 方法确保是稳定的。如果一个排序确保不会改变比较结果相等的元素的相对顺序就称其为稳定的 --- 这有利于进行多重排序（例如先按部门、再按薪级排序）。

有关排序示例和简要排序教程，请参阅 sortinghowto 。

在一个列表被排序期间，尝试改变甚至进行检测也会造成未定义的影响。Python 的 C 实现会在排序期间将列表显示为空，如果发现列表在排序期间被改变将会引发*ValueError*。

4.7.5 元组

元组是不可变序列，通常用于储存异构数据的多项集（例如由*enumerate()* 内置函数所产生的二元组）。元组也被用于需要同构数据的不可变序列的情况（例如允许存储到*set* 或*dict* 的实例）。

class tuple([*iterable*])

可以用多种方式构建元组：

- 使用一对圆括号来表示空元组: `()`
- 使用一个后缀的逗号来表示单元组: `a,` 或 `(a,)`
- 使用以逗号分隔的多个项: `a, b, c` or `(a, b, c)`
- 使用内置的*tuple()*: `tuple()` 或 `tuple(iterable)`

构造器将构造一个元组，其中的项与 *iterable* 中的项具有相同的值与顺序。*iterable* 可以是序列、支持迭代的容器或其他可迭代对象。如果 *iterable* 已经是一个元组，会不加改变地将其返回。例如，`tuple('abc')` 返回 `('a', 'b', 'c')` 而 `tuple( [1, 2, 3] )` 返回 `(1, 2, 3)`。如果没有给出参数，构造器将创建一个空元组 `()`。

请注意决定生成元组的其实是逗号而不是圆括号。圆括号只是可选的，生成空元组或需要避免语法歧义的情况除外。例如，`f(a, b, c)` 是在调用函数时附带三个参数，而 `f((a, b, c))` 则是在调用函数时附带一个三元组。

元组实现了所有一般 序列的操作。

对于通过名称访问相比通过索引访问更清晰的异构数据多项集，*collections.namedtuple()* 可能是比简单元组对象更为合适的选择。

4.7.6 range 对象

range 类型表示不可变的数字序列，通常用于在 `for` 循环中循环指定的次数。

class range(*stop*)

class range(*start*, *stop*[, *step*])

range 构造器的参数必须为整数（可以是内置的*int* 或任何实现了 `__index__()` 特殊方法的对象）。如果省略 *step* 参数，则默认为 `1`。如果省略 *start* 参数，则默认为 `0`。如果 *step* 为零，则会引发*ValueError*。

如果 *step* 为正值，确定 range `r` 内容的公式为 `r[i] = start + step*i` 其中 `i >= 0` 且 `r[i] < stop`。

如果 *step* 为负值，确定 range 内容的公式仍然为 `r[i] = start + step*i`，但限制条件改为 `i >= 0` 且 `r[i] > stop`.

如果 `r[0]` 不符合值的限制条件，则该 range 对象为空。range 对象确实支持负索引，但是会将其解读为从正索引所确定的序列的末尾开始索引。

元素绝对值大于*sys.maxsize* 的 range 对象是被允许的，但某些特性 (例如*len()*) 可能引发*OverflowError*。

一些 range 对象的例子:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

range 对象实现了一般 序列的所有操作，但拼接和重复除外（这是由于 range 对象只能表示符合严格模式的序列，而重复和拼接通常都会违反这样的模式）。

start

start 形参的值 (如果该形参未提供则为 `0`)

stop

stop 形参的值

step

step 形参的值 (如果该形参未提供则为 `1`)

range 类型相比常规*list* 或*tuple* 的优势在于一个*range* 对象总是占用固定数量的（较小）内存，不论其所表示的范围有多大（因为它只保存了 `start`, `stop` 和 `step` 值，并会根据需要计算具体单项或子范围的值）。

range 对象实现了*collections.abc.Sequence* ABC，提供如包含检测、元素索引查找、切片等特性，并支持负索引 (参见*序列类型 --- list, tuple, range*):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

使用 == 和 != 检测 range 对象是否相等是将其作为序列来比较。也就是说，如果两个 range 对象表示相同的值序列就认为它们是相等的。(请注意比较结果相等的两个 range 对象可能会具有不同的*start*, *stop* 和*step* 属性，例如 `range(0) == range(2, 1, 3)` 而 `range(0, 3, 2) == range(0, 4, 2)`。)

在 3.2 版本发生变更: 实现 Sequence ABC。支持切片和负数索引。使用*int* 对象在固定时间内进行成员检测，而不是逐一迭代所有项。

在 3.3 版本发生变更: 定义'==' 和'!=' 以根据 range 对象所定义的值序列来进行比较（而不是根据对象的标识）。

增加了*start*, *stop* 和*step* 属性。

参见

- linspace recipe 演示了如何实现一个惰性求值版本的适合浮点数应用的 range 对象。

4.8 文本序列类型 --- `str`

在 Python 中处理文本数据是使用*str* 对象，也称为 *字符串*。字符串是由 Unicode 码位构成的不可变*序列*。字符串字面值有多种不同的写法:

- 单引号: `'允许包含有" 双" 引号'`
- 双引号: `" 允许嵌入 '单' 引号"`
- 三重引号: `'''三重单引号''',""" 三重双引号"""`

使用三重引号的字符串可以跨越多行——其中所有的空白字符都将包含在该字符串字面值中。

作为单一表达式组成部分，之间只由空格分隔的多个字符串字面值会被隐式地转换为单个字符串字面值。也就是说，`("spam " "eggs") == "spam eggs"`。

请参阅 strings 了解有关各种字符串字面值形式的更多信息，包括所支持的 转义序列，以及禁用大多数转义序列处理的 `r` ("raw") 前缀。

字符串也可以通过使用*str* 构造器从其他对象创建。

由于不存在单独的“字符”类型，对字符串做索引操作将产生一个长度为 1 的字符串。也就是说，对于一个非空字符串 *s*, `s[0] == s[0:1]`。

不存在可变的字符串类型，但是*str.join()* 或*io.StringIO* 可以被被用来根据多个片段高效率地构建字符串。

在 3.3 版本发生变更: 为了与 Python 2 系列的向下兼容，再次允许字符串字面值使用 `u` 前缀。它对字符串字面值的含义没有影响，并且不能与 `r` 前缀同时出现。

class str(*object=''*)

class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

返回 *object* 的字符串 版本。如果未提供 *object* 则返回空字符串。在其他情况下 `str()` 的行为取决于 *encoding* 或 *errors* 是否有给出，具体见下。

如果 *encoding* 或 *errors* 均未给出，则 `str(object)` 将返回 `type(object).__str__(object)`，这是 *object* 的“非正式”而适合显示的字符串表示形式。对于字符串对象，这就是该字符串本身。如果 *object* 没有 `__str__()` 方法，则 *str()* 将回退为返回 *repr(object)*。

如果 *encoding* 或 *errors* 至少给出其中之一，则 *object* 应该是一个 *bytes-like object* (例如 *bytes* 或 *bytearray*)。在此情况下，如果 *object* 是一个 *bytes* (或 *bytearray*) 对象，则 `str(bytes, encoding, errors)` 等价于 *bytes.decode(encoding, errors)*。否则的话，会在调用 *bytes.decode()* 之前获取缓冲区对象下层的 bytes 对象。请参阅二进制序列类型 --- *bytes, bytearray, memoryview* 与 bufferobjects 了解有关缓冲区对象的信息。

将一个 *bytes* 对象传入 *str()* 而不给出 *encoding* 或 *errors* 参数的操作属于第一种情况，将返回非正式的字符串表示（另请参阅 Python 的 `-b` 命令行选项）。例如:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

有关 `str` 类及其方法的更多信息，请参阅下面的文本序列类型 --- *str* 和字符串的方法 小节。要输出格式化字符串，请参阅 f-strings 和格式字符串语法 小节。此外还可以参阅文本处理服务 小节。

4.8.1 字符串的方法

字符串实现了所有一般 序列的操作，还额外提供了以下列出的一些附加方法。

字符串还支持两种字符串格式化样式，一种提供了很大程度的灵活性和可定制性 (参阅 *str.format()*，格式字符串语法 和自定义字符串格式化) 而另一种是基于 C `printf` 样式的格式化，它可处理的类型范围较窄，并且更难以正确使用，但对于它可处理的情况往往会更为快速 (*printf* 风格的字符串格式化)。

标准库的文本处理服务 部分涵盖了许多其他模块，提供各种文本相关工具（例如包含于 *re* 模块中的正则表达式支持）。

`str.`**capitalize**()

返回原字符串的副本，其首个字符大写，其余为小写。

在 3.8 版本发生变更: 第一个字符现在被放入了 titlecase 而不是 uppercase。这意味着复合字母类字符将只有首个字母改为大写，而再不是全部字符大写。

`str.`**casefold**()

返回原字符串消除大小写的副本。消除大小写的字符串可用于忽略大小写的匹配。

消除大小写类似于转为小写，但是更加彻底一些，因为它会移除字符串中的所有大小写变化形式。例如，德语小写字母 `'ß'` 相当于 `"ss"`。由于它已经是小写了，*lower()* 不会对 `'ß'` 做任何改变；而 *casefold()* 则会将其转换为 `"ss"`。

大小写折叠算法在 ‘ Unicode 标准第 3.13 节’Default Case Folding’ 中描述 <https://www.unicode.org/versions/Unicode15.1.0/ch03.pdf>‘__。

Added in version 3.3.

`str.`**center**(*width*[, *fillchar*])

返回长度为 *width* 的字符串，原字符串在其正中。使用指定的 *fillchar* 填充两边的空位（默认使用 ASCII 空格符）。如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

`str.`**count**(*sub*[, *start*[, *end*]])

返回子字符串 *sub* 在 [*start*, *end*] 范围内非重叠出现的次数。可选参数 *start* 与 *end* 会被解读为切片表示法。

如果 *sub* 为空，则返回字符之间的空字符串数，即字符串的长度加一。

str.**encode**(*encoding='utf-8'*, *errors='strict'*)

返回编码为*bytes* 的字符串。

encoding 默认为 'utf-8'；请参阅标准编码 了解其他可能的值。

errors 控制如何处理编码错误。如为 'strict' (默认值)，则会引发*UnicodeError*。其他可能的值有 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' 以及通过*codecs.register_error()* 注册的任何其他名称。请参阅错误处理方案 了解详情。

出于性能原因，除非真正发生了编码错误，启用了*Python 开发模式* 或使用了 调试编译版否则不会检查 *errors* 值的有效性。

在 3.1 版本发生变更: 加入了对关键字参数的支持。

在 3.9 版本发生变更: 现在会在*Python 开发模式* 和 调试模式下检查 *errors* 参数的值。

str.**endswith**(*suffix*[, *start*[, *end*]])

如果字符串以指定的 *suffix* 结束返回 True，否则返回 False。*suffix* 也可以为由多个供查找的后缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

str.**expandtabs**(*tabsize=8*)

返回字符串的副本，其中所有的制表符会由一个或多个空格替换，具体取决于当前列位置和给定的制表符宽度。每 *tabsize* 个字符设为一个制表位（默认值 8 时设定的制表位在列 0, 8, 16 依次类推）。要展开字符串，当前列将被设为零并逐一检查字符串中的每个字符。如果字符为制表符 (\t)，则会在结果中插入一个或多个空格符，直到当前列等于下一个制表位。（制表符本身不会被复制。）如果字符为换行符 (\n) 或回车符 (\r)，它会被复制并将当前列重设为零。任何其他字符会被不加修改地复制并将当前列加一，不论该字符在被打印时会如何显示。

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

str.**find**(*sub*[, *start*[, *end*]])

返回子字符串 *sub* 在 s[start:end] 切片内被找到的最小索引。可选参数 *start* 与 *end* 会被解读为切片表示法。如果 *sub* 未被找到则返回 -1。

备注

find() 方法应该只在你需要知道 *sub* 所在位置时使用。要检查 *sub* 是否为子字符串，请使用 in 操作符:

```
>>> 'Py' in 'Python'
True
```

str.**format**(**args*, ***kwargs*)

执行字符串格式化操作。调用此方法的字符串可以包含字符串字面值或者以花括号 {} 括起来的替换域。每个替换域可以包含一个位置参数的数字索引，或者一个关键字参数的名称。返回的字符串副本中每个替换域都会被替换为对应参数的字符串值。

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

请参阅格式字符串语法 了解有关可以在格式字符串中指定的各种格式选项的说明。

备注

当使用 n 类型 (例如: '{:n}'.format(1234)) 来格式化数字 (*int*, *float*, *complex*, *decimal.Decimal* 及其子类) 的时候，该函数会临时性地将 LC_CTYPE 区域设置为 LC_NUMERIC 区域以解

码 localeconv() 的 decimal_point 和 thousands_sep 字段，如果它们是非 ASCII 字符或长度超过 1 字节的话，并且 LC_NUMERIC 区域会与 LC_CTYPE 区域不一致。这个临时更改会影响其他线程。

在 3.7 版本发生变更: 当使用 n 类型格式化数字时，该函数在某些情况下会临时性地将 LC_CTYPE 区域设置为 LC_NUMERIC 区域。

str.**format_map**(*mapping*, /)

类似于 str.format(**mapping)，不同之处在于 mapping 会被直接使用而不是复制到一个*dict*。适宜使用此方法的一个例子是当 mapping 为 dict 的子类的情况：

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Added in version 3.2.

str.**index**(*sub*[, *start*[, *end*]])

类似于*find()*，但在找不到子字符串时会引发*ValueError*。

str.**isalnum**()

如果字符串中的所有字符都是字母或数字且至少有一个字符，则返回 True，否则返回 False。如果 c.isalpha()，c.isdecimal()，c.isdigit()，或 c.isnumeric() 之中有一个返回 True，则字符 c 是字母或数字。

str.**isalpha**()

如果字符串中的所有字符都为字母类并且至少有一个字符则返回 True，否则返回 False。字母类字符是指在 Unicode 字符数据库中被定义为”Letter” 的字符，即通用类别属性为”Lm”, ”Lt”, ”Lu”, ”Ll” 或”Lo” 之一的字符。请注意这不同于 在 Unicode 标准 4.10 节’Letters, Alphabetic, and Ideographic’ 中定义的 Alphabetic 属性。

str.**isascii**()

如果字符串为空或字符串中的所有字符都是 ASCII，返回 True，否则返回 False。ASCII 字符的码点范围是 U+0000-U+007F。

Added in version 3.7.

str.**isdecimal**()

如果字符串中的所有字符都是十进制字符且该字符串至少有一个字符，则返回 True，否则返回 False。十进制字符指那些可以用来组成 10 进制数字的字符，例如 U+0660，即阿拉伯字母数字 0。严格地讲，十进制字符是 Unicode 通用类别”Nd” 中的一个字符。

str.**isdigit**()

如果字符串中的所有字符都是数字，并且至少有一个字符，返回 True，否则返回 False。数字包括十进制字符和需要特殊处理的数字，如兼容性上标数字。这包括了不能用来组成 10 进制数的数字，如 Kharosthi 数。严格地讲，数字是指属性值为 Numeric_Type=Digit 或 Numeric_Type=Decimal 的字符。

str.**isidentifier**()

如果字符串是有效的标识符，返回 True，依据语言定义，identifiers 节。

keyword.iskeyword() 可被用来测试字符串 s 是否为保留的标识符，如 def 和 class。

示例：

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
```

(续下页)

(接上页)

```
(True, False)
>>> 'def'.isidentifier(), iskeyword('def')
(True, True)
```

str.**islower**()

如果字符串中至少有一个区分大小写的字符[4] 且此类字符均为小写则返回 `True` ，否则返回 `False` 。

str.**isnumeric**()

如果字符串中至少有一个字符且所有字符均为数值字符则返回 `True` ，否则返回 `False` 。数值字符包括数字字符，以及所有在 Unicode 中设置了数值特性属性的字符，例如 U+2155, VULGAR FRACTION ONE FIFTH。正式的定义为：数值字符就是具有特征属性值 Numeric_Type=Digit, Numeric_Type=Decimal 或 Numeric_Type=Numeric 的字符。

str.**isprintable**()

如果字符串中的所有字符均为可打印字符则返回真值，如果包含至少一个不可打印字符则返回假值。

这里的“可打印”是指字符适用于在其输出中*repr()* 中；“不可打印”则意味着内置类型的*repr()* 将以十六进制转义代码表示该字符。它不会影响对写入到*sys.stdout* 或*sys.stderr* 的字符串的处理。

可打印字符就是在 Unicode 字符数据库 (参见*unicodedata*) 中分组为主类别 Letter, Mark, Number, Punctuation 或 Symbol (L, M, N, P 或 S) 的字符；加上 ASCII 空格符 0x20。不可打印字符就是分组为 Separator 或 Other (Z 或 C) 的字符，ASCII 空格符除外。

str.**isspace**()

如果字符串中只有空白字符且至少有一个字符则返回 `True` ，否则返回 `False` 。

空白字符是指在 Unicode 字符数据库 (参见*unicodedata*) 中主要类别为 `Zs` (”Separator, space”) 或所属双向类为 `WS`, `B` 或 `S` 的字符。

str.**istitle**()

如果字符串中至少有一个字符且为标题字符串则返回 `True` ，例如大写字符之后只能带非大写字符而小写字符必须有大写字符打头。否则返回 `False` 。

str.**isupper**()

如果字符串中至少有一个区分大小写的字符[4] 且此类字符均为大写则返回 `True` ，否则返回 `False` 。

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

str.**join**(*iterable*)

返回一个由 *iterable* 中的字符串拼接而成的字符串。如果 *iterable* 中存在任何非字符串值包括*bytes* 对象则会引发*TypeError*。调用该方法的字符串将作为元素之间的分隔。

str.**ljust**(*width*[, *fillchar*])

返回长度为 *width* 的字符串，原字符串在其中靠左对齐。使用指定的 *fillchar* 填充空位 (默认使用 ASCII 空格符)。如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

str.**lower**()

返回原字符串的副本，其所有区分大小写的字符[4] 均转换为小写。

所使用的小写算法 在 Unicode 标准 3.13 节’Default Case Folding’ 中描述。

[4] 区分大小写的字符是指所属一般类别属性为”Lu” (Letter, uppercase), ”Ll” (Letter, lowercase) 或”Lt” (Letter, titlecase) 之一的字符。

str.**lstrip**(*[chars]*)

返回原字符串的副本，移除其中的前导字符。*chars* 参数为指定要移除字符的字符串。如果省略或为 None，则 *chars* 参数默认移除空白符。实际上 *chars* 参数并非指定单个前缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

参见*str.removeprefix()*，该方法将删除单个前缀字符串，而不是全部给定集合中的字符。例如:

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

static str.**maketrans**(*x*[, *y*[, *z*]])

此静态方法返回一个可供*str.translate()*使用的转换对照表。

如果只有一个参数，则它必须是一个将 Unicode 码位序号（整数）或字符（长度为 1 的字符串）映射到 Unicode 码位序号、（任意长度的）字符串或 None 的字典。字符键将会被转换为码位序号。

如果有两个参数，则它们必须是两个长度相等的字符串，并且在结果字典中，x 中每个字符将被映射到 y 中相同位置的字符。如果有第三个参数，它必须是一个字符串，其中的字符将在结果中被映射到 None。

str.**partition**(*sep*)

在 *sep* 首次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含字符本身以及两个空字符串。

str.**removeprefix**(*prefix*, */*)

如果字符串以 *prefix* 字符串开头，返回 string[len(prefix):]。否则，返回原始字符串的副本:

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

Added in version 3.9.

str.**removesuffix**(*suffix*, */*)

如果字符串以 *suffix* 字符串结尾，并且 *suffix* 非空，返回 string[:-len(suffix)]。否则，返回原始字符串的副本:

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

Added in version 3.9.

str.**replace**(*old*, *new*, *count=-1*)

返回字符串的副本，其中出现的所有子字符串 *old* 都将被替换为 *new*。如果给出了 *count*，则只替换前 *count* 次出现。如果 *count* 未指定或为 -1，则全部替换。

在 3.13 版本发生变更: 现在可支持 *count* 关键字参数。

str.**rfind**(*sub*[, *start*[, *end*]])

返回子字符串 *sub* 在字符串内被找到的最大（最右）索引，这样 *sub* 将包含在 s[start:end] 当中。可选参数 *start* 与 *end* 会被解读为切片表示法。如果未找到则返回 -1。

str.**rindex**(*sub*[, *start*[, *end*]])

类似于*rfind()*，但在子字符串 *sub* 未找到时会引发*ValueError*。

str.**rjust**(*width*[, *fillchar*])

返回长度为 *width* 的字符串，原字符串在其中靠右对齐。使用指定的 *fillchar* 填充空位 (默认使用 ASCII 空格符)。如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

str.**rpartition**(*sep*)

在 *sep* 最后一次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含两个空字符串以及字符串本身。

str.**rsplit**(*sep=None*, *maxsplit=-1*)

返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分，从 *最右边* 开始。如果 *sep* 未指定或为 `None`，任何空白字符串都会被作为分隔符。除了从右边开始拆分，*rsplit()* 的其他行为都类似于下文所述的*split()*。

str.**rstrip**([*chars*])

返回原字符串的副本，移除其中的末尾字符。*chars* 参数为指定要移除字符的字符串。如果省略或为 `None`，则 *chars* 参数默认移除空白符。实际上 *chars* 参数并非指定单个后缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

要删除单个后缀字符串，而不是全部给定集合中的字符，请参见*str.removesuffix()* 方法。例如:

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

str.**split**(*sep=None*, *maxsplit=-1*)

返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分（因此，列表最多会有 `maxsplit+1` 个元素）。如果 *maxsplit* 未指定或为 `-1`，则不限制拆分次数（进行所有可能的拆分）。

如果给出了 *sep*，则连续的分隔符不会被组合在一起而是会被视为分隔空字符串 (例如 `'1,,2'.split(',')` 将返回 `['1', '', '2']`)。*sep* 参数可能是由多个字符组成的单个分隔符 (要使用多个分隔符进行拆分，请使用*re.split()*)。使用指定的分隔符拆分一个空字符串将返回 `['']`。

例如:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
>>> '1<>2<>3<4'.split('<>')
['1', '2', '3<4']
```

如果 *sep* 未指定或为 `None`，则会应用另一种拆分算法：连续的空格会被视为单个分隔符，其结果将不包含开头或末尾的空字符串，如果字符串包含前缀或后缀空格的话。因此，使用 `None` 拆分空字符串或仅包含空格的字符串将返回 `[]`。

例如:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

str.**splitlines**(*keepends=False*)

返回由原字符串中各行组成的列表，在行边界的位置拆分。结果列表中不包含行边界，除非给出了 *keepends* 且为真值。

此方法会以下列行边界进行拆分。特别地，行边界是*universal newlines* 的一个超集。

表示符	描述
\n	换行
\r	回车
\r\n	回车 + 换行
\v 或 \x0b	行制表符
\f 或 \x0c	换表单
\x1c	文件分隔符
\x1d	组分隔符
\x1e	记录分隔符
\x85	下一行 (C1 控制码)
\u2028	行分隔符
\u2029	段分隔符

在 3.2 版本发生变更: \v 和 \f 被添加到行边界列表

例如:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

不同于*split()*，当给出了分隔字符串 *sep* 时，对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

作为比较，split('\n') 的结果为:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

str.**startswith**(*prefix*[, *start*[, *end*]])

如果字符串以指定的 *prefix* 开始则返回 True，否则返回 False。*prefix* 也可以为由多个供查找的前缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

str.**strip**([*chars*])

返回原字符串的副本，移除其中的前导和末尾字符。*chars* 参数为指定要移除字符的字符串。如果省略或为 None，则 *chars* 参数默认移除空白符。实际上 *chars* 参数并非指定单个前缀或后缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

最外侧的前导和末尾 *chars* 参数值将从字符串中移除。开头端的字符的移除将在遇到一个未包含于 *chars* 所指定字符集的字符时停止。类似的操作也将在结尾端发生。例如:

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

str.**swapcase**()

返回原字符串的副本，其中大写字符转换为小写，反之亦然。请注意 `s.swapcase().swapcase() == s` 并不一定为真值。

str.**title**()

返回原字符串的标题版本，其中每个单词第一个字母为大写，其余字母为小写。

例如:

```
>>> 'Hello world'.title()
'Hello World'
```

该算法使用一种简单的与语言无关的定义，将连续的字母组合视为单词。该定义在多数情况下都很有效，但它也意味着代表缩写形式与所有格的撇号也会成为单词边界，这可能导致不希望的结果:

```
>>> "they're bill's friends from the UK".title()
"They're Bill'S Friends From The Uk"
```

string.capwords() 函数没有此问题，因为它只用空格来拆分单词。

作为替代，可以使用正则表达式来构造针对撇号的变通处理:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0).capitalize(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

str.**translate**(*table*)

返回原字符串的副本，其中每个字符按给定的转换表进行映射。转换表必须是一个通过 `__getitem__()` 来实现索引操作的对象，通常为*mapping* 或*sequence*。当以 Unicode 码位序号（整数）为索引时，转换表对象可以做以下任何一种操作：返回 Unicode 码位序号或字符串，将字符映射为一个或多个其他字符；返回 `None`，将字符从返回的字符串中删除；或引发*LookupError* 异常，将字符映射为其自身。

你可以使用*str.maketrans()* 基于不同格式的字符到字符映射来创建一个转换映射表。

另请参阅*codecs* 模块以了解定制字符映射的更灵活方式。

str.**upper**()

返回原字符串的副本，其中所有区分大小写的字符[Page 52, 4] 均转换为大写。请注意如果 `s` 包含不区分大小写的字符或者如果结果字符的 Unicode 类别不是”Lu” (Letter, uppercase) 而是”Lt” (Letter, titlecase) 则 `s.upper().isupper()` 有可能为 `False`。

所使用的大写转换算法 在 Unicode 标准的第 3.13 节’Default Case Folding’ 中描述。

str.**zfill**(*width*)

返回原字符串的副本，在左边填充 ASCII `'0'` 数码使其长度变为 *width*。正负值前缀 (`'+'`/`'-'`) 的处理方式是在正负符号 之后 填充而非在之前。如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

例如：

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.8.2 格式化字符串字面值（f-字符串）

Added in version 3.6.

在 3.7 版本发生变更: `await` 和 `async for` 可在 f-字符串内部的表达式中使用。

在 3.8 版本发生变更: 增加了调试运算符 (=)

在 3.12 版本发生变更: 许多针对 f-字符串内部的表达式的限制已被移除。例如，嵌套字符串、注释和反斜杠现在都是允许的。

f-字符串 (正式名称为 *格式化字符串字面值*) 是带有 `f` 或 `F` 前缀的字符串字面值。这种类型的字符串字面值允许将任意 Python 表达式嵌入到由花括号 (`{}`) 标记的 *替换字段* 内部。这些表达式将在运行时被求值，这与*str.format()* 类似，并被转换为常规的*str* 对象。例如：

```
>>> who = 'nobody'
>>> nationality = 'Spanish'
>>> f'{who.title()} expects the {nationality} Inquisition!'
'Nobody expects the Spanish Inquisition!'
```

也可以使用包含多行的 f-字符串：

```
>>> f'''This is a string
... on two lines'''
'This is a string\non two lines'
```

一个单独的左花括号，`'{'`，标记一个可包含任意 Python 表达式的 *替换字段*：

```
>>> nationality = 'Spanish'
>>> f'The {nationality} Inquisition!'
'The Spanish Inquisition!'
```

要包括 `{` 或 `}` 字面值，请使用双花括号：

```
>>> x = 42
>>> f'{{x}} is {x}'
'{x} is 42'
```

还可以使用函数，以及格式说明符：

```
>>> from math import sqrt
>>> f'√2 \N{ALMOST EQUAL TO} {sqrt(2):.5f}'
'√2 ≈ 1.41421'
```

在默认情况下，任何非字符串表达式都将使用*str()* 来转换：

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 3)}'
'1/3'
```

要使用显式转换，请使用 ！(叹号) 运算符，后面跟任意的有效格式说明符，包括：

转换符	含意
!a	*ascii()*
!r	*repr()*
!s	*str()*

例如:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 3)!s}'
'1/3'
>>> f'{Fraction(1, 3)!r}'
'Fraction(1, 3)'
>>> question = '¿Dónde está el Presidente?'
>>> print(f'{question!a}')
'\xbfD\xf3nde est\xe1 el Presidente?'
```

在调试期间同时看到表达式和值会很有帮助，具体是在表达式后使用等号 (=)。这将保留花括号内部的空格，并可以使用转换器。在默认情况下，调试运算符使用 *repr()* (!r) 转换器。例如:

```
>>> from fractions import Fraction
>>> calculation = Fraction(1, 3)
>>> f'{calculation=}'
'calculation=Fraction(1, 3)'
>>> f'{calculation = }'
'calculation = Fraction(1, 3)'
>>> f'{calculation = !s}'
'calculation = 1/3'
```

输出一旦已被求值，就可以用格式说明符 后面跟一个冒号 (':') 来格式化它。在表达式已被求值，并可能被转换为字符串之后，就会调用结果的 __format__() 方法并附带该格式说明符，或者如果未给出格式说明符则附带空字符串。随后将使用已格式化的结果作为替换字段最终的值。例如:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 7):.6f}'
'0.142857'
>>> f'{Fraction(1, 7):_^+10}'
'___+1/7___'
```

4.8.3 printf 风格的字符串格式化

> **备注**
>
> 此处介绍的格式化操作具有多种怪异特性，可能导致许多常见错误（例如无法正确显示元组和字典）。使用较新的 格式化字符串字面值，*str.format()* 接口或模板字符串 有助于避免这样的错误。这些替代方案中的每一种都更好地权衡并提供了简单、灵活以及可扩展性优势。

字符串具有一种特殊的内置操作即 % (求模) 运算符。这也被称为字符串的 格式化或 插值运算符。对于给定的 format % values (其中 *format* 是一个字符串)，在 *format* 中的 % 转换标记符将被替换为零个或多个 *values* 中的元素。其效果类似于在 C 语言中使用 sprintf() 函数。例如:

```
>>> print('%s has %d quote types.' % ('Python', 2))
Python has 2 quote types.
```

如果 *format* 要求一个单独参数，则 *values* 可以为一个非元组对象。[5] 否则的话，*values* 必须或者是一个包含项数与格式字符串中指定的转换符项数相同的元组，或者是一个单独映射对象（例如字典）。

[5] 若只是要格式化一个元组，则应提供一个单例元组，其中只包含一个元素，就是需要格式化的那个元组。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 映射键（可选），由加圆括号的字符序列组成 (例如 `(somename)`)。
3. 转换旗标（可选），用于影响某些转换类型的结果。
4. 最小字段宽度（可选）。如果指定为 `'*'` (星号)，则实际宽度会从 *values* 元组的下一元素中读取，要转换的对象则为最小字段宽度和可选的精度之后的元素。
5. 精度（可选），以在 `'.'` (点号) 之后加精度值的形式给出。如果指定为 `'*'` (星号)，则实际精度会从 *values* 元组的下一元素中读取，要转换的对象则为精度之后的元素。
6. 长度修饰符（可选）。
7. 转换类型。

当右边的参数为一个字典（或其他映射类型）时，字符串中的格式 *必须* 包含加圆括号的映射键，对应 `'%'` 字符之后字典中的每一项。映射键将从映射中选取要格式化的值。例如：

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

在此情况下格式中不能出现 `*` 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

旗标	含意
`'#'`	值的转换将使用“替代形式”（具体定义见下文）。
`'0'`	转换将为数字值填充零字符。
`'-'`	转换值将靠左对齐（如果同时给出 `'0'` 转换，则会覆盖后者）。
`' '`	(空格) 符号位转换产生的正数（或空字符串）前将留出一个空格。
`'+'`	符号字符 (`'+'` 或 `'-'`) 将显示于转换结果的开头（会覆盖“空格”旗标）。

可以给出长度修饰符 (`h`, `l` 或 `L`)，但会被忽略，因为对 Python 来说没有必要 -- 所以 `%ld` 等价于 `%d`。

转换类型为：

转换符	含意	备注
`'d'`	有符号十进制整数。	
`'i'`	有符号十进制整数。	
`'o'`	有符号八进制数。	(1)
`'u'`	过时类型 -- 等价于 `'d'`。	(6)
`'x'`	有符号十六进制数（小写）。	(2)
`'X'`	有符号十六进制数（大写）。	(2)
`'e'`	浮点指数格式（小写）。	(3)
`'E'`	浮点指数格式（大写）。	(3)
`'f'`	浮点十进制格式。	(3)
`'F'`	浮点十进制格式。	(3)
`'g'`	浮点格式。如果指数小于 -4 或不小于精度则使用小写指数格式，否则使用十进制格式。	(4)
`'G'`	浮点格式。如果指数小于 -4 或不小于精度则使用大写指数格式，否则使用十进制格式。	(4)
`'c'`	单个字符（接受整数或单个字符的字符串）。	
`'r'`	字符串（使用*repr()* 转换任何 Python 对象）。	(5)
`'s'`	字符串（使用*str()* 转换任何 Python 对象）。	(5)
`'a'`	字符串（使用*ascii()* 转换任何 Python 对象）。	(5)
`'%'`	不转换参数，在结果中输出一个 `'%'` 字符。	

注释:

(1) 此替代形式会在第一个数码之前插入标示八进制数的前缀 (`'0o'`)。

(2) 此替代形式会在第一个数码之前插入 `'0x'` 或 `'0X'` 前缀（取决于是使用 `'x'` 还是 `'X'` 格式）。

(3) 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

小数点后的数码位数由精度决定，默认为 6。

(4) 此替代形式总是会在结果中包含一个小数点，末尾各位的零不会如其他情况下那样被移除。

小数点前后的有效数码位数由精度决定，默认为 6。

(5) 如果精度为 `N`，输出将截短为 `N` 个字符。

(6) 参见 **PEP 237**。

由于 Python 字符串显式指明长度，`%s` 转换不会将 `'\0'` 视为字符串的结束。

在 3.1 版本发生变更: 绝对值超过 1e50 的 `%f` 转换不会再被替换为 `%g` 转换。

4.9 二进制序列类型 --- bytes, bytearray, memoryview

操作二进制数据的核心内置类型是*bytes* 和*bytearray*。它们由*memoryview* 提供支持，该对象使用 缓冲区协议来访问其他二进制对象所在内存，不需要创建对象的副本。

array 模块支持高效地存储基本数据类型，例如 32 位整数和 IEEE754 双精度浮点值。

4.9.1 bytes 对象

bytes 对象是由单个字节构成的不可变序列。由于许多主要二进制协议都基于 ASCII 文本编码，因此 bytes 对象提供了一些仅在处理 ASCII 兼容数据时可用，并且在许多特性上与字符串对象紧密相关的方法。

class bytes([*source*[, *encoding*[, *errors*]]])

首先，表示 bytes 字面值的语法与字符串字面值的大致相同，只是添加了一个 `b` 前缀:

- 单引号: `b'同样允许嵌入" 双" 引号'`。
- 双引号: `b" 仍然允许嵌入 '单' 引号"`
- 三重引号: `b'''三重单引号'''`, `b""" 三重双引号"""`

bytes 字面值中只允许 ASCII 字符（无论源代码声明的编码格式为何）。任何超出 127 的二进制值必须使用相应的转义序列形式加入 bytes 字面值。

像字符串字面值一样，bytes 字面值也可以使用 `r` 前缀来禁用转义序列处理。请参阅 strings 了解有关各种 bytes 字面值形式的详情，包括所支持的转义序列。

虽然 bytes 字面值和表示法是基于 ASCII 文本的，但 bytes 对象的行为实际上更像是不可变的整数序列，序列中的每个值的大小被限制为 `0 <= x < 256` (如果违反此限制将引发*ValueError*)。这种限制是有意设计用以强调以下事实，虽然许多二进制格式都包含基于 ASCII 的元素，可以通过某些面向文本的算法进行有用的操作，但情况对于任意二进制数据来说通常却并非如此（盲目地将文本处理算法应用于不兼容 ASCII 的二进制数据格式往往将导致数据损坏）。

除了字面值形式，bytes 对象还可以通过其他几种方式来创建:

- 指定长度的以零值填充的 bytes 对象: `bytes(10)`
- 通过由整数组成的可迭代对象: `bytes(range(20))`
- 通过缓冲区协议复制现有的二进制数据: `bytes(obj)`

另请参阅*bytes* 内置类型。

由于两个十六进制数码精确对应一个字节，因此十六进制数是描述二进制数据的常用格式。相应地，bytes 类型具有从此种格式读取数据的附加类方法:

classmethod fromhex(*string*)

此*bytes*类方法返回一个解码给定字符串的 bytes 对象。字符串必须由表示每个字节的两个十六进制数码构成，其中的 ASCII 空白符会被忽略。

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

在 3.7 版本发生变更: *bytes.fromhex()* 现在会忽略所有 ASCII 空白符而不只是空格符。

存在一个反向转换函数，可以将 bytes 对象转换为对应的十六进制表示。

hex(*[sep[, bytes_per_sep]]*)

返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

如果你希望令十六进制数字符串更易读，你可以指定单个字符分隔符作为 *sep* 形参包含于输出中。默认情况下，该分隔符会放在每个字节之间。第二个可选的 *bytes_per_sep* 形参控制间距。正值会从右开始计算分隔符的位置，负值则是从左开始。

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

Added in version 3.5.

在 3.8 版本发生变更: *bytes.hex()* 现在支持可选的 *sep* 和 *bytes_per_sep* 形参以在十六进制输出的字节之间插入分隔符。

由于 bytes 对象是由整数构成的序列（类似于元组），因此对于一个 bytes 对象 *b*，`b[0]` 将为一个整数，而 `b[0:1]` 将为一个长度为 1 的 bytes 对象。（这与文本字符串不同，索引和切片所产生的将都是一个长度为 1 的字符串）。

bytes 对象的表示使用字面值格式 (`b'...'`)，因为它通常都要比像 `bytes([46, 46, 46])` 这样的格式更好用。你总是可以使用 `list(b)` 将 bytes 对象转换为一个由整数构成的列表。

4.9.2 bytearray 对象

bytearray 对象是*bytes* 对象的可变对应物。

class bytearray(*[source[, encoding[, errors]]]*)

bytearray 对象没有专属的字面值语法，它们总是通过调用构造器来创建：

- 创建一个空实例: `bytearray()`
- 创建一个指定长度的以零值填充的实例: `bytearray(10)`
- 通过由整数组成的可迭代对象: `bytearray(range(20))`
- 通过缓冲区协议复制现有的二进制数据: `bytearray(b'Hi!')`

由于 bytearray 对象是可变的，该对象除了*bytes 和 bytearray 操作* 中所描述的 bytes 和 bytearray 共有操作之外，还支持可变 序列操作。

另请参见*bytearray* 内置类型。

由于两个十六进制数码精确对应一个字节，因此十六进制数是描述二进制数据的常用格式。相应地，bytearray 类型具有从此种格式读取数据的附加类方法：

classmethod fromhex(*string*)

bytearray 类方法返回一个解码给定字符串的 bytearray 对象。字符串必须由表示每个字节的两个十六进制数码构成，其中的 ASCII 空白符会被忽略。

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

在 3.7 版本发生变更: *bytearray.fromhex()* 现在会忽略所有 ASCII 空白符而不只是空格符。

存在一个反向转换函数，可以将 bytearray 对象转换为对应的十六进制表示。

hex([*sep*[, *bytes_per_sep*]])

返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Added in version 3.5.

在 3.8 版本发生变更: 与*bytes.hex()* 相似，*bytearray.hex()* 现在支持可选的 *sep* 和 *bytes_per_sep* 参数以在十六进制输出的字节之间插入分隔符。

由于 bytearray 对象是由整数构成的序列（类似于列表），因此对于一个 bytearray 对象 *b*，`b[0]` 将为一个整数，而 `b[0:1]` 将为一个长度为 1 的 bytearray 对象。（这与文本字符串不同，索引和切片所产生的将都是一个长度为 1 的字符串）。

bytearray 对象的表示使用 bytes 对象字面值格式 (`bytearray(b'...')`)，因为它通常都要比 `bytearray([46, 46, 46])` 这样的格式更好用。你总是可以使用 `list(b)` 将 bytearray 对象转换为一个由整数构成的列表。

4.9.3 bytes 和 bytearray 操作

bytes 和 bytearray 对象都支持通用 序列操作。它们不仅能与相同类型的操作数，也能与任何*bytes-like object* 进行互操作。由于这样的灵活性，它们可以在操作中自由地混合而不会导致错误。但是，操作结果的返回值类型可能取决于操作数的顺序。

备注

bytes 和 bytearray 对象的方法不接受字符串作为其参数，就像字符串的方法不接受 bytes 对象作为其参数一样。例如，你必须使用以下写法:

```
a = "abc"
b = a.replace("a", "f")
```

和:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

某些 bytes 和 bytearray 操作假定使用兼容 ASCII 的二进制格式，因此在处理任意二进数数据时应当避免使用。这些限制会在下文中说明。

备注

使用这些基于 ASCII 的操作来处理未以基于 ASCII 的格式存储的二进制数据可能会导致数据损坏。

bytes 和 bytearray 对象的下列方法可以用于任意二进制数据。

`bytes`.**count**(*sub*[, *start*[, *end*]])

bytearray.**count**(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在 [*start*, *end*] 范围内非重叠出现的次数。可选参数 *start* 与 *end* 会被解读为切片表示法。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

如果 *sub* 为空，则返回字符之间的空切片的数量即字节串对象的长度加一。

在 3.3 版本发生变更: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**removeprefix**(*prefix*, /)

bytearray.**removeprefix**(*prefix*, /)

如果二进制数据以 *prefix* 字符串开头，返回 `bytes[len(prefix):]`。否则，返回原始二进制数据的副本:

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

prefix 可以是任意*bytes-like object*。

备注

此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

Added in version 3.9.

bytes.**removesuffix**(*suffix*, /)

bytearray.**removesuffix**(*suffix*, /)

如果二进制数据以 *suffix* 字符串结尾，并且 *suffix* 非空，返回 `bytes[:-len(suffix)]`。否则，返回原始二进制数据的副本:

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

suffix 可以是任意*bytes-like object*。

备注

此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

Added in version 3.9.

bytes.**decode**(*encoding='utf-8'*, *errors='strict'*)

bytearray.**decode**(*encoding='utf-8'*, *errors='strict'*)

返回解码为*str* 的字节串。

encoding 默认为 `'utf-8'`；请参阅*标准编码* 了解其他可能的值。

errors 控制如何处理编码错误。如为 `'strict'` (默认值)，则会引发*UnicodeError*。其他可能的值有 `'ignore'`, `'replace'` 以及通过*codecs.register_error()* 注册的任何其他名称。请参阅*错误处理方案* 了解详情。

出于性能原因，除非真正发生了编码错误，启用了*Python 开发模式* 或使用了 调试编译版否则不会检查 *errors* 值的有效性。

> **备注**
>
> 将 *encoding* 参数传给 `str` 允许直接解码任何 *bytes-like object*，无须创建临时的 `bytes` 或 `bytearray` 对象。

在 3.1 版本发生变更: 加入了对关键字参数的支持。

在 3.9 版本发生变更: 现在会在 *Python 开发模式* 和 调试模式下检查 *errors* 参数的值。

`bytes.`**`endswith`**(*suffix*[, *start*[, *end*]])

`bytearray.`**`endswith`**(*suffix*[, *start*[, *end*]])

如果二进制数据以指定的 *suffix* 结束则返回 `True`，否则返回 `False`。*suffix* 也可以为由多个供查找的后缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

要搜索的后缀可以是任意 *bytes-like object*。

`bytes.`**`find`**(*sub*[, *start*[, *end*]])

`bytearray.`**`find`**(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在数据中被找到的最小索引，*sub* 包含于切片 `s[start:end]` 之内。可选参数 *start* 与 *end* 会被解读为切片表示法。如果 *sub* 未被找到则返回 `-1`。

要搜索的子序列可以是任意 *bytes-like object* 或是 0 至 255 范围内的整数。

> **备注**
>
> `find()` 方法应该只在你需要知道 *sub* 所在位置时使用。要检查 *sub* 是否为子串，请使用 `in` 操作符:
>
> ```
> >>> b'Py' in b'Python'
> True
> ```

在 3.3 版本发生变更: 也接受 0 至 255 范围内的整数作为子序列。

`bytes.`**`index`**(*sub*[, *start*[, *end*]])

`bytearray.`**`index`**(*sub*[, *start*[, *end*]])

类似于 `find()`，但在找不到子序列时会引发 `ValueError`。

要搜索的子序列可以是任意 *bytes-like object* 或是 0 至 255 范围内的整数。

在 3.3 版本发生变更: 也接受 0 至 255 范围内的整数作为子序列。

`bytes.`**`join`**(*iterable*)

`bytearray.`**`join`**(*iterable*)

返回一个由 *iterable* 中的二进制数据序列拼接而成的 bytes 或 bytearray 对象。如果 *iterable* 中存在任何非字节类对象 包括存在 `str` 对象值则会引发 `TypeError`。提供该方法的 bytes 或 bytearray 对象的内容将作为元素之间的分隔。

`static` `bytes.`**`maketrans`**(*from*, *to*)

`static` `bytearray.`**`maketrans`**(*from*, *to*)

此静态方法返回一个可用于 `bytes.translate()` 的转换对照表，它将把 *from* 中的每个字符映射为 *to* 中相同位置上的字符；*from* 与 *to* 必须都是字节类对象 并且具有相同的长度。

Added in version 3.1.

`bytes.`**`partition`**(*sep*)

bytearray.**partition**(*sep*)

在 *sep* 首次出现的位置拆分序列，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身或其 bytearray 副本，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含原序列以及两个空的 bytes 或 bytearray 对象。

要搜索的分隔符可以是任意*bytes-like object*。

bytes.**replace**(*old*, *new*[, *count*])

bytearray.**replace**(*old*, *new*[, *count*])

返回序列的副本，其中出现的所有子序列 *old* 都将被替换为 *new*。如果给出了可选参数 *count*，则只替换前 *count* 次出现。

要搜索的子序列及其替换序列可以是任意*bytes-like object*。

> **备注**
>
> 此方法的 bytearray 版本 *并非*原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**rfind**(*sub*[, *start*[, *end*]])

bytearray.**rfind**(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在序列内被找到的最大（最右）索引，这样 *sub* 将包含在 `s[start:end]` 当中。可选参数 *start* 与 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

在 3.3 版本发生变更: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**rindex**(*sub*[, *start*[, *end*]])

bytearray.**rindex**(*sub*[, *start*[, *end*]])

类似于`rfind()`，但在子序列 *sub* 未找到时会引发`ValueError`。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

在 3.3 版本发生变更: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**rpartition**(*sep*)

bytearray.**rpartition**(*sep*)

在 *sep* 最后一次出现的位置拆分序列，返回一个 3 元组，其中包含分隔符之前的部分，分隔符本身或其 bytearray 副本，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含两个空的 bytes 或 bytearray 对象以及原序列的副本。

要搜索的分隔符可以是任意*bytes-like object*。

bytes.**startswith**(*prefix*[, *start*[, *end*]])

bytearray.**startswith**(*prefix*[, *start*[, *end*]])

如果二进制数据以指定的 *prefix* 开头则返回 `True`，否则返回 `False`。*prefix* 也可以为由多个供查找的前缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

要搜索的前缀可以是任意*bytes-like object*。

bytes.**translate**(*table*, */*, *delete=b''*)

bytearray.**translate**(*table*, */*, *delete=b''*)

返回原 bytes 或 bytearray 对象的副本，移除其中所有在可选参数 *delete* 中出现的 bytes，其余 bytes 将通过给定的转换表进行映射，该转换表必须是长度为 256 的 bytes 对象。

你可以使用`bytes.maketrans()`方法来创建转换表。

对于仅需移除字符的转换，请将 *table* 参数设为 `None`:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

在 3.6 版本发生变更: 现在支持将 *delete* 作为关键字参数。

以下 bytes 和 bytearray 对象的方法的默认行为会假定使用兼容 ASCII 的二进制格式，但通过传入适当的参数仍然可用于任意二进制数据。请注意本小节中所有的 bytearray 方法都 不是原地执行操作，而是会产生新的对象。

bytes.**center**(*width*[, *fillbyte*])

bytearray.**center**(*width*[, *fillbyte*])

返回原对象的副本，在长度为 *width* 的序列内居中，使用指定的 *fillbyte* 填充两边的空位（默认使用 ASCII 空格符）。对于*bytes* 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

> **备注**
>
> 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**ljust**(*width*[, *fillbyte*])

bytearray.**ljust**(*width*[, *fillbyte*])

返回原对象的副本，在长度为 *width* 的序列中靠左对齐。使用指定的 *fillbyte* 填充空位（默认使用 ASCII 空格符）。对于*bytes* 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

> **备注**
>
> 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**lstrip**([*chars*])

bytearray.**lstrip**([*chars*])

返回原序列的副本，移除指定的前导字节。*chars* 参数为指定要移除字节值集合的二进制序列——这个名称表明此方法通常是用于 ASCII 字符。如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。*chars* 参数并非指定单个前缀；而是会移除参数值的所有组合:

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

要移除的二进制序列可以是任意*bytes-like object* 。要删除单个前缀字符串，而不是全部给定集合中的字符，请参见*str.removeprefix()* 方法。例如:

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

> **备注**
>
> 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**rjust**(*width*[, *fillbyte*])

bytearray.**rjust**(*width*[, *fillbyte*])

返回原对象的副本，在长度为 *width* 的序列中靠右对齐。使用指定的 *fillbyte* 填充空位（默认使用 ASCII 空格符）。对于*bytes* 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

> **备注**
>
> 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**rsplit**(*sep=None*, *maxsplit=-1*)

bytearray.**rsplit**(*sep=None*, *maxsplit=-1*)

将二进制序列拆分为相同类型的子序列，使用 *sep* 作为分隔符。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分，从 最右边开始。如果 *sep* 未指定或为 `None`，任何只包含 ASCII 空白符的子序列都会被作为分隔符。除了从右边开始拆分，*rsplit()* 的其他行为都类似于下文所述的*split()*。

bytes.**rstrip**([*chars*])

bytearray.**rstrip**([*chars*])

返回原序列的副本，移除指定的末尾字节。*chars* 参数为指定要移除字节值集合的二进制序列——这个名称表明此方法通常是用于 ASCII 字符。如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。*chars* 参数并非指定单个后缀；而是会移除参数值的所有组合:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

要移除的二进制序列可以是任意*bytes-like object* 。要删除单个后缀字符串，而不是全部给定集合中的字符，请参见*str.removesuffix()* 方法。例如:

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

> **备注**
>
> 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**split**(*sep=None*, *maxsplit=-1*)

bytearray.**split**(*sep=None*, *maxsplit=-1*)

将二进制序列拆分为相同类型的子序列，使用 *sep* 作为分隔符。如果给出了 *maxsplit* 且非负值，则最多进行 *maxsplit* 次拆分（因此，列表最多会有 `maxsplit+1` 个元素）。如果 *maxsplit* 未指定或为 `-1`，则不限制拆分次数（进行所有可能的拆分）。

如果给出了 *sep*，则连续的分隔符不会被组合在一起而是会被视为分隔空子序列 (例如 `b'1,,2'.split(b',')` 将将返回 `[b'1', b'', b'2']`)。*sep* 参数可能是由多个序列组成的单个分隔符。使用指定的分隔符拆分一个空序列将返回 `[b'']` 或 `[bytearray(b'')]`，具体取决于被拆分对象的类型。*sep* 参数可以是任何*bytes-like object*。

例如:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
>>> b'1<>2<>3<4'.split(b'<>')
[b'1', b'2', b'3<4']
```

如果 *sep* 未指定或为 `None`，则会应用另一种拆分算法：连续的 ASCII 空白符会被视为单个分隔符，其结果将不包含序列开头或末尾的空白符。因此，在不指定分隔符的情况下对空序列或仅包含 ASCII 空白符的序列进行拆分将返回 `[]`。

例如：

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

bytes.**strip**(*[chars]*)

bytearray.**strip**(*[chars]*)

返回原序列的副本，移除指定的开头和末尾字节。*chars* 参数为指定要移除字节值集合的二进制序列——这个名称表明此方法通常是用于 ASCII 字符。如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。*chars* 参数并非指定单个前缀或后缀；而是会移除参数值的所有组合：

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

要移除的字节值二进制序列可以是任意*bytes-like object*。

备注

此方法的 bytearray 版本 *并非* 原地操作——它总是产生一个新对象，即便没有做任何改变。

以下 bytes 和 bytearray 对象的方法会假定使用兼容 ASCII 的二进制格式，不应当被应用于任意二进制数据。请注意本小节中所有的 bytearray 方法都 *不是* 原地执行操作，而是会产生新的对象。

bytes.**capitalize**()

bytearray.**capitalize**()

返回原序列的副本，其中每个字节将都将被解读为一个 ASCII 字符，并且第一个字节的字符大写而其余的小写。非 ASCII 字节值将保持原样不变。

备注

此方法的 bytearray 版本 *并非* 原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**expandtabs**(*tabsize=8*)

bytearray.**expandtabs**(*tabsize=8*)

返回序列的副本，其中所有的 ASCII 制表符会由一个或多个 ASCII 空格替换，具体取决于当前列位置和给定的制表符宽度。每 *tabsize* 个字节设为一个制表位（默认值 8 时设定的制表位在列 0, 8, 16 依次类推）。要展开序列，当前列位置将被设为零并逐一检查序列中的每个字节。如果字节为 ASCII 制表符 (`b'\t'`)，则并在结果中插入一个或多个空格符，直到当前列等于下一个制表位。（制表符本身不会被复制。）如果当前字节为 ASCII 换行符 (`b'\n'`) 或回车符 (`b'\r'`)，它会被复制并将当前列重设为零。任何其他字节会被不加修改地复制并将当前列加一，不论该字节值在被打印时会如何显示：

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

备注

此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**isalnum**()

bytearray.**isalnum**()

如果序列中所有字节都是字母类 ASCII 字符或 ASCII 十进制数码并且序列非空则返回 `True`，否则返回 `False`。字母类 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。ASCII 十进制数码就是字节值包含在序列 `b'0123456789'` 中的字符。

例如：

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()

bytearray.**isalpha**()

如果序列中所有字节都是字母类 ASCII 字符并且序列不非空则返回 `True`，否则返回 `False`。字母类 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

例如：

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()

bytearray.**isascii**()

如果序列为空或序列中所有字节都是 ASCII 字节则返回 `True`，否则返回 `False`。ASCII 字节的取值范围是 0-0x7F。

Added in version 3.7.

bytes.**isdigit**()

bytearray.**isdigit**()

如果序列中所有字节都是 ASCII 十进制数码并且序列非空则返回 `True`，否则返回 `False`。ASCII 十进制数码就是字节值包含在序列 `b'0123456789'` 中的字符。

例如：

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()

bytearray.**islower**()

如果序列中至少有一个小写的 ASCII 字符并且没有大写的 ASCII 字符则返回 `True`，否则返回 `False`。

例如：

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

`bytes.**isspace**()`

`bytearray.**isspace**()`

如果序列中所有字节都是 ASCII 空白符并且序列非空则返回 `True`，否则返回 `False`。ASCII 空白符就是字节值包含在序列 `b' \t\n\r\x0b\f'` (空格, 制表, 换行, 回车, 垂直制表, 进纸) 中的字符。

`bytes.**istitle**()`

`bytearray.**istitle**()`

如果序列为 ASCII 标题大小写形式并且序列非空则返回 `True`，否则返回 `False`。请参阅*bytes.title()* 了解有关“标题大小写”的详细定义。

例如：

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

`bytes.**isupper**()`

`bytearray.**isupper**()`

如果序列中至少有一个大写字母 ASCII 字符并且没有小写 ASCII 字符则返回 `True`，否则返回 `False`。

例如：

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

`bytes.**lower**()`

`bytearray.**lower**()`

返回原序列的副本，其所有大写 ASCII 字符均转换为对应的小写形式。

例如：

```
>>> b'Hello World'.lower()
b'hello world'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

> **备注**
>
> 此方法的 bytearray 版本 *并非* 原地操作——它总是产生一个新对象，即便没有做任何改变。

`bytes.**splitlines**(*keepends=False*)`

`bytearray.**splitlines**(*keepends=False*)`

返回由原二进制序列中各行组成的列表，在 ASCII 行边界符的位置拆分。此方法使用*universal newlines* 方式来分行。结果列表中不包含换行符，除非给出了 *keepends* 且为真值。

例如：

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

不同于*split()*，当给出了分隔符 *sep* 时，对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.**swapcase**()

bytearray.**swapcase**()

返回原序列的副本，其所有小写 ASCII 字符均转换为对应的大写形式，反之亦反。

例如:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

不同于*str.swapcase()*，在二进制版本下 `bin.swapcase().swapcase() == bin` 始终成立。大小写转换在 ASCII 中是对称的，即使其对于任意 Unicode 码位来说并不总是成立。

备注

此方法的 bytearray 版本 *并非* 原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**title**()

bytearray.**title**()

返回原二进制序列的标题版本，其中每个单词以一个大写 ASCII 字符为开头，其余字母为小写。不区别大小写的字节值将保持原样不变。

例如:

```
>>> b'Hello world'.title()
b'Hello World'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。所有其他字节值都不区分大小写。

该算法使用一种简单的与语言无关的定义，将连续的字母组合视为单词。该定义在多数情况下都很有效，但它也意味着代表缩写形式与所有格的撇号也会成为单词边界，这可能导致不希望的结果:

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

可以使用正则表达式来构建针对撇号的特别处理:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
```

(续下页)

(接上页)

```
...                     s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

备注

此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**upper**()

bytearray.**upper**()

返回原序列的副本，其所有小写 ASCII 字符均转换为对应的大写形式。

例如：

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

备注

此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**zfill**(*width*)

bytearray.**zfill**(*width*)

返回原序列的副本，在左边填充 `b'0'` 数码使序列长度为 *width*。正负值前缀 (`b'+'`/`b'-'`) 的处理方式是在正负符号 之后 填充而非在之前。对于 *bytes* 对象，如果 *width* 小于等于 `len(seq)` 则返回原序列。

例如：

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

备注

此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

4.9.4 printf 风格的字节串格式化

备注

此处介绍的格式化操作具有多种怪异特性，可能导致许多常见错误（例如无法正确显示元组和字典）。如果要打印的值可能为元组或字典，请将其放入一个元组中。

字节串对象 (`bytes`/`bytearray`) 具有一种特殊的内置操作：使用 `%` (取模) 运算符。这也被称为字节串的 *格式化* 或 *插值* 运算符。对于 `format % values` (其中 *format* 为一个字节串对象)，在 *format* 中的 `%` 转换标记符将被替换为零个或多个 *values* 条目。其效果类似于在 C 语言中使用 `sprintf()`。

如果 *format* 要求一个单独参数，则 *values* 可以为一个非元组对象。[Page 58, 5] 否则的话，*values* 必须或是是一个包含项数与格式字节串对象中指定的转换符项数相同的元组，或者是一个单独的映射对象（例如元组）。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 映射键（可选），由加圆括号的字符序列组成 (例如 `(somename)`)。
3. 转换旗标（可选），用于影响某些转换类型的结果。
4. 最小字段宽度（可选）。如果指定为 `'*'` (星号)，则实际宽度会从 *values* 元组的下一元素中读取，要转换的对象则为最小字段宽度和可选的精度之后的元素。
5. 精度（可选），以在 `'.'` (点号) 之后加精度值的形式给出。如果指定为 `'*'` (星号)，则实际精度会从 *values* 元组的下一元素中读取，要转换的对象则为精度之后的元素。
6. 长度修饰符（可选）。
7. 转换类型。

当右边的参数为一个字典（或其他映射类型）时，字节串对象中的格式 *必须* 包含加圆括号的映射键，对应 `'%'` 字符之后字典中的每一项。映射键将从映射中选取要格式化的值。例如：

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

在此情况下格式中不能出现 `*` 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

旗标	含意
`'#'`	值的转换将使用“替代形式”（具体定义见下文）。
`'0'`	转换将为数字值填充零字符。
`'-'`	转换值将靠左对齐（如果同时给出 `'0'` 转换，则会覆盖后者）。
`' '`	(空格) 符号位转换产生的正数（或空字符串）前将留出一个空格。
`'+'`	符号字符 (`'+'` 或 `'-'`) 将显示于转换结果的开头（会覆盖”空格”旗标）。

可以给出长度修饰符 (`h`, `l` 或 `L`)，但会被忽略，因为对 Python 来说没有必要 -- 所以 `%ld` 等价于 `%d`。

转换类型为：

转换符	含意	备注
'd'	有符号十进制整数。	
'i'	有符号十进制整数。	
'o'	有符号八进制数。	(1)
'u'	过时类型 -- 等价于 'd'。	(8)
'x'	有符号十六进制数（小写）。	(2)
'X'	有符号十六进制数（大写）。	(2)
'e'	浮点指数格式（小写）。	(3)
'E'	浮点指数格式（大写）。	(3)
'f'	浮点十进制格式。	(3)
'F'	浮点十进制格式。	(3)
'g'	浮点格式。如果指数小于 -4 或不小于精度则使用小写指数格式，否则使用十进制格式。	(4)
'G'	浮点格式。如果指数小于 -4 或不小于精度则使用大写指数格式，否则使用十进制格式。	(4)
'c'	单个字节（接受整数或单个字节对象）。	
'b'	字节串（任何遵循 缓冲区协议或是具有 __bytes__() 的对象）。	(5)
's'	's' 是 'b' 的一个别名，只应当在基于 Python2/3 的代码中使用。	(6)
'a'	字节串（使用 repr(obj).encode('ascii', 'backslashreplace') 来转换任意 Python 对象）。	(5)
'r'	'r' 是 'a' 的一个别名，只应当在基于 Python2/3 的代码中使用。	(7)
'%'	不转换参数，在结果中输出一个 '%' 字符。	

注释：

(1) 此替代形式会在第一个数码之前插入标示八进制数的前缀 ('0o')。

(2) 此替代形式会在第一个数码之前插入 '0x' 或 '0X' 前缀（取决于是使用 'x' 还是 'X' 格式）。

(3) 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

小数点后的数码位数由精度决定，默认为 6。

(4) 此替代形式总是会在结果中包含一个小数点，末尾各位的零不会如其他情况下那样被移除。

小数点前后的有效数码位数由精度决定，默认为 6。

(5) 如果精度为 N，输出将截短为 N 个字符。

(6) b'%s' 已弃用，但在 3.x 系列中将不会被移除。

(7) b'%r' 已弃用，但在 3.x 系列中将不会被移除。

(8) 参见 **PEP 237**。

备注

此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

参见

PEP 461 - 为 bytes 和 bytearray 添加% 格式化

Added in version 3.5.

4.9.5 内存视图

memoryview 对象允许 Python 代码访问一个对象的内部数据，只要该对象支持 缓冲区协议而无需进行拷贝。

class memoryview(*object*)

创建一个引用 *object* 的*memoryview* 。*object* 必须支持缓冲区协议。支持缓冲区协议的内置对象有*bytes* 和*bytearray* 。

memoryview 有 **元素**的概念，**元素**指由原始 *object* 处理的原子内存单元。对于许多简单的类型，如*bytes* 和*bytearray* ，一个元素是一个字节，但其他类型，如*array.array* 可能有更大的元素。

`len(view)` 等于*tolist* 的长度，即视图的嵌套列表表示形式。如果 `view.ndim = 1`，它将等于视图中元素的数量。

在 3.12 版本发生变更: 如果 `view.ndim == 0`，现在 `len(view)` 将引发*TypeError* 而不是返回 1.

itemsize 属性将给出单个元素的字节数。

memoryview 支持通过切片和索引访问其元素。一维切片的结果将是一个子视图:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

如果*format* 是一个来自于*struct* 模块的原生格式说明符，则也支持使用整数或由整数构成的元组进行索引，并返回具有正确类型的单个 *元素*。一维内存视图可以使用一个整数或由一个整数构成的元组进行索引。多维内存视图可以使用由恰好 *ndim* 个整数构成的元素进行索引，*ndim* 即其维度。零维内存视图可以使用空元组进行索引。

这里是一个使用非字节格式的例子:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

如果下层对象是可写的，则内存视图支持一维切片赋值。改变大小则不被允许:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
```

(续下页)

(接上页)

```
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

格式符为'B', 'b' 或'c' 的*hashable* (只读) 类型的一维内存视图也是可哈希对象。哈希被定义为 `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

在 3.3 版本发生变更: 一维内存视图现在可以被切片。格式符为'B', 'b' 或'c' 的一维内存视图现在是*hashable*。

在 3.4 版本发生变更: 内存视图现在会自动注册为*collections.abc.Sequence*

在 3.5 版本发生变更: 内存视图现在可使用整数元组进行索引。

memoryview 具有以下一些方法:

__eq__(*exporter*)

memoryview 与 **PEP 3118** 中的导出器这两者如果形状相同，并且如果当使用*struct* 语法解读操作数的相应格式代码时所有对应值都相同，则它们就是等价的。

对于*tolist()* 当前所支持的*struct* 格式字符串子集，如果 `v.tolist() == w.tolist()` 则 `v` 和 `w` 相等:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

如果两边的格式字符串都不被*struct* 模块所支持，则两对象比较结果总是不相等（即使格式字符串和缓冲区内容相同）:

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

请注意，与浮点数的情况一样，对于内存视图对象来说，`v is w` 也 并不意味着 `v == w`。

在 3.3 版本发生变更: 之前的版本比较原始内存时会忽略条目的格式与逻辑数组结构。

tobytes(*order='C'*)

将缓冲区中的数据作为字节串返回。这相当于在内存视图上调用*bytes* 构造器。

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

对于非连续数组，结果等于平面化表示的列表，其中所有元素都转换为字节串。*tobytes()* 支持所有格式字符串，不符合*struct* 模块语法的那些也包括在内。

Added in version 3.8: *order* 可以为 {'C', 'F', 'A'}。当 *order* 为'C' 或'F' 时，原始数组的数据会被转换至 C 或 Fortran 顺序。对于连续视图，'A' 会返回物理内存的精确副本。特别地，内存中的 Fortran 顺序会被保留。对于非连续视图，数据会先被转换为 C 形式。*order=None* 与 *order='C'* 是相同的。

hex(*[sep[, bytes_per_sep]]*)

返回一个字符串对象，其中分别以两个十六进制数码表示缓冲区里的每个字节。

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Added in version 3.5.

在 3.8 版本发生变更: 与*bytes.hex()* 相似，*memoryview.hex()* 现在支持可选的 *sep* 和 *bytes_per_sep* 参数以在十六进制输出的字节之间插入分隔符。

tolist()

将缓冲区内的数据以一个元素列表的形式返回。

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

在 3.3 版本发生变更: *tolist()* 现在支持*struct* 模块语法中的所有单字符原生格式以及多维表示形式。

toreadonly()

返回 memoryview 对象的只读版本。原始的 memoryview 对象不会被改变。

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[97, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

Added in version 3.8.

release()

释放由内存视图对象所公开的底层缓冲区。许多对象在被视图所获取时都会采取特殊动作（例如，*bytearray* 将会暂时禁止调整大小）；因此，调用 release() 可以方便地尽早去除这些限制（并释放任何多余的资源）。

在此方法被调用后，任何对该视图的进一步操作都将引发*ValueError* (除了可被多次调用的*release()* 本身):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

使用 with 语句，可以通过上下文管理协议达到类似的效果:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Added in version 3.2.

cast(*format*[, *shape*])

将内存视图转化为新的格式或形状。*shape* 默认为 [byte_length//new_itemsize]，这意味着结果视图将是一维的。返回值是一个新的内存视图，但缓冲区本身不会被复制。支持的转化有 1D -> C-*contiguous* 和 C-contiguous -> 1D。

目标格式被限制为*struct* 语法中的单一元素的原生格式。这些格式中的一种必须为字节格式（'B', 'b' 或'c'）。结果的字节长度必须与原始长度相同。请注意全部字节长度可能取决于具体操作系统。

将 1D/long 转换为 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

将 1D/unsigned bytes 转换为 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  ...
TypeError: memoryview: invalid type for format 'B'
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

将 1D/bytes 转换为 3D/ints 再转换为 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

将 1D/unsigned long 转换为 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Added in version 3.3.

在 3.5 版本发生变更: 当转换为字节视图时，源格式将不再受限。

还存在一些可用的只读属性:

obj

内存视图的下层对象:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Added in version 3.3.

nbytes

nbytes == product(shape) * itemsize == len(m.tobytes())。这是数组在连续表示时将会占用的空间总字节数。它不一定等于 len(m):

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

多维数组:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

Added in version 3.3.

readonly

一个表明内存是否只读的布尔值。

format

一个字符串，包含视图中每个元素的格式（表示为*struct* 模块样式）。内存视图可以从具有任意格式字符串的导出器创建，但某些方法 (例如*tolist()*) 仅限于原生的单元素格式。

在 3.3 版本发生变更: 格式 'B' 现在会按照 struct 模块语法来处理。这意味着 memoryview(b'abc')[0] == b'abc'[0] == 97。

itemsize

memoryview 中每个元素以字节表示的大小:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

一个整数，表示内存所代表的多维数组具有多少个维度。

shape

一个整数元组，通过*ndim* 的长度值给出内存所代表的 N 维数组的形状。

在 3.3 版本发生变更: 当 ndim = 0 时值为空元组而不再为 `None`。

strides

一个整数元组，通过*ndim* 的长度给出以字节表示的大小，以便访问数组中每个维度上的每个元素。

在 3.3 版本发生变更: 当 ndim = 0 时值为空元组而不再为 `None`。

suboffsets

供 PIL 风格的数组内部使用。该值仅作为参考信息。

c_contiguous

一个表明内存是否为 C-*contiguous* 的布尔值。

Added in version 3.3.

f_contiguous

一个表明内存是否为 Fortran *contiguous* 的布尔值。

Added in version 3.3.

contiguous

一个表明内存是否为*contiguous* 的布尔值。

Added in version 3.3.

4.10 集合类型 --- set, frozenset

set 对象是由具有唯一性的*hashable* 对象所组成的无序多项集。常见的用途包括成员检测、从序列中去除重复项以及数学中的集合类计算，例如交集、并集、差集与对称差集等等。(关于其他容器对象请参看*dict*, *list* 与*tuple* 等内置类，以及*collections* 模块。)

与其他多项集一样，集合也支持 `x in set`, `len(set)` 和 `for x in set`。作为一种无序的多项集，集合并不记录元素位置或插入顺序。相应地，集合不支持索引、切片或其他序列类的操作。

目前有两种内置集合类型，*set* 和*frozenset*。*set* 类型是可变的 --- 其内容可以使用 `add()` 和 `remove()` 这样的方法来改变。由于是可变类型，它没有哈希值，且不能被用作字典的键或其他集合的元素。*frozenset* 类型是不可变并且为*hashable* --- 其内容在被创建后不能再改变；因此它可以被用作字典的键或其他集合的元素。

除了可以使用*set* 构造器，非空的 set (不是 frozenset) 还可以通过将以逗号分隔的元素列表包含于花括号之内来创建，例如: `{'jack', 'sjoerd'}`。

两个类的构造器具有相同的作用方式:

class set([*iterable*])

class frozenset([*iterable*])

返回一个新的 set 或 frozenset 对象，其元素来自于 *iterable*。集合的元素必须为*hashable*。要表示由集合对象构成的集合，所有的内层集合必须为*frozenset* 对象。如果未指定 *iterable*，则将返回一个新的空集合。

集合可用多种方式来创建:

- 使用花括号内以逗号分隔元素的方式: `{'jack', 'sjoerd'}`
- 使用集合推导式: `{c for c in 'abracadabra' if c not in 'abc'}`
- 使用类型构造器: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

set 和*frozenset* 的实例提供以下操作:

len(s)

返回集合 *s* 中的元素数量（即 *s* 的基数)。

x in s

检测 *x* 是否为 *s* 中的成员。

x not in s

检测 *x* 是否非 *s* 中的成员。

isdisjoint(*other*)

如果集合中没有与 *other* 共有的元素则返回 `True`。当且仅当两个集合的交集为空集合时，两者为不相交集合。

issubset(*other*)

set <= other

检测是否集合中的每个元素都在 *other* 之中。

set < other

检测集合是否为 *other* 的真子集，即 `set <= other and set != other`。

issuperset(*other*)

set >= other

检测是否 *other* 中的每个元素都在集合之中。

set > other

检测集合是否为 *other* 的真超集，即 `set >= other and set != other`。

union(**others*)

set | other | ...

返回一个新集合，其中包含来自原集合以及 others 指定的所有集合中的元素。

intersection(**others*)

set & other & ...

返回一个新集合，其中包含原集合以及 others 指定的所有集合中共有的元素。

difference(**others*)

set - other - ...

返回一个新集合，其中包含原集合中在 others 指定的其他集合中不存在的元素。

symmetric_difference(*other*)

set ^ other

返回一个新集合，其中的元素或属于原集合或属于 *other* 指定的其他集合，但不能同时属于两者。

copy()

返回原集合的浅拷贝。

注意，*union()*、*intersection()*、*difference()*、*symmetric_difference()*、*issubset()* 和 *issuperset()* 方法的非运算符版本可以接受任何可迭代对象作为一个参数。相比之下，基于运算符的对应方法则要求参数为集合对象。这就避开了像 `set('abc') & 'cbs'` 这样容易出错的结构，而换成了可读性更好的 `set('abc').intersection('cbs')`。

set 和 *frozenset* 均支持集合与集合的比较。两个集合当且仅当每个集合中的每个元素均包含于另一个集合之内（即各为对方的子集）时则相等。一个集合当且仅当其为另一个集合的真子集（即为后者的子集但两者不相等）时则小于另一个集合。一个集合当且仅当其为另一个集合的真超集（即为后者的超集但两者不相等）时则大于另一个集合。

set 的实例与 *frozenset* 的实例之间基于它们的成员进行比较。例如 `set('abc') == frozenset('abc')` 返回 `True`，`set('abc') in set([frozenset('abc')])` 也一样。

子集与相等比较并不能推广为完全排序函数。例如，任意两个非空且不相交的集合不相等且互不为对方的子集，因此以下 所有比较均返回 `False`: `a<b`, `a==b`, or `a>b`。

由于集合仅定义了部分排序（子集关系），因此由集合构成的列表 *list.sort()* 方法的输出并无定义。

集合的元素，与字典的键类似，必须为*hashable*。

混合了*set* 实例与*frozenset* 的二进制位运算将返回与第一个操作数相同的类型。例如: `frozenset('ab') | set('bc')` 将返回*frozenset* 的实例。

下表列出了可用于*set* 而不能用于不可变的*frozenset* 实例的操作:

update(**others*)

set |= other | ...

更新集合，添加来自 others 中的所有元素。

intersection_update(**others*)

set &= other & ...

更新集合，只保留其中在所有 others 中也存在的元素。

difference_update(**others*)

set -= other | ...

更新集合，移除其中也存在于 others 中的元素。

symmetric_difference_update(*other*)

set ^= other

更新集合，只保留存在于集合的一方而非共同存在的元素。

add(*elem*)

将元素 *elem* 添加到集合中。

remove(*elem*)

从集合中移除元素 *elem*。如果 *elem* 不存在于集合中则会引发*KeyError*。

discard(*elem*)

如果元素 *elem* 存在于集合中则将其移除。

pop()

从集合中移除并返回任意一个元素。如果集合为空则会引发*KeyError*。

clear()

从集合中移除所有元素。

请注意，非运算符版本的*update()*，*intersection_update()*，*difference_update()* 和*symmetric_difference_update()* 方法将接受任意可迭代对象作为参数。

请注意，`__contains__()`，*remove()* 和*discard()* 方法的 *elem* 参数可以是一个集合。为支持搜索等价的冻结集合，将根据 *elem* 临时创建一个相应的对象。

4.11 映射类型 --- dict

mapping 对象会将*hashable* 值映射到任意对象。映射属于可变对象。目前仅有一种标准映射类型 字典。(关于其他容器对象请参看*list*, *set* 与*tuple* 等内置类，以及*collections* 模块。)

字典的键 几乎可以为任何值。不是*hashable* 的值，即包含列表、字典或其他可变类型（按值比较而非按对象标识比较）的值不可被用作键。比较结果相等的值（如 `1`, `1.0` 和 `True` 等）可被互换使用以索引同一个字典条目。

class dict(***kwargs*)

class dict(*mapping*, ***kwargs*)

class dict(*iterable*, ***kwargs*)

返回一个新的字典，基于可选的位置参数和可能为空的关键字参数集来初始化。

字典可用多种方式来创建:

- 使用花括号内以逗号分隔 键: 值对的方式: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`

- 使用字典推导式: `{}`, `{x: x ** 2 for x in range(10)}`
- 使用类型构造器: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

如果没有给出位置参数，将创建一个空字典。如果给出一个位置参数并且其定义了 `keys()` 方法，则通过在该参数上调用 `__getitem__()` 创建一个字典并包含从该方法返回的每个键。在其他情况下，位置参数必须是一个*iterable* 对象。该可迭代对象中的每一项本身必须是一个恰好包含两个元素的可迭代对象。每一项中的第一个元素将成为新字典的一个键，第二个元素将成为其对应的值。如果一个键出现多次，该键的最后一个值将成为其在新字典中的对应值。

如果给出了关键字参数，则关键字参数及其值会被加入到基于位置参数创建的字典。如果要加入的键已存在，来自关键字参数的值将替代来自位置参数的值。

作为演示，以下示例返回的字典均等于 `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

像第一个例子那样提供关键字参数的方式只能使用有效的 Python 标识符作为键。其他方式则可使用任何有效的键。

这些是字典所支持的操作（因而自定义的映射类型也应当支持）:

list(d)

返回字典 *d* 中使用的所有键的列表。

len(d)

返回字典 *d* 中的项数。

d[key]

返回 *d* 中以 *key* 为键的项。如果映射中不存在 *key* 则会引发*KeyError*。

如果字典的子类定义了方法 `__missing__()` 并且 *key* 不存在，则 `d[key]` 操作将调用该方法并附带键 *key* 作为参数。`d[key]` 随后将返回或引发 `__missing__(key)` 调用所返回或引发的任何对象或异常。没有其他操作或方法会唤起 `__missing__()`。如果未定义 `__missing__()`，则会引发*KeyError*。`__missing__()` 必须是一个方法；它不能是一个实例变量:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
...
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

上面的例子显示了*collections.Counter* 实现的部分代码。还有另一个不同的 `__missing__` 方法是由*collections.defaultdict* 所使用的。

d[key] = value

将 `d[key]` 设为 *value*。

del d[key]

将 `d[key]` 从 *d* 中移除。如果映射中不存在 *key* 则会引发*KeyError*。

key in d

如果 *d* 中存在键 *key* 则返回 `True`，否则返回 `False`。

key not in d

等价于 `not key in d`。

iter(d)

返回以字典的键为元素的迭代器。这是 `iter(d.keys())` 的快捷方式。

clear()

移除字典中的所有元素。

copy()

返回原字典的浅拷贝。

classmethod fromkeys(*iterable*, *value=None*, */*)

使用来自 *iterable* 的键创建一个新字典，并将键值设为 *value*。

fromkeys() 是一个返回新字典的类方法。*value* 默认为 `None`。所有值都只引用一个单独的实例，因此让 *value* 成为一个可变对象例如空列表通常是没有意义的。要获取不同的值，请改用字典推导式。

get(*key*, *default=None*, */*)

如果 *key* 存在于字典中则返回 *key* 的值，否则返回 *default*。如果 *default* 未给出则默认为 `None`，因而此方法绝不会引发*KeyError*。

items()

返回由字典项 ((键，值) 对) 组成的一个新视图。参见视图对象文档。

keys()

返回由字典键组成的一个新视图。参见视图对象文档。

pop(*key*[, *default*])

如果 *key* 存在于字典中则将其移除并返回其值，否则返回 *default*。如果 *default* 未给出且 *key* 不存在于字典中，则会引发*KeyError*。

popitem()

从字典中移除并返回一个 (键，值) 对。键值对会按 LIFO (后进先出) 的顺序被返回。

popitem() 适用于对字典进行消耗性的迭代，这在集合算法中经常被使用。如果字典为空，调用*popitem()* 将引发*KeyError*。

在 3.7 版本发生变更: 现在会确保采用 LIFO 顺序。在之前的版本中，*popitem()* 会返回一个任意的键/值对。

reversed(d)

返回一个逆序获取字典键的迭代器。这是 `reversed(d.keys())` 的快捷方式。

Added in version 3.8.

setdefault(*key*, *default=None*, */*)

如果字典存在键 *key* ，返回它的值。如果不存在，插入值为 *default* 的键 *key* ，并返回 *default* 。*default* 默认为 `None`。

update([*other*])

使用来自 *other* 的键/值对更新字典，覆盖原有的键。返回 `None`。

update() 接受另一个具有 `keys()` 方法的对象（在此情况下 `__getitem__()` 将被调用并附带从该方法返回的键）或一个包含键/值对（以长度为二的元组或其他可迭代对象表示）的可迭代对象。如果指定了关键字参数，则会以其所对应的键/值对更新字典: `d.update(red=1, blue=2)`。

values()

返回由字典值组成的一个新视图。参见视图对象文档。

两个 dict.values() 视图之间的相等性比较将总是返回 False。这在 dict.values() 与其自身比较时也同样适用:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

d | other

合并 *d* 和 *other* 中的键和值来创建一个新的字典，两者必须都是字典。当 *d* 和 *other* 有相同键时，*other* 的值优先。

Added in version 3.9.

d |= other

用 *other* 的键和值更新字典 *d* ，*other* 可以是*mapping* 或*iterable* 的键值对。当 *d* 和 *other* 有相同键时，*other* 的值优先。

Added in version 3.9.

两个字典的比较当且仅当它们具有相同的（键，值）对时才会相等（不考虑顺序）。排序比较 ('<', '<=', '>=', '>') 会引发*TypeError*。

字典会保留插入时的顺序。请注意对键的更新不会影响顺序。删除并再次添加的键将被插入到末尾。

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

在 3.7 版本发生变更: 字典顺序会确保为插入顺序。此行为是自 3.6 版开始的 CPython 实现细节。

字典和字典视图都是可逆的。

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

在 3.8 版本发生变更: 字典现在是可逆的。

参见

types.MappingProxyType 可被用来创建一个*dict* 的只读视图。

4.11.1 字典视图对象

由*dict.keys()*, *dict.values()* 和*dict.items()* 所返回的对象是 视图对象。该对象提供字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。

字典视图可以被迭代以产生与其对应的数据，并支持成员检测：

len(dictview)

返回字典中的条目数。

iter(dictview)

返回字典中的键、值或项（以（键，值）为元素的元组表示）的迭代器。

键和值是按插入时的顺序进行迭代的。这样就允许使用*zip()* 来创建（值，键）对: `pairs = zip(d.values(), d.keys())`。另一个创建相同列表的方式是 `pairs = [(v, k) for (k, v) in d.items()]`.

在添加或删除字典中的条目期间对视图进行迭代可能引发*RuntimeError* 或者无法完全迭代所有条目。

在 3.7 版本发生变更: 字典顺序会确保为插入顺序。

x in dictview

如果 *x* 是对应字典中存在的键、值或项（在最后一种情况下 *x* 应为一个（键，值）元组）则返回 `True`。

reversed(dictview)

返回一个逆序获取字典键、值或项的迭代器。视图将按与插入时相反的顺序进行迭代。

在 3.8 版本发生变更: 字典视图现在是可逆的。

dictview.mapping

返回*types.MappingProxyType* 对象，封装了字典视图指向的原始字典。

Added in version 3.10.

键视图与集合类似因为其条目是唯一的并且为*hashable*。条视图也有类似集合的操作因为 (键, 值) 对是唯一的并且键是可哈希的。如果条目视图中的所有值也都是可哈希的，那么条目视图就可以与其他集合执行互操作。(值视图不会被认为与集合类似因为条目通常不是唯一的)。对于与集合类似的视图，可以使用为抽象基类*collections.abc.Set* 定义的所有操作（例如，==, < 或 ^ 等）。虽然使用了集合运算符，但与集合类似的视图接受任何可迭代对象作为其操作数，而不像集合那样只接受集合作为输入。

一个使用字典视图的示例:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # 迭代
>>> n = 0
>>> for val in values:
...     n += val
...
>>> print(n)
504

>>> # 键和值将以相同顺序（插入顺序）被迭代
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # 视图对象是动态的并会反映字典的改变
>>> del dishes['eggs']
>>> del dishes['sausage']
```

(续下页)

(接上页)

```
>>> list(keys)
['bacon', 'spam']

>>> # 集合运算
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'} == {'juice', 'sausage', 'bacon', 'spam'}
True
>>> keys | ['juice', 'juice', 'juice'] == {'bacon', 'spam', 'juice'}
True

>>> # 获取原始字典的只读代理
>>> values.mapping
mappingproxy({'bacon': 1, 'spam': 500})
>>> values.mapping['spam']
500
```

4.12 上下文管理器类型

Python 的 `with` 语句支持通过上下文管理器所定义的运行时上下文这一概念。此对象的实现使用了一对专门方法，允许用户自定义类来定义运行时上下文，在语句体被执行前进入该上下文，并在语句执行完毕时退出该上下文：

contextmanager.**__enter__**()

进入运行时上下文并返回此对象或关联到该运行时上下文的其他对象。此方法的返回值会绑定到使用此上下文管理器的 `with` 语句的 `as` 子句中的标识符。

一个返回其自身的上下文管理器的例子是*file object*。文件对象会从 __enter__() 返回其自身，以允许*open()* 被用作 `with` 语句中的上下文表达式。

一个返回关联对象的上下文管理器的例子是*decimal.localcontext()* 所返回的对象。此种管理器会将活动的 decimal 上下文设为原始 decimal 上下文的一个副本并返回该副本。这允许对 `with` 语句的语句体中的当前 decimal 上下文进行更改，而不会影响 `with` 语句以外的代码。

contextmanager.**__exit__**(*exc_type*, *exc_val*, *exc_tb*)

退出运行时上下文并返回一个布尔值旗标来表明所发生的任何异常是否应当被屏蔽。如果在执行 `with` 语句的语句体期间发生了异常，则参数会包含异常的类型、值以及回溯信息。在其他情况下三个参数均为 `None`。

自此方法返回一个真值将导致 `with` 语句屏蔽异常并继续执行紧随在 `with` 语句之后的语句。否则异常将在此方法结束执行后继续传播。在此方法执行期间发生的异常将会取代 `with` 语句的语句体中发生的任何异常。

传入的异常绝对不应当被显式地重新引发——相反地，此方法应当返回一个假值以表明方法已成功完成并且不希望屏蔽被引发的异常。这允许上下文管理代码方便地检测 __exit__() 方法是否确实已失败。

Python 定义了一些上下文管理器来支持简易的线程同步、文件或其他对象的快速关闭，以及更方便地操作活动的十进制算术上下文。除了实现上下文管理协议以外，不同类型不会被特殊处理。请参阅*contextlib* 模块查看相关的示例。

Python 的*generator* 和*contextlib.contextmanager* 装饰器提供了实现这些协议的便捷方式。如果使用*contextlib.contextmanager* 装饰器来装饰一个生成器函数，它将返回一个实现了必要的*__enter__()* 和*__exit__()* 方法的上下文管理器，而不再是由未经装饰的生成器所产生的迭代器。

请注意，Python/C API 中 Python 对象的类型结构中并没有针对这些方法的专门槽位。想要定义这些方法的扩展类型必须将它们作为普通的 Python 可访问方法来提供。与设置运行时上下文的开销相比，单个类字典查找的开销可以忽略不计。

4.13 类型注解的类型 --- Generic Alias 、Union

type annotations 的内置类型为*Generic Alias* 和*Union*。

4.13.1 GenericAlias 类型

`GenericAlias` 对象通常是通过 抽取一个类来创建的。它们最常被用于 容器类，如*list* 或*dict*。举例来说，`list[int]` 这个 `GenericAlias` 对象是通过附带*int* 参数抽取 `list` 类来创建的。`GenericAlias` 对象的主要目的是用于类型标注。

> **备注**
>
> 通常一个类只有在实现了特殊方法 `__class_getitem__()` 时才支持抽取操作。

`GenericAlias` 对象可作为*generic type* 的代理，实现了 *形参化泛型*。

对于一个容器类，提供给类的 抽取操作的参数可以指明对象所包含的元素类型。例如，`set[bytes]` 可在类型标注中用来表示一个*set* 中的所有元素均为*bytes* 类型。

对于一个定义了 `__class_getitem__()` 但不属于容器的类，提供给类的抽取操作的参数往往会指明在对象上定义的一个或多个方法的返回值类型。例如，正则表达式 可以被用在*str* 数据类型和*bytes* 数据类型上:

- 如果 `x = re.search('foo', 'foo')`，则 `x` 将为一个*re.Match* 对象而 `x.group(0)` 和 `x[0]` 的返回值将均为*str* 类型。我们可以在类型标注中使用 `GenericAlias` `re.Match[str]` 来代表这种对象。
- 如果 `y = re.search(b'bar', b'bar')`，(注意 `b` 表示*bytes*)，则 `y` 也将为一个 `re.Match` 的实例，但 `y.group(0)` 和 `y[0]` 的返回值将均为*bytes* 类型。在类型标注中，我们将使用 `re.Match[bytes]` 来代表这种形式的*re.Match* 对象。

`GenericAlias` 对象是*types.GenericAlias* 类的实例，该类也可被用来直接创建 `GenericAlias` 对象。

`T[X, Y, ...]`

创建一个代表由类型 *X*, *Y* 来参数化的类型 `T` 的 `GenericAlias`，此类型会更依赖于所使用的 `T`。例如，一个接受包含*float* 元素的*list* 的函数:

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

另一个例子是关于*mapping* 对象的，用到了*dict*，泛型的两个类型参数分别代表了键类型和值类型。本例中的函数需要一个 `dict`，其键的类型为*str*，值的类型为*int*:。

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

内置函数*isinstance()* 和*issubclass()* 不接受第二个参数为 `GenericAlias` 类型:

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

Python 运行时不会强制执行类型标注。这种行为扩展到了泛型及其类型形参。当由 `GenericAlias` 创建容器对象时，并不会检查容器中为元素指定的类型。例如，以下代码虽然不被鼓励，但运行时并不会报错:

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

不仅如此，在创建对象的过程中，应用了参数后的泛型还会抹除类型参数:

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

在泛型上调用 *repr()* 或 *str()* 会显示应用参数之后的类型：

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

调用泛型容器的 __getitem__() 方法将引发异常以防出现 dict[str][str] 之类的错误：

```
>>> dict[str][str]
Traceback (most recent call last):
  ...
TypeError: dict[str] is not a generic class
```

不过，当使用了类型变量 时这种表达式是无效的。索引必须有与 GenericAlias 对象的 *__args__* 中的类型变量条目数量相当的元素。

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
>>> dict[str, Y][int]
dict[str, int]
```

标准泛型类

下列标准库类支持形参化的泛型。此列表并不是详尽无遗的。

- *tuple*
- *list*
- *dict*
- *set*
- *frozenset*
- *type*
- *asyncio.Future*
- *asyncio.Task*
- *collections.deque*
- *collections.defaultdict*
- *collections.OrderedDict*
- *collections.Counter*
- *collections.ChainMap*
- *collections.abc.Awaitable*
- *collections.abc.Coroutine*
- *collections.abc.AsyncIterable*
- *collections.abc.AsyncIterable*

- *collections.abc.AsyncGenerator*
- *collections.abc.Iterable*
- *collections.abc.Iterator*
- *collections.abc.Generator*
- *collections.abc.Reversible*
- *collections.abc.Container*
- *collections.abc.Collection*
- *collections.abc.Callable*
- *collections.abc.Set*
- *collections.abc.MutableSet*
- *collections.abc.Mapping*
- *collections.abc.MutableMapping*
- *collections.abc.Sequence*
- *collections.abc.MutableSequence*
- *collections.abc.ByteString*
- *collections.abc.MappingView*
- *collections.abc.KeysView*
- *collections.abc.ItemsView*
- *collections.abc.ValuesView*
- *contextlib.AbstractContextManager*
- *contextlib.AbstractAsyncContextManager*
- *dataclasses.Field*
- *functools.cached_property*
- *functools.partialmethod*
- *os.PathLike*
- *queue.LifoQueue*
- *queue.Queue*
- *queue.PriorityQueue*
- *queue.SimpleQueue*
- *re.Pattern*
- *re.Match*
- *shelve.BsdDbShelf*
- *shelve.DbfilenameShelf*
- *shelve.Shelf*
- *types.MappingProxyType*
- *weakref.WeakKeyDictionary*
- *weakref.WeakMethod*
- *weakref.WeakSet*
- *weakref.WeakValueDictionary*

GenericAlias 对象的特殊属性

应用参数后的泛型都实现了一些特殊的只读属性：

genericalias.**__origin__**

本属性指向未应用参数之前的泛型类：

```
>>> list[int].__origin__
<class 'list'>
```

genericalias.**__args__**

该属性是传给泛型类的原始 `__class_getitem__()` 的泛型所组成的*tuple*（长度可能为 1）：

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

genericalias.**__parameters__**

该属性是延迟计算出来的一个元组（可能为空），包含了 `__args__` 中的类型变量。

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

> **备注**
>
> 带有参数*typing.ParamSpec*的 GenericAlias 对象，在类型替换后其 `__parameters__` 可能会不准确，因为*typing.ParamSpec*主要用于静态类型检查。

genericalias.**__unpacked__**

一个布尔值，如果别名已使用 `*` 运算符进行解包则为真值（参见*TypeVarTuple*）。

Added in version 3.11.

> **参见**
>
> **PEP 484 ——类型注解**
> 介绍 Python 中用于类型标注的框架。
>
> **PEP 585 - 标准多项集中的类型提示泛型**
> 介绍了对标准库类进行原生形参化的能力，只要它们实现了特殊的类方法 `__class_getitem__()`。
>
> **泛型（*Generic*），用户自定义泛型 和*typing.Generic***
> 有关如何实现可在运行时被形参化并能被静态类型检查器所识别的泛用类的文档。

Added in version 3.9.

4.13.2 union 类型

联合对象包含了在多个类型对象上执行 `|`（按位或）运算后的值。这些类型主要用于类型标注。与*typing.Union*相比，联合类型表达式可以实现更简洁的类型提示语法。

X | Y | ...

定义包含了 *X*、*Y* 等类型的 union 对象。`X | Y` 表示 X 或 Y。相当于 `typing.Union[X, Y]`。比如以下函数的参数应为类型*int*或*float*：

```
def square(number: int | float) -> int | float:
    return number ** 2
```

备注

不可在运行时使用 | 操作数来定义有一个或多个成员为前向引用的并集。例如，`int | "Foo"`，其中 `"Foo"` 是指向某个尚未定义的类的引用，在运行时将会失败。对于包括前向引用的并集，请将整个表达式用字符串来表示，例如 `"int | Foo"`。

union_object == other

union 对象可与其他 union 对象进行比较。详细结果如下：

- 多次组合的结果会平推：

```
(int | str) | float == int | str | float
```

- 冗余的类型会被删除：

```
int | str | int == int | str
```

- 在相互比较时，会忽略顺序：

```
int | str == str | int
```

- 与 `typing.union` 兼容：

```
int | str == typing.Union[int, str]
```

- Optional 类型可表示为与 `None` 的组合。

```
str | None == typing.Optional[str]
```

isinstance(obj, union_object)

issubclass(obj, union_object)

isinstance() 和*issubclass()* 也支持 union 对象：

```
>>> isinstance("", int | str)
True
```

但是联合对象中的*参数化泛型* 将无法被检测：

```
>>> isinstance(1, int | list[int])  # 短路求值
True
>>> isinstance([1], int | list[int])
Traceback (most recent call last):
  ...
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

union 对象构成的用户类型可以经由*types.UnionType* 访问，并可用于*isinstance()* 检查。而不能由类型直接实例化为对象：

```
>>> import types
>>> isinstance(int | str, types.UnionType)
True
>>> types.UnionType()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'types.UnionType' instances
```

备注

为了支持 `X | Y` 语法，类型对象加入了 `__or__()` 方法。如果一个元类实现了 `__or__()`，Union 可以重载它：

```
>>> class M(type):
...     def __or__(self, other):
...         return "Hello"
...
>>> class C(metaclass=M):
...     pass
...
>>> C | int
'Hello'
>>> int | C
int | C
```

参见

PEP 604 ——提出了 `X | Y` 语法和 union 类型。

Added in version 3.10.

4.14 其他内置类型

解释器支持一些其他种类的对象。这些对象大都仅支持一两种操作。

4.14.1 模块

模块唯一的特殊操作是属性访问: `m.name`，这里 *m* 为一个模块而 *name* 访问定义在 *m* 的符号表中的一个名称。模块属性可以被赋值。（请注意 `import` 语句严格来说也是对模块对象的一种操作；`import foo` 不要求存在一个名为 *foo* 的模块对象，而是要求存在一个对于名为 *foo* 的模块的 (永久性) 定义。）

每个模块都有一个特殊属性 `__dict__`。这是包含模块的符号表的字典。修改此字典将实际改变模块的符号表，但是无法直接对 `__dict__` 赋值 (你可以写 `m.__dict__['a'] = 1`，这会将 `m.a` 定义为 `1`，但是你不能写 `m.__dict__ = {}`)。不建议直接修改 `__dict__`。

内置于解释器中的模块会写成这样: `<module 'sys' (built-in)>`。如果是从一个文件加载，则会写成 `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`。

4.14.2 类与类实例

关于这些类型请参阅 objects 和 class。

4.14.3 函数

函数对象是通过函数定义创建的。对函数对象的唯一操作是调用它: `func(argument-list)`。

实际上存在两种不同的函数对象：内置函数和用户自定义函数。两者支持同样的操作（调用函数），但实现方式不同，因此对象类型也不同。

更多信息请参阅 function。

4.14.4 方法

方法是使用属性表示法来调用的函数。存在两种形式: 内置方法 (如列表的 append()) 和 类实例方法。内置方法由支持它们的类型来描述。

如果你通过一个实例来访问方法（即定义在类命名空间内的函数），你会得到一个特殊对象: 绑定方法 (或称 实例方法) 对象。当被调用时，它会将 self 参数添加到参数列表。绑定方法具有两个特殊的只读属性: m.__self__ 操作该方法的对象，而 m.__func__ 是实现该方法的函数。调用 m(arg-1, arg-2, ..., arg-n) 完全等价于调用 m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)。

与 函数对象类似，绑定方法对象也支持获取任意属性。但是，由于方法属性实际上保存于下层的函数对象中 (method.__func__)，因此不允许设置绑定方法的方法属性。尝试设置方法的属性将会导致引发*AttributeError*。想要设置方法属性，你必须在下层的函数对象中显式地设置它。

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

请参阅 instance-methods 了解更多信息。

4.14.5 代码对象

代码对象被具体实现用来表示“伪编译”的可执行 Python 代码例如一个函数体。它们不同于函数对象，因为它们不包含对其全局执行环境的引用。代码对象由内置的*compile()* 函数返回，并可通过函数对象的 __code__ 属性来提取。另请参阅*code* 模块。

访问 __code__ 会引发一个审计事件 object.__getattr__，并附带参数 obj 和 "__code__"。

可以通过将代码对象（而非源码字符串）传给*exec()* 或*eval()* 内置函数来执行或求值。

更多信息请参阅 types。

4.14.6 类型对象

类型对象表示各种对象类型。对象的类型可通过内置函数*type()* 来获取。类型没有特殊的操作。标准库模块*types* 定义了所有标准内置类型的名称。

类型以这样的写法来表示: <class 'int'>。

4.14.7 空对象

此对象会由不显式地返回值的函数所返回。它不支持任何特殊的操作。空对象只有一种值 None (这是个内置名称)。type(None)() 会生成同一个单例。

该对象的写法为 None。

4.14.8 省略符对象

此对象常被用于切片 (参见 slicings)。它不支持任何特殊的操作。省略符对象只有一种值*Ellipsis* (这是个内置名称)。type(Ellipsis)() 会生成*Ellipsis* 单例。

该对象的写法为 Ellipsis 或 ...。

4.14.9 未实现对象

此对象会被作为比较和二元运算被应用于它们所不支持的类型时的返回值。请参阅 comparisons 了解更多信息。未实现对象只有一种值*NotImplemented*。`type(NotImplemented)()` 会生成这个单例。

其写法为 `NotImplemented`。

4.14.10 内部对象

相关信息请参阅 types。其中描述了 栈帧对象, 回溯对象以及切片对象等。

4.15 特殊属性

语言实现为部分对象类型添加了一些特殊的只读属性，它们具有各自的作用。其中一些并不会被*dir()*内置函数所列出。

`definition.`**`__name__`**
: 类、函数、方法、描述器或生成器实例的名称。

`definition.`**`__qualname__`**
: 类、函数、方法、描述器或生成器实例的*qualified name*。

  Added in version 3.3.

`definition.`**`__module__`**
: 类或函数定义所在的模块的名称。

`definition.`**`__doc__`**
: 类或函数的文档字符串，如果未定义则为 `None`。

`definition.`**`__type_params__`**
: 泛型类、函数和*类型别名*的 类型形参。对于非泛型类和函数，这将为空元组。

  Added in version 3.12.

4.16 整数字符串转换长度限制

CPython 对于*int* 和*str* 之间的转换有一个全局限制以缓解拒绝服务攻击。此限制 仅会作用于十进制或其他以非二的乘方为基数的数字。十六进制、八进制和二进制转换不受限制。该限制可以被配置。

int 类型在 CPython 中是存储为二进制形式的任意长度的数字（通常称为 “大数字”）。不存在可在线性时间内将一个字符串转换为二进制整数或将一个二进制整数转换为字符串的算法，除非基数为 2 的乘方。对于基数为 10 来说已知最好的算法也有亚二次方复杂度。转换一个大数值如 `int('1' * 500_000)` 在快速的 CPU 上也会花费一秒以上的时间。

限制转换大小是一项避免 **CVE 2020-10735** 的务实解决方式。

此限制会在可能涉及非线性转换算法时作用于输入或输出字符串中的数字型字符数量。下划线和正负号不计入限制数量。

当一个操作会超出限制时，将引发*ValueError*:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # 含义如名称所示，这是默认值。
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion: value has 5432␣
↪digits; use sys.set_int_max_str_digits() to increase the limit
>>> i = int('2' * 4300)
>>> len(str(i))
4300
```

(续下页)

(接上页)

```
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion; use sys.set_int_
↪max_str_digits() to increase the limit
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # 十六进制数没有限制。
```

默认限制为 4300 位即*sys.int_info.default_max_str_digits* 的值。最低限制可被配置为 640 位即*sys.int_info.str_digits_check_threshold*。

验证:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

Added in version 3.11.

4.16.1 受影响的 API

此限制仅会作用于*int* 和*str* 和*bytes* 之间存在速度变慢可能的转换:

- `int(string)` 默认以 10 为基数。
- `int(string, base)` 用于所有不为 2 的乘方的基数。
- `str(integer)`。
- `repr(integer)`。
- 任何其他目标是以 10 为基数的字符串转换，例如 `f"{integer}"`, `"{}".format(integer)` 或 `b"%d" % integer`。

此限制不会作用于使用线性算法的函数:

- `int(string, base)` 中 base 可以为 2, 4, 8, 16 或 32。
- *int.from_bytes()* 和*int.to_bytes()*。
- *hex()*, *oct()*, *bin()*。
- 格式规格迷你语言 用于十六进制、八进制和二进制数。
- *str* 至*float*。
- *str* 至*decimal.Decimal*。

4.16.2 配置限制值

在 Python 启动之前你可以使用环境变量或解释器命令行旗标来配置限制值:

- `PYTHONINTMAXSTRDIGITS`，例如 `PYTHONINTMAXSTRDIGITS=640 python3` 是将限制设为 640 而 `PYTHONINTMAXSTRDIGITS=0 python3` 是禁用此限制。
- `-X int_max_str_digits`，例如`python3 -X int_max_str_digits=640`
- *sys.flags.int_max_str_digits* 包含 `PYTHONINTMAXSTRDIGITS` 或 `-X int_max_str_digits` 的值。如果环境变量和 `-X` 选项均有设置，则 `-X` 选项优先。值为 *-1* 表示两者均未设置，因此会在初始化时使用*sys.int_info.default_max_str_digits* 的值。

从代码中，你可以检查当前的限制并使用这些*sys* API 来设置新值:

- *sys.get_int_max_str_digits()* 和*sys.set_int_max_str_digits()* 是解释器级限制的读取器和设置器。子解释器具有它们自己的限制。

有关默认值和最小值的信息可在*sys.int_info* 中找到:

- *sys.int_info.default_max_str_digits* 是已编译的默认限制。
- *sys.int_info.str_digits_check_threshold* 是该限制可接受的最低值（禁用该限制的 0 除外）。

Added in version 3.11.

小心

设置较低的限制值 可能导致问题。虽然不常见，但还是会有在其源代码中包含超出最小阈值的十进制整数常量的代码存在。设置此限制的一个后果将是包含比此限制长的十进制整数字面值的 Python 源代码将在解析期间遇到错误，通常是在启动时或导入时甚至是在安装时——只要对于某个代码还不存在已更新的 `.pyc` 就会发生。一种在包含此类大数值常量的源代码中绕过该问题的办法是将它们转换为不受限制的 `0x` 十六进制形式。

如果你使用了较低的限制则请要彻底地测试你的应用程序。确保你的测试通过环境变量或旗标尽早设置该限制来运行以便在启动期间甚至是在可能唤起 Python 来将 `.py` 源文件预编译为 `.pyc` 文件的任何安装步骤其间应用该限制。

4.16.3 推荐配置

默认的*sys.int_info.default_max_str_digits* 被预期对于大多数应用程序来说都是合理的。如果你的应用程序需要不同的限制值，请使用不预设 Python 版本的代码从你的主入口点进行设置，因为这些 API 是在 3.12 之前的版本所发布的安全补丁中添加的。

示例:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

如果你需要完全禁用它，请将其设为 `0`。

备注

CHAPTER 5

内置异常

在 Python 中，所有异常必须为一个派生自 *BaseException* 的类的实例。在带有提及一个特定类的 except 子句的 try 语句中，该子句也会处理任何派生自该类的异常类（但不处理 它所派生出的异常类）。通过子类化创建的两个不相关异常类永远是不等效的，既使它们具有相同的名称。

本章中列出的内置异常可由解释器或内置函数来生成。除非另有说明，它们都会具有一个提示导致错误详细原因的“关联值”。这可以是一个字符串或由多个信息项（例如一个错误码和一个解释该错误码的字符串）。关联值通常会作为参数被传给异常类的构造器。

用户代码可以引发内置异常。这可被用于测试异常处理程序或报告错误条件，“就像”在解释器引发了相同异常的情况时一样；但是请注意，没有任何机制能防止用户代码引发不适当的错误。

内置异常类可以被子类化以定义新的异常；鼓励程序员从 *Exception* 类或它的某个子类而不是从 *BaseException* 来派生新的异常。关于定义异常的更多信息可以在 Python 教程的 tut-userexceptions 部分查看。

5.1 异常上下文

异常对象上的三个属性提供了有关引发异常所在上下文的信息：

BaseException.**__context__**

BaseException.**__cause__**

BaseException.**__suppress_context__**

当有其他异常已经被处理的情况下又引发一个新异常的时候，新异常的 __context__ 属性会被自动设为已经被处理的异常。异常可以在使用了 except 或 finally 子句，或者 with 语句的时候被处理。

这个隐式异常上下文可以通过使用 from 配合 raise 来补充一个显式的原因：

```
raise new_exc from original_exc
```

跟在 from 之后的表达式必须为一个异常或 None。它将在所引发的异常上被设为 __cause__。设置 __cause__ 还会隐式地将 __suppress_context__ 属性设为 True，这样使用 raise new_exc from None 可以有效地将旧异常替换为新异常来显示其目的 (例如将 *KeyError* 转换为 *AttributeError*)，同时让旧异常在 __context__ 中保持可用以便在调试时执行内省。

除了异常本身的回溯以外，默认的回溯还会显示这些串连的异常。__cause__ 中的显式串连异常如果存在将总是显示。__context__ 中的隐式串连异常仅在 __cause__ 为 *None* 且 __suppress_context__ 为假值时显示。

不论在哪种情况下，异常本身总会在任何串连异常之后显示，以便回溯的最后一行总是显示所引发的最后一个异常。

5.2 从内置异常继承

用户代码可以创建继承自某个异常类型的子类。建议每次仅子类化一个异常类型以避免多个基类处理 `args` 属性的不同方式，以及内存布局不兼容可能导致的冲突。

大多数内置异常都用 C 实现以保证运行效率，参见: Objects/exceptions.c。其中一些具有自定义内存布局，这使得创建继承自多个异常类型的子类成为不可能。一个类型的内存布局属于实现细节并可能随着 Python 版本升级而改变，导致在未来可能产生新的冲突。因此，建议完全避免子类化多个异常类型。

5.3 基类

下列异常主要被用作其他异常的基类。

exception BaseException

所有内置异常的基类。它不应该被用户自定义类直接继承 (这种情况请使用*Exception*)。如果在此类的实例上调用*str()*，则会返回实例的参数表示，或者当没有参数时返回空字符串。

args

传给异常构造器的参数元组。某些内置异常 (例如*OSError*) 接受特定数量的参数并赋予此元组中的元素特殊的含义，而其他异常通常只接受一个给出错误信息的单独字符串。

with_traceback(*tb*)

此方法会将 *tb* 设为新的异常回溯信息并返回异常对象。它在 **PEP 3134** 的异常链特性可用之前更为常用。下面的例子演示了我们如何将一个 `SomeException` 实例转换为 `OtherException` 实例而保留回溯信息。异常一旦被引发，当前帧会被推至 `OtherException` 的回溯栈顶端，就像当我们允许原始 `SomeException` 被传播给调用方时它的回溯栈将会发生的情形一样。:

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

__traceback__

保存关联到该异常的 回溯对象的可写字段。另请参阅: raise。

add_note(*note*)

将字符串 `note` 添加到在异常字符串之后的标准回溯中显示的注释中。如果 `note` 不是一个字符串则会引发*TypeError*。

Added in version 3.11.

__notes__

由此异常的注释组成的列表，它是通过*add_note()* 添加的。该属性是在调用*add_note()* 时创建的。

Added in version 3.11.

exception Exception

所有内置的非系统退出类异常都派生自此类。所有用户自定义异常也应当派生自此类。

exception ArithmeticError

此基类用于派生针对各种算术类错误而引发的内置异常: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*。

exception BufferError

当与 缓冲区相关的操作无法执行时将被引发。

exception LookupError

此基类用于派生当映射或序列所使用的键或索引无效时引发的异常: *IndexError*, *KeyError*。这可以通过*codecs.lookup()* 来直接引发。

5.4 具体异常

以下异常属于经常被引发的异常。

exception AssertionError

当 `assert` 语句失败时将被引发。

exception AttributeError

当属性引用 (参见 attribute-references) 或赋值失败时将被引发。(当一个对象根本不支持属性引用或属性赋值时则将引发*TypeError*。)

`name` 和 `obj` 属性可以使用构造器的仅限关键字参数来设置。它们如果被设置则分别代表要尝试访问的属性名称以及所访问的该属性的对象。

在 3.10 版本发生变更: 增加了 `name` 和 `obj` 属性。

exception EOFError

当*input()* 函数未读取任何数据即达到文件结束条件 (EOF) 时将被引发。(另外，`io.IOBase.read()` 和*io.IOBase.readline()* 方法在遇到 EOF 则将返回一个空字符串。)

exception FloatingPointError

目前未被使用。

exception GeneratorExit

当一个*generator* 或*coroutine* 被关闭时将被引发；参见 `generator.close()` 和 `coroutine.close()`。它直接继承自*BaseException* 而不是*Exception*，因为从技术上来说它并不是一个错误。

exception ImportError

当 `import` 语句尝试加载模块遇到麻烦时将被引发。并且当 `from ... import` 中的"from list" 存在无法找到的名称时也会被引发。

可选的 *name* 和 *path* 仅限关键字参数设置相应的属性:

name

尝试导入的模块的名称。

path

指向任何触发异常的文件的路径。

在 3.3 版本发生变更: 添加了*name* 与*path* 属性。

exception ModuleNotFoundError

ImportError 的子类，当一个模块无法被定位时将由 `import` 引发。当在*sys.modules* 中找到 `None` 时也会被引发。

Added in version 3.6.

exception IndexError

当序列抽取超出范围时将被引发。(切片索引会被静默截短到允许的范围；如果指定索引不是整数则*TypeError* 会被引发。)

exception KeyError

当在现有键集合中找不到指定的映射（字典）键时将被引发。

exception KeyboardInterrupt

当用户按下中断键 (通常为 `Control-C` 或 `Delete`) 时将被引发。在执行期间，会定期检测中断信号。该异常继承自*BaseException* 以确保不会被处理*Exception* 的代码意外捕获，这样可以避免退出解释器。

> **备注**
>
> 捕获*KeyboardInterrupt* 需要特别考虑。因为它可能会在不可预知的点位被引发，在某些情况下，它可能使运行中的程序陷入不一致的状态。通常最好是让*KeyboardInterrupt* 尽快结束程序或者完全避免引发它。(参见*有关信号处理器和异常的注释*。)

exception MemoryError

当一个操作耗尽内存但情况仍可（通过删除一些对象）进行挽救时将被引发。关联的值是一个字符串，指明是哪种（内部）操作耗尽了内存。请注意由于底层的内存管理架构（C 的 `malloc()` 函数），解释器也许并不总是能够从这种情况下完全恢复；但它毕竟可以引发一个异常，这样就能打印出栈回溯信息，以便找出导致问题的失控程序。

exception NameError

当某个局部或全局名称未找到时将被引发。此异常仅用于非限定名称。关联的值是一条错误信息，其中包含未找到的名称。

`name` 属性可以使用构造器的仅限关键字参数来设置。它如果被设置则代表要尝试访问的变量名称。

在 3.10 版本发生变更: 增加了 `name` 属性。

exception NotImplementedError

此异常派生自*RuntimeError*。在用户自定义的基类中，抽象方法应当在其要求所派生类重写该方法，或是在其要求所开发的类提示具体实现尚待添加时引发此异常。

> **备注**
>
> 它不应当用来表示一个运算符或方法根本不能被支持 -- 在此情况下应当让特定运算符 / 方法保持未定义，或者在子类中将其设为*None*。

> **小心**
>
> `NotImplementedError` 和 `NotImplemented` 不能互相替代。此异常应当仅以上文所描述的方式使用；请参阅*NotImplemented* 了解正确使用该内置常量的相关细节。

exception OSError(*[arg]*)

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

此异常在一个系统函数返回系统相关的错误时将被引发，此类错误包括 I/O 操作失败例如” 文件未找到” 或” 磁盘已满” 等（不包括非法参数类型或其他偶然性错误）。

构造器的第二种形式可设置如下所述的相应属性。如果未指定这些属性则默认为*None*。为了能向下兼容，如果传入了三个参数，则*args* 属性将仅包含由前两个构造器参数组成的 2 元组。

构造器实际返回的往往是*OSError* 的某个子类，如下文*OS exceptions* 中所描述的。具体的子类取决于最终的*errno* 值。此行为仅在直接或通过别名来构造*OSError* 时发生，并且在子类化时不会被继承。

errno

来自于 C 变量 `errno` 的数字错误码。

winerror

在 Windows 下，此参数将给出原生的 Windows 错误码。而*errno* 属性将是该原生错误码在 POSIX 平台下的近似转换形式。

在 Windows 下，如果 *winerror* 构造器参数是一个整数，则*errno* 属性会根据 Windows 错误码来确定，而 *errno* 参数会被忽略。在其他平台上，*winerror* 参数会被忽略，并且*winerror* 属性将不存在。

strerror

操作系统所提供的相应错误信息。它在 POSIX 平台中由 C 函数 `perror()` 来格式化，在 Windows 中则是由 `FormatMessage()`。

filename

filename2

对于与文件系统路径有关 (例如*open()* 或*os.unlink()*) 的异常，*filename* 是传给函数的文件名。对于涉及两个文件系统路径的函数 (例如*os.rename()*)，*filename2* 将是传给函数的第二个文件名。

在 3.3 版本发生变更: *EnvironmentError*, *IOError*, *WindowsError*, *socket.error*, *select.error* 与 `mmap.error` 已被合并到*OSError*，构造器可能返回其中一个子类。

在 3.4 版本发生变更: *filename* 属性现在是传给函数的原始文件名，而不是基于*filesystem encoding and error handler* 进行编码或解码之后的名称。此外，还添加了 *filename2* 构造器参数和属性。

exception OverflowError

当算术运算的结果大到无法表示时将被引发。这对整数来说不可能发生（宁可引发*MemoryError* 也不会放弃尝试）。但是出于历史原因，有时也会在整数超出要求范围的情况下引发 OverflowError。因为在 C 中缺少对浮点异常处理的标准化，大多数浮点运算都不会做检查。

exception PythonFinalizationError

该异常派生自*RuntimeError*。它会在解释器关闭或称*Python 终结化* 期间当有操作被阻止时被引发。

在 Python 终结化期间操作被阻止并引发*PythonFinalizationError* 的例子:

- 新建一个 Python 线程。
- *os.fork()*。

另请参阅*sys.is_finalizing()* 函数。

Added in version 3.13: 在此之前将只引发*RuntimeError*。

exception RecursionError

此异常派生自*RuntimeError*。它会在解释器检测发现超过最大递归深度 (参见*sys.getrecursionlimit()*) 时被引发。

Added in version 3.5: 在此之前将只引发*RuntimeError*。

exception ReferenceError

此异常将在使用*weakref.proxy()* 函数所创建的弱引用来访问该引用的某个已被作为垃圾回收的属性时被引发。有关弱引用的更多信息请参阅*weakref* 模块。

exception RuntimeError

当检测到一个不归属于任何其他类别的错误时将被引发。关联的值是一个指明究竟发生了什么问题的字符串。

exception StopIteration

由内置函数*next()* 和*iterator* 的*__next__()* 方法所引发，用来表示该迭代器不能产生下一项。

value

该异常对象只有一个属性 `value`，它在构造该异常时作为参数给出，默认值为*None*。

当一个*generator* 或*coroutine* 函数返回时，将引发一个新的*StopIteration* 实例，函数返回的值将被用作异常构造器的*value* 形参。

如果某个生成器代码直接或间接地引发了*StopIteration*，它会被转换为*RuntimeError* (并将*StopIteration* 保留为导致新异常的原因)。

在 3.3 版本发生变更: 添加了 `value` 属性及其被生成器函数用作返回值的功能。

在 3.5 版本发生变更: 引入了通过 `from __future__ import generator_stop` 来实现 RuntimeError 转换，参见 **PEP 479**。

在 3.7 版本发生变更: 默认对所有代码启用 **PEP 479**: 在生成器中引发的*StopIteration* 错误将被转换为*RuntimeError*。

exception StopAsyncIteration

必须由一个*asynchronous iterator* 对象的 `__anext__()` 方法来引发以停止迭代操作。

Added in version 3.5.

exception SyntaxError(*message*, *details*)

当解析器遇到语法错误时引发。这可以发生在 `import` 语句，对内置函数*compile()*, *exec()* 或*eval()* 的调用，或是读取原始脚本或标准输入（也包括交互模式）的时候。

异常实例的*str()* 只返回错误消息。错误详情为一个元组，其成员也可在单独的属性中分别获取。

filename

发生语法错误所在文件的名称。

lineno

发生错误所在文件中的行号。行号索引从 1 开始：文件中首行的 `lineno` 为 1。

offset

发生错误所在文件中的列号。列号索引从 1 开始：行中首个字符的 `offset` 为 1。

text

错误所涉及的源代码文本。

end_lineno

发生的错误在文件中的末尾行号。这个索引是从 1 开始的：文件中首行的 `lineno` 为 1。

end_offset

发生的错误在文件中的末尾列号。这个索引是从 1 开始：行中首个字符的 `offset` 为 1。

对于 f-字符串字段中的错误，消息会带有"f-string: " 前缀并且其位置是基于替换表达式构建的文本中的位置。例如，编译 f'Bad {a b} field' 将产生这样的 args 属性: ('f-string: ...', (", 1, 2, '(a b)n', 1, 5))。

在 3.10 版本发生变更: 增加了*end_lineno* 和*end_offset* 属性。

exception IndentationError

与不正确的缩进相关的语法错误的基类。这是*SyntaxError* 的一个子类。

exception TabError

当缩进包含对制表符和空格符不一致的使用时将被引发。这是*IndentationError* 的一个子类。

exception SystemError

当解释器发现内部错误，但情况看起来尚未严重到要放弃所有希望时将被引发。关联的值是一个指明发生了什么问题的字符串（使用低层级的表示形式）。在*CPython* 中，这可能会因不正确地使用 Python 的 C API 而引发，例如返回 `NULL` 值而不设置一个异常。

如果你确信此异常不是你的问题，或你所使用的软件包的问题，你应当将此问题报告给你所用 Python 解释器的作者或维护者。请确保报告 Python 解释器的版本 (`sys.version`；它也会在交互式 Python 会话开始时被打印出来)，具体的错误消息 (异常所关联的值) 以及可能触发该错误的程序的源代码。

exception SystemExit

此异常由*sys.exit()* 函数引发。它继承自*BaseException* 而不是*Exception* 以确保不会被处理*Exception* 的代码意外捕获。这允许此异常正确地向上传播并导致解释器退出。如果它未被处理，则 Python 解释器就将退出；不会打印任何栈回溯信息。构造器接受的可选参数与传递给*sys.exit()* 的相同。如果该值为一个整数，则它指明系统退出状态码（会传递给 C 的 `exit()` 函数）；如果该值为 `None`，则退出状态码为零；如果该值为其他类型（例如字符串），则会打印对象的值并将退出状态码设为一。

对*sys.exit()* 的调用会被转换为一个异常以便能执行清理处理程序 (`try` 语句的 `finally` 子句)，并且使得调试器可以执行一段脚本而不必冒失去控制的风险。如果绝对确实地需要立即退出（例如在调用*os.fork()* 之后的子进程中）则可使用*os._exit()*.

code

传给构造器的退出状态码或错误信息（默认为 None。）

exception TypeError

当一个操作或函数被应用于类型不适当的对象时将被引发。关联的值是一个字符串，给出有关类型不匹配的详情。

此异常可以由用户代码引发，以表明尝试对某个对象进行的操作不受支持也不应当受支持。如果某个对象应当支持给定的操作但尚未提供相应的实现，所要引发的适当异常应为*NotImplementedError*。

传入参数的类型错误 (例如在要求*int* 时却传入了*list*) 应当导致*TypeError*，但传入参数的值错误 (例如传入要求范围之外的数值) 则应当导致*ValueError*。

exception UnboundLocalError

当在函数或方法中对某个局部变量进行引用，但该变量并未绑定任何值时将被引发。此异常是*NameError* 的一个子类。

exception UnicodeError

当发生与 Unicode 相关的编码或解码错误时将被引发。此异常是*ValueError* 的一个子类。

UnicodeError 具有一些描述编码或解码错误的属性。例如 `err.object[err.start:err.end]` 会给出导致编解码器失败的特定无效输入。

encoding

引发错误的编码名称。

reason

描述特定编解码器错误的字符串。

object

编解码器试图要编码或解码的对象。

start

object 中无效数据的开始位置索引。

end

object 中无效数据的末尾位置索引（不含）。

exception UnicodeEncodeError

当在编码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception UnicodeDecodeError

当在解码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception UnicodeTranslateError

在转写过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception ValueError

当操作或函数接收到具有正确类型但值不适合的参数，并且情况不能用更精确的异常例如*IndexError* 来描述时将被引发。

exception ZeroDivisionError

当除法或取余运算的第二个参数为零时将被引发。关联的值是一个字符串，指明操作数和运算的类型。

下列异常被保留以与之前的版本相兼容；从 Python 3.3 开始，它们都是*OSError* 的别名。

exception EnvironmentError

exception IOError

exception WindowsError

限在 Windows 中可用。

5.4.1 OS 异常

下列异常均为*OSError* 的子类，它们将根据系统错误代码被引发。

exception BlockingIOError

当一个操作将在设置为非阻塞操作的对象（例如套接字）上发生阻塞时将被引发。对应于 errno *EAGAIN*, *EALREADY*, *EWOULDBLOCK* 和*EINPROGRESS*。

除了*OSError* 已有的属性，*BlockingIOError* 还有一个额外属性：

characters_written

一个整数，表示在被阻塞前已写入到流的字符数。当使用来自*io* 模块的带缓冲 I/O 类时此属性可用。

exception ChildProcessError

当一个子进程上的操作失败时将被引发。对应于 errno *ECHILD*。

exception ConnectionError

与连接相关问题的基类。

其子类有*BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* 和*ConnectionResetError*。

exception BrokenPipeError

ConnectionError 的子类，当试图写入一个管道而其另一端已关闭，或者试图写入一个套接字而其已关闭写入时将被引发。对应于 errno *EPIPE* 和*ESHUTDOWN*。

exception ConnectionAbortedError

ConnectionError 的子类，当一个连接尝试被对端中止时将被引发。对应于 errno *ECONNABORTED*。

exception ConnectionRefusedError

ConnectionError 的子类，当一个连接尝试被对端拒绝时将被引发。对应于 errno *ECONNREFUSED*。

exception ConnectionResetError

ConnectionError 的子类，当一个连接尝试被对端重置时将被引发。对应于 errno *ECONNRESET*。

exception FileExistsError

当试图创建一个已存在的文件或目录时将被引发。对应于 errno *EEXIST*。

exception FileNotFoundError

当所请求的文件或目录不存在时将被引发。对应于 errno *ENOENT*。

exception InterruptedError

当一个系统调用被传入的信号中断时将被引发。对应于 errno *EINTR*。

在 3.5 版本发生变更: 当系统调用被某个信号中断时，Python 现在会重试系统调用，除非该信号的处理程序引发了其它异常 (原理参见 **PEP 475**) 而不是引发*InterruptedError*。

exception IsADirectoryError

当请求对一个目录执行文件操作 (如*os.remove()*) 时将被引发。对应于 errno *EISDIR*。

exception NotADirectoryError

当请求对一个非目录执行目录操作 (如*os.listdir()*) 时将被引发。在大多数 POSIX 平台上，它还可能在某个操作试图将一个非目录作为目录打开或遍历时被引发。对应于 errno *ENOTDIR*。

exception PermissionError

当在没有足够访问权限的情况下试图运行某个操作时将被引发——例如文件系统权限。对应于 errno *EACCES*, *EPERM* 和*ENOTCAPABLE*。

在 3.11.1 版本发生变更: WASI 的*ENOTCAPABLE* 现在被映射至*PermissionError*。

exception ProcessLookupError

当给定的进程不存在时将被引发。对应于 errno *ESRCH*。

exception TimeoutError

当一个系统函数在系统层级发生超时的情况下将被引发。对应于 errno *ETIMEDOUT*。

Added in version 3.3: 添加了以上所有*OSError* 的子类。

参见

PEP 3151 - 重写 OS 和 IO 异常的层次结构

5.5 警告

下列异常被用作警告类别；请参阅警告类别 文档了解详情。

exception Warning

警告类别的基类。

exception UserWarning

用户代码所产生警告的基类。

exception DeprecationWarning

如果所发出的警告是针对其他 Python 开发者的，则以此作为与已弃用特性相关警告的基类。

会被默认警告过滤器忽略，在 __main__ 模块中的情况除外 (**PEP 565**)。启用*Python 开发模式* 时会显示此警告。

这个弃用政策是在 **PEP 387** 中描述的。

exception PendingDeprecationWarning

对于已过时并预计在未来弃用，但目前尚未弃用的特性相关警告的基类。

这个类很少被使用，因为针对未来可能的弃用发出警告的做法并不常见，而针对当前已有的弃用则推荐使用*DeprecationWarning*。

会被默认警告过滤器忽略。启用*Python 开发模式* 时会显示此警告。

这个弃用政策是在 **PEP 387** 中描述的。

exception SyntaxWarning

与模糊的语法相关的警告的基类。

exception RuntimeWarning

与模糊的运行时行为相关的警告的基类。

exception FutureWarning

如果所发出的警告是针对以 Python 所编写应用的最终用户的，则以此作为与已弃用特性相关警告的基类。

exception ImportWarning

与在模块导入中可能的错误相关的警告的基类。

会被默认警告过滤器忽略。启用*Python 开发模式* 时会显示此警告。

exception UnicodeWarning

与 Unicode 相关的警告的基类。

exception EncodingWarning

与编码格式相关的警告的基类。

请参阅选择性的 *EncodingWarning* 来了解详情。

Added in version 3.10.

exception BytesWarning

与*bytes* 和*bytearray* 相关的警告的基类。

exception ResourceWarning

资源使用相关警告的基类。

会被默认警告过滤器忽略。启用*Python 开发模式* 时会显示此警告。

Added in version 3.2.

5.6 异常组

下列异常是在有必要引发多个不相关联的异常时使用的。它们是异常层级结构的一部分因此它们可以像所有其他异常一样通过 except 来处理。此外，它们还可被 except* 所识别，此语法将基于所包含异常的类型来匹配其子分组。

exception ExceptionGroup(*msg*, *excs*)

exception BaseExceptionGroup(*msg*, *excs*)

这两个异常类型都将多个异常包装在序列 excs 中。msg 形参必须为一个字符串。这两个类之间的区别在于*BaseExceptionGroup* 扩展了*BaseException* 并且它可以包装任何异常，而*ExceptionGroup* 则扩展了*Exception* 并且它只能包装*Exception* 的子类。这样的设计是为了使得 except Exception 只捕获*ExceptionGroup* 而不捕获*BaseExceptionGroup*。

BaseExceptionGroup 构造器返回一个*ExceptionGroup* 而不是*BaseExceptionGroup*，如果所包含的全部异常都是*Exception* 的实例的话，因此它可以被用来制造自动化的选择。在另一方面，*ExceptionGroup* 构造器则会引发*TypeError*，如果所包含的任何异常不是*Exception* 的子类的话。

message

传给构造器的 msg 参数。这是一个只读属性。

exceptions

传给构造器的 excs 序列中的由异常组成的元组。这是一个只读属性。

subgroup(*condition*)

返回一个只包含来自当前组的匹配 *condition* 的异常的异常组，或者如果结果为空则返回 None。

该条件可以是一个异常类型或由异常类型组成的元组，在后一种情况中将对每个异常使用在 except 子句中所使用的相同检测方式来检测是否匹配。该条件也可以是一个可调用对象（而非类型对象），它接受一个异常作为其唯一参数并会针对应当属于特定子分组的异常返回真值。

当前异常的嵌套结构会在结果中保留，就如其*message*，*__traceback__*，*__cause__*，*__context__* 和*__notes__* 字段的值一样。空的嵌套组会在结果中被略去。

条件检测会针对嵌套异常组中的所有异常执行，包括最高层级的和任何嵌套的异常组。如果针对此类异常组的条件为真值，它将被完整包括在结果中。

Added in version 3.13: condition 可以是任意不为类型对象的可调用对象。

split(*condition*)

类似于*subgroup()*，但将返回 (match, rest) 对，其中 match 为 subgroup(condition) 而 rest 为剩余的非匹配部分。

derive(*excs*)

返回一个具有相同*message* 的异常组，但会将异常包装在 excs 中。

此方法是由*subgroup()* 和*split()* 使用的，它们被用于在各种上下文中拆分异常组。子类需要重写它以便让*subgroup()* 和*split()* 返回子类的实例而不是*ExceptionGroup*。

subgroup() 和*split()* 会从原始异常组拷贝*__traceback__*，*__cause__*，*__context__* 和*__notes__* 字段到*derive()* 所返回的异常组，这样这些字段就不需要被*derive()* 更新。

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, excs):
...         return MyGroup(self.message, excs)
...
>>> e = MyGroup("eg", [ValueError(1), TypeError(2)])
>>> e.add_note("a note")
>>> e.__context__ = Exception("context")
>>> e.__cause__ = Exception("cause")
>>> try:
...     raise e
... except Exception as e:
...     exc = e
...
>>> match, rest = exc.split(ValueError)
>>> exc, exc.__context__, exc.__cause__, exc.__notes__
(MyGroup('eg', [ValueError(1), TypeError(2)]), Exception('context'), Exception(
↪'cause'), ['a note'])
>>> match, match.__context__, match.__cause__, match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'), Exception('cause'), ['a note
↪'])
>>> rest, rest.__context__, rest.__cause__, rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'), Exception('cause'), ['a note
↪'])
>>> exc.__traceback__ is match.__traceback__ is rest.__traceback__
True
```

请注意*BaseExceptionGroup* 定义了 `__new__()`，因此需要不同构造器签名的子类必须重写该方法而不是 `__init__()`。例如，下面定义了一个接受 exit_code 并根据它来构造分组消息的异常组子类。

```
class Errors(ExceptionGroup):
   def __new__(cls, errors, exit_code):
      self = super().__new__(Errors, f"exit code: {exit_code}", errors)
      self.exit_code = exit_code
      return self

   def derive(self, excs):
      return Errors(excs, self.exit_code)
```

类似于*ExceptionGroup*，任何*BaseExceptionGroup* 的子类也是*Exception* 的子类，只能包装*Exception* 的实例。

Added in version 3.11.

5.7 异常层次结构

内置异常的类层级结构如下：

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
```

(续下页)

(接上页)

```
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    ├── PythonFinalizationError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

CHAPTER 6

文本处理服务

本章介绍的模块提供了广泛的字符串操作和其他文本处理服务。

在二进制数据服务 之下描述的*codecs* 模块也与文本处理高度相关。此外也请参阅 Python 内置字符串类型的文档文本序列类型 --- *str*。

6.1 `string` --- 常见的字符串操作

源代码： Lib/string.py

参见

文本序列类型 --- *str*

字符串的方法

6.1.1 字符串常量

此模块中定义的常量为：

`string.`**`ascii_letters`**

下文所述*ascii_lowercase* 和*ascii_uppercase* 常量的拼连。该值不依赖于语言区域。

`string.`**`ascii_lowercase`**

小写字母 `'abcdefghijklmnopqrstuvwxyz'`。该值不依赖于语言区域，不会发生改变。

`string.`**`ascii_uppercase`**

大写字母 `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`。该值不依赖于语言区域，不会发生改变。

`string.`**`digits`**

字符串 `'0123456789'`。

`string.`**`hexdigits`**

字符串 `'0123456789abcdefABCDEF'`。

string.**octdigits**

字符串 '01234567'。

string.**punctuation**

由在 C 区域设置中被视为标点符号的 ASCII 字符所组成的字符串: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.

string.**printable**

由被视为可打印符号的 ASCII 字符组成的字符串。这是*digits*, *ascii_letters*, *punctuation* 和*whitespace* 的总和。

> **备注**
>
> 根据设计，*string.printable.isprintable()* 将返回*False*。特别地，string.printable 在 POSIX 中是不可打印的 (参见 *LC_CTYPE*)。

string.**whitespace**

由被视为空白符号的 ASCII 字符组成的字符串。其中包括空格、制表、换行、回车、进纸和纵向制表符。

6.1.2 自定义字符串格式化

内置的字符串类提供了通过使用 **PEP 3101** 所描述的*format()* 方法进行复杂变量替换和值格式化的能力。*string* 模块中的*Formatter* 类允许你使用与内置*format()* 方法相同的实现来创建并定制你自己的字符串格式化行为。

class string.**Formatter**

Formatter 类包含下列公有方法:

format(*format_string*, /, **args*, ***kwargs*)

首要的 API 方法。它接受一个格式字符串和任意一组位置和关键字参数。它只是一个调用*vformat()* 的包装器。

在 3.7 版本发生变更: 格式字符串参数现在是仅限位置参数。

vformat(*format_string*, *args*, *kwargs*)

此函数执行实际的格式化操作。它被公开为一个单独的函数，用于需要传入一个预定义字母作为参数，而不是使用 *args 和 **kwargs 语法将字典解包为多个单独参数并重打包的情况。*vformat()* 完成将格式字符串分解为字符数据和替换字段的工作。它会调用下文所述的几种不同方法。

此外，*Formatter* 还定义了一些旨在被子类替换的方法:

parse(*format_string*)

循环遍历 format_string 并返回一个由可迭代对象组成的元组 (*literal_text*, *field_name*, *format_spec*, *conversion*)。它会被*vformat()* 用来将字符串分解为文本字面值或替换字段。

元组中的值在概念上表示一段字面文本加上一个替换字段。如果没有字面文本（如果连续出现两个替换字段就会发生这种情况），则 *literal_text* 将是一个长度为零的字符串。如果没有替换字段，则 *field_name*, *format_spec* 和 *conversion* 的值将为 None。

get_field(*field_name*, *args*, *kwargs*)

给定 *field_name* 作为*parse()* (见上文) 的返回值，将其转换为要格式化的对象。返回一个元组 (obj, used_key)。默认版本接受在 **PEP 3101** 所定义形式的字符串，例如"0[name]" 或"label.title"。*args* 和 *kwargs* 与传给*vformat()* 的一样。返回值 *used_key* 与*get_value()* 的 *key* 形参具有相同的含义。

get_value(*key*, *args*, *kwargs*)

提取给定的字段值。*key* 参数将为整数或字符串。如果是整数，它表示 *args* 中位置参数的索引；如果是字符串，它表示 *kwargs* 中的关键字参数名。

args 形参会被设为*vformat()* 的位置参数列表，而 *kwargs* 形参会被设为由关键字参数组成的字典。

对于复合字段名称，仅会为字段名称的第一个组件调用这些函数；后续组件会通过普通属性和索引操作来进行处理。

因此举例来说，字段表达式'0.name' 将导致调用*get_value()* 时附带 *key* 参数值 0。在*get_value()* 通过调用内置的*getattr()* 函数返回后将会查找 `name` 属性。

如果索引或关键字引用了一个不存在的项，则将引发*IndexError* 或*KeyError*。

check_unused_args(*used_args*, *args*, *kwargs*)

在必要时实现对未使用参数进行检测。此函数的参数是是格式字符串中实际引用的所有参数键的集合（整数表示位置参数，字符串表示名称参数），以及被传给 vformat 的 *args* 和 *kwargs* 的引用。未使用参数的集合可以根据这些形参计算出来。如果检测失败则*check_unused_args()* 应会引发一个异常。

format_field(*value*, *format_spec*)

format_field() 会简单地调用内置全局函数*format()*。提供该方法是为了让子类能够重载它。

convert_field(*value*, *conversion*)

使用给定的转换类型（来自*parse()* 方法所返回的元组）来转换（由*get_field()* 所返回的）值。默认版本支持's' (str), 'r' (repr) 和'a' (ascii) 等转换类型。

6.1.3 格式字符串语法

str.format() 方法和*Formatter* 类共享相同的格式字符串语法（虽然对于*Formatter* 来说，其子类可以定义它们自己的格式字符串语法）。具体语法与 格式化字符串字面值相似，但较为简单一些，并且关键的一点是不支持任意表达式。

格式字符串包含有以花括号 `{}` 括起来的“替换字段”。不在花括号之内的内容被视为字面文本，会不加修改地复制到输出中。如果你需要在字面文本中包含花括号字符，可以通过重复来转义: `{{` and `}}`。

替换字段的语法如下:

```
replacement_field ::= "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::= arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::= [identifier | digit+]
attribute_name    ::= identifier
element_index     ::= digit+ | index_string
index_string      ::= <any source character except "]"> +
conversion        ::= "r" | "s" | "a"
format_spec       ::= format-spec:format_spec
```

用不太正式的术语来描述，替换字段开头可以用一个 *field_name* 指定要对值进行格式化并取代替换字符被插入到输出结果的对象。*field_name* 之后有可选的 *conversion* 字段，它是一个感叹号 `'!'` 加一个 *format_spec*，并以一个冒号 `':'` 打头。这些指明了替换值的非默认格式。

另请参阅格式规格迷你语言 一节。

field_name 本身以一个数字或关键字形式的 *arg_name* 打头。如果为数字，则它指向一个位置参数，而如果为关键字，则它指向一个命名关键字参数。如果在字符串上调用*str.isdecimal()* 会返回真值则 *arg_name* 会被当作数字来处理。如果格式字段串中的数字 arg_names 为 0, 1, 2, ... 的序列，它们可以全部（而非部分）被省略并且数字 0, 1, 2, ... 将按顺序被自动插入。由于 *arg_name* 不使用引号分隔，因此无法在格式字符串中指定任意的字典键（例如字符串 `'10'` 或 `':-]'` 等）。*arg_name* 之后可以跟任意数量的索引或属性表达式。`'.name'` 形式的表达式会使用*getattr()* 来选择命名属性，而 `'[index]'` 形式的表达式会使用 `__getitem__()` 来执行索引查找。

在 3.1 版本发生变更: 位置参数说明符对于*str.format()* 可以省略，因此 `'{} {}'.format(a, b)` 等价于 `'{0} {1}'.format(a, b)`。

在 3.4 版本发生变更: 位置参数说明符对于*Formatter* 可以省略。

一些简单的格式字符串示例

```
"First, thou shalt count to {0}"  # 引用第一个位置参数
"Bring me a {}"                   # 隐式引用第一个位置参数
"From {} to {}"                   # 等同于 "From {0} to {1}"
"My quest is {name}"              # 引用关键字参数 'name'
"Weight in tons {0.weight}"       # 第一个位置参数的 'weight' 属性
"Units destroyed: {players[0]}"   # 关键字参数 'players' 的第一个元素。
```

conversion 字段会在格式化之前进行类型强制转换。通常，格式化一个值的工作是由该值本身的 `__format__()` 方法完成的。但是，在某些情况下最好是强制将类型格式化为一个字符串，覆盖其本身的格式化定义。通过在调用 `__format__()` 之间将值转换为字符串，可以绕过正常的格式化逻辑。

目前支持的转换旗标有三种: `'!s'` 会对值调用*str()*，`'!r'` 调用*repr()* 而 `'!a'` 则调用*ascii()*。

示例如下:

```
"Harold's a clever {0!s}"        # 先在参数上调用 str()
"Bring out the holy {name!r}"    # 先在参数上调用 repr()
"More {!a}"                      # 先在参数上调用 ascii()
```

format_spec 字段包含值应如何呈现的规格描述，例如字段宽度、对齐、填充、小数精度等细节信息。每种值类型可以定义自己的“格式化迷你语言”或对 *format_spec* 的解读方式。

大多数内置类型都支持同样的格式化迷你语言，具体描述见下一节。

format_spec 字段还可以在其内部包含嵌套的替换字段。这些嵌套的替换字段可能包括字段名称、转换旗标和格式规格描述，但是不再允许更深层的嵌套。format_spec 内部的替换字段会在解读 *format_spec* 字符串之前先被解读。这将允许动态地指定特定值的格式。

请参阅格式示例 一节查看相关示例。

格式规格迷你语言

“格式规格”在格式字符串所包含的替换字段内部使用，用于定义单个值应如何呈现 (参见格式字符串语法 和 f-strings)。它们也可以被直接传给内置的*format()* 函数。每种可格式化的类型都可以自行定义如何对格式规格进行解读。

大多数内置类型都为格式规格实现了下列选项，不过某些格式化选项只被数值类型所支持。

一般约定空的格式描述将产生与在值上调用*str()* 相同的结果。非空格式描述通常会修改此结果。

标准格式说明符的一般形式如下:

```
format_spec ::= [options][width][grouping]["." precision][type]
options     ::= [[fill]align][sign]["z"]["#"]["0"]
fill        ::= <any character>
align       ::= "<" | ">" | "=" | "^"
sign        ::= "+" | "-" | " "
width       ::= digit+
grouping    ::= "," | "_"
precision   ::= digit+
type        ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g"
                | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

如果指定了一个有效的 *align* 值，则可以在该值前面加一个 *fill* 字符，它可以为任意字符，如果省略则默认为空格符。在 格式化字符串字面值或在使用*str.format()* 方法时是无法使用花括号字面值 (“{” or “}”) 作为 *fill* 字符的。但是，通过嵌套替换字段插入花括号则是可以的。这个限制不会影响*format()* 函数。

各种对齐选项的含义如下:

选项	含意
'<'	强制字段在可用空间内左对齐（这是大多数对象的默认值）。
'>'	强制字段在可用空间内右对齐（这是数字的默认值）。
'='	强制在符号（如果有）之后数字之前放置填充。这被用于以'+000000120' 形式打印字段。这个对齐选项仅适用于数字类型，*complex* 除外。当'0' 紧接在字段宽度之前时这是默认行为。
'^'	强制字段在可用空间内居中。

请注意，除非定义了最小字段宽度，否则字段宽度将始终与填充它的数据大小相同，因此在这种情况下，对齐选项没有意义。

sign 选项仅对数字类型有效，可以是以下之一：

选项	含意
'+'	表示正负号应当同时用于正数与负数。
'-'	表示正负号应当仅用于负数（这是默认的行为）。
space	表示应当对正数使用前导空格，而对负数使用负号。

The 'z' option coerces negative zero floating-point values to positive zero after rounding to the format precision. This option is only valid for floating-point presentation types.

在 3.11 版本发生变更: 增加了 'z' 选项 (另请参阅 **PEP 682**)。

'#' 选项可让“替代形式”被用于执行转换。替代形式会针对不同的类型分别定义。此选项仅适用于整数、浮点数和复数类型。对于整数类型，当使用二进制、八进制或十六进制输出时，此选项会为输出值分别添加相应的 '0b','0o','0x' 或 '0X' 前缀。对于浮点数和复数类型，替代形式会使得转换结果总是包含小数点符号，即使其不带小数部分。通常只有在带有小数部分的情况下，此类转换的结果中才会出现小数点符号。此外，对于 'g' 和 'G' 转换，末尾的零不会从结果中被移除。

width 是一个定义最小总字段宽度的十进制整数，包括任何任何前缀、分隔符和其他格式化字符。如果未指定，则字段宽度将由内容确定。

当未显式给出对齐方式时，在 *width* 字符前加一个零 ('0') 字符将为数字类型启用感知正负号的零填充，*complex* 除外。这相当于将 *fill* 字符设为 '0' 并将 *alignment* 类型设为 '='。

在 3.10 版本发生变更: 在 *width* 字段之前添加 '0' 不会再影响字符串的默认对齐。

在 *width* 字段之后的 *grouping* 选项指定用于数字的整数部分的数位组分隔符。它可以为下列值之一：

选项	含意
','	对于整数表示类型 'd' 和 'n' 以外的浮点数表示类型每 3 个数位插入一个逗号。对于其他表示类型，此选项不受支持。
'_'	对于整数表示类型 'd' 和 'n' 以外的浮点数表示类型，每 3 个数位插入一个下划线。对于整数表示类型 'b','o','x' 和 'X'，则每 4 个数位插入一个下划线。对于其他表示类型，此选项不受支持。

想要能感知语言区域的分隔符，请改用 'n' 表示类型。

在 3.1 版本发生变更: 添加了 ',' 选项 (另请参阅 **PEP 378**)。

在 3.6 版本发生变更: 添加了 '_' 选项 (另请参阅 **PEP 515**)。

precision 是一个十进制整数，它表示对于以表示类型 'f' 和 'F' 格式化的数值应当在小数点后显示多少个数位，或者对于以表示类型 'g' 或 'G' 格式化的数值应当在小数点前后显示多少个数位。对于字符串表示类型，该字段表示最大的字段大小——换句话说，就是要使用多少个来自字段内容的字符。不允许对整数表示类型指定 *precision* 字段。

最后，*type* 确定了数据应如何呈现。

可用的字符串表示类型是:

类型	含意
's'	字符串格式。这是字符串的默认类型，可以省略。
None	和 's' 一样。

可用的整数表示类型是:

类型	含意
'b'	二进制格式。输出以 2 为基数的数字。
'c'	字符。在打印之前将整数转换为相应的 unicode 字符。
'd'	十进制整数。输出以 10 为基数的数字。
'o'	八进制格式。输出以 8 为基数的数字。
'x'	十六进制格式。输出以 16 为基数的数字，使用小写字母表示 9 以上的数码。
'X'	十六进制格式。输出以 16 为基数的数字，使用大写字母表示 9 以上的数码。在指定 '#' 的情况下，前缀 '0x' 也将被转为大写形式 '0X'。
'n'	数字。这与 'd' 相似，区别在于它会使用当前语言区域设置来插入适当的数字组分隔符。
None	和 'd' 相同。

在上述的表示类型之外，整数还可以通过下列的浮点表示类型来格式化 (除了 'n' 和 None)。当这样做时，会在格式化之前使用 *float()* 将整数转换为浮点数。

float 和 *Decimal* 值的可用表示类型有:

类型	含意
'e'	科学计数法。对于给定的精度 p，将数字格式化为以字母'e' 分隔系数和指数的科学计数法表示形式。系数在小数点之前有一位而在之后有 p 位，总计 p + 1 个有效数位。如未指定精度，则会对*float* 采用小数点之后 6 位精度，而对*Decimal* 则显示所有系数位。如果 p=0，则小数点会被略去，除非使用了 # 选项。
'E'	科学计数法。与 'e' 相似，不同之处在于它使用大写字母'E' 作为分隔字符。
'f'	定点表示法。对于给定的精度 p，将数字格式化为小数点之后恰好有 p 位的小数形式。如未指定精度，则会对*float* 采用小数点之后 6 位精度，而对*Decimal* 则使用大到足够显示所有系数位的精度。如果 p=0，则小数点会被略去，除非使用了 # 选项。
'F'	定点表示。与 'f' 相似，但会将 nan 转为 NAN 并将 inf 转为 INF。
'g'	常规格式。对于给定精度 p >= 1，这会将数值舍入到 p 个有效数位，再将结果以定点表示法或科学计数法进行格式化，具体取决于其值的大小。精度 0 会被视为等价于精度 1。 准确的规则如下：假设使用表示类型 'e' 和精度 p-1 进行格式化的结果具有指数值 exp。那么如果 m <= exp < p，其中 m 以 -4 表示浮点值而以 -6 表示*Decimal* 值，该数字将使用类型 'f' 和精度 p-1-exp 进行格式化。否则的话，该数字将使用表示类型 'e' 和精度 p-1 进行格式化。在两种情况下，都会从有效数字中移除无意义的末尾零，如果小数点之后没有余下数字则小数点也会被移除，除非使用了 '#' 选项。 如未指定精度，会对*float* 采用 6 个有效数位的精度。对于*Decimal*，结果的系数会沿用原值的系数数位；对于绝对值小于 1e-6 的值以及最小有效数位的位值大于 1 的数值将会使用科学计数法，在其他情况下则会使用定点表示法。 正负无穷，正负零和 nan 会分别被格式化为 inf, -inf, 0, -0 和 nan，无论精度如何设定。
'G'	常规格式。类似于 'g'，不同之处在于当数值非常大时会切换为 'E'。无穷与 NaN 也会表示为大写形式。
'n'	数字。这与 'g' 相似，区别在于它会使用当前语言区域来为数字的整数部分插入适当的数字组分隔符。
'%'	百分比。将数字乘以 100 并显示为定点 ('f') 格式，后面带一个百分号。
None	对于*float* 来说这类似于 'g' 类型，不同之处在于当使用定点表示形式来格式化结果时，它将总是包括小数点后至少一位，并在 exp >= p - 1 时切换为科学计数法表示形式。当未指定精度时，后者的精度将大到足以精确表示给定的值。 对于*Decimal* 来说这相当于 'g' 或 'G'，具体取决于当前 decimal 上下文的 context.capitals 值。 总体效果是将*str()* 的输出匹配为其他格式化因子所调整出的样子。

结果应当被正确地舍入到给定的小数点后 p 位精度。针对*float* 的舍入模式与*round()* 内置函数的相匹配。对于*Decimal*，将使用当前上下文 的舍入模式。

针对*complex* 的可用表示类型与针对*float* 的相同（但 '%' 不被允许）。复数的实部和虚部都将被作为浮点数来格式化，使用指定的表示类型。它们将以必须带有的虚部正负符号来分隔，虚部还将带有 j 后缀。如果未提供表示类型，结果将匹配*str()* 的输出（具有非零实部的复数还将带有圆括号），并可能被其他格式说明符所修改。

格式示例

本节包含*str.format()* 语法的示例以及与旧式 % 格式化的比较。

该语法在大多数情况下与旧式的 % 格式化类似，只是增加了 {} 和 : 来取代 %。例如，, '%03.2f' 可以被改写为 '{:03.2f}'。

新的格式语法还支持新增的不同选项，将在以下示例中说明。

按位置访问参数:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
```

(续下页)

(接上页)

```
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # 解包参数序列
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # 参数的索引可重复使用
'abracadabra'
```

按名称访问参数:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

访问参数的属性:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

访问参数的项:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

替代 `%s` 和 `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

对齐文本以及指定宽度:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # 使用 '*' 作为填充字符
'***********centered***********'
```

替代 `%+f`, `%-f` 和 `% f` 以及指定正负号:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # 总是显示
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # 对正数显示一个空格
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # 只显示负号 -- 等同于 '{:f}; {:f}'
'3.140000; -3.140000'
```

替代 %x 和 %o 以及转换基于不同进位制的值:

```
>>> # 格式也支持二进制数
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

使用逗号或下划线作为数字组分隔符:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
>>> '{:_}'.format(1234567890)
'1_234_567_890'
>>> '{:_b}'.format(1234567890)
'100_1001_1001_0110_0000_0010_1101_0010'
>>> '{:_x}'.format(1234567890)
'4996_02d2'
```

表示为百分数:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

使用特定类型的专属格式化:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

嵌套参数以及更复杂的示例:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 模板字符串

模板字符串提供了由 **PEP 292** 所描述的更简便的字符串替换方式。模板字符串的一个主要用例是文本国际化 (i18n)，因为在此情境下，更简单的语法和功能使得文本翻译过程比使用 Python 的其他内置字符串格式化工具更方便。作为基于模板字符串构建以实现 i18n 的库的一个例子，请参看 flufl.i18n 包。

模板字符串支持基于 `$` 的替换，使用以下规则：

- `$$` 为转义符号；它会被替换为单个的 `$`。
- `$identifier` 为替换占位符，它会匹配一个名为 `"identifier"` 的映射键。在默认情况下，`"identifier"` 限制为任意 ASCII 字母数字（包括下划线）组成的字符串，不区分大小写，以下划线或 ASCII 字母开头。在 `$` 字符之后的第一个非标识符字符将表明占位符的终结。
- `${identifier}` 等价于 `$identifier`。当占位符之后紧跟着有效的但又不是占位符一部分的标识符字符时需要使用，例如 `"${noun}ification"`。

在字符串的其他位置出现 `$` 将导致引发*ValueError*。

string 模块提供了实现这些规则的*Template* 类。*Template* 有下列方法：

class string.**Template**(*template*)

该构造器接受一个参数作为模板字符串。

substitute(*mapping={}*, */*, ***kwds*)

执行模板替换，返回一个新字符串。*mapping* 为任意字典类对象，其中的键将匹配模板中的占位符。或者你也可以提供一组关键字参数，其中的关键字即对应占位符。当同时给出 *mapping* 和 *kwds* 并且存在重复时，则以 *kwds* 中的占位符为优先。

safe_substitute(*mapping={}*, */*, ***kwds*)

类似于*substitute()*，不同之处是如果有占位符未在 *mapping* 和 *kwds* 中找到，不是引发*KeyError* 异常，而是将原始占位符不加修改地显示在结果字符串中。另一个与*substitute()* 的差异是任何在其他情况下出现的 `$` 将简单地返回 `$` 而不是引发*ValueError*。

此方法被认为“安全”，因为虽然仍有可能发生其他异常，但它总是尝试返回可用的字符串而不是引发一个异常。从另一方面来说，*safe_substitute()* 也可能根本算不上安全，因为它将静默地忽略错误格式的模板，例如包含多余的分隔符、不成对的花括号或不是合法 Python 标识符的占位符等等。

is_valid()

如果模板有会导致*substitute()* 引发*ValueError* 的无效占位符则返回假值。

Added in version 3.11.

get_identifiers()

返回模板中有效占位符的列表，按它们首次出现的顺序排列，忽略任何无效标识符。

Added in version 3.11.

Template 的实例还提供一个公有数据属性：

template

这是作为构造器的 *template* 参数被传入的对象。一般来说，你不应该修改它，但并不强制要求只读访问。

以下是一个如何使用模版的示例：

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
```

(续下页)

(接上页)

```
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

进阶用法：你可以派生*Template*的子类来自定义占位符语法、分隔符，或用于解析模板字符串的整个正则表达式。为此目的，你可以重载这些类属性：

- *delimiter* -- 这是用来表示占位符的起始的分隔符的字符串字面值。默认值为 `$`。请注意此参数 不能为正则表达式，因为其实现将在必要时对此字符串调用*re.escape()*。还要注意你不能在创建类之后改变此分隔符（例如在子类的类命名空间中必须设置不同的分隔符）。
- *idpattern* -- 这是用来描述不带花括号的占位符的模式的正则表达式。默认值为正则表达式 `(?a:[_a-z][_a-z0-9]*)`。如果给出了此属性并且 *braceidpattern* 为 `None` 则此模式也将作用于带花括号的占位符。

  备注

  由于默认的 *flags* 为 `re.IGNORECASE`，模式 `[a-z]` 可以匹配某些非 ASCII 字符。因此我们在这里使用了局部旗标 `a`。

  在 3.7 版本发生变更: *braceidpattern* 可被用来定义对花括号内部和外部进行区分的模式。

- *braceidpattern* -- 此属性类似于 *idpattern* 但是用来描述带花括号的占位符的模式。默认值 `None` 意味着回退到 *idpattern* (即在花括号内部和外部使用相同的模式)。如果给出此属性，这将允许你为带花括号和不带花括号的占位符定义不同的模式。

  Added in version 3.7.

- *flags* -- 将在编译用于识别替换内容的正则表达式被应用的正则表达式旗标。默认值为 `re.IGNORECASE`。请注意 `re.VERBOSE` 总是会被加为旗标，因此自定义的 *idpattern* 必须遵循详细正则表达式的约定。

  Added in version 3.2.

作为另一种选项，你可以通过重载类属性 *pattern* 来提供整个正则表达式模式。如果你这样做，该值必须为一个具有四个命名捕获组的正则表达式对象。这些捕获组对应于上面已经给出的规则，以及无效占位符的规则：

- *escaped* -- 这个组匹配转义序列，在默认模式中即 `$$`。
- *named* -- 这个组匹配不带花括号的占位符名称；它不应当包含捕获组中的分隔符。
- *braced* -- 这个组匹配带有花括号的占位符名称；它不应当包含捕获组中的分隔符或者花括号。
- *invalid* -- 这个组匹配任何其他分隔符模式（通常为单个分隔符），并且它应当出现在正则表达式的末尾。

如果模式匹配模板但这些命名分组均不匹配则该类上的方法将引发*ValueError*。

6.1.5 辅助函数

`string.`**`capwords`**(*s*, *sep=None*)

使用*str.split()*将参数拆分为单词，使用*str.capitalize()*将单词转为大写形式，使用*str.join()*将大写的单词进行拼接。如果可选的第二个参数 *sep* 被省略或为 `None`，则连续的空白字符会被替换为单个空格符并且开头和末尾的空白字符会被移除，否则 *sep* 会被用来拆分和拼接单词。

6.2 re --- 正则表达式操作

源代码: Lib/re/

本模块提供了与 Perl 语言类似的正则表达式匹配操作。

模式和被搜索的字符串即可以是 Unicode 字符串 (*str*)，也可以是 8 位字节串 (*bytes*)。但是，Unicode 字符串与 8 位字节串不能混用：也就是说，不能将 Unicode 字符串与字节串模式进行匹配，反之亦然；同样地，在执行替换时，替换字串的类型也必须与所用的模式和搜索字串的类型一致。

正则表达式使用反斜杠字符 (`'\'`) 表示特殊形式或是允许在使用特殊字符时不引发它们的特殊含义。这会与 Python 在字符串字面值中对于相同字符出于相同目的规定的用法发生冲突；例如，要匹配一个反斜杠字面值，用户将必须写成 `'\\\\'` 因为正则表达式必须为 `\\`，而每个反斜杠在普通 Python 字符串字面值中又必须表示为 `\\`。而且，还要注意在 Python 的字符串字面值中使用的反斜杠现在如果有任何无效的转义序列将会产生*SyntaxWarning* 并将在未来改为*SyntaxError*。此行为即使对于正则表达式来说有效的转义字符同样会发生。

解决办法是对于正则表达式模式（patterns）使用 Python 的原始字符串表示法；在带有 `'r'` 前缀的字符串字面值中，反斜杠不必做任何特殊处理。因此 `r"\n"` 表示包含 `'\'` 和 `'n'` 两个字符的字符串，而 `"\n"` 则表示只包含一个换行符的字符串。模式在 Python 代码中通常都使用原始字符串表示法。

绝大多数正则表达式操作都提供为模块函数和方法，在编译正则表达式. 这些函数是一个捷径，不需要先编译正则对象，但是损失了一些优化参数。

> **参见**
>
> 第三方模块 regex 提供了与标准库*re* 模块兼容的 API，还提供了附加功能和更全面的 Unicode 支持。

6.2.1 正则表达式语法

正则表达式（或 RE）指定了一组与之匹配的字符串；模块内的函数可以检查某个字符串是否与给定的正则表达式匹配（或者正则表达式是否匹配到字符串，这两种说法含义相同）。

正则表达式可以拼接；如果 *A* 和 *B* 都是正则表达式，则 *AB* 也是正则表达式。通常，如果字符串 *p* 匹配 *A*，并且另一个字符串 *q* 匹配 *B*，那么 *pq* 可以匹配 AB。除非 *A* 或者 *B* 包含低优先级操作，*A* 和 *B* 存在边界条件；或者命名组引用。所以，复杂表达式可以很容易的从这里描述的简单源语表达式构建。更多正则表达式理论和实现，详见 the Friedl book [Frie09]，或者其他构建编译器的书籍。

以下是正则表达式格式的简要说明。更详细的信息和演示，参考 regex-howto。

正则表达式可以包含普通或者特殊字符。绝大部分普通字符，比如 `'A'`, `'a'`, 或者 `'0'`，都是最简单的正则表达式。它们就匹配自身。你可以拼接普通字符，所以 `last` 匹配字符串 `'last'`.（在这一节的其他部分，我们将用 `this special style` 这种方式表示正则表达式，通常不带引号，要匹配的字符串用 `'in single quotes'`，单引号形式。）

有些字符，比如 `'|'` 或者 `'('`，属于特殊字符。特殊字符既可以表示它的普通含义，也可以影响它旁边的正则表达式的解释。

重复运算符或数量限定符 (`*`, `+`, `?`, `{m,n}` 等) 不能被直接嵌套。这避免了非贪婪修饰符后缀 `?` 的歧义，也避免了其他实现中其他修饰符的歧义。要将第二层重复应用到内层的重复中，可以使用圆括号。例如，表达式 `(?:a{6})*` 将匹配六个 `'a'` 字符的任意多次重复。

特殊字符有：

`.`
: （点号.）在默认模式下，匹配除换行符以外的任意字符。如果指定了旗标*DOTALL*，它将匹配包括换行符在内的任意字符。`(?s:.)` 将匹配任意字符而无视相关旗标。

`^`
: (插入符) 匹配字符串的开头，并且在*MULTILINE* 模式下也匹配换行后的首个符号。

$

匹配字符串尾或者在字符串尾的换行符的前一个字符，在*MULTILINE* 模式下也会匹配换行符之前的文本。`foo` 匹配'foo' 和'foobar'，但正则表达式 `foo$` 只匹配'foo'。更有趣的是，在 `'foo1\nfoo2\n'` 中搜索 `foo.$`，通常匹配'foo2'，但在*MULTILINE* 模式下可以匹配到'foo1'；在 `'foo\n'` 中搜索 `$` 会找到两个（空的）匹配：一个在换行符之前，一个在字符串的末尾。

*****

对它前面的正则式匹配 0 到任意次重复，尽量多的匹配字符串。`ab*` 会匹配 `'a'`，`'ab'`，或者 `'a'` 后面跟随任意个 `'b'`。

+

对它前面的正则式匹配 1 到任意次重复。`ab+` 会匹配 `'a'` 后面跟随 1 个以上到任意个 `'b'`，它不会匹配 `'a'`。

?

对它前面的正则式匹配 0 到 1 次重复。`ab?` 会匹配 `'a'` 或者 `'ab'`。

***?, +?, ??**

`'*'`, `'+'` 和 `'?'` 数量限定符都是 *贪婪的*；它们会匹配尽可能多的文本。有时这种行为并不被需要；如果 RE `<.*>` 针对 `'<a> b <c>'` 进行匹配，它将匹配整个字符串，而不只是 `'<a>'`。在数量限定符之后添加 `?` 将使其以 *非贪婪* 或 *最小风格* 来执行匹配；也就是将匹配数量尽可能 *少的* 字符。使用 RE `<.*?>` 将只匹配 `'<a>'`。

***+, ++, ?+**

类似于 `'*'`, `'+'` 和 `'?'` 数量限定符，添加了 `'+'` 的形式也将匹配尽可能多的次数。但是，不同于真正的贪婪型数量限定符，这些形式在之后的表达式匹配失败时不允许反向追溯。这些形式被称为 *占有型* 数量限定符。例如，`a*a` 将匹配 `'aaaa'` 因为 `a*` 将匹配所有的 4 个 `'a'`，但是，当遇到最后一个 `'a'` 时，表达式将执行反向追溯以便最终 `a*` 最后变为匹配总计 3 个 `'a'`，而第四个 `'a'` 将由最后一个 `'a'` 来匹配。然而，当使用 `a*+a` 时如果要匹配 `'aaaa'`，`a*+` 将匹配所有的 4 个 `'a'`，但是在最后一个 `'a'` 无法找到更多字符来匹配时，表达式将无法被反向追溯并将因此匹配失败。`x*+`，`x++` 和 `x?+` 分别等价于 `(?>x*)`，`(?>x+)` 和 `(?>x?)`。

Added in version 3.11.

{m}

对其之前的正则式指定匹配 *m* 个重复；少于 *m* 的话就会导致匹配失败。比如，`a{6}` 将匹配 6 个 `'a'`，但是不能是 5 个。

{m,n}

对正则式进行 *m* 到 *n* 次匹配，在 *m* 和 *n* 之间取尽量多。比如，`a{3,5}` 将匹配 3 到 5 个 `'a'`。忽略 *m* 意为指定下界为 0，忽略 *n* 指定上界为无限次。比如 `a{4,}b` 将匹配 `'aaaab'` 或者 1000 个 `'a'` 尾随一个 `'b'`，但不能匹配 `'aaab'`。逗号不能省略，否则无法辨别修饰符应该忽略哪个边界。

{m,n}?

将导致结果 RE 匹配之前 RE 的 *m* 至 *n* 次重复，尝试匹配尽可能 *少的* 重复次数。这是之前数量限定符的非贪婪版本。例如，在 6 个字符的字符串 `'aaaaaa'` 上，`a{3,5}` 将匹配 5 个 `'a'` 字符，而 `a{3,5}?` 将只匹配 3 个字符。

{m,n}+

将导致结果 RE 匹配之前 RE 的 *m* 至 *n* 次重复，尝试匹配尽可能多的重复而 *不会* 建立任何反向追溯点。这是上述数量限定符的占有型版本。例如，在 6 个字符的字符串 `'aaaaaa'` 上，`a{3,5}+aa` 将尝试匹配 5 个 `'a'` 字符，然后，要求再有 2 个 `'a'`，这将需要比可用的更多的字符因而会失败，而 `a{3,5}aa` 的匹配将使 `a{3,5}` 先捕获 5 个，然后通过反向追溯再匹配 4 个 `'a'`，然后用模式中最后的 `aa` 来匹配最后的 2 个 `'a'`。`x{m,n}+` 就等同于 `(?>x{m,n})`。

Added in version 3.11.

****

转义特殊字符（允许你匹配 `'*'`, `'?'`, 或者此类其他），或者表示一个特殊序列；特殊序列之后进行讨论。

如果你没有使用原始字符串（`r'raw'`）来表达样式，要牢记 Python 也使用反斜杠作为转义序列；如果转义序列不被 Python 的分析器识别，反斜杠和字符才能出现在字符串中。如果 Python 可以识别这个序列，那么反斜杠就应该重复两次。这将导致理解障碍，所以高度推荐，就算是最简单的表达式，也要使用原始字符串。

[]

用于表示一个字符集合。在一个集合中：

- 字符可以单独列出，比如 `[amk]` 匹配 `'a'`，`'m'`，或者 `'k'`。
- 可以表示字符范围，通过用 `'-'` 将两个字符连起来。比如 `[a-z]` 将匹配任何小写 ASCII 字符，`[0-5][0-9]` 将匹配从 `00` 到 `59` 的两位数字，`[0-9A-Fa-f]` 将匹配任何十六进制数位。如果 `-` 进行了转义（比如 `[a\-z]`）或者它的位置在首位或者末尾（如 `[-a]` 或 `[a-]`），它就只表示普通字符 `'-'`。
- 除反斜杠外的特殊字符在集中中会失去其特殊含义。例如，`[(+*)]` 将匹配字符字面值 `'('`，`'+'`，`'*'`，或 `')'` 中的任何一个。
- 反斜杠或者用于转义集合中具有特殊含义的字符如 `'-'`，`']'`，`'^'` 及 `'\\'` 本身或者用于提示代表单个字符的特殊序列如 `\xa0` 或 `\n` 或者用于字符类如 `\w` 或 `\S` (定义见下文)。请注意 `\b` 是表示单个"backspace" 字符，而不是如在集合以外那样表示单词边界，还有数字转义符如 `\1` 将总是为八进制形式的转义，而不是分组引用。不匹配单个字符的特殊转义符如 `\A` 和 `\Z` 是不被允许的。are not allowed.
- 不在集合范围内的字符可以通过 取反来进行匹配。如果集合首字符是 `'^'`，所有 不在集合内的字符将会被匹配，比如 `[^5]` 将匹配所有字符，除了 `'5'`，`[^^]` 将匹配所有字符，除了 `'^'`. `^` 如果不在集合首位，就没有特殊含义。
- 要在集合内匹配一个 `']'` 字面值，可以在它前面加上反斜杠，或是将它放到集合的开头。例如，`[()[\]{}]` 和 `[]()[{}]` 都可以匹配右方括号，以及左方括号，花括号和圆括号。
- Unicode Technical Standard #18 里的嵌套集合和集合操作支持可能在未来添加。这将会改变语法，所以为了帮助这个改变，一个*FutureWarning* 将会在有多义的情况里被 `raise`，包含以下几种情况，集合由 `'['` 开始，或者包含下列字符序列 `'--'`，`'&&'`，`'~~'`，和 `'||'`。为了避免警告，需要将它们用反斜杠转义。

在 3.7 版本发生变更: 如果一个字符串构建的语义在未来会改变的话，一个*FutureWarning* 会 `raise` 。

|

`A|B`，*A* 和 *B* 可以是任意正则表达式，创建一个正则表达式，匹配 *A* 或者 *B*. 任意个正则表达式可以用 `'|'` 连接。它也可以在组合（见下列）内使用。扫描目标字符串时，`'|'` 分隔开的正则样式从左到右进行匹配。当一个样式完全匹配时，这个分支就被接受。意思就是，一旦 *A* 匹配成功，*B* 就不再进行匹配，即便它能产生一个更好的匹配。或者说，`'|'` 操作符绝不贪婪。如果要匹配 `'|'` 字符，使用 `\|`，或者把它包含在字符集里，比如 `[|]`.

(...)

(组合)，匹配括号内的任意正则表达式，并标识出组合的开始和结尾。匹配完成后，组合的内容可以被获取，并可以在之后用 `\number` 转义序列进行再次匹配，之后进行详细说明。要匹配字符 `'('` 或者 `')'`, 用 `\(` 或 `\)`, 或者把它们包含在字符集合里: `[(]`, `[)]`.

(?…)

这是个扩展标记法（一个 `'?'` 跟随 `'('` 并无含义）。`'?'` 后面的第一个字符决定了这个构建采用什么样的语法。这种扩展通常并不创建新的组合；`(?P<name>...)` 是唯一的例外。以下是目前支持的扩展。

(?aiLmsux)

(一个或多个来自 `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'` 集合的字母。) 分组将与空字符串相匹配；这些字母将为整个正则表达式设置相应的旗标：

- *re.A* (仅限 ASCII 匹配)
- *re.I* (忽略大小写)
- *re.L* (依赖于语言区域)
- *re.M* (多行)
- *re.S* (点号匹配所有字符)
- *re.U* (Unicode 匹配)
- *re.X* (详细)

（该旗标在模块内容 中有介绍。）这适用于当你希望将该旗标包括为正则表达式的一部分，而不是向*re.compile()* 函数传入 *flag* 参数的情况。旗标应当在表达式字符串的开头使用。

在 3.11 版本发生变更: 此构造只能在表达式的开头使用。

(?:…)
正则括号的非捕获版本。匹配在括号内的任何正则表达式，但该分组所匹配的子字符串 不能在执行匹配后被获取或是之后在模式中被引用。

(?aiLmsux-imsx:…)
（零个或多个来自 'a', 'i', 'L', 'm', 's', 'u', 'x' 集合的字母，后面可以带 '-' 再跟一个或多个来自 'i', 'm', 's', 'x' 集合的字母。）这些字母将为这部分表达式设置或移除相应的旗标:

- *re.A* (仅限 ASCII 匹配)
- *re.I* (忽略大小写)
- *re.L* (依赖于语言区域)
- *re.M* (多行)
- *re.S* (点号匹配所有字符)
- *re.U* (Unicode 匹配)
- *re.X* (详细)

（这些旗标在模块内容 中有介绍。）

字母 'a', 'L' 和 'u' 在用作内联旗标时是互斥的，所以它们不能相互组合或者带 '-'。相反，当它们中的某一个出现于内联的分组时，它将覆盖外层分组中匹配的模式。在 Unicode 模式中 (?a:...) 将切换至仅限 ASCII 匹配，而 (?u:...) 将切换至 Unicode 匹配（默认）。在字节串模式中 (?L:...) 将切换为基于语言区域的匹配，而 (?a:...) 将切换为仅限 ASCII 匹配（默认）。这种覆盖将只在内联分组范围内生效，而在分组之外将恢复为原始的匹配模式。

Added in version 3.6.

在 3.7 版本发生变更: 符号 'a', 'L' 和 'u' 同样可以用在一个组合内。

(?>...)
尝试匹配 ... 就像它是一个单独的正则表达式，如果匹配成功，则继续匹配在它之后的剩余表达式。如果之后的表达式匹配失败，则栈只能回溯到 (?>...) 之前的点，因为一旦退出，这个被称为 *原子化分组* 的表达式将会丢弃其自身所有的栈点位。因此，(?>.*). 将永远不会匹配任何东西因为首先 .* 将匹配所有可能的字符，然后，由于没有任何剩余的字符可供匹配，最后的 . 将匹配失败。由于原子化分组中没有保存任何栈点位，并且在它之前也没有任何栈点位，因此整个表达式将匹配失败。

Added in version 3.11.

(?P<name>…)
与常规的圆括号类似，但分组所匹配到了子字符串可通过符号分组名称 *name* 来访问。分组名称必须是有效的 Python 标识符，并且在*bytes* 模式中它们只能包含 ASCII 范围内的字节值。每个分组名称在一个正则表达式中只能定义一次。一个符号分组同时也是一个编号分组，就像这个分组没有被命名过一样。

命名组合可以在三种上下文中引用。如果样式是 (?P<quote>['"]).*?(?P=quote) （也就是说，匹配单引号或者双引号括起来的字符串):

引用组合"quote" 的上下文	引用方法
在正则式自身内	• `(?P=quote)` (如示) • `\1`
处理匹配对象 *m*	• `m.group('quote')` • `m.end('quote')` (等)
传递到 `re.sub()` 里的 *repl* 参数中	• `\g<quote>` • `\g<1>` • `\1`

在 3.12 版本发生变更: 在*bytes* 模式中，分组 *name* 只能包含 ASCII 范围内的字节值 (`b'\x00'-b'\x7f'`)。

(?P=name)
反向引用一个命名组合；它匹配前面那个叫 *name* 的命名组中匹配到的串同样的字串。

(?#…)
注释；里面的内容会被忽略。

(?=…)
当 …匹配时，匹配成功，但不消耗字符串中的任何字符。这个叫做 *前视断言*（lookahead assertion）。比如，`Isaac (?=Asimov)` 将会匹配 `'Isaac '`，仅当其后紧跟 `'Asimov'`。

(?!…)
当 …不匹配时，匹配成功。这个叫 *否定型前视断言*（negative lookahead assertion）。例如，`Isaac (?!Asimov)` 将会匹配 `'Isaac '`，仅当它后面 *不是* `'Asimov'`。

(?<=…)
如果 `...` 的匹配内容出现在当前位置的左侧，则匹配。这叫做 *肯定型后视断言*（positive lookbehind assertion）。`(?<=abc)def` 将会在 `'abcdef'` 中找到一个匹配，因为后视会回退 3 个字符并检查内部表达式是否匹配。内部表达式（匹配的内容）必须是固定长度的，意思就是 `abc` 或 `a|b` 是允许的，但是 `a*` 和 `a{3,4}` 不可以。注意，以肯定型后视断言开头的正则表达式，匹配项一般不会位于搜索字符串的开头。很可能你应该使用*search()* 函数，而不是*match()* 函数:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

这个例子搜索一个跟随在连字符后的单词:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

在 3.5 版本发生变更: 添加定长组合引用的支持。

(?<!…)
如果 `...` 的匹配内容没有出现在当前位置的左侧，则匹配。这个叫做 *否定型后视断言*（negative lookbehind assertion）。类似于肯定型后视断言，内部表达式（匹配的内容）必须是固定长度的。以否定型后视断言开头的正则表达式，匹配项可能位于搜索字符串的开头。

(?(id/name)yes-pattern|no-pattern)
如果给定的 *id* 或 *name* 存在，将会尝试匹配 `yes-pattern`，否则就尝试匹配 `no-pattern`，`no-pattern` 可选，也可以被忽略。比如，`(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` 是一个 email 样式匹配，将匹配 `'<user@host.com>'` 或 `'user@host.com'`，但不会匹配 `'<user@host.com'`，也不会匹配 `'user@host.com>'`。

在 3.12 版本发生变更: 分组 *id* 只能包含 ASCII 数码。在*bytes* 模式中，分组 *name* 只能包含 ASCII 范围内的字节值 (`b'\x00'-b'\x7f'`)。

由 `'\'` 和一个字符组成的特殊序列在以下列出。如果普通字符不是 ASCII 数位或者 ASCII 字母，那么正则样式将匹配第二个字符。比如，`\$` 匹配字符 `'$'`.

\number

匹配数字代表的组合。每个括号是一个组合，组合从 1 开始编号。比如 `(.+) \1` 匹配 `'the the'` 或者 `'55 55'`, 但不会匹配 `'thethe'` (注意组合后面的空格)。这个特殊序列只能用于匹配前面 99 个组合。如果 *number* 的第一个数位是 0，或者 *number* 是三个八进制数，它将不会被看作是一个组合，而是八进制的数字值。在 `'['` 和 `']'` 字符集合内，任何数字转义都被看作是字符。

\A

只匹配字符串开始。

\b

匹配空字符串，但只在单词开始或结尾的位置。一个单词被定义为一个单词字符的序列。注意在通常情况下，`\b` 被定义为 `\w` 和 `\W` 字符之间的边界（反之亦然），或是 `\w` 和字符串开始或结尾之间的边界。这意味着 `r'\bat\b'` 将匹配 `'at'`, `'at.'`, `'(at)'` 和 `'as at ay'` 但不匹配 `'attempt'` 或 `'atlas'`。

Unicode (str) 模式中默认的单词类字符是 Unicode 字母数字和下划线，但这可以通过使用*ASCII* 旗标来改变。如果使用了*LOCALE* 旗标则单词边界将根据当前语言区域来确定。

> **备注**
>
> 在一个字符范围内，`\b` 代表退格符，以便与 Python 的字符串字面值保持兼容。

\B

匹配空字符串，但仅限于它 不在单词的开头或结尾的情况。这意味着 `r'at\B'` 将匹配 `'athens'`, `'atom'`, `'attorney'`，但不匹配 `'at'`, `'at.'` 或 `'at!'`。`\B` 与 `\b` 正相反，这样 Unicode (str) 模式中的单词类字符是 Unicode 字母数字或下划线，但这可以通过使用*ASCII* 旗标来改变。如果使用了*LOCALE* 旗标则单词边界将根据当前语言区域来确定。

> **备注**
>
> 请注意 `\B` 不会匹配空字符串，这与其他编程语言如 Perl 的 RE 实现不同。此行为是出于兼容性考虑而保留的。

\d

对于 Unicode (str) 样式:
匹配任意 Unicode 十进制数码（也就是说，任何属于 Unicode 字符类别 [Nd] 的字符）。这包括 `[0-9]`，还包括许多其他的数码类字符。

如果使用了*ASCII* 旗标则匹配 `[0-9]`

对于 8 位 (bytes) 样式:
匹配 ASCII 字符集内的任意十进制数码；这等价于 `[0-9]`。

\D

匹配不属于十进制数码的任意字符。这与 `\d` 正相反。

如果使用了*ASCII* 旗标则匹配 `[^0-9]`

\s

对于 Unicode (str) 样式:
匹配 Unicode 空白字符（如*str.isspace()* 所定义的）。这包括 `[ \t\n\r\f\v]`，还包括许多其他字符，例如许多语言中由排版规则约定的非中断空白字符。

如果使用了*ASCII* 旗标则匹配 `[ \t\n\r\f\v]`。

对于 8 位 (bytes) 样式：
匹配 ASCII 中的空白字符，就是 `[ \t\n\r\f\v]` 。

\S
匹配不属于空白字符的任意字符。这与 `\s` 正相反。

如果使用了*ASCII* 旗标则匹配 `[^ \t\n\r\f\v]`

\w

对于 Unicode (str) 样式：
匹配 Unicode 单词类字符；这包括所有 Unicode 字母数字类字符 (由*str.isalnum()* 定义)，以及下划线 (_)。

如果使用了*ASCII* 旗标则匹配 `[a-zA-Z0-9_]`。

对于 8 位 (bytes) 样式：
匹配在 ASCII 字符集中被视为字母数字的字符；这等价于 `[a-zA-Z0-9_]`。如果使用了*LOCALE* 旗标，则匹配在当前语言区域中被视为字母数字的字符以及下划线。

\W
匹配不属于单词类字符的任意字符。这与 `\w` 正相反。在默认情况下，将匹配除下划线 (_) 以外的*str.isalnum()* 返回 `False` 的字符。

如果使用了*ASCII* 旗标则匹配 `[^a-zA-Z0-9_]`。

如果使用了*LOCALE* 旗标，则匹配在当前语言区域中不属于字母数字且不为下划线的字符。

\Z
只匹配字符串尾。

Python 字符串字面值支持的大多数 转义序列也被正则表达式解析器所接受:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

（注意 `\b` 被用于表示词语的边界，它只在字符集合内表示退格，比如 `[\b]` 。）

`'\u'`, `'\U'` 和 `'\N'` 转义序列仅在 Unicode (str) 模式中可被识别。在字节串模式中它们会导致错误。未知的 ASCII 字母转义符被保留在未来使用并会被视为错误。

八进制转义包含为一个有限形式。如果首位数字是 0，或者有三个八进制数位，那么就认为它是八进制转义。其他的情况，就看作是组引用。对于字符串文本，八进制转义最多有三个数位长。

在 3.3 版本发生变更: 增加了 `'\u'` 和 `'\U'` 转义序列。

在 3.6 版本发生变更: 由 `'\'` 和一个 ASCII 字符组成的未知转义会被看成错误。

在 3.8 版本发生变更: 增加了 `'\N{name}'` 转义序列。与在字符串字面值中一样，它扩展了指定的 Unicode 字符 (例如 `'\N{EM DASH}'`)。

6.2.2 模块内容

模块定义了几个函数、常量，和一个异常。有些函数是编译后的正则表达式方法的简化版本（少了一些特性）。重要的应用程序大多会在使用前先编译正则表达式。

标志

在 3.6 版本发生变更: 标志常量现在是*RegexFlag* 类的实例，这个类是*enum.IntFlag* 的子类。

class re.**RegexFlag**
包含以下列出的正则表达式选项的*enum.IntFlag* 类。

Added in version 3.11: - added to __all__

re.**A**

re.**ASCII**

使 \w, \W, \b, \B, \d, \D, \s 和 \S 执行仅限 ASCII 匹配而不是完整的 Unicode 匹配。这仅对 Unicode (str) 模式有意义，而对字节串模式将被忽略。

对应于内联旗标 (?a)。

> **备注**
>
> *U* 旗标仍然存在以保持下向兼容性，但在 Python 3 中是多余的因为对于 str 模式默认使用 Unicode，并且 Unicode 匹配对于 bytes 模式则是不允许的。*UNICODE* 和内联旗标 (?u) 同样也是多余的。

re.**DEBUG**

显示有关被编译表达式的调试信息。

没有对应的内联旗标。

re.**I**

re.**IGNORECASE**

执行忽略大小写的匹配；[A-Z] 这样的表达式也将匹配小写字母。完全的 Unicode 匹配 (如 Ü 将匹配 ü) 同样适用，除非使用了*ASCII* 旗标来禁用非 ASCII 匹配。当前语言区域不会改变该旗标的效果，除非还使用了*LOCALE* 旗标。

对应于内联旗标 (?i)。

请注意当 Unicode 模式 [a-z] 或 [A-Z] 与*IGNORECASE* 旗标一起使用时，它们将匹配 52 个 ASCII 字母和 4 个额外的非 ASCII 字母: 'İ' (U+0130, 大写拉丁字母 I 带有上方的点), 'ı' (U+0131, 小写拉丁字母 i 不带上方的点), 'ſ' (U+017F, 小写拉丁字母长 s) 和'K' (U+212A, 开尔文标记)。如果使用了*ASCII* 旗标，则只匹配字母'a' 到'z' 和'A' 到'Z'。

re.**L**

re.**LOCALE**

使 \w, \W, \b, \B 和忽略大小写的匹配依赖于当前语言区域。该旗标仅适用于 bytes 模式。

对应于内联旗标 (?L)。

> **警告**
>
> 该旗标已不建议使用；请考虑改用 Unicode 匹配。语言区域机制相当不可靠因为它每次只能处理一种 “文化” 并且只适用于 8 位语言区域。Unicode (str) 模式默认启用 Unicode 匹配并且能够处理不同的语言区域和语言。

在 3.6 版本发生变更: *LOCALE* 仅适用于 bytes 模式并且不能兼容*ASCII*。

在 3.7 版本发生变更: 设置了*LOCALE* 旗标的已编译正则表达式对象不会再依赖于编译时的语言区域。只有在匹配时的语言区域才会影响匹配结果。

re.**M**

re.**MULTILINE**

在指定之后，模式字符 '^' 将匹配字符串的开始和每一行的开头（紧随在换行符之后）；而模式字符 '$' 将匹配字符串的末尾和每一行的末尾（紧接在换行符之前）。在默认情况下，'^' 只匹配字符串的开头，而 “'$'“ 只匹配字符串的末尾和紧接在字符串末尾（可能存在的）换行符之前。

对应于内联旗标 (?m)。

re.**NOFLAG**

表示未应用任何旗标，该值为 0。该旗标可被用作某个函数关键字参数的默认值或者用作将与其他旗标进行有条件 OR 运算的基准值。用作默认值的例子:

```
def myfunc(text, flag=re.NOFLAG):
    return re.match(text, flag)
```

Added in version 3.11.

re.**S**

re.**DOTALL**

使 '.' 特殊字符匹配任意字符，包括换行符；如果没有这个旗标，'.' 将匹配 除去换行符以外的任意字符。

对应于内联旗标 (?s)。

re.**U**

re.**UNICODE**

在 Python 3 中，str 模式默认将匹配 Unicode 字符。因此这个旗标多余且 **无任何效果**，仅保留用于向下兼容。

请参阅*ASCII* 了解如何改为仅限匹配 ASCII 字符。

re.**X**

re.**VERBOSE**

这个旗标允许你通过在视觉上分隔表达式的逻辑段落和添加注释来编写更为友好并更具可读性的正则表达式。表达式中的空白符会被忽略，除非是在字符类中，或前面有一个未转义的反斜杠，或者是在 *?, (?: 或 (?P<...> 等形符之内。例如，(? : 和 * ? 是不被允许的。当一个行内包含不在字符类中并且前面没有未转义反斜杠的 # 时，则从最左边的此 # 直至行尾的所有字符都会被忽略。

意思就是下面两个正则表达式等价地匹配一个十进制数字：

```
a = re.compile(r"""\d +  # 整数部分
                   \.    # 小数点
                   \d *  # 几个小数位""", re.X)
b = re.compile(r"\d+\.\d*")
```

对应内联标记 (?x) 。

函数

re.**compile**(*pattern*, *flags=0*)

将正则表达式的样式编译为一个*正则表达式对象* （正则对象），可以用于匹配，通过这个对象的方法*match()*, *search()* 以及其他如下描述。

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

序列

```
prog = re.compile(pattern)
result = prog.match(string)
```

等价于

```
result = re.match(pattern, string)
```

如果需要多次使用这个正则表达式的话，使用*re.compile()* 和保存这个正则对象以便复用，可以让程序更加高效。

> **备注**
>
> 通过*re.compile()* 编译后的样式，和模块级的函数会被缓存，所以少数的正则表达式使用无需考虑编译的问题。

re.**search**(*pattern*, *string*, *flags=0*)

扫描整个 *string* 查找正则表达式 *pattern* 产生匹配的第一个位置，并返回相应的`Match`。如果字符串中没有与模式匹配的位置则返回 None；请注意这不同于在字符串的某个位置上找到零长度匹配。

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

re.**match**(*pattern*, *string*, *flags=0*)

如果 *string* 开头的零个或多个字符与正则表达式 *pattern* 匹配，则返回相应的`Match`。如果字符串与模式不匹配则返回 None；请注意这与零长度匹配是不同的。

注意即便是`MULTILINE` 多行模式，`re.match()` 也只匹配字符串的开始位置，而不匹配每行开始。

如果你想定位 *string* 的任何位置，使用`search()` 来替代（也可参考*search() vs. match()* ）

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

re.**fullmatch**(*pattern*, *string*, *flags=0*)

如果整个 *string* 与正则表达式 *pattern* 匹配，则返回相应的`Match`。如果字符串与模式不匹配则返回 None；请注意这与零长度匹配是不同的。

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

Added in version 3.4.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

用 *pattern* 分开 *string* 。如果在 *pattern* 中捕获到括号，那么所有的组里的文字也会包含在列表里。如果 *maxsplit* 非零，最多进行 *maxsplit* 次分隔，剩下的字符全部返回到列表的最后一个元素。

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', maxsplit=1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

如果分隔符里有捕获组合，并且匹配到字符串的开始，那么结果将会以一个空字符串开始。对于结尾也是一样

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

这样的话，分隔组将会出现在结果列表中同样的位置。

样式的空匹配仅在与前一个空匹配不相邻时才会拆分字符串。

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

在 3.1 版本发生变更: 增加了可选标记参数。

在 3.7 版本发生变更: 增加了空字符串的样式分隔。

自 3.13 版本弃用: 以位置参数形式传入 *maxsplit* 和 *flags* 的做法已被弃用。在未来的 Python 版本中它们将为仅限关键字形参。

re.**findall**(*pattern*, *string*, *flags=0*)

返回 *pattern* 在 *string* 中的所有非重叠匹配，以字符串列表或字符串元组列表的形式。对 *string* 的扫描从左至右，匹配结果按照找到的顺序返回。空匹配也包括在结果中。

返回结果取决于模式中捕获组的数量。如果没有组，返回与整个模式匹配的字符串列表。如果有且仅有一个组，返回与该组匹配的字符串列表。如果有多个组，返回与这些组匹配的字符串元组列表。非捕获组不影响结果。

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

在 3.7 版本发生变更: 非空匹配现在可以在前一个空匹配之后出现了。

re.**finditer**(*pattern*, *string*, *flags=0*)

针对正则表达式 *pattern* 在 *string* 里的所有非重叠匹配返回一个产生*Match* 对象的*iterator*。*string* 将被从左至右地扫描，并且匹配也将按被找到的顺序返回。空匹配也会被包括在结果中。

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

在 3.7 版本发生变更: 非空匹配现在可以在前一个空匹配之后出现了。

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

返回通过使用 *repl* 替换在 *string* 最左边非重叠出现的 *pattern* 而获得的字符串。如果样式没有找到，则不加改变地返回 *string*。*repl* 可以是字符串或函数；如为字符串，则其中任何反斜杠转义序列都会被处理。也就是说，`\n` 会被转换为一个换行符，`\r` 会被转换为一个回车符，依此类推。未知的 ASCII 字符转义序列保留在未来使用，会被当作错误来处理。其他未知转义序列例如 `\&` 会保持原样。向后引用像是 `\6` 会用样式中第 6 组所匹配到的子字符串来替换。例如:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

如果 *repl* 是一个函数，则它会针对每次 *pattern* 的非重叠出现的情况被调用。该函数接受单个*Match* 参数，并返回替换字符串。例如:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
...
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

模式可以是一个字符串或者*Pattern*。

可选参数 *count* 是要替换的最大次数；*count* 必须是非负整数。如果省略这个参数或设为 0，所有的匹配都会被替换。样式的空匹配仅在与前一个空匹配不相邻时才会被替换，所以 `sub('x*', '-', 'abxd')` 返回 `'-a-b--d-'` 。

在字符串类型的 *repl* 参数里，如上所述的转义和向后引用中，`\g<name>` 会使用命名组合 `name`，(在 `(?P<name>…)` 语法中定义) `\g<number>` 会使用数字组；`\g<2>` 就是 `\2`，但它避免了二义性，如 `\g<2>0`。`\20` 就会被解释为组 20，而不是组 2 后面跟随一个字符 `'0'`。向后引用 `\g<0>` 把 *pattern* 作为一整个组进行引用。

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (| 运算符) 进行组合。

在 3.1 版本发生变更: 增加了可选标记参数。

在 3.5 版本发生变更: 不匹配的组合替换为空字符串。

在 3.6 版本发生变更: *pattern* 中的未知转义（由 `'\'` 和一个 ASCII 字符组成）被视为错误。

在 3.7 版本发生变更: 在 *repl* 中由 `'\'` 加一个 ASCII 字母组成的未知转义符号现在将导致错误。对该模式的空匹配在与之前的非空匹配相邻时将被替换。

在 3.12 版本发生变更: 分组 *id* 只能包含 ASCII 数码。在*bytes* 替换字符串中，分组 *name* 只能包含 ASCII 范围内的字节值 (`b'\x00'-b'\x7f'`)。

自 3.13 版本弃用: 以位置参数形式传入 *count* 和 *flags* 的做法已被弃用。在未来的 Python 版本中它们将为仅限关键字形参。

`re.`**`subn`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

行为与*sub()* 相同，但是返回一个元组（字符串，替换次数）.

表达式的行为可通过指定 *flags* 值来修改。值可以是任意*flags* 变量，可使用按位 OR (`|` 运算符) 进行组合。

`re.`**`escape`**(*pattern*)

转义 *pattern* 中的特殊字符。如果你想对任意可能包含正则表达式元字符的文本字符串进行匹配，它就是有用的。比如

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&\'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

这个函数不能被用于*sub()* 和*subn()* 的替换字符串，只有反斜杠应该被转义。例如:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

在 3.3 版本发生变更: `'_'` 不再被转义。

在 3.7 版本发生变更: 只有在正则表达式中具有特殊含义的字符才会被转义。因此，`'!'`, `'"'`, `'%'`, `"'"`, `','`, `'/'`, `':'`, `';'`, `'<'`, `'='`, `'>'`, `'@'` 和 `"`"` 将不再会被转义。

`re.`**`purge`**()

清除正则表达式的缓存。

异常

`exception` `re.`**`PatternError`**(*msg*, *pattern=None*, *pos=None*)

当传递给某个函数的字符串不是合法的正则表达式（例如，它可能包含不匹配的圆括号）或者当在编译或匹配期间出现其他错误时所引发的异常。如果字符串未包含对某个模式的匹配绝不会导致错误。`PatternError` 实例具有下列附加属性:

`msg`

未格式化的错误消息。

`pattern`

正则表达式的模式串。

pos

编译失败的 *pattern* 的位置索引（可以是 None ）。

lineno

对应 *pos* (可以是 None) 的行号。

colno

对应 *pos* (可以是 None) 的列号。

在 3.5 版本发生变更: 增加了额外的属性。

在 3.13 版本发生变更: PatternError 原名为 error；后者被保留作为一个别名用于向下兼容。

6.2.3 正则表达式对象（正则对象）

class re.**Pattern**

由*re.compile()* 返回的已编译正则表达式对象。

在 3.9 版本发生变更: *re.Pattern* 支持用 [] 表示 Unicode (str) 或字节串类型的模式。参见*GenericAlias 类型*。

Pattern.**search**(*string*[, *pos*[, *endpos*]])

扫描整个 *string* 查找该正则表达式产生匹配的第一个位置，并返回相应的*Match*。如果字符串中没有与模式匹配的位置则返回 None；请注意这不同于在字符串的某个位置上找到零长度匹配。

可选的第二个参数 *pos* 给出了字符串中开始搜索的位置索引；默认为 0，它不完全等价于字符串切片；'^' 样式字符匹配字符串真正的开头，和换行符后面的第一个字符，但不会匹配索引规定开始的位置。

可选参数 *endpos* 限定了字符串搜索的结束；它假定字符串长度到 *endpos* ，所以只有从 pos 到 endpos - 1 的字符会被匹配。如果 *endpos* 小于 *pos*，就不会有匹配产生；另外，如果 *rx* 是一个编译后的正则对象，rx.search(string, 0, 50) 等价于 rx.search(string[:50], 0)。

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # 在索引0处匹配
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # 没有匹配项；搜索不包括 “D”
```

Pattern.**match**(*string*[, *pos*[, *endpos*]])

如果字符串 开头的零个或多个字符与此正则表达式匹配，则返回相应的*Match*。如果字符串与模式不匹配则返回 None；请注意这与零长度匹配是不同的。

可选参数 *pos* 和 *endpos* 与*search()* 含义相同。

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # 不匹配因为 "o" 不在 "dog" 的开头。
>>> pattern.match("dog", 1)   # 匹配因为 "o" 是 "dog" 的第 2 个字符。
<re.Match object; span=(1, 2), match='o'>
```

如果你想定位匹配在 *string* 中的位置，使用*search()* 来替代（另参考*search() vs. match()*）。

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])

如果整个 *string* 与此正则表达式匹配，则返回相应的*Match*。如果字符串与模式不匹配则返回 None；请注意这与零长度匹配是不同的。

可选参数 *pos* 和 *endpos* 与*search()* 含义相同。

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # 不匹配因为 "o" 不在 "dog" 的开头。
>>> pattern.fullmatch("ogre")     # 不匹配因为不是整个字符串都匹配。
>>> pattern.fullmatch("doggie", 1, 3)   # 在给定的限制下匹配。
<re.Match object; span=(1, 3), match='og'>
```

Added in version 3.4.

Pattern.**split**(*string*, *maxsplit=0*)

等价于*split()* 函数，使用了编译后的样式。

Pattern.**findall**(*string*[, *pos*[, *endpos*]])

类似函数*findall()* ，使用了编译后样式，但也可以接收可选参数 *pos* 和 *endpos* ，限制搜索范围，就像*search()*。

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])

类似函数*finditer()* ，使用了编译后样式，但也可以接收可选参数 *pos* 和 *endpos* ，限制搜索范围，就像*search()*。

Pattern.**sub**(*repl*, *string*, *count=0*)

等价于*sub()* 函数，使用了编译后的样式。

Pattern.**subn**(*repl*, *string*, *count=0*)

等价于*subn()* 函数，使用了编译后的样式。

Pattern.**flags**

正则表达式匹配旗标。这是一个传给*compile()* 的旗标组合，模式中的任何 `(?...)` 内联旗标，以及隐式旗标如当模式为 Unicode 字符串时的*UNICODE*。

Pattern.**groups**

捕获到的模式串中组的数量。

Pattern.**groupindex**

映射由 `(?P<id>)` 定义的命名符号组合和数字组合的字典。如果没有符号组，那字典就是空的。

Pattern.**pattern**

编译对象的原始样式字符串。

在 3.7 版本发生变更: 添加*copy.copy()* 和*copy.deepcopy()* 函数的支持。编译后的正则表达式对象被认为是原子性的。

6.2.4 匹配对象

匹配对象总是有一个布尔值 `True`。如果没有匹配的话*match()* 和*search()* 返回 `None` 所以你可以简单的用 `if` 语句来判断是否匹配

```
match = re.search(pattern, string)
if match:
    process(match)
```

class re.**Match**

由成功的 `match` 和 `search` 所返回的匹配对象。

在 3.9 版本发生变更: *re.Match* 支持用 `[]` 表示 Unicode (str) 或字节串类型的匹配。参见*GenericAlias 类型*。

Match.**expand**(*template*)

返回通过在模板字符串 *template* 上执行反斜杠替换所获得的字符串，就像*sub()* 方法所做的那样。转义符例如 `\n` 将被转换为适当的字符，而数字反向引用 (`\1`, `\2`) 和命名反向引用 (`\g<1>`, `\g<name>`) 将被替换为相应分组的内容。反向引用 `\g<0>` 将被替换为整个匹配的内容。

在 3.5 版本发生变更: 不匹配的组合替换为空字符串。

Match.**group**([*group1*, ...])

返回一个或者多个匹配的子组。如果只有一个参数，结果就是一个字符串，如果有多个参数，结果就是一个元组（每个参数对应一个项），如果没有参数，组 1 默认到 0（整个匹配都被返回）。如果一个组 N 参数值为 0，相应的返回值就是整个匹配字符串；如果它是一个范围 [1..99]，结果就是相应的括号组字符串。如果一个组号是负数，或者大于样式中定义的组数，就引发一个*IndexError* 异常。如果一个组包含在样式的一部分，并被匹配多次，就返回最后一个匹配。:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # 整个匹配
'Isaac Newton'
>>> m.group(1)       # 第一个圆括号标记的子分组。
'Isaac'
>>> m.group(2)       # 第二个圆括号标记的子分组。
'Newton'
>>> m.group(1, 2)    # 使用多个参数则返回一个元组。
('Isaac', 'Newton')
```

如果正则表达式使用了 `(?P<name>...)` 语法，*groupN* 参数就也可能是命名组合的名字。如果一个字符串参数在样式中未定义为组合名，就引发一个*IndexError* 异常。

一个相对复杂的例子

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

命名组合同样可以通过索引值引用

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

如果一个组匹配成功多次，就只返回最后一个匹配

```
>>> m = re.match(r"(..)+", "a1b2c3")  # 匹配 3 次。
>>> m.group(1)                        # 只返回最后一个匹配。
'c3'
```

`Match.__getitem__`(*g*)

这个等价于 `m.group(g)`。这允许更方便的引用一个匹配

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # 整个匹配
'Isaac Newton'
>>> m[1]       # 第一个圆括号标记的子分组。
'Isaac'
>>> m[2]       # 第二个圆括号标记的子分组。
'Newton'
```

命名分组也是受支持的:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Isaac Newton")
>>> m['first_name']
'Isaac'
>>> m['last_name']
'Newton'
```

Added in version 3.6.

`Match.groups`(*default=None*)

返回一个元组，包含所有匹配的子组，在样式中出现的从 1 到任意多的组合。*default* 参数用于不参与匹配的情况，默认为 `None`。

例如:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

如果我们使小数点可选，那么不是所有的组都会参与到匹配当中。这些组合默认会返回一个 None，除非指定了 *default* 参数。

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # 第二个分组默认为 None。
('24', None)
>>> m.groups('0')   # 现在，第二个分组默认为 '0'。
('24', '0')
```

Match.**groupdict**(*default=None*)

返回一个字典，包含了所有的 命名子组。key 就是组名。*default* 参数用于不参与匹配的组合；默认为 None。例如

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.**start**([*group*])

Match.**end**([*group*])

返回 *group* 匹配到的字串的开始和结束标号。*group* 默认为 0（意思是整个匹配的子串）。如果 *group* 存在，但未产生匹配，就返回 -1。对于一个匹配对象 *m*，和一个未参与匹配的组 *g*，组 *g* (等价于 m.group(g)) 产生的匹配是

```
m.string[m.start(g):m.end(g)]
```

注意 m.start(group) 将会等于 m.end(group)，如果 *group* 匹配一个空字符串的话。比如，在 m = re.search('b(c?)', 'cba') 之后，m.start(0) 为 1, m.end(0) 为 2, m.start(1) 和 m.end(1) 都是 2, m.start(2) 引发一个 *IndexError* 异常。

这个例子会从 email 地址中移除掉 *remove_this*

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.**span**([*group*])

对于一个匹配 *m*，返回一个二元组 (m.start(group), m.end(group))。注意如果 *group* 没有在这个匹配中，就返回 (-1, -1)。*group* 默认为 0，就是整个匹配。

Match.**pos**

pos 的值，会传递给 *search()* 或 *match()* 的方法 a 正则对象。这个是正则引擎开始在字符串搜索一个匹配的索引位置。

Match.**endpos**

endpos 的值，会传递给 *search()* 或 *match()* 的方法 a 正则对象。这个是正则引擎停止在字符串搜索一个匹配的索引位置。

Match.**lastindex**

捕获组的最后一个匹配的整数索引值，或者 None 如果没有匹配产生的话。比如，对于字符串 'ab'，表达式 (a)b, ((a)(b)), 和 ((ab)) 将得到 lastindex == 1，而 (a)(b) 会得到 lastindex == 2。

Match.**lastgroup**

最后一个匹配的命名组名字，或者 None 如果没有产生匹配的话。

Match.**re**

返回产生这个实例的正则对象，这个实例是由正则对象的*match()* 或*search()* 方法产生的。

Match.**string**

传递到*match()* 或*search()* 的字符串。

在 3.7 版本发生变更: 添加了对*copy.copy()* 和*copy.deepcopy()* 的支持。匹配对象被看作是原子性的。

6.2.5 正则表达式例子

检查对子

在这个例子里，我们使用以下辅助函数来更好地显示匹配对象:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

假设你在写一个扑克程序，一个玩家的一手牌为五个字符的串，每个字符表示一张牌，”a” 就是 A, ”k” K，”q” Q, ”j” J, ”t” 为 10, ”2” 到”9” 表示 2 到 9。

要看给定的字符串是否有效，我们可以按照以下步骤

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # 有效。
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # 无效。
>>> displaymatch(valid.match("akt"))    # 无效。
>>> displaymatch(valid.match("727ak"))  # 有效。
"<Match: '727ak', groups=()>"
```

最后一手牌，"727ak"，包含了一个对子，或者两张同样数值的牌。要用正则表达式匹配它，应该使用向后引用如下

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # 成对的 7。
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # 没有成对。
>>> displaymatch(pair.match("354aa"))     # 成对的 a。
"<Match: '354aa', groups=('a',)>"
```

要找出对子由什么牌组成，开发者可以按照下面的方式来使用匹配对象的*group()* 方法:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# 会报错因为 re.match() 返回 None，它没有 group() 方法。
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

模拟 scanf()

目前 Python 没有 `scanf()` 的等价物。正则表达式通常比 `scanf()` 格式字符串更强大，但也更冗长。下表提供了 `scanf()` 格式符和正则表达式之间一些大致等价的映射。

scanf() 形符	正则表达式
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

从文件名和数字提取字符串

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

你应当这样使用 `scanf()` 格式

```
%s - %d errors, %d warnings
```

等价的正则表达式是：

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python 基于正则表达式提供了不同的原始操作：

- *re.match()* 只在字符串的开头位置检测匹配。
- *re.search()* 在字符串中的任何位置检测匹配（这也是 Perl 在默认情况下所做的）
- *re.fullmatch()* 检测整个字符串是否匹配

例如：

```
>>> re.match("c", "abcdef")    # 不匹配
>>> re.search("c", "abcdef")   # 匹配
<re.Match object; span=(2, 3), match='c'>
>>> re.fullmatch("p.*n", "python") # 匹配
<re.Match object; span=(0, 6), match='python'>
>>> re.fullmatch("r.*n", "python") # 不匹配
```

在*search()* 中，可以用 `'^'` 作为开始来限制匹配到字符串的首位

```
>>> re.match("c", "abcdef")    # 不匹配
>>> re.search("^c", "abcdef")  # 不匹配
>>> re.search("^a", "abcdef")  # 匹配
<re.Match object; span=(0, 1), match='a'>
```

注意*MULTILINE* 多行模式中函数*match()* 只匹配字符串的开始，但使用*search()* 和以 `'^'` 开始的正则表达式会匹配每行的开始

```
>>> re.match("X", "A\nB\nX", re.MULTILINE)  # 不匹配
>>> re.search("^X", "A\nB\nX", re.MULTILINE)  # 匹配
<re.Match object; span=(4, 5), match='X'>
```

制作一个电话本

split() 将字符串用参数传递的样式分隔开。这个方法对于转换文本数据到易读而且容易修改的数据结构，是很有用的，如下面的例子证明。

首先，这里是输入。它通常来自一个文件，这里我们使用三重引号字符串语法

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

条目用一个或者多个换行符分开。现在我们将字符串转换为一个列表，每个非空行都有一个条目:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

最终，将每个条目分割为一个由名字、姓氏、电话号码和地址组成的列表。我们为*split()* 使用了 `maxsplit` 形参，因为地址中包含有被我们作为分割模式的空格符:

```
>>> [re.split(":? ", entry, maxsplit=3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

`:?` 样式匹配姓后面的冒号，因此它不出现在结果列表中。如果 `maxsplit` 设置为 `4` ，我们还可以从地址中获取到房间号:

```
>>> [re.split(":? ", entry, maxsplit=4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

文字整理

sub() 替换字符串中出现的样式的每一个实例。这个例子证明了使用*sub()* 来整理文字，或者随机化每个字符的位置，除了首位和末尾字符

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
...
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

查找所有副词

findall() 匹配样式 所有的出现，不仅是像*search()* 中的第一个匹配。比如，如果一个作者希望找到文字中的所有副词，他可能会按照以下方法用*findall()*

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

查找所有的副词及其位置

如果想要获得比匹配文本更多的关于模式的所有匹配信息，则*finditer()* 会很有用处因为它提供了*Match* 对象而不是字符串。继续前面的例子，如果某位作者想要查找某段文本中的所有副词 以及它们的位置，可以按以下方式使用*finditer()*:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly\b", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

原始字符串标记

原始字符串记法 (`r"text"`) 保持正则表达式正常。否则，每个正则式里的反斜杠 (`'\'`) 都必须前缀一个反斜杠来转义。比如，下面两行代码功能就是完全一致的

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

当需要匹配一个字符反斜杠，它必须在正则表达式中转义。在原始字符串记法，就是 `r"\\"`。否则就必须用 `"\\\\"`，来表示同样的意思

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

写一个词法分析器

一个 词法器或词法分析器 分析字符串，并分类成目录组。这是写一个编译器或解释器的第一步。

文字目录是由正则表达式指定的。这个技术是通过将这些样式合并为一个主正则式，并且循环匹配来实现的

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # 整数或小数
        ('ASSIGN',   r':='),           # 赋值运算符
        ('END',      r';'),            # 语句结束符
```

(续下页)

(接上页)

```
        ('ID',       r'[A-Za-z]+'),    # 标识符
        ('OP',       r'[+\-*/]'),      # 算术运算符
        ('NEWLINE',  r'\n'),           # 行结束符
        ('SKIP',     r'[ \t]+'),       # 跳过空格和制表符
        ('MISMATCH', r'.'),            # 任何其他字符
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

该词法器产生以下的输出

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.3 difflib --- 计算差异的辅助工具

源代码: Lib/difflib.py

此模块提供用于比较序列的类和函数。例如，它可被用于比较文件，并可产生多种格式的不同文件差异信息，包括 HTML 和上下文以及统一的 diff 数据。有关比较目录和文件，另请参阅*filecmp* 模块。

class difflib.**SequenceMatcher**

这是一个灵活的类，可用于比较任何类型的序列对，只要序列元素为*hashable* 对象。其基本算法要早于由 Ratcliff 和 Obershelp 于 1980 年代末期发表并以“格式塔模式匹配”的夸张名称命名的算法，并且更加有趣一些。其思路是找到不包含“垃圾”元素的最长连续匹配子序列；所谓“垃圾”元素是指其在某种意义上没有价值，例如空白行或空白符。（处理垃圾元素是对 Ratcliff 和 Obershelp 算法的一个扩展。）然后同样的思路将递归地应用于匹配序列的左右序列片段。这并不能产生最小编辑序列，但确实能产生在人们看来“正确”的匹配。

耗时: 基本 Ratcliff-Obershelp 算法在最坏情况下为立方时间而在一般情况下为平方时间。*SequenceMatcher* 在最坏情况下为平方时间而在一般情况下的行为受到序列中有多少相同元素这一因素的微妙影响；在最佳情况下则为线性时间。

自动垃圾启发式计算: *SequenceMatcher* 支持使用启发式计算来自动将特定序列项视为垃圾。这种启发式计算会统计每个单独项在序列中出现的次数。如果某一项（在第一项之后）的重复次数超过序列长度的 1% 并且序列长度至少有 200 项，该项会被标记为“热门”并被视为序列匹配中的垃圾。这种启发式计算可以通过在创建*SequenceMatcher* 时将 `autojunk` 参数设为 `False` 来关闭。

在 3.2 版本发生变更: 增加了 *autojunk* 形参。

class difflib.**Differ**

这个类的作用是比较由文本行组成的序列，并产生可供人阅读的差异或增量信息。Differ 统一使用*SequenceMatcher* 来完成行序列的比较以及相似（接近匹配）行内部字符序列的比较。

Differ 增量的每一行均以双字母代码打头：

双字母代码	含意
`'- '`	行为序列 1 所独有
`'+ '`	行为序列 2 所独有
`'  '`	行在两序列中相同
`'? '`	行不存在于任一输入序列

以’?’ 打头的行尝试将视线🄵至行以外而不存在于任一输入序列的差异。如果序列包含空白符，例如空格、制表或换行则这些行可能会令人感到迷惑。

class difflib.**HtmlDiff**

这个类可用于创建 HTML 表格（或包含表格的完整 HTML 文件）以并排地逐行显示文本比较，行间与行外的更改将突出显示。此表格可以基于完全或上下文差异模式来生成。

这个类的构造函数：

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

初始化*HtmlDiff* 的实例。

tabsize 是一个可选关键字参数，指定制表位的间隔，默认值为 `8`。

wrapcolumn 是一个可选关键字参数，指定行文本自动打断并换行的列位置，默认值为 `None` 表示不自动换行。

linejunk 和 *charjunk* 均是可选关键字参数，会传入*ndiff()* (被*HtmlDiff* 用来生成并排显示的 HTML 差异)。请参阅*ndiff()* 文档了解参数默认值及其说明。

下列是公开的方法

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, ***, *charset='utf-8'*)

比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个完整 HTML 文件，其中包含各行差异的表格，行间与行外的更改将突出显示。

fromdesc 和 *todesc* 均是可选关键字参数，指定来源/目标文件的列标题字符串（默认均为空白字符串）。

context 和 *numlines* 均是可选关键字参数。当只要显示上下文差异时就将 *context* 设为 `True`，否则默认值 `False` 为显示完整文件。*numlines* 默认为 `5`。当 *context* 为 `True` 时 *numlines* 将控制围绕突出显示差异部分的上下文行数。当 *context* 为 `False` 时 *numlines* 将控制在使用"next" 超链接时突出显示差异部分之前所显示的行数（设为零则会导致"next" 超链接将下一个突出显示差异部分放在浏览器顶端，不添加任何前导上下文）。

> **备注**
>
> *fromdesc* 和 *todesc* 会被当作未转义的 HTML 来解读，当接收不可信来源的输入时应该适当地进行转义。

在 3.5 版本发生变更: 增加了 *charset* 关键字参数。HTML 文档的默认字符集从 `'ISO-8859-1'` 更改为 `'utf-8'`。

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个包含各行差异的完整 HTML 表格，行间与行外的更改将突出显示。

此方法的参数与*make_file()* 方法的相同。

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

比较 *a* 和 *b* (字符串列表)；返回上下文差异格式的增量信息 (一个产生增量行的*generator*)。

所谓上下文差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为之前/之后的样式。上下文行数由 *n* 设定，默认为三行。

默认情况下，差异控制行（以 `***` or `---` 表示）是通过末尾换行符来创建的。这样做的好处是从*io.IOBase.readlines()* 创建的输入将得到适用于*io.IOBase.writelines()* 的差异信息，因为输入和输出都带有末尾换行符。

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

上下文差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串默认为空。

```
>>> import sys
>>> from difflib import *
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py',
...                       tofile='after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

difflib.**get_close_matches**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

返回由最佳“近似”匹配构成的列表。*word* 为一个指定目标近似匹配的序列（通常为字符串），*possibilities* 为一个由用于匹配 *word* 的序列构成的列表（通常为字符串列表）。

可选参数 *n* (默认为 3) 指定最多返回多少个近似匹配；*n* 必须大于 0.

可选参数 *cutoff* (默认为 0.6) 是一个 [0, 1] 范围内的浮点数。与 *word* 相似度得分未达到该值的候选匹配将被忽略。

候选匹配中（不超过 *n* 个）的最佳匹配将以列表形式返回，按相似度得分排序，最相似的排在最前面。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

比较 *a* 和 *b* (字符串列表)；返回*Differ* 形式的增量信息 (一个产生增量行的*generator*)。

可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 None)：

linejunk: 此函数接受单个字符串参数，如果其为垃圾字符串则返回真值，否则返回假值。默认为 None。此外还有一个模块层级的函数*IS_LINE_JUNK()*，它会过滤掉没有可见字符的行，除非该行添加了至多一个井号符 ('#') -- 但是下层的*SequenceMatcher* 类会动态分析哪些行的重复频繁到足以形成噪音，这通常会比使用此函数的效果更好。

charjunk: 此函数接受一个字符（长度为 1 的字符串），如果其为垃圾字符则返回真值，否则返回假值。默认为模块层级的函数*IS_CHARACTER_JUNK()*，它会过滤掉空白字符（空格符或制表符；但包含换行符可不是个好主意！）。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

返回两个序列中产生增量的那一个。

给出一个由*Differ.compare()* 或*ndiff()* 产生的 序列，提取出来自文件 1 或 2 (*which* 形参) 的行，去除行前缀。

示例:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
```

(续下页)

(接上页)

```
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

比较 *a* 和 *b* (字符串列表)；返回统一差异格式的增量信息 (一个产生增量行的*generator*)。

所以统一差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为内联的样式 (而不是分开的之前/之后文本块)。上下文行数由 *n* 设定，默认为三行。

默认情况下，差异控制行 (以 `---`, `+++` 或 `@@` 表示) 是通过末尾换行符来创建的。这样做的好处是从*io.IOBase.readlines()* 创建的输入将得到适用于*io.IOBase.writelines()* 的差异信息，因为输入和输出都带有末尾换行符。

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

统一的差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串将默认为空。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile='after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

使用 *dfunc* 比较 *a* 和 *b* (字节串对象列表)；产生以 *dfunc* 所返回格式表示的差异行列表（也是字节串）。*dfunc* 必须是可调用对象，通常为*unified_diff()* 或*context_diff()*。

允许你比较编码未知或不一致的数据。除 *n* 之外的所有输入都必须为字节串对象而非字符串。作用方式为无损地将所有输入 (除 *n* 之外) 转换为字符串，并调用 `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`。*dfunc* 的输出会被随即转换回字节串，这样你所得到的增量行将具有与 *a* 和 *b* 相同的未知/不一致编码。

Added in version 3.5.

difflib.**IS_LINE_JUNK**(*line*)

对于可忽略的行返回 `True`。如果 *line* 为空行或只包含单个 `'#'` 则 *line* 行就是可忽略的，否则就是不可忽略的。此函数被用作较旧版本*ndiff()* 中 *linejunk* 形参的默认值。

difflib.**IS_CHARACTER_JUNK**(*ch*)

对于可忽略的字符返回 `True`。字符 *ch* 如果为空格符或制表符则 *ch* 就是可忽略的，否则就是不可忽略的。此函数被用作*ndiff()* 中 *charjunk* 形参的默认值。

参见

Pattern Matching: The Gestalt Approach
John W. Ratcliff 和 D. E. Metzener 对于一种类似算法的讨论。此文于 1988 年 7 月发表于 Dr. Dobb's Journal。

6.3.1 SequenceMatcher 对象

SequenceMatcher 类具有这样的构造器：

class difflib.**SequenceMatcher**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

可选参数 *isjunk* 必须为 None (默认值) 或为接受一个序列元素并当且仅当其为应忽略的“垃圾”元素时返回真值的单参数函数。传入 None 作为 *isjunk* 的值就相当于传入 `lambda x: False`；也就是说不忽略任何值。例如，传入：

```
lambda x: x in " \t"
```

如果你以字符序列的形式对行进行比较，并且不希望区分空格符或硬制表符。

可选参数 *a* 和 *b* 为要比较的序列；两者默认为空字符串。两个序列的元素都必须为*hashable*。

可选参数 *autojunk* 可用于启用自动垃圾启发式计算。

在 3.2 版本发生变更: 增加了 *autojunk* 形参。

SequenceMatcher 对象接受三个数据属性: *bjunk* 是 *b* 当中 *isjunk* 为 True 的元素集合；*bpopular* 是被启发式计算（如果其未被禁用）视为热门候选的非垃圾元素集合；*b2j* 是将 *b* 当中剩余元素映射到一个它们出现位置列表的字典。所有三个数据属性将在 *b* 通过*set_seqs()* 或*set_seq2()* 重置时被重置。

Added in version 3.2: *bjunk* 和 *bpopular* 属性。

SequenceMatcher 对象具有以下方法：

set_seqs(*a*, *b*)

设置要比较的两个序列。

SequenceMatcher 计算并缓存有关第二个序列的详细信息，这样如果你想要将一个序列与多个序列进行比较，可使用*set_seq2()* 一次性地设置该常用序列并重复地对每个其他序列各调用一次*set_seq1()*。

set_seq1(*a*)

设置要比较的第一个序列。要比较的第二个序列不会改变。

set_seq2(*b*)

设置要比较的第二个序列。要比较的第一个序列不会改变。

find_longest_match(*alo=0*, *ahi=None*, *blo=0*, *bhi=None*)

找出 `a[alo:ahi]` 和 `b[blo:bhi]` 中的最长匹配块。

如果 *isjunk* 被省略或为 None，*find_longest_match()* 将返回 `(i, j, k)` 使得 `a[i:i+k]` 等于 `b[j:j+k]`，其中 `alo <= i <= i+k <= ahi` 并且 `blo <= j <= j+k <= bhi`。对于所有满足这些条件的 `(i', j', k')`，如果 `i == i', j <= j'` 也被满足，则附加条件 `k >= k', i <= i'`。换句话说，对于所有最长匹配块，返回在 *a* 当中最先出现的一个，而对于在 *a* 当中最先出现的所有最长匹配块，则返回在 *b* 当中最先出现的一个。

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

如果提供了 *isjunk*，将按上述规则确定第一个最长匹配块，但额外附加不允许块内出现垃圾元素的限制。然后将通过（仅）匹配两边的垃圾元素来尽可能地扩展该块。这样结果块绝对不会匹配垃圾元素，除非同样的垃圾元素正好与有意义的匹配相邻。

这是与之前相同的例子，但是将空格符视为垃圾。这将防止 `' abcd'` 直接与第二个序列末尾的 `' abcd'` 相匹配。而只可以匹配 `'abcd'`，并且是匹配第二个序列最左边的 `'abcd'`：

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

如果未找到匹配块，此方法将返回 `(alo, blo, 0)`。

此方法将返回一个*named tuple* `Match(a, b, size)`。

在 3.9 版本发生变更: 加入默认参数。

get_matching_blocks()

返回描述非重叠匹配子序列的三元组列表。每个三元组的形式为 `(i, j, n)`，其含义为 `a[i:i+n] == b[j:j+n]`。这些三元组按 *i* 和 *j* 单调递增排列。

最后一个三元组用于占位，其值为 `(len(a), len(b), 0)`。它是唯一 `n == 0` 的三元组。如果 `(i, j, n)` 和 `(i', j', n')` 是在列表中相邻的三元组，且后者不是列表中的最后一个三元组，则 `i+n < i'` 或 `j+n < j'`；换句话说，相邻的三元组总是描述非相邻的相等块。

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

返回描述如何将 *a* 变为 *b* 的 5 元组列表，每个元组的形式为 `(tag, i1, i2, j1, j2)`。在第一个元组中 `i1 == j1 == 0`，而在其余的元组中 *i1* 等于前一个元组的 *i2*，并且 *j1* 也等于前一个元组的 *j2*。

tag 值为字符串，其含义如下:

值	含意
`'replace'`	`a[i1:i2]` 应由 `b[j1:j2]` 替换。
`'delete'`	`a[i1:i2]` 应被删除。请注意在此情况下 `j1 == j2`。
`'insert'`	`b[j1:j2]` 应插入到 `a[i1:i1]`。请注意在此情况下 `i1 == i2`。
`'equal'`	`a[i1:i2] == b[j1:j2]` (两个子序列相同)。

例如:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

返回一个带有最多 *n* 行上下文的分组的*generator*。

从*get_opcodes()* 所返回的组开始，此方法会拆分出较小的更改簇并消除没有更改的间隔区域。

这些分组以与*get_opcodes()* 相同的格式返回。

ratio()

返回一个取值范围 [0, 1] 的浮点数作为序列相似性度量。

其中 T 是两个序列中元素的总数量，M 是匹配的数量，即 2.0*M / T。请注意如果两个序列完全相同则该值为 `1.0`，如果两者完全不同则为 `0.0`。

如果*get_matching_blocks()* 或*get_opcodes()* 尚未被调用则此方法运算消耗较大，在此情况下你可能需要先调用*quick_ratio()* 或*real_quick_ratio()* 来获取一个上界。

> **备注**
>
> 注意: *ratio()* 调用的结果可能会取决于参数的顺序。例如:
>
> ```
> >>> SequenceMatcher(None, 'tide', 'diet').ratio()
> 0.25
> >>> SequenceMatcher(None, 'diet', 'tide').ratio()
> 0.5
> ```

quick_ratio()

相对快速地返回一个*ratio()* 的上界。

real_quick_ratio()

非常快速地返回一个*ratio()* 的上界。

这三个返回匹配部分点总字符数之比的三种方法可能由于不同的近似级别而给出不同的结果，但是*quick_ratio()* 和*real_quick_ratio()* 总是会至少与*ratio()* 一样大:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher 的示例

以下示例比较两个字符串，并将空格视为“垃圾”:

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() 返回一个 [0, 1] 范围内的浮点数，用来衡量序列的相似度。根据经验，*ratio()* 值超过 0.6 就意味着两个序列非常接近匹配:

```
>>> print(round(s.ratio(), 3))
0.866
```

如果您只对序列的匹配的位置感兴趣，则*get_matching_blocks()* 就很方便:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

请注意*get_matching_blocks()* 返回的最后一个元组 (len(a), len(b), 0) 始终只用于占位，这也是元组的末尾元素（匹配的元素个数）为 0 的唯一情况。

如果你想要知道如何将第一个序列转成第二个序列，可以使用*get_opcodes()*:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

参见

- 此模块中的*get_close_matches()* 函数显示了如何基于*SequenceMatcher* 构建简单的代码来执行有用的功能。
- 针对使用*SequenceMatcher* 构建的小型应用程序的 简易版本控制方案。

6.3.3 Differ 对象

请注意*Differ* 所生成的增量并不保证是 **最小**差异。相反，最小差异往往是违反直觉的，因为它们会同步任何可能的地方，有时甚至意外产生相距 100 页的匹配。将同步点限制为连续匹配保留了一些局部性概念，这偶尔会带来产生更长差异的代价。

Differ 类具有这样的构造器：

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)

可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 None)：

linejunk: 接受单个字符串作为参数的函数，如果其为垃圾字符串则返回真值。默认值为 None，意味着没有任何行会被视为垃圾行。

charjunk: 接受单个字符（长度为 1 的字符串）作为参数的函数，如果其为垃圾字符则返回真值。默认值为 None，意味着没有任何字符会被视为垃圾字符。

这些垃圾过滤函数可加快查找差异的匹配速度，并且不会导致任何差异行或字符被忽略。请阅读*find_longest_match()* 方法的 *isjunk* 形参的描述了解详情。

Differ 对象是通过一个单独方法来使用（生成增量）的：

compare(*a*, *b*)

比较两个由行组成的序列，并生成增量（一个由行组成的序列）。

每个序列必须包含一个以换行符结尾的单行字符串。这样的序列可以通过文件型对象的*readlines()* 方法来获取。所生成的增量同样由以换行符结尾的字符串构成，可以通过文件型对象的*writelines()* 方法原样打印出来。

6.3.4 Differ 示例

此示例比较两段文本。首先我们设置文本为以换行符结尾的单行字符串组成的序列（这样的序列也可以通过文件型对象的*readlines()* 方法来获取）：

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

接下来我们实例化一个 Differ 对象：

```
>>> d = Differ()
```

请注意在实例化*Differ* 对象时我们可以传入函数来过滤掉“垃圾”行和字符。详情参见*Differ()* 构造器说明。

最后，我们比较两个序列：

```
>>> result = list(d.compare(text1, text2))
```

result 是一个字符串列表，让我们将其美化打印出来：

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                         ^\n',
 '+   5. Flat is better than nested.\n']
```

作为单独的多行字符串显示出来则是这样：

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?           ++++ ^                         ^
+   5. Flat is better than nested.
```

6.3.5 difflib 的命令行接口

这个例子演示了如何使用 difflib 来创建类似 diff 的工具。

```
""" difflib.py 的命令行接口，提供四种格式的 diff:

* ndiff:    列出每一行并高亮各行间的变化。
* context:  以前/后格式高亮变化内容集。
* unified:  以内联格式高亮变化内容集。
* html:     生成高亮变化内容的并排比较。

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
```

(续下页)

(接上页)

```
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate, todate,␣
↪n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate, todate,␣
↪n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.3.6 ndiff 示例

这个例子演示了如何使用*difflib.ndiff()*。

```
"""ndiff [-q] file1 file2
    或
ndiff (-r1 | -r2) < ndiff_output > file1_or_file2

打印对人类友好的文件差异报告至标准输出。 将会标明
行间与行内差异。 在第二种形式下，会根据标准输入上的
ndiff 报告在标准输出上重建 file1 (-r1) 或 file2 (-r2)。

在第一种形式下，如果未指明 -q ("quiet")，则输出的
前两行为

-: file1
+: file2

其余的每一行将以两个字符的代码打头：

    "- "    行不同于 file1
```

(续下页)

(接上页)

```
    "+ "    行不同于 file2
    "  "    行在两个文件中相同
    "? "    行不存在于某一个输入文件

以 "? " 打头的行会尝试关注行内差异，并且
不存在于某一个输入文件中。 当源文件包含
制表符时这些行可能会令人迷惑。

第一个文件可通过只保留以" " 或 "- " 打头的行，
并删除那些 2 字符前缀来恢复；使用 ndiff 时传入 -r1。

第二个文件可通过类似方式来恢复，即只保留
" " 和 "+ " 打头的行；使用 ndiff 并传入 -r2；或者
在 Unix 上，第二个文件可通过管道操作来恢复

    sed -n '/^[+ ] /s/^..//p'
"""

__version__ = 1, 7, 0

import difflib, sys

def fail(msg):
    out = sys.stderr.write
    out(msg + "\n\n")
    out(__doc__)
    return 0

# 打开一个文件并返回文件对象；如无法打开文件
# 则返回 0
def fopen(fname):
    try:
        return open(fname)
    except IOError as detail:
        return fail("couldn't open " + fname + ": " + str(detail))

# 打开两个文件并报告差异至标准输出；如有问题则返回假值
def fcompare(f1name, f2name):
    f1 = fopen(f1name)
    f2 = fopen(f2name)
    if not f1 or not f2:
        return 0

    a = f1.readlines(); f1.close()
    b = f2.readlines(); f2.close()
    for line in difflib.ndiff(a, b):
        print(line, end=' ')

    return 1

# 解析参数 (sys.argv[1:] 正常) 并进行比较；
# 如有问题则返回假值

def main(args):
    import getopt
    try:
        opts, args = getopt.getopt(args, "qr:")
    except getopt.error as detail:
        return fail(str(detail))
    noisy = 1
    qseen = rseen = 0
```

(续下页)

(接上页)

```
    for opt, val in opts:
        if opt == "-q":
            qseen = 1
            noisy = 0
        elif opt == "-r":
            rseen = 1
            whichfile = val
    if qseen and rseen:
        return fail("can't specify both -q and -r")
    if rseen:
        if args:
            return fail("no args allowed with -r option")
        if whichfile in ("1", "2"):
            restore(whichfile)
            return 1
        return fail("-r value must be 1 or 2")
    if len(args) != 2:
        return fail("need 2 filename args")
    f1name, f2name = args
    if noisy:
        print('-:', f1name)
        print('+:', f2name)
    return fcompare(f1name, f2name)

# 从标准输入读取 ndiff 输出，并打印 file1 (which=='1') 或
# file2 (which=='2') 到标准输出

def restore(which):
    restored = difflib.restore(sys.stdin.readlines(), which)
    sys.stdout.writelines(restored)

if __name__ == '__main__':
    main(sys.argv[1:])
```

6.4 textwrap --- 文本自动换行与填充

源代码: Lib/textwrap.py

---

textwrap 模块提供了一些快捷函数，以及可以完成所有工作的类*TextWrapper*。如果你只是要对一两个文本字符串进行自动换行或填充，快捷函数应该就够用了；否则的话，你应该使用*TextWrapper* 的实例来提高效率。

textwrap.**wrap**(*text*, *width=70*, *, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为 *width* 个字符。返回由输出行组成的列表，行尾不带换行符。

与*TextWrapper* 的实例属性对应的可选的关键字参数，具体文档见下。

请参阅*TextWrapper.wrap()* 方法了解有关*wrap()* 行为的详细信息。

textwrap.**fill**(*text*, *width=70*, *, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

对 *text* 中的单独段落自动换行，并返回一个包含被自动换行段落的单独字符串。*fill()* 是以下语句的快捷方式

```
"\n".join(wrap(text, ...))
```

特别要说明的是，*fill()* 接受与*wrap()* 完全相同的关键字参数。

textwrap.**shorten**(*text*, *width*, *, *fix_sentence_endings=False*, *break_long_words=True*, *break_on_hyphens=True*, *placeholder='[...]'*)

折叠并截短给定的 *text* 以符合给定的 *width*。

首先 *text* 中的空格会被折叠（所有连续会替换为单个空格）。如果结果能适合 *width*，它将被返回。在其他情况下，将在末尾丢弃足够数量的单词以使剩余的单词加 *placeholder* 能适合 *width*:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

可选的关键字参数对应于*TextWrapper* 的实际属性，具体见下文。请注意文本在被传入*TextWrapper* 的*fill()* 函数之前会被折叠，因此改变*tabsize*，*expand_tabs*，*drop_whitespace* 和*replace_whitespace* 的值将没有任何效果。

Added in version 3.4.

textwrap.**dedent**(*text*)

移除 *text* 中每一行的任何相同前缀空白符。

这可以用来清除三重引号字符串行左侧空格，而仍然在源码中显示为缩进格式。

请注意制表符和空格符都被视为是空白符，但它们并不相等：以下两行 `"  hello"` 和 `"\thello"` 不会被视为具有相同的前缀空白符。

只包含空白符的行会在输入时被忽略并在输出时被标准化为单个换行符。

例如:

```
def test():
    # 第一行以 \ 结束以避免出现空行!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)

将 *prefix* 添加到 *text* 中选定行的开头。

通过调用 `text.splitlines(True)` 来对行进行拆分。

默认情况下，*prefix* 会被添加到所有不是只由空白符（包括任何行结束符）组成的行。

例如:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

可选的 *predicate* 参数可用来控制哪些行要缩进。例如，可以很容易地为空行或只有空白符的行添加 *prefix*:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
```

(续下页)

(接上页)

```
+
+ world
```

Added in version 3.3.

wrap(), *fill()* 和*shorten()* 的作用方式为创建一个*TextWrapper* 实例并在其上调用单个方法。该实例不会被重用，因此对于要使用*wrap()* 和/或*fill()* 来处理许多文本字符串的应用来说，创建你自己的*TextWrapper* 对象可能会更有效率。

文本最好在空白符位置自动换行，包括带连字符单词的连字符之后；长单词仅在必要时会被拆分，除非*TextWrapper.break_long_words* 被设为假值。

class textwrap.**TextWrapper**(***kwargs*)

TextWrapper 构造器接受多个可选的关键字参数。每个关键字参数对应一个实例属性，比如说

```
wrapper = TextWrapper(initial_indent="* ")
```

相当于：

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

你可以多次重用相同的*TextWrapper* 对象，并且你也可以在使用期间通过直接向实例属性赋值来修改它的任何选项。

TextWrapper 的实例属性（以及构造器的关键字参数）如下所示：

width

(默认: 70) 自动换行的最大行长度。只要输入文本中没有长于*width* 的单个单词，*TextWrapper* 就能保证没有长于*width* 个字符的输出行。

expand_tabs

(默认值: True) 如果为真值，则 *text* 中的所有制表符将使用 *text* 的*expandtabs()* 方法扩展为空格符。

tabsize

(默认: 8) 如果*expand_tabs* 为真值，则 *text* 中所有的制表符将扩展为零个或多个空格，具体取决于当前列位置和给定的制表宽度。

Added in version 3.3.

replace_whitespace

(default: True) 如果为真值，在制表符扩展之后、自动换行之前，*wrap()* 方法将把每个空白字符都替换为单个空格。会被替换的空白字符如下：制表，换行，垂直制表，进纸和回车 ('\t\n\v\f\r')。

> **备注**
>
> 如果*expand_tabs* 为假值且*replace_whitespace* 为真值，每个制表符将被替换为单个空格，这与制表符扩展是 不一样的。

> **备注**
>
> 如果*replace_whitespace* 为假值，在一行的中间有可能出现换行符并导致怪异的输出。因此，文本应当（使用*str.splitlines()* 或类似方法）拆分为段落并分别进行自动换行。

drop_whitespace

(默认: `True`) 如果为真值，每一行开头和末尾的空白字符（在包装之后、缩进之前）会被丢弃。但是段落开头的空白字符如果后面不带任何非空白字符则不会被丢弃。如果被丢弃的空白字符占据了一个整行，则该整行将被丢弃。

initial_indent

(默认: `''`) 将被添加到被自动换行输出内容的第一行的字符串。其长度会被计入第一行的长度。空字符串不会被缩进。

subsequent_indent

(default: `''`) 将被添加到被自动换行输出内容除第一行外的所有行的字符串。其长度会被计入除行一行外的所有行的长度。

fix_sentence_endings

(默认: `False`) 如果为真值，*TextWrapper* 将尝试检测句子结尾并确保句子间总是以恰好两个空格符分隔。对于使用等宽字体的文本来说通常都需要这样。但是句子检测算法并不完美：它假定句子结尾是一个小写字母加字符 `'.'`, `'!'` 或 `'?'` 之一，并可能跟一个 `'"'` 或 `"'"`，再跟一个空格。此算法的一个问题是它无法区分以下文本中的"Dr."

```
[...] Dr. Frankenstein's monster [...]
```

和以下文本中的"Spot."

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings 默认为假值。

由于句子检测算法依赖于 `string.lowercase` 来确定“小写字母”，以及约定在句点后使用两个空格来分隔处于同一行的句子，因此只适用于英语文本。

break_long_words

(默认: `True`) 如果为真值，则长度超过*width* 的单词将被分开以保证行的长度不会超过*width*。如果为假值，超长单词不会被分开，因而某些行的长度可能会超过*width*。(超长单词将被单独作为一行，以尽量减少超出*width* 的情况。)

break_on_hyphens

(默认: `True`) 如果为真值，将根据英语的惯例首选在空白符和复合词的连字符之后自动换行。如果为假值，则只有空白符会被视为合适的潜在断行位置，但如果你确实不希望出现分开的单词则你必须将*break_long_words* 设为假值。之前版本的默认行为总是允许分开带有连字符的单词。

max_lines

(默认: `None`) 如果不为 `None`，则输出内容将最多包含 *max_lines* 行，并使 *placeholder* 出现在输出内容的末尾。

Added in version 3.4.

placeholder

(默认: `' [...]'`) 该文本将在输出文本被截短时出现在文本末尾。

Added in version 3.4.

TextWrapper 还提供了一些公有方法，类似于模块层级的便捷函数：

wrap(*text*)

对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为*width* 个字符。所有自动换行选项均获取自*TextWrapper* 实例的实例属性。返回由输出行组成的列表，行尾不带换行符。如果自动换行输出结果没有任何内容，则返回空列表。

fill(*text*)

对 *text* 中的单独段落自动换行并返回包含被自动换行段落的单独字符串。

6.5 unicodedata --- Unicode 数据库

此模块提供了对 Unicode Character Database (UCD) 的访问，其中定义了所有 Unicode 字符的字符属性。此数据库中包含的数据编译自 UCD 版本 15.1.0。

该模块使用与 Unicode 标准附件 #44 “Unicode 字符数据库” 中所定义的相同名称和符号。它定义了以下函数：

unicodedata.**lookup**(*name*)

按名称查找字符。如果找到具有给定名称的字符，则返回相应的字符。如果没有找到，则*KeyError* 被引发。

在 3.3 版本发生变更: 已添加对名称别名[1] 和命名序列[2] 的支持。

unicodedata.**name**(*chr*[, *default*])

返回分配给字符 *chr* 的名称作为字符串。如果没有定义名称，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**decimal**(*chr*[, *default*])

返回分配给字符 *chr* 的十进制值作为整数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**digit**(*chr*[, *default*])

返回分配给字符 *chr* 的数字值作为整数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**numeric**(*chr*[, *default*])

返回分配给字符 *chr* 的数值作为浮点数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**category**(*chr*)

返回分配给字符 *chr* 的常规类别为字符串。

unicodedata.**bidirectional**(*chr*)

返回分配给字符 *chr* 的双向类作为字符串。如果未定义此类值，则返回空字符串。

unicodedata.**combining**(*chr*)

返回分配给字符 *chr* 的规范组合类作为整数。如果没有定义组合类，则返回 `0` 。

unicodedata.**east_asian_width**(*chr*)

返回分配给字符 *chr* 的东亚宽度作为字符串。

unicodedata.**mirrored**(*chr*)

返回分配给字符 *chr* 的镜像属性为整数。如果字符在双向文本中被识别为“镜像”字符，则返回 `1` ，否则返回 `0` 。

unicodedata.**decomposition**(*chr*)

返回分配给字符 *chr* 的字符分解映射作为字符串。如果未定义此类映射，则返回空字符串。

unicodedata.**normalize**(*form*, *unistr*)

返回 Unicode 字符串 *unistr* 的正常形式 *form* 。*form* 的有效值为'NFC' 、'NFKC' 、'NFD' 和'NFKD' 。

Unicode 标准基于规范等价和兼容性等效的定义定义了 Unicode 字符串的各种规范化形式。在 Unicode 中，可以以各种方式表示多个字符。例如，字符 U+00C7（带有 CEDILLA 的 LATIN CAPITAL LETTER C ）也可以表示为序列 U+0043（LATIN CAPITAL LETTER C ）U+0327（COMBINING CEDILLA ）。

[1] https://www.unicode.org/Public/15.1.0/ucd/NameAliases.txt
[2] https://www.unicode.org/Public/15.1.0/ucd/NamedSequences.txt

对于每个字符，有两种正规形式：正规形式 C 和正规形式 D 。正规形式 D（NFD）也称为规范分解，并将每个字符转换为其分解形式。正规形式 C（NFC）首先应用规范分解，然后再次组合预组合字符。

除了这两种形式之外，还有两种基于兼容性等效的其他常规形式。在 Unicode 中，支持某些字符，这些字符通常与其他字符统一。例如，U+2160（ROMAN NUMERAL ONE）与 U+0049（LATIN CAPITAL LETTER I）完全相同。但是，Unicode 支持它与现有字符集（例如 gb2312 ）的兼容性。

正规形式 KD（NFKD）将应用兼容性分解，即用其等价项替换所有兼容性字符。正规形式 KC（NFKC）首先应用兼容性分解，然后是规范组合。

即使两个 unicode 字符串被规范化并且人类读者看起来相同，如果一个具有组合字符而另一个没有，则它们可能无法相等。

unicodedata.**is_normalized**(*form*, *unistr*)

判断 Unicode 字符串 *unistr* 是否为正规形式 *form*。*form* 的有效值为'NFC', 'NFKC', 'NFD' 和'NFKD'。

Added in version 3.8.

此外，该模块暴露了以下常量：

unicodedata.**unidata_version**

此模块中使用的 Unicode 数据库的版本。

unicodedata.**ucd_3_2_0**

这是一个与整个模块具有相同方法的对象，但对于需要此特定版本的 Unicode 数据库（如 IDNA ）的应用程序，则使用 Unicode 数据库版本 3.2 。

示例：

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

备注

6.6 stringprep --- 因特网字符串预处理

源代码: Lib/stringprep.py

在标识因特网上的事物（例如主机名），经常需要比较这些标识是否（相等）。这种比较的具体执行可能会取决于应用域的不同，例如是否要区分大小写等等。有时也可能需要限制允许的标识为仅由“可打印”字符组成。

RFC 3454 定义了在因特网协议中 Unicode 字符串的“预备”过程。在将字符串连线传输之前，它们会先使用预备过程进行处理，之后它们将具有特定的标准形式。该 RFC 定义了一系列表格，它们可以被组合为选项配置。每个配置必须定义所使用的表格，stringprep 过程的其他可选项也是配置的组成部分。stringprep 配置的一个例子是 nameprep，它被用于国际化域名。

stringprep 模块只公开了来自 **RFC 3454** 的表格。由于以字典或列表形式表示这些表格将会非常庞大，因此该模块在内部使用 Unicode 字符数据库。该模块本身的源代码是使用 `mkstringprep.py` 工具生成的。

因此，这些表格以函数而非数据结构的形式公开。在 RFC 中有两种表格：集合与映射。对于集合，*stringprep* 提供了“特征函数”，即如果形参是集合的一部分则返回值为 `True` 的函数。对于映射，它提供了映射函数：它会根据给定的键返回所关联的值。以下是模块中所有可用函数的列表。

stringprep.**in_table_a1**(*code*)
确定 *code* 是否属于 tableA.1 (Unicode 3.2 中的未分配码位)。

stringprep.**in_table_b1**(*code*)
确定 *code* 是否属于 tableB.1 (通常映射为空值)。

stringprep.**map_table_b2**(*code*)
返回 *code* 依据 tableB.2 (配合 NFKC 使用的大小写转换映射) 所映射的值。

stringprep.**map_table_b3**(*code*)
返回 *code* 依据 tableB.3 (不附带正规化的大小写折叠映射) 所映射的值。

stringprep.**in_table_c11**(*code*)
确定 *code* 是否属于 tableC.1.1 (ASCII 空白字符)。

stringprep.**in_table_c12**(*code*)
确定 *code* 是否属于 tableC.1.2 (非 ASCII 空白字符)。

stringprep.**in_table_c11_c12**(*code*)
确定 *code* 是否属于 tableC.1 (空白字符，C.1.1 和 C.1.2 的并集)。

stringprep.**in_table_c21**(*code*)
确定 *code* 是否属于 tableC.2.1 (ASCII 控制字符)。

stringprep.**in_table_c22**(*code*)
确定 *code* 是否属于 tableC.2.2 (非 ASCII 控制字符)。

stringprep.**in_table_c21_c22**(*code*)
确定 *code* 是否属于 tableC.2 (控制字符，C.2.1 和 C.2.2 的并集)。

stringprep.**in_table_c3**(*code*)
确定 *code* 是否属于 tableC.3 (私有使用)。

stringprep.**in_table_c4**(*code*)
确定 *code* 是否属于 tableC.4 (非字符码位)。

stringprep.**in_table_c5**(*code*)
确定 *code* 是否属于 tableC.5 (替代码)。

stringprep.**in_table_c6**(*code*)
确定 *code* 是否属于 tableC.6 (不适用于纯文本)。

stringprep.**in_table_c7**(*code*)
确定 *code* 是否属于 tableC.7 (不适用于规范表示)。

stringprep.**in_table_c8**(*code*)
确定 *code* 是否属于 tableC.8 (改变显示属性或已弃用)。

stringprep.**in_table_c9**(*code*)
确定 *code* 是否属于 tableC.9 (标记字符)。

stringprep.**in_table_d1**(*code*)
确定 *code* 是否属于 tableD.1 (带有双向属性”R” 或”AL” 的字符)。

stringprep.**in_table_d2**(*code*)
确定 *code* 是否属于 tableD.2 (带有双向属性”L” 的字符)。

6.7 readline --- GNU readline 接口

readline 模块定义了许多方便从 Python 解释器完成和读取/写入历史文件的函数。此模块可以直接使用，或通过支持在交互提示符下完成 Python 标识符的*rlcompleter* 模块使用。使用此模块进行的设置会同时影响解释器的交互提示符以及内置*input()* 函数提供的提示符。

Readline 的按键绑定可以通过一个初始化文件来配置，通常是你的用户目录中的 `.inputrc`。请参阅 GNU Readline 手册中的 Readline 初始化文件 来了解有关该文件的格式和允许的结构，以及 Readline 库的一般功能。

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或*WebAssembly 平台* 上不受支持。

备注

下层的 Readline 库 API 可能使用 `editline` (`libedit`) 库而不是 GNU readline 来实现。在 macOS 上*readline* 模块会在运行时检测所使用的是哪个库。

用于 `editline` 的配置文件与 GNU readline 的不同。如果你要在程序中载入配置字符串你可以使用*backend* 来确定正在使用的是哪个库。

如果你是在 macOS 上使用 `editline`/`libedit` readline 模拟，则位于你的主目录中的初始化文件的名称为 `.editrc`。例如，`~/.editrc` 中的以下内容将开启 *vi* 按键绑定和 TAB 补全:

```
python:bind -v
python:bind ^I rl_complete
```

另外请注意不同的库可能使用不同的历史文件格式。当切换下层的库时，现有的历史文件可能会无法使用。

readline.**backend**

被使用的下层 Readline 库的名称，可以是 `"readline"` 或 `"editline"`。

Added in version 3.13.

6.7.1 初始化文件

下列函数与初始化文件和用户配置有关:

readline.**parse_and_bind**(*string*)

执行在 *string* 参数中提供的初始化行。这将调用底层库中的 `rl_parse_and_bind()`。

readline.**read_init_file**([*filename*])

执行 readline 初始化文件。默认的文件名是上一个被使用的文件名。这将调用底层库中的 `rl_read_init_file()`。

6.7.2 行缓冲区

下列函数会在行缓冲区上操作。

readline.**get_line_buffer**()

返回行缓冲区的当前内容 (底层库中的 `rl_line_buffer`)。

readline.**insert_text**(*string*)

将文本插入到行缓冲区的当前游标位置。该函数会调用底层库中的 `rl_insert_text()`，但会忽略其返回值。

readline.**redisplay**()

改变屏幕的显示以反映行缓冲区的当前内容。该函数会调用底层库中的 `rl_redisplay()`。

6.7.3 历史文件

下列函数会在历史文件上操作：

readline.**read_history_file**([*filename*])

载入一个 readline 历史文件，并将其添加到历史列表。默认文件名为 `~/.history`。此函数会调用底层库中的 `read_history()`。

readline.**write_history_file**([*filename*])

将历史列表保存到 readline 历史文件，覆盖任何现有的文件。默认文件名为 `~/.history`。此函数会调用底层库中的 `write_history()`。

readline.**append_history_file**(*nelements*[, *filename*])

将历史列表的最后 *nelements* 项添加到历史文件。默认文件名为 `~/.history`。该文件必须已存在。此函数会调用底层库中的 `append_history()`。此函数仅当 Python 是针对支持该功能的库版本编译时才会存在。

Added in version 3.5.

readline.**get_history_length**()

readline.**set_history_length**(*length*)

设置或返回需要保存到历史文件的条目行数。*write_history_file()* 函数会通过调用底层库中的 `history_truncate_file()` 以使用该值来截取历史文件。负值意味着不限制历史文件的大小。

6.7.4 历史列表

以下函数会在全局历史列表上操作：

readline.**clear_history**()

清空当前历史。此函数会调用底层库中的 `clear_history()`。此 Python 函数仅当 Python 是针对支持该功能的库版本编译时才会存在。

readline.**get_current_history_length**()

返回历史列表的当前项数。（此函数不同于*get_history_length()*，后者是返回将被写入历史文件的最大行数。）

readline.**get_history_item**(*index*)

返回 *index* 号位置上的历史条目的当前内容。条目索引从一开始。此函数会调用底层库中的 `history_get()`。

readline.**remove_history_item**(*pos*)

从历史列表中移除指定位置上的历史条目。条目位置从零开始。此函数会调用底层库中的 `remove_history()`。

readline.**replace_history_item**(*pos*, *line*)

将指定位置上的历史条目替换为 *line*。条目位置是从零开始的。此函数会调用底层库中的 `replace_history_entry()`。

readline.**add_history**(*line*)

将 *line* 添加到历史缓冲区，就像它是最近输入的一行。此函数会调用底层库中的 `add_history()`。

readline.**set_auto_history**(*enabled*)

启用或禁用当通过 readline 读取输入时对 `add_history()` 的自动调用。*enabled* 参数应为一个布尔值，当其为真值时启用自动历史，当其为假值时则禁用自动历史。

Added in version 3.6.

自动历史将默认启用，对此设置的改变不会在多个会话中保持。

6.7.5 启动钩子

readline.**set_startup_hook**([*function*])

设置或移除底层库的 rl_startup_hook 回调所唤起的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，则任何已安装的函数将被移除。钩子函数将在 readline 打印第一个提示之前不带参数地被调用。

readline.**set_pre_input_hook**([*function*])

设置或移除底层库的 rl_pre_input_hook 回调所唤起的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，则任何已安装的函数将被移除。钩子函数将在打印第一个提示之后 readline 开始读取输入字符之前不带参数地被调用。此函数仅当 Python 为针对支持此功能的库编译时才会存在。

6.7.6 Completion

以下函数与自定义单词补全函数的实现有关。这通常使用 Tab 键进行操作，能够提示并自动补全正在输入的单词。默认情况下，Readline 设置为由 *rlcompleter* 来补全交互模式解释器的 Python 标识符。如果 *readline* 模块要配合自定义的补全函数来使用，则需要设置不同的单词分隔符。

readline.**set_completer**([*function*])

设置或移除补全函数。如果指定了 *function*，它将被用作新的补全函数；如果省略或为 None，任何已安装的补全函数将被移除。补全函数的调用形式为 function(text, state)，其中 *state* 为 0, 1, 2, ...，直至其返回一个非字符串值。它应当返回下一个以 *text* 开头的候选补全内容。

已安装的补全函数将由传递给底层库中 rl_completion_matches() 的 *entry_func* 回调来唤起。*text* 字符串来自于底层库中 rl_attempted_completion_function 回调的第一个形参。

readline.**get_completer**()

获取补全函数，如果没有设置补全函数则返回 None。

readline.**get_completion_type**()

获取正在尝试的补全的类型。此函数会将底层库中的 rl_completion_type 变量作为一个整数返回。

readline.**get_begidx**()

readline.**get_endidx**()

获取补全域的开始和结束索引。这些索引就是传递给下层库的 rl_attempted_completion_function 回调的 *start* 和 *end* 参数。具体值在同一输入编辑场景中根据下层的 C readline 实现的不同可能会不一样。例如：已知 libedit 的行为就不同于 libreadline。

readline.**set_completer_delims**(*string*)

readline.**get_completer_delims**()

设置或获取补全的单词分隔符。这些分隔符确定了要考虑补全的单词的开始和结束位置（即补全域）。这些函数会访问底层库中的 rl_completer_word_break_characters 变量。

readline.**set_completion_display_matches_hook**([*function*])

设置或移除补全显示函数。如果指定了 *function*，它将被用作新的补全显示函数；如果省略或为 None，任何已安装的补全显示函数将被移除。此函数会设置或清除底层库中的 rl_completion_display_matches_hook 回调。补全显示函数会在每次需要显示匹配项时以 function(substitution, [matches], longest_match_length) 的形式被调用。

6.7.7 示例

以下示例演示了如何使用 *readline* 模块的历史读取或写入函数来自动加载和保存用户主目录下名为 .python_history 的历史文件。以下代码通常应当在交互会话期间从用户的 PYTHONSTARTUP 文件自动执行。

```
import atexit
import os
```

(续下页)

(接上页)

```
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # 默认的历史长度为 -1 (无限), 这可能导致增长失控
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

此代码实际上会在 Python 运行于 交互模式时自动运行 (参见*Readline 配置*)。

以下示例实现了同样的目标，但是通过只添加新历史的方式来支持并发的交互会话。

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

以下示例扩展了*code.InteractiveConsole* 类以支持历史保存/恢复。

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 rlcompleter --- 用于 GNU readline 的补全函数

源代码: Lib/rlcompleter.py

---

rlcompleter 模块定义了一个适合被传给 readline 模块中 set_completer() 的补全函数。

当此模块在具有 readline 模块的 Unix 平台上被导入时，会自动创建一个 Completer 实例并将其 complete() 方法设为 *readline completer*。该方法提供了对有效的 Python 标识符和关键字的补全功能。

示例:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

rlcompleter 模块是为 Python 的交互模式而设计的。除非 Python 是附带 -S 选项运行的，这个模块总是会被自动地导入并配置 (参见 *Readline 配置*)。

在没有 readline 的平台, 此模块定义的 Completer 类仍然可以用于自定义行为.

class rlcompleter.**Completer**

Completer 对象具有以下方法:

complete(*text*, *state*)

返回针对 *text* 的下一个可能的补全项。

当被 readline 模块调用时，此方法将被连续调用并附带 state == 0, 1, 2, ... 直到该方法返回 None。

如果指定的 *text* 不包含句点字符 ('.')，它将根据当前 __main__, builtins 和保留关键字（定义于 keyword 模块）所定义的名称进行补全。

如果为带有点号的名称执行调用，它将尝试尽量求值直到最后一部分为止而产生附带影响（函数不会被求值，但它可以生成对 __getattr__() 的调用），并通过 dir() 函数来匹配剩余部分。在对表达式求值期间引发的任何异常都会被捕获、静默处理并返回 None。

CHAPTER 7

二进制数据服务

本章介绍的模块提供了一些操作二进制数据的基本服务操作。有关二进制数据的其他操作，特别是与文件格式和网络协议有关的操作，将在相关章节中介绍。

下面描述的一些库文本处理服务 也可以使用 ASCII 兼容的二进制格式（例如*re*）或所有二进制数据（例如*difflib*）。

另外，请参阅 Python 的内置二进制数据类型的文档二进制序列类型 --- *bytes, bytearray, memoryview* 。

7.1 `struct` --- 将字节串解读为打包的二进制数据

源代码： Lib/struct.py

此模块可在 Python 值和以 Python *bytes* 对象表示的 C 结构体之间进行转换。通过紧凑格式字符串 描述预期的 Python 值转换目标/来源。此模块的函数和对象可被用于两种相当不同的应用程序，与外部源（文件或网络连接）进行数据交换，或者在 Python 应用和 C 层级之间进行数据传输。

备注

当未给出前缀字符时，将默认为原生模式。它会基于构建 Python 解释器的平台和编译器来打包和解包数据。打包一个给定 C 结构体的结果包括为所涉及的 C 类型保持正确对齐的填充字节；类似地，当解包时也会将对齐纳入考虑。相反地，当在外部源之间进行数据通信时，将由程序员负责定义字节顺序和元素之间的填充。请参阅字节顺序，大小和对齐方式 了解详情。

某些*struct* 的函数（以及*Struct* 的方法）接受一个 *buffer* 参数。这将指向实现了 bufferobjects 并提供只读或是可读写缓冲的对象。用于此目的的最常见类型为*bytes* 和*bytearray*，但许多其他可被视为字节数组的类型也实现了缓冲协议，因此它们无需额外从*bytes* 对象复制即可被读取或填充。

7.1.1 函数和异常

此模块定义了下列异常和函数：

exception `struct.error`

会在多种场合下被引发的异常；其参数为一个描述错误信息的字符串。

struct.**pack**(*format*, *v1*, *v2*, ...)

返回一个 bytes 对象，其中包含根据格式字符串 *format* 打包的值 *v1*, *v2*, ... 参数个数必须与格式字符串所要求的值完全匹配。

struct.**pack_into**(*format*, *buffer*, *offset*, *v1*, *v2*, ...)

根据格式字符串 *format* 打包 *v1*, *v2*, ... 等值并将打包的字节串写入可写缓冲区 *buffer* 从 *offset* 开始的位置。请注意 *offset* 是必需的参数。

struct.**unpack**(*format*, *buffer*)

根据格式字符串 *format* 从缓冲区 *buffer* 解包（假定是由 `pack(format, ...)` 打包）。结果为一个元组，即使其只包含一个条目。缓冲区的字节大小必须匹配格式所要求的大小，如*calcsize()* 所示。

struct.**unpack_from**(*format*, */*, *buffer*, *offset=0*)

对 *buffer* 从位置 *offset* 开始根据格式字符串 *format* 进行解包。结果为一个元组，即使其中只包含一个条目。缓冲区的字节大小从位置 *offset* 开始必须至少为*calcsize()* 显示的格式所要求的大小。

struct.**iter_unpack**(*format*, *buffer*)

根据格式字符串 *format* 以迭代方式从缓冲区 *buffer* 中解包。此函数返回一个迭代器，它将从缓冲区读取大小相等的块直到其所有内容耗尽为止。缓冲区的字节大小必须是格式所要求的大小的整数倍，如*calcsize()* 所显示的。

每次迭代将产生一个如格式字符串所指定的元组。

Added in version 3.4.

struct.**calcsize**(*format*)

返回与格式字符串 *format* 相对应的结构的大小（亦即 `pack(format, ...)` 所产生的字节串对象的大小）。

7.1.2 格式字符串

格式字符串描述了打包和解包数据时的数据布局。它们是使用格式字符 来构建的，格式字符指明被打包/解包的数据的类型。此外，还有用来控制字节顺序、大小和对齐 的特殊字符。每个格式字符串都是由一个可选的描述数据总体属性的前缀字符和一个或多个描述实际数据值和填充的格式字符组成的。

字节顺序，大小和对齐方式

在默认情况下，C 类型将以所在机器的原生格式和字节顺序来表示，并在必要时通过跳过填充字节来正确地对齐（根据 C 编译器所使用的规则）。选择此行为是为了使已打包结构体的字节与对应的 C 结构体的内存布局完全对应。使用原生字节顺序和填充还是标准格式取决于应用程序本身。

或者，根据下表，格式字符串的第一个字符可用于指示打包数据的字节顺序，大小和对齐方式：

字符	字节顺序	大小	对齐方式
`@`	按原字节	按原字节	按原字节
`=`	按原字节	标准	无
`<`	小端	标准	无
`>`	大端	标准	无
`!`	网络（= 大端）	标准	无

如果第一个字符不是其中之一，则假定为 `'@'` 。

备注

数字 1023 (十六进制的 `0x3ff`) 具有以下字节表示形式：

- 大端序 (>) 的 `03 ff`
- 小端序 (<) 的 `ff 03`

Python 示例：

```
>>> import struct
>>> struct.pack('>h', 1023)
b'\x03\xff'
>>> struct.pack('<h', 1023)
b'\xff\x03'
```

原生字节顺序可能为大端序或小端序，具体取决于主机系统。例如，Intel x86, AMD64 (x86-64) 和 Apple M1 是小端序的；IBM z 和许多旧式架构则是大端序的。请使用*sys.byteorder*来检查你的系统字节顺序。

本机大小和对齐方式是使用 C 编译器的 `sizeof` 表达式来确定的。这总是会与本机字节顺序相绑定。

标准大小仅取决于格式字符；请参阅格式字符 部分中的表格。

请注意 `'@'` 和 `'='` 之间的区别：两个都使用本机字节顺序，但后者的大小和对齐方式是标准化的。

形式 `'!'` 代表网络字节顺序总是使用在 IETF RFC 1700 中所定义的大端序。

没有什么方式能指定非本机字节顺序（强制字节对调）；请正确选择使用 `'<'` 或 `'>'`。

注释：

(1) 填充只会在连续结构成员之间自动添加。填充不会添加到已编码结构的开头和末尾。

(2) 当使用非本机大小和对齐方式即'<', '>', '=', and '!' 时不会添加任何填充。

(3) 要将结构的末尾对齐到符合特定类型的对齐要求，请以该类型代码加重复计数的零作为格式结束。参见例子。

格式字符

格式字符具有以下含义；C 和 Python 值之间的按其指定类型的转换应当是相当明显的。‘标准大小’列是指当使用标准大小时以字节表示的已打包值大小；也就是当格式字符串以 `'<'`, `'>'`, `'!'` 或 `'='` 之一开头的情况。当使用本机大小时，已打包值的大小取决于具体的平台。

格式	C 类型	Python 类型	标准大小	备注
x	填充字节	无		(7)
c	char	长度为 1 的字节串	1	
b	signed char	整数	1	(1), (2)
B	unsigned char	整数	1	(2)
?	_Bool	bool	1	(1)
h	short	整数	2	(2)
H	unsigned short	整数	2	(2)
i	int	整数	4	(2)
I	unsigned int	整数	4	(2)
l	long	整数	4	(2)
L	unsigned long	整数	4	(2)
q	long long	整数	8	(2)
Q	unsigned long long	整数	8	(2)
n	ssize_t	整数		(3)
N	size_t	整数		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	字节串		(9)
p	char[]	字节串		(8)
P	void*	整数		(5)

在 3.3 版本发生变更: 增加了对 `'n'` 和 `'N'` 格式的支持

在 3.6 版本发生变更: 添加了对 'e' 格式的支持。

注释:

(1) '?' 转换码对应于自 C99 开始由 C 标准所定义的 _Bool 类型。在标准模式下，它总是以一个字节来表示。

(2) 当尝试使用任何整数转换码打包一个非整数时，如果该非整数具有 __index__() 方法，则会在打包之前将参数转换为一个整数。

在 3.2 版本发生变更: 增加了用于非整数的 __index__() 方法。

(3) 'n' 和 'N' 转换码仅对本机大小可用（选择为默认或使用 '@' 字节顺序字符）。对于标准大小，你可以使用适合你的应用的任何其他整数格式。

(4) 对于 'f', 'd' 和 'e' 转换码，打包表示形式将使用 IEEE 754 binary32, binary64 或 binary16 格式 (分别对应于 'f', 'd' 或 'e')，无论平台使用何种浮点格式。

(5) 'P' 格式字符仅对本机字节顺序可用（选择为默认或使用 '@' 字节顺序字符）。字节顺序字符 '=' 选择使用基于主机系统的小端或大端排序。struct 模块不会将其解读为本机排序，因此 'P' 格式将不可用。

(6) IEEE 754 binary16 " 半精度" 类型是在 IEEE 754 标准 的 2008 修订版中引入的。它包含一个符号位，5 个指数位和 11 个精度位 (明确存储 10 位)，可以完全精确地表示大致范围在 6.1e-05 和 6.5e+04 之间的数字。此类型并不被 C 编译器广泛支持：在一台典型的机器上，可以使用 unsigned short 进行存储，但不会被用于数学运算。请参阅维基百科页面 half-precision floating-point format 了解详情。

(7) 在打包时，'x' 会插入一个 NUL 字节。

(8) 'p' 格式字符用于编码 "Pascal 字符串"，即存储在由计数指定的 固定长度字节中的可变长度短字符串。所存储的第一个字节为字符串长度或 255 中的较小值。之后是字符串对应的字节。如果传入*pack()* 的字符串过长（超过计数值减 1），则只有字符串前 count-1 个字节会被存储。如果字符串短于 count-1，则会填充空字节以使得恰好使用了 count 个字节。请注意对于*unpack()*，'p' 格式字符会消耗 count 个字节，但返回的字符串永远不会包含超过 255 个字节。

(9) 对于 's' 格式字符，计数会被解读为字节的长度，而不是像其他格式字符那样的重复计数；例如，'10s' 表示一个与特定的 Python 字节串互相映射的长度为 10 的字节数据，而 '10c' 则表示个 10 个与十个不同的 Python 字节对象互相映射的独立的一字节字符元素 (如 ccccccccc)。(其中的差别的具体演示请参见例子。) 如果未给出计数，则默认值为 1。对于打包操作，字节串会被适当地截断或填充空字节以符合尺寸要求。对于解包操作，结果字节对象总是会恰好具有指定数量的字节。作为特例，'0s' 表示单个空字节串 (而 '0c' 表示 0 个字符)。

格式字符之前可以带有整数重复计数。例如，格式字符串 '4h' 的含义与 'hhhh' 完全相同。

格式之间的空白字符会被忽略；但是计数及其格式字符中不可有空白字符。

当使用某一种整数格式 ('b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q') 打包值 x 时，如果 x 在该格式的有效范围之外则将引发*struct.error*。

在 3.1 版本发生变更: 在之前版本中，某些整数格式包装了超范围的值并会引发*DeprecationWarning* 而不是*struct.error*。

对于 '?' 格式字符，返回值为*True* 或*False*。在打包时将会使用参数对象的逻辑值。以本机或标准 bool 类型表示的 0 或 1 将被打包，任何非零值在解包时将为 True。

例子

> **备注**
>
> 原生字节顺序的示例 (由 '@' 格式前缀或不带任何前缀字符的形式指定) 可能与读者机器所产生的内容不匹配，因为这取决于具体的平台和编译器。

打包和解包三种不同大小的整数，使用大端序:

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

尝试打包一个对于所定义字段来说过大的整数:

```
>>> pack(">h", 99999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
struct.error: 'h' format requires -32768 <= number <= 32767
```

显示 `'s'` and `'c'` 格式字符之间的差异:

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
>>> pack("@3s", b'123')
b'123'
```

解包的字段可通过将它们赋值给变量或将结果包装为一个具名元组来命名:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

格式字符的顺序可能会因为填充是隐式的而对在原生模式中的大小产生影响。在标准模式下，用户要负责插入任何必要的填充。请注意下面的第一个 `pack` 调用中在已打包的 `'#'` 之后添加了三个 NUL 字节以便在四字节边界上对齐到下面的整数。在这个例子中，输出是在一台小端序的机器上产生的:

```
>>> pack('@ci', b'#', 0x12131415)
b'#\x00\x00\x00\x15\x14\x13\x12'
>>> pack('@ic', 0x12131415, b'#')
b'\x15\x14\x13\x12#'
>>> calcsize('@ci')
8
>>> calcsize('@ic')
5
```

以下格式 `'llh0l'` 将会在末尾添加两个填充字节，假定平台的 long 类型按 4 个字节的边界对齐的话:

```
>>> pack('@llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

参见

模块*array*
被打包为二进制存储的同质数据。

模块*json*
JSON 编码器和解码器。

模块*pickle*
Python 对象序列化。

7.1.3 应用

struct 模块存在两个主要应用，即在一个应用程序或使用相同编译器编译的另一个应用程序中 Python 和 C 代码之间的数据交换 (*原生格式*)，以及使用商定的数据布局的应用程序之间的数据交换 (*标准格式*)。一般来说，针对这两个领域构造的格式字符串是不一样的。

原生格式

当构造模仿原生布局的格式字符串时，编译器和机器架构会决定字节顺序和填充。在这种情况下，应当使用 `@` 格式字符来指明原生字节顺序和数据大小。内部填充字节通常是自动插入的。为了正确对齐连续的数据块可能会在格式字符串末尾需要一个零重复的格式代码以舍入到正确的字节边界。

请看这两个简单的示例（在 64 位的小端序机器上）：

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

在不使用额外填充的情况下不会将数据填充到第二个格式字符串末尾的 8 字节边界上。零重复的格式代码解决了这个问题:

```
>>> calcsize('@llh0l')
24
```

`'x'` 格式代码可被用来指定重复，但对于原生格式来说最好是使用 `'0l'` 这样的零重复格式。

在默认情况下，将使用原生字节顺序和对齐，但最好是显式指定并使用 `'@'` 前缀字符。

标准格式

当与你的进程之外如网络或存储交换数据时，请务必保持精确。准确地指定字节顺序、大小和对齐。不要假定它们与特定机器的原生顺序相匹配。例如，网络字节顺序是大端序的，而许多流行的 CPU 则是小端序的。通过显式定义，用户将无需关心他们的代码运行所在平台的具体规格。第一个字符通常应为 `<` 或 `>` (或者 `!`)。程序员要负责填充操作。零重复格式字符是无效的。相反，用户必须在需要时显式地添加 `'x'` 填充字节。回顾上一节中的示例，我们得到:

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

上述结果（在 64 位机器上执行）不保证在不同的机器上执行时仍能匹配。例如，以下示例是在 32 位机器上执行的:

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

7.1.4 类

struct 模块还定义了以下类型：

class struct.**Struct**(*format*)

返回一个新的 Struct 对象，它会根据格式字符串 *format* 来写入和读取二进制数据。一次性地创建 Struct 对象并调用其方法相比调用相同格式的模块层级函数效率更高因为格式字符串只会被编译一次。

> **备注**
>
> 传递线路模块层级函数的已编译版最新格式字符串会被缓存，因此只使用少量格式字符串的程序无需担心重用单独的*Struct* 实例。

已编译的 Struct 对象支持以下方法和属性：

pack(*v1*, *v2*, ...)

等价于*pack()* 函数，使用了已编译的格式。(len(result) 将等于*size*。)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

等价于*pack_into()* 函数，使用了已编译的格式。

unpack(*buffer*)

等价于*unpack()* 函数，使用了已编译的格式。缓冲区的字节大小必须等于*size*。

unpack_from(*buffer*, *offset=0*)

等价于*unpack_from()* 函数，使用了已编译的格式。缓冲区的字节大小从位置 *offset* 开始必须至少为*size*。

iter_unpack(*buffer*)

等价于*iter_unpack()* 函数，使用了已编译的格式。缓冲区的大小必须为*size* 的整数倍。

Added in version 3.4.

format

用于构造此 Struct 对象的格式字符串。

在 3.7 版本发生变更: 格式字符串类型现在是*str* 而不再是*bytes*。

size

计算出对应于*format* 的结构大小（亦即*pack()* 方法所产生的字节串对象的大小）。

在 3.13 版本发生变更: 结构体的 *repr()* 已被修改。现在将为：

```
>>> Struct('i')
Struct('i')
```

7.2 codecs --- 编解码器注册和相关基类

源代码： Lib/codecs.py

这个模块定义了标准 Python 编解码器（编码器和解码器）的基类并提供对内部 Python 编解码器注册表的访问，该注册表负责管理编解码器和错误处理的查找过程。大多数标准编解码器都属于文本编码格式，它们可将文本编码为字节串（以及将字节串解码为文本），但也提供了一些将文本编码为文本，以及将字节串编码为字节串的编解码器。自定义编解码器可以在任意类型间进行编码和解码，但某些模块特性被限制为仅适用于文本编码格式 或将数据编码为*bytes* 的编解码器。

该模块定义了以下用于使用任何编解码器进行编码和解码的函数:

codecs.**encode**(*obj*, *encoding='utf-8'*, *errors='strict'*)

使用为 *encoding* 注册的编解码器对 *obj* 进行编码。

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 `'strict'` 表示编码错误将引发 *ValueError* (或更特定编解码器相关的子类，例如 *UnicodeEncodeError*)。请参阅编解码器基类了解有关编解码器错误处理的更多信息。

codecs.**decode**(*obj*, *encoding='utf-8'*, *errors='strict'*)

使用为 *encoding* 注册的编解码器对 *obj* 进行解码。

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 `'strict'` 表示编码错误将引发 *ValueError* (或更特定编解码器相关的子类，例如 *UnicodeDecodeError*)。请参阅编解码器基类了解有关编解码器错误处理的更多信息。

每种编解码器的完整细节也可以直接查找获取：

codecs.**lookup**(*encoding*)

在 Python 编解码器注册表中查找编解码器信息，并返回一个 *CodecInfo* 对象，其定义见下文。

首先将会在注册表缓存中查找编码，如果未找到，则会扫描注册的搜索函数列表。如果没有找到 *CodecInfo* 对象，则将引发 *LookupError*。否则，*CodecInfo* 对象将被存入缓存并返回给调用者。

class codecs.**CodecInfo**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)

查找编解码器注册表所得到的编解码器细节信息。构造器参数将保存为同名的属性：

name

编码名称

encode

decode

无状态的编码和解码函数。它们必须是具有与 Codec 的 *encode()* 和 *decode()* 方法相同接口的函数或方法 (参见 *Codec 接口*)。这些函数或方法应当工作于无状态的模式。

incrementalencoder

incrementaldecoder

增量式的编码器和解码器类或工厂函数。这些函数必须分别提供由基类 *IncrementalEncoder* 和 *IncrementalDecoder* 所定义的接口。增量式编解码器可以保持状态。

streamwriter

streamreader

流式写入器和读取器类或工厂函数。这些函数必须分别提供由基类 *StreamWriter* 和 *StreamReader* 所定义的接口。流式编解码器可以保持状态。

为了简化对各种编解码器组件的访问，本模块提供了以下附加函数，它们使用 *lookup()* 来执行编解码器查找：

codecs.**getencoder**(*encoding*)

查找给定编码的编解码器并返回其编码器函数。

在编码无法找到时将引发 *LookupError*。

codecs.**getdecoder**(*encoding*)

查找给定编码的编解码器并返回其解码器函数。

在编码无法找到时将引发 *LookupError*。

codecs.**getincrementalencoder**(*encoding*)

查找给定编码的编解码器并返回其增量式编码器类或工厂函数。

在编码无法找到或编解码器不支持增量式编码器时将引发 *LookupError*。

codecs.**getincrementaldecoder**(*encoding*)

查找给定编码的编解码器并返回其增量式解码器类或工厂函数。

在编码无法找到或编解码器不支持增量式解码器时将引发*LookupError*。

codecs.**getreader**(*encoding*)

查找给定编码的编解码器并返回其*StreamReader*类或工厂函数。

在编码无法找到时将引发*LookupError*。

codecs.**getwriter**(*encoding*)

查找给定编码的编解码器并返回其*StreamWriter*类或工厂函数。

在编码无法找到时将引发*LookupError*。

自定义编解码器的启用是通过注册适当的编解码器搜索函数：

codecs.**register**(*search_function*)

注册一个编解码器搜索函数。搜索函数预期接收一个参数，即全部以小写字母表示的编码格式名称，其中中连字符和空格会被转换为下划线，并返回一个*CodecInfo*对象。在搜索函数无法找到给定编码格式的情况下，它应当返回 None。

在 3.9 版本发生变更: 连字符和空格会被转换为下划线。

codecs.**unregister**(*search_function*)

注销一个编解码器搜索函数并清空注册表缓存。如果指定搜索函数未被注册，则不做任何操作。

Added in version 3.10.

虽然内置的*open()*和相关联的*io*模块是操作已编码文本文件的推荐方式，但本模块也提供了额外的工具函数和类，允许在操作二进制文件时使用更多种类的编解码器：

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)

使用给定的 *mode* 打开已编码的文件并返回一个*StreamReaderWriter*的实例，提供透明的编码/解码。默认的文件模式为 'r'，表示以读取模式打开文件。

> **备注**
>
> 如果 *encoding* 不为 None，则下层的已编码文件总是以二进制模式打开。在读取和写入时不会自动执行 '\n' 的转换。*mode* 参数可以是内置*open()*函数所接受的任意二进制模式；'b' 会被自动添加。

encoding 指定文件所要使用的编码格式。允许任何编码为字节串或从字节串解码的编码格式，而文件方法所支持的数据类型则取决于所使用的编解码器。

可以指定 *errors* 来定义错误处理方案。默认值 'strict' 表示在出现编码错误时引发*ValueError*。

buffering 的含义与内置*open()*函数中的相同。默认值 -1 表示将使用默认的缓冲区大小。

在 3.11 版本发生变更: 'U' 模式已被移除。

codecs.**EncodedFile**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

返回一个*StreamRecoder*实例，它提供了 *file* 的透明转码包装版本。当包装版本被关闭时原始文件也会被关闭。

写入已包装文件的数据会根据给定的 *data_encoding* 解码，然后以使用 *file_encoding* 的字节形式写入原始文件。从原始文件读取的字节串将根据 *file_encoding* 解码，其结果将使用 *data_encoding* 进行编码。

如果 *file_encoding* 未给定，则默认为 *data_encoding*。

可以指定 *errors* 来定义错误处理方案。默认值 'strict' 表示在出现编码错误时引发*ValueError*。

codecs.**iterencode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

使用增量式编码器通过迭代来编码由 *iterator* 所提供的输入。此函数属于*generator*。*errors* 参数（以及任何其他关键字参数）会被传递给增量式编码器。

此函数要求编解码器接受*str* 对象形式的文本进行编码。因此它不支持字节到字节的编码器，例如 `base64_codec`。

codecs.**iterdecode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

使用增量式解码器通过迭代来解码由 *iterator* 所提供的输入。此函数属于*generator*。*errors* 参数（以及任何其他关键字参数）会被传递给增量式解码器。

此函数要求编解码器接受*bytes* 对象进行解码。因此它不支持文本到文本的编码器，例如 `rot_13`，但是 `rot_13` 可以通过同样效果的*iterencode()* 来使用。

本模块还提供了以下常量，适用于读取和写入依赖于平台的文件：

codecs.**BOM**

codecs.**BOM_BE**

codecs.**BOM_LE**

codecs.**BOM_UTF8**

codecs.**BOM_UTF16**

codecs.**BOM_UTF16_BE**

codecs.**BOM_UTF16_LE**

codecs.**BOM_UTF32**

codecs.**BOM_UTF32_BE**

codecs.**BOM_UTF32_LE**

这些常量定义了多种字节序列，即一些编码格式的 Unicode 字节顺序标记（BOM）。它们在 UTF-16 和 UTF-32 数据流中被用以指明所使用的字节顺序，并在 UTF-8 中被用作 Unicode 签名。*BOM_UTF16* 是*BOM_UTF16_BE* 或*BOM_UTF16_LE*，具体取决于平台的本机字节顺序，*BOM* 是*BOM_UTF16* 的别名，*BOM_LE* 是*BOM_UTF16_LE* 的别名，*BOM_BE* 是*BOM_UTF16_BE* 的别名。其他序列则表示 UTF-8 和 UTF-32 编码格式中的 BOM。

7.2.1 编解码器基类

codecs 模块定义了一系列基类用来定义配合编解码器对象进行工作的接口，并且也可用作定制编解码器实现的基础。

每种编解码器必须定义四个接口以便用作 Python 中的编解码器：无状态编码器、无状态解码器、流读取器和流写入器。流读取器和写入器通常会重用无状态编码器/解码器来实现文件协议。编解码器作者还需要定义编解码器将如何处理编码和解码错误。

错误处理方案

为了简化和标准化错误处理，编解码器可以通过接受 *errors* 字符串参数来实现不同的错误处理方案：

```
>>> 'German ß, ♬'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ♬'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

以下错误处理器可以用于所有的 Python 标准编码 编解码器：

值	含意
'strict'	引发*UnicodeError* (或其子类)，这是默认的方案。在*strict_errors()* 中实现。
'ignore'	忽略错误格式的数据并且不加进一步通知就继续执行。在*ignore_errors()* 中实现。
'replace'	用一个替代标记来替换。在编码时，使用 ? (ASCII 字符)。在解码时，使用 (U+FFFD，官方的 REPLACEMENT CHARACTER)。在*replace_errors()* 中实现。
'backslashreplace'	用反斜杠转义序列来替换。在编码时，使用格式为 \x*hh* \u*xxxx* \U*xxxxxxxx* 的 Unicode 码位十六进制表示形式。在解码时，使用格式为 \x*hh* 的字节值十六进制表示形式。在*backslashreplace_errors()* 中实现。
'surrogateescape'	在解码时，将字节替换为 U+DC80 至 U+DCFF 范围内的单个代理代码。当在编码数据时使用 'surrogateescape' 错误处理方案时，此代理将被转换回相同的字节。(请参阅 **PEP 383** 了解详情。)

下列错误处理器仅在编码时适用（在文本编码格式 类别以内）：

值	含意
'xmlcharrefre	用 XML/HTML 数字字符引用来替换，即格式为 &#*num*; 的 Unicode 码位十进制表示形式。在*xmlcharrefreplace_errors()* 中实现。
'namereplace'	用 \N{...} 转义序列来替换，出现在花括号中的是来自 Unicode 字符数据库的 Name 属性。在*namereplace_errors()* 中实现。

此外，以下错误处理方案被专门用于指定的编解码器：

值	编解码器	含意
'surrogat	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	允许将代理码位 (U+D800 - U+DFFF) 作为正常码位来编码和解码。否则这些编解码器会将*str* 中出现的代理码位视为错误。

Added in version 3.1: 'surrogateescape' 和 'surrogatepass' 错误处理方案。

在 3.4 版本发生变更: 'surrogatepass' 错误处理器现在可适用于 utf-16* 和 utf-32* 编解码器。

Added in version 3.5: 'namereplace' 错误处理方案。

在 3.5 版本发生变更: 'backslashreplace' 错误处理器现在可适用于解码和转码。

允许的值集合可以通过注册新命名的错误处理方案来扩展：

codecs.**register_error**(*name*, *error_handler*)

在名称 *name* 之下注册错误处理函数 *error_handler*。当 *name* 被指定为错误形参时，*error_handler* 参数所指定的对象将在编码和解码期间发生错误的情况下被调用，

对于编码操作，将会调用 *error_handler* 并传入一个*UnicodeEncodeError* 实例，其中包含有关错误位置的信息。错误处理程序必须引发此异常或别的异常，或者也可以返回一个元组，其中包含输入的不可编码部分的替换对象，以及应当继续进行编码的位置。替换对象可以为*str* 或*bytes* 类型。如果替换对象为字节串，编码器将简单地将其复制到输出缓冲区。如果替换对象为字符串，编码器将对替换对象进行编码。对原始输入的编码操作会在指定位置继续进行。负的位置值将被视为相对于输入字符串的末尾。如果结果位置超出范围则将引发*IndexError*。

解码和转换的做法很相似，不同之处在于将把*UnicodeDecodeError* 或*UnicodeTranslateError* 传给处理程序，并且来自错误处理程序的替换对象将被直接放入输出。

之前注册的错误处理方案（包括标准错误处理方案）可通过名称进行查找：

codecs.**lookup_error**(*name*)

返回之前在名称 *name* 之下注册的错误处理方案。

在处理方案无法找到时将引发*LookupError*。

以下标准错误处理方案也可通过模块层级函数的方式来使用：

codecs.**strict_errors**(*exception*)

实现了 'strict' 错误处理。

每个编码或解码错误都将引发*UnicodeError*。

codecs.**ignore_errors**(*exception*)

实现了 'ignore' 错误处理。

错误格式的数据会被忽略；编码或解码将继续执行而不再通知。

codecs.**replace_errors**(*exception*)

实现了 'replace' 错误处理。

替换 ? (ASCII 字符) 表示编码错误或者 (U+FFFD，官方的 REPLACEMENT CHARACTER) 表示解码错误。

codecs.**backslashreplace_errors**(*exception*)

实现了 'backslashreplace' 错误处理。

错误格式的数据会用反斜杠转义序列来替换。在编码时，使用格式为 \x*hh* \u*xxxx* \U*xxxxxxxx* 的 Unicode 码位十六进制表示形式。在解码时，使用格式为 \x*hh* 的字节值十六进制表示形式。

在 3.5 版本发生变更: 适用于解码和转码。

codecs.**xmlcharrefreplace_errors**(*exception*)

实现 'xmlcharrefreplace' 错误处理（仅限*text encoding* 范围内的编码操作）。

不可编码的字符会被替换为适当的 XML/HTML 数值字符引用，即格式为 &#*num*; 的十进制形式 Unicode 码位。

codecs.**namereplace_errors**(*exception*)

实现 'namereplace' 错误处理（仅限*text encoding* 范围内的编码操作）。

不可编码的字符会被替换为 \N{...} 转义序列。出现在花括号内的字符集合是来自于 Unicode 字符数据库的 Name 属性。例如，德语小写字母 'ß' 将被转换为字符序列 \N{LATIN SMALL LETTER SHARP S}。

Added in version 3.5.

无状态的编码和解码

基本*Codec* 类定义了这些方法，同时还定义了无状态编码器和解码器的函数接口：

class codecs.**Codec**

encode(*input*, *errors='strict'*)

编码 *input* 对象并返回一个元组 (输出对象, 消耗长度)。例如，*text encoding* 会使用特定的字符集编码格式 (例如 cp1252 或 iso-8859-1) 将字符串转换为字节串对象。

errors 参数定义了要应用的错误处理方案。默认为 'strict' 处理方案。

此方法不一定会在*Codec* 实例中保存状态。可使用必须保存状态的*StreamWriter* 作为编解码器以便高效地进行编码。

编码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

decode(*input*, *errors='strict'*)

解码 *input* 对象并返回一个元组 (输出对象, 消耗长度)。例如，*text encoding* 的解码操作会使用特定的字符集编码格式将字节串对象转换为字符串对象。

对于文本编码格式和字节到字节编解码器，*input* 必须为一个字节串对象或提供了只读缓冲区接口的对象 -- 例如，缓冲区对象和映射到内存的文件。

errors 参数定义了要应用的错误处理方案。默认为 `'strict'` 处理方案。

此方法不一定会在`Codec`实例中保存状态。可使用必须保存状态的`StreamReader`作为编解码器以便高效地进行解码。

解码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

增量式的编码和解码

`IncrementalEncoder`和`IncrementalDecoder`类提供了增量式编码和解码的基本接口。对输入的编码/解码不是通过对无状态编码器/解码器的一次调用，而是通过对增量式编码器/解码器的`encode()`/`decode()`方法的多次调用。增量式编码器/解码器会在方法调用期间跟踪编码/解码过程。

调用`encode()`/`decode()`方法后的全部输出相当于将所有通过无状态编码器/解码器进行编码/解码的单个输入连接在一起所得到的输出。

IncrementalEncoder 对象

`IncrementalEncoder`类用来对一个输入进行分步编码。它定义了以下方法，每个增量式编码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalEncoder**(*errors='strict'*)

`IncrementalEncoder`实例的构造器。

所有增量式编码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

`IncrementalEncoder`可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。可用的值请参阅错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在`IncrementalEncoder`对象的生命期内在不同的错误处理策略之间进行切换。

encode(*object*, *final=False*)

编码 *object* (会将编码器的当前状态纳入考虑) 并返回已编码的结果对象。如果这是对`encode()`的最终调用则 *final* 必须为真值（默认为假值）。

reset()

将编码器重置为初始状态。输出将被丢弃：调用 `.encode(object, final=True)`，在必要时传入一个空字节串或字符串，重置编码器并得到输出。

getstate()

返回编码器的当前状态，该值必须为一个整数。实现应当确保 `0` 是最常见的状态。(比整数更复杂的状态表示可以通过编组/选择状态并将结果字符串的字节数据编码为整数来转换为一个整数值)。

setstate(*state*)

将编码器的状态设为 *state*。*state* 必须为`getstate()`所返回的一个编码器状态。

IncrementalDecoder 对象

`IncrementalDecoder`类用来对一个输入进行分步解码。它定义了以下方法，每个增量式解码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalDecoder**(*errors='strict'*)

IncrementalDecoder 实例的构造器。

所有增量式解码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

IncrementalDecoder 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。可用的值请参阅*错误处理方案*。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*IncrementalDecoder* 对象的生命期内在不同的错误处理策略之间进行切换。

decode(*object*, *final=False*)

解码 *object* (会将解码器的当前状态纳入考虑) 并返回已解码的结果对象。如果这是对*decode()* 的最终调用则 *final* 必须为真值（默认为假值）。如果 *final* 为真值则解码器必须对输入进行完全解码并且必须刷新所有缓冲区。如果这无法做到（例如由于在输入结束时字节串序列不完整）则它必须像在无状态的情况下那样初始化错误处理（这可能引发一个异常）。

reset()

将解码器重置为初始状态。

getstate()

返回解码器的当前状态。这必须为一个二元组，第一项必须是包含尚未解码的输入的缓冲区。第二项必须为一个整数，可以表示附加状态信息。(实现应当确保 `0` 是最常见的附加状态信息。) 如果此附加状态信息为 `0` 则必须可以将解码器设为没有已缓冲输入并且以 `0` 作为附加状态信息，以便将先前已缓冲的输入馈送到解码器使其返回到先前的状态而不产生任何输出。(比整数更复杂的附加状态信息可以通过编组/选择状态信息并将结果字符串的字节数据编码为整数来转换为一个整数值。)

setstate(*state*)

将解码器的状态设为 *state*。*state* 必须为*getstate()* 所返回的一个解码器状态。

流式的编码和解码

The *StreamWriter* 和*StreamReader* 类提供了一些泛用工作接口，可被用来非常方便地实现新的编码格式子模块。请参阅 `encodings.utf_8` 中的示例了解如何做到这一点。

StreamWriter 对象

StreamWriter 类是*Codec* 的子类，它定义了以下方法，每个流式写入器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

StreamWriter 实例的构造器。

所有流式写入器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream 参数必须为一个基于特定编解码器打开用于写入文本或二进制数据的文件型对象。

StreamWriter 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。请参阅*错误处理方案* 了解下层的流式编解码器可支持的标准错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamWriter* 对象的生命期内在不同的错误处理策略之间进行切换。

write(*object*)

将编码后的对象内容写入到流。

writelines(*list*)

将拼接后的字符串可迭代对象写入到流（可能通过重用*write()* 方法）。无限长或非常大的可迭代对象不受支持。标准的字节到字节编解码器不支持此方法。

reset()

重置用于保持内部状态的编解码器缓冲区。

调用此方法应当确保在干净的状态下放入输出数据，以允许直接添加新的干净数据而无须重新扫描整个流来恢复状态。

除了上述的方法，*StreamWriter* 还必须继承来自下层流的所有其他方法和属性。

StreamReader 对象

StreamReader 类是*Codec* 的子类，它定义了以下方法，每个流式读取器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamReader**(*stream*, *errors='strict'*)

StreamReader 实例的构造器。

所有流式读取器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream 参数必须为一个基于特定编解码器打开用于读取文本或二进制数据的文件型对象。

StreamReader 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。请参阅错误处理方案了解下层的流式编解码器可支持的标准错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamReader* 对象的生命期内在不同的错误处理策略之间进行切换。

errors 参数所允许的值集合可以使用*register_error()* 来扩展。

read(*size=-1*, *chars=-1*, *firstline=False*)

解码来自流的数据并返回结果对象。

chars 参数指明要返回的解码后码位或字节数量。*read()* 方法绝不会返回超出请求数量的数据，但如果可用数量不足，它可能返回少于请求数量的数据。

size 参数指明要读取并解码的已编码字节或码位的最大数量近似值。解码器可以适当地修改此设置。默认值 -1 表示尽可能多地读取并解码。此形参的目的是防止一次性解码过于巨大的文件。

firstline 旗标指明如果在后续行发生解码错误，则仅返回第一行就足够了。

此方法应当使用“贪婪”读取策略，这意味着它应当在编码格式定义和给定大小所允许的情况下尽可能多地读取数据，例如，如果在流上存在可选的编码结束或状态标记，这些内容也应当被读取。

readline(*size=None*, *keepends=True*)

从输入流读取一行并返回解码后的数据。

如果给定了 *size*，则将其作为 size 参数传递给流的*read()* 方法。

如果 *keepends* 为假值，则行结束符将从返回的行中去除。

readlines(*sizehint=None*, *keepends=True*)

从输入流读取所有行并将其作为一个行列表返回。

行结束符会使用编解码器的*decode()* 方法来实现，并且如果 *keepends* 为真值则会将其包含在列表条目中。

如果给定了 *sizehint*，则将其作为 *size* 参数传递给流的*read()* 方法。

reset()

重置用于保持内部状态的编解码器缓冲区。

请注意不应当对流进行重定位。使用此方法的主要目的是为了能够从解码错误中恢复。

除了上述的方法，*StreamReader* 还必须继承来自下层流的所有其他方法和属性。

StreamReaderWriter 对象

StreamReaderWriter 是一个方便的类，允许对同时工作于读取和写入模式的流进行包装。

其设计使得开发者可以使用*lookup()* 函数所返回的工厂函数来构造实例。

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)

创建一个*StreamReaderWriter* 实例。*stream* 必须为一个文件型对象。*Reader* 和 *Writer* 必须为分别提供了*StreamReader* 和*StreamWriter* 接口的工厂函数或类。错误处理通过与流式读取器和写入器所定义的相同方式来完成。

StreamReaderWriter 实例定义了*StreamReader* 和*StreamWriter* 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

StreamRecoder 对象

StreamRecoder 将数据从一种编码格式转换为另一种，这对于处理不同编码环境的情况有时会很有用。

其设计使得开发者可以使用*lookup()* 函数所返回的工厂函数来构造实例。

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

创建一个实现了双向转换的*StreamRecoder* 实例: *encode* 和 *decode* 工作于前端——调用*read()* 和*write()* 的代码可见的数据，而 *Reader* 和 *Writer* 工作于后端——*stream* 中的数据。

你可以使用这些对象来进行透明转码，例如从 Latin-1 转为 UTF-8 以及反向转换。

stream 参数必须为一个文件型对象。

encode 和 *decode* 参数必须遵循*Codec* 接口。*Reader* 和 *Writer* 必须为分别提供了*StreamReader* 和*StreamWriter* 接口对象的工厂函数或类。

错误处理通过与流式读取器和写入器所定义的相同方式来完成。

StreamRecoder 实例定义了*StreamReader* 和*StreamWriter* 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

7.2.2 编码格式与 Unicode

字符串在系统内部存储为 U+0000--U+10FFFF 范围内的码位序列。（请参阅 **PEP 393** 了解有关实现的详情。）一旦字符串对象要在 CPU 和内存以外使用，字节的大小端顺序和字节数组的存储方式就成为一个影响因素。如同使用其他编解码器一样，将字符串序列化为字节序列被称为 *编码*，而从字节序列重建字符串被称为 *解码*。存在许多不同的文本序列化编解码器，它们被统称为*文本编码格式*。

最简单的文本编码格式 (称为 'latin-1' 或 'iso-8859-1') 将码位 0--255 映射为字节值 0x0--0xff，这意味着包含 U+00FF 以上码位的字符串对象无法使用此编解码器进行编码。这样做将引发*UnicodeEncodeError*，其形式类似下面这样（不过详细的错误信息可能会有所不同）: UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256)。

还有另外一组编码格式（所谓的字符映射编码）会选择全部 Unicode 码位的不同子集并设定如何将这些码位映射为字节值 0x0--0xff。要查看这是如何实现的，只需简单地打开相应源码例如 encodings/cp1252.py (这是一个主要在 Windows 上使用的编码格式)。其中会有一个包含 256 个字符的字符串常量，指明每个字符所映射的字节值。

所有这些编码格式只能对 Unicode 所定义的 1114112 个码位中的 256 个进行编码。一种能够存储每个 Unicode 码位的简单而直接的办法就是将每个友们存储为四个连续的字节。存在两种不同的可能性: 以大端序存储或以小端序存储。这两种编码格式分别被称为 UTF-32-BE 和 UTF-32-LE。他们共有的缺点可以举例说明: 如果你在一台小端序的机器上使用 UTF-32-BE 则你必须在编码和解码时翻转字节。UTF-32 避免了这个问题: 字节的排列将总是使用自然端序。当这些字节被具有不同端序的 CPU 读取时，则必须进行字节翻转。为了能够检测 UTF-16 或 UTF-32 字节序列的大小端序，可以使用所谓的 BOM (" 字节顺序标记")。这对应于 Unicode 字符 U+FEFF。此字符可被添加到每个 UTF-16 或 UTF-32 字节序列的开头。此字符的字节翻转版本 (0xFFFE) 是一个不可出现于 Unicode 文本中的非法字符。因此当发现一个 UTF-16 或 UTF-32 字节序列的首个字符是 U+FFFE 时，就必须在解码时进行字节翻转。不幸的是字符 U+FEFF 还有第二个含义 ZERO WIDTH NO-BREAK SPACE: 即宽度为零并且不允许用来拆分单词的字符。它可以被用

来为语言分析算法提供提示。在 Unicode 4.0 中使用 U+FEFF 表示 ZERO WIDTH NO-BREAK SPACE 已被弃用 (改用 U+2060 (WORD JOINER) 负责此任务)。然而 Unicode 软件仍然必须能够处理 U+FEFF 的两个含义：作为 BOM 它被用来确定已编码字节的存储布局，并在字节序列被解码为字符串后将其去除；作为 ZERO WIDTH NO-BREAK SPACE 它是一个普通字符，将像其他字符一样被解码。

还有另一种编码格式能够对所有 Unicode 字符进行编码：UTF-8。UTF-8 是一种 8 位编码，这意味着在 UTF-8 中没有字节顺序问题。UTF-8 字节序列中的每个字节由两部分组成：标志位（最重要的位）和内容位。标志位是由零至四个值为 1 的二进制位加一个值为 0 的二进制位构成的序列。Unicode 字符会按以下形式进行编码（其中 x 为内容位，当拼接为一体时将给出对应的 Unicode 字符）：

范围	编码
U-00000000 ... U-0000007F	0xxxxxxx
U-00000080 ... U-000007FF	110xxxxx 10xxxxxx
U-00000800 ... U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 ... U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode 字符最不重要的一个位就是最右侧的二进制位 x。

由于 UTF-8 是一种 8 位编码格式，因此 BOM 是不必要的，并且已编码字符串中的任何 U+FEFF 字符（即使是作为第一个字符）都会被视为是 ZERO WIDTH NO-BREAK SPACE。

在没有外部信息的情况下将不可能毫无疑义地确定一个字符串使用了何种编码格式。每种字符映射编码格式都可以解码任意的随机字节序列。然而对 UTF-8 来说这却是不可能的，因为 UTF-8 字节序列具有不允许任意字节序列的特别结构。为了提升 UTF-8 编码格式检测的可靠性，Microsoft 发明了一种 UTF-8 的变体形式 (Python 称之为 "utf-8-sig") 专门用于其 Notepad 程序：在任何 Unicode 字节被写入文件之前，会先写入一个 UTF-8 编码的 BOM (它看起来是这样一个字节序列: 0xef, 0xbb, 0xbf)。由于任何字符映射编码后的文件都不大可能以这些字节值开头 (例如它们会映射为

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

对于 iso-8859-1 编码格式来说)，这提升了根据字节序列来正确猜测 utf-8-sig 编码格式的成功率。所以在这里 BOM 的作用并不是帮助确定生成字节序列所使用的字节顺序，而是作为帮助猜测编码格式的记号。在进行编码时 utf-8-sig 编解码器将把 0xef, 0xbb, 0xbf 作为头三个字节写入文件。在进行解码时 utf-8-sig 将跳过这三个字节，如果它们作为文件的头三个字节出现的话。在 UTF-8 中并不推荐使用 BOM，通常应当避免它们的出现。

7.2.3 标准编码

Python 自带了许多内置的编解码器，它们的实现或者是通过 C 函数，或者是通过映射表。以下表格是按名称排序的编解码器列表，并提供了一些常见别名以及编码格式通常针对的语言。别名和语言列表都不是详尽无遗的。请注意仅有大小写区别或使用连字符替代下划线的拼写形式也都是有效的别名；因此，'utf-8' 是 'utf_8' 编解码器的有效别名。

有些常见编码格式可以绕过编解码器查找机制来提升性能。这些优化机会对于 CPython 来说仅能通过一组有限的别名（大小写不敏感）来识别：utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows 专属), ascii, us-ascii, utf-16, utf16, utf-32, utf32, 也包括使用下划线替代连字符的的形式。使用这些编码格式的其他别名可能会导致更慢的执行速度。

在 3.6 版本发生变更: 可识别针对 us-ascii 的优化机会。

许多字符集都支持相同的语言。它们在个别字符（例如是否支持 EURO SIGN 等）以及给字符所分配的码位方面存在差异。特别是对于欧洲语言来说，通常存在以下几种变体：

- 某个 ISO 8859 编码集
- 某个 Microsoft Windows 编码页，通常是派生自某个 8859 编码集，但会用附加的图形字符来替换控制字符。
- 某个 IBM EBCDIC 编码页

- 某个 IBM PC 编码页，通常会兼容 ASCII

编码	别名	语言
ascii	646, us-ascii	英语
big5	big5-tw, csbig5	繁体中文
big5hkscs	big5-hkscs, hkscs	繁体中文
cp037	IBM037, IBM039	英语
cp273	273, IBM273, csIBM273	德语 Added in version 3.4.
cp424	EBCDIC-CP-HE, IBM424	希伯来语
cp437	437, IBM437	英语
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西欧
cp720		阿拉伯语
cp737		希腊语
cp775	IBM775	波罗的海语言
cp850	850, IBM850	西欧
cp852	852, IBM852	中欧和东欧
cp855	855, IBM855	白俄罗斯语，保加利亚语，马其顿语，俄语，塞尔维亚语
cp856		希伯来语
cp857	857, IBM857	土耳其语
cp858	858, IBM858	西欧
cp860	860, IBM860	葡萄牙语
cp861	861, CP-IS, IBM861	冰岛语
cp862	862, IBM862	希伯来语
cp863	863, IBM863	加拿大语
cp864	IBM864	阿拉伯语
cp865	865, IBM865	丹麦语/挪威语
cp866	866, IBM866	俄语
cp869	869, CP-GR, IBM869	希腊语
cp874		泰语
cp875		希腊语
cp932	932, ms932, mskanji, ms-kanji, windows-31j	日语
cp949	949, ms949, uhc	韩语
cp950	950, ms950	繁体中文
cp1006		乌尔都语
cp1026	ibm1026	土耳其语
cp1125	1125, ibm1125, cp866u, ruscii	乌克兰语 Added in version 3.4.
cp1140	ibm1140	西欧
cp1250	windows-1250	中欧和东欧
cp1251	windows-1251	白俄罗斯语，保加利亚语，马其顿语，俄语，塞尔维亚语
cp1252	windows-1252	西欧
cp1253	windows-1253	希腊语
cp1254	windows-1254	土耳其语
cp1255	windows-1255	希伯来语
cp1256	windows-1256	阿拉伯语
cp1257	windows-1257	波罗的海语言
cp1258	windows-1258	越南语
euc_jp	eucjp, ujis, u-jis	日语
euc_jis_2004	jisx0213, eucjis2004	日语
euc_jisx0213	eucjisx0213	日语

续下页

表 1 – 接上页

编码	别名	语言
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韩语
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	简体中文
gbk	936, cp936, ms936	统一汉语
gb18030	gb18030-2000	统一汉语
hz	hzgb, hz-gb, hz-gb-2312	简体中文
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日语
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日语
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	日语，韩语，简体中文，西欧，希腊语
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日语
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日语
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日语
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韩语
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	西欧
iso8859_2	iso-8859-2, latin2, L2	中欧和东欧
iso8859_3	iso-8859-3, latin3, L3	世界语，马耳他语
iso8859_4	iso-8859-4, latin4, L4	波罗的海语言
iso8859_5	iso-8859-5, cyrillic	白俄罗斯语，保加利亚语，马其顿语，俄语，塞尔维亚语
iso8859_6	iso-8859-6, arabic	阿拉伯语
iso8859_7	iso-8859-7, greek, greek8	希腊语
iso8859_8	iso-8859-8, hebrew	希伯来语
iso8859_9	iso-8859-9, latin5, L5	土耳其语
iso8859_10	iso-8859-10, latin6, L6	北欧语言
iso8859_11	iso-8859-11, thai	泰语
iso8859_13	iso-8859-13, latin7, L7	波罗的海语言
iso8859_14	iso-8859-14, latin8, L8	凯尔特语
iso8859_15	iso-8859-15, latin9, L9	西欧
iso8859_16	iso-8859-16, latin10, L10	东南欧
johab	cp1361, ms1361	韩语
koi8_r		俄语
koi8_t		塔吉克 Added in version 3.5.
koi8_u		乌克兰语
kz1048	kz_1048, strk1048_2002, rk1048	哈萨克语 Added in version 3.5.
mac_cyrillic	maccyrillic	白俄罗斯语，保加利亚语，马其顿语，俄语，塞尔维亚语
mac_greek	macgreek	希腊语
mac_iceland	maciceland	冰岛语
mac_latin2	maclatin2, maccentraleurope, mac_centeuro	中欧和东欧
mac_roman	macroman, macintosh	西欧
mac_turkish	macturkish	土耳其语
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	哈萨克语
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日语

续下页

表 1 – 接上页

编码	别名	语言
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	日语
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日语
utf_32	U32, utf32	所有语言
utf_32_be	UTF-32BE	所有语言
utf_32_le	UTF-32LE	所有语言
utf_16	U16, utf16	所有语言
utf_16_be	UTF-16BE	所有语言
utf_16_le	UTF-16LE	所有语言
utf_7	U7, unicode-1-1-utf-7	所有语言
utf_8	U8, UTF, utf8, cp65001	所有语言
utf_8_sig		所有语言

在 3.4 版本发生变更: utf-16* 和 utf-32* 编码器将不再允许编码代理码位 (`U+D800--U+DFFF`)。utf-32* 解码器将不再解码与代理码位相对应的字节序列。

在 3.8 版本发生变更: `cp65001` 现在是 `utf_8` 的一个别名。

7.2.4 Python 专属的编码格式

有一些预定义编解码器是 Python 专属的，因此它们在 Python 之外没有意义。这些编解码器按其所预期的输入和输出类型在下表中列出（请注意虽然文本编码是编解码器最常见的使用场景，但下层的编解码器架构支持任意数据转换而不仅是文本编码）。对于非对称编解码器，该列描述的含义是编码方向。

文字编码

以下编解码器提供了`str` 到`bytes` 的编码和*bytes-like object* 到`str` 的解码，类似于 Unicode 文本编码。

编码	别名	含意
idna		实现 **RFC 3490**，另请参阅*encodings.idna* 。仅支持 `errors='strict'` 。
mbcs	ansi, dbcs	Windows 专属：根据 ANSI 代码页（CP_ACP）对操作数进行编码。
oem		Windows 专属：根据 OEM 代码页（CP_OEMCP）对操作数进行编码。 Added in version 3.6.
palmos		PalmOS 3.5 的编码格式
punycode		实现 **RFC 3492**。不支持有状态编解码器。
raw_unicode_escape		Latin-1 编码格式附带对其他码位以 `\u`*XXXX* 和 `\U`*XXXXXXXX* 进行编码。现有的反斜杠不会以任何方式转义。它被用于 Python 的 pickle 协议。
undefined		所有转换都将引发异常，甚至对空字符串也不例外。错误处理方案会被忽略。
unicode_escape		适合用于以 ASCII 编码的 Python 源代码中的 Unicode 字面值内容的编码格式，但引号不会被转义。对 Latin-1 源代码进行解码。请注意 Python 源代码实际上默认使用 UTF-8。

在 3.8 版本发生变更: ”unicode_internal” 编解码器已被移除。

二进制转换

以下编解码器提供了二进制转换: *bytes-like object* 到*bytes* 的映射。它们不被*bytes.decode()* 所支持（该方法只生成*str* 类型的输出）。

编码	别名	含意	编码器/解码器
base64_codec[1]	base64, base_64	将操作数转换为多行 MIME base64 (结果总是包含一个末尾的 `'\n'`) 在 3.4 版本发生变更: 接受任意*bytes-like object* 作为输入用于编码和解码	*base64.encodebytes()* / *base64.decodebytes()*
bz2_codec	bz2	使用 bz2 压缩操作数	*bz2.compress()* / *bz2.decompress()*
hex_codec	hex	将操作数转换为十六进制表示，每个字节有两位数	*binascii.b2a_hex()* / *binascii.a2b_hex()*
quopri_codec	quopri, quoted-printable, quoted_printab	将操作数转换为 MIME 带引号的可打印数据	*quopri.encode()* 且 `quotetabs=True` / *quopri.decode()*
uu_codec	uu	使用 uuencode 转换操作数	
zlib_codec	zip, zlib	使用 gzip 压缩操作数	*zlib.compress()* / *zlib.decompress()*

Added in version 3.2: 恢复二进制转换。

在 3.4 版本发生变更: 恢复二进制转换的别名。

文字转换

以下编解码器提供了文本转换: *str* 到 *str* 的映射。它不被 *str.encode()* 所支持（该方法只生成 *bytes* 类型的输出）。

编码	别名	含意
rot_13	rot13	返回操作数的凯撒密码加密结果

Added in version 3.2: 恢复 `rot_13` 文本转换。

在 3.4 版本发生变更: 恢复 `rot13` 别名。

7.2.5 `encodings.idna` --- 应用程序中的国际化域名

此模块实现了 **RFC 3490** (应用程序中的国际化域名) 和 **RFC 3492** (Nameprep: 用于国际化域名 (IDN) 的 Stringprep 配置文件)。它是在 `punycode` 编码格式和 *stringprep* 的基础上构建的。

如果你需要来自 **RFC 5891** 和 **RFC 5895** 的 IDNA 2008 标准，请使用第三方的 idna 模块。

这些 RFC 共同定义了一个在域名中支持非 ASCII 字符的协议。一个包含非 ASCII 字符的域名 (例如 `www.Alliancefrançaise.nu`) 会被转换为兼容 ASCII 的编码格式 (简称 ACE，例如 `www.xn--alliancefranaise-npb.nu`)。随后此域名的 ACE 形式可以用于所有由于特定协议而不允许使用任意字符的场合，例如 DNS 查询，HTTP *Host* 字段等等。此转换是在应用中进行的；如有可能将对用户可见：应用应当透明地将 Unicode 域名标签转换为线上的 IDNA，并在 ACE 标签被呈现给用户之前将其转换回 Unicode。

Python 以多种方式支持这种转换: `idna` 编解码器执行 Unicode 和 ACE 之间的转换，基于在 **section 3.1 of RFC 3490** 中定义的分隔字符将输入字符串拆分为标签，再根据需要将每个标签转换为 ACE，相反地又会基于 `.` 分隔符将输入字节串拆分为标签，再将找到的任何 ACE 标签转换为 Unicode。此外，*socket* 模块可透明地将 Unicode 主机名转换为 ACE，以便应用在将它们传给 socket 模块时无须自行转换主机名。除此之外，许多包含以主机名作为函数参数的模块例如 *http.client* 和 *ftplib* 都接受 Unicode 主机名（并且 *http.client* 也会在 *Host* 字段中透明地发送 IDNA 主机名，如果它需要发送该字段的话）。

当从线路接收主机名时（例如反向名称查找），到 Unicode 的转换不会自动被执行：希望向用户提供此种主机名的应用应当将它们解码为 Unicode。

encodings.idna 模块还实现了 nameprep 过程，该过程会对主机名执行特定的规范化操作，以实现国际域名的大小写不敏感特性与合并相似的字符。如果有需要可以直接使用 nameprep 函数。

`encodings.idna.`**`nameprep`**(*label*)

返回 *label* 经过名称处理操作的版本。该实现目前基于查询字符串，因此 `AllowUnassigned` 为真值。

`encodings.idna.`**`ToASCII`**(*label*)

将标签转换为 ASCII，规则定义见 **RFC 3490**。`UseSTD3ASCIIRules` 预设为假值。

`encodings.idna.`**`ToUnicode`**(*label*)

将标签转换为 Unicode，规则定义见 **RFC 3490**。

7.2.6 `encodings.mbcs` --- Windows ANSI 代码页

此模块实现 ANSI 代码页（CP_ACP）。

Availability: Windows.

在 3.2 版本发生变更: 在 3.2 版之前，*errors* 参数会被忽略；总是会使用 `'replace'` 进行编码，并使用 `'ignore'` 进行解码。

[1] 除了字节类对象，`'base64_codec'` 也接受仅包含 ASCII 的 *str* 实例用于解码

在 3.3 版本发生变更: 支持任何错误处理

7.2.7 `encodings.utf_8_sig` --- 带 BOM 签名的 UTF-8 编解码器

此模块实现了 UTF-8 编解码器的一个变种：在编码时将把 UTF-8 已编码 BOM 添加到 UTF-8 编码字节数据的开头。对于有状态编码器此操作只执行一次（当首次写入字节流时）。在解码时将跳过数据开头作为可选项的 UTF-8 已编码 BOM。

CHAPTER 8

数据类型

本章所描述的模块提供了许多专门的数据类型，如日期和时间、固定类型的数组、堆队列、双端队列、以及枚举。

Python 也提供一些内置数据类型，特别是，*dict*、*list*、*set*、*frozenset*、以及*tuple*。*str* 这个类是用来存储 Unicode 字符串的，而*bytes* 和*bytearray* 这两个类是用来存储二进制数据的。

本章包含以下模块的文档：

8.1 datetime --- 基本日期和时间类型

源代码： Lib/datetime.py

`datetime` 模块提供了用于操作日期和时间的类。

在支持日期时间数学运算的同时，实现的关注点更着重于如何能够更有效地解析其属性用于格式化输出和数据操作。

小技巧

跳到格式代码。

参见

模块*calendar*
通用日历相关函数

模块*time*
时间的访问和转换

***zoneinfo* 模块**
代表 IANA 时区数据库的具体时区。

dateutil 包
具有扩展时区和解析支持的第三方库。

包 DateType
引入了几种独特的静态类型的第三方库，例如允许静态类型检查器 区分简单型和感知型日期时间。

8.1.1 感知型对象和简单型对象

日期和时间对象可以根据它们是否包含时区信息而分为“感知型”和“简单型”两类。

充分掌握应用性算法和政治性时间调整信息例如时区和夏令时的情况下，一个 **感知型**对象就能相对于其他感知型对象来精确定位自身时间点。感知型对象是用来表示一个没有解释空间的固定时间点。[1]

简单型对象没有包含足够多的信息来无歧义地相对于其他 date/time 对象来定位自身时间点。不论一个简单型对象所代表的是世界标准时间（UTC）、当地时间还是某个其他时区的时间完全取决于具体程序，就像一个特定数字所代表的是米、英里还是质量完全取决于具体程序一样。简单型对象更易于理解和使用，代价则是忽略了某些现实性考量。

对于要求感知型对象的应用，*datetime* 和 *time* 对象具有一个可选的时区信息属性 `tzinfo`，它可被设为抽象类 *tzinfo* 的子类的一个实例。这些 *tzinfo* 对象会捕获与 UTC 时间的差值、时区名称以及夏令时是否生效等信息。

`datetime` 模块只提供了一个具体的 *tzinfo* 类，即 *timezone* 类。`timezone` 类可以表示具有相对于 UTC 的固定时差的简单时区，例如 UTC 本身或北美 EST 和 EDT 时区等。支持时区的详细程度取决于具体的应用。世界各地的时间调整规则往往是政治性多于合理性，经常会发生变化，除了 UTC 之外并没有一个能适合所有应用的标准。

8.1.2 常量

`datetime` 模块导出了以下常量：

`datetime.`**`MINYEAR`**
date 或 *datetime* 对象允许的最小年份数值。*MINYEAR* 为 1。

`datetime.`**`MAXYEAR`**
date 或 *datetime* 对象允许的最大年份数值。*MAXYEAR* 为 9999。

`datetime.`**`UTC`**
UTC 时区单例 *datetime.timezone.utc* 的别名。

Added in version 3.11.

8.1.3 有效的类型

class `datetime.`**`date`**
一个理想化的简单型日期，它假设当今的公历在过去和未来永远有效。属性: *year*, *month*, and *day*。

class `datetime.`**`time`**
一个独立于任何特定日期的理想化时间，它假设每一天都恰好等于 24*60*60 秒。（这里没有“闰秒”的概念。）包含属性: *hour*, *minute*, *second*, *microsecond* 和 *tzinfo*。

class `datetime.`**`datetime`**
日期和时间的结合。属性： *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class `datetime.`**`timedelta`**
将两个 *datetime* 或 *date* 实例之间的差值表示为微秒级精度的持续时间。

class `datetime.`**`tzinfo`**
一个描述时区信息对象的抽象基类。用来给 *datetime* 和 *time* 类提供自定义的时间调整概念（例如处理时区和/或夏令时）。

[1] 就是说如果我们忽略相对论效应的话。

class datetime.**timezone**

一个实现了*tzinfo*抽象基类的子类，用于表示相对于世界标准时间（UTC）的偏移量。

Added in version 3.2.

这些类型的对象都是不可变的。

子类关系

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

通用的特征属性

date, *datetime*, *time* 和*timezone* 类型共享这些通用特性:

- 这些类型的对象都是不可变的。
- 这些类型的对象是*hashable*，意味着它们可以被用作字典的键。
- 这些类型的对象支持通过*pickle* 模块进行高效的封存。

确定一个对象是感知型还是简单型

date 类型的对象都是简单型的。

time 或*datetime* 类型的对象可以是感知型或者简单型。

一个*datetime* 对象 `d` 在以下条件同时成立时将是感知型的:

1. `d.tzinfo` 不为 `None`
2. `d.tzinfo.utcoffset(d)` 不返回 `None`

在其他情况下，`d` 将是简单型的。

一个*time* 对象 `t` 在以下条件同时成立时将是感知型的:

1. `t.tzinfo` 不为 `None`
2. `t.tzinfo.utcoffset(None)` 不返回 `None`。

在其他情况下，`t` 将是简单型的。

感知型和简单型之间的区别不适用于*timedelta* 对象。

8.1.4 `timedelta` 类对象

timedelta 对象表示一段持续的时间，即两个*datetime* 或*date* 实例之间的差值。

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

所有参数都是可选的并且默认为 0。这些参数可以是整数或者浮点数，并可以为正值或者负值。

只有 *days*, *seconds* 和 *microseconds* 会存储在内部。参数单位的换算规则如下:

- 1 毫秒会转换成 1000 微秒。
- 1 分钟会转换成 60 秒。
- 1 小时会转换成 3600 秒。
- 1 星期会转换成 7 天。

日期、秒、微秒都是标准化的，所以它们的表达方式也是唯一的，例:

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (一天的秒数)
- `-999999999 <= days <= 999999999`

下面的例子演示了如何对 *days*, *seconds* 和 *microseconds* 以外的任意参数执行“合并”操作并标准化为以上三个结果属性:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # 只保留日期、秒和微秒
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

在有任何参数为浮点型并且 microseconds 值为小数的情况下，从所有参数中余下的微秒数将被合并，并使用四舍五入偶不入奇的规则将总计值舍入到最接近的整数微秒值。如果没有任何参数为浮点型的情况下，则转换和标准化过程将是完全精确的（不会丢失信息）。

如果标准化后的 days 数值超过了指定范围，将会抛出*OverflowError* 异常。

请注意对负数值进行标准化的结果可能会令人感到惊讶。例如:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

类属性:

`timedelta.`**`min`**

The most negative *timedelta* object, `timedelta(-999999999)`.

`timedelta.`**`max`**

The most positive *timedelta* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.`**`resolution`**

两个不相等的*timedelta* 类对象最小的间隔为 `timedelta(microseconds=1)`。

请注意，因为标准化的缘故，`timedelta.max` 大于 `-timedelta.min`。`-timedelta.max` 不可以表示为一个*timedelta* 对象。

实例属性（只读）:

`timedelta.`**`days`**

-999,999,999 至 999,999,999 开区间。

`timedelta.`**`seconds`**

0 至 86,399 开区间。

> **小心**
>
> 一个有点常见的代码错误是当实际是要获取*total_seconds()* 值时无意中使用了这个属性:

```
>>> from datetime import timedelta
>>> duration = timedelta(seconds=11235813)
>>> duration.days, duration.seconds
(130, 3813)
>>> duration.total_seconds()
11235813.0
```

timedelta.**microseconds**

0 至 999,999 开区间。

支持的运算：

运算	结果：
`t1 = t2 + t3`	`t2` 和 `t3` 之和。运算后 `t1 - t2 == t3` 且 `t1 - t3 == t2` 为真值。(1)
`t1 = t2 - t3`	`t2` 和 `t3` 之差。运算后 `t1 == t2 - t3` 且 `t2 == t1 + t3` 为真值。(1)(6)
`t1 = t2 * i or t1 = i * t2`	时差乘以一个整数。运算后如果 `i != 0` 则 `t1 // i == t2` 为真值。
	通常情况下，`t1 * i == t1 * (i-1) + t1` 为真值。(1)
`t1 = t2 * f or t1 = f * t2`	乘以一个浮点数，结果会被舍入到 timedelta 最接近的整数倍。精度使用四舍五偶入奇不入规则。
`f = t2 / t3`	总时长 `t2` 除以间隔单位 `t3` (3)。返回一个*float* 对象。
`t1 = t2 / f or t1 = t2 / i`	除以一个浮点数或整数。结果会被舍入到 timedelta 最接近的整数倍。精度使用四舍五偶入奇不入规则。
`t1 = t2 // i or t1 = t2 // t3`	计算底数，其余部分（如果有）将被丢弃。在第二种情况下，将返回整数。(3)
`t1 = t2 % t3`	余数为一个*timedelta* 对象。(3)
`q, r = divmod(t1, t2)`	计算商和余数: `q = t1 // t2` (3) 和 `r = t1 % t2`。`q` 是一个整数而 `r` 是一个*timedelta* 对象。
`+t1`	返回一个相同数值的*timedelta* 对象。
`-t1`	等价于 `timedelta(-t1.days, -t1.seconds, -t1.microseconds)`，以及 `t1 * -1`。(1)(4)
`abs(t)`	当 `t.days >= 0` 时等于 `+t`，而当 `t.days < 0` 时等于 `-t`。(2)
`str(t)`	返回一个形如 `[D day[s], ][H]H:MM:SS[.UUUUUU]` 的字符串，当 `t` 为负数的时候，D 也为负数。(5)
`repr(t)`	返回一个*timedelta* 对象的字符串表示形式，作为附带正规属性值的构造器调用。

注释：

(1) 结果正确，但可能会溢出。

(2) 结果正确，不会溢出。

(3) 除以零将会引发*ZeroDivisionError*。

(4) `-timedelta.max` 不可以表示为一个*timedelta* 对象。

(5) *timedelta* 对象的字符串表示形式类似于其内部表示形式被规范化。对于负时间增量，这会导致一些不寻常的结果。例如:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) 表达式 `t2 - t3` 通常与 `t2 + (-t3)` 是等价的，除非 t3 等于 `timedelta.max`; 在这种情况下前者会返回结果，而后者则会溢出。

除了上面列举的操作以外，*timedelta* 对象还支持与*date* 和*datetime* 对象进行特定的相加和相减运算（见下文）。

在 3.2 版本发生变更: 现在已支持*timedelta* 对象与另一个*timedelta* 对象相整除或相除，包括求余运算和*divmod()* 函数。现在也支持*timedelta* 对象加上或乘以一个*float* 对象。

timedelta 对象支持相等性和顺序比较。

在布尔运算中，*timedelta* 对象当且仅当其不等于 `timedelta(0)` 时则会被视为真值。

实例方法：

`timedelta.`**`total_seconds`**`()`

返回期间占用了多少秒。等价于 `td / timedelta(seconds=1)`。对于秒以外的间隔单位，直接使用除法形式 (例如 `td / timedelta(microseconds=1)`)。

需要注意的是，时间间隔较大时，这个方法的结果中的微秒将会失真（大多数平台上大于 270 年视为一个较大的时间间隔）。

Added in version 3.2.

`timedelta` 用法示例

一个标准化的附加示例:

```
>>> # another_year 的部分增加恰好 365 天
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

timedelta 算术运算的示例:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 `date` 对象

date 对象代表一个理想化历法中的日期（年、月和日），即当今的格列高利历向前后两个方向无限延伸。公元 1 年 1 月 1 日是第 1 日，公元 1 年 1 月 2 日是第 2 日，依此类推。[2]

`class` `datetime.`**`date`**`(`*year*, *month*, *day*`)`

所有参数都是必要的。参数必须是在下面范围内的整数：

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= 日期 <= 给定年月对应的天数`

[2] 这与 Dershowitz 和 Reingold 所著 *Calendrical Calculations* 中“预期格列高利”历法的定义一致，它是适用于该书中所有运算的基础历法。请参阅该书了解在预期格利高利历序列与许多其他历法系统之间进行转换的算法。

如果参数不在这些范围内，则抛出*ValueError* 异常。

其它构造器，所有的类方法：

classmethod date.**today**()

返回当前的本地日期。

这等价于 date.fromtimestamp(time.time())。

classmethod date.**fromtimestamp**(*timestamp*)

返回对应于 POSIX 时间戳的当地时间，例如*time.time()* 返回的就是时间戳。

这可能引发*OverflowError*，如果时间戳数值超出所在平台 C localtime() 函数的支持范围的话，并且会在 localtime() 出错时引发*OSError*。通常该数值会被限制在 1970 年至 2038 年之间。请注意在时间戳概念包含闰秒的非 POSIX 系统上，闰秒会被*fromtimestamp()* 所忽略。

在 3.3 版本发生变更: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C localtime() 函数的支持范围的话，并会在 localtime() 出错时引发*OSError* 而不是*ValueError*。

classmethod date.**fromordinal**(*ordinal*)

返回对应于预期格列高利历序号的日期，其中公元 1 年 1 月 1 日的序号为 1。

除非 1 <= ordinal <= date.max.toordinal() 否则会引发*ValueError*。对于任意的日期 d，均有 date.fromordinal(d.toordinal()) == d。

classmethod date.**fromisoformat**(*date_string*)

返回一个对应于以任何有效 ISO 8601 格式给出的 *date_string* 的*date*，下列格式除外：

1. 目前不支持降低精度的日期 (YYYY-MM, YYYY)。
2. 目前不支持扩展日期表示形式 (±YYYYYY-MM-DD)。
3. 目前不支持序数日期 (YYYY-OOO)。

示例:

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('2021-W01-1')
datetime.date(2021, 1, 4)
```

Added in version 3.7.

在 3.11 版本发生变更: 在之前版本中，此方法仅支持一种格式 YYYY-MM-DD。

classmethod date.**fromisocalendar**(*year*, *week*, *day*)

返回指定 year, week 和 day 所对应 ISO 历法日期的*date*。这是函数*date.isocalendar()* 的逆操作。

Added in version 3.8.

类属性：

date.**min**

最小的日期 date(MINYEAR, 1, 1)。

date.**max**

最大的日期，date(MAXYEAR, 12, 31)。

date.**resolution**

两个日期对象的最小间隔，timedelta(days=1)。

实例属性（只读）：

date.**year**

在*MINYEAR* 和*MAXYEAR* 之间，包含边界。

date.**month**

1 至 12（含）

date.**day**

返回 1 到指定年月的天数间的数字。

支持的运算：

运算	结果：
`date2 = date1 + timedelta`	`date2` 将为 `date1` 之后的 `timedelta.days` 日。(1)
`date2 = date1 - timedelta`	计算 `date2` 使得 `date2 + timedelta == date1`。(2)
`timedelta = date1 - date2`	(3)
`date1 == date2` `date1 != date2`	相等性比较。(4)

`date1 < date2`
`date1 > date2`
`date1 <= date2`
`date1 >= date2`
	顺序比较。(5)

注释：

(1) 如果 `timedelta.days > 0` 则 *date2* 将在时间线上前进，如果 `timedelta.days < 0` 则将后退。操作完成后 `date2 - date1 == timedelta.days`。`timedelta.seconds` 和 `timedelta.microseconds` 会被忽略。如果 `date2.year` 将小于*MINYEAR* 或大于*MAXYEAR* 则会引发*OverflowError*。

(2) `timedelta.seconds` 和 `timedelta.microseconds` 会被忽略。

(3) 该值是精确的，且不会溢出。运算后 `timedelta.seconds` 和 `timedelta.microseconds` 均为 0，且 `date2 + timedelta == date1`。

(4) *date* 对象在表示相同的日期时相等。

不属于*datetime* 实例的 `date` 对象永远不会与 `datetime` 对象相等，即使它们表示相同的日期。

(5) 当 *date1* 的时间在 *date2* 之前则认为 *date1* 小于 *date2*。换句话说，当且仅当 `date1.toordinal() < date2.toordinal()` 时 `date1 < date2`。

不同时为*datetime* 实例的 `date` 实例和 `datetime` 对象之间的顺序比较将会引发*TypeError*。

在 3.13 版本发生变更: 在*datetime* 对象和不属于 `datetime` 子类的*date* 子类的实例之间进行比较时不会再将后者转换为 `date`，并忽略时间部分和时区信息。此默认行为可以通过在子类中重写特殊比较方法来更改。

在布尔运算中，所有*date* 对象都会被视为真值。

实例方法：

date.**replace**(*year=self.year*, *month=self.month*, *day=self.day*)

返回一个具有同样的值，但更新了指定形参的新的*date* 对象。

示例：

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

泛型函数*copy.replace()* 也支持*date* 对象。

date.**timetuple**()

返回一个*time.struct_time*，即*time.localtime()* 所返回的类型。

hours, minutes 和 seconds 值均为 0，且 DST 旗标值为 -1。

`d.timetuple()` 等价于:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

其中 `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` 是当前年份中的日期序号，起始值 1 表示 1 月 1 日。

date.**toordinal**()

返回日期的预期格列高利历序号，其中公元 1 年 1 月 1 日的序号为 1. 对于任意的*date* 对象 `d`，均有 `date.fromordinal(d.toordinal()) == d`。

date.**weekday**()

返回一个整数代表星期几，星期一为 0，星期天为 6。例如，`date(2002, 12, 4).weekday() == 2`，表示的是星期三。参阅*isoweekday()*。

date.**isoweekday**()

返回一个整数代表星期几，星期一为 1，星期天为 7。例如：`date(2002, 12, 4).isoweekday() == 3`, 表示星期三。参见*weekday()*, *isocalendar()*。

date.**isocalendar**()

返回一个由三部分组成的*named tuple* 对象: `year`, `week` 和 `weekday`。

ISO 历法是一种被广泛使用的格列高利历。[3]

ISO 年由 52 或 53 个完整星期构成，每个星期开始于星期一结束于星期日。一个 ISO 年的第一个星期就是（格列高利）历法的一年中第一个包含星期四的星期。这被称为 1 号星期，这个星期四所在的 ISO 年与其所在的格列高利年相同。

例如，2004 年的第一天是星期四，因此 ISO 2004 年的第一个星期开始于 2003 年 12 月 29 日星期一，结束于 2004 年 1 月 4 日星期日:

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

在 3.9 版本发生变更: 结果由元组改为*named tuple*。

date.**isoformat**()

返回一个以 ISO 8601 格式 `YYYY-MM-DD` 来表示日期的字符串:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

date.**__str__**()

对于日期 `d`，`str(d)` 等价于 `d.isoformat()`。

[3] 请参阅 R. H. van Gent 所著 ISO 8601 历法的数学指南 以获取更完整的说明。

date.**ctime**()

返回一个表示日期的字符串:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

d.ctime() 等效于:

```
time.ctime(time.mktime(d.timetuple()))
```

在原生 C ctime() 函数遵循 C 标准的平台上 (*time.ctime()* 会发起对该函数的调用，但*date.ctime()* 并不会) 。

date.**strftime**(*format*)

返回一个由显式格式字符串所控制的，代表日期的字符串。表示时、分或秒的格式代码值将为 0。另请参阅*strftime() 和 strptime() 的行为* 和*date.isoformat()*。

date.**__format__**(*format*)

与*date.strftime()* 相同。此方法使得在 格式化字符串字面值中以及使用*str.format()* 时为*date* 对象指定格式字符串成为可能。另请参阅*strftime() 和 strptime() 的行为* 和*date.isoformat()*。

date 用法示例

计算距离特定事件天数的例子:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
...
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

使用*date* 的更多例子:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 1. 1. 0001 之后的第 730920 天
>>> d
datetime.date(2002, 3, 11)

>>> # 有关格式化字符串输出的方法
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

(续下页)

(接上页)

```
>>> # 用于提取不同历法中的‘部分’的方法
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # 年
3                   # 月
11                  # 日
0
0
0
0                   # 周序号 (0 = 星期一)
70                  # 当年的第 70 天
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO 年
11                  # ISO 第几周
1                   # ISO 周序号 ( 1 = 星期一 )

>>> # 日期对象是不可变的；所有操作都将产生一个新对象
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

8.1.6 datetime 对象

datetime 对象是包含来自 *date* 对象和 *time* 对象的所有信息的单一对象。

与 *date* 对象一样，*datetime* 假定当前的格列高利历向前后两个方向无限延伸；与 *time* 对象一样，*datetime* 假定每一天恰好有 3600*24 秒。

构造器：

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

year, *month* 和 *day* 参数是必须的。*tzinfo* 可以是 None 或者是一个 *tzinfo* 子类的实例。其余的参数必须是在下面范围内的整数：

- MINYEAR <= year <= MAXYEAR,
- 1 <= month <= 12,
- 1 <= day <= 指定年月的天数,
- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

如果参数不在这些范围内，则抛出 *ValueError* 异常。

在 3.6 版本发生变更: 增加了 *fold* 形参。

其它构造器，所有的类方法：

classmethod datetime.**today**()

返回表示当前地方时的 date 和 time，其中 *tzinfo* 为 None。

等价于:

```
datetime.fromtimestamp(time.time())
```

另请参阅*now()*，*fromtimestamp()*。

此方法的功能等价于*now()*，但是不带 `tz` 形参。

classmethod datetime.**now**(*tz=None*)

返回表示当前地方时的 date 和 time 对象。

如果可选参数 *tz* 为 `None` 或未指定，这就类似于*today()*，但该方法会在可能的情况下提供比通过*time.time()* 时间戳所获时间值更高的精度（例如，在提供了 C `gettimeofday()` 函数的平台上就可以做到这一点）。

如果 *tz* 不为 `None`，它必须是*tzinfo* 子类的一个实例，并且当前日期和时间将被转换到 *tz* 时区。

此函数可以替代*today()* 和*utcnow()*。

备注

对 `datetime.now()` 的后续调用可能由于下层时钟的精度返回相同的时刻。

classmethod datetime.**utcnow**()

返回表示当前 UTC 时间的 date 和 time，其中*tzinfo* 为 `None`。

这类似于*now()*，但返回的是当前 UTC 日期和时间，类型为简单型*datetime* 对象。感知型的当前 UTC 日期时间可通过调用 `datetime.now(timezone.utc)` 来获得。另请参阅*now()*。

警告

由于简单型 `datetime` 对象会被许多 `datetime` 方法当作本地时间来处理，最好是使用感知型日期时间对象来表示 UTC 时间。因此，创建表示当前 UTC 时间的对象的推荐方式是通过调用 `datetime.now(timezone.utc)`。

自 3.12 版本弃用: 改用带*UTC* 的*datetime.now()*。

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)

返回 POSIX 时间戳对应的本地日期和时间，如*time.time()* 返回的。如果可选参数 *tz* 指定为 `None` 或未指定，时间戳将转换为平台的本地日期和时间，并且返回的*datetime* 对象将为简单型。

如果 *tz* 不为 `None`，它必须是*tzinfo* 子类的一个实例，并且时间戳将被转换到 *tz* 指定的时区。

fromtimestamp() 可能会引发*OverflowError*，如果时间戳数值超出所在平台 C `localtime()` 或 `gmtime()` 函数的支持范围的话，并会在 `localtime()` 或 `gmtime()` 报错时引发*OSError*。通常该数值会被限制在 1970 年至 2038 年之间。请注意在时间戳概念包含闰秒的非 POSIX 系统上，闰秒会被*fromtimestamp()* 所忽略，结果可能导致两个相差一秒的时间戳产生相同的*datetime* 对象。相比*utcfromtimestamp()* 更推荐使用此方法。

在 3.3 版本发生变更: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C `localtime()` 或 `gmtime()` 函数的支持范围的话。并会在 `localtime()` 或 `gmtime()` 出错时引发*OSError* 而不是*ValueError*。

在 3.6 版本发生变更: *fromtimestamp()* 可能返回*fold* 值设为 1 的实例。

classmethod datetime.**utcfromtimestamp**(*timestamp*)

返回对应于 POSIX 时间戳的 UTC *datetime*，其中*tzinfo* 值为 `None`。（结果为简单型对象。）

这可能引发*OverflowError*，如果时间戳数值超出所在平台 C `gmtime()` 函数的支持范围的话，并会在 `gmtime()` 报错时引发*OSError*。通常该数值会被限制在 1970 至 2038 年之间。

要得到一个感知型*datetime* 对象，应调用*fromtimestamp()*：

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

在 POSIX 兼容的平台上，它等价于以下表达式:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

不同之处在于后一种形式总是支持完整年份范围：从*MINYEAR* 到*MAXYEAR* 的开区间。

> **警告**
>
> 由于简单型 `datetime` 对象会被许多 `datetime` 方法当作本地时间来处理，最好是使用感知型日期时间对象来表示 UTC 时间。因此，创建表示特定 UTC 时间戳的日期时间对象的推荐方式是通过调用 `datetime.fromtimestamp(timestamp, tz=timezone.utc)`。

在 3.3 版本发生变更: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C `gmtime()` 函数的支持范围的话。并会在 `gmtime()` 出错时引发*OSError* 而不是*ValueError*。

自 3.12 版本弃用: 改用带*UTC* 的*datetime.fromtimestamp()*。

classmethod datetime.**fromordinal**(*ordinal*)

返回对应于预期格列高利历序号的*datetime*，其中公元 1 年 1 月 1 日的序号为 1。除非 `1 <= ordinal <= datetime.max.toordinal()` 否则会引发*ValueError*。结果的 hour, minute, second 和 microsecond 值均为 0，并且*tzinfo* 值为 `None`。

classmethod datetime.**combine**(*date*, *time*, *tzinfo=time.tzinfo*)

返回一个新的*datetime* 对象，其日期部分等于给定的*date* 对象的值，而其时间部分等于给定的*time* 对象的值。如果提供了 *tzinfo* 参数，其值会被用来设置结果的*tzinfo* 属性，否则将使用 *time* 参数的*tzinfo* 属性。如果 *date* 参数是一个*datetime* 对象，则其时间部分和*tzinfo* 属性将被忽略。

对于任意*datetime* 对象 `d`，`d == datetime.combine(d.date(), d.time(), d.tzinfo)`。

在 3.6 版本发生变更: 增加了 *tzinfo* 参数。

classmethod datetime.**fromisoformat**(*date_string*)

返回一个对应于以任何有效的 8601 格式给出的 *date_string* 的*datetime*，下列格式除外:

1. 时区时差可能会有带小数的秒值。
2. `T` 分隔符可以用任何单个 unicode 字符来替换。
3. 带小数的时和分是不受支持的。
4. 目前不支持降低精度的日期 (`YYYY-MM`, `YYYY`)。
5. 目前不支持扩展日期表示形式 (`±YYYYYY-MM-DD`)。
6. 目前不支持序数日期 (`YYYY-OOO`)。

示例:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('20111104')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04T00:05:23Z')
datetime.datetime(2011, 11, 4, 0, 5, 23, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('20111104T000523')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-W01-2T00:05:23.283')
datetime.datetime(2011, 1, 4, 0, 5, 23, 283000)
```

(续下页)

(接上页)

```
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

Added in version 3.7.

在 3.11 版本发生变更: 在之前版本中，此方法仅支持可以由*date.isoformat()* 或*datetime.isoformat()* 发出的格式。

classmethod datetime.**fromisocalendar**(*year*, *week*, *day*)

返回以 year, week 和 day 值指明的 ISO 历法日期所对应的*datetime*。该 datetime 对象的非日期部分将使用其标准默认值来填充。这是函数*datetime.isocalendar()* 的逆操作。

Added in version 3.8.

classmethod datetime.**strptime**(*date_string*, *format*)

返回一个对应于 *date_string*，根据 *format* 进行解析得到的*datetime* 对象。

如果 *format* 不包含微秒或时区信息，这将等价于:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

如果 date_string 和 format 无法被*time.strptime()* 解析或它返回一个不是时间元组的值则将引发*ValueError*。另请参阅*strftime() 和 strptime() 的行为* 和*datetime.fromisoformat()*。

在 3.13 版本发生变更: 现在如果 *format* 指定了月份日期但没有年份则会发出*DeprecationWarning*。这是为了避免在代码中当格式缺少年份仅能解析月份和日期时将使用非闰年的默认年份而导致四年轮回的闰年程序错误。这样的 *format* 值在 Python 3.15 将可能引发错误。绕过此问题的办法是始终在你的 *format* 中包括年份。如果是解析不带年份的 *date_string* 值，则在解析之前显式地添加一个属于闰年的年份:

```
>>> from datetime import datetime
>>> date_string = "02/29"
>>> when = datetime.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap year bug.
>>> when.strftime("%B %d")
'February 29'
```

类属性:

datetime.**min**

最早的可表示*datetime*，datetime(MINYEAR, 1, 1, tzinfo=None)。

datetime.**max**

最晚的可表示*datetime*，datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)。

datetime.**resolution**

两个不相等的*datetime* 对象之间可能的最小间隔，timedelta(microseconds=1)。

实例属性（只读):

datetime.**year**

在*MINYEAR* 和*MAXYEAR* 之间，包含边界。

datetime.**month**

1 至 12（含）

datetime.**day**

返回 1 到指定年月的天数间的数字。

datetime.**hour**

取值范围是 range(24)。

datetime.**minute**

取值范围是 range(60)。

datetime.**second**

取值范围是 range(60)。

datetime.**microsecond**

取值范围是 range(1000000)。

datetime.**tzinfo**

作为 *tzinfo* 参数被传给 *datetime* 构造器的对象，如果没有传入值则为 None。

datetime.**fold**

取值范围是 [0, 1]。用于在重复的时间段中消除边界时间的歧义。(当夏令时结束时回拨时钟或由于政治原因导致当前时区的 UTC 时差减少就会出现重复时间段。) 取值 0 和 1 分别表示两个相同边界时间表示形式的前一个和后一个时间。

Added in version 3.6.

支持的运算：

运算	结果：
datetime2 = datetime1 + timedelta	(1)
datetime2 = datetime1 - timedelta	(2)
timedelta = datetime1 - datetime2	(3)
datetime1 == datetime2 datetime1 != datetime2	相等性比较。(4)
datetime1 < datetime2 datetime1 > datetime2 datetime1 <= datetime2 datetime1 >= datetime2	顺序比较。(5)

(1) datetime2 是 datetime1 去掉 timedelta 时间段的结果，如果 timedelta.days > 0 则是在时间线上前进，如果 timedelta.days < 0 则时在时间线上后退。该结果具有与输入的 datetime 相同的 *tzinfo* 属性，并且运算后 datetime2 - datetime1 == timedelta。如果 datetime2.year 将要小于 *MINYEAR* 或大于 *MAXYEAR* 则会引发 *OverflowError*。请注意即使输入的是一个感知型对象该方法也不会进行时区调整。

(2) 计算 datetime2 使得 datetime2 + timedelta == datetime1。与相加操作一样，结果具有与输入的 datetime 相同的 *tzinfo* 属性，即使输入的是一个感知型对象该方法也不会进行时区调整。

(3) 从一个 *datetime* 减去一个 *datetime* 仅对两个操作数均为简单型或均为感知型时有定义。如果一个是感知型而另一个是简单型，则会引发 *TypeError*。

如果两个操作数都是简单型，或都是感知型并且具有相同的 *tzinfo* 属性，则 *tzinfo* 属性会被忽略，并且结果会是一个使得 datetime2 + t == datetime1 的 *timedelta* 对象 t。在此情况下不会进行时区调整。

如果两者均为感知型且具有不同的 *tzinfo* 属性，a-b 的效果就如同 a 和 b 首先被转换为简单型 UTC 日期时间。结果将为 (a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())，不同之处在于具体实现绝对不会溢出。

(4) *datetime* 对象如果在考虑时区的情况下表示相同的日期和时间那么就是相等的。

简单型和感知型 datetime 对象绝不会相等。

如果两个操作数均为感知型，且具有相同的 tzinfo 属性，则 tzinfo 和*fold* 属性将被忽略并对基本日期时间值进行比较。如果两个操作数均为感知型且具有不同的*tzinfo* 属性，则比较行为将如同两个操作数首先被转换为 UTC，不同之处是具体实现绝对不会溢出。具有重复间隔的 datetime 实例绝对不会等于属性其他时区的 datetime 实例。

(5) 在考虑时区的情况下，当 *datetime1* 的时间在 *datetime2* 之前则认为 *datetime1* 小于 *datetime2*。

简单型和感知型*datetime* 对象之间的顺序比较将会引发*TypeError*。

如果两个操作数均为感知型，且具有相同的 tzinfo 属性，则 tzinfo 和*fold* 属性将被忽略并对基本日期时间值进行比较。如果两个操作数均为感知型且具有不同的*tzinfo* 属性，则比较行为将如同两个操作数首先被转换为 UTC 日期时间，不同之处是具体实现绝对不会溢出。

在 3.3 版本发生变更: 感知型和简单型*datetime* 实例之间的相等比较不会引发*TypeError*。

在 3.13 版本发生变更: 在*datetime* 对象和不属于 datetime 子类的*date* 子类的实例之间进行比较时不会再将后者转换为 date，并忽略时间部分和时区信息。此默认行为可以通过在子类中重写特殊比较方法来更改。

实例方法：

datetime.**date**()

返回具有同样 year, month 和 day 值的*date* 对象。

datetime.**time**()

返回具有同样 hour, minute, second, microsecond 和 fold 值的*time* 对象。*tzinfo* 值为 None。另请参见*timetz()* 方法。

在 3.6 版本发生变更: fold 值会被复制给返回的*time* 对象。

datetime.**timetz**()

返回具有同样 hour, minute, second, microsecond, fold 和 tzinfo 属性性的*time* 对象。另请参见*time()* 方法。

在 3.6 版本发生变更: fold 值会被复制给返回的*time* 对象。

datetime.**replace**(*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*)

返回一个具有同样属性的新的*datetime* 对象，但更新指定的形参。请注意可以通过指定 tzinfo=None 基于一个感知型 datetime 创建一个简单型 datetime 而不必转换日期和时间数据。

datetime 对象也被泛型函数*copy.replace()* 所支持。

在 3.6 版本发生变更: 增加了 *fold* 形参。

datetime.**astimezone**(*tz=None*)

返回一个具有新的*tzinfo* 属性 *tz* 的*datetime* 对象，并会调整日期和时间数据使得结果对应的 UTC 时间与 *self* 相同，但为 *tz* 时区的本地时间。

如果给出了 *tz*，则它必须是一个*tzinfo* 子类的实例，并且其*utcoffset()* 和*dst()* 方法不可返回 None。如果 *self* 为简单型，它会被假定为基于系统时区表示的时间。

如果调用时不传入参数 (或传入 tz=None) 则将假定目标时区为系统的本地时区。转换后 datetime 实例的 .tzinfo 属性将被设为一个*timezone* 实例，时区名称和时差值将从 OS 获取。

如果 self.tzinfo 为 *tz*，self.astimezone(tz) 等于 *self*：不会对日期或时间数据进行调整。否则结果为 *tz* 时区的本地时间，代表的 UTC 时间与 *self* 相同：在 astz = dt.astimezone(tz) 之后，astz - astz.utcoffset() 将具有与 dt - dt.utcoffset() 相同的日期和时间数据。

如果你只是想要附加一个*timezone* 对象 *tz* 到一个 datetime 对象 *dt* 而不调整日期和时间数据，请使用 dt.replace(tzinfo=tz)。如果你只是想要从一个感知型 datetime 对象 *dt* 移除 timezone 对象，请使用 dt.replace(tzinfo=None)。

请注意默认的*tzinfo.fromutc()* 方法在*tzinfo* 的子类中可以被重写，从而影响*astimezone()* 的返回结果。如果忽略出错的情况，*astimezone()* 的行为就类似于:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # 将 self 转换为 UTC，并附加新的时区对象
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # 从 UTC 转换为 tz 的地方时。
    return tz.fromutc(utc)
```

在 3.3 版本发生变更: *tz* 现在可以被省略。

在 3.6 版本发生变更: *astimezone()* 方法可以由简单型实例调用，这将假定其表示本地时间。

datetime.**utcoffset**()

如果*tzinfo* 为 None，则返回 None，否则返回 `self.tzinfo.utcoffset(self)`，并且在后者不返回 None 或者一个幅度小于一天的*timedelta* 对象时将引发异常。

在 3.7 版本发生变更: UTC 时差不再限制为一个整数分钟值。

datetime.**dst**()

如果*tzinfo* 为 None，则返回 None，否则返回 `self.tzinfo.dst(self)`，并且在后者不返回 None 或者一个幅度小于一天的*timedelta* 对象时将引发异常。

在 3.7 版本发生变更: DST 差值不再限制为一个整数分钟值。

datetime.**tzname**()

如果*tzinfo* 为 None，则返回 None，否则返回 `self.tzinfo.tzname(self)`，如果后者不返回 None 或者一个字符串对象则将引发异常。

datetime.**timetuple**()

返回一个*time.struct_time*，即*time.localtime()* 所返回的类型。

`d.timetuple()` 等价于:

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

其中 `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` 是日期在当前年份中的序号，起始值 1 表示 1 月 1 日。结果的*tm_isdst* 旗标会根据*dst()* 方法来设定：如果*tzinfo* 为 None 或*dst()* 返回 None，则 tm_isdst 将设为 -1；否则如果*dst()* 返回非零值，则 tm_isdst 将设为 1；在其他情况下 tm_isdst 将设为 0。

datetime.**utctimetuple**()

如果*datetime* 实例 d 为简单型，这将与 `d.timetuple()` 相当，区别在于*tm_isdst* 会被强制设为 0 而无视 `d.dst()` 返回值。DST 对于 UTC 时间必定无效。

如果 d 为感知型，则 d 会通过减去 `d.utcoffset()` 来标准化为 UTC 时间，并返回表示该标准化时间的*time.struct_time*。tm_isdst 将被强制设为 0。请注意如果 `d.year` 为 MINYEAR 或 MAXYEAR 且 UTC 调整超出一年的边界则可能引发*OverflowError*。

> **警告**
>
> 由于简单型 datetime 对象会被许多 datetime 方法当作本地时间来处理，最好是使用感知型日期时间来表示 UTC 时间；因此，使用*datetime.utctimetuple()* 可能会给出误导性的结果。如果你有一个表示 UTC 的简单型 datetime，请使用 `datetime.replace(tzinfo=timezone.utc)` 将其改为感知型，这样你才能使用*datetime.timetuple()*。

datetime.**toordinal**()

返回日期的预期格列高利历序号。与 `self.date().toordinal()` 相同。

datetime.**timestamp**()

返回对应于*datetime* 实例的 POSIX 时间戳。此返回值是与*time.time()* 返回值类似的*float* 对象。

简单型*datetime* 实例会被假定为代表本地时间并且此方法依赖于平台的 C `mktime()` 函数来执行转换。由于在许多平台上*datetime* 支持的取值范围比 `mktime()` 更广，对于极其遥远的过去或未来此方法可能会引发*OverflowError* 或*OSError*。

对于感知型*datetime* 实例，返回值的计算方式为:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

Added in version 3.3.

在 3.6 版本发生变更: *timestamp()* 方法使用*fold* 属性来消除重复间隔中的时间歧义。

备注

没有一个方法能直接从表示 UTC 时间的简单型*datetime* 实例获取 POSIX 时间戳。如果你的应用程序使用此惯例并且你的系统时区不是设为 UTC，你可以通过提供 `tzinfo=timezone.utc` 来获取 POSIX 时间戳:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

或者通过直接计算时间戳:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

datetime.**weekday**()

返回一个整数代表星期几，星期一为 0，星期天为 6。相当于 `self.date().weekday()`。另请参阅*isoweekday()*。

datetime.**isoweekday**()

返回一个整数代表星期几，星期一为 1，星期天为 7。相当于 `self.date().isoweekday()`。另请参阅*weekday()*, *isocalendar()*。

datetime.**isocalendar**()

返回一个由三部分组成的*named tuple*: `year`, `week` 和 `weekday`。等同于 `self.date().isocalendar()`。

datetime.**isoformat**(*sep='T'*, *timespec='auto'*)

返回一个以 ISO 8601 格式表示的日期和时间字符串:

- `YYYY-MM-DDTHH:MM:SS.ffffff`，如果*microsecond* 不为 0
- `YYYY-MM-DDTHH:MM:SS`，如果*microsecond* 为 0

如果*utcoffset()* 返回值不为 `None`，则添加一个字符串来给出 UTC 时差:

- `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`，如果*microsecond* 不为 0
- `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`，如果*microsecond* 为 0

示例:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

可选参数 *sep* (默认为 `'T'`) 为单个分隔字符，会被放在结果的日期和时间两部分之间。例如:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

可选参数 *timespec* 要包含的额外时间组件值 (默认为 `'auto'`)。它可以是以下值之一:

- `'auto'`: 如果*microsecond* 为 0 则与 `'seconds'` 相同，否则与 `'microseconds'` 相同。
- `'hours'`: 以两个数码的 `HH` 格式包含*hour*。
- `'minutes'`: 以 `HH:MM` 格式包含*hour* 和*minute*。
- `'seconds'`: 以 `HH:MM:SS` 格式包含*hour*, *minute* 和*second*。
- `'milliseconds'`: 包含完整时间，但将秒值的小数部分截断至毫秒。格式为 `HH:MM:SS.sss`。
- `'microseconds'`: 以 `HH:MM:SS.ffffff` 格式包含完整时间。

> **备注**
>
> 排除掉的时间部分将被截断，而不是被舍入。

对于无效的 *timespec* 参数将引发*ValueError*:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

在 3.6 版本发生变更: 增加了 *timespec* 形参。

`datetime.__str__()`

对于*datetime* 实例 `d`，`str(d)` 等价于 `d.isoformat(' ')`。

`datetime.ctime()`

返回一个表示日期和时间的字符串:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

输出字符串将 *并不* 包括时区信息，无论输入的是感知型还是简单型。

`d.ctime()` 等效于:

```
time.ctime(time.mktime(d.timetuple()))
```

在原生 C `ctime()` 函数遵循 C 标准的平台上 (*time.ctime()* 会发起对该函数的调用，但*datetime.ctime()* 并不会)。

`datetime.strftime(format)`

返回一个由显式格式字符串所控制的，代表日期和时间的字符串。另请参阅*strftime() 和 strptime() 的行为* 和*datetime.isoformat()*。

datetime.**__format__**(*format*)

与*datetime.strftime()* 相同。此方法使得在 格式化字符串字面值中以及使用*str.format()* 时为*datetime* 对象指定格式字符串成为可能。另请参阅*strftime() 和 strptime() 的行为* 和*datetime.isoformat()*。

用法示例: datetime

使用*datetime* 对象的例子:

```
>>> from datetime import datetime, date, time, timezone

>>> # 使用 datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # 使用 datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # 使用 datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # 使用 datetime.timetuple() 来获取由所有属性组成的元组
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # 年
11      # 月
21      # 日
16      # 时
30      # 分
0       # 秒
1       # 周序号 (0 = 星期一)
325     # 自 1 月 1 日开始的天数
-1      # dst - 方法 tzinfo.dst() 返回 None

>>> # ISO 格式的日期
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO 年
47      # ISO 第几周
2       # ISO 周序号

>>> # 格式化日期时间对象
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day", "month
↪", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

以下示例定义了一个*tzinfo* 子类，它捕获 Kabul, Afghanistan 时区的信息，该时区使用 +4 UTC 直到 1945 年，之后则使用 +4:30 UTC:

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # 喀布尔曾使用 +4 直到 1945 年，后改为 +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # 带有歧义（“虚幻”）的半小时区间代表
            # 由于从 +4 改为 +4:30 导致的时间‘折叠’。
            # 如果 dt 落在此虚幻区间，则使用该折叠
            # 确定如何计算。 参见 PEP495。
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # 遵循与在 datetime.tzinfo 中相同的效力
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # 需要一个针对 fromutc 的自定义实现
        # 因为此函数的输入是 utc 日期时间值
        # 但其 tzinfo 设为 self。
        # 参见 datetime.astimezone 或 fromtimestamp。
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
        # 喀布尔不实行夏令时。
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

上述 `KabulTz` 的用法:

```
>>> tz1 = KabulTz()

>>> # 修改前的日期时间
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # 修改后的日期时间
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # 将日期时间转换至另一个时区
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
```

(续下页)

(接上页)

```
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

8.1.7 time 对象

一个*time*对象代表某日的（本地）时间，它独立于任何特定日期，并可通过*tzinfo*对象来调整。

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

所有参数都是可选的。*tzinfo* 可以是 None，或者是一个*tzinfo*子类的实例。其余的参数必须是在下面范围内的整数：

- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

如果给出一个此范围以外的参数，则会引发*ValueError*。所有参数默认值均为 0 但 *tzinfo* 除外，其默认值为 None。

类属性：

time.**min**

早最的可表示*time*, time(0, 0, 0, 0)。

time.**max**

最晚的可表示*time*, time(23, 59, 59, 999999)。

time.**resolution**

两个不相等的*time*对象之间可能的最小间隔，timedelta(microseconds=1)，但是请注意*time*对象并不支持算术运算。

实例属性（只读）：

time.**hour**

取值范围是 range(24)。

time.**minute**

取值范围是 range(60)。

time.**second**

取值范围是 range(60)。

time.**microsecond**

取值范围是 range(1000000)。

time.**tzinfo**

作为 tzinfo 参数被传给*time*构造器的对象，如果没有传入值则为 None。

time.**fold**

取值范围是 [0, 1]。用于在重复的时间段中消除边界时间的歧义。（当夏令时结束时回拨时钟或由于政治原因导致当前时区的 UTC 时差减少就会出现重复时间段。）取值 0 和 1 分别表示两个相同边界时间表示形式的前一个和后一个时间。

Added in version 3.6.

*time*对象支持相等和顺序比较，当 a 的时间在 b 之前则认为 a 小于 b。

简单型和感知型 time 对象绝对不会相等。简单型和感知型 time 对象之间的顺序比较将会引发*TypeError*。

如果两个操作数均为感知型，且具有相同的*tzinfo*属性，则 tzinfo 和 fold 属性会被忽略并对基本时间值进行比较。如果两个操作数均为感知型且具有不同的 tzinfo 属性，则两个操作数将首先通过减去它们的 UTC 时差（从 self.utcoffset() 获取）来进行调整。

在 3.3 版本发生变更: 感知型和简单型*time*实例之间的相等性比较不会引发*TypeError*。

在布尔运算时，*time*对象总是被视为真值。

在 3.5 版本发生变更: 在 Python 3.5 之前，如果一个*time*对象代表 UTC 午夜零时则会被视为假值。此行为被认为容易引发困惑和错误，因此从 Python 3.5 起已被去除。详情参见 bpo-13936。

其他构造方法：

classmethod time.**fromisoformat**(*time_string*)

返回一个对应于以任何有效的 ISO 8601 格式给出的 *time_string* 的*time*，下列格式除外:

1. 时区时差可能会有带小数的秒值。
2. 打头的 T，通常在当日期和时间之间可能存在歧义时才有必要，不是必需的。
3. 带小数的秒值可以有任意多位数码（超过 6 位将被截断）。
4. 带小数的时和分是不受支持的。

示例：

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T042301')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01,000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
>>> time.fromisoformat('04:23:01Z')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
>>> time.fromisoformat('04:23:01+00:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

Added in version 3.7.

在 3.11 版本发生变更: 在之前版本中，此方法仅支持可由*time.isoformat()*发出的格式。

实例方法：

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*)

返回一个具有同样属性的新的*time*对象，但更新指定的形参。请注意可以通过指定 tzinfo=None 基于一个感知型*time*创建一个简单型*time*，而不必转换时间数据。

*time*对象也被泛型函数*copy.replace()*所支持。

在 3.6 版本发生变更: 增加了 *fold* 形参。

time.**isoformat**(*timespec='auto'*)

返回表示为下列 ISO 8601 格式之一的时间字符串：

- HH:MM:SS.ffffff，如果*microsecond*不为 0
- HH:MM:SS，如果*microsecond*为 0
- HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]，如果*utcoffset()*不返回 None
- HH:MM:SS+HH:MM[:SS[.ffffff]]，如果*microsecond*为 0 并且*utcoffset()*不返回 None

可选参数 *timespec* 要包含的额外时间组件值 (默认为 `'auto'`)。它可以是以下值之一:

- `'auto'`: 如果*microsecond* 为 0 则与 `'seconds'` 相同，否则与 `'microseconds'` 相同。
- `'hours'`: 以两个数码的 `HH` 格式包含*hour*。
- `'minutes'`: 以 `HH:MM` 格式包含*hour* 和*minute*。
- `'seconds'`: 以 `HH:MM:SS` 格式包含*hour*, *minute* 和*second*。
- `'milliseconds'`: 包含完整时间，但将秒值的小数部分截断至毫秒。格式为 `HH:MM:SS.sss`。
- `'microseconds'`: 以 `HH:MM:SS.ffffff` 格式包含完整时间。

> **备注**
>
> 排除掉的时间部分将被截断，而不是被舍入。

对于无效的 *timespec* 参数将引发*ValueError*。

示例:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec='minutes
↪')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

在 3.6 版本发生变更: 增加了 *timespec* 形参。

`time.__str__()`

对于时间对象 `t`，`str(t)` 等价于 `t.isoformat()`。

`time.strftime(format)`

返回一个由显式格式字符串所控制的，代表时间的字符串。另请参阅*strftime() 和 strptime() 的行为* 和*time.isoformat()*。

`time.__format__(format)`

与*time.strftime()* 相同。此方法使得在 格式化字符串字面值中以及使用*str.format()* 时为*time* 对象指定格式字符串成为可能。另请参阅*strftime() 和 strptime() 的行为* 和*time.isoformat()*。

`time.utcoffset()`

如果*tzinfo* 为 `None`，则返回 `None`，否则返回 `self.tzinfo.utcoffset(None)`，并且在后者不返回 `None` 或一个幅度小于一天的 a *timedelta* 对象时将引发异常。

在 3.7 版本发生变更: UTC 时差不再限制为一个整数分钟值。

`time.dst()`

如果*tzinfo* 为 `None`，则返回 `None`，否则返回 `self.tzinfo.dst(None)`，并且在后者不返回 `None` 或者一个幅度小于一天的*timedelta* 对象时将引发异常。

在 3.7 版本发生变更: DST 差值不再限制为一个整数分钟值。

`time.tzname()`

如果*tzinfo* 为 `None`，则返回 `None`，否则返回 `self.tzinfo.tzname(None)`，如果后者不返回 `None` 或者一个字符串对象则将引发异常。

用法示例: time

使用*time* 对象的例子:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.8 tzinfo 对象

class datetime.**tzinfo**

这是一个抽象基类，也就是说该类不应被直接实例化。请定义*tzinfo* 的子类来捕获有关特定时区的信息。

tzinfo 的（某个实体子类）的实例可以被传给*datetime* 和*time* 对象的构造器。这些对象会将它们的属性视为对应于本地时间，并且*tzinfo* 对象支持展示本地时间与 UTC 的差值、时区名称以及 DST 差值的方法，都是与传给它们的日期或时间对象的相对值。

你需要派生一个实体子类，并且（至少）提供你使用*datetime* 方法所需要的标准*tzinfo* 方法的实现。datetime 模块提供了*timezone*，这是*tzinfo* 的一个简单实体子类，它能以与 UTC 的固定差值来表示不同的时区，例如 UTC 本身或北美的 EST 和 EDT。

对于封存操作的特殊要求：一个*tzinfo* 子类必须具有可不带参数调用的 __init__() 方法，否则它虽然可以被封存，但可能无法再次解封。这是个技术性要求，在未来可能会被取消。

一个*tzinfo* 的实体子类可能需要实现以下方法。具体需要实现的方法取决于感知型 datetime 对象如何使用它。如果有疑问，可以简单地全部实现它们。objects. If in doubt, simply implement all of them.

tzinfo.**utcoffset**(*dt*)

将本地时间与 UTC 时差返回为一个*timedelta* 对象，如果本地时区在 UTC 以东则为正值。如果本地时区在 UTC 以西则为负值。

这表示与 UTC 的总计时差；举例来说，如果一个*tzinfo* 对象同时代表时区和 DST 调整，则*utcoffset()* 应当返回两者的和。如果 UTC 时差不确定则返回 None。在其他情况下返回值必须为一个*timedelta* 对象，其取值严格限制于 -timedelta(hours=24) 和 timedelta(hours=24) 之间（差值的幅度必须小于一天）。大多数*utcoffset()* 的实现看起来可能像是以下两者之一:

```
return CONSTANT                 # 固定偏移类
return CONSTANT + self.dst(dt)  # 夏令时感知类
```

如果*utcoffset()* 返回值不为 None，则*dst()* 也不应返回 None。

默认的*utcoffset()* 实现会引发*NotImplementedError*。

在 3.7 版本发生变更: UTC 时差不再限制为一个整数分钟值。

tzinfo.**dst**(*dt*)

将夏令时（DST）调整返回为一个*timedelta* 对象，如果 DST 信息未知则返回 None。

如果 DST 未启用则返回 timedelta(0)。如果 DST 已启用，则将差值作为一个*timedelta* 对象返回（请参阅*utcoffset()* 了解详情）。请注意 DST 差值如果可用，就会直接被加入*utcoffset()* 所返回的 UTC 时差，因此无需额外查询*dst()*，除非你希望单独获取 DST 信息。例如，*datetime.timetuple()* 会调用其*tzinfo* 属性的*dst()* 方法来确定应该如何设置*tm_isdst* 旗标，而*tzinfo.fromutc()* 会调用*dst()* 来在跨越时区时处理 DST 的改变。

一个可以同时处理标准时和夏令时的*tzinfo* 子类的实例 *tz* 必须在此情形中保持一致:

```
tz.utcoffset(dt) - tz.dst(dt)
```

必须为具有 dt.tzinfo == tz 的每个*datetime* *dt* 返回同样的结果。对于同样的*tzinfo* 子类，此表达式会产生特定时区的“标准时差”，它不应依赖于具体日期或时间，而只依赖于地理位置。*datetime.astimezone()* 的实现依赖于此方法，但无法检测违反规则的情况；确保符合规则是程序员的责任。如果一个*tzinfo* 子类不能保证这一点，也许可以重写*tzinfo.fromutc()* 的默认实现以便在任何情况下与*astimezone()* 正确配合。

大多数*dst()* 的实现可能会如以下两者之一:

```
def dst(self, dt):
    # 固定偏移类: 不考虑夏令时
    return timedelta(0)
```

或者:

```
def dst(self, dt):
    # 此代码根据输入的 dt.year 设置时区的夏令时
    # 切换的开始和结束时刻 dston 和 dstoff, 并以
    # 标准地方时表示。

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

默认的*dst()* 实现会引发*NotImplementedError*。

在 3.7 版本发生变更: DST 差值不再限制为一个整数分钟值。

tzinfo.**tzname**(*dt*)

将对应于*datetime* 对象 *dt* 的时区名称作为字符串返回。datetime 模块未定义任何有关字符串名称的内容，也不要求它具有任何特定含义。例如 “”GMT”, ``"UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" 都是有效的返回值。如果字符串名称未知则返回 None。请注意这是一个方法而不是一个固定的字符串，这主要是因为某些*tzinfo* 子类可能需要根据所传入的特定 *dt* 值返回不同的名称，特别是在*tzinfo* 类要负责处理夏令时的场合中。

默认的*tzname()* 实现会引发*NotImplementedError*。

这些方法会被*datetime* 或*time* 对象调用，用来与它们的同名方法相对应。*datetime* 对象会将自身作为传入参数，而*time* 对象会将 None 作为传入参数。这样*tzinfo* 子类的方法应当准备好接受 *dt* 参数值为 None 或是*datetime* 类的实例。

当传入 None 时，应当由类的设计者来决定最佳回应方式。例如，返回 None 适用于希望该类提示时间对象不参与*tzinfo* 协议处理。让 utcoffset(None) 返回标准 UTC 时差也许会更有用处，因为并没有其他可用于发现标准时差的约定惯例。

当传入一个*datetime* 对象来回应*datetime* 方法时，dt.tzinfo 与 *self* 是同一对象。*tzinfo* 方法可以依赖这一点，除非用户代码直接调用了*tzinfo* 方法。此行为的目的是使得*tzinfo* 方法将 *dt* 解读为本地时间，而不需要担心其他时区的相关对象。

还有一个额外的*tzinfo*方法，某个子类可能会希望重写它：

tzinfo.**fromutc**(*dt*)

此方法会由默认的*datetime.astimezone()*实现来调用。当被其调用时，dt.tzinfo 为 *self*，并且 *dt* 的日期和时间数据会被视为表示 UTC 时间。*fromutc()* 的目标是调整日期和时间数据，返回一个等价的表示 *self* 的本地时间的 datetime。

大多数*tzinfo*子类应该能够毫无问题地继承默认的*fromutc()*实现。它的健壮性足以处理固定差值的时区以及同时负责标准时和夏令时的时区，对于后者甚至还能处理 DST 转换时间在各个年份有变化的情况。一个默认*fromutc()*实现可能无法在所有情况下正确处理的例子是（与 UTC 的）标准时差取决于所经过的特定日期和时间，这种情况可能由于政治原因而出现。默认的*astimezone()*和*fromutc()*实现可能无法生成你希望的结果，如果这个结果恰好是跨越了标准时差发生改变的时刻当中的某个小时值的话。

忽略针对错误情况的代码，默认*fromutc()*实现的行为方式如下：

```
def fromutc(self, dt):
    # 如果 dt.tzinfo 不为 self 则引发 ValueError
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # 如果 dtoff 为 None 或 dtdst 为 None 则引发 ValueError
    delta = dtoff - dtdst  # 这是 self 的标准偏移
    if delta:
        dt += delta   # 转换为标准地方时
        dtdst = dt.dst()
        # 如果 dtdst 为 None 则引发 ValueError
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

在以下 tzinfo_examples.py 文件中有一些*tzinfo*类的例子：

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# 一个考虑系统平台地方时处理方式的类。
#  （可能导致在 UTC 时差和/或夏令时规则
#  曾经发生过变化的时区的某些历史时间
#  出现错误的值。）
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # 删除时间折叠
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
```

(续下页)

(接上页)

```
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# 针对美国各大时区当前夏令时规则的完整实现。

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US 夏令时规则
#
# 这是一个有关美国夏令时起止时间的有所简化的规则集
# （即在某些情况下可能出错。）要获取最新的完整夏令时
# 规则和时区定义集，请访问 Olson 数据库（或尝试 pytz）：
# http://www.twinsun.com/tz/tz-link.htm
# https://sourceforge.net/projects/pytz/（更新可能不及时）
#
# 在美国，从 2007 年起，夏令时开始于 3 月的第二个星期天
# 上午 2 时（标准时），即从 3 月 8 日起的第一个星期天。
DSTSTART_2007 = datetime(1, 3, 8, 2)
# 结束于 11 月的第一天星期天上午 2 时（夏令时）。
DSTEND_2007 = datetime(1, 11, 1, 2)
# 从 1987 到 2006 年，夏令时开始于 4 月的第一个星期天
# 上午 2 时（标准时），结束于 10 月的最后一个星期天上午 2 时
# （夏令时），即从 10 月 25 日起的第一个星期天。
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# 从 1967 到 1986 年，夏令时开始于 4 月的最后一个星期天
# 上午 2 时（标准时），（即从 4 月 24 日起的第一个星期天）
# 结束于 10 月的最后一个星期天上午 2 时（夏令时），
# 即从 10 月 25 日起的第一个星期天。
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006
```

(续下页)

(接上页)

```
def us_dst_range(year):
    # 找到美国夏令时的起止时间。 针对 1967 之前的年份，
    # 将为非夏令时返回 start = end。
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # 对于某种或两种情况，可以考虑在此设置异常。
            # 具体取决于你想如何处理它们。
            # 默认的 fromutc() 实现 (由默认的 astimezone() 实现调用)
            # 将传入一个 dt.tzinfo 为 self 的日期时间。
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # 无法将简单型与感知型对象进行比较，
        # 因此先从 dt 移除时区。
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # 夏令时已生效。
            return HOUR
        if end - HOUR <= dt < end:
            # 折叠 (有歧义的小时值)：使用 dt.fold 来消除歧义。
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # 空档 (不存在的小时值)：逆向折叠规则。
            return HOUR if dt.fold else ZERO
        # 夏令时已结束。
        return ZERO
```

(续下页)

(接上页)

```
    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # 重复的小时值
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # 标准时
            return std_time
        if start <= std_time < end - HOUR:
            # 夏令时
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

请注意同时负责标准时和夏令时的*tzinfo*子类在每年两次的 DST 转换点上会出现不可避免的微妙问题。具体而言，考虑美国东部时区 (UTC -0500)，它的 EDT 从三月的第二个星期天 1:59 (EST) 之后一分钟开始，并在十一月的第一天星期天 1:59 (EDT) 之后一分钟结束:

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

当 DST 开始时（即"start" 行），本地时钟从 1:59 跳到 3:00。形式为 2:MM 的时间值在那一天是没有意义的，因此在 DST 开始那一天 `astimezone(Eastern)` 不会输出包含 `hour == 2` 的结果。例如，在 2016 年春季时钟向前调整时，我们得到:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

当 DST 结束时（见"end" 行），会有更糟糕的潜在问题：本地时间值中有一个小时是不可能没有歧义的：夏令时的最后一小时。即以北美东部时间表示当天夏令时结束时的形式为 5:MM UTC 的时间。本地时钟从 1:59（夏令时）再次跳回到 1:00（标准时）。形式为 1:MM 的本地时间就是有歧义的。此时*astimezone()*是通过将两个相邻的 UTC 小时映射到两个相同的本地小时来模仿本地时钟的行为。在这个北美东部时间的示例中，形式为 5:MM 和 6:MM 的 UTC 时间在转换为北美东部时间时都将被映射到 1:MM，但前一个时间会将*fold*属性设为 0 而后一个时间会将其设为 1。例如，在 2016 年秋季时钟往回调整时，我们得到:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

请注意不同的*datetime* 实例仅通过*fold* 属性值来加以区分，它们在比较时会被视为相等。

不允许时间显示存在歧义的应用程序需要显式地检查*fold* 属性的值，或者避免使用混合式的*tzinfo* 子类；当使用*timezone* 或者任何其他固定差值的*tzinfo* 子类（例如仅表示 EST (固定差值 -5 小时)，或仅表示 EDT (固定差值 -4 小时) 的类时是不会有歧义的)。

参见

zoneinfo
datetime 模块有一个基本*timezone* 类（用来处理任意与 UTC 的固定时差）及其*timezone.utc* 属性（UTC timezone 实例)。

zoneinfo 为 Python 带来了 *IANA 时区数据库*（也被称为 Olson 数据库)，推荐使用它。

IANA 时区数据库
该时区数据库 (通常称为 tz, tzdata 或 zoneinfo) 包含大量代码和数据用来表示全球许多有代表性的地点的本地时间的历史信息。它会定期进行更新以反映各政治实体对时区边界、UTC 差值和夏令时规则的更改。

8.1.9 timezone 对象

timezone 类是*tzinfo* 的子类，它的每个实例都代表一个以与 UTC 的固定时差来定义的时区。

此类的对象不可被用于代表某些特殊地点的时区信息，这些地点在一年的不同日期会使用不同的时差，或是在历史上对民用时间进行过调整。

class datetime.**timezone**(*offset*, *name=None*)

offset 参数必须指定为一个*timedelta* 对象，表示本地时间与 UTC 的时差。它必须严格限制于 -timedelta(hours=24) 和 timedelta(hours=24) 之间，否则会引发*ValueError*。

name 参数是可选的。如果指定则必须为一个字符串，它将被用作*datetime.tzname()* 方法的返回值。

Added in version 3.2.

在 3.7 版本发生变更: UTC 时差不再限制为一个整数分钟值。

timezone.**utcoffset**(*dt*)

返回当*timezone* 实例被构造时指定的固定值。

dt 参数会被忽略。返回值是一个*timedelta* 实例，其值等于本地时间与 UTC 之间的时差。

在 3.7 版本发生变更: UTC 时差不再限制为一个整数分钟值。

timezone.**tzname**(*dt*)

返回当*timezone* 实例被构造时指定的固定值。

如果没有在构造器中提供 *name*，则 tzname(dt) 所返回的名称将根据 offset 值按以下规则生成。如果 *offset* 为 timedelta(0)，则名称为 “UTC”，否则为字符串 UTC±HH:MM，其中 ± 为 offset 的正负符号，HH 和 MM 分别为表示 offset.hours 和 offset.minutes 的两个数码。

在 3.6 版本发生变更：由 `offset=timedelta(0)` 生成的名称现在是简单的 `'UTC'`，而不是 `'UTC+00:00'`。

`timezone.`**`dst`**`(`*dt*`)`

总是返回 `None`。

`timezone.`**`fromutc`**`(`*dt*`)`

返回 `dt + offset`。*dt* 参数必须为一个感知型*datetime* 实例，其中 `tzinfo` 值设为 `self`。

类属性：

`timezone.`**`utc`**

UTC 时区，`timezone(timedelta(0))`。

8.1.10 `strftime()` 和 `strptime()` 的行为

date, *datetime* 和*time* 对象都支持 `strftime(format)` 方法，可用来创建由一个显式格式字符串所控制的表示时间的字符串。

相反地，*datetime.strptime()* 类会根据表示日期和时间的字符串和相应的格式字符串来创建一个*datetime* 对象。

下表提供了*strftime()* 与*strptime()* 的高层级比较：

	strftime	strptime
用法	根据给定的格式将对象转换为字符串	将字符串解析为给定相应格式的*datetime* 对象
方法类型	实例方法	类方法
方法	*date*; *datetime*; *time*	*datetime*
签名	`strftime(format)`	`strptime(date_string, format)`

`strftime()` 和 `strptime()` 格式码

这些方法接受可被用于解析和格式化日期的格式代码：

```
>>> datetime.strptime('31/01/22 23:59:59.999999',
...                   '%d/%m/%y %H:%M:%S.%f')
datetime.datetime(2022, 1, 31, 23, 59, 59, 999999)
>>> _.strftime('%a %d %b %Y, %I:%M%p')
'Mon 31 Jan 2022, 11:59PM'
```

以下列表显示了 1989 版 C 标准所要求的全部格式代码，它们在带有标准 C 实现的所有平台上均可用。

指示符	含意	示例	备注
%a	当地工作日的缩写。	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	本地化的星期中每日的完整名称。	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	以十进制数显示的工作日，其中 0 表示星期日，6 表示星期六。	0, 1, ..., 6	
%d	补零后，以十进制数显示的月份中的一天。	01, 02, ..., 31	(9)
%b	当地月份的缩写。	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	本地化的月份全名。	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%m	补零后，以十进制数显示的月份。	01, 02, ..., 12	(9)
%y	补零后，以十进制数表示的，不带世纪的年份。	00, 01, ..., 99	(9)
%Y	十进制数表示的带世纪的年份。	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%H	以补零后的十进制数表示的小时（24 小时制）。	00, 01, ..., 23	(9)
%I	以补零后的十进制数表示的小时（12 小时制）。	01, 02, ..., 12	(9)
%p	本地化的 AM 或 PM 。	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	补零后，以十进制数显示的分钟。	00, 01, ..., 59	(9)
%S	补零后，以十进制数显示的秒。	00, 01, ..., 59	(4), (9)
%f	微秒作为一个十进制数，零填充到 6 位。	000000, 000001, ..., 999999	(5)
%z	UTC 偏移量，格式为 ±HHMM[SS[.ffffff]]（如果是简单型对象则为空字符串）。	(空), +0000, -0400, +1030, +063415, -030712.345216	(6)
%Z	时区名称（如果对象为简单型则为空字符串）。	(空), UTC, GMT	(6)
%j	以补零后的十进制数表示的一年中的日序号。	001, 002, ..., 366	(9)
%U	以补零后的十进制数表示的一年中的周序号	00, 01, ..., 53	(7), (9)

为了方便起见，还包括了 C89 标准不需要的其他一些指示符。这些参数都对应于 ISO 8601 日期值。

指示符	含意	示例	备注
%G	带有世纪的 ISO 8601 年份，表示包含大部分 ISO 星期 (%V) 的年份。	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
%u	以十进制数显示的 ISO 8601 星期中的日序号，其中 1 表示星期一。	1, 2, ..., 7	
%V	以十进制数显示的 ISO 8601 星期，以星期一作为每周的第一天。第 01 周为包含 1 月 4 日的星期。	01, 02, ..., 53	(8), (9)
%:z	±HH:MM[:SS[.ffffff]] 形式的 UTC 偏移量（如果是简单型对象则为空字符串）。	(空), +00:00, -04:00, +10:30, +06:34:15, -03:07:12.345216	(6)

这些代码可能不是在所有平台上都可与*strftime()* 方法配合使用。ISO 8601 年份和 ISO 8601 星期指示符并不能与上面的年份和星期序号指示符相互替代。调用*strptime()* 时传入不完整或有歧义的 ISO 8601 指示符将引发*ValueError*。

对完整格式代码集的支持在不同平台上有所差异，因为 Python 要调用所在平台的 C 库的 strftime() 函数，而不同平台的差异是很常见的。要查看你所用平台所支持的完整格式代码集，请参阅 *strftime(3)* 文档。不同的平台在处理不支持的格式说明符方面也有差异。

Added in version 3.6: 增加了 %G, %u 和 %V。

Added in version 3.12: 增加了 %:z。

技术细节

总体而言，d.strftime(fmt) 类似于*time* 模块的 time.strftime(fmt, d.timetuple()) 但是并非所有对象都支持*timetuple()* 方法。

对于*datetime.strptime()* 类方法，默认值为 1900-01-01T00:00:00.000: 任何未在格式字符串中指定的部分都将从默认值中获取。[4]

使用 datetime.strptime(date_string, format) 等价于:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

除非格式中包含秒以下的部分或时区差值信息，它们在 datetime.strptime 中受支持但会被 time.strptime 所丢弃。

对于*time* 对象，年、月、日的格式代码不应被使用，因为 time 对象没有这些值。如果它们仍被使用，则年份将被替换为 1900 而月和日将被替换为 1。

对于*date* 对象，时、分、秒和微秒的格式代码不应被使用，因为*date* 对象没有这些值。如果它们仍被使用，则都将被替换为 0。

出于相同的原因，对于包含当前区域设置字符集所无法表示的 Unicode 码位的格式字符串的处理方式也取决于具体平台。在某些平台上这样的码位会不加修改地原样输出，而在其他平台上 strftime 则可能引发*UnicodeError* 或只返回一个空字符串。

注释:

(1) 因为该格式依赖于当前语言区域，所以在假定输出值时应当仔细考虑。字段顺序可能会有变化（例如”month/day/year” 和”day/month/year”），并且输出还可能包含非 ASCII 字符。

(2) *strptime()* 方法能够解析整个 [1, 9999] 范围内的年份，但 < 1000 的年份必须加零填充为 4 位数字宽度。

在 3.2 版本发生变更: 在之前的版本中，*strftime()* 方法只限用于 >= 1900 的年份。

在 3.3 版本发生变更: 在 3.2 版中，*strftime()* 方法只限用于 >= 1000 的年份。

[4] 传入 datetime.strptime('Feb 29', '%b %d') 将导致错误，因为 1900 不是闰年。

(3) 当与*strptime()* 方法一起使用时，如果使用 %I 指示符来解析时，则 %p 指示符只会影响输出时字段。

(4) 与*time* 模块不同的是，datetime 模块不支持闰秒。

(5) 当与*strptime()* 方法一起使用时，%f 指示符可接受一至六个数码及左边的零填充。%f 是对 C 标准中格式字符集的扩展（但单独在 datetime 对象中实现，因此它总是可用）。

(6) 对于简单型对象，%z, %:z 和 %Z 格式代码会被替换为空字符串。

对于一个感知型对象而言：

%z

utcoffset() 会被转换为 ±HHMM[SS[.ffffff]] 形式的字符串，其中 HH 为给出 UTC 时差的小时部分的 2 位数码字符串，MM 为给出 UTC 时差的分钟部分的 2 位数码字符串，SS 为给出 UTC 时差的秒部分的 2 位数码字符串，而 ffffff 则为给出 UTC 时差的微秒部分的 6 位数码字符串。当时差为整数秒时 ffffff 部分将被省略，而当时差为整数分钟时 ffffff 和 SS 部分都将被省略。举例来说，如果*utcoffset()* 返回 timedelta(hours=-3, minutes=-30)，则 %z 会被替换为字符串 '-0330'。

在 3.7 版本发生变更: UTC 时差不再限制为一个整数分钟值。

在 3.7 版本发生变更: 当向*strptime()* 方法提供 %z 指示符时，UTC 差值可以在时、分和秒之间使用冒号作为分隔符。例如，'+01:00:00' 将被解读为一小时的差值。此外，提供 'Z' 就相当于 '+00:00'。

%:z

行为与 %z 相同，但在时，分和秒之间有冒号分隔符。

%Z

在*strftime()* 中，如果*tzname()* 返回 None 则 %Z 会被替换为一个空字符串；在其他情况下 %Z 会被替换为该返回值，它必须为一个字符串。

strptime() 仅接受特定的 %Z 值：

1. 你的机器的区域设置可以是 time.tzname 中的任何值
2. 硬编码的值 UTC 和 GMT

这样生活在日本的人可用的值为 JST, UTC 和 GMT，但可能没有 EST。它将引发 ValueError 表示无效的值。

在 3.2 版本发生变更: 当提供 %z 指示符给*strptime()* 方法时，将产生一个感知型*datetime* 对象。结果的 tzinfo 将被设为一个*timezone* 实例。

(7) 当与*strptime()* 方法一起使用时，%U 和 %W 仅用于指定了星期值和日历年份 (%Y) 的计算。

(8) 类似于 %U 和 %W，%V 仅用于在*strptime()* 格式字符串中指定了星期值和 ISO 年份 (%G) 的计算。还要注意 %G 和 %Y 是不可互换的。

(9) 当与*strptime()* 方法一起使用时，前导的零在格式 %d, %m, %H, %I, %M, %S, %j, %U, %W 和 %V 中是可选的。格式 %y 则要求有前导的零。

(10) 当使用*strptime()* 解析月份和日期时，始终在格式字符串中包括年份。如果你需要解析的值缺少年份，则添加一个显式的占位用闰年。否则当你的代码遇到一个闰日时将引发异常因为解析器所使用的默认年份不是一个闰年。用户会每隔四年碰到这个程序错误...

```
>>> month_day = "02/29"
>>> datetime.strptime(f"{month_day};1984", "%m/%d;%Y")  # 没有闰年错误。
datetime.datetime(1984, 2, 29, 0, 0)
```

Deprecated since version 3.13, will be removed in version 3.15: 使用包含不带年份的月份日期的格式字符串的*strptime()* 调用现在将引发*DeprecationWarning*。在 3.15 或更新的版本中我们可能将其修改为引发错误或将默认年从修改为一年闰年。参见 gh-70647。

备注

8.2 zoneinfo --- IANA 时区支持

Added in version 3.9.

源代码: Lib/zoneinfo

---

zoneinfo 模块根据 **PEP 615** 中的原始规范说明提供了一个具体的时区实现来支持 IANA 时区数据库。在默认情况下，*zoneinfo* 会在可能的情况下使用系统的时区数据；如果系统时区数据不可用，该库将回退为使用 PyPI 上提供的 tzdata 第一方包。

参见

模块: *datetime*
提供*time* 和*datetime* 类型，*ZoneInfo* 类被设计为可配合这两个类型使用。

包 tzdata
由 CPython 核心开发者维护以通过 PyPI 提供时区数据的第一方包。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

8.2.1 使用 ZoneInfo

ZoneInfo 是*datetime.tzinfo* 抽象基类的具体实现，其目标是通过构造器、*datetime.replace* 方法或*datetime.astimezone* 来与 `tzinfo` 建立关联:

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

以此方式构造的日期时间对象可兼容日期时间运算并可在无需进一步干预的情况下处理夏令时转换:

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

这些时区还支持在 **PEP 495** 中引入的*fold*。在可能导致时间歧义的时差转换中（例如夏令时到标准时的转换），当 `fold=0` 时会使用转换*之前*的时差，而当 `fold=1` 时则使用转换*之后*的时差，例如:

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

当执行来自另一时区的转换时，fold 将被设置为正确的值:

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # 在 PDT -> PST 转换之前
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # 在 PDT -> PST 转换之后
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

8.2.2 数据源

zoneinfo 模块不直接提供时区数据，而是在可能的情况下从系统时区数据库或使用 PyPI 上的第一方包 tzdata 来获取时区信息。某些系统，特别是 Windows 系统也包括在内，并没有可用的 IANA 数据库，因此对于要保证获取时区信息的跨平台兼容性的项目，推荐针对 tzdata 声明依赖。如果系统数据和 tzdata 均不可用，则所有对*ZoneInfo* 的调用都将引发*ZoneInfoNotFoundError*。

配置数据源

当 ZoneInfo(key) 被调用时，此构造器首先会在*TZPATH* 所指定的目录下搜索匹配 key 的文件，失败时则会在 tzdata 包中查找匹配。此行为可通过三种方式来配置:

1. 默认的*TZPATH* 未通过其他方式指定时可在编译时 进行配置。
2. *TZPATH* 可使用环境变量 进行配置。
3. 在运行时，搜索路径可使用*reset_tzpath()* 函数来修改。

编译时配置

默认的*TZPATH* 包括一些时区数据库的通用部署位置（Windows 除外，该系统没有时区数据的“通用”位置）。在 POSIX 系统中，下游分发者和从源码编译 Python 的开发者知道系统时区数据部署位置，它们可以通过指定编译时选项 TZPATH (或者更常见的是通过 配置旗标 --with-tzpath) 来改变默认的时区路径，该选项应当是一个由*os.pathsep* 分隔的字符串。

在所有平台上，配置值会在*sysconfig.get_config_var()* 中以 TZPATH 键的形式提供。

环境配置

当初始化*TZPATH* 时（在导入时或不带参数调用*reset_tzpath()* 时），zoneinfo 模块将使用环境变量 PYTHONTZPATH，如果变量存在则会设置搜索路径。

PYTHONTZPATH

这是一个以*os.pathsep* 分隔的字符串，其中包含要使用的时区搜索路径。它必须仅由绝对路径而非相对路径组成。在 PYTHONTZPATH 中指定的相对路径部分将不会被使用，但在其他情况下当指定相对路径时的行为该实现是有定义的；CPython 将引发*InvalidTZPathWarning*，而其他实现可自由地忽略错误部分或是引发异常。

要设置让系统忽略系统数据并改用 tzdata 包，请设置 PYTHONTZPATH=""。

运行时配置

TZ 搜索路径也可在运行时使用*reset_tzpath()* 函数来配置。通常并不建议如此操作，不过在需要使用指定时区路径（或者需要禁止访问系统时区）的测试函数中使用它则是合理的。

8.2.3 ZoneInfo 类

class zoneinfo.**ZoneInfo**(*key*)

一个具体的*datetime.tzinfo* 子类，它代表一个由字符串 key 所指定的 IANA 时区。对主构造器的调用将总是返回可进行标识比较的对象；但是另一种方式，对所有的 key 值通过*ZoneInfo.clear_cache()* 禁止缓存失效，对以下断言将总是为真值:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

key 必须采用相对的标准化 POSIX 路径的形式，其中没有对上一层级的引用。如果传入了不合要求的键则构造器将引发*ValueError*。

如果没有找到匹配 key 的文件，构造器将引发*ZoneInfoNotFoundError*。

ZoneInfo 类具有两个替代构造器:

classmethod ZoneInfo.**from_file**(*fobj*, */*, *key=None*)

基于一个返回字节串的文件型对象（例如一个以二进制模式打开的文件或是一个*io.BytesIO* 对象）构造 ZoneInfo 对象。不同于主构造器，此构造器总是会构造一个新对象。

key 形参设置时区名称以供 __str__() 和 __repr__() 使用。

由此构造器创建的对象不可被封存 (参见*pickling*)。

classmethod ZoneInfo.**no_cache**(*key*)

一个绕过构造器缓存的替代构造器。它与主构造器很相似，但每次调用都会返回一个新对象。此构造器在进行测试或演示时最为适用，但它也可以被用来创建具有不同缓存失效策略的系统。

由此构造器创建的对象在被解封时也会绕过反序列化进程的缓存。

> **小心**
>
> 使用此构造器可以会以令人惊讶的方式改变日期时间对象的语义，只有在你确定你的需求时才使用它。

也可以使用以下的类方法:

classmethod ZoneInfo.**clear_cache**(**, only_keys=None*)

一个可在 ZoneInfo 类上禁用缓存的方法。如果不传入参数，则会禁用所有缓存并且下次对每个键调用主构造器将返回一个新实例。

如果将一个键名称的可迭代对象传给 only_keys 形参，则将只有指定的键会被从缓存中移除。传给 only_keys 但在缓存中找不到的键会被忽略。

> **警告**
>
> 唤起此函数可能会以令人惊讶的方式改变使用 ZoneInfo 的日期时间对象的语义；这会修改模块的状态并因此可能产生大范围的影响。你只有在确定有必要时才可以使用它。

该类具有一个属性:

ZoneInfo.**key**

这是一个只读的*attribute*，它返回传给构造器的 key 的值，该值应为一个 IANA 时区数据库的查找键 (例如 America/New_York, Europe/Paris 或 Asia/Tokyo)。

对于不指定 key 形参而是基于文件构造时区，该属性将设为 None。

> **备注**
>
> 尽管将这些信息暴露给最终用户是一种比较普通的做法，但是这些值被设计作为代表相关时区的主键而不一定是面向用户的元素。CLDR (Unicode 通用区域数据存储库) 之类的项目可被用来根据这些键获取更为用户友好的字符串。

字符串表示

当在*ZoneInfo* 对象上调用*str* 时返回的字符串表示默认会使用*ZoneInfo.key* 属性（参见该属性文档中的用法注释）：

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

对于基于文件而非指定 `key` 形参所构建的对象，`str` 会回退为调用*repr()*。`ZoneInfo` 的 `repr` 是由具体实现定义的并且不一定会在不同版本间保持稳定，但它保证不会是一个有效的 `ZoneInfo` 键。

封存序列化

`ZoneInfo` 对象的序列化是基于键的，而不是序列化所有过渡数据，并且基于文件构造的 `ZoneInfo` 对象（即使是指定了 `key` 值的对象）不能被封存。

`ZoneInfo` 文件的行为取决于它的构造方式:

1. `ZoneInfo(key)`: 当使用主构造器构造时，会基于键序列化一个 `ZoneInfo` 对象，而当反序列化时，反序列化过程会使用主构造器，因此预期它们与其他对同一时区的引用会是同一对象。例如，如果 `europe_berlin_pkl` 是一个包含基于 `ZoneInfo("Europe/Berlin")` 构建的封存数据的字符串，你可以预期出现以下的行为:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. `ZoneInfo.no_cache(key)`: 当通过绕过缓存的构造器构造时，`ZoneInfo` 对象也会基于键序列化，但当反序列化时，反序列化过程会使用绕过缓存的构造器。如果 `europe_berlin_pkl_nc` 是一个包含基于 `ZoneInfo.no_cache("Europe/Berlin")` 构造的封存数据的字符串，你可以预期出现以下的行为:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl_nc)
   >>> a is b
   False
   ```

3. `ZoneInfo.from_file(fobj, /, key=None)`: 当通过文件构造时，`ZoneInfo` 对象会在封存时引发异常。如果最终用户想要封存通过文件构造的 `ZoneInfo`，则推荐他们使用包装类型或自定义序列化函数：或者基于键序列化，或者存储文件对象的内容并将其序列化。

该序列化方法要求所需键的时区数据在序列化和反序列化中均可用，类似于在序列化和反序列化环境中都预期存在对类和函数的引用的方式。这还意味着在具有不同时区数据版本的环境中当解封被封存的 `ZoneInfo` 时并不会保证结果的一致性。

8.2.4 函数

zoneinfo.**available_timezones**()

获取一个包含可用 IANA 时区的在时区路径的任何位置均可用的全部有效键的集合。每次调用该函数时都会重新计算。

此函数仅包括规范时区名称而不包括“特殊”时区如位于 posix/ 和 right/ 目录下的时区或 posixrules 时区。

小心

此函数可能会打开大量的文件，因为确定时区路径上某个文件是否为有效时区的最佳方式是读取开头位置的“魔术字符串”。

备注

这些值并不被设计用来对外公开给最终用户；对于面向用户的元素，应用程序应当使用 CLDR (Unicode 通用区域数据存储库) 之类来获取更为用户友好的字符串。另请参阅*ZoneInfo.key* 中的提示性说明。

zoneinfo.**reset_tzpath**(*to=None*)

设置或重置模块的时区搜索路径 (*TZPATH*)。当不带参数调用时，*TZPATH* 会被设为默认值。

调用 reset_tzpath 将不会使*ZoneInfo* 缓存失效，因而在缓存未命中的情况下对主 ZoneInfo 构造器的调用将只使用新的 TZPATH。

to 形参必须是由字符串或*os.PathLike* 组成的*sequence* 或而不是字符串，它们必须都是绝对路径。如果所传入的不是绝对路径则将引发*ValueError*。

8.2.5 全局变量

zoneinfo.**TZPATH**

一个表示时区搜索路径的只读序列 -- 当通过键构造 ZoneInfo 时，键会与 TZPATH 中的每个条目进行合并，并使用所找到的第一个文件。

TZPATH 可以只包含绝对路径，绝不包含相对路径，无论它是如何配置的。

zoneinfo.TZPATH 所指向的对象可能随着对*reset_tzpath()* 的调用而改变，因此推荐使用 zoneinfo.TZPATH 而不是从 zoneinfo 导入 TZPATH 或是将 zoneinfo.TZPATH 赋值给一个长期变量。

有关配置时区搜索路径的更多信息，请参阅*配置数据源*。

8.2.6 异常与警告

exception zoneinfo.**ZoneInfoNotFoundError**

当一个*ZoneInfo* 对象的构造由于在系统中找不到指定的键而失败时引发。这是*KeyError* 的一个子类。

exception zoneinfo.**InvalidTZPathWarning**

当 *PYTHONTZPATH* 包含将被过滤掉的无效组件，例如一个相对路径时引发。

8.3 calendar --- 通用日历相关函数

源代码： Lib/calendar.py

这个模块让你可以输出像 Unix **cal** 那样的日历，它还提供了其它与日历相关的实用函数。默认情况下，这些日历把星期一作为一周的第一天，星期天作为一周的最后一天（这是欧洲惯例）。可以使用*setfirstweekday()* 方法设置一周的第一天为星期天 (6) 或者其它任意一天。函数全部接收整数类型的参数用来指定日期。其它相关功能参见*datetime* 和*time* 模块。

在这个模块中定义的函数和类都基于一个理想化的日历——向过去和未来两个方向无限扩展的现行公历。这与 Dershowitz 和 Reingold 的书 “历法计算” 中所有计算的基本日历”proleptic Gregorian” 历的定义相符。0 和负数年份按照 ISO 8601 标准解释：0 年指公元前 1 年，-1 年指公元前 2 年，依此类推。

class calendar.**Calendar**(*firstweekday=0*)

创建一个*Calendar* 对象。*firstweekday* 是一个用来指定每星期第一天的整数。*MONDAY* 是 `0` （默认值），*SUNDAY* 是 `6`。

Calendar 对象提供了一些可用于对日历数据进行格式化的准备的方法。这个类本身不执行任何格式化操作。这部分任务应由子类来完成。

Calendar 实例具有以下方法和属性：

firstweekday

以整数 (0--6) 表示的每星期第一天。

该特征属性也可分别使用*setfirstweekday()* 和*getfirstweekday()* 来设置和读取。

getfirstweekday()

返回一个*int* 表示当前的每周第一天 (0--6)。

相当于读取*firstweekday* 特征属性。

setfirstweekday(*firstweekday*)

将每周第一天设为 *firstweekday*，以*int* 形式传入 (0--6)

相当于设置*firstweekday* 特征属性。

iterweekdays()

返回用于表示一星期中日期序号的迭代器。迭代器的第一个值将与*firstweekday* 特性属性的值相同。

itermonthdates(*year*, *month*)

为 *year* 年 *month* 月 (1-12) 返回一个迭代器。这个迭代器返回当月的所有日期（使用*datetime.date* 对象），日期包含了本月头尾用于组成完整一周的日期。

itermonthdays(*year*, *month*)

为 *year* 年 *month* 月返回一个与*itermonthdates()* 类似的迭代器，但不会受*datetime.date* 范围的限制。返回的为每一天的日期相对于当月 1 日过去的天数。对于不在当月的日期，返回数字 `0`。

itermonthdays2(*year*, *month*)

为 *year* 年 *month* 月返回一个与*itermonthdates()* 类似的迭代器，但不会受*datetime.date* 范围的限制。迭代器中的每一个元素为由日数和代表星期几的数字组成的元组。

itermonthdays3(*year*, *month*)

为 *year* 年 *month* 月返回一个与*itermonthdates()* 类似的迭代器，但不会受*datetime.date* 范围的限制。迭代器的元素为一个由年、月、日组成的元组。

Added in version 3.7.

itermonthdays4(*year*, *month*)

为 *year* 年 *month* 月返回一个与*itermonthdates()* 类似的迭代器，但不会受*datetime.date* 范围的限制。迭代器的元素为一个由年、月、日和代表星期几的数字组成的元组。

Added in version 3.7.

monthdatescalendar(*year*, *month*)

返回 *year* 年 *month* 月的周组成的列表。列表中的每一个周是由七个*datetime.date* 对象组成的列表。

monthdays2calendar(*year*, *month*)

返回 *year* 年 *month* 月的周组成的列表。列表中的每一个周是七个由日数和代表星期几的数字组成的元组的列表。

monthdayscalendar(*year*, *month*)

返回 *year* 年 *month* 月的周组成的列表。列表中的每一个周是由七个日数组成的列表。

yeardatescalendar(*year*, *width=3*)

返回可以用来格式化的指定年月的数据。返回的值是一个列表，列表是月份组成的行。每一行包含了最多 *width* 个月 (默认为 3)。每个月包含了 4 到 6 周，每周包含 1--7 天。每一天使用 *datetime.date* 对象。

yeardays2calendar(*year*, *width=3*)

返回可以用来模式化的指定年月的数据 (与 *yeardatescalendar()* 类似)。周列表的元素是由表示日期的数字和表示星期几的数字组成的元组。不在这个月的日子为 0。

yeardayscalendar(*year*, *width=3*)

返回可以用来模式化的指定年月的数据 (与 *yeardatescalendar()* 类似)。周列表的元素是表示日期的数字。不在这个月的日子为 0。

class calendar.**TextCalendar**(*firstweekday=0*)

可以使用这个类生成纯文本日历。

TextCalendar 实例有以下方法：

formatday(*theday*, *weekday*, *width*)

返回一个代表格式化为指定 *width* 的单独日期的字符串表示形式。如果 *theday* 为 `0`，则返回指定宽度的空格字符串，代表一个空日期。*weekday* 形参未被使用。

formatweek(*theweek*, *w=0*)

返回以不带换行符的字符串表示的单个星期。如果提供了 *w*，它将指定日期列的宽度，日期列将居中对齐。具体内容还依赖于构造器或 *setfirstweekday()* 方法指定每周的星期几为第一天。

formatweekday(*weekday*, *width*)

返回一个代表单个周日期名称的格式化为 *width* 所指定宽度的字符串。*weekday* 形参是表示某个周日期的整数，其中 `0` 为星期一而 `6` 为星期日。

formatweekheader(*width*)

返回一个包含周日期名称标题行的字符串，其中每一列格式化为 *width* 所给定的宽度。这些名称依赖于语言区域设置并将被填充至指定的宽度。

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)

返回指定月的用多行字符串表示的月历。*w* 为日期列的宽度，日期列居中打印。*l* 指定了周与周之间的行距。返回的日历还依赖于构造器或者 *setfirstweekday()* 方法指定的每周的第一天是哪一天。

formatmonthname(*theyear*, *themonth*, *width=0*, *withyear=True*)

返回一个代表月份名称的在以 *width* 指定的宽度内居中的字符串。如果 *withyear* 为 `True`，则会在输出中包括年份。*theyear* 和 *themonth* 形参指定年份和月份以便其名称可以相应地被格式化。

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)

调用 *formatmonth()* 方法并打印返回的月历。

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

返回指定年的用多行字符串表示的 *m* 列年历。可选参数 *w*、*l* 和 *c* 分别表示日期列宽，周的行距，和月与月之间的纵向间隔。同样依赖于构造器或者 *setfirstweekday()* 方法指定的每周的第一天是哪一天。可以生成年历的最早的年是哪一年依赖于使用的平台。

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

调用 *formatyear()* 方法并打印返回的年历。

class calendar.**HTMLCalendar**(*firstweekday=0*)

可以使用这个类生成 HTML 日历。

`HTMLCalendar` 实例有以下方法：

formatmonth(*theyear*, *themonth*, *withyear=True*)

返回一个 HTML 表格作为指定年月的日历。*withyear* 为真，则年份将会包含在表头，否则只显示月份。

formatyear(*theyear*, *width=3*)

返回一个 HTML 表格作为指定年份的日历。*width* (默认为 3) 用于规定每一行显示月份的数量。

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)

返回一个完整的 HTML 页面作为指定年份的日历。*width*(默认为 3) 用于规定每一行显示的月份数量。*css* 为层叠样式表的名字。如果不使用任何层叠样式表，可以使用`None`。*encoding* 为输出页面的编码 (默认为系统的默认编码)。

formatmonthname(*theyear*, *themonth*, *withyear=True*)

将一个月份名称以 HTML 表格行的形式返回。如果 *withyear* 为真值则年份将被包括在行中，否则将只使用月份名称。

`HTMLCalendar` 有以下属性，你可以重写它们来自定义应用日历的样式。

cssclasses

一个对应星期一到星期天的 CSS class 列表。默认列表为

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

可以向每天加入其它样式

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

需要注意的是，列表的长度必须为 7。

cssclass_noday

工作日的 CSS 类在上个月或下个月发生。

Added in version 3.7.

cssclasses_weekday_head

用于标题行中的工作日名称的 CSS 类列表。默认值与*cssclasses* 相同。

Added in version 3.7.

cssclass_month_head

月份的头 CSS 类（由*formatmonthname()* 使用）。默认值为 `"month"`。

Added in version 3.7.

cssclass_month

某个月的月历的 CSS 类（由*formatmonth()* 使用）。默认值为 `"month"`。

Added in version 3.7.

cssclass_year

某年的年历的 CSS 类（由*formatyear()* 使用）。默认值为 `"year"`。

Added in version 3.7.

cssclass_year_head

年历的 · 表头 CSS 类（由*formatyear()* 使用）。默认值为 `"year"`。

Added in version 3.7.

需要注意的是，尽管上面命名的样式类都是单独出现的 (如：`cssclass_month cssclass_noday`)，但我们可以使用空格将样式类列表中的多个元素分隔开，例如：

```
"text-bold text-red"
```

下面是一个如何自定义 HTMLCalendar 的示例

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)

可以向这个*TextCalendar* 的子类的构造器传入一个语言区域名称并将返回指定语言区域下的月份和星期名称。

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)

可以向这个*HTMLCalendar* 的子类的构造器传入一个语言区域名称并将返回指定语言区域下的月份和星期名称。

备注

这两个类的构造器、formatweekday() 和 formatmonthname() 方法会临时将 LC_TIME 语言区域更改为给定的 *locale*。因为当前语言区域是进程级的设置，所以它们不是线程安全的。

这个模块为简单的文本日历提供了下列函数。

calendar.**setfirstweekday**(*weekday*)

设置每一周的开始（0 表示星期一，6 表示星期天）。提供了*MONDAY*、*TUESDAY*、*WEDNESDAY*、*THURSDAY*、*FRIDAY*、*SATURDAY* 和*SUNDAY* 几个常量值作为方便。例如，设置每周的第一天为星期天：

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()

返回当前设置的每星期的第一天的数值。

calendar.**isleap**(*year*)

如果 *year* 是闰年则返回*True*，否则返回*False*。

calendar.**leapdays**(*y1*, *y2*)

返回在范围 *y1* 至 *y2* （不包括 y2）之间的闰年的年数，其中 *y1* 和 *y2* 是年份。

此函数对于跨越世纪初的范围也适用。

calendar.**weekday**(*year*, *month*, *day*)

返回某年（1970 -- ...），某月（1 -- 12），某日（1 -- 31）是星期几（0 是星期一）。

calendar.**weekheader**(*n*)

返回一个包含星期几的缩写名的头。*n* 指定星期几缩写的字符宽度。

calendar.**monthrange**(*year*, *month*)

返回指定 *年份*的指定 *月份*的第一天是星期几和这个月的天数。

calendar.**monthcalendar**(*year*, *month*)

返回表示一个月的日历的矩阵。每一行代表一周；此月份外的日子由零表示。每周从周一开始，除非使用*setfirstweekday()* 改变设置。

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)

打印由*month()* 返回的一个月的日历。

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)

使用*TextCalendar*类的*formatmonth()*返回多行字符串形式的月份日历。

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)

打印由*calendar()*返回的整年的日历。

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)

使用*TextCalendar*类的*formatyear()*返回一个整年的 3 列日历。

calendar.**timegm**(*tuple*)

一个不相关但很好用的函数，它接受一个时间元组例如*time*模块中的*gmtime()*函数的返回并返回相应的 Unix 时间戳值，假定 1970 年开始计数，POSIX 编码。实际上，*time.gmtime()*和*timegm()*是彼此相反的。

*calendar*模块导出以下数据属性：

calendar.**day_name**

在当前语言区域下表示周内日期的序列，其中“星期一”为 0 号日。

```
>>> import calendar
>>> list(calendar.day_name)
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

calendar.**day_abbr**

在当前语言区域下简写表示周内日期的序列，其中“一”为 0 号日。

```
>>> import calendar
>>> list(calendar.day_abbr)
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

calendar.**MONDAY**

calendar.**TUESDAY**

calendar.**WEDNESDAY**

calendar.**THURSDAY**

calendar.**FRIDAY**

calendar.**SATURDAY**

calendar.**SUNDAY**

星期内每日序号的别名，其中`MONDAY`是`0`而`SUNDAY`是`6`。

Added in version 3.12.

class calendar.**Day**

将星期内的每一天定义为整数常量的枚举。该枚举的成员以*MONDAY*至*SUNDAY*的形式导出到模块作用域。

Added in version 3.12.

calendar.**month_name**

在当前语言区域中表示一年中每个月份的序列。这遵循一月的月序号为 1 的通常惯例，所以其长度为 13 且`month_name[0]`为空字符串。

```
>>> import calendar
>>> list(calendar.month_name)
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
↪'September', 'October', 'November', 'December']
```

calendar.**month_abbr**

在当前语言区域下简写表示一年中月份的序列。这遵循一月的月份序号为 1 的通常惯例，所以其长度为 13 且`month_abbr[0]`为空字符串。

```
>>> import calendar
>>> list(calendar.month_abbr)
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
```

calendar.**JANUARY**

calendar.**FEBRUARY**

calendar.**MARCH**

calendar.**APRIL**

calendar.**MAY**

calendar.**JUNE**

calendar.**JULY**

calendar.**AUGUST**

calendar.**SEPTEMBER**

calendar.**OCTOBER**

calendar.**NOVEMBER**

calendar.**DECEMBER**

一年中各个月份的别名，其中 JANUARY 是 1 而 DECEMBER 是 12。

Added in version 3.12.

class calendar.**Month**

将一年中各个月份定义为整数常量的枚举。该枚举的成员以*JANUARY* 至*DECEMBER* 的形式导出到模块作用域。

Added in version 3.12.

calendar 模块定义了以下异常:

exception calendar.**IllegalMonthError**(*month*)

ValueError 的子类，当给定的月份数字超出 1-12 范围（不包括边界值）时引发。

month

无效的月份数字。

exception calendar.**IllegalWeekdayError**(*weekday*)

ValueError 的子类，当给定的星期数字超出 0-6 范围（不包括边界值）时引发。

weekday

无效的星期数字。

参见

模块*datetime*
为日期和时间提供与*time* 模块相似功能的面向对象接口。

模块*time*
底层时间相关函数。

8.3.1 命令行用法

Added in version 2.5.

calendar 模块可以作为脚本从命令行执行以实现交互式地打印日历。

```
python -m calendar [-h] [-L LOCALE] [-e ENCODING] [-t {text,html}]
                   [-w WIDTH] [-l LINES] [-s SPACING] [-m MONTHS] [-c CSS]
                   [-f FIRST_WEEKDAY] [year] [month]
```

例如，打印 2000 年的日历：

```
$ python -m calendar 2000
                                  2000

      January                   February                  March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6            1  2  3  4  5
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       6  7  8  9 10 11 12
10 11 12 13 14 15 16      14 15 16 17 18 19 20      13 14 15 16 17 18 19
17 18 19 20 21 22 23      21 22 23 24 25 26 27      20 21 22 23 24 25 26
24 25 26 27 28 29 30      28 29                     27 28 29 30 31
31

       April                     May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2       1  2  3  4  5  6  7               1  2  3  4
 3  4  5  6  7  8  9       8  9 10 11 12 13 14       5  6  7  8  9 10 11
10 11 12 13 14 15 16      15 16 17 18 19 20 21      12 13 14 15 16 17 18
17 18 19 20 21 22 23      22 23 24 25 26 27 28      19 20 21 22 23 24 25
24 25 26 27 28 29 30      29 30 31                  26 27 28 29 30

        July                    August                  September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6                  1  2  3
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       4  5  6  7  8  9 10
10 11 12 13 14 15 16      14 15 16 17 18 19 20      11 12 13 14 15 16 17
17 18 19 20 21 22 23      21 22 23 24 25 26 27      18 19 20 21 22 23 24
24 25 26 27 28 29 30      28 29 30 31               25 26 27 28 29 30
31

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5                  1  2  3
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       4  5  6  7  8  9 10
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      11 12 13 14 15 16 17
16 17 18 19 20 21 22      20 21 22 23 24 25 26      18 19 20 21 22 23 24
23 24 25 26 27 28 29      27 28 29 30               25 26 27 28 29 30 31
30 31
```

可以接受以下选项：

--help, -h

显示帮助信息并退出。

--locale LOCALE, **-L** LOCALE

月份和星期名称所使用的语言区域。默认为英语。

--encoding ENCODING, **-e** ENCODING

输出所使用的编码格式。如果设置了*--locale* 则*--encoding* 将是必须的。

--type {text,html}, **-t** {text,html}

将日历以文本或 HTML 文档的形式打印到终端。

--first-weekday FIRST_WEEKDAY, **-f** FIRST_WEEKDAY

每个星期的开始星期序号。必须为 0 (星期一) 到 6 (星期日) 之间的数字。默认为 0。

Added in version 3.13.

year

要打印日历的年份。默认为当前年份。

month

指定*year* 中要打印日历的月份。必须是 1 到 12 之间的数字，且只能在文本模式下使用。默认打印全年的日历。

文本模式选项:

--width WIDTH, **-w** WIDTH

以终端的列数表示的日期列宽度。日期将打印在列中央。小于 2 的值将被忽略。默认为 2。

--lines LINES, **-l** LINES

以终端的行数表示的每周的行数。日期将顶端对齐打印。小于 1 的值将被忽略。默认为 1。

--spacing SPACING, **-s** SPACING

列中的月份之间的空格。小于 2 的值将被忽略。默认为 6。

--months MONTHS, **-m** MONTHS

每行打印的月份数。默认为 3。

HTML 模式选项:

--css CSS, **-c** CSS

日历要使用的 CSS 样式表的路径。该路径必须是相对于所生成的 HTML，或是一个绝对 HTTP 或 `file:///` URL。

8.4 `collections` --- 容器数据类型

源代码： Lib/collections/__init__.py

这个模块实现了一些专门化的容器，提供了对 Python 的通用内建容器*dict*、*list*、*set* 和*tuple* 的补充。

namedtuple()	一个工厂函数，用来创建元组的子类，子类的字段是有名称的。
deque	类似列表的容器，但 append 和 pop 在其两端的速度都很快。
ChainMap	类似字典的类，用于创建包含多个映射的单个视图。
Counter	用于计数*hashable* 对象的字典子类
OrderedDict	字典的子类，能记住条目被添加进去的顺序。
defaultdict	字典的子类，通过调用用户指定的工厂函数，为键提供默认值。
UserDict	封装了字典对象，简化了字典子类化
UserList	封装了列表对象，简化了列表子类化
UserString	封装了字符串对象，简化了字符串子类化

8.4.1 `ChainMap` 对象

Added in version 3.3.

ChainMap 类将多个映射迅速地链到一起，这样它们就可以作为一个单元处理。这通常比创建一个新字典再重复地使用*update()* 要快得多。

这个类可以用于模拟嵌套作用域，并且对模版化有用。

class collections.**ChainMap**(**maps*)

一个*ChainMap* 将多个字典或者其他映射组合在一起，创建一个单独的可更新的视图。如果没有指定任何 *maps*，一个空字典会被作为 *maps*。这样，每个新链至少包含一个映射。

底层映射被存储在一个列表中。这个列表是公开的，可以通过 *maps* 属性存取和更新。没有其他的状态。

搜索查询底层映射，直到一个键被找到。不同的是，写，更新和删除只操作第一个映射。

一个*ChainMap* 通过引用合并底层映射。所以，如果一个底层映射更新了，这些更改会反映到*ChainMap* 。

支持所有常用字典方法。另外还有一个 *maps* 属性 (attribute)，一个创建子上下文的方法 (method)，一个存取它们首个映射的属性 (property):

maps

一个可以更新的映射列表。这个列表是按照第一次搜索到最后一次搜索的顺序组织的。它是仅有的存储状态，可以被修改。列表最少包含一个映射。

new_child(*m=None*, ***kwargs*)

返回一个新的*ChainMap*，其中包含一个新的映射，后面跟随当前实例中的所有映射。如果指定了 `m`，它会成为新的映射加在映射列表的前面；如果未指定，则会使用一个空字典，因此调用 `d.new_child()` 就等价于 `ChainMap({}, *d.maps)`。如果指定了任何关键字参数，它们会更新所传入的映射或新的空字典。此方法被用于创建子上下文，它可在不改变任何上级映射的情况下被更新。

在 3.4 版本发生变更: 添加了可选的 `m` 形参。

在 3.10 版本发生变更: 增加了对关键字参数的支持。

parents

属性返回一个新的*ChainMap* 包含所有的当前实例的映射，除了第一个。这样可以在搜索的时候跳过第一个映射。使用的场景类似在*nested scopes* 嵌套作用域中使用 `nonlocal` 关键词。用例也可以类比内建函数*super()* 。一个 `d.parents` 的引用等价于 `ChainMap(*d.maps[1:])` 。

注意，*ChainMap* 的迭代顺序是通过从后往前扫描所有映射来确定的:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

使得顺序与从最后一个映射开始调用一系列*dict.update()* 得到的字典的迭代顺序相同:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

在 3.9 版本发生变更: 增加了对 `|` 和 `|=` 运算符的支持，相关说明见 **PEP 584**。

参见

- MultiContext class 在 Enthought CodeTools package 有支持写映射的选项。
- Django 中用于模板的 Context class 是只读的映射链。它还具有上下文推送和弹出特性，类似于*new_child()* 方法和*parents* 特征属性。
- Nested Contexts recipe 提供了对于写入和其他修改是只应用于链路中第一个映射还是所有映射的选项。
- 一个 极简的只读版 Chainmap.

ChainMap 例子和方法

这一节提供了多个使用链映射的案例。

模拟 Python 内部 lookup 链的例子

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

让用户指定的命令行参数优先于环境变量，优先于默认值的例子

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

用*ChainMap*类模拟嵌套上下文的例子

```
c = ChainMap()          # 创建根上下文
d = c.new_child()       # 创建嵌套的子上下文
e = c.new_child()       # c 的子上下文，独立于 d
e.maps[0]               # 当前上下文字典 -- 类似 Python 的 locals()
e.maps[-1]              # 根上下文 -- 类似 Python 的 globals()
e.parents               # 闭包的上下文链 -- 类似 Python 的 nonlocals

d['x'] = 1              # 在当前上下文中设置值
d['x']                  # 在上下文链中获取第一个键
del d['x']              # 在当前上下文中删除
list(d)                 # 所有嵌套的值
k in d                  # 检查所有嵌套的值
len(d)                  # 嵌套的值的数量
d.items()               # 所有嵌套的条目
dict(d)                 # 展平为一个常规字典
```

*ChainMap*类只更新链中的第一个映射，但 lookup 会搜索整个链。然而，如果需要深度写和删除，也可以很容易的通过定义一个子类来实现它

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # 更新向下两级的现有键
>>> d['snake'] = 'red'           # 添加新键到最高层级的字典
>>> del d['elephant']            # 移除向下一级的现有键
>>> d                            # 显示结果
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.4.2 Counter 对象

一个计数器工具，为的是可以方便快速地计账。例如：

```
>>> # 统计一个列表中各单词的出现次数
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
...
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # 找出《哈姆雷特》中出现次数排前十的单词
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

Counter 是*dict* 的子类，用于计数*hashable* 对象。它是一个多项集，元素存储为字典的键而它们的计数存储为字典的值。计数可以是任何整数，包括零或负的计数值。*Counter* 类与其他语言中的 bag 或 multiset 很相似。

它可以通过计数一个 *iterable* 中的元素来初始化，或用其它 *mapping* (包括 counter) 初始化：

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter 对象的接口类似于字典，不同的是，如果查询的键不在 Counter 中，它会返回一个 0 而不是引发一个*KeyError*：

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

设置一个计数为 0 不会从计数器中移去一个元素。使用 `del` 来删除它：

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

Added in version 3.1.

在 3.7 版本发生变更: 作为*dict* 的子类，*Counter* 继承了记住插入顺序的功能。*Counter* 对象间的数学运算也是保序的。结果首先把左操作数中存在的元素按照它们在左操作数中的顺序排序，后面跟着其它元素，按它们在右操作数中的顺序排序。

Counter 对象在对所有字典可用的方法以外还支持一些附加方法：

elements()

返回一个迭代器，其中每个元素将重复出现计数值所指定次。元素会按首次出现的顺序返回。如果一个元素的计数值小于一，*elements()* 将会忽略它。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])

返回一个列表，其中包含 *n* 个最常见的元素及出现次数，按常见程度由高到低排序。如果 *n* 被省略或为 `None`，*most_common()* 将返回计数器中的 *所有* 元素。计数值相等的元素按首次出现的顺序排序：

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract(*[iterable-or-mapping]*)

减去一个 可迭代对象或 映射对象 (或 counter) 中的元素。类似于*dict.update()* 但是是减去而非替换。输入和输出都可以是 0 或负数。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Added in version 3.2.

total()

计算总计数值。

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

Added in version 3.10.

通常字典方法都可用于*Counter* 对象，除了有两个方法工作方式与字典并不相同。

fromkeys(*iterable*)

这个类方法没有在*Counter* 中实现。

update(*[iterable-or-mapping]*)

加上一个 可迭代对象或 映射对象 (或 counter) 中的元素。类似于*dict.update()* 但是是加上而非替换。另外，可迭代对象应当是一个元素序列，而不是一个 `(key, value)` 对的序列。

计数对象支持相等性、子集和超集关系等富比较运算符: ==, !=, <, <=, >, >=。所有这些检测会将不存在的元素当作计数值为零，因此 `Counter(a=1) == Counter(a=1, b=0)` 将返回真值。

在 3.10 版本发生变更: 增加了富比较运算。

在 3.10 版本发生变更: 在相等性检测中，不存在的元素会被当作计数值为零。在此之前，`Counter(a=3)` 和 `Counter(a=3, b=0)` 会被视为不同。

Counter 对象的常用案例

```
c.total()                       # 所有计数的总和
c.clear()                       # 重置所有计数
list(c)                         # 列出不同的元素
set(c)                          # 转换为集合
dict(c)                         # 转换为常规字典
c.items()                       # 访问 (元素, 计数) 对
Counter(dict(list_of_pairs))    # 转换自 (元素, 计数) 对的列表
c.most_common()[:-n-1:-1]       # n 个最不常见的元素
+c                              # 移除为零和负的计数
```

提供了几种数学运算用来合并*Counter* 对象，产生多集（所有计数值均大于零的 counter）。加减运算通过增加或减少两者间对应元素的计数来合并 counter。交并运算返回对应计数的最小值和最大值。相等和包含运算比较对应的计数。每个运算的参数都可以含有有符号的计数，但输出将排除计数小于等于零的元素。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # 将两个计数器相加:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # 相减 (只保留为正的计数)
```

(续下页)

(接上页)

```
Counter({'a': 2})
>>> c & d                       # 交集:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # 并集:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d                      # 相等:  c[x] == d[x]
False
>>> c <= d                      # 包括:  c[x] <= d[x]
False
```

单目加和减（一元操作符）意思是从空计数器加或者减去。

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

Added in version 3.3: 添加了对一元加，一元减和位置集合操作的支持。

备注

计数器主要是为了表达运行的正的计数而设计；但是，小心不要预先排除负数或者其他类型。为了帮助这些用例，这一节记录了最小范围和类型限制。

- *Counter* 类是一个字典的子类，不限制键和值。值用于表示计数，但你实际上 可以存储任何其他值。
- *most_common()* 方法在值需要排序的时候用。
- 参与原地操作如 `c[key] += 1` 的值的类型只需要支持加和减，所以分数、小数和 decimals 都可以用，也支持负数。*update()* 和*subtract()* 当然也一样，输入和输出都支持 0 和负数。
- 多集方法是专为只会遇到正值的使用情况设计的。输入可以是 0 或负数，但只输出计数为正的值。没有类型限制，但值的类型需支持加、减和比较操作。
- *elements()* 方法要求正整数计数。忽略 0 和负数计数。

参见

- Bag class 在 Smalltalk。
- Wikipedia 链接 Multisets.
- C++ multisets 教程和例子。
- 数学操作和多集合用例，参考 *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19* 。
- 在给定数量和集合元素枚举所有不同的多集合，参考*itertools.combinations_with_replacement()*

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.4.3 deque 对象

class collections.**deque**([*iterable*[, *maxlen*]])

返回一个新的双向队列对象，从左到右初始化 (用方法*append()*)，从 *iterable* （迭代对象) 数据创建。如果 *iterable* 没有指定，新队列为空。

Deque 队列是对栈或 queue 队列的泛化（该名称的发音为"deck"，是"double-ended queue" 的简写形式)。Deque 支持线程安全，高度节省内存地从 deque 的任一端添加和弹出条目，在两个方向上的大致性能均为 *O*(1)。

虽然*list* 对象也支持类似的操作，但它们是针对快速的固定长度的操作进行优化而 `pop(0)` 和 `insert(0, v)` 操作对下层数据表示的大小和位置改变都将产生 *O*(*n*) 的内存移动开销。

如果 *maxlen* 没有指定或者是 `None` ，deques 可以增长到任意长度。否则，deque 就限定到指定最大长度。一旦限定长度的 deque 满了，当新项加入时，同样数量的项就从另一端弹出。限定长度 deque 提供类似 Unix filter `tail` 的功能。它们同样可以用与追踪最近的交换和其他数据池活动。

双向队列 (deque) 对象支持以下方法：

append(*x*)

添加 *x* 到右端。

appendleft(*x*)

添加 *x* 到左端。

clear()

移除所有元素，使其长度为 0.

copy()

创建一份浅拷贝。

Added in version 3.5.

count(*x*)

计算 deque 中元素等于 *x* 的个数。

Added in version 3.2.

extend(*iterable*)

扩展 deque 的右侧，通过添加 iterable 参数中的元素。

extendleft(*iterable*)

扩展 deque 的左侧，通过添加 iterable 参数中的元素。注意，左添加时，在结果中 iterable 参数中的顺序将被反过来添加。

index(*x*[, *start*[, *stop*]])

返回 *x* 在 deque 中的位置（在索引 *start* 之后，索引 *stop* 之前)。返回第一个匹配项，如果未找到则引发*ValueError*。

Added in version 3.5.

insert(*i*, *x*)

在位置 *i* 插入 *x* 。

如果插入会导致一个限长 deque 超出长度 *maxlen* 的话，就引发一个*IndexError*。

Added in version 3.5.

pop()

移去并且返回一个元素，deque 最右侧的那一个。如果没有元素的话，就引发一个*IndexError*。

popleft()

移去并且返回一个元素，deque 最左侧的那一个。如果没有元素的话，就引发*IndexError*。

remove(*value*)

移除找到的第一个 *value*。如果没有的话就引发*ValueError*。

reverse()

将 deque 逆序排列。返回 `None` 。

Added in version 3.2.

rotate(*n=1*)

向右循环移动 *n* 步。如果 *n* 是负数，就向左循环。

如果 deque 不是空的，向右循环移动一步就等价于 `d.appendleft(d.pop())`，向左循环一步就等价于 `d.append(d.popleft())`。

Deque 对象同样提供了一个只读属性:

maxlen

Deque 的最大尺寸，如果没有限定的话就是 `None`。

Added in version 3.1.

在上述操作以外，deque 还支持迭代, 封存, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, 使用 `in` 运算符的成员检测以及下标引用例如通过 `d[0]` 访问首个元素等。索引访问在两端的时间复杂度均为 *O*(1) 但在中间则会低至 *O*(*n*)。对于快速随机访问，请改用列表。

Deque 从版本 3.5 开始支持 `__add__()`, `__mul__()`, 和 `__imul__()`。

示例:

```
>>> from collections import deque
>>> d = deque('ghi')                 # 新建一个包含三项的双端队列
>>> for elem in d:                   # 迭代双端队列的元素
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # 添加一个新条目到右端
>>> d.appendleft('f')                # 添加一个新条目到左端
>>> d                                # 显示双端队列的表示形式
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # 返回并移除最右端的项
'j'
>>> d.popleft()                      # 返回并移除最左端的项
'f'
>>> list(d)                          # 列出双端队列的内容
['g', 'h', 'i']
>>> d[0]                             # 查看最左端的项
'g'
>>> d[-1]                            # 查看最右端的项
'i'

>>> list(reversed(d))                # 反向列出双端队列的内容
['i', 'h', 'g']
>>> 'h' in d                         # 搜索双端队列
True
>>> d.extend('jkl')                  # 一次添加多个元素
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # 向右轮转
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                     # 向左轮转
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))               # 新建一个反向的双端队列
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                        # 清空双端队列
>>> d.pop()                          # 无法从空的双端队列弹出元素
Traceback (most recent call last):
```

(续下页)

(接上页)

```
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')              # extendleft() 将反转输入顺序
>>> d
deque(['c', 'b', 'a'])
```

deque 用法

这一节展示了 deque 的多种用法。

限长 deque 提供了类似 Unix `tail` 过滤功能

```
def tail(filename, n=10):
    '返回文件的最后 n 行'
    with open(filename) as f:
        return deque(f, n)
```

另一个用法是维护一个近期添加元素的序列，通过从右边添加和从左边弹出

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

一个 轮询调度器 可以通过在*deque* 中放入迭代器来实现。值从当前迭代器的位置 0 被取出并暂存 (yield)。如果这个迭代器消耗完毕，就用*popleft()* 将其从对列中移去；否则，就通过*rotate()* 将它移到队列的末尾

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # 移除已耗尽的迭代器。
            iterators.popleft()
```

rotate() 方法提供了一种方式来实现*deque* 切片和删除。例如，一个纯的 Python `del d[n]` 实现依赖于 `rotate()` 来定位要弹出的元素

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

要实现*deque* 切片，使用一个类似的方法，应用*rotate()* 将目标元素放到左边。通过*popleft()* 移去老的条目（entries），通过*extend()* 添加新的条目，然后反向 rotate。这个方法可以最小代价实现命令式的栈操作，诸如 `dup`, `drop`, `swap`, `over`, `pick`, `rot`, 和 `roll` 。

8.4.4 defaultdict 对象

class collections.**defaultdict**(*default_factory=None*, */*[, ...])

返回一个新的类似字典的对象。*defaultdict* 是内置*dict* 类的子类。它重写了一个方法并添加了一个可写的实例变量。其余的功能与*dict* 类相同因而不在此文档中写明。

本对象包含一个名为*default_factory* 的属性，构造时，第一个参数用于为该属性提供初始值，默认为 None。所有其他参数（包括关键字参数）都相当于传递给*dict* 的构造函数。

defaultdict 对象除了支持标准*dict* 的操作，还支持以下方法作为扩展：

__missing__(*key*)

如果*default_factory* 属性为 None，则调用本方法会抛出*KeyError* 异常，附带参数 *key*。

如果*default_factory* 不为 None，则它会被（不带参数地）调用来为 *key* 提供一个默认值，这个值和 *key* 作为一对键值对被插入到字典中，并作为本方法的返回值返回。

如果调用*default_factory* 时抛出了异常，这个异常会原封不动地向外层传递。

当请求的键未找到时本方法会被*dict* 类的 __getitem__() 方法调用；它返回或引发的任何对象都会被 __getitem__() 返回或引发。

请注意除了 __getitem__() 以外*__missing__()* 将 不会被调用以执行任何操作。这意味着*get()* 会像普通字典一样返回 None 作为默认值而不是使用*default_factory*。

defaultdict 对象支持以下实例变量：

default_factory

本属性由*__missing__()* 方法来调用。如果构造对象时提供了第一个参数，则本属性会被初始化成那个参数，如果未提供第一个参数，则本属性为 None。

在 3.9 版本发生变更: 增加了合并 (|) 与更新 (|=) 运算符，相关说明见 **PEP 584**。

defaultdict 例子

使用*list* 作为*default_factory*，很轻松地将（键-值对组成的）序列转换为（键-列表组成的）字典：

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

当每个键首次被遇到时，它还不在映射之中；所以会使用*default_factory* 函数自动创建一个条目，该函数返回一个空的*list*。随后将使用 list.append() 操作将值添加到这个新列表中。当再次遇到该键时，将正常地执行查找并且 list.append() 操作会将另一个值添加到列表中。这个做法相比使用*dict.setdefault()* 的等价做法更简单更快速：

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

设置*default_factory* 为*int*，使*defaultdict* 用于计数（类似其他语言中的 bag 或 multiset）：

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
```

(续下页)

(接上页)

```
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

当一个字母首次遇到时，它会查询失败，则*default_factory* 会调用*int()* 来提供一个整数 0 作为默认值。后续的自增操作建立起对每个字母的计数。

函数*int()* 总是返回 0，这是常数函数的特殊情况。一个更快和灵活的方法是使用 lambda 函数，可以提供任何常量值（不只是 0）：

```
>>> def constant_factory(value):
...     return lambda: value
...
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

设置*default_factory* 为*set* 使*defaultdict* 用于构建 set 集合：

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.4.5 namedtuple() 命名元组的工厂函数

命名元组赋予每个位置一个含义，提供可读性和自文档性。它们可以用于任何普通元组，并添加了通过名字获取值的能力，通过索引值也是可以的。

collections.**namedtuple**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

返回一个新的名为 *typename* 的元组子类。这个新子类将被用于创建具有即可通过索引和迭代又可通过属性查找来访问的字段的元组型对象。这样的子类实例还将具有文档字符串 (包含 *typename* 和 *field_names*) 和以 `name=value` 格式列出元组内容的 `__repr__()` 方法以方便使用。

field_names 是一个像 `['x', 'y']` 一样的字符串序列。另外 *field_names* 可以是一个纯字符串，用空白或逗号分隔开元素名，比如 `'x y'` 或者 `'x, y'`。

任何有效的 Python 标识符都可以作为字段名，除了下划线开头的那些。有效标识符由字母，数字，下划线组成，但首字母不能是数字或下划线，另外不能是关键词*keyword* 比如 *class*, *for*, *return*, *global*, *pass*, 或 *raise*。

如果 *rename* 为真，无效字段名会自动转换成位置名。比如 `['abc', 'def', 'ghi', 'abc']` 转换成 `['abc', '_1', 'ghi', '_3']`，消除关键词 `def` 和重复字段名 `abc`。

defaults 可以为 `None` 或者是一个默认值的*iterable*。如果一个默认值域必须跟其他没有默认值的域在一起出现，*defaults* 就应用到最右边的参数。比如如果域名 `['x', 'y', 'z']` 和默认值 `(1, 2)`，那么 `x` 就必须指定一个参数值，`y` 默认值 `1`，`z` 默认值 `2`。

如果定义了 *module*，则命名元组的 `__module__` 属性将被设为该值。

具名元组实例毋需字典来保存每个实例的不同属性，所以它们轻量，占用的内存和普通元组一样。

要支持封存操作，应当将命名元组类赋值给一个匹配 *typename* 的变量。

在 3.1 版本发生变更: 添加了对 *rename* 的支持。

在 3.6 版本发生变更: *verbose* 和 *rename* 参数成为仅限关键字参数.

在 3.6 版本发生变更: 添加了 *module* 参数。

在 3.7 版本发生变更: 移除了 *verbose* 形参和 `_source` 属性。

在 3.7 版本发生变更: 添加了 *defaults* 形参和*_field_defaults* 属性。

```
>>> # 基本示例
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # 使用位置或关键字参数进行实例化
>>> p[0] + p[1]             # 像普通元组 (11, 22) 一样可索引
33
>>> x, y = p                # 像普通元素一样解包
>>> x, y
(11, 22)
>>> p.x + p.y               # 字段也可按名称访问
33
>>> p                       # 名称=值 风格的易读的 __repr__
Point(x=11, y=22)
```

命名元组尤其有用于赋值*csv* *sqlite3* 模块返回的元组

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

除了继承元组的方法，命名元组还支持三个额外的方法和两个属性。为了防止字段名冲突，方法和属性以下划线开始。

classmethod somenamedtuple.**_make**(*iterable*)

类方法从存在的序列或迭代实例创建一个新实例。

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()

返回一个新的*dict* ，它将字段名称映射到它们对应的值:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

在 3.1 版本发生变更: 返回一个*OrderedDict* 而不是*dict* 。

在 3.8 版本发生变更: 返回一个常规*dict* 而不是*OrderedDict*。因为自 Python 3.7 起，常规字典已经保证有序。如果需要*OrderedDict* 的额外特性，推荐的解决方案是将结果转换为需要的类型: `OrderedDict(nt._asdict())`。

somenamedtuple.**_replace**(***kwargs*)

返回一个新的命名元组实例，并将指定域替换为新的值

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum], timestamp=time.
↪now())
```

泛型函数*copy.replace()* 也支持具名元组。

在 3.13 版本发生变更: 对于无效的关键字参数将引发*TypeError* 而不是*ValueError*。

somenamedtuple.**_fields**

字符串元组列出了字段名。用于提醒和从现有元组创建一个新的命名元组类型。

```
>>> p._fields            # 查看字段名
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**

字典将字段名称映射到默认值。

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

要获取这个名字域的值，使用*getattr()* 函数:

```
>>> getattr(p, 'x')
11
```

转换一个字典到命名元组，使用 ** 两星操作符 (所述如 tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

因为一个命名元组是一个正常的 Python 类，它可以很容易的通过子类更改功能。这里是如何添加一个计算域和定宽输出打印格式:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

上面的子类设置 __slots__ 为一个空元组。通过阻止创建实例字典保持了较低的内存开销。

子类化对于添加和存储新的名字域是无效的。应当通过*_fields* 创建一个新的命名元组来实现它:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

文档字符串可以自定义，通过直接赋值给 __doc__ 属性:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
```

(续下页)

(接上页)

```
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

在 3.5 版本发生变更: 文档字符串属性变成可写。

参见

- 请参阅*typing.NamedTuple*，以获取为命名元组添加类型提示的方法。它还使用 class 关键字提供了一种优雅的符号:

```
class Component(NamedTuple):
    part_number: int
    weight: float
    description: Optional[str] = None
```

- 对于以字典为底层的可变域名，参考*types.SimpleNamespace()*。
- *dataclasses* 模块提供了一个装饰器和一些函数，用于自动将生成的特殊方法添加到用户定义的类中。

8.4.6 OrderedDict 对象

有序词典就像常规词典一样，但有一些与排序操作相关的额外功能。由于内置的*dict*类获得了记住插入顺序的能力（在 Python 3.7 中保证了这种新行为），它们变得不那么重要了。

一些与*dict*的不同仍然存在:

- 常规的*dict*被设计为非常擅长映射操作。跟踪插入顺序是次要的。
- *OrderedDict*旨在擅长重新排序操作。空间效率、迭代速度和更新操作的性能是次要的。
- *OrderedDict*算法能比*dict*更好地处理频繁的重排序操作。如下面的例程所示，这使得它更适用于实现各种 LRU 缓存。
- 对于*OrderedDict*，相等操作检查匹配顺序。

  常规的*dict*可以使用 `p == q and all(k1 == k2 for k1, k2 in zip(p, q))` 进行模拟顺序相等性测试。
- *OrderedDict*的*popitem()*方法具有不同的签名。它接受一个可选参数来指定要弹出哪一项。

  常规的*dict*可以使用 `d.popitem()` 模拟 OrderedDict 的 `od.popitem(last=True)`，其保证会返回最右边（最后）的项。

  常规的*dict*可以通过 `(k := next(iter(d)), d.pop(k))` 来模拟 OrderedDict 的 `od.popitem(last=False)`，它将返回并移除最左边（开头）的条目，如果条目存在的话。
- *OrderedDict*具有一个*move_to_end()*方法以高效地将元素移到任一端点。

  常规的*dict*可以通过 `d[k] = d.pop(k)` 来模拟 OrderedDict 的 `od.move_to_end(k, last=True)`，它将把键及其所关联的值移到最右边（末尾）的位置。

  常规的*dict*没有 OrderedDict 的 `od.move_to_end(k, last=False)` 的高效等价物，它会把键及其所关联的值移到最左边（开头）的位置。
- 在 Python 3.8 之前，*dict*都缺少 `__reversed__()` 方法。

class collections.**OrderedDict**([*items*])

返回一个*dict*子类的实例，它具有专门用于重新排列字典顺序的方法。

Added in version 3.1.

popitem(*last=True*)

有序字典的*popitem()*方法移除并返回一个 (key, value) 键值对。如果 *last* 值为真，则按 LIFO 后进先出的顺序返回键值对，否则就按 FIFO (first-in, first-out) 先进先出的顺序返回键值对。

move_to_end(*key*, *last=True*)

将一个现有的 *key* 移到序字典的任一端。如果 *last* 为真值（默认）则将条目移到右端，或者如果 *last* 为假值则将条目移到开头。如果 *key* 不存在则会引发*KeyError*:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d)
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d)
'bacde'
```

Added in version 3.2.

相对于通常的映射方法，有序字典还另外提供了逆序迭代的支持，通过*reversed()* 。

OrderedDict 对象之间的相等性检测对顺序敏感并且大致等价于 `list(od1.items())==list(od2.items())`。

OrderedDict 对象和其他*Mapping* 对象之间的相等性检测像常规字典那样对顺序不敏感。这允许*OrderedDict* 对象在任何可使用字典的地方被替代。

在 3.5 版本发生变更: *OrderedDict* 的项 (item)，键 (key) 和值 (value) 视图 现在支持逆序迭代，通过*reversed()* 。

在 3.6 版本发生变更: 随着 **PEP 468** 的通过，传给*OrderedDict* 构造器及其*update()* 方法的关键字参数顺序将被保留。

在 3.9 版本发生变更: 增加了合并 (|) 与更新 (|=) 运算符，相关说明见 **PEP 584**。

OrderedDict 例子和用法

创建记住键值 最后插入顺序的有序字典变体很简单。如果新条目覆盖现有条目，则原始插入位置将更改并移至末尾:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

一个*OrderedDict* 对于实现*functools.lru_cache()* 的变体也很有用:

```
from collections import OrderedDict
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()      # { args : (timestamp, result)}
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
```

(续下页)

(接上页)

```
        if len(self.cache) > self.maxsize:
            self.cache.popitem(last=False)
        return result
```

```
class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict()   # { uncached_key : request_count }
        self.cache = OrderedDict()      # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests  # 未缓存请求的最大数量
        self.maxsize = maxsize          # 已存储返回值的最大数量
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(last=False)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(last=False)
        return result
```

8.4.7 UserDict 对象

UserDict 类是用作字典对象的外包装。对这个类的需求已部分由直接创建*dict* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的字典可以作为属性来访问。

class collections.**UserDict**([*initialdata*])

模拟字典的类。这个实例的内容保存在一个常规字典中，它可以通过*UserDict* 实例的*data* 属性来访问。如果提供了 *initialdata*，则*data* 会用其内容来初始化；请注意对 *initialdata* 的引用将不会被保留，以允许它被用于其他目的。

UserDict 实例提供了以下属性作为扩展方法和操作的支持:

data

一个真实的字典，用于保存*UserDict* 类的内容。

8.4.8 UserList 对象

这个类封装了列表对象。它是一个有用的基础类，对于你想自定义的类似列表的类，可以继承和覆盖现有的方法，也可以添加新的方法。这样我们可以对列表添加新的行为。

对这个类的需求已部分由直接创建*list* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的列表可以作为属性来访问。

class collections.**UserList**([*list*])

模拟一个列表。这个实例的内容被保存为一个正常列表，通过*UserList* 的*data* 属性存取。实例内容被初始化为一个 *list* 的 copy，默认为 [] 空列表。*list* 可以是迭代对象，比如一个 Python 列表，或者一个*UserList* 对象。

UserList 提供了以下属性作为可变序列的方法和操作的扩展:

data

一个*list* 对象用于存储*UserList* 的内容。

子类化的要求: *UserList* 的子类需要提供一个构造器，可以无参数调用，或者一个参数调用。返回一个新序列的列表操作需要创建一个实现类的实例。它假定了构造器可以以一个参数进行调用，这个参数是一个序列对象，作为数据源。

如果一个分离的类不希望依照这个需求，所有的特殊方法就必须重写；请参照源代码进行修改。

8.4.9 UserString 对象

UserString 类是用作字符串对象的外包装。对这个类的需求已部分由直接创建*str* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的字符串可以作为属性来访问。

class collections.**UserString**(*seq*)

模拟一个字符串对象。这个实例对象的内容保存为一个正常字符串，通过*UserString* 的*data* 属性存取。实例内容初始化设置为 *seq* 的 copy。*seq* 参数可以是任何可通过内建*str()* 函数转换为字符串的对象。

UserString 提供了以下属性作为字符串方法和操作的额外支持:

data

一个真正的*str* 对象用来存放*UserString* 类的内容。

在 3.5 版本发生变更: 新方法 __getnewargs__, __rmod__, casefold, format_map, isprintable, 和 maketrans。

8.5 collections.abc --- 容器的抽象基类

Added in version 3.3: 该模块曾是*collections* 模块的组成部分。

源代码: Lib/_collections_abc.py

本模块提供了一些*抽象基类*，它们可被用于测试一个类是否提供某个特定的接口；例如，它是否为*hashable* 或是否为*mapping* 等。

一个接口的*issubclass()* 或*isinstance()* 测试采用以下三种方式之一。

1) 新编写的类可以直接继承自某个抽象基类。该类必须提供所面的抽象方法。其他混入方法来自于继承并且可在需要时被重写。其他方法可以被按需添加:

```
class C(Sequence):                      # 直接继承
    def __init__(self): ...             # ABC 所不需要的额外方法
    def __getitem__(self, index):  ...  # 需要的抽象方法
    def __len__(self):  ...             # 需要的抽象方法
    def count(self, value): ...         # 可选覆盖一个混入方法
```

```
>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2) 已有的类和内置类可被注册为 ABC 的“虚拟子类”。这些类应当定义完整的 API 包括所有抽象方法和所有混合方法。这使得用户能依靠*issubclass()* 或*isinstance()* 测试来确定完整接口是否受到支持。此规则的例外情况是那些从 API 的其他部分自动推断出来的方法:

```
class D:                                    # 无继承
    def __init__(self): ...                 # ABC 所不需要的额外方法
    def __getitem__(self, index):  ...      # 抽象方法
    def __len__(self):  ...                 # 抽象方法
    def count(self, value): ...             # 混入方法
    def index(self, value): ...             # 混入方法

Sequence.register(D)                        # 注册而非继承
```

```
>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

在这个例子中，D 类不需要定义 __contains__, __iter__ 和 __reversed__，因为 in 运算符, 迭代逻辑和*reversed()* 函数会自动回退为使用 __getitem__ 和 __len__。

3) 某些简单接口可以根据所需方法是否存在来直接识别 (除非这些方法已被设置为*None*):

```
class E:
    def __iter__(self): ...
    def __next__(self): ...
```

```
>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

复杂的接口不支持最后这种技术手段因为接口并不只是作为方法名称存在。接口指明了方法之间的语义和关系，这些是无法根据特定方法名称的存在推断出来的。例如，知道一个类提供了 __getitem__, __len__ 和 __iter__ 并不足以区分*Sequence* 和*Mapping*。

Added in version 3.9: 这些抽象类现在都支持 []。参见*GenericAlias 类型* 和 **PEP 585**。

8.5.1 容器抽象基类

这个容器模块提供了以下*ABCs*:

抽象基类	继承自	抽象方法	Mixin 方法
Container[1]		__contains__	
Hashable[1]		__hash__	
Iterable[1][2]		__iter__	
Iterator[1]	*Iterable*	__next__	__iter__
Reversible[1]	*Iterable*	__reversed__	
Generator[1]	*Iterator*	send, throw	close, __iter__, __next__
Sized[1]		__len__	
Callable[1]		__call__	
Collection[1]	*Sized*, *Iterable*, *Containe*	__contains__, __iter__, __len__	
Sequence	*Reversib* *Collecti*	__getitem__, __len__	__contains__, __iter__, __reversed__, index, and count
MutableSequence	*Sequence*	__getitem__, __setitem__, __delitem__, __len__, insert	继承了*Sequence* 的方法以及 append, clear, reverse, extend, pop, remove 和 __iadd__
ByteString	*Sequence*	__getitem__, __len__	继承自*Sequence* 的方法
Set	*Collecti*	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __rsub__, __xor__, __rxor__ 和 isdisjoint
MutableSet	*Set*	__contains__, __iter__, __len__, add, discard	继承自*Set* 的方法以及 clear, pop, remove, __ior__, __iand__, __ixor__，和 __isub__
Mapping	*Collecti*	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, and __ne__
MutableMapping	*Mapping*	__getitem__, __setitem__, __delitem__, __iter__, __len__	继承自*Mapping* 的方法以及 pop, popitem, clear, update，和 setdefault
MappingView	*Sized*		__init__, __len__ 和 __repr__
ItemsView	*MappingV* *Set*		__contains__, __iter__
KeysView	*MappingV* *Set*		__contains__, __iter__
ValuesView	*MappingV* *Collecti*		__contains__, __iter__
Awaitable[1]		__await__	
Coroutine[1]	*Awaitable*	send, throw	close
AsyncIterable[1]		__aiter__	
AsyncIterator[1]	*AsyncIte*	__anext__	__aiter__
AsyncGenerator[1]	*AsyncIte*	asend, athrow	aclose, __aiter__, __anext__
Buffer[1]		__buffer__	

[1] 这些 ABC 重写了*__subclasshook__()* 以便支持通过验证所需的方法是否存在并且没有被设为*None* 来测试一个接口。这只适用于简单的接口。更复杂的接口需要注册或者直接子类化。

[2] 检查 isinstance(obj, Iterable) 是否侦测到被注册为*Iterable* 或者具有*__iter__()* 方法的类，但它不能侦测到使用 __getitem__() 方法进行迭代的类。确定一个对象是否为*iterable* 的唯一可靠方式是调用 iter(obj)。

附注

8.5.2 多项集抽象基类 -- 详细描述

class collections.abc.**Container**

提供了 `__contains__()` 方法的抽象基类。

class collections.abc.**Hashable**

提供了 `__hash__()` 方法的抽象基类。

class collections.abc.**Sized**

用于提供 `__len__()` 方法的类的 ABC

class collections.abc.**Callable**

用于提供 `__call__()` 方法的类的 ABC

有关如何在类型标注中使用 `Callable` 的详细信息请参阅标注可调用对象。

class collections.abc.**Iterable**

用于提供*`__iter__()`* 方法的类的 ABC

检查 `isinstance(obj, Iterable)` 是否侦测到被注册为*`Iterable`* 或者具有*`__iter__()`* 方法的类，但它不能侦测到使用 `__getitem__()` 方法进行迭代的类。确定一个对象是否为*iterable* 的唯一可靠方式是调用 `iter(obj)`。

class collections.abc.**Collection**

集合了 Sized 和 Iterable 类的抽象基类。

Added in version 3.6.

class collections.abc.**Iterator**

提供了*`__iter__()`* 和*`__next__()`* 方法的抽象基类。参见*iterator* 的定义。

class collections.abc.**Reversible**

用于同时提供了 `__reversed__()` 方法的可迭代类的 ABC

Added in version 3.6.

class collections.abc.**Generator**

用于实现了 **PEP 342** 中定义的协议的*generator* 类的 ABC，它通过 `send()`, `throw()` 和 `close()` 方法对迭代器 进行了扩展。

有关在类型标注中使用 `Generator` 的详细信息请参阅标注生成器和协程。

Added in version 3.5.

class collections.abc.**Sequence**

class collections.abc.**MutableSequence**

class collections.abc.**ByteString**

只读的与可变的序列 的抽象基类。

实现注意事项：某些混入方法，如*`__iter__()`*, `__reversed__()` 和 `index()`，会重复调用下层的 `__getitem__()` 方法。因此，如果 `__getitem__()` 被实现为常数级访问速度，则混入方法的性能将为线性级；但是，如果下层的方法是线性的（例如链表就是如此），则混入方法的性能将为平方级并可能需要被重写。

在 3.5 版本发生变更: index() 方法支持 *stop* 和 *start* 参数。

Deprecated since version 3.12, will be removed in version 3.14: *`ByteString`* ABC 已被弃用。当用于类型标注时，建议改为并集形式，如 `bytes | bytearray`，或*`collections.abc.Buffer`*。当用作 ABC 时，建议改为*`Sequence`* 或*`collections.abc.Buffer`*。

class collections.abc.**Set**

class collections.abc.**MutableSet**

用于只读和可变集合 的 ABC。

class collections.abc.**Mapping**

class collections.abc.**MutableMapping**

只读的与可变的映射 的抽象基类。

class collections.abc.**MappingView**

class collections.abc.**ItemsView**

class collections.abc.**KeysView**

class collections.abc.**ValuesView**

映射及其键和值的视图 的抽象基类。

class collections.abc.**Awaitable**

针对*awaitable* 对象的 ABC，它可被用于 `await` 表达式。根据惯例所有实现都必须提供 `__await__()` 方法。

协程 对象和*Coroutine* ABC 的实例都是这个 ABC 的实例。

> **备注**
>
> 在 CPython 中，基于生成器的协程 (使用*@types.coroutine* 装饰的生成器) 都是 可等待对象，即使它们没有 `__await__()` 方法。对它们使用 `isinstance(gencoro, Awaitable)` 将返回 `False`。请使用*inspect.isawaitable()* 来检测它们。

Added in version 3.5.

class collections.abc.**Coroutine**

用于*coroutine* 兼容类的 ABC。实现了如下定义在 coroutine-objects 里的方法: `send()`, `throw()` 和 `close()`。根据惯例所有实现都还需要实现 `__await__()`。所有的*Coroutine* 实例同时也是*Awaitable* 的实例。

> **备注**
>
> 在 CPython 中，基于生成器的协程 (使用*@types.coroutine* 装饰的生成器) 都是 可等待对象，即使它们没有 `__await__()` 方法。对它们使用 `isinstance(gencoro, Coroutine)` 将返回 `False`。请使用*inspect.isawaitable()* 来检测它们。

有关在类型标注中使用 `Coroutine` 的详细信息请参阅标注生成器和协程。类型形参的变化和顺序与*Generator* 的相对应。

Added in version 3.5.

class collections.abc.**AsyncIterable**

针对提供了 `__aiter__` 方法的类的 ABC。另请参阅*asynchronous iterable* 的定义。

Added in version 3.5.

class collections.abc.**AsyncIterator**

提供了 `__aiter__` 和 `__anext__` 方法的抽象基类。参见*asynchronous iterator* 的定义。

Added in version 3.5.

class collections.abc.**AsyncGenerator**

针对实现了在 **PEP 525** 和 **PEP 492** 中定义的协议的*asynchronous generator* 类的 ABC。

有关在类型标注中使用 `AsyncGenerator` 的详细信息请参阅标注生成器和协程。

Added in version 3.6.

class collections.abc.**Buffer**

针对提供 __buffer__() 方法的类的 ABC，实现了 缓冲区协议。参见 **PEP 688**。

Added in version 3.12.

8.5.3 例子和配方

ABC 允许我们询问类或实例是否提供特定的功能，例如:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

有些 ABC 还适用于作为混入类，这可以更容易地开发支持容器 API 的类。例如，要写一个支持完整*Set* API 的类，只需要提供三个下层抽象方法: __contains__(), *__iter__()* 和 __len__()。ABC 会提供其余的方法如 __and__() 和*isdisjoint()*:

```
class ListBasedSet(collections.abc.Set):
    ''' 空间重于速度并且不要求集合元素可哈希的
        替代性集合实现。 '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # 自动支持 __and__() 方法
```

当把*Set* 和*MutableSet* 用作混入类时需注意:

(1) 由于某些集合操作会创建新的集合，默认的混入方法需要一种根据*iterable* 创建新实例的方式。类构造器应当具有 ClassName(iterable) 形式的签名。这样它将被重构为一个执行 _from_iterable() 的内部*classmethod*，该方法会调用 cls(iterable) 来产生一个新的集合。如果*Set* 混入类在具有不同构造器签名的类中被使用，你将需要通过一个能根据可迭代对象参数构造新实例的类方法或常规方法来重写 _from_iterable()。

(2) 要重写比较运算(应该是为了提高速度，因为其语义是固定的)，请重新定义 __le__() 和 __ge__()，然后其他运算将自动跟进。

(3) *Set* 混入类提供了一个 _hash() 方法为集合计算哈希值；但是，__hash__() 没有被定义因为并非所有集合都是*hashable* 或不可变对象。要使用混入类为集合添加可哈希性，请同时继承*Set()* 和*Hashable()*，然后定义 __hash__ = Set._hash。

参见

- OrderedSet recipe 是基于*MutableSet* 构建的一个示例。
- 对于抽象基类，参见*abc* 模块和 **PEP 3119**。

8.6 heapq --- 堆队列算法

源码： Lib/heapq.py

这个模块实现了堆队列算法，即优先队列算法。

堆是一种二叉树，其中每个上级节点的值都小于等于它的任意子节点。我们将这一条件称为堆的不变性。

这个实现使用了数组，其中对于所有从 0 开始计数的 *k* 都有 `heap[k] <= heap[2*k+1]` 且 `heap[k] <= heap[2*k+2]`。为了便于比较，不存在的元素将被视为无穷大。堆最有趣的特性在于其最小的元素始终位于根节点 `heap[0]`。

这个 API 与教材的堆算法实现有所不同，具体区别有两方面：(a) 我们使用了从零开始的索引。这使得节点和其孩子节点索引之间的关系不太直观但更加适合，因为 Python 使用从零开始的索引。(b) 我们的 pop 方法返回最小的项而不是最大的项（这在教材中称为“最小堆”；而“最大堆”在教材中更为常见，因为它更适用于原地排序）。

基于这两方面，把堆看作原生的 Python list 也没什么奇怪的：`heap[0]` 表示最小的元素，同时 `heap.sort()` 维护了堆的不变性！

要创建一个堆，可以新建一个空列表 `[]`，或者用函数*heapify()* 把一个非空列表变为堆。

定义了以下函数：

heapq.**heappush**(*heap*, *item*)

将 *item* 的值加入 *heap* 中，保持堆的不变性。

heapq.**heappop**(*heap*)

弹出并返回 *heap* 的最小的元素，保持堆的不变性。如果堆为空，抛出*IndexError* 。使用 `heap[0]` ，可以只访问最小的元素而不弹出它。

heapq.**heappushpop**(*heap*, *item*)

将 *item* 放入堆中，然后弹出并返回 *heap* 的最小元素。该组合操作比先调用*heappush()* 再调用*heappop()* 运行起来更有效率。

heapq.**heapify**(*x*)

将 list *x* 转换成堆，原地，线性时间内。

heapq.**heapreplace**(*heap*, *item*)

弹出并返回 *heap* 中最小的一项，同时推入新的 *item*。堆的大小不变。如果堆为空则引发*IndexError*。

这个单步骤操作比*heappop()* 加*heappush()* 更高效，并且在使用固定大小的堆时更为适宜。pop/push 组合总是会从堆中返回一个元素并将其替换为 *item*。

返回的值可能会比新加入的值大。如果不希望如此，可改用*heappushpop()*。它的 push/pop 组合返回两个值中较小的一个，将较大的留在堆中。

该模块还提供了三个基于堆的通用目的函数。

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)

将多个已排序的输入合并为一个已排序的输出（例如，合并来自多个日志文件的带时间戳的条目）。返回已排序值的*iterator*。

类似于 `sorted(itertools.chain(*iterables))` 但返回一个可迭代对象，不会一次性地将数据全部放入内存，并假定每个输入流都是已排序的（从小到大）。

具有两个可选参数，它们都必须指定为关键字参数。

key 指定带有单个参数的*key function*，用于从每个输入元素中提取比较键。默认值为 `None` (直接比较元素)。

reverse 为一个布尔值。如果设为 `True`，则输入元素将按比较结果逆序进行合并。要达成与 `sorted(itertools.chain(*iterables), reverse=True)` 类似的行为，所有可迭代对象必须是已从大到小排序的。

在 3.5 版本发生变更: 添加了可选的 *key* 和 *reverse* 形参。

heapq.**nlargest**(*n*, *iterable*, *key=None*)

从 *iterable* 所定义的数据集中返回前 *n* 个最大元素组成的列表。如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。等价于: `sorted(iterable, key=key, reverse=True)[:n]`。

heapq.**nsmallest**(*n*, *iterable*, *key=None*)

从 *iterable* 所定义的数据集中返回前 *n* 个最小元素组成的列表。如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。等价于: `sorted(iterable, key=key)[:n]`。

后两个函数在 *n* 值较小时性能最好。对于更大的值，使用*sorted()* 函数会更有效率。此外，当 `n==1` 时，使用内置的*min()* 和*max()* 函数会更有效率。如果需要重复使用这些函数，请考虑将可迭代对象转为真正的堆。

8.6.1 基本示例

堆排序 可以通过将所有值推入堆中然后每次弹出一个最小值项来实现。

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这类似于 `sorted(iterable)`，但与*sorted()* 不同的是这个实现是不稳定的。

堆元素可以为元组。这有利于以下做法——在被跟踪的主记录旁边添一个额外的值（例如任务的优先级）用于互相比较：

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.6.2 优先队列实现说明

优先队列 是堆的常用场合，并且它的实现包含了多个挑战：

- 排序稳定性：如何让两个相同优先级的任务按它们最初被加入队列的顺序返回？
- 如果 priority 相同且 task 之间未定义默认比较顺序，则两个 (priority, task) 元组之间的比较会报错。
- 如果任务优先级发生改变，你该如何将其移至堆中的新位置？
- 或者如果一个挂起的任务需要被删除，你该如何找到它并将其移出队列？

针对前两项挑战的一种解决方案是将条目保存为包含优先级、条目计数和任务对象 3 个元素的列表。条目计数可用来打破平局，这样具有相同优先级的任务将按它们的添加顺序返回。并且由于没有哪两个条目计数是相同的，元组比较将永远不会直接比较两个任务。

两个 task 之间不可比的问题的另一种解决方案是——创建一个忽略 task，只比较 priority 字段的包装器类：

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
```

(续下页)

(接上页)

```
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

其余的挑战主要包括找到挂起的任务并修改其优先级或将其完全移除。找到一个任务可使用一个指向队列中条目的字典来实现。

移除条目或改变其优先级的操作实现起来更为困难，因为它会破坏堆结构不变量。因此，一种可能的解决方案是将条目标记为已移除，再添加一个改变了优先级的新条目:

```
pq = []                         # 由在堆中处理的条目组成的列表
entry_finder = {}               # 从任务到条目的映射
REMOVED = '<removed-task>'      # 已移除任务的占位符
counter = itertools.count()     # 唯一序列计数

def add_task(task, priority=0):
    '新增任务或更新现有任务的优先级'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    '将现有任务标记为已移除。 如未找到则引发 KeyError。'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    '移除并返回最低优先级的任务。 如为空则引发 KeyError。'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.6.3 理论

堆是通过数组来实现的，其中的元素从 0 开始计数，对于所有的 *k* 都有 `a[k] <= a[2*k+1]` 且 `a[k] <= a[2*k+2]`。为了便于比较，不存在的元素被视为无穷大。堆最有趣的特性在于 `a[0]` 总是其中最小的元素。

上面的特殊不变量是用来作为一场锦标赛的高效内存表示。下面的数字是 *k* 而不是 `a[k]`:

```
                               0

             1                                 2

      3              4                5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

在上面的树中，每个 *k* 单元都位于 `2*k+1` 和 `2*k+2` 之上。体育运动中我们经常见到二元锦标赛模式，每个胜者单元都位于另两个单元之上，并且我们可以沿着树形图向下追溯胜者所遇到的所有对手。但是，在许多采用这种锦标赛模式的计算机应用程序中，我们并不需要追溯胜者的历史。为了获得更高的内存利用效率，当一个胜者晋级时，我们会用较低层级的另一条目来替代它，因此规则变为一个单元和它之下的两个单元包含三个不同条目，上方单元“胜过”了两个下方单元。

如果这个堆的不变性始终受到保护，则索引号 0 显然是最终胜出者。移除它并找出“下一个”胜出者的最简单算法形式是将某个输家（让我们假定是上图中的 30 号单元）移至 0 号位，然后将这个新的 0 号沿着树结构下行，不断进行值的交换，直到不变性得到重建。这显然会是树中条目总数的对数。通过迭代所有条目，你将得到一个 $O(n \log n)$ 复杂度的排序。

此排序有一个很好的特性就是你可以在排序进行期间高效地插入新条目，前提是插入的条目不比你最近取出的 0 号元素“更好”。这在模拟上下文时特别有用，在这种情况下树保存的是所有传入事件，“胜出”条件是最小调度时间。当一个事件将其他事件排入执行计划时，它们的调试时间向未来方向延长，这样它们可方便地入堆。因此，堆结构很适宜用来实现调度器，我的 MIDI 音序器就是用的这个:-)。

用于实现调度器的各种结构都得到了充分的研究，堆是非常适宜的一种，因为它们的速度相当快，并且几乎是恒定的，最坏的情况与平均情况没有太大差别。虽然还存在其他总体而言更高效的实现方式，但其最坏的情况却可能非常糟糕。

堆在大磁盘排序中也非常有用。你应该已经了解大规模排序会有多个“运行轮次”（即预排序的序列，其大小通常与 CPU 内存容量相关），随后这些轮次会进入合并通道，轮次合并的组织往往非常巧妙[1]。非常重要的一点是初始排序应产生尽可能长的运行轮次。锦标赛模式是达成此目标的好办法。如果你使用全部有用内存来进行锦标赛，替换和安排恰好适合当前运行轮次的条目，你将可以对于随机输入生成两倍于内存大小的运行轮次，对于模糊排序的输入还会有更好的效果。

另外，如果你输出磁盘上的第 0 个条目并获得一个可能不适合当前锦标赛的输入（因为其值要“胜过”上一个输出值），它无法被放入堆中，因此堆的尺寸将缩小。被释放的内存可以被巧妙地立即重用以逐步构建第二个堆，其增长速度与第一个堆的缩减速度正好相同。当第一个堆完全消失时，你可以切换新堆并启动新的运行轮次。这样做既聪明又高效！

总之，堆是值得了解的有用内存结构。我在一些应用中用到了它们，并且认为保留一个’heap’ 模块是很有意义的。:-)

备注

8.7 `bisect` --- 数组二分算法

源代码： Lib/bisect.py

本模块提供对维护一个已排序列表而无须在每次插入后对该列表重排序的支持。对于具有大量条目需要大量比较运算的长列表，这改进了原来的线性搜索或频繁重排序。

本模块被命名为*bisect* 是因为它使用了基本的二分算法来完成任务。不同于其他搜索特定值的二分算法工具，本模块的函数被设计为定位一个插入点。相应地，这些函数绝不会调用 `__eq__()` 方法来确定是否找到特定值。相反，这些函数只会调用 `__lt__()` 方法并将返回一个数组的值之间的插入点。

定义了以下函数：

`bisect.`**`bisect_left`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

在 *a* 中找到 *x* 合适的插入点以维持有序。参数 *lo* 和 *hi* 可以被用于确定需要考虑的子集；默认情况下整个列表都会被使用。如果 *x* 已经在 *a* 里存在，那么插入点会在已存在元素之前（也就是左边）。如果 *a* 是列表（list）的话，返回值是可以被放在 `list.insert()` 的第一个参数的。

返回的插入点 *ip* 将数组 *a* 分为两个切片使得对于左侧切片 `all(elem < x for elem in a[lo : ip])` 为真值而对于右侧切片 `all(elem >= x for elem in a[ip : hi])` 为真值。

key 指定带有单个参数的*key function* 用来从数组的每个元素中提取比较键。为了支持搜索复杂记录，键函数不会被应用到 *x* 值。

如果 *key* 为 `None`，则将直接比较元素而不调用任何键函数。

在 3.10 版本发生变更: 增加了 *key* 形参。

`bisect.`**`bisect_right`**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

[1] 当前时代的磁盘平衡算法与其说是巧妙，不如说是麻烦，这是由磁盘的寻址能力导致的结果。在无法寻址的设备例如大型磁带机上，情况则相当不同，开发者必须非常聪明地（极为提前地）确保每次磁带转动都尽可能地高效（就是说能够最好地加入到合并“进程”中）。有些磁带甚至能够反向读取，这也被用来避免倒带的耗时。请相信我，真正优秀的磁带机排序看起来是极其壮观的，排序从来都是一门伟大的艺术！:-)

bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

类似于*bisect_left()*，但是返回的插入点是在 *a* 中任何现有条目 *x* 之后（即其右侧）。

返回的插入点 *ip* 将数组 *a* 分为两个切片使得对于左侧切片 `all(elem <= x for elem in a[lo : ip])` 为真值而对于右侧切片 `all(elem > x for elem in a[ip : hi])` 为真值。

在 3.10 版本发生变更: 增加了 *key* 形参。

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

按照已排序顺序将 *x* 插入到 *a* 中。

此函数会先运行*bisect_left()* 来定位一个插入点。然后，它会在 *a* 上运行 `insert()` 方法在适当的位置插入 *x* 以保持排序顺序。

为了支持将记录插入到表中，*key* 函数（如果存在）将被应用到 *x* 用于搜索步骤但不会用于插入步骤。

请记住 *O*(log *n*) 搜索是由缓慢的 *O*(*n*) 插入步骤主导的。

在 3.10 版本发生变更: 增加了 *key* 形参。

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*, *, *key=None*)

类似于*insort_left()*，但是会把 *x* 插入到 *a* 中任何现有条目 *x* 之后。

此函数会先运行*bisect_right()* 来定位一个插入点。然后，它会在 *a* 上运行 `insert()` 方法在适当的位置插入 *x* 以保持排序顺序。

为了支持将记录插入到表中，*key* 函数（如果存在）将被应用到 *x* 用于搜索步骤但不会用于插入步骤。

请记住 *O*(log *n*) 搜索是由缓慢的 *O*(*n*) 插入步骤主导的。

在 3.10 版本发生变更: 增加了 *key* 形参。

8.7.1 性能说明

当使用 *bisect()* 和 *insort()* 编写时间敏感的代码时，请记住以下概念。

- 二分法对于搜索一定范围的值是很高效的。对于定位特定的值，则字典的性能更好。
- *insort()* 函数的时间复杂度为 *O*(*n*) 因为对数时间的搜索步骤被线性时间的插入步骤所主导。
- 这些搜索函数都是无状态的并且会在它们被使用后丢弃键函数的结果。因此，如果在一个循环中使用搜索函数，则键函数可能会在同一个数据元素上被反复调用。如果键函数速度不够快，请考虑用*functools.cache()* 来包装它以避免重复计算。另外，也可以考虑搜索一个预先计算好的键数组来定位插入点（如下面的示例小节所演示的）。

参见

- Sorted Collections 是一个使用 *bisect* 来管理数据的已排序多项集的高性能模块。
- SortedCollection recipe 使用 bisect 构建了一个功能完整的多项集类，拥有直观的搜索方法和对键函数的支持。所有键函数都是预先计算好的以避免在搜索期间对键函数的不必要的调用。

8.7.2 搜索有序列表

上面的*bisect functions* 对于找到插入点是有用的，但在一般的搜索任务中可能会有点尴尬。下面的五个函数展示了如何将其转换为针对有序列表的标准查找函数:

```
def index(a, x):
    '定位恰好等于 x 的最靠左的值'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
```

(续下页)

(接上页)

```
        return i
    raise ValueError

def find_lt(a, x):
    '找到小于 x 的最靠右的值'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    '找到小于等于 x 的最靠右的值'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    '找到大于 x 的最靠左的值'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    '找到大于等于 x 的最靠左的项'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.7.3 例子

bisect() 函数对于数字表查询也是适用的。这个例子使用*bisect()* 根据一组有序的数字划分点来查找考试成绩对应的字母等级: (如) 90 及以上为'A'，80 至 89 为'B'，依此类推:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

bisect() 和*insort()* 对于列表和元组也是适用的。*key* 参数可以提取用于表中记录排序的字段:

```
>>> from collections import namedtuple
>>> from operator import attrgetter
>>> from bisect import bisect, insort
>>> from pprint import pprint

>>> Movie = namedtuple('Movie', ('name', 'released', 'director'))

>>> movies = [
...     Movie('Jaws', 1975, 'Spielberg'),
...     Movie('Titanic', 1997, 'Cameron'),
...     Movie('The Birds', 1963, 'Hitchcock'),
...     Movie('Aliens', 1986, 'Cameron')
... ]

>>> # Find the first movie released after 1960
```

(续下页)

(接上页)

```
>>> by_year = attrgetter('released')
>>> movies.sort(key=by_year)
>>> movies[bisect(movies, 1960, key=by_year)]
Movie(name='The Birds', released=1963, director='Hitchcock')

>>> # Insert a movie while maintaining sort order
>>> romance = Movie('Love Story', 1970, 'Hiller')
>>> insort(movies, romance, key=by_year)
>>> pprint(movies)
[Movie(name='The Birds', released=1963, director='Hitchcock'),
 Movie(name='Love Story', released=1970, director='Hiller'),
 Movie(name='Jaws', released=1975, director='Spielberg'),
 Movie(name='Aliens', released=1986, director='Cameron'),
 Movie(name='Titanic', released=1997, director='Cameron')]
```

如果键函数较为消耗资源，可以通过搜索一个预先计算的键列表来查找记录的索引以避免重复的函数调用:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])       # 或者使用 operator.itemgetter(1)。
>>> keys = [r[1] for r in data]         # 预计算一个由键组成的列表。
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.8 `array` --- 高效的数字值数组

此模块定义了一种对象类型，可以紧凑地表示由基本值（字符、整数、浮点数）组成的数组。数组是序列类型，其行为与列表非常相似，不同之处在于其中存储的对象类型是受限的，在数组对象创建时用单个字符的 类型码来指定。已定义的类型码如下:

类型码	C 类型	Python 类型	最小字节数	备注
`'b'`	signed char	int	1	
`'B'`	unsigned char	int	1	
`'u'`	wchar_t	Unicode 字符	2	(1)
`'w'`	Py_UCS4	Unicode 字符	4	
`'h'`	signed short	int	2	
`'H'`	unsigned short	int	2	
`'i'`	signed int	int	2	
`'I'`	unsigned int	int	2	
`'l'`	signed long	int	4	
`'L'`	unsigned long	int	4	
`'q'`	signed long long	int	8	
`'Q'`	unsigned long long	int	8	
`'f'`	float	float	4	
`'d'`	double	float	8	

备注:

(1) 可能为 16 位或 32 位，取决于具体的平台。

在 3.9 版本发生变更: `array('u')` 现在使用 `wchar_t` 作为 C 类型而不是已不建议使用的 `Py_UNICODE`。这个改变不会影响其行为，因为 `Py_UNICODE` 自 Python 3.3 起就是 `wchar_t` 的别名。

Deprecated since version 3.3, will be removed in version 3.16: 请迁移到 `'w'` 类型码。

值的实际表示是由机器架构（严格说是由 C 实现）决定的。实际大小可以通过*array.itemsize* 属性来访问。

此模块定义了以下项目：

`array.`**typecodes**

一个由所有可用的类型码组成的字符串。

此模块定义了以下类型：

class `array.`**array**(*typecode*[, *initializer*])

一个由 *typecode* 限定其条目的新数组，并能根据可选的 *initializer* 值来初始化。*initializer* 必须是*bytes* 或*bytearray* 对象、Unicode 字符串或元素类型合适的可迭代对象。

如果给定了一个*bytes* 或*bytearray* 对象，则将 *initializer* 传给新数组的*frombytes()* 方法；如果给定了一个 Unicode 字符串，则将 *initializer* 传给*fromunicode()* 方法；在其他情况下，则将 *initializer* 的迭代器传给*extend()* 方法以向数组添加初始条目。

数组对象支持普通的序列操作如索引、切片、拼接和重复等。当使用切片赋值时，所赋的值必须为具有相同类型码的数组对象；所有其他情况都将引发*TypeError*。数组对象也实现了缓冲区接口，可以用于所有支持类字节对象 的场合。

引发一个审计事件 `array.__new__` 并附带参数 `typecode, initializer`。

typecode

在创建数组时使用的类型码字符。

itemsize

内部表示中，单个数组项的长度。单位为字节。

append(*x*)

添加一个值为 *x* 的新项到数组末尾。

buffer_info()

返回一个元组 `(address, length)` 给出存放数组内容的内存缓冲区的当前地址和长度（以元素个数为单位）。以字节为单位的的内存缓冲区大小可通过 `array.buffer_info()[1] * array.itemsize` 来计算。工作在需要内存地址的底层（因此天然地不够安全）的 I/O 接口上时，这有时会有用，例如某些 `ioctl()` 操作。只要数组还存在，并且没有对其应用过改变长度的操作，则返回的数值就是有效的。

> **备注**
>
> 只有在使用以 C 或 C++ 编写的代码中的数组对象时，才能有效利用该信息，但此时，更合理的是，使用数组对象支持的缓冲区接口。因此，该方法的存在仅仅是为了向后兼容性，应避免在新代码中使用。缓冲区接口的文档参见 bufferobjects。

byteswap()

“字节对调”所有数组项。此方法只支持大小为 1, 2, 4 或 8 字节的值；对于其它类型的值将引发*RuntimeError*。当要从另一种字节顺序的机器生成的文件中读取数据时，它很有用。

count(*x*)

返回 *x* 在数组中的出现次数。

extend(*iterable*)

将来自 *iterable* 的项添加到数组末尾。如果 *iterable* 是另一个数组，它必须具有 完全相同的类型码；否则将引发*TypeError*。如果 *iterable* 不是一个数组，则它必须为可迭代对象且其元素的类型须为可添加到数组的适当类型。

frombytes(*buffer*)

添加来自*bytes-like object*的条目，将其内容解读为由机器值组成的数组（就像是使用`fromfile()`方法从文件中读取内容一样）。

Added in version 3.2: `fromstring()` 被重命名为含义更准确的`frombytes()`。

fromfile(*f*, *n*)

从*file object* *f* 中读取 *n* 项（视为机器值）并将它们添加到数组末尾。如果可用的项少于 *n* 项，则会引发`EOFError`，但可用的项仍然会被加进数组。

fromlist(*list*)

将来自列表的项添加到数组末尾。等价于 `for x in list: a.append(x)`，而不同之处在于，若发生类型错误，数组则不会被改变。

fromunicode(*s*)

使用来自给定的 Unicode 字符串的数据扩展该数组。该数组的类型码必须为 `'u'` 或 `'w'`；否则将引发`ValueError`。请使用 `array.frombytes(unicodestring.encode(enc))` 将 Unicode 数据添加到其他类型的数组。

index(*x*[, *start*[, *stop*]])

返回以这样的方式找到的最小的 *i*：*i* 为数组中第一个 *x* 的下标。可选参数 *start* 和 *stop* 用于在数组的一个指定的子段中搜索 *x*。如果未找到 *x* 则会引发`ValueError`。

在 3.10 版本发生变更: 添加了可选的 *start* 和 *stop* 形参。

insert(*i*, *x*)

在数组的位置 *i* 之前插入一个值为 *x* 的新项。负值被视为相对于数组末尾的位置。

pop([*i*])

从数组中移除下标为 *i* 的项并将其返回。参数默认值为 `-1`，因此默认移除并返回末项。

remove(*x*)

从数组中移除第一个出现的 *x*。

clear()

从数组中移除所有元素。

Added in version 3.13.

reverse()

反转数组中各项的顺序。

tobytes()

将数组转换为一个由机器值组成的数组并返回其字节表示（和用`tofile()`方法写入文件的字节序列相同）。

Added in version 3.2: `tostring()` 被重命名为含义更准确的`tobytes()`。

tofile(*f*)

将所有项（作为机器值）写入*file object* *f*。

tolist()

将数组转换为由相同的项组成的普通列表。

tounicode()

将数组转换为一个 Unicode 字符串。数组的类型必须为 `'u'` 或 `'w'`；否则将引发`ValueError`。请使用 `array.tobytes().decode(enc)` 来从其他类型的数组获取 Unicode 字符串。

数组对象的字符串表示形式是 `array(typecode, initializer)`。如果数组为空则 *initializer* 将被省略，否则如果 *typecode* 为 `'u'` 或 `'w'` 则为 Unicode 字符串，否则为由数字组成的列表。只要`array`类是使用 `from array import array` 导入的，该字符串表示形式就保证能使用`eval()`转换回具有相同类型和值的数组。如果它包含相应的浮点数值则还必须定义变量 `inf` 和 `nan`。例如:

```
array('l')
array('w', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14, -inf, nan])
```

参见

***struct* 模块**
打包和解包异构二进制数据。

NumPy
NumPy 包定义了另一数组类型。

8.9 weakref --- 弱引用

源代码： Lib/weakref.py

---

weakref 模块允许 Python 程序员创建对象的 *弱引用*。

在下文中，术语 *所指对象* 表示弱引用所指向的对象。

对象的弱引用不能保证对象存活：当所指对像的引用只剩弱引用时，*垃圾回收* 可以销毁所指对象，并将其内存重新用于其它用途。但是，在实际销毁对象之前，即使没有强引用，弱引用也能返回该对象。

弱引用的一个主要用途是实现一个存储大型对象的缓存或映射，但又不希望该大型对象仅因为它只出现在这个缓存或映射中而保持存活。

例如，如果你有许多大型二进制图像对象，你可能希望为每个对象关联一个名称。如果你使用 Python 字典来将名称映射到图像，或将图像映射到名称，那么图像对象将因为它们在字典中作为值或键而保持存活。*weakref* 模块提供的*WeakKeyDictionary* 和*WeakValueDictionary* 类可以替代 Python 字典，它们使用弱引用来构造映射，这种映射不会仅因为对象出现在映射中而使对象保持存活。例如，如果一个图像对象是*WeakValueDictionary* 中的值，那么当对该图像对象的剩余引用是弱映射对象所持有的弱引用时，垃圾回收器将回收该对象，并删除弱映射对象中相应的条目。

WeakKeyDictionary 和*WeakValueDictionary* 在它们的实现中使用了弱引用，并在弱引用上设置当键或值被垃圾回收器回收时通知弱字典的回调函数。*WeakSet* 实现了*set* 接口，但像*WeakKeyDictionary* 一样，只持有其元素的弱引用。

finalize 提供了一种直接的方法来注册当对象被垃圾收集时要调用的清理函数。这比在普通的弱引用上设置回调函数的方式更简单，因为模块会自动确保对象被回收前终结器一直保持存活。

这些弱容器类型之一或者*finalize* 就是大多数程序所需要的——通常不需要直接创建自己的弱引用。*weakref* 模块暴露了底层机制，以便用于高级用途。

并非所有对象都可以被弱引用。支持弱引用的对象包括类实例、用 Python（而非用 C）编写的函数、实例方法、集合、冻结集合、某些文件对象、生成器、类型对象、套接字、数组、双端队列、正则表达式模式对象以及代码对象。

在 3.2 版本发生变更: 添加了对 thread.lock，threading.Lock 和代码对象的支持。

一些内置类型，如*list* 和*dict*，不直接支持弱引用，但可以通过子类化添加支持:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # 此对象是可弱引用的
```

其他内置类型，如*tuple* 和*int*，不支持弱引用，即使通过子类化也不支持。

可以轻松地使扩展类型支持弱引用；参见 weakref-support。

当为某个给定类型定义了 __slots__ 时，弱引用支持会被禁用，除非将 '__weakref__' 字符串也加入到 __slots__ 声明的字符串序列中。请参阅 __slots__ 文档了解详情。

class weakref.**ref**(*object*[, *callback*])

返回 *object* 的弱引用。如果所指对象存活，则可以通过调用引用对象来获取原始对象；如果所指对象不存在，则调用引用对象将得到*None*。如果提供了值不是*None*的 *callback*，并且返回的弱引用对象仍然存活，则在对象即将终结时将调用回调函数；弱引用对象将作为回调函数的唯一参数传递；然后所指对象将不再可用。

允许为同一个对象的构造多个弱引用。每个弱引用注册的回调函数将按从最近注册的回调函数，到最早注册的回调函数的顺序调用。

由回调函数引发的异常将记录于标准错误输入，但无法传播该异常；这些异常的处理方式与对象 __del__() 方法引发异常的处理方式相同。

如果 *object* 可哈希，则弱引用也*可哈希*。即使在 *object* 被删除之后，弱引用仍将保持其哈希值。如果在 *object* 被删除之后才首次调用*hash()*，则该调用将引发*TypeError*。

弱引用支持相等性测试，但不支持排序。如果所指对象仍然存活，两个引用具有与它们的所指对象具有一致的相等关系（无论 *callback* 是否相同）。如果删除了任一所指对象，则仅在两个引用指向同一对象时，二者才相等。

这是一个可子类化的类型，而非一个工厂函数。

__callback__

这个只读属性会返回当前关联到弱引用的回调函数。如果回调函数不存在，或弱引用的所指对象已不存在，则此属性的值为 None。

在 3.4 版本发生变更: 添加了*__callback__* 属性。

weakref.**proxy**(*object*[, *callback*])

返回一个使用弱引用的 *object* 代理。此函数支持在大多数上下文中使用代理，而不要求显式地解引用弱引用对象。返回的对象类型将为 ProxyType 或 CallableProxyType，具体取决于 *object* 是否为可调用对象。无论所指对象是否可哈希，代理对象都不属于*可哈希*对象；这避免了与它们的基本可变性质相关的许多问题，且防止代理被用作字典的键。*callback* 形参含义与*ref()*函数的同名形参含义相同。

在所指对象被作为垃圾回收后访问代理对象的属性将引发*ReferenceError*。

在 3.8 版本发生变更: 扩展代理对象所支持的运算符，包括矩阵乘法运算符 @ 和 @=。

weakref.**getweakrefcount**(*object*)

返回指向 *object* 的弱引用和代理的数量。

weakref.**getweakrefs**(*object*)

返回由指向 *object* 的所有弱引用和代理构成的列表。

class weakref.**WeakKeyDictionary**([*dict*])

弱引用键的映射类。当不再存在对键的强引用时，字典中的相关条目将被丢弃。这可用于将额外数据与应用程序中其它部分拥有的对象相关联，而无需向这些对象添加属性。这对于重写了属性访问的对象来说特别有用。

请注意，当把一个与现有键具有相同值（但是标识号不相等）的键插入字典时，它会替换该值，但不会替换现有的键。由于这一点，当删除对原来的键的引用时，也将同时删除字典中的对应条目:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> d[k2] = 2   # d = {k1: 2}
>>> del k1      # d = {}
```

一种变通做法是在重新赋值之前先移除键:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> del d[k1]
>>> d[k2] = 2   # d = {k2: 2}
>>> del k1      # d = {k2: 2}
```

在 3.9 版本发生变更: 增加了对 | 和 |= 运算符的支持，相关说明见 **PEP 584**。

WeakKeyDictionary 对象具有一个额外方法，可以直接公开内部引用。这些引用不保证在它们被使用时仍然保持“存活”，因此这些引用的调用结果需要在使用前进行检测。此方法可用于避免创建会导致垃圾回收器将保留键超出实际需要时长的引用。

WeakKeyDictionary.**keyrefs**()

返回包含对键的弱引用的可迭代对象。

class weakref.**WeakValueDictionary**([*dict*])

弱引用值的映射类。当不再存在对该值的强引用时，字典中的条目将被丢弃。

在 3.9 版本发生变更: 增加了对 | 和 |= 运算符的支持，相关说明见 **PEP 584**。

WeakValueDictionary 对象具有一个额外方法，此方法存在与*WeakKeyDictionary.keyrefs()* 方法相同的问题。

WeakValueDictionary.**valuerefs**()

返回包含对值的弱引用的可迭代对象。

class weakref.**WeakSet**([*elements*])

保持对其元素弱引用的集合类。当某个元素没有强引用时，该元素将被丢弃。

class weakref.**WeakMethod**(*method*[, *callback*])

一个模拟对绑定方法（即在类中定义并在实例中查找的方法）进行弱引用的自定义*ref* 子类。由于绑定方法是临时性的，标准弱引用无法保持它。*WeakMethod* 包含特别代码用来重新创建绑定方法，直到对象或初始函数被销毁:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

callback 与*ref()* 函数的同名形参含义相同。

Added in version 3.4.

class weakref.**finalize**(*obj*, *func*, */*, **args*, ***kwargs*)

返回一个可调用的终结器对象，该对象将在 *obj* 作为垃圾回收时被调用。与普通的弱引用不同，终结器将总是存活，直到引用对象被回收，这极大地简化了生命周期管理。

终结器总是被视为 *存活* 直到它被调用（显式调用或在垃圾回收时隐式调用），调用之后它将 *死亡*。调用存活的终结器将返回 `func(*arg, **kwargs)` 的求值结果，而调用死亡的终结器将返回*None*。

在垃圾收集期间由终结器回调所引发的异常将显示在标准错误输出中，但无法被传播。它们会按与对象的 `__del__()` 方法或或弱引用的回调所引发的异常相同的方式被处理。

当程序退出时，剩余的存活终结器会被调用，除非它们的*atexit* 属性已被设为假值。它们会按与创建时相反的顺序被调用。

终结器在*interpreter shutdown* 的后期绝不会唤起其回调函数，此时模块全局变量很可能已被替换为*None*。

__call__()

如果 *self* 为存活状态则将其标记为已死亡，并返回调用 `func(*args, **kwargs)` 的结果。如果 *self* 已死亡则返回*None*。

detach()

如果 *self* 为存活状态则将其标记为已死亡，并返回元组 `(obj, func, args, kwargs)`。如果 *self* 已死亡则返*None*。

peek()

如果 *self* 为存活状态则返回元组 `(obj, func, args, kwargs)`。如果 *self* 已死亡则返回*None*。

alive

如果终结器为存活状态则该特征属性为真值，否则为假值。

atexit

一个可写的布尔型特征属性，默认为真值。当程序退出时，它会调用所有*atexit* 为真值的剩余存活终结器。它们会按与创建时相反的顺序被调用。

备注

很重要的一点是确保 *func*, *args* 和 *kwargs* 不拥有任何对 *obj* 的引用，无论是直接的或是间接的，否则的话 *obj* 将永远不会被作为垃圾回收。特别地，*func* 不应当是 *obj* 的一个绑定方法。

Added in version 3.4.

`weakref.`**ReferenceType**

弱引用对象的类型对象。

`weakref.`**ProxyType**

不可调用对象的代理的类型对象。

`weakref.`**CallableProxyType**

可调用对象的代理的类型对象。

`weakref.`**ProxyTypes**

包含所有代理的类型对象的序列。这可以用于更方便地检测一个对象是否是代理，而不必依赖于两种代理对象的名称。

参见

PEP 205 - 弱引用
此特性的提议和理由，包括早期实现的链接和其他语言中类似特性的相关信息。

8.9.1 弱引用对象

弱引用对象没有*ref.__callback__* 以外的方法和属性。一个弱引用对象如果存在，就允许通过调用它来获取引用:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

如果引用已不存在，则调用引用对象将返回*None*:

```
>>> del o, o2
>>> print(r())
None
```

检测一个弱引用对象是否仍然存在应该使用表达式 `ref() is not None`。通常，需要使用引用对象的应用代码应当遵循这样的模式:

```
# r 是一个弱引用对象
o = r()
if o is None:
    # 引用已被作为垃圾回收
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

使用单独的“存活”测试会在多线程应用中制造竞争条件；其他线程可能导致某个弱引用在该弱引用被调用前就失效；上述的写法在多线程应用和单线程应用中都是安全的。

特别版本的*ref* 对象可以通过子类化来创建。在*WeakValueDictionary* 的实现中就使用了这种方式来减少映射中每个条目的内存开销。这对于将附加信息关联到引用的情况最为适用，但也可以被用于在调用中插入额外处理来提取引用。

这个例子演示了如何将*ref* 的一个子类用于存储有关对象的附加信息并在引用被访问时影响其所返回的值:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """返回一个包含引用和该引用被调用次数的
        二元组。
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.9.2 示例

这个简单的例子演示了一个应用如何使用对象 ID 来提取之前出现过的对象。然后对象的 ID 可以在其它数据结构中使用，而无须强制对象保持存活，但处于存活状态的对象也仍然可以通过 ID 来提取。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.9.3 终结器对象

使用*finalize*的主要好处在于它能更简便地注册回调函数，而无须保留所返回的终结器对象。例如

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

终结器也可以被直接调用。但是终结器最多只能对回调函数发起一次调用。

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

你可以使用*detach()*方法来注销一个终结器。该方法将销毁终结器并返回其被创建时传给构造器的参数。

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

除非你将*atexit*属性设为*False*，否则终结器在程序退出时如果仍然存活就将被调用。例如

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.9.4 比较终结器与 __del__() 方法

假设我们想创建一个类，用它的实例来代表临时目录。当以下事件中的某一个发生时，这个目录应当与其内容一起被删除：

- 对象被作为垃圾回收，
- 对象的 remove() 方法被调用，或
- 程序退出。

我们可以像下面这样尝试使用 __del__() 方法来实现这个类：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

从 Python 3.4 开始，__del__() 方法会不再阻止循环引用被作为垃圾回收，并且模块全局变量在*interpreter shutdown* 期间不会再被强制设为*None*。因此这段代码在 CPython 上应该会正常运行而不会出现任何问题。

然而，__del__() 方法的处理严重受影响于具体实现，因为它依赖于解释器的垃圾回收实现方式的内部细节。

更健壮的替代方式可以是定义一个终结器，只引用它所需要的特定函数和对象，而不是获取对整个对象状态的访问权：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

像这样定义后，我们的终结器将只接受一个对其完成正确清理目录任务所需细节的引用。如果对象一直未被作为垃圾回收，终结器仍会在退出时被调用。

基于弱引用的终结器还具有另一项优势，就是它们可被用来为定义由第三方控制的类注册终结器，例如当一个模块被卸载时运行特定代码：

```
import weakref, sys
def unloading_module():
    # 从函数体隐式地引用模块的 globals
weakref.finalize(sys.modules[__name__], unloading_module)
```

备注

如果当程序退出时你恰好在守护线程中创建终结器对象，则有可能该终结器不会在退出时被调用。但是，在一个守护线程中*atexit.register()*，try: ... finally: ... 和 with: ... 同样不能保证执行清理。

8.10 types --- 动态类型创建和内置类型名称

源代码: Lib/types.py

此模块定义了一些工具函数，用于协助动态创建新的类型。

它还为某些对象类型定义了名称，这些名称由标准 Python 解释器所使用，但并不像内置的*int* 或*str* 那样对外公开。

最后，它还额外提供了一些类型相关但重要程度不足以作为内置对象的工具类和函数。

8.10.1 动态类型创建

types.**new_class**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

使用适当的元类动态地创建一个类对象。

前三个参数是组成类定义头的部件：类名称，基类 (有序排列)，关键字参数 (例如 metaclass)。

exec_body 参数是一个回调函数，用于填充新创建类的命名空间。它应当接受类命名空间作为其唯一的参数并使用类内容直接更新命名空间。如果未提供回调函数，则它就等效于传入 lambda ns: None。

Added in version 3.3.

types.**prepare_class**(*name*, *bases=()*, *kwds=None*)

计算适当的元类并创建类命名空间。

参数是组成类定义头的部件：类名称，基类 (有序排列) 以及关键字参数 (例如 metaclass)。

返回值是一个 3 元组: metaclass, namespace, kwds

metaclass 是适当的元类，*namespace* 是预备好的类命名空间而 *kwds* 是所传入 *kwds* 参数移除每个 'metaclass' 条目后的已更新副本。如果未传入 *kwds* 参数，这将为一个空字典。

Added in version 3.3.

在 3.6 版本发生变更: 所返回元组中 namespace 元素的默认值已被改变。现在当元类没有 __prepare__ 方法时将会使用一个保留插入顺序的映射。

参见

metaclasses
这些函数所支持的类创建过程的完整细节

PEP 3115 - Python 3000 中的元类
引入 __prepare__ 命名空间钩子

types.**resolve_bases**(*bases*)

动态地解析 MRO 条目，具体描述见 **PEP 560**。

此函数会在 *bases* 中查找不是*type* 的实例的项，并返回一个元组，其中每个具有 `__mro_entries__()` 方法的此种对象将被替换为调用该方法解包后的结果。如果一个 *bases* 项是*type* 的实例，或它不具有 `__mro_entries__()` 方法，则它将不加改变地被包括在返回的元组中。

Added in version 3.7.

types.**get_original_bases**(*cls*, */*)

在 `__mro_entries__()` 方法在任何基类上被调用之前返回最初是作为 *cls* 的基类给出的对象元组（根据 **PEP 560** 所描述的机制）。这在对泛型 进行内省时很有用处。

对于具有 `__orig_bases__` 属性的类，此函数将返回 `cls.__orig_bases__` 的值。对于没有 `__orig_bases__` 属性的类，则将返回 `cls.__bases__`。

示例:

```
from typing import TypeVar, Generic, NamedTuple, TypedDict

T = TypeVar("T")
class Foo(Generic[T]): ...
class Bar(Foo[int], float): ...
class Baz(list[str]): ...
Eggs = NamedTuple("Eggs", [("a", int), ("b", str)])
Spam = TypedDict("Spam", {"a": int, "b": str})

assert Bar.__bases__ == (Foo, float)
assert get_original_bases(Bar) == (Foo[int], float)

assert Baz.__bases__ == (list,)
assert get_original_bases(Baz) == (list[str],)

assert Eggs.__bases__ == (tuple,)
assert get_original_bases(Eggs) == (NamedTuple,)

assert Spam.__bases__ == (dict,)
assert get_original_bases(Spam) == (TypedDict,)

assert int.__bases__ == (object,)
assert get_original_bases(int) == (object,)
```

Added in version 3.12.

参见

PEP 560 - 对 typing 模块和泛型类型的核心支持

8.10.2 标准解释器类型

此模块为许多类型提供了实现 Python 解释器所要求的名称。它刻意地避免了包含某些仅在处理过程中偶然出现的类型，例如 `listiterator` 类型。

此种名称的典型应用如*isinstance()* 或*issubclass()* 检测。

如果你要实例化这些类型中的任何一种，请注意其签名在不同 Python 版本之间可能出现变化。

以下类型有相应的标准名称定义：

types.**NoneType**

None 的类型。

Added in version 3.10.

types.**FunctionType**

types.**LambdaType**

用户自定义函数以及由 lambda 表达式所创建函数的类型。

引发一个审计事件 function.__new__ 并附带参数 code。

此审计事件只会被函数对象的直接实例化引发，而不会被普通编译所引发。

types.**GeneratorType**

generator 迭代器对象的类型，由生成器函数创建。

types.**CoroutineType**

coroutine 对象的类型，由 async def 函数创建。

Added in version 3.5.

types.**AsyncGeneratorType**

asynchronous generator 迭代器对象的类型，由异步生成器函数创建。

Added in version 3.6.

class types.**CodeType**(***kwargs*)

代码对象例如*compile()* 返回值的类型。

引发一个审计事件 code.__new__ 并附带参数 code, filename, name, argcount, posonlyargcount, kwonlyargcount, nlocals, stacksize, flags。

请注意被审计的参数可能与初始化代码所要求的名称或位置不相匹配。审计事件只会被代码对象的直接实例化引发，而不会被普通编译所引发。

types.**CellType**

单元对象的类型：这种对象被用作函数中闭包变量 的容器。

Added in version 3.8.

types.**MethodType**

用户自定义类实例方法的类型。

types.**BuiltinFunctionType**

types.**BuiltinMethodType**

内置函数例如*len()* 或*sys.exit()* 以及内置类方法的类型。(这里所说的“内置”是指“以 C 语言编写”。)

types.**WrapperDescriptorType**

某些内置数据类型和基类的方法的类型，例如 object.__init__() 或 object.__lt__()。

Added in version 3.7.

types.**MethodWrapperType**

某些内置数据类型和基类的 绑定方法的类型。例如 object().__str__ 所属的类型。

Added in version 3.7.

types.**NotImplementedType**

NotImplemented 的类型。

Added in version 3.10.

types.**MethodDescriptorType**

某些内置数据类型方法例如*str.join()* 的类型。

Added in version 3.7.

types.**ClassMethodDescriptorType**

某些内置数据类型 非绑定类方法例如 dict.__dict__['fromkeys'] 的类型。

Added in version 3.7.

class types.**ModuleType**(*name*, *doc=None*)

模块 的类型。构造器接受待创建模块的名称并以其*docstring* 作为可选参数。

> **参见**
>
> **模块对象的文档**
> 提供了有关可在 ModuleType 的实例上找到的特殊属性的详情。
>
> ***importlib.util.module_from_spec()***
> 使用 ModuleType 构造器创建的模块在被创建时将有许多特殊属性被设置或重设其默认值。module_from_spec() 提供了一种创建 ModuleType 实例的更健壮方式，可确保各个属性被正确地设置。

types.**EllipsisType**

Ellipsis 的类型。

Added in version 3.10.

class types.**GenericAlias**(*t_origin*, *t_args*)

形参化泛型 的类型，例如 list[int]。

t_origin 应当是一个非形参化的泛型类，例如 list, tuple 或 dict。t_args 应当是一个形参化 t_origin 的*tuple* (长度可以为 1):

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

Added in version 3.9.

在 3.9.2 版本发生变更: 此类型现在可以被子类化。

> **参见**
>
> **泛用别名类型**
> 有关 types.GenericAlias 实例的详细文档
>
> **PEP 585 - 标准多项集中的类型提示泛型**
> 引入 types.GenericAlias 类

class types.**UnionType**

合并类型表达式 的类型。

Added in version 3.10.

class types.**TracebackType**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)

回溯对象的类型，如在 sys.exception().__traceback__ 中找到的一样。

请查看 语言参考了解可用属性和操作的细节，以及动态地创建回溯对象的指南。

types.**FrameType**

帧对象的类型，例如当 tb 是一个回溯对象时 tb.tb_frame 中的对象。

types.**GetSetDescriptorType**

使用 PyGetSetDef 在扩展模块中定义的对象的类型，例如 FrameType.f_locals 或 array.array.typecode。此类型被用作对象属性的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

types.**MemberDescriptorType**

使用 PyMemberDef 在扩展模块中定义的对象的类型，例如 datetime.timedelta.days。此类型被用作使用标准转换函数的简单 C 数据成员的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

此外，当一个类定义了 __slots__ 属性时，对于每个槽位，都将添加一个 MemberDescriptorType 的实例作为该类上的属性。这将允许槽位显示在类的 __dict__ 中。

在 Python 的其它实现中，此类型可能与 GetSetDescriptorType 完全相同。

class types.**MappingProxyType**(*mapping*)

一个映射的只读代理。它提供了对映射条目的动态视图，这意味着当映射发生改变时，视图会反映这些改变。

Added in version 3.3.

在 3.9 版本发生变更: 更新为支持 **PEP 584** 所新增的合并 (|) 运算符，它会简单地委托给下层的映射。

key in proxy

如果下层的映射中存在键 *key* 则返回 True，否则返回 False。

proxy[key]

返回下层的映射中以 *key* 为键的项。如果下层的映射中不存在键 *key* 则引发*KeyError*。

iter(proxy)

返回由下层映射的键为元素的迭代器。这是 iter(proxy.keys()) 的快捷方式。

len(proxy)

返回下层映射中的项数。

copy()

返回下层映射的浅拷贝。

get(*key*[, *default*])

如果 *key* 存在于下层映射中则返回 *key* 的值，否则返回 *default*。如果 *default* 未给出则默认为 None，因而此方法绝不会引发*KeyError*。

items()

返回由下层映射的项 ((键, 值) 对) 组成的一个新视图。

keys()

返回由下层映射的键组成的一个新视图。

values()

返回由下层映射的值组成的一个新视图。

reversed(proxy)

返回一个包含下层映射的键的反向迭代器。

Added in version 3.9.

hash(proxy)

返回下层映射的哈希值。

Added in version 3.12.

class types.**CapsuleType**

capsule 对象的类型。

Added in version 3.13.

8.10.3 附加工具类和函数

class types.**SimpleNamespace**

一个简单的*object* 子类，提供了访问其命名空间的属性，以及一个有意义的 repr。

与*object* 不同的是，对于 SimpleNamespace 你可以添加和移除属性。

SimpleNamespace 对象可以使用与*dict* 相同的方式来初始化：关键字参数、单个位置参数或者两者兼有。当使用关键字参数来初始化时，参数值会被直接加入到下层的命名空间。而当使用一个位置参数来初始化时，下层的命名空间将以来自该参数（为一个映射对象或是产生键值对的*iterable* 对象）的键值对来更新。所有这样的键都必须为字符串。

此类型大致等价于以下代码:

```
class SimpleNamespace:
    def __init__(self, mapping_or_iterable=(), /, **kwargs):
        self.__dict__.update(mapping_or_iterable)
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other, SimpleNamespace):
           return self.__dict__ == other.__dict__
        return NotImplemented
```

SimpleNamespace 可被用于替代 class NS: pass。但是，对于结构化记录类型则应改用*namedtuple()*。

SimpleNamespace 对象受到*copy.replace()* 的支持。

Added in version 3.3.

在 3.9 版本发生变更: repr 中的属性顺序由字母顺序改为插入顺序 (类似 dict)。

在 3.13 版本发生变更: 增加了对可选的位置参数的支持。

types.**DynamicClassAttribute**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

在类上访问 __getattr__ 的路由属性。

这是一个描述器，用于定义通过实例与通过类访问时具有不同行为的属性。当实例访问时保持正常行为，但当类访问属性时将被路由至类的 __getattr__ 方法；这是通过引发 AttributeError 来完成的。

这允许有在实例上激活的特性属性，同时又有在类上的同名虚拟属性 (一个例子请参见*enum.Enum*)。

Added in version 3.4.

8.10.4 协程工具函数

types.**coroutine**(*gen_func*)

此函数可将*generator* 函数转换为一个返回基于生成器的协程的*coroutine function*。基于生成器的协程仍然属于*generator iterator*，但同时又可被视为*coroutine* 对象兼*awaitable*。不过，它没有必要实现 __await__() 方法。

如果 *gen_func* 是一个生成器函数，它将被原地修改。

如果 *gen_func* 不是一个生成器函数，则它会被包装。如果它返回一个*collections.abc.Generator* 的实例，该实例将被包装在一个 *awaitable* 代理对象中。所有其他对象类型将被原样返回。

Added in version 3.5.

8.11 copy --- 浅层及深层拷贝操作

源代码: Lib/copy.py

Python 的赋值语句不复制对象，而是创建目标和对象的绑定关系。对于自身可变，或包含可变项的集合，有时要生成副本用于改变操作，而不必改变原始对象。本模块提供了通用的浅层复制和深层复制操作，（如下所述）。

接口摘要：

copy.**copy**(*obj*)
: 返回 *obj* 的浅拷贝。

copy.**deepcopy**(*obj*[, *memo*])
: 返回 *obj* 的深拷贝。

copy.**replace**(*obj*, /, ***changes*)
: 新建一个与 *obj* 类型相同的对象，使用来自 *changes* 的值替换字段。

  Added in version 3.13.

exception copy.**Error**
: 针对模块特定错误引发。

浅层与深层复制的区别仅与复合对象（即包含列表或类的实例等其他对象的对象）相关：

- *浅层复制*构造一个新的复合对象，然后（在尽可能的范围内）将原始对象中找到的对象的*引用*插入其中。
- *深层复制*构造一个新的复合对象，然后，递归地将在原始对象里找到的对象的*副本*插入其中。

深度复制操作通常存在两个问题，而浅层复制操作并不存在这些问题：

- 递归对象（直接或间接包含对自身引用的复合对象）可能会导致递归循环。
- 由于深层复制会复制所有内容，因此可能会过多复制（例如本应该在副本之间共享的数据）。

deepcopy() 函数用以下方式避免了这些问题：

- 保留在当前复制过程中已复制的对象的"备忘录"（memo）字典；以及
- 允许用户定义的类重写复制操作或复制的组件集合。

此模块不会复制模块、方法、栈追踪、栈帧、文件、套接字、窗口以及任何相似的类型。它会通过不加修改地返回原始对象来（浅层或深层地）"复制"函数和类；这与*pickle*模块处理这类问题的方式是兼容的。

制作字典的浅层复制可以使用*dict.copy()*方法，而制作列表的浅层复制可以通过赋值整个列表的切片完成，例如，`copied_list = original_list[:]`。

类可以使用与控制序列化（pickling）操作相同的接口来控制复制操作，关于这些方法的描述信息请参考*pickle*模块。实际上，*copy*模块使用的正是从*copyreg*模块中注册的 pickle 函数。

为了让一个类能够定义它自己的拷贝实现，它可以定义特殊方法*__copy__()*和*__deepcopy__()*。

object.**__copy__**(*self*)
: 调用以实现浅拷贝操作；无须传入任何额外参数。

object.**__deepcopy__**(*self*, *memo*)
: 调用以实现深拷贝操作；它将传入一个参数，即 *memo* 字典。如果 __deepcopy__ 实现需要创建一个组件的深拷贝，它应当调用*deepcopy()*函数并将该组件作为第一个参数而将 *memo* 字典作为第二个参数。*memo* 字典应当被当作不透明对象来处理。

函数 `copy.replace()` 相比*copy()*和*deepcopy()*受到更多的限制，并且仅支持由*namedtuple()*创建的元组，*dataclasses*及其他定义了*__replace__()*方法的类。

object.**__replace__**(*self*, */*, ***changes*)

此函数应当新建一个具有相同类型的对象，使用来自 *changes* 的值替换字段。

参见

模块*pickle*
讨论了支持对象状态检索和恢复的特殊方法。

8.12 pprint --- 数据美化输出

源代码： Lib/pprint.py

---

pprint 模块提供了“美化打印”任意 Python 数据结构的功能，这种美化形式可用作对解释器的输入。如果经格式化的结构包含非基本 Python 类型的对象，则其美化形式可能无法被加载。包含文件、套接字或类对象，以及许多其他不能用 Python 字面值来表示的对象都有可能导致这样的结果。

已格式化的表示形式会在可能的情况下将对象放在单行中，而当它们不能在允许宽度中被容纳时将其分为多行，允许宽度可由默认为 80 个字符的 *width* 形参加以调整。

字典在计算其显示形式前会先根据键来排序。

在 3.9 版本发生变更: 添加了对美化打印*types.SimpleNamespace* 的支持。

在 3.10 版本发生变更: 添加了对美化打印*dataclasses.dataclass* 的支持。

8.12.1 函数

pprint.**pp**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=False*, *underscore_numbers=False*)

打印 *object* 的格式化表示形式，末尾加一个换行符。此函数可以在交互式解释器中代替*print()* 函数用于检查对象值。提示：你可以执行重赋值 `print = pprint.pp` 以在指定作用域内使用。

参数

- **object** -- 要打印的对象。
- **stream** (*file-like object* | None) -- 一个文件型对象，可通过调用其 `write()` 方法将输出写入该对象。如为 `None` (默认值)，则使用*sys.stdout*。
- **indent** (`int`) -- 要为每个嵌套层级添加的缩进量。
- **width** (`int`) -- 输出中每行所允许的最大字符数。如果一个结构无法在宽度限制内被格式化，则将尽可能的接近。
- **depth** (`int` | *None*) -- 可被找钱的嵌套层级数量。如果要打印的数据结构具有过深的层级，则其包含的下一层级将用 `...` 替换。如为 `None` (默认值)，则不会限制被格式化对象的层级深度。
- **compact** (`bool`) -- 控制长*序列* 的格式化方式。如为 `False` (默认值)，则序列的每一项将被格式化为单独的行，否则在格式化每个输出行时将根据 *width* 限制容纳尽可能多的条目。
- **sort_dicts** (`bool`) -- 如为 `True`，则在格式化字典时将基于键进行排序，否则将按插入顺序显示它们（默认）。
- **underscore_numbers** (`bool`) -- 如为 `True`，则在格式化整数时将使用 _ 字符作为千位分隔符，否则将不显示下划线（默认）。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pp(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

Added in version 3.8.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

默认将 *sort_dicts* 设为 `True` 的*pp()* 的别名，它将自动按字典的键进行排序，你也可以选择使用该参数默认为 `False` 的*pp()*。

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

将 *object* 的格式化表示形式作为字符串返回。*indent*, *width*, *depth*, *compact*, *sort_dicts* 和 *underscore_numbers* 将作为格式化形参传递给*PrettyPrinter* 构造器，它们的含义请参阅前面文档中的说明。

pprint.**isreadable**(*object*)

确定 *object* 的格式化表示是否“可读”，或是否可被用来通过*eval()* 重新构建对象的值。此函数对于递归对象总是返回 `False`。

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

确定 *object* 是否需要递归的表示。此函数会受到下面*saferepr()* 所提及的同样限制的影响并可能在无法检测到可递归对象时引发*RecursionError*。

pprint.**saferepr**(*object*)

返回 *object* 的字符串表示，并为某些通用数据结构提供防递归保护，包括*dict*, *list* 和*tuple* 或其未重载 `__repr__` 的子类的实例。如果该对象表示形式公开了一个递归条目，该递归引用会被表示为 `<Recursion on typename with id=number>`。否则该表示形式将不会被格式化。

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.12.2 PrettyPrinter 对象

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

构造一个*PrettyPrinter* 实例。

参数的含义与*pp()* 的相同。注意它们的顺序有所不同，并且 *sort_dicts* 默认为 `True`。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
```

(续下页)

(接上页)

```
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

在 3.4 版本发生变更: 增加了 *compact* 形参。

在 3.8 版本发生变更: 增加了 *sort_dicts* 形参。

在 3.10 版本发生变更: 添加了 *underscore_numbers* 形参。

在 3.11 版本发生变更: 如果 `sys.stdout` 为 `None` 则将不会尝试向其中写入。

PrettyPrinter 的实例具有下列方法:

PrettyPrinter.**pformat**(*object*)

返回 *object* 格式化表示。这会将传给*PrettyPrinter* 构造器的选项纳入考虑。

PrettyPrinter.**pprint**(*object*)

在所配置的流上打印 *object* 的格式化表示，并附加一个换行符。

下列方法提供了与同名函数相对应的实现。在实例上使用这些方法效率会更高一些，因为不需要创建新的*PrettyPrinter* 对象。

PrettyPrinter.**isreadable**(*object*)

确定对象的格式化表示是否“可读”，或者是否可使用*eval()* 重建对象值。请注意此方法对于递归对象将返回 `False`。如果设置了*PrettyPrinter* 的 *depth* 形参并且对象深度超出允许范围，此方法将返回 `False`。

PrettyPrinter.**isrecursive**(*object*)

确定对象是否需要递归表示。

此方法作为一个钩子提供，允许子类修改将对象转换为字符串的方式。默认实现使用*saferepr()* 实现的内部方式。

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)

返回三个值：字符串形式的 *object* 已格式化版本，指明结果是否可读的旗标，以及指明是否检测到递归的旗标。第一个参数是要表示的对象。第二个是以对象*id()* 为键的字典，这些对象是当前表示上下文的一部分（影响 *object* 表示的直接和间接容器）；如果需要呈现一个已经在 *context* 中表示的对象，则第三个返回值应当为 `True`。对*format()* 方法的递归调用应当将容器的附加条目添加到此字典中。第三个参数 *maxlevels* 给出了对递归的请求限制；如果没有请求限制则其值将为 `0`。此参数应当不加修改地传给递归调用。第四个参数 *level* 给出于当前层级；传给递归调用的参数值应当小于当前调用的值。

8.12.3 示例

为了演示 To demonstrate several uses of the *pp()* 函数及其形参的几种用法，让我们从 PyPI 获取关于某个项目的信息:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/1.2.0/json') as resp:
...     project_info = json.load(resp)['info']
```

在其基本形式中，*pp()* 会显示整个对象:

```
>>> pprint.pp(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

结果可以被限制到特定的 *depth* (更深层的内容将使用省略号):

```
>>> pprint.pp(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
```

(续下页)

(接上页)

```
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

此外，还可以设置建议的最大字符 *width*。如果一个对象无法被拆分，则将超出指定宽度:

```
>>> pprint.pp(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
```

(续下页)

(接上页)

```
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.13 reprlib --- 替代性 repr() 实现

源代码: Lib/reprlib.py

---

reprlib 模块提供了一种对象表示的产生方式，它会对结果字符串的大小进行限制。该方式被用于 Python 调试器并可能同样适用于某些其他场景。

此模块提供了一个类、一个实例和一个函数:

class reprlib.**Repr**(*, *maxlevel=6*, *maxtuple=6*, *maxlist=6*, *maxarray=5*, *maxdict=4*, *maxset=6*, *maxfrozenset=6*, *maxdeque=6*, *maxstring=30*, *maxlong=40*, *maxother=30*, *fillvalue='...'*, *indent=None*)

该类提供了格式化服务适用于实现与内置*repr()* 相似的方法；其中附加了针对不同对象类型的大小限制，以避免生成超长的表示。

该构造器的关键字参数可被用作设置*Repr* 实例属性的快捷方式。这意味着以下的初始化:

```
aRepr = reprlib.Repr(maxlevel=3)
```

等价于:

```
aRepr = reprlib.Repr()
aRepr.maxlevel = 3
```

请参阅*Repr Objects* 小节了解有关*Repr* 属性的信息。

在 3.12 版本发生变更: 允许通过关键字参数来设置属性。

reprlib.**aRepr**

这是*Repr* 的一个实例，用于提供如下所述的*repr()* 函数。改变此对象的属性将会影响*repr()* 和 Python 调试器所使用的大小限制。

reprlib.**repr**(*obj*)

这是 aRepr 的*repr()* 方法。它会返回与同名内置函数所返回字符串相似的字符串，区别在于附带了对多数类型的大小限制。

在大小限制工具以外，此模块还提供了一个装饰器用于检测对 __repr__() 的递归调用并改用一个占位符来替换。

@reprlib.**recursive_repr**(*fillvalue='...'*)

用于为 __repr__() 方法检查同一线程内部递归调用的装饰器。如果执行了递归调用，则返回 *fillvalue*，在其他情况下，将执行正常的 __repr__() 调用。例如：

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Added in version 3.2.

8.13.1 Repr 对象

Repr 实例对象包含一些属性可以用于为不同对象类型的表示提供大小限制，还包含一些方法可以格式化特定的对象类型。

Repr.**fillvalue**

该字符串将针对递归引用显示。它默认为 ...。

Added in version 3.11.

Repr.**maxlevel**

创建递归表示形式的深度限制。默认为 6。

Repr.**maxdict**

Repr.**maxlist**

Repr.**maxtuple**

Repr.**maxset**

Repr.**maxfrozenset**

Repr.**maxdeque**

Repr.**maxarray**

表示命名对象类型的条目数量限制。对于*maxdict* 的默认值为 4，对于*maxarray* 为 5，对于其他则为 6。

Repr.**maxlong**

表示整数的最大字符数量。数码会从中间被丢弃。默认值为 40。

Repr.**maxstring**

表示字符串的字符数量限制。请注意字符源会使用字符串的“正常”表示形式：如果表示中需要用到转义序列，在缩短表示时它们可能会被破坏。默认值为 30。

Repr.**maxother**

此限制用于控制在*Repr* 对象上没有特定的格式化方法可用的对象类型的大小。它会以类似*maxstring* 的方式被应用。默认值为 20。

Repr.**indent**

如果该属性被设为 None (默认值)，输出将被格式化为不带换行或缩进，像标准的*repr()* 一样。例如:

```
>>> example = [
...     1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
>>> import reprlib
>>> aRepr = reprlib.Repr()
>>> print(aRepr.repr(example))
[1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
```

如果*indent* 被设为一个字符串，每个递归层级将放在单独行中，并用该字符串来缩进:

```
>>> aRepr.indent = '-->'
>>> print(aRepr.repr(example))
[
-->1,
-->'spam',
-->{
-->-->'a': 2,
-->-->'b': 'spam eggs',
-->-->'c': {
-->-->-->3: 4.5,
-->-->-->6: [],
-->-->},
-->},
-->'ham',
]
```

将*indent* 设为一个正整数时其行为与设为相应数量的空格是相同的:

```
>>> aRepr.indent = 4
>>> print(aRepr.repr(example))
[
    1,
    'spam',
    {
        'a': 2,
        'b': 'spam eggs',
        'c': {
            3: 4.5,
            6: [],
        },
    },
    'ham',
]
```

Added in version 3.12.

Repr.**repr**(*obj*)

内置*repr()* 的等价形式，它使用实例专属的格式化。

Repr.**repr1**(*obj*, *level*)

供*repr()* 使用的递归实现。此方法使用 *obj* 的类型来确定要调用哪个格式化方法，并传入 *obj* 和 *level*。类型专属的方法应当调用*repr1()* 来执行递归格式化，在递归调用中使用 `level - 1` 作为 *level* 的值。

Repr.**repr_TYPE**(*obj*, *level*)

特定类型的格式化方法会被实现为基于类型名称来命名的方法。在方法名称中，**TYPE** 会被替换为 `'_'.join(type(obj).__name__.split())`。对这些方法的分派会由*repr1()* 来处理。需要对值进行递归格式化的类型专属方法应当调用 `self.repr1(subobj, level - 1)`。

8.13.2 子类化 Repr 对象

通过*Repr.repr1()* 使用动态分派允许*Repr* 的子类添加额外内置对象类型的支持，或是修改对已支持类型的处理。这个例子演示了如何添加对文件对象的特殊支持：

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))         # 打印 '<stdin>'
```

```
<stdin>
```

8.14 enum --- 对枚举的支持

Added in version 3.4.

源代码： Lib/enum.py

Important

此页面仅包含 API 参考信息。教程信息和更多高级用法的讨论，请参阅

- 基础教程
- 进阶教程
- 枚举指南

一个枚举:

- 是绑定到唯一值的符号名称（成员）集合
- 可以被执行迭代以按定义顺序返回其规范的（即非别名的）成员
- 使用 *调用* 语法按值返回成员
- 使用 *索引* 语法按名称返回成员

枚举是通过使用 `class` 语法或是通过使用函数调用语法来创建的:

```
>>> from enum import Enum

>>> # 类语法
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # 函数语法
>>> Color = Enum('Color', [('RED', 1), ('GREEN', 2), ('BLUE', 3)])
```

虽然我们可以使用 `class` 语法来创建枚举，但枚举并不是常规的 Python 类。请参阅 枚举有什么不同？了解更多细节。

备注

命名法

- 类 `Color` 是一个 枚举（或 *enum* ）
- 属性 `Color.RED` 、`Color.GREEN` 等是 枚举成员（或 *members* ）并且在功能上是常量。
- 枚举成员有 *names* 和 *values* (`Color.RED` 的名称是 `RED`，`Color.BLUE` 的值是 `3`，等等)

8.14.1 模块内容

EnumType

The `type` for Enum and its subclasses.

Enum

用于创建枚举常量的基类。

IntEnum

用于创建枚举常量的基类，这些常量也是*int* 的子类。(*Notes*)

StrEnum

用于创建枚举常量的基类，这些常量也是*str* 的子类。(*Notes*)

Flag

创建可与位运算符搭配使用，又不会失去*Flag* 成员资格的枚举常量的基类。

IntFlag

创建可与位运算符搭配使用，又不失去*IntFlag* 成员资格的枚举常量的基类。*IntFlag* 成员也是*int* 的子类。(*Notes*)

ReprEnum

由*IntEnum* 、*StrEnum* 和*IntFlag* 用来保持混合类型的*str()* 。

EnumCheck

具有值 `CONTINUOUS`、`NAMED_FLAGS` 和 `UNIQUE` 的枚举，用于*verify()* 以确保给定枚举满足各种约束。

FlagBoundary

具有值 `STRICT` 、`CONFORM` 、`EJECT` 和 `KEEP` 的枚举，允许对枚举中无效值的处理方式进行更细粒度的控制。

EnumDict

一个被用于子类化*EnumType* 的*dict* 子类。

auto

实例被替换为枚举成员的适当值。*StrEnum* 默认为成员名称的小写版本，而其他枚举默认为 1 并由此递增。

property()

允许*Enum* 成员拥有属性而不会与成员名称相冲突。`value` 和 `name` 属性都是以这样的方式实现的。

unique()

确保一个名称只绑定一个值的 Enum 类装饰器。

verify()

检查枚举的用户可选择约束的枚举类装饰器。

member()

使 obj 成为成员。可以用作装饰器。

nonmember()

使 obj 不为成员。可以用作装饰器。

global_enum()

修改枚举的*str()* 和*repr()* 以将其成员显示为属于模块而不是其类，并将枚举成员导出到全局命名空间。

show_flag_values()

返回标志中包含的所有二次幂整数的列表。

Added in version 3.6: Flag, IntFlag, auto

Added in version 3.11: StrEnum, EnumCheck, ReprEnum, FlagBoundary, property, member, nonmember, global_enum, show_flag_values

Added in version 3.13: EnumDict

8.14.2 数据类型

class enum.**EnumType**

EnumType 是 *enum* 枚举的*metaclass* 。可以对 *EnumType* 进行子类化——有关详细信息，请参阅 Subclassing EnumType。

EnumType 负责在最终的 *enum* 上设置正确的 __repr__(), __str__(), __format__(), and __reduce__() 方法，以及创建枚举成员，正确处理重复项，提供对枚举类的迭代等。

__call__(*cls*, *value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

此方法以两种不同的方式调用：

- 查找现有成员：

  cls
  被调用的枚举类。

  value
  要查找的值。

- 使用 cls 枚举创建新枚举（仅当现有枚举没有任何成员时）：

  cls
  被调用的枚举类。

  value
  要创建的新枚举的名称。

  names
  新枚举成员的名称/值。

  module -- 模块
  在其中创建新枚举的模块的名称。

  qualname
  可以找到此枚举的模块中的实际位置。

  type -- 类型
  新枚举的混合类型。

start
枚举的第一个整数值（由*auto*使用）。

边界
如何处理来自位操作的超出范围的值（仅限*Flag*）。

__contains__(*cls*, *member*)

如果成员属于 “cls“ 则返回 “True“

```
>>> some_var = Color.RED
>>> some_var in Color
True
>>> Color.RED.value in Color
True
```

在 3.12 版本发生变更: 在 Python 3.12 之前，如果在包含检测中使用了非枚举成员则会引发 TypeError。

__dir__(*cls*)

返回 ['__class__', '__doc__', '__members__', '__module__'] 和 *cls* 中的成员名称

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__contains__', '__doc__', '__getitem__', '__
↪init_subclass__', '__iter__', '__len__', '__members__', '__module__', '__name__',
↪ '__qualname__']
```

__getitem__(*cls*, *name*)

返回 *cls* 中匹配 *name* 的 Enum 成员，或者引发*KeyError*:

```
>>> Color['BLUE']
<Color.BLUE: 3>
```

__iter__(*cls*)

按定义顺序返回 *cls* 中的每个成员:

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]
```

__len__(*cls*)

返回 *cls* 中成员的数量:

```
>>> len(Color)
3
```

__members__

返回一个从每个枚举名称到其成员的映射，包括别名

__reversed__(*cls*)

按定义的逆序返回 *cls* 中的每个成员:

```
>>> list(reversed(Color))
[<Color.BLUE: 3>, <Color.GREEN: 2>, <Color.RED: 1>]
```

_add_alias_()

增加一个新名称作为现有成员的别名。如果该名称已被分配给不同的成员则会引发*NameError*。

_add_value_alias_()

增加一个新值作为现有成员的别名。如果该值已经链接到不同的成员则会引发*ValueError*。

Added in version 3.11: 在 3.11 之前 EnumType 被称为 EnumMeta，该名称作为别名仍然可用。

class enum.**Enum**

Enum 是所有 *enum* 枚举的基类。

name

用于定义 Enum 成员的名称:

```
>>> Color.BLUE.name
'BLUE'
```

value

赋给 Enum 成员的值:

```
>>> Color.RED.value
1
```

成员的值，可在*__new__()* 中设置。

> **备注**
>
> Enum 成员值
>
> 成员值可以为任意类型: *int*, *str* 等等。如果具体的值不重要则你可以使用*auto* 实例这将为你选择一个适当的值。详情参见*auto*。
>
> 虽然可以使用可变/不可哈希的值，比如*dict*, *list* 或是可变的*dataclass*，但它们在创建期间会产生基于枚举中可变/不可哈希的值数量的指数级性能影响。

name

成员的名称。

value

成员的值，可在*__new__()* 中设置。

order

已不再使用，保留以便向下兼容。(类属性，在类创建期间移除)。

ignore

ignore 仅在创建期间使用并会在创建完成后立即从枚举中移除。

ignore 是由不会被作为成员的名称组成的列包，并且这些名称还将从完成的枚举中移除。请参阅 TimePeriod 获取示例。

__dir__(*self*)

返回 ['__class__', '__doc__', '__module__', 'name', 'value'] 以及在 *self.__class__* 上定义的任何公有方法:

```
>>> from datetime import date
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     @classmethod
...     def today(cls):
...         print('today is %s' % cls(date.today().isoweekday()).name)
...
>>> dir(Weekday.SATURDAY)
```

(续下页)

(接上页)

```
['__class__', '__doc__', '__eq__', '__hash__', '__module__', 'name', 'today',
↪'value']
```

_generate_next_value_(*name*, *start*, *count*, *last_values*)

name
定义的成员名称（例如'RED'）。

start
Enum 的起始值；默认为 1。

count
当前定义的成员数量，不包括这一个。

last_values
由前面的值组成的列表。

一个用来确定由 *auto* 所返回的下一个值的 静态方法:

```
>>> from enum import auto
>>> class PowersOfThree(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return 3 ** (count + 1)
...     FIRST = auto()
...     SECOND = auto()
...
>>> PowersOfThree.SECOND.value
9
```

__init__(*self*, **args*, ***kwds*)

在默认情况下，将不做任何事。如果在成员赋值时给出了多个值，这些值将成为传给 `__init__` 的单独参数；例如

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1, 'Mon'
```

`Weekday.__init__()` 将以 `Weekday.__init__(self, 1, 'Mon')` 的形式被调用

__init_subclass__(*cls*, ***kwds*)

一个用来进一步配置后续子类的 类方法。在默认情况下，将不做任何事。

missing(*cls*, *value*)

一个用来查找不存在于 *cls* 中的值的 类方法。在默认情况下它将不做任何事，但可以被重写以实现自定义的搜索行为:

```
>>> from enum import StrEnum
>>> class Build(StrEnum):
...     DEBUG = auto()
...     OPTIMIZED = auto()
...     @classmethod
...     def _missing_(cls, value):
...         value = value.lower()
...         for member in cls:
...             if member.value == value:
...                 return member
...         return None
...
>>> Build.DEBUG.value
'debug'
```

(续下页)

(接上页)

```
>>> Build('deBUG')
<Build.DEBUG: 'debug'>
```

__new__(*cls*, **args*, ***kwds*)

在默认情况下，将不会存在。如果指定，则或是在枚举类定义中或是在混入类定义中 (比如 `int`)，在成员赋值时给出的所有值都将被传递；例如

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     TWENTYSIX = '1a', 16
```

`int('1a', 16)` 调用的结果和成员的值 `26`。

> **备注**
>
> 当编写自定义的 `__new__` 时，不要使用 `super().__new__` -- 而要调用适当的 `__new__`。

__repr__(*self*)

返回用于 *repr()* 调用的字符串。在默认情况下，将返回 *Enum* 名称、成员名称和值，但也可以被重写:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __repr__(self):
...         cls_name = self.__class__.__name__
...         return f'{cls_name}.{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(OtherStyle.ALTERNATE, 'OtherStyle.ALTERNATE', 'OtherStyle.ALTERNATE')
```

__str__(*self*)

返回用于 *str()* 调用的字符串。在默认情况下，返回 *Enum* 名称和成员名称，但也可以被重写:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __str__(self):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'ALTERNATE', 'ALTERNATE')
```

__format__(*self*)

返回用于 *format()* 和 *f-string* 调用的字符串。在默认情况下，将返回*__str__()* 的返回值，但也可以被重写:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __format__(self, spec):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'OtherStyle.ALTERNATE', 'ALTERNATE')
```

> **备注**
>
> 将*auto* 用于*Enum* 将得到递增的整数值，从 1 开始。

在 3.12 版本发生变更: 增加了 enum-dataclass-support

class enum.**IntEnum**

IntEnum 和*Enum* 是一样的，但其成员还属于整数并可被用在任何可以使用整数的地方。如果对一个 *IntEnum* 成员执行整数运算，结果值将失去其枚举状态。

```
>>> from enum import IntEnum
>>> class Number(IntEnum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...
>>> Number.THREE
<Number.THREE: 3>
>>> Number.ONE + Number.TWO
3
>>> Number.THREE + 5
8
>>> Number.THREE == 3
True
```

> **备注**
>
> 将*auto* 用于*IntEnum* 将得到递增的整数值，从 1 开始。

在 3.11 版本发生变更: `__str__()` 现在是 `int.__str__()` 以更好地支持 现有常量的替换应用场景。出于同样的原因 `__format__()` 也已经是 `int.__format__()`。

class enum.**StrEnum**

`StrEnum` 和*Enum* 是一样的，但其成员还属于字符串并可被用在任何可以使用字符串的地方。如果对一个 *StrEnum* 成员执行字符串操作其结果值将不再是该枚举的一部分。

> **备注**
>
> 在标准库中有些地方会检查是否是真正的*str* 而不是*str* 的子类 (例如使用 `type(unknown) == str` 而不是 `isinstance(unknown, str)`)，在这些地方你将需要使用 `str(StrEnum.member)`。

> **备注**
>
> 将*auto* 用于*StrEnum* 将得到小写形式的成员名称字符串值。

> **备注**
>
> `__str__()` 是 `str.__str__()` 以更好地支持 现有常量的替换应用场景。出于同样的原因 `__format__()` 也是 `str.__format__()`。

Added in version 3.11.

class enum.**Flag**

`Flag` 与*Enum* 的相同，但其成员支持按位运算符 `&` (*AND*), `|` (*OR*), `^` (*XOR*) 和 `~` (*INVERT*)；这些运算的结果都是枚举成员（的别名）。

__contains__(*self*, *value*)

如果 value 是 self 之中则返回 *True*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> purple = Color.RED | Color.BLUE
>>> white = Color.RED | Color.GREEN | Color.BLUE
>>> Color.GREEN in purple
False
>>> Color.GREEN in white
True
>>> purple in white
True
>>> white in purple
False
```

__iter__(self):

返回所有包含的非别名成员:

```
>>> list(Color.RED)
[<Color.RED: 1>]
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 4>]
```

Added in version 3.11.

__len__(self):

返回旗标中成员的数量:

```
>>> len(Color.GREEN)
1
>>> len(white)
3
```

Added in version 3.11.

__bool__(self):

如果旗标中有成员则返回 *True*，否则返回 *False*:

```
>>> bool(Color.GREEN)
True
>>> bool(white)
True
>>> black = Color(0)
>>> bool(black)
False
```

__or__(*self*, *other*)

返回当前旗标与另一个旗标执行二进制或运算的结果:

```
>>> Color.RED | Color.GREEN
<Color.RED|GREEN: 3>
```

__and__(*self*, *other*)

返回当前旗标与另一个旗标执行二进制与运算的结果:

```
>>> purple & white
<Color.RED|BLUE: 5>
>>> purple & Color.GREEN
<Color: 0>
```

__xor__(*self*, *other*)

返回当前旗标与另一个旗标执行二进制异或运算的结果:

```
>>> purple ^ white
<Color.GREEN: 2>
>>> purple ^ Color.GREEN
<Color.RED|GREEN|BLUE: 7>
```

__invert__(self):

返回 *type(self)* 中所有不在 *self* 中的旗标:

```
>>> ~white
<Color: 0>
>>> ~purple
<Color.GREEN: 2>
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

_numeric_repr_()

用于格式化任何其他未命名数字值的函数。默认为数字值的 repr；常见的选择有*hex()*和*oct()*。

备注

将*auto*用于*Flag*将得到二的整数次方，从 1 开始。

在 3.11 版本发生变更: 零值旗标的 *repr()* 已被修改。现在将是:

```
>>> Color(0)
<Color: 0>
```

class enum.**IntFlag**

IntFlag 与*Flag*相同，但其成员还属于整数类型并能被用于任何可以使用整数的地方。

```
>>> from enum import IntFlag, auto
>>> class Color(IntFlag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> Color.RED & 2
<Color: 0>
>>> Color.RED | 2
<Color.RED|GREEN: 3>
```

如果对一个 *IntFlag* 成员执行任何整数运算，结果将不再是一个 *IntFlag*:

```
>>> Color.RED + 2
3
```

如果对一个 *IntFlag* 成员执行*Flag*操作并且:

- 结果是一个合法的 *IntFlag*: 将返回一个 *IntFlag*
- 其结果将不是合法的 *IntFlag*: 具体结果将取决于 *FlagBoundary* 设置

未命名的零值旗标 *repr()* 已被修改。现在将是:

```
>>> Color(0)
<Color: 0>
```

备注

将 *auto* 用于 *IntFlag* 将得到二的整数次方，从 1 开始。

在 3.11 版本发生变更: `__str__()` 现在是 `int.__str__()` 以更好地支持现有常量的替换应用场景。出于同样的原因 `__format__()` 也已经是 `int.__format__()`。

对一个 `IntFlag` 的反转现在将返回一个等于不在给定旗标中的所有旗标的并集的正值，而非一个负值。这与现有 *Flag* 的行为相匹配。

class enum.**ReprEnum**

ReprEnum 将使用 *Enum* 的 *repr()*，但使用混入数据类型的 *str()*:

- `int.__str__()` 用于 *IntEnum* 和 *IntFlag*
- `str.__str__()` 用于 *StrEnum*

从 ReprEnum 继承以存放混入数据类型的 *str()* / *format()* 而不是使用 *Enum* 默认的 *str()*。

Added in version 3.11.

class enum.**EnumCheck**

EnumCheck 包含由 *verify()* 装饰器用来确保各种约束的选项；失败的约束将导致 *ValueError*。

UNIQUE

确保每个值只有一个名称:

```
>>> from enum import Enum, verify, UNIQUE
>>> @verify(UNIQUE)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     CRIMSON = 1
Traceback (most recent call last):
...
ValueError: aliases found in <enum 'Color'>: CRIMSON -> RED
```

CONTINUOUS

确保在最低值成员和最高值成员之间没有缺失的值:

```
>>> from enum import Enum, verify, CONTINUOUS
>>> @verify(CONTINUOUS)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 5
Traceback (most recent call last):
...
ValueError: invalid enum 'Color': missing values 3, 4
```

NAMED_FLAGS

确保任何旗标分组/掩码只包含已命名的旗标 -- 在值是明确指定而不是由 *auto()* 生成时将很有用处:

```
>>> from enum import Flag, verify, NAMED_FLAGS
>>> @verify(NAMED_FLAGS)
... class Color(Flag):
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     WHITE = 15
...     NEON = 31
Traceback (most recent call last):
...
ValueError: invalid Flag 'Color': aliases WHITE and NEON are missing combined␣
↪values of 0x18 [use enum.show_flag_values(value) for details]
```

备注

CONTINUOUS 和 NAMED_FLAGS 被设计用于配合整数值成员。

Added in version 3.11.

class enum.**FlagBoundary**

FlagBoundary 控制在*Flag* 及其子类中如何处理超出范围的值。

STRICT

超出范围的值将导致引发*ValueError*。这是*Flag* 的默认设置:

```
>>> from enum import Flag, STRICT, auto
>>> class StrictFlag(Flag, boundary=STRICT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> StrictFlag(2**2 + 2**4)
Traceback (most recent call last):
...
ValueError: <flag 'StrictFlag'> invalid value 20
    given 0b0 10100
  allowed 0b0 00111
```

CONFORM

超出范围的值将导致无效的值被移除，保留有效的*Flag* 值:

```
>>> from enum import Flag, CONFORM, auto
>>> class ConformFlag(Flag, boundary=CONFORM):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> ConformFlag(2**2 + 2**4)
<ConformFlag.BLUE: 4>
```

EJECT

超出范围的值将失去其*Flag* 成员资格并转换为*int*。

```
>>> from enum import Flag, EJECT, auto
>>> class EjectFlag(Flag, boundary=EJECT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
```

(续下页)

(接上页)

```
>>> EjectFlag(2**2 + 2**4)
20
```

KEEP

超出范围的值将被保留，*Flag* 成员资格也将被保留。这是*IntFlag* 的默认设置:

```
>>> from enum import Flag, KEEP, auto
>>> class KeepFlag(Flag, boundary=KEEP):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> KeepFlag(2**2 + 2**4)
<KeepFlag.BLUE|16: 20>
```

Added in version 3.11.

class enum.**EnumDict**

EnumDict 是一个用作定义枚举类的命名空间的*dict* 子类 (参见 prepare)。它被对外公开以允许*EnumType* 的子类具有高级行为如每个成员可包含多个值。它在调用时应当传入被创建的枚举类的名称，否则私有名称和内部类将无法被正确地处理。

请注意只有*MutableMapping* 接口 (__setitem__() 和*update()*) 会被重写。有可能使用其他 dict 操作如 |= 来绕过此项检查。

member_names

由成员名称组成的列表。

Added in version 3.13.

支持的 __dunder__ 名称

__members__ 是由 member_name:member 条目组成的只读有序映射。它只在类上可用。

如果指定了*__new__()*，它必须创建并返回枚举成员；相应地设置成员的 _value_ 也是一个很好的主意。一旦所有成员都创建完成它将不再被使用。

支持的 _sunder_ 名称

- *_add_alias_()* -- 添加一个新名称作为现有成员的别名。
- *_add_value_alias_()* -- 添加一个新值作为现有成员的别名。
- *_name_* -- 成员的名称
- *_value_* -- 成员的值；可在 __new__ 中设置
- *_missing_()* -- 当未找到某个值时所使用的查找函数；可被重写
- *_ignore_* -- 一个名称列表，可以为*list* 或*str*，它不会被转化为成员，并将从最终类中移除
- *_order_* -- 已不再使用，保留以便向下兼容（类属性，在类创建期间移除）
- *_generate_next_value_()* -- 用于为枚举成员获取适当的值；可被重写

> **备注**
>
> 对于标准的*Enum* 类来说下一个被选择的值将是已有的最高值加一。
>
> 对于*Flag* 类来说下一个选择的值将是下一个最高的二的幂数。

- 虽然 _sunder_ 名称通常被保留用于*Enum* 类的后续开发因而不可被使用，但有一些则被显式地允许：
  - _repr_* (例如 _repr_html_)，用于 IPython 的丰富显示

Added in version 3.6: _missing_, _order_, _generate_next_value_

Added in version 3.7: _ignore_

Added in version 3.13: _add_alias_, _add_value_alias_, _repr_*

8.14.3 工具与装饰器

class enum.**auto**

auto 可被用来替换某个值。如果使用，*Enum* 机制将调用一个*Enum* 的*_generate_next_value_()* 来获取适当的值。对于*Enum* 和*IntEnum* 这个适当的值将为最后的值加一；对于*Flag* 和*IntFlag* 它将为首个大于最高值的二的整数次方；对于*StrEnum* 它将为成员名称的小写版本。如果将 *auto()* 与手动指定的值混用则必须十分小心。

auto 实际仅会在赋值操作的最高层级上被解析：

- FIRST = auto() 将是可用的 (auto() 会被替换为 1);
- SECOND = auto(), -2 将是可用的 (auto 会被替换为 2，因此将使用 2, -2 来创建 SECOND 枚举成员；
- THREE = [auto(), -3] 将 不可用 (<auto instance>, -3 将被用来创建 THREE 枚举成员)

在 3.11.1 版本发生变更: 在之前的版本中，auto() 必须为赋值行中唯一的内容才是可用的。

_generate_next_value_ 可以被重写以便自定义 *auto* 所使用的值。

> **备注**
>
> 在 3.13 中默认的 _generate_next_value_ 将总是返回最高成员值递增 1 的结果，并且如果有任何成员为不兼容的类型则将失败。, and will fail if any member is an incompatible type.

@enum.**property**

一个类似于内置 *property* 的装饰器，但是专用于枚举。它允许成员属性具有与成员自身相同的名称。

> **备注**
>
> *property* 和成员必须在单独的类中定义；例如 *value* 和 *name* 属性是在 *Enum* 类中定义，而 *Enum* 的子类可以定义名称为 value 和 name 的成员。

Added in version 3.11.

@enum.**unique**

一个专用于枚举的 class 装饰器。它将搜索一个枚举的*__members__*，收集它所找到的任何别名；如果找到了任何别名则会引发*ValueError* 并附带详情：

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
```

(续下页)

(接上页)

```
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

@enum.**verify**

一个专用于枚举的 class 装饰器。将使用来自*EnumCheck* 的成员指明应当在被装饰的枚举上检查哪些约束。

Added in version 3.11.

@enum.**member**

一个在枚举中使用的装饰器：它的目标将成为一个成员。

Added in version 3.11.

@enum.**nonmember**

一个在枚举中使用的装饰器：它的目标将不会成员一个成员。

Added in version 3.11.

@enum.**global_enum**

一个修改枚举的*str()* 和*repr()* 来将其成员显示为属于模块而不是类的装饰器。应当仅在枚举成员被导出到模块全局命名空间时（请参看*re.RegexFlag* 获取示例）使用。

Added in version 3.11.

enum.**show_flag_values**(*value*)

返回旗标 *value* 中包含的所有二的整数次幂的列表。

Added in version 3.11.

8.14.4 备注

IntEnum, *StrEnum* 和*IntFlag*

这三个枚举类型被设计用来快速替代现有的基于整数和字符串的值；为此，它们都有额外的限制:

- __str__ 使用枚举成员的值而不是名称
- __format__，因为它使用了 __str__，也将使用枚举成员的值而不是其名称

如果你不需要/希望有这些限制，你可以通过自行混入 int 或 str 类型来创建你自己的基类:

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     pass
```

或者你也可以在你的枚举中重新赋值适当的*str()* 等:

```
>>> from enum import Enum, IntEnum
>>> class MyIntEnum(IntEnum):
...     __str__ = Enum.__str__
```

8.15 graphlib --- 操作类似图的结构的功能

源代码: Lib/graphlib.py

---

class graphlib.**TopologicalSorter**(*graph=None*)

提供以拓扑方式对由*hashable* 节点组成的图进行排序的功能。

拓扑排序是指图中顶点的线性排序，使得对于每条从顶点 u 到顶点 v 的有向边 u -> v，顶点 u 都排在顶点 v 之前。例如，图的顶点可以代表要执行的任务，而边代表某一个任务必须在另一个任务之前执行的约束条件；在这个例子中，拓扑排序只是任务的有效序列。完全拓扑排序当且仅当图不包含有向环，也就是说为有向无环图时，完全拓扑排序才是可能的。

如果提供了可选的 *graph* 参数则它必须为一个表示有向无环图的字典，其中的键为节点而值为包含图中该节点的所有上级节点（即具有指向键中的值的边的节点）的可迭代对象。额外的节点可以使用*add()* 方法添加到图中。

在通常情况下，对给定的图执行排序所需的步骤如下:

- 通过可选的初始图创建一个*TopologicalSorter* 的实例。
- 添加额外的节点到图中。
- 在图上调用*prepare()*。
- 当*is_active()* 为 True 时，迭代*get_ready()* 所返回的节点并加以处理。完成处理后在每个节点上调用*done()*。

在只需要对图中的节点进行立即排序并且不涉及并行性的情况下，可以直接使用便捷方法*TopologicalSorter.static_order()*:

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

这个类被设计用来在节点就绪时方便地支持对其并行处理。例如:

```
topological_sorter = TopologicalSorter()

# 添加节点到 'topological_sorter'...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # 工作线程或进程接受节点在
        # 'task_queue' 队列上工作和移除
        task_queue.put(node)

    # 当一个节点的工作完成时，工作线程或进程就会将节点放入
    # 'finalized_tasks_queue' 这样我们就可以继续操作其他节点。
    # 'is_active()' 的定义能够确保这一点，这时，至少有一个节点
    # 已被放置在 'task_queue' 上而尚未被传给 'done()'，因此这个
    # 阻塞中的 'get()' 必须（最终）成功。 在调用 'done()' 之后，
    # 我们将环回至再次调用 'get_ready()'，因此只要逻辑上可能
    # 就应尽快将新近可用的节点放入 'task_queue' 中。
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)

将一个新节点及其上级节点添加到图中。*node* 和 *predecessors* 中的所有元素都必须是*hashable*。

如果附带相同的节点参数多次调用，则依赖项的集合将为所有被传入依赖项的并集。

可以添加不带依赖项的节点 (即不提供 *predecessors*) 或者重复提供依赖项。如果有先前未提供的节点包含在 *predecessors* 中则它将被自动添加到图中并且不带自己的上级节点。

如果在*prepare()* 之后被调用则会引发*ValueError*。

prepare()

将图标记为已完成并检查图中是否存在环。如何检测到任何环，则将引发*CycleError*，但*get_ready()* 仍可被用来获取尽可能多的节点直到环阻塞了操作过程。在调用此函数后，图将无法再修改，因此不能再使用*add()* 添加更多的节点。

is_active()

如果可以取得更多进展则返回 `True`，否则返回 `False`。如果环没有阻塞操作，并且还存在尚未被*TopologicalSorter.get_ready()* 返回的已就绪节点或者已标记为*TopologicalSorter.done()* 的节点数量少于已被*TopologicalSorter.get_ready()* 所返回的节点数量则还可以取得进展。

该类的 `__bool__()` 方法要使用此函数，因此除了:

```
if ts.is_active():
    ...
```

可能会简单地执行:

```
if ts:
    ...
```

如果之前未调用*prepare()* 就调用此函数则会引发*ValueError*。

done(**nodes*)

将*TopologicalSorter.get_ready()* 所返回的节点集合标记为已处理，解除对 *nodes* 中每个节点的后续节点的阻塞以便在将来通过对*TopologicalSorter.get_ready()* 的调用来返回它们。

如果 *nodes* 中的任何节点已经被之前对该方法的调用标记为已处理或者如果未通过使用*TopologicalSorter.add()* 将一个节点添加到图中，如果未调用*prepare()* 即调用此方法或者如果节点尚未被*get_ready()* 所返回则将引发*ValueError*。

get_ready()

返回由所有已就绪节点组成的 `tuple`。初始状态下它将返回所有不带上级节点的节点，并且一旦通过调用*TopologicalSorter.done()* 将它们标记为已处理，之后的调用将返回所有上级节点已被处理的新节点。一旦无法再取得进展，则会返回空元组。

如果之前未调用*prepare()* 就调用此函数则会引发*ValueError*。

static_order()

返回一个迭代器，它将按照拓扑顺序来迭代所有节点。当使用此方法时，*prepare()* 和*done()* 不应被调用。此方法等价于:

```
def static_order(self):
    self.prepare()
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

所返回的特定顺序可能取决于条目被插入图中的顺序。例如:

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

这是由于实际上"0" 和"2" 在图中的级别相同（它们将在对*get_ready()* 的同一次调用中被返回）并且它们之间的顺序是由插入顺序决定的。

如果检测到任何环，则将引发*CycleError*。

Added in version 3.9.

8.15.1 异常

graphlib 模块定义了以下异常类:

exception graphlib.**CycleError**

ValueError 的子类，当特定的图中存在环时将由*TopologicalSorter.prepare()* 引发。如果存在多个环，则将只报告其中一个未定义的选项并将其包括在异常中。

检测到的环可通过异常实例的*args* 属性的第二个元素访问，它由一个节点列表组成，在图中每个节点都是列表中下一个节点的直接上级节点。在报告的列表中，开头和末尾的节点将是同一对象，以表明它是一个环。

CHAPTER 9

数字和数学模块

本章介绍的模块提供与数字和数学相关的函数和数据类型。*numbers* 模块定义了数字类型的抽象层次结构。*math* 和 *cmath* 模块包含浮点数和复数的各种数学函数。*decimal* 模块支持使用任意精度算术的十进制数的精确表示。

本章包含以下模块的文档:

9.1 numbers --- 数字抽象基类

源代码: Lib/numbers.py

numbers 模块 (**PEP 3141**) 定义了数字抽象基类 的层级结构，其中逐级定义了更多的操作。此模块中定义的类型都不可被实例化。

class numbers.**Number**

数字的层次结构的基础。如果你只想确认参数 *x* 是不是数字而不关心其类型，则使用 `isinstance(x, Number)`。

9.1.1 数字的层次

class numbers.**Complex**

这个类型的子类描述了复数并包括了适用于内置 *complex* 类型的操作。这些操作有: 转换为 *complex* 和 *bool*, *real*, *imag*, +, -, *, /, **, *abs()*, *conjugate()*, == 以及 !=。除 - 和 != 之外所有操作都是抽象的。

real

抽象的。得到该数字的实数部分。

imag

抽象的。得到该数字的虚数部分。

abstractmethod conjugate()

抽象的。返回共轭复数。例如 `(1+3j).conjugate() == (1-3j)`。

class numbers.**Real**

相对于*Complex*，Real 加入了只适用于实数的操作。

简单的说，它们是：转化至*float*，*math.trunc()*、*round()*、*math.floor()*、*math.ceil()*、*divmod()*、//、%、<、<=、>、和 >=。

实数同样默认支持*complex()*、*real*、*imag* 和*conjugate()*。

class numbers.**Rational**

子类型*Real* 并加入*numerator* 和*denominator* 两种特征属性。它还为*float()* 提供了默认值。

numerator 和*denominator* 值应为*Integral* 的实例并且应当是具有*denominator* 正值的最低项。

numerator

抽象的。

denominator

抽象的。

class numbers.**Integral**

子类型*Rational* 还增加了到*int* 的转换操作。为*float()*，*numerator* 和*denominator* 提供了默认支持。为*pow()* 方法增加了求余和按位字符串运算的抽象方法: <<, >>, &, ^, |, ~。

9.1.2 有关类型实现的注释

实现方应当注意让相等的数值保证相等并使它们哈希运算的结果值相同。当相互比较的实数属于两个不同的扩展模块时可能会有微妙的差异。举例来说，*fractions.Fraction* 是这样实现*hash()* 的:

```
def __hash__(self):
    if self.denominator == 1:
        # 正确获取整数。
        return hash(self.numerator)
    # 高开销的检查，但肯定正确。
    if self == float(self):
        return hash(float(self))
    else:
        # 使用元组的哈希值来避免简单分数的
        # 高碰撞率。
        return hash((self.numerator, self.denominator))
```

加入更多数字的 ABC

当然，这里有更多支持数字的 ABC，如果不加入这些，就将缺少层次感。你可以用如下方法在*Complex* 和*Real* 中加入 MyFoo:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

实现算术运算

我们想要实现算术运算，因此混合模式运算要么调用一个开发者知道两个参数类型的实现，要么将两个参数转换为最接近的内置类型再执行运算。对于*Integral* 的子类，这意味着 __add__() 和 __radd__() 应当被定义为:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
```

(续下页)

(接上页)

```
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Complex 有 5 种不同的混合类型的操作。我将上面提到的所有代码作为“模板”称作 `MyIntegral` 和 `OtherTypeIKnowAbout`。`a` 是*Complex* 的子类型 `A` 的实例 (`a : A <: Complex`)，同时 `b : B <: Complex`。我将要计算 `a + b`:

1. 如果 `A` 定义了接受 `b` 的 `__add__()`，一切都没有问题。
2. 如果 `A` 回退至基础代码，它将返回一个来自 `__add__()` 的值，我们就没有机会为 `B` 定义更加智能的 `__radd__()`，因此基础代码应当从 `__add__()` 返回*NotImplemented*。(或者 `A` 可能完全不实现 `__add__()`。)
3. 那么 `B` 的 `__radd__()` 将有机会发挥作用。如果它接受 `a`，一切都没有问题。
4. 如果没有成功回退到模板，就没有更多的方法可以去尝试，因此这里将使用默认的实现。
5. 如果 `B <: A`，Python 在 `A.__add__` 之前尝试 `B.__radd__`。这是可行的，是通过对 `A` 的认识实现的，因此这可以在交给*Complex* 处理之前处理这些实例。

如果 `A <: Complex` 和 `B <: Real` 没有共享任何其他信息，那么内置*complex* 的共享操作就是最适当的，两个 `__radd__()` 都将应用该操作，因此 `a+b == b+a`。

由于对任何一直类型的大部分操作是十分相似的，可以定义一个帮助函数，即一个生成后续或相反的实例的生成器。例如，使用*fractions.Fraction* 如下:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, Complex):
            return fallback_operator(complex(a), complex(b))
        else:
```

(续下页)

(接上页)

```
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math --- 数学函数

该模块提供了对 C 标准定义的数学函数的访问。

这些函数不适用于复数；如果你需要计算复数，请使用*cmath* 模块中的同名函数。将支持计算复数的函数区分开的目的，在于大多数用户并不愿意为了理解复数而去学习太多数学知识。得到一个异常而不是一个复数结果能让复数当作参数的情况更早被监测到，进而程序员可以第一时间调查其产生的原因。

该模块提供了以下函数。除非另有明确说明，否则所有返回值均为浮点数。

数论函数	
comb(n, k)	不重复且无顺序地从 *n* 项中选择 *k* 项的方式总数
factorial(n)	*n* 的阶乘
*gcd(*integers)*	整数参数的最大公约数
isqrt(n)	非负整数 *n* 的整数平方根
*lcm(*integers)*	整数参数的最小公倍数
perm(n, k)	无重复且无顺序地从 *n* 项中选择 *k* 项的方式总数
浮点算数	
ceil(x)	*x* 向上取整，即大于等于 *x* 的最小整数。
fabs(x)	*x* 的绝对值
floor(x)	*x* 向下取整，即小于等于 *x* 的最大整数。
fma(x, y, z)	合并乘法加法运算: `(x * y) + z`
fmod(x, y)	除法运算 `x / y` 的余数
modf(x)	*x* 的小数和整数部分
remainder(x, y)	*x* 对于 *y* 的余数
trunc(x)	*x* 的整数部分
浮点操作函数	
copysign(x, y)	使用 *y* 的符号放大 *x* 的（绝对值）
frexp(x)	*x* 的尾数和指数
isclose(a, b, rel_tol, abs_tol)	检查 *a* 和 *b* 的值是否彼此接近
isfinite(x)	检查 *x* 是否不为无穷大也不为 NaN
isinf(x)	检查 *x* 是否为正或负无穷大
isnan(x)	检查 *x* 是否为 NaN (非数字)
ldexp(x, i)	`x * (2**i)`，即函数*frexp()* 的反函数
nextafter(x, y, steps)	从 *x* 往 *y* 方向的步数浮点值 *steps*
ulp(x)	*x* 的最小有效比特位的值
幂、指数和对数函数	
cbrt(x)	*x* 的立方根

续下页

表 1 – 接上页

exp(x)	*e* 的 *x* 次幂
exp2(x)	*2* 的 *x* 次幂
expm1(x)	*e* 的 *x* 次幂，减 1
log(x, base)	*x* 的指定底数 (默认为 *e*) 的对数
log1p(x)	*1+x* 的自然对数 (以 *e* 为底)
log2(x)	*x* 的以 2 为底的对数
log10(x)	*x* 的以 10 为底的对数
pow(x, y)	*x* 的 *y* 次幂
sqrt(x)	*x* 的平方根
加总和乘积函数	
dist(p, q)	以坐标的可迭代对象形式给出的 *p* 和 *q* 两点之间的欧几里得距离
fsum(iterable)	输入的 *iterable* 值的总计
*hypot(*coordinates)*	坐标的可迭代对象的欧几里得范数
prod(iterable, start)	具有 *start* 值的输入 *iterable* 中元素的积
sumprod(p, q)	来自两个可迭代对象 *p* 和 *q* 的值的乘积的总计值。
角度转换	
degrees(x)	将度数值 *x* 从弧度转换为角度
radians(x)	将度数值 *x* 从角度转换为弧度
三角函数	
acos(x)	*x* 的反余弦
asin(x)	*x* 的反正弦
atan(x)	*x* 的反正切
atan2(y, x)	`atan(y / x)`
cos(x)	*x* 的余弦
sin(x)	*x* 的正弦
tan(x)	*x* 的正切
双曲函数	
acosh(x)	*x* 的反双曲余弦
asinh(x)	*x* 的反双曲正弦
atanh(x)	*x* 的反双曲正切
cosh(x)	*x* 的双曲余弦
sinh(x)	*x* 的双曲正弦
tanh(x)	*x* 的双曲正切
特殊函数	
erf(x)	在 *x* 处的 误差函数
erfc(x)	在 *x* 处的 互补误差函数
gamma(x)	在 *x* 处的 伽马函数
lgamma(x)	在 *x* 处的 伽马函数 的绝对值的自然对数
常量	
pi	$\pi = 3.141592...$
e	$e = 2.718281...$
tau	$\tau = 2\pi = 6.283185...$
inf	正无穷
nan	” 非数字” (NaN)

9.2.1 数论函数

`math.`**`comb`**(*n*, *k*)

返回不重复且无顺序地从 *n* 项中选择 *k* 项的方式总数。

当 `k <= n` 时取值为 `n! / (k! * (n - k)!)`；当 `k > n` 时取值为零。

也称为二项式系数，因为它等价于 `(1 + x)**n` 的多项式展开中第 k 项的系数。

如果任一参数不为整数则会引发*TypeError*。如果任一参数为负数则会引发*ValueError*。

Added in version 3.8.

math.**factorial**(*n*)

将 *n* 的阶乘作为整数返回。如果 *n* 不是正数或为负值则会引发*ValueError*。

在 3.10 版本发生变更: 具有整数值的浮点数 (如 `5.0`) 将不再被接受。

math.**gcd**(**integers*)

返回给定的整数参数的最大公约数。如果有一个参数非零，则返回值将是能同时整除所有参数的最大正整数。如果所有参数为零，则返回值为 `0`。不带参数的 `gcd()` 返回 `0`。

Added in version 3.5.

在 3.9 版本发生变更: 添加了对任意数量的参数的支持。之前的版本只支持两个参数。

math.**isqrt**(*n*)

返回非负整数 *n* 的整数平方根。这就是对 *n* 的实际平方根向下取整，或者相当于使得 $a^2 \leq n$ 的最大整数 *a*。

对于某些应用来说，可以更适合取值为使得 $n \leq a^2$ 的最小整数 *a*，或者换句话说就是 *n* 的实际平方根向上取整。对于正数 *n*，这可以使用 `a = 1 + isqrt(n - 1)` 来计算。

Added in version 3.8.

math.**lcm**(**integers*)

返回给定的整数参数的最小公倍数。如果所有参数均非零，则返回值将是为所有参数的整数倍的最小正整数。如果参数之一为零，则返回值为 `0`。不带参数的 `lcm()` 返回 `1`。

Added in version 3.9.

math.**perm**(*n*, *k=None*)

返回不重复且有顺序地从 *n* 项中选择 *k* 项的方式总数。

当 `k <= n` 时取值为 `n! / (n - k)!`；当 `k > n` 时取值为零。

如果 *k* 未指定或为 `None`，则 *k* 默认值为 *n* 并且函数将返回 `n!`。

如果任一参数不为整数则会引发*TypeError*。如果任一参数为负数则会引发*ValueError*。

Added in version 3.8.

9.2.2 浮点算数

math.**ceil**(*x*)

返回 *x* 的向上取整，即大于或等于 *x* 的最小的整数。如果 *x* 不是浮点数，委托给 `x.__ceil__`，它应该返回一个*Integral* 的值。

math.**fabs**(*x*)

返回 *x* 的绝对值。

math.**floor**(*x*)

返回 *x* 的向下取整，小于或等于 *x* 的最大整数。如果 *x* 不是浮点数，则委托给 `x.__floor__`，它应返回一个*Integral* 值。

math.**fma**(*x*, *y*, *z*)

融合的乘法-加法运算。返回 `(x * y) + z`，类似于使用无限精度和取值范围进行计算然后执行一次舍入到 `float` 格式。此运算往往可提供比直接使用表达式 `(x * y) + z` 更高的精确度。

此函数遵循在 IEEE 754 标准中描述的 fusedMultiplyAdd 运算规范。该标准定义了相同场景的统一实现，即 `fma(0, inf, nan)` 和 `fma(inf, 0, nan)`。在这些场景中，`math.fma` 将返回 NaN，且不会引发任何异常。

Added in version 3.13.

math.**fmod**(*x*, *y*)

返回 `x / y` 的以浮点表示的余数，如平台的 C 库函数 `fmod(x, y)` 所定义的。请注意 Python 表达式 `x % y` 可能不会返回相同的结果。C 标准的目的是 `fmod(x, y)` 完全地（在数学概念中；精度无限）等于 `x - n*y` 对于整数 *n* 使得结果具有与 *x* 相同的正负号和小于 `abs(y)` 的量级。Python 的 `x % y` 则返回与具有 *y* 相同的正负号的结果，而对浮点参数来说可能不是完全可计算的。例如，`fmod(-1e-100, 1e100)` 是 `-1e-100`，但 Python 的 `-1e-100 % 1e100` 则是 `1e100-1e-100`，它不能准确表示为一个浮点数，并会舍入为令人惊讶的 `1e100`。出于这个原因，函数*fmod()* 在处理浮点数时通常都是首选，而 Python 的 `x % y` 则在处理整数时是首选。

math.**modf**(*x*)

返回 *x* 的小数和整数部分。两个结果都带有 *x* 的符号并且是浮点数。

请注意*modf()* 具有与它的 C 对应物不同的调用/返回模式：它接受单个参数并返回一对值，而不是通过' 输出形参' 返回它的第二个返回值 (Python 中没有这个概念)。

math.**remainder**(*x*, *y*)

返回 IEEE 754 风格的 *x* 相对于 *y* 的余数。对于有限 *x* 和有限非零 *y*，这是差异 `x - n*y`，其中 `n` 是与商 `x / y` 的精确值最接近的整数。如果 `x / y` 恰好位于两个连续整数之间，则将最接近的 偶数用作 `n`。余数 `r = remainder(x, y)` 因此总是满足 `abs(r) <= 0.5 * abs(y)`。

特殊情况遵循 IEEE 754：特别是 `remainder(x, math.inf)` 对于任何有限 *x* 都是 *x*，而 `remainder(x, 0)` 和 `remainder(math.inf, x)` 引发*ValueError* 适用于任何非 NaN 的 *x*。如果余数运算的结果为零，则该零将具有与 *x* 相同的符号。

在使用 IEEE 754 二进制浮点的平台上，此操作的结果始终可以完全表示：不会引入舍入错误。

Added in version 3.7.

math.**trunc**(*x*)

返回去除小数部分的 *x*，只留下整数部分。这样就可以四舍五入到 0 了：`trunc()` 对于正的 *x* 相当于*floor()*，对于负的 *x* 相当于*ceil()*。如果 *x* 不是浮点数，委托给 `x.__trunc__`，它应该返回一个*Integral* 值。

对于*ceil()*，*floor()* 和*modf()* 函数，请注意 所有足够大的浮点数都是精确整数。Python 浮点数通常不超过 53 位的精度（与平台 C double 类型相同），在这种情况下，任何浮点 *x* 与 `abs(x) >= 2**52` 必然没有小数位。

9.2.3 浮点操作函数

math.**copysign**(*x*, *y*)

返回一个基于 *x* 的绝对值和 *y* 的符号的浮点数。在支持带符号零的平台上，`copysign(1.0, -0.0)` 返回 *-1.0*.

math.**frexp**(*x*)

以 `(m, e)` 对的形式返回 *x* 的尾数和指数。*m* 是一个浮点数，*e* 是一个整数，正好是 `x == m * 2**e`。如果 *x* 为零，则返回 `(0.0, 0)`，否则返回 `0.5 <= abs(m) < 1`。这用于以可移植方式 “分离” 浮点数的内部表示。

注意*frexp()* 具有与它的 C 对应物不同的调用/返回模式：它接受单个参数并返回一对值，而不是通过' 输出形参' 返回它的第二个返回值 (Python 中没有这个概念)。

math.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

若 *a* 和 *b* 的值比较接近则返回 `True`，否则返回 `False`。

两个值是否会被视为相近是根据给定的绝对和相对可接受差异度来确定的。如果未发生错误，结果将为: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`。

rel_tol 是相对容差 -- 它是 *a* 和 *b* 之间的最大允许差值，相对于 *a* 或 *b* 中绝对值较大的一个而言。例如，要设置 5% 的容差，则传入 `rel_tol=0.05`。默认的容差为 `1e-09`，这将确保两个值在大约 9 个十进制数位内是相同的。*rel_tol* 必须为非零值并且小于 `1.0`。

abs_tol 是绝对容差；其默认值为 `0.0` 且必须为非负数。当将 `x` 与 `0.0` 比较时，`isclose(x, 0)` 将按 `abs(x) <= rel_tol * abs(x)` 来计算，对于任何非零的 `x` 和小于 `1.0` 的 *rel_tol* 来说均为 `False`。因此请为该调用添加一个适当的正数 *abs_tol* 参数。

IEEE 754 特殊值 `NaN`，`inf` 和 `-inf` 将根据 IEEE 规则处理。具体来说，`NaN` 不被认为接近任何其他值，包括 `NaN`。`inf` 和 `-inf` 只被认为接近自己。

Added in version 3.5.

> **参见**
>
> **PEP 485** ——用于测试近似相等的函数

`math.isfinite(x)`

如果 *x* 既不是无穷大也不是 NaN，则返回 `True`，否则返回 `False`。（注意 `0.0` 被认为是有限的。）

Added in version 3.2.

`math.isinf(x)`

如果 *x* 是正或负无穷大，则返回 `True`，否则返回 `False`。

`math.isnan(x)`

如果 *x* 是 NaN（不是数字），则返回 `True`，否则返回 `False`。

`math.ldexp(x, i)`

返回 `x * (2**i)`。这基本上是函数*frexp()* 的反函数。

`math.nextafter(x, y, steps=1)`

返回从 *x* 到 *y* 的步数的浮点值 *steps*。

如果 *x* 等于 *y*，则返回 *y*，除非 *steps* 值为零。

示例：

- `math.nextafter(x, math.inf)` 的方向朝上：趋向于正无穷。
- `math.nextafter(x, -math.inf)` 的方向朝下：趋向于负无穷。
- `math.nextafter(x, 0.0)` 趋向于零。
- `math.nextafter(x, math.copysign(math.inf, x))` 趋向于零的反方向。

另请参阅*math.ulp()*。

Added in version 3.9.

在 3.12 版本发生变更: 增加了 *steps* 参数。

`math.ulp(x)`

返回浮点数 *x* 的最小有效比特位的值:

- 如果 *x* 是 NaN (非数字)，则返回 *x*。
- 如果 *x* 为负数，则返回 `ulp(-x)`。
- 如果 *x* 为正数，则返回 *x*。
- 如果 *x* 等于零，则返回 去正规化的可表示最小正浮点数 (小于 正规化的最小正浮点数*sys.float_info.min*)。
- 如果 *x* 等于可表示最大正浮点数，则返回 *x* 的最低有效比特位的值，使得小于 *x* 的第一个浮点数为 `x - ulp(x)`。
- 在其他情况下 (*x* 是一个有限的正数)，则返回 *x* 的最低有效比特位的值，使得大于 *x* 的第一个浮点数为 `x + ulp(x)`。

ULP 即"Unit in the Last Place" 的缩写。

另请参阅*math.nextafter()* 和*sys.float_info.epsilon*。

Added in version 3.9.

9.2.4 幂、指数和对数函数

math.**cbrt**(*x*)

返回 *x* 的立方根。

Added in version 3.11.

math.**exp**(*x*)

返回 *e* 的 *x* 次幂，其中 *e* = 2.718281... 是自然对数的基数。这通常比 `math.e ** x` 或 `pow(math.e, x)` 更精确。

math.**exp2**(*x*)

返回 *2* 的 *x* 次幂。

Added in version 3.11.

math.**expm1**(*x*)

返回 *e* 的 *x* 次方减 1。这里 *e* 是自然对数的底。对于小浮点数 *x*，在 `exp(x) - 1` 中的减法运算可能导致 明显的精度损失；*expm1()* 函数提供了一种以完整精度计算此数量的办法：

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

Added in version 3.2.

math.**log**(*x*[, *base*])

使用一个参数，返回 *x* 的自然对数（底为 *e* ）。

使用两个参数，返回给定的 *base* 的对数 *x* ，计算为 `log(x)/log(base)` 。

math.**log1p**(*x*)

返回 *1+x* 的自然对数（以 *e* 为底）。以对于接近零的 *x* 精确的方式计算结果。

math.**log2**(*x*)

返回 *x* 以 2 为底的对数。这通常比 `log(x, 2)` 更准确。

Added in version 3.3.

> **参见**
>
> *int.bit_length()* 返回表示二进制整数所需的位数，不包括符号和前导零。

math.**log10**(*x*)

返回 *x* 底为 10 的对数。这通常比 `log(x, 10)` 更准确。

math.**pow**(*x*, *y*)

返回 *x* 的 *y* 次幂。特殊情况将尽可能遵循 IEEE 754 标准。具体来说，`pow(1.0, x)` 和 `pow(x, 0.0)` 总是返回 `1.0`，即使当 *x* 为零或 NaN 时也是如此。如果 *x* 和 *y* 均为有限值，*x* 是负数，而 *y* 不是负数则 `pow(x, y)` 将是未定义的，并会引发*ValueError*。

与内置的 `**` 运算符不同，*math.pow()* 将其参数转换为*float* 类型。使用 `**` 或内置的*pow()* 函数来计算精确的整数幂。

在 3.11 版本发生变更: 特殊情况 `pow(0.0, -inf)` 和 `pow(-0.0, -inf)` 已改为返回 `inf` 而不是引发*ValueError*，以便同 IEEE 754 保持一致。

math.**sqrt**(*x*)

返回 *x* 的平方根。

9.2.5 加总和乘积函数

math.**dist**(*p*, *q*)

返回 *p* 与 *q* 两点之间的欧几里得距离，以一个坐标序列（或可迭代对象）的形式给出。两个点必须具有相同的维度。

大致相当于：

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

Added in version 3.8.

math.**fsum**(*iterable*)

返回可迭代对象中的值的精确浮点总计值。通过跟踪多个中间部分和来避免精度损失。

该算法的准确性取决于 IEEE-754 算术保证和舍入模式为半偶的典型情况。在某些非 Windows 版本中，底层 C 库使用扩展精度添加，并且有时可能会使中间和加倍，导致它在最低有效位中关闭。

有关进一步的讨论和两种替代方式，请参阅 ASPN cookbook recipes for accurate floating-point summation。

math.**hypot**(**coordinates*)

返回欧几里得范数，`sqrt(sum(x**2 for x in coordinates))`。这是从原点到坐标给定点的向量长度。

对于一个二维点 `(x, y)`，这等价于使用毕达哥拉斯定义 `sqrt(x*x + y*y)` 计算一个直角三角形的斜边。

在 3.8 版本发生变更: 添加了对 n 维点的支持。之前的版本只支持二维点。

在 3.10 版本发生变更: 改进了算法的精确性，使得最大误差在 1 ulp (最后一位的单位数值) 以下。更为常见的情况是，结果几乎总是能正确地舍入到 1/2 ulp 范围之内。

math.**prod**(*iterable*, *, *start=1*)

计算输入的 *iterable* 中所有元素的积。积的默认 *start* 值为 `1`。

当可迭代对象为空时，返回起始值。此函数特别针对数字值使用，并会拒绝非数字类型。

Added in version 3.8.

math.**sumprod**(*p*, *q*)

两个可迭代对象 *p* 和 *q* 中的值的乘积的总计值。

如果输入值的长度不相等则会引发 *ValueError*。

大致相当于：

```
sum(itertools.starmap(operator.mul, zip(p, q, strict=True)))
```

对于浮点数或混合整数/浮点数的输入，中间的乘积和总计值将使用扩展精度来计算。

Added in version 3.12.

9.2.6 角度转换

math.**degrees**(*x*)

将角度 *x* 从弧度转换为度数。

math.**radians**(*x*)

将角度 *x* 从度数转换为弧度。

9.2.7 三角函数

math.**acos**(*x*)

返回以弧度为单位的 *x* 的反余弦值。结果范围在 0 到 pi 之间。

math.**asin**(*x*)

返回以弧度为单位的 *x* 的反正弦值。结果范围在 -pi/2 到 pi/2 之间。

math.**atan**(*x*)

返回以弧度为单位的 *x* 的反正切值。结果范围在 -pi/2 到 pi/2 之间。.

math.**atan2**(*y*, *x*)

以弧度为单位返回 atan(y / x) 。结果是在 -pi 和 pi 之间。从原点到点 (x, y) 的平面矢量使该角度与正 X 轴成正比。*atan2()* 的点的两个输入的符号都是已知的，因此它可以计算角度的正确象限。例如，atan(1) 和 atan2(1, 1) 都是 pi/4 ，但 atan2(-1, -1) 是 -3*pi/4 。

math.**cos**(*x*)

返回 *x* 弧度的余弦值。

math.**sin**(*x*)

返回 *x* 弧度的正弦值。

math.**tan**(*x*)

返回 *x* 弧度的正切值。

9.2.8 双曲函数

双曲函数 是基于双曲线而非圆来对三解函数进行的模拟。

math.**acosh**(*x*)

返回 *x* 的反双曲余弦值。

math.**asinh**(*x*)

返回 *x* 的反双曲正弦值。

math.**atanh**(*x*)

返回 *x* 的反双曲正切值。

math.**cosh**(*x*)

返回 *x* 的双曲余弦值。

math.**sinh**(*x*)

返回 *x* 的双曲正弦值。

math.**tanh**(*x*)

返回 *x* 的双曲正切值。

9.2.9 特殊函数

math.**erf**(*x*)

返回 *x* 处的 误差函数 。

可以使用*erf()* 函数来计算传统的统计函数如 累积标准正态分布:

```
def phi(x):
    '标准正态分布的累积分布函数'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Added in version 3.2.

math.**erfc**(*x*)

返回 *x* 处的互补误差函数。互补误差函数 定义为 `1.0 - erf(x)`。它用于大的 *x* 取值，以避免直接用 1 减其误差函数值导致的 有效位数损失。

Added in version 3.2.

math.**gamma**(*x*)

返回 *x* 处的 伽马函数 值。

Added in version 3.2.

math.**lgamma**(*x*)

返回 Gamma 函数在 *x* 绝对值的自然对数。

Added in version 3.2.

9.2.10 常量

math.**pi**

数学常数 $\pi = 3.141592...$，精确到可用精度。

math.**e**

数学常数 $e = 2.718281...$，精确到可用精度。

math.**tau**

数学常数 $\tau = 6.283185...$，精确到可用精度。Tau 是一个圆周常数，等于 2π，圆的周长与半径之比。更多关于 Tau 的信息可参考 Vi Hart 的视频 Pi is (still) Wrong。吃两倍多的派来庆祝 Tau 日 吧！

Added in version 3.6.

math.**inf**

浮点正无穷大。（对于负无穷大，使用 `-math.inf` 。）相当于 `float('inf')` 的输出。

Added in version 3.5.

math.**nan**

一个浮点数值"Not a Number" (NaN)。相当于 `float('nan')` 的输出。根据 IEEE-754 标准 要求，`math.nan` 和 `float('nan')` 不会被视为等于任何其他数值，包括其本身。要检查一个数字是否为 NaN，请使用*isnan()* 函数来测试 NaN 而不能使用 `is` 或 `==`。例如:

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

Added in version 3.5.

在 3.11 版本发生变更: 该常量现在总是可用。

math 模块主要包含围绕平台 C 数学库函数的简单包装器。特殊情况下的行为在适当情况下遵循 C99 标准的附录 F。当前的实现将引发*ValueError* 用于无效操作，如 `sqrt(-1.0)` 或 `log(0.0)` （其中 C99 附件 F 建议发出无效操作信号或被零除），和*OverflowError* 用于溢出的结果（例如，`exp(1000.0)` ）。除非一个或多个输入参数是 NaN，否则不会从上述任何函数返回 NaN；在这种情况下，大多数函数将返回一个 NaN，但是（再次遵循 C99 附件 F）这个规则有一些例外，例如 `pow(float('nan'), 0.0)` 或 `hypot(float('nan'), float('inf'))` 。

请注意，Python 不会将显式 NaN 与静默 NaN 区分开来，并且显式 NaN 的行为仍未明确。典型的行为是将所有 NaN 视为静默的。

参见

cmath 模块
这里很多函数的复数版本。

9.3 cmath --- 针对复数的数学函数

本模块提供了一些适用于复数的数学函数。本模块中的函数接受整数、浮点数或复数作为参数。它们也接受任意具有 `__complex__()` 或 `__float__()` 方法的 Python 对象：这些方法分别用于将对象转换为复数或浮点数，然后再将函数应用于转换后的结果。

备注

对于涉及分支切割的函数，我们会有确定如何在切割本身上定义这些函数的问题。根据 Kahan 的论文"Branch cuts for complex elementary functions"，以及 C99 的附录 G 和之后的 C 标准，我们使用零符号来区别分支切割的一侧和另一侧：对于沿实轴（一部分）的分支切割我们要看虚部的符号，而对于沿虚轴的分支切割我们则要看实部的符号。

例如，*cmath.sqrt()* 函数有一个沿着负实轴的分支切割。参数 `complex(-2.0, -0.0)` 会被当作位于切支切割的 下方来处理，因而将给出一个负虚轴上的结果。

```
>>> cmath.sqrt(complex(-2.0, -0.0))
-1.4142135623730951j
```

但是参数 `complex(-2.0, 0.0)` 则会被当作是位于支割线的上方来处理:

```
>>> cmath.sqrt(complex(-2.0, 0.0))
1.4142135623730951j
```

针对极坐标的转换	
phase(z)	返回 *z* 的相位
polar(z)	返回 *z* 的极坐标表示形式
rect(r, phi)	返回复数 *z* 的极坐标值 *r* 和 *phi*
幂函数与对数函数	
exp(z)	返回 *e* 的 *z* 次幂
log(z[, base])	返回 *z* 的指定底数 *base* (默认为 *e*) 的对数
log10(z)	返回 *z* 的以 10 为底的对数
sqrt(z)	返回 *z* 的平方根
三角函数	
acos(z)	返回 *z* 的反余弦
asin(z)	返回 *z* 的反正弦
atan(z)	返回 *z* 的反正切
cos(z)	返回 *z* 的余弦
sin(z)	返回 *z* 的正弦
tan(z)	返回 *z* 的正切
双曲函数	
acosh(z)	返回 *z* 的反双曲余弦
asinh(z)	返回 *z* 的反双曲正弦
atanh(z)	返回 *z* 反双曲正切
cosh(z)	返回 *z* 的双曲余弦
sinh(z)	返回 *z* 的双曲正弦
tanh(z)	返回 *z* 的双曲正切
分类函数	

续下页

表 2 – 接上页

isfinite(z)	检测是否 *z* 的所有部分均为有限值
isinf(z)	检测是否 *z* 的每个部分均为无穷大
isnan(z)	检测是否 *z* 的每个部分均为 NaN
*isclose(a, b, *, rel_tol, abs_tol)*	检查 *a* 和 *b* 的值是否彼此接近
常量	
pi	$\pi = 3.141592...$
e	$e = 2.718281...$
tau	$\tau = 2\pi = 6.283185...$
inf	正无穷
infj	纯虚部无穷
nan	” 非数字” (NaN)
nanj	纯实部 NaN

9.3.1 到极坐标和从极坐标的转换

Python 复数 z 是使用 直角或 笛卡尔坐标在内部存储的。这完全取决于其 实部 `z.real` 及其 虚部 `z.imag` 的值。

极坐标提供了另一种复数的表示方法。在极坐标中，一个复数 *z* 由模量 *r* 和相位角 *phi* 来定义。模量 *r* 是从 *z* 到坐标原点的距离，而相位角 *phi* 是以弧度为单位的，逆时针的，从正 X 轴到连接原点和 *z* 的线段间夹角的角度。

下面的函数可用于原生直角坐标与极坐标的相互转换。

`cmath.`**`phase`**`(z)`

将 *z* 的相位 (或称 *z* 的 参数) 作为一个浮点数返回。`phase(z)` 等价于 `math.atan2(z.imag, z.real)`。结果将位于 $[-\pi, \pi]$ 范围内，且此操作的支割线将位于负实轴上。结果的正负号将与 `z.imag` 的正负号相同，即使 `z.imag` 值为零:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

备注

一个复数 *z* 的模数（绝对值）可以使用内置的*abs()* 函数来计算。没有用于此操作的单独的*cmath* 模块函数。

`cmath.`**`polar`**`(z)`

返回在极坐标中 *z* 的表示形式。返回一个数值对 `(r, phi)` 其中 *r* 是 *z* 的模数而 *phi* 是 *z* 的相位。`polar(z)` 等价于 `(abs(z), phase(z))`。

`cmath.`**`rect`**`(r, phi)`

将复数 *z* 返回为极坐标 *r* 和 *phi* 形式。等价于 `complex(r * math.cos(phi), r * math.sin(phi))`。

9.3.2 幂函数与对数函数

`cmath.`**`exp`**`(z)`

返回 *e* 的 *z* 次方，其中 *e* 是自然对数的底数。

`cmath.`**`log`**`(z[, base])`

返回 *z* 的以给定的 *base* 为底的对数。如果没有指定 *base*，则返回 *z* 的自然对数。存在一条支割线，即沿着负实轴从 0 到 -∞。

cmath.**log10**(*z*)

返回 *z* 的以 10 为底的对数。它具有与*log()* 相同的支割线。

cmath.**sqrt**(*z*)

返回 *z* 的平方根。它具有与*log()* 相同的支割线。

9.3.3 三角函数

cmath.**acos**(*z*)

返回 *z* 的反余弦。存在两条支割线：一条沿着实轴从 1 到 ∞。另一条沿着实轴从 -1 向左延伸到 -∞。

cmath.**asin**(*z*)

返回 *z* 的反正弦。它具有与*acos()* 相同的支割线。

cmath.**atan**(*z*)

返回 *z* 的反正切。存在两条支割线：一条沿着虚轴从 `1j` 到 `∞j`。另一条沿着虚轴从 `-1j` 延伸到 `-∞j`。

cmath.**cos**(*z*)

返回 *z* 的余弦。

cmath.**sin**(*z*)

返回 *z* 的正弦。

cmath.**tan**(*z*)

返回 *z* 的正切。

9.3.4 双曲函数

cmath.**acosh**(*z*)

返回 *z* 的反双曲余弦。存在一条支割线，沿着实轴从 1 向左延伸到 -∞。

cmath.**asinh**(*z*)

返回 *z* 的反双曲正弦。存在两条支割线：一条沿着虚轴从 `1j` 注册会计师到 `∞j`。另一条沿着虚轴从 `-1j` 延伸到 `-∞j`。

cmath.**atanh**(*z*)

返回 *z* 的反双曲正切。存在两条支割线：一条沿着实轴从 `1` 延伸到 ∞。另一条沿着实轴从 `-1` 延伸到 `-∞`。

cmath.**cosh**(*z*)

返回 *z* 的双曲余弦。

cmath.**sinh**(*z*)

返回 *z* 的双曲正弦。

cmath.**tanh**(*z*)

返回 *z* 的双曲正切。

9.3.5 分类函数

cmath.**isfinite**(*z*)

如果 *z* 的实部和虚部均为有限值则返回 `True`，否则返回 `False`。

Added in version 3.2.

cmath.**isinf**(*z*)

如果 *z* 的实部或虚部为无穷大则返回 `True`，否则返回 `False`。

cmath.**isnan**(*z*)

如果 *z* 的实部或虚部为 NaN 则返回 `True`，否则返回 `False`。

cmath.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)

若 *a* 和 *b* 的值比较接近则返回 `True`，否则返回 `False`。

两个值是否会被视为相近是根据给定的绝对和相对容差来确定的。如果未发生错误，结果将为: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`。

rel_tol 是相对容差 -- 它是 *a* 和 *b* 之间的最大允许差值，相对于 *a* 或 *b* 中绝对值较大的一个而言。例如，要设置 5% 的容差，则传入 `rel_tol=0.05`。默认的容差为 `1e-09`，这将确保两个值在大约 9 个十进制数位内是相同的。*rel_tol* 必须为非负值并且小于 `1.0`。

abs_tol 是绝对容差；其默认值为 `0.0` 并且必须为非负值。当将 `x` 与 `0.0` 比较时，`isclose(x, 0)` 将按 `abs(x) <= rel_tol * abs(x)` 来计算，对于 `x` 和小于 `1.0` 的 rel_tol 来说均为 `False`。因此请为该调用添一个为适当正值的 abs_tol。

IEEE 754 特殊值 `NaN`，`inf` 和 `-inf` 将根据 IEEE 规则处理。具体来说，`NaN` 不被认为接近任何其他值，包括 `NaN`。`inf` 和 `-inf` 只被认为接近自己。

Added in version 3.5.

> **参见**
>
> **PEP 485** ——用于测试近似相等的函数

9.3.6 常量

cmath.**pi**

数学常数 π，作为一个浮点数。

cmath.**e**

数学常数 e，作为一个浮点数。

cmath.**tau**

数学常数 τ，作为一个浮点数。

Added in version 3.6.

cmath.**inf**

浮点正无穷大。相当于 `float('inf')`。

Added in version 3.6.

cmath.**infj**

具有零实部和正无穷虚部的复数。相当于 `complex(0.0, float('inf'))`。

Added in version 3.6.

cmath.**nan**

浮点“非数字”（NaN）值。相当于 `float('nan')`。

Added in version 3.6.

cmath.**nanj**

具有零实部和 NaN 虚部的复数。相当于 `complex(0.0, float('nan'))`。

Added in version 3.6.

请注意，函数的选择与模块*math* 中的函数选择相似，但不完全相同。拥有两个模块的原因是因为有些用户对复数不感兴趣，甚至根本不知道它们是什么。它们宁愿 `math.sqrt(-1)` 引发异常，也不想返回一个复数。另请注意，被*cmath* 定义的函数始终会返回一个复数，尽管答案可以表示为一个实数（在这种情况下，复数的虚数部分为零）。

关于支割线的注释：它们是沿着给定函数无法连续的曲线。它们是许多复变函数的必要特征。假设您需要使用复变函数进行计算，您将会了解支割线的概念。请参阅几乎所有关于复变函数的（不太基本）的书来获得启发。对于如何正确地基于数值目的来选择支割线的相关信息，一个良好的参考如下：

参见

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165--211.

9.4 decimal --- 十进制定点和浮点算术

源码： Lib/decimal.py

decimal 模块提供了对快速且正确舍入的十进制浮点运算的支持。与*float* 数据类型相比它具有以下优势:

- Decimal 类型的“设计是基于考虑人类习惯的浮点数模型，并且因此具有以下最高指导原则——计算机必须提供与人们在学校所学习的算术相一致的算术。”——摘自 decimal 算术规范描述。
- Decimal 数字可以完全精确地表示。相比之下，`1.1` 和 `2.2` 这样的数字在二进制浮点形式下没有精确的表示。最终用户通常不希望 `1.1 + 2.2` 像在二进制浮点形式下那样被显示为 `3.3000000000000003`。
- 这样的精确性会延续到算术运算中。对于 decimal 浮点数，`0.1 + 0.1 + 0.1 - 0.3` 会精确地等于零。而对于二进制浮点数，结果则为 `5.5511151231257827e-017`。虽然接近于零，但其中的误差将妨碍到可靠的相等性检测并且这样的误差还会不断累积。因此，decimal 更适合具有严格相等不变性要求的会计类应用。
- decimal 模块包含有效位的概念因而使得 `1.30 + 1.20` 等于 `2.50`。末尾的零会被保留以表明有效位。这是货币相关应用的惯例表示方式。对于乘法，则按“教科书”方式来使用被乘数中的所有数位。例如，`1.3 * 1.2` 结果为 `1.56` 而 `1.30 * 1.20` 结果为 `1.5600`。
- 与基于硬件的二进制浮点不同，十进制模块具有用户可更改的精度（默认为 28 位），可以与给定问题所需的一样大:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 二进制和 decimal 浮点数都是根据已发布的标准实现的。虽然内置浮点类型只公开其功能的一小部分，但 decimal 模块公开了标准的所有必需部分。在需要时，程序员可以完全控制舍入和信号处理。这包括通过使用异常来阻止任何不精确操作来强制执行精确算术的选项。
- decimal 模块旨在支持“无偏差，精确无舍入的十进制算术（有时称为定点数算术）和有舍入的浮点数算术”。——摘自 decimal 算术规范说明。

该模块的设计以三个概念为中心：decimal 数值，算术上下文和信号。

decimal 数值属于不可变对象。它由一个符号、一个系数值及一个指数值组成。为了保留有效位，系数值不会截去末尾的零。decimal 数值还包括特殊值如 `Infinity`, `-Infinity` 和 `NaN`。该标准还会区分 `-0` 和 `+0`。

算术的上下文是指定精度、舍入规则、指数限制、指示操作结果的标志以及确定符号是否被视为异常的陷阱启用器的环境。舍入选项包括*ROUND_CEILING* 、*ROUND_DOWN* 、*ROUND_FLOOR* 、*ROUND_HALF_DOWN*, *ROUND_HALF_EVEN* 、*ROUND_HALF_UP* 、*ROUND_UP* 以及*ROUND_05UP*.

信号是在计算过程中出现的异常条件组。根据应用程序的需要，信号可能会被忽略，被视为信息，或被视为异常。十进制模块中的信号有：*Clamped* 、*InvalidOperation* 、*DivisionByZero* 、*Inexact* 、*Rounded* 、*Subnormal* 、*Overflow* 、*Underflow* 以及*FloatOperation* 。

对于每个信号，都有一个标志和一个陷阱启动器。遇到信号时，其标志设置为 1 ，然后，如果陷阱启用器设置为 1 ，则引发异常。标志是粘性的，因此用户需要在监控计算之前重置它们。

参见

- IBM 的通用十进制算术规范描述，The General Decimal Arithmetic Specification。

9.4.1 快速入门教程

通常使用 decimal 的方式是先导入该模块，通过*getcontext()* 查看当前上下文，并在必要时为精度、舍入或启用的陷阱设置新值:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # 设置新的精度
```

Decimal 实例可以基于整数、字符串、浮点数或元组来构建。基于整数或浮点数进行构建将执行该整数或浮点数值的精确转换。Decimal 数字包括特殊值如代表“非数字”的 `NaN`，正的和负的 `Infinity` 以及 `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

如果*FloatOperation* 信号被捕获，构造函数中的小数和浮点数的意外混合或排序比较会引发异常

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

Added in version 3.3.

新 Decimal 的重要性仅由输入的位数决定。上下文精度和舍入仅在算术运算期间发挥作用。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
```

(续下页)

(接上页)

```
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

如果超出了 C 版本的内部限制，则构造一个 decimal 将引发*InvalidOperation*

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

在 3.3 版本发生变更.

Decimal 能很好地与 Python 的其余部分交互。以下是一个小小的 decimal 浮点数飞行马戏团：

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Decimal 也可以使用一些数学函数：

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

quantize() 方法将舍入为固定的指数。此方法对于将结果舍入到固定位置的货币应用程序来说很有用处：

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
```

(续下页)

(接上页)

```
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

如上所示，*getcontext()* 函数访问当前上下文并允许更改设置。这种方法满足大多数应用程序的需求。

对于更高级的工作，使用 Context() 构造函数创建备用上下文可能很有用。要使用备用活动，请使用*setcontext()* 函数。

根据标准，*decimal* 模块提供了两个现成的标准上下文*BasicContext* 和*ExtendedContext* 。前者对调试特别有用，因为许多陷阱都已启用：

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

上下文还具有用于监视计算期间遇到的异常情况的信号旗标。这些旗标将保持设置直到被显式地清除，因此最好是通过使用*clear_flags()* 方法来清除每组受监控的计算之前的旗标。

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

flags 条目显示对 pi 的有理逼近被舍入（超出上下文精度的数字会被丢弃）并且结果是不精确的（某些被丢弃的数字为非零值）。

单个陷阱是使用上下文的 `traps` 属性中的字典来设置的：

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

大多数程序仅在程序开始时调整当前上下文一次。并且，在许多应用程序中，数据在循环内单个强制转换为*Decimal* 。通过创建上下文集和小数，程序的大部分操作数据与其他 Python 数字类型没有区别。

9.4.2 Decimal 对象

class decimal.**Decimal**(*value='0'*, *context=None*)

根据 *value* 构造一个新的*Decimal* 对象。

value 可以是整数，字符串，元组，*float* ，或另一个*Decimal* 对象。如果没有给出 *value*，则返回 `Decimal('0')`。如果 *value* 是一个字符串，它应该在前导和尾随空格字符以及下划线被删除之后符合十进制数字字符串语法:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits `'\uff10'` through `'\uff19'`. Case is not significant, so, for example, `inf`, `Inf`, `INFINITY`, and `iNfINity` are all acceptable spellings for positive infinity.

如果 *value* 是一个*tuple*，它应当有三个组成部分，一个符号 (`0` 表示正数 `1` 表示负数)，一个由数字组成的*tuple*，以及一个整数指数值。例如，`Decimal((0, (1, 4, 1, 4), -3))` 将返回 `Decimal('1.414')`。

如果 *value* 是*float*，则二进制浮点值将无损地转换为其精确的十进制等效值。此转换通常需要 53 位或更多位数的精度。例如，`Decimal(float('1.1'))` 转换为 `Decimal('1.1000000000000000888178419700125232338905334472656 25')`。

context 精度不会影响存储的位数。这完全由 *value* 中的位数决定。例如，`Decimal('3.00000')` 记录所有五个零，即使上下文精度只有三。

context 参数的目的是确定当 *value* 为错误格式的字符串时要怎么做。如果上下文捕获了*InvalidOperation*，将会引发异常；在其他情况下，构造器将返回一个值为 `NaN` 的新 Decimal。

构造完成后，*Decimal* 对象是不可变的。

在 3.2 版本发生变更: 现在允许构造函数的参数为*float* 实例。

在 3.3 版本发生变更: *float* 参数在设置*FloatOperation* 陷阱时引发异常。默认情况下，陷阱已关闭。

在 3.6 版本发生变更: 允许下划线进行分组，就像代码中的整数和浮点文字一样。

十进制浮点对象与其他内置数值类型共享许多属性，例如*float* 和*int* 。所有常用的数学运算和特殊方法都适用。同样，十进制对象可以复制、pickle、打印、用作字典键、用作集合元素、比较、排序和强制转换为另一种类型（例如*float* 或*int* 等）。

算术对十进制对象和算术对整数和浮点数有一些小的差别。当余数运算符 `%` 应用于 Decimal 对象时，结果的符号是 *被除数* 的符号，而不是除数的符号:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

整数除法运算符 `//` 的行为类似，返回真商的整数部分（截断为零）而不是它的向下取整，以便保留通常的标识 `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

`%` 和 `//` 运算符实现了 `remainder` 和 `divide-integer` 操作（分别），如规范中所述。

十进制对象通常不能与浮点数或*fractions.Fraction* 实例在算术运算中结合使用：例如，尝试将*Decimal* 加到*float*，将引发*TypeError*。但是，可以使用 Python 的比较运算符来比较*Decimal* 实例 `x` 和另一个数字 `y`。这样可以避免在对不同类型的数字进行相等比较时混淆结果。

在 3.2 版本发生变更: 现在完全支持*Decimal* 实例和其他数字类型之间的混合类型比较。

除了标准的数字属性，十进制浮点对象还有许多专门的方法：

adjusted()

在移出系数最右边的数字之后返回调整后的指数，直到只剩下前导数字：`Decimal('321e+5').adjusted()` 返回 7。用于确定最高有效位相对于小数点的位置。

as_integer_ratio()

返回一对 `(n, d)` 整数，表示给定的*Decimal* 实例作为分数、最简形式项并带有正分母：

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

转换是精确的。在 Infinity 上引发 OverflowError，在 NaN 上引起 ValueError。

Added in version 3.6.

as_tuple()

返回一个*named tuple* 表示的数字：`DecimalTuple(sign, digits, exponent)`。

canonical()

返回参数的规范编码。目前，一个*Decimal* 实例的编码始终是规范的，因此该操作返回其参数不变。

compare(*other*, *context=None*)

比较两个 Decimal 实例的值。*compare()* 返回一个 Decimal 实例，如果任一操作数是 NaN，那么结果是 NaN

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)

除了所有 NaN 信号之外，此操作与*compare()* 方法相同。也就是说，如果两个操作数都不是信令 NaN，那么任何静默的 NaN 操作数都被视为信令 NaN。

compare_total(*other*, *context=None*)

使用它们的抽象表示而不是它们的数值来比较两个操作数。类似于*compare()* 方法，但结果给出了一个总排序*Decimal* 实例。两个*Decimal* 实例具有相同的数值但不同的表示形式在此排序中比较不相等：

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

静默和发出信号的 NaN 也包括在总排序中。这个函数的结果是 `Decimal('0')` 如果两个操作数具有相同的表示，或是 `Decimal('-1')` 如果第一个操作数的总顺序低于第二个操作数，或是 `Decimal('1')` 如果第一个操作数在总顺序中高于第二个操作数。有关总排序的详细信息，请参阅规范。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

compare_total_mag(*other*, *context=None*)

比较两个操作数使用它们的抽象表示而不是它们的值，如*compare_total()*，但忽略每个操作数的符号。`x.compare_total_mag(y)` 相当于 `x.copy_abs().compare_total(y.copy_abs())`。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

conjugate()

只返回 self，这种方法只符合 Decimal 规范。

copy_abs()

返回参数的绝对值。此操作不受上下文影响并且是静默的：没有更改标志且不执行舍入。

copy_negate()

回到参数的否定。此操作不受上下文影响并且是静默的：没有标志更改且不执行舍入。

copy_sign(*other*, *context=None*)

返回第一个操作数的副本，其符号设置为与第二个操作数的符号相同。例如：

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

exp(*context=None*)

返回给定数字的（自然）指数函数 `e**x` 的值。结果使用*ROUND_HALF_EVEN* 舍入模式正确舍入。

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

classmethod from_float(*f*)

另一个构造函数，只接受*float* 或*int* 的实例。

请注意 `Decimal.from_float(0.1)` 与 `Decimal('0.1')` 是不同的。由于 0.1 不能以二进制浮点数精确表示，该值将被存储为最接受的可表示值 `0x1.999999999999ap-4`。与其等价的十进制值为 `0.1000000000000000055511151231257827021181583404541015625`。

> **备注**
>
> 从 Python 3.2 开始，*Decimal* 实例也可以直接从*float* 构造。

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

Added in version 3.1.

fma(*other*, *third*, *context=None*)

混合乘法加法。返回 self*other+third ， 中间乘积 self*other 没有舍入。

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()

如果参数是规范的，则为返回*True*，否则为*False*。目前，*Decimal* 实例总是规范的，所以这个操作总是返回*True*。

is_finite()

如果参数是一个有限的数，则返回为*True*；如果参数为无穷大或 NaN，则返回为*False*。

is_infinite()

如果参数为正负无穷大，则返回为*True*，否则为*False*。

is_nan()

如果参数为 NaN（无论是否静默），则返回为*True*，否则为*False*。

is_normal(*context=None*)

如果参数是一个 标准的有限数则返回*True*。如果参数为零、次标准数、无穷大或 NaN 则返回*False*。

is_qnan()

如果参数为静默 NaN，返回*True*，否则返回*False*。

is_signed()

如果参数带有负号，则返回为*True*，否则返回*False*。注意，0 和 NaN 都可带有符号。

is_snan()

如果参数为显式 NaN，则返回*True*，否则返回*False*。

is_subnormal(*context=None*)

如果参数为次标准数，则返回*True*，否则返回*False*。

is_zero()

如果参数是 0（正负皆可），则返回*True*，否则返回*False*。

ln(*context=None*)

返回操作数的自然对数（以 e 为底）。结果是使用*ROUND_HALF_EVEN* 舍入模式正确舍入的。

log10(*context=None*)

返回操作数的以十为底的对数。结果是使用*ROUND_HALF_EVEN* 舍入模式正确舍入的。

logb(*context=None*)

对于一个非零数，返回其运算数的调整后指数作为一个*Decimal* 实例。如果运算数为零将返回 `Decimal('-Infinity')` 并且产生 the *DivisionByZero* 标志。如果运算数是无限大则返回 `Decimal('Infinity')`。

logical_and(*other*, *context=None*)

logical_and() 是需要两个 逻辑运算数的逻辑运算（参考逻辑操作数）。按位输出两运算数的 and 运算的结果。

logical_invert(*context=None*)

logical_invert() 是一个逻辑运算。结果是操作数的按位求反。

logical_or(*other*, *context=None*)

logical_or() 是需要两个 *logical operands* 的逻辑运算（请参阅逻辑操作数）。结果是两个运算数的按位的 or 运算。

logical_xor(*other*, *context=None*)

logical_xor() 是需要两个 逻辑运算数的逻辑运算（参考逻辑操作数）。结果是按位输出的两运算数的异或运算。

max(*other*, *context=None*)

类似于 `max(self, other)` 只是上下文舍入规则是在返回之前被应用并且对于 `NaN` 值会发出信号或忽略（依赖于上下文以及它们是否要发送信号或保持静默）。

max_mag(*other*, *context=None*)

与*max()* 方法相似，但是操作数使用绝对值完成比较。

min(*other*, *context=None*)

类似于 `min(self, other)` 只是上下文舍入规则是在返回之前被应用并且对于 `NaN` 值会发出信号或忽略（依赖于上下文以及它们是发出了信号还是保持静默）。

min_mag(*other*, *context=None*)

与*min()* 方法相似，但是操作数使用绝对值完成比较。

next_minus(*context=None*)

返回小于给定操作数的上下文中可表示的最大数字（或者当前线程的上下文中的可表示的最大数字如果没有给定上下文）。

next_plus(*context=None*)

返回大于给定操作数的上下文中可表示的最小数字（或者当前线程的上下文中的可表示的最小数字如果没有给定上下文）。

next_toward(*other*, *context=None*)

如果两运算数不相等，返回在第二个操作数的方向上最接近第一个操作数的数。如果两操作数数值上相等，返回将符号设置为与第二个运算数相同的第一个运算数的拷贝。

normalize(*context=None*)

用于在当前上下文或指定上下文中产生等价的类的规范值。

该操作具有与单目取正值运算相同的语义，区别在于如果最终结果为有限值则将缩减到最简形式，即移除所有末尾的零并保留正负号。也就是说，当系数为非零值且为十的倍数时则将该系数除以十并将指数加 1。否则（当系数为零）则将指数设为 0。在任何情况下正负号都将保持不变。

例如，`Decimal('32.100')` 和 `Decimal('0.321000e+2')` 均将标准化为等价的值 `Decimal('32.1')`。

请注意舍入的应用将在缩减到最简形式之前执行。

在此规范的最新版本中，该操作也被称为 `reduce`。

number_class(*context=None*)

返回一个字符串描述运算数的 *class* 。返回值是以下十个字符串中的一个。

- `"-Infinity"`，指示运算数为负无穷大。
- `"-Normal"`，指示该运算数是负正常数字。
- `"-Subnormal"`，指示该运算数是负的次标准数。
- `"-Zero"`，指示该运算数是负零。
- `"-Zero"`，指示该运算数是正零。
- `"+Subnormal"`，指示该运算数是正的次标准数。
- `"+Normal"`，指示该运算数是正的标准数。
- `"+Infinity"`，指示该运算数是正无穷。
- `"NaN"`，指示该运算数是肃静 NaN （非数字）。
- `"sNaN"`，指示该运算数是信号 NaN 。

quantize(*exp*, *rounding=None*, *context=None*)

返回的值等于舍入后的第一个运算数并且具有第二个操作数的指数。

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

与其他运算不同，如果量化运算后的系数长度大于精度，那么会发出一个*InvalidOperation*信号。这保证了除非有一个错误情况，量化指数恒等于右手运算数的指数。

与其他运算不同，量化永不信号下溢，即使结果不正常且不精确。

如果第二个运算数的指数大于第一个运算数的指数那或许需要舍入。在这种情况下，舍入模式由给定 `rounding` 参数决定，其余的由给定 `context` 参数决定；如果参数都未给定，使用当前线程上下文的舍入模式。

每当结果的指数大于 `Emax` 或小于*Etiny()* 就将返回一个错误。

radix()

返回 `Decimal(10)`，即*Decimal* 类进行所有算术运算所用的数制（基数）。这是为保持与规范描述的兼容性而加入的。

remainder_near(*other*, *context=None*)

返回 *self* 除以 *other* 的余数。这与 `self % other` 的区别在于所选择的余数要使其绝对值最小化。更准确地说，返回值为 `self - n * other` 其中 `n` 是最接近 `self / other` 的实际值的整数，并且如果两个整数与实际值的差相等则会选择其中的偶数。

如果结果为零则其符号将为 *self* 的符号。

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

返回对第一个操作数的数码按第二个操作数所指定的数量进行轮转的结果。第二个操作数必须为 -precision 至 precision 精度范围内的整数。第二个操作数的绝对值给出要轮转的位数。如果第二个操作数为正值则向左轮转；否则向右轮转。如有必要第一个操作数的系数会在左侧填充零以达到 precision 所指定的长度。第一个操作数的符号和指数保持不变。

same_quantum(*other*, *context=None*)

检测自身与 other 是否具有相同的指数或是否均为 `NaN`。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

scaleb(*other*, *context=None*)

返回第一个操作数使用第二个操作数对指数进行调整的结果。等价于返回第一个操作数乘以 `10**other` 的结果。第二个操作数必须为整数。

shift(*other*, *context=None*)

返回第一个操作数的数码按第二个操作数所指定的数量进行移位的结果。第二个操作数必须为 -precision 至 precision 范围内的整数。第二个操作数的绝对值给出要移动的位数。如果第二个操作数为正值则向左移位；否则向右移位。移入系数的数码为零。第一个操作数的符号和指数保持不变。

sqrt(*context=None*)

返回参数的平方根精确到完整精度。

to_eng_string(*context=None*)

转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

例如，此方法会将 `Decimal('123E+1')` 转换为 `Decimal('1.23E+3')`。

to_integral(*rounding=None*, *context=None*)

与*to_integral_value()* 方法相同。保留 `to_integral` 名称是为了与旧版本兼容。

to_integral_exact(*rounding=None*, *context=None*)

舍入到最接近的整数，发出信号*Inexact* 或者如果发生舍入则相应地发出信号*Rounded*。如果给出 `rounding` 形参则由其确定舍入模式，否则由给定的 `context` 来确定。如果没有给定任何形参则会使用当前上下文的舍入模式。

to_integral_value(*rounding=None*, *context=None*)

舍入到最接近的整数而不发出*Inexact* 或*Rounded* 信号。如果给出 *rounding* 则会应用其所指定的舍入模式；否则使用所提供的 *context* 或当前上下文的舍入方法。

可以使用*round()* 函数对 Decimal 数字执行舍入：

round(number)

round(number, ndigits)

如果 *ndigits* 未给出或为 `None`，则返回最接近 *number* 的*int*，同样接近时向偶数舍入，并忽略*Decimal* 上下文的舍入模式。如果 *number* 为无穷大则引发*OverflowError* 或者如果为（静默或有信号）NaN 则引发*ValueError*。

如果 *ndigits* 是一个*int*，则将遵循上下文的舍入模式并返回代表 *number* 的舍入到最接近 `Decimal('1E-ndigits')` 的倍数的*Decimal*；在此情况下，`round(number, ndigits)` 等价于 `self.quantize(Decimal('1E-ndigits'))`。如果 *number* 是一个静默 NaN 则返回 `Decimal('NaN')`。如果 *number* 为无穷大、有信号 NaN，或者如果量化操作后的系数长度大于当前上下文的精度则会引发*InvalidOperation*。换句话说，对于非边际情况：

- 如果 *ndigits* 为正值，则返回 *number* 舍入到 *ndigits* 个十进制数位的结果；
- 如果 *ndigits* 为零，则返回 *number* 舍入到最接近整数的结果；
- 如果 *ndigits* 为负值，则返回 *number* 舍入到最接近 `10**abs(ndigits)` 的倍数的结果。

例如:

```
>>> from decimal import Decimal, getcontext, ROUND_DOWN
>>> getcontext().rounding = ROUND_DOWN
>>> round(Decimal('3.75'))      # 上下文舍入设置将被忽略
4
>>> round(Decimal('3.5'))       # 两边相等则舍入到偶数
4
>>> round(Decimal('3.75'), 0)   # 使用上下文舍入设置
Decimal('3')
>>> round(Decimal('3.75'), 1)
Decimal('3.7')
>>> round(Decimal('3.75'), -1)
Decimal('0E+1')
```

逻辑操作数

logical_and(), *logical_invert()*, *logical_or()* 和*logical_xor()* 方法均期望其参数为 *逻辑操作数*。*逻辑操作数*即指数位和符号位均为零，且其数字位均为 `0` 或 `1` 的*Decimal* 实例。

9.4.3 上下文对象

上下文是算术运算所在的环境。它们管理精度、设置舍入规则、确定将哪些信号视为异常，并限制指数的范围。

每个线程都有自己的当前上下文，可使用*getcontext()* 和*setcontext()* 函数来读取或修改：

decimal.**getcontext**()

返回活动线程的当前上下文。

decimal.**setcontext**(*c*)

将活动线程的当前上下文设为 *c*。

你也可以使用 with 语句和*localcontext()* 函数来临时改变活动上下文。

decimal.**localcontext**(*ctx=None*, ***kwargs*)

返回一个将在进入 with 语句时将活动线程的上下文设为 *ctx* 的一个副本并在退出该 with 语句时恢复之前上下文的上下文管理器。如果未指定上下文，则会使用当前上下文的一个副本。*kwargs* 参数将被用来设置新上下文的属性。

例如，以下代码会将当前 decimal 精度设为 42 位，执行一个运算，然后自动恢复之前的上下文:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # 执行高精度的运算
    s = calculate_something()
s = +s  # 将最终结果舍入到默认精度
```

使用关键字参数，代码将如下所示:

```
from decimal import localcontext

with localcontext(prec=42) as ctx:
    s = calculate_something()
s = +s
```

如果 *kwargs* 提供了*Context* 所不支持的属性则会引发*TypeError*。如果 *kwargs* 提供了无效的属性值则会引发*TypeError* 或*ValueError*。

在 3.11 版本发生变更: *localcontext()* 现在支持通过使用关键字参数来设置上下文属性。

新的上下文也可使用下述的*Context* 构造器来创建。此外，模块还提供了三种预设的上下文:

decimal.**BasicContext**

这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为*ROUND_HALF_UP*。清除所有旗标。启用所有陷阱（视为异常），但*Inexact*, *Rounded* 和*Subnormal* 除外。

由于启用了许多陷阱，此上下文适用于进行调试。

decimal.**ExtendedContext**

这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为*ROUND_HALF_EVEN*。清除所有旗标。不启用任何陷阱（因此在计算期间不会引发异常）。

由于禁用了陷阱，此上下文适用于希望结果值为 NaN 或 Infinity 而不是引发异常的应用程序。这允许应用程序在出现当其他情况下会中止程序的条件时仍能完成运行。

decimal.**DefaultContext**

此上下文被*Context* 构造器用作新上下文的原型。改变一个字段（例如精度）的效果将是改变*Context* 构造器所创建的新上下文的默认值。

此上下文最适用于多线程环境。在线程开始前改变一个字段具有设置全系统默认值的效果。不推荐在线程开始后改变字段，因为这会要求线程同步避免竞争条件。

在单线程环境中，最好完全不使用此上下文。而是简单地电显式创建上下文，具体如下所述。

默认值为 Context.prec=28, Context.rounding=*ROUND_HALF_EVEN*, 并为*Overflow*, *InvalidOperation* 和*DivisionByZero* 启用陷阱。

在已提供的三种上下文之外，还可以使用*Context* 构造器创建新的上下文。

class decimal.**Context**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

创建一个新上下文。如果某个字段未指定或为*None*，则从*DefaultContext* 拷贝默认值。如果 *flags* 字段未指定或为*None*，则清空所有旗标。

prec 是一个用于设置该上下文中算术运算的精度的 [1, *MAX_PREC*] 范围内的整数。

rounding 选项应为*Rounding Modes* 小节中列出的常量之一。

traps 和 *flags* 字段列出要设置的任何信号。通常，新上下文应当只设置 traps 而让 flags 为空。

Emin 和 *Emax* 字段是指定指数所允许的外部上限的整数。*Emin* 必须在 [*MIN_EMIN*, 0] 范围内，*Emax* 必须在 [0, *MAX_EMAX*] 范围内。

capitals 字段为 0 或 1 (默认值)。如果设为 1，指数将附带大写的 E 打印出来；在其他情况下将使用小写的 e: `Decimal('6.02e+23')`。

clamp 字段为 0 (默认值) 或 1。如果设为 1，则*Decimal* 实例的指数 e 的表示范围在此上下文中将严格限制在 `Emin - prec + 1 <= e <= Emax - prec + 1` 范围内。如果 *clamp* 为 0 则将适用较弱的条件：调整后的*Decimal* 实例指数最大值为 Emax。当 *clamp* 为 1 时，一个较大的普通数值将在可能的情况下减小其指数并为其系数添加相应数量的零，以便符合指数值限制；这可以保留数字值但会丢失有效末尾零的信息。例如:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

将 *clamp* 值设为 1 即允许与 IEEE 754 所描述的固定宽度十进制交换格式保持兼容性。

Context 类定义了几种通用方法以及大量直接在给定上下文中进行算术运算的方法。此外，对于上述的每种*Decimal* 方法（除了*adjusted()* 和*as_tuple()* 方法）都有一个对应的*Context* 方法。例如，对于一个*Context* 的实例 C 和*Decimal* 的实例 x，`C.exp(x)` 就等价于 `x.exp(context=C)`。每个*Context* 方法都接受一个 Python 整数（即*int* 的实例）在任何接受 Decimal 实例的地方使用。

clear_flags()

将所有旗标重置为 0。

clear_traps()

将所有陷阱重置为 0。

Added in version 3.3.

copy()

返回上下文的一个副本。

copy_decimal(*num*)

返回 Decimal 实例 num 的一个副本。

create_decimal(*num*)

基于 *num* 创建一个新 Decimal 实例但使用 *self* 作为上下文。与*Decimal* 构造器不同，该上下文的精度、舍入方法、旗标和陷阱会被应用于转换过程。

此方法很有用处，因为常量往往被给予高于应用所需的精度。另一个好处在于立即执行舍入可以消除超出当前精度的数位所导致的意外效果。在下面的示例中，使用未舍入的输入意味着在总和中添加零会改变结果:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

此方法实现了 IBM 规格描述中的转换为数字操作。如果参数为字符串，则不允许有开头或末尾的空格或下划线。

create_decimal_from_float(*f*)

基于浮点数 *f* 创建一个新的 Decimal 实例，但会使用 *self* 作为上下文来执行舍入。与*Decimal.from_float()* 类方法不同，上下文的精度、舍入方法、旗标和陷阱会应用到转换中。

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

Added in version 3.1.

Etiny()

返回一个等于 `Emin - prec + 1` 的值即次标准化结果中的最小指数值。当发生向下溢出时，指数会设为*Etiny*。

Etop()

返回一个等于 `Emax - prec + 1` 的值。

使用 decimal 的通常方式是创建*Decimal* 实例然后对其应用算术运算，这些运算发生在活动线程的当前上下文中。一种替代方式则是使用上下文的方法在特定上下文中进行计算。这些方法类似于*Decimal* 类的方法，在此仅简单地重新列出。

abs(*x*)

返回 *x* 的绝对值。

add(*x*, *y*)

返回 *x* 与 *y* 的和。

canonical(*x*)

返回相同的 Decimal 对象 *x*。

compare(*x*, *y*)

对 *x* 与 *y* 进行数值比较。

compare_signal(*x*, *y*)

对两个操作数进行数值比较。

compare_total(*x*, *y*)

对两个操作数使用其抽象表示进行比较。

compare_total_mag(*x*, *y*)

对两个操作数使用其抽象表示进行比较，忽略符号。

copy_abs(*x*)

返回 *x* 的副本，符号设为 0。

copy_negate(*x*)

返回 *x* 的副本，符号取反。

copy_sign(*x*, *y*)

从 *y* 拷贝符号至 *x*。

divide(*x*, *y*)

返回 *x* 除以 *y* 的结果。

divide_int(*x*, *y*)

返回 *x* 除以 *y* 的结果，截短为整数。

divmod(*x*, *y*)
两个数字相除并返回结果的整数部分。

exp(*x*)
返回 `e ** x`。

fma(*x*, *y*, *z*)
返回 *x* 乘以 *y* 再加 *z* 的结果。

is_canonical(*x*)
如果 *x* 是规范的则返回 `True`；否则返回 `False`。

is_finite(*x*)
如果 *x* 为有限的则返回 `True`；否则返回 `False`。

is_infinite(*x*)
如果 *x* 是无限的则返回 `True`；否则返回 `False`。

is_nan(*x*)
如果 *x* 是 qNaN 或 sNaN 则返回 `True`；否则返回 `False`。

is_normal(*x*)
如果 *x* 是标准数则返回 `True`；否则返回 `False`。

is_qnan(*x*)
如果 *x* 是静默 NaN 则返回 `True`；否则返回 `False`。

is_signed(*x*)
x 是负数则返回 `True`；否则返回 `False`。

is_snan(*x*)
如果 *x* 是显式 NaN 则返回 `True`；否则返回 `False`。

is_subnormal(*x*)
如果 *x* 是次标准数则返回 `True`；否则返回 `False`。

is_zero(*x*)
如果 *x* 为零则返回 `True`；否则返回 `False`。

ln(*x*)
返回 *x* 的自然对数（以 e 为底）。

log10(*x*)
返回 *x* 的以 10 为底的对数。

logb(*x*)
返回操作数的 MSD 等级的指数。

logical_and(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *and*。

logical_invert(*x*)
反转 *x* 中的所有数位。

logical_or(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *or*。

logical_xor(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *xor*。

max(*x*, *y*)
对两个值执行数字比较并返回其中的最大值。

max_mag(*x*, *y*)

对两个值执行忽略正负号的数字比较。

min(*x*, *y*)

对两个值执行数字比较并返回其中的最小值。

min_mag(*x*, *y*)

对两个值执行忽略正负号的数字比较。

minus(*x*)

对应于 Python 中的单目前缀取负运算符执行取负操作。

multiply(*x*, *y*)

返回 *x* 和 *y* 的积。

next_minus(*x*)

返回小于 *x* 的最大数字表示形式。

next_plus(*x*)

返回大于 *x* 的最小数字表示形式。

next_toward(*x*, *y*)

返回 *x* 趋向于 *y* 的最接近的数字。

normalize(*x*)

将 *x* 改写为最简形式。

number_class(*x*)

返回 *x* 的类的表示。

plus(*x*)

对应于 Python 中的单目前缀取正运算符执行取正操作。此操作将应用上下文精度和舍入，因此它 不是标识运算。

power(*x*, *y*, *modulo=None*)

返回 `x` 的 `y` 次方，如果给出了模数 `modulo` 则取其余数。

传入两个参数时，计算 `x**y`。如果 `x` 为负值则 `y` 必须为整数。除非 `y` 为整数且结果为有限值并可在'precision' 位内精确表示否则结果将是不精确的。所在上下文的舍入模式将被使用。结果在 Python 版中总是会被正确地舍入。

`Decimal(0) ** Decimal(0)` 结果为 `InvalidOperation`，而如果 `InvalidOperation` 未被捕获，则结果为 `Decimal('NaN')`。

在 3.3 版本发生变更: C 模块计算*power()* 时会使用已正确舍入的*exp()* 和*ln()* 函数。结果是有良好定义的但仅限于“几乎总是正确舍入”。

带有三个参数时，计算 `(x**y) % modulo`。对于三个参数的形式，参数将会应用以下限制：

- 三个参数必须都是整数
- `y` 必须是非负数
- `x` 或 `y` 至少有一个不为零
- `modulo` 必须不为零且至多有'precision' 位

来自 `Context.power(x, y, modulo)` 的结果值等于使用无限精度计算 `(x**y) % modulo` 所得到的值，但其计算过程更高效。结果的指数为零，无论 `x`, `y` 和 `modulo` 的指数是多少。结果值总是完全精确的。

quantize(*x*, *y*)

返回的值等于 *x*（舍入后），并且指数为 *y*。

radix()

恰好返回 10，因为这是 Decimal 对象:)

remainder(*x*, *y*)

返回整除所得到的余数。

结果的符号，如果不为零，则与原始除数的符号相同。

remainder_near(*x*, *y*)

返回 `x - y * n`，其中 *n* 为最接近 `x / y` 实际值的整数（如结果为 0 则其符号将与 *x* 的符号相同）。

rotate(*x*, *y*)

返回 *x* 翻转 *y* 次的副本。

same_quantum(*x*, *y*)

如果两个操作数具有相同的指数则返回 `True`。

scaleb(*x*, *y*)

返回第一个操作数添加第二个值的指数后的结果。

shift(*x*, *y*)

返回 *x* 变换 *y* 次的副本。

sqrt(*x*)

非负数基于上下文精度的平方根。

subtract(*x*, *y*)

返回 *x* 和 *y* 的差。

to_eng_string(*x*)

转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

to_integral_exact(*x*)

舍入到一个整数。

to_sci_string(*x*)

使用科学计数法将一个数字转换为字符串。

9.4.4 常量

本节中的常量仅与 C 模块相关。它们也被包含在纯 Python 版本以保持兼容性。

	32 位	64 位
decimal.**MAX_PREC**	425000000	999999999999999999
decimal.**MAX_EMAX**	425000000	999999999999999999
decimal.**MIN_EMIN**	-425000000	-999999999999999999
decimal.**MIN_ETINY**	-849999999	-1999999999999999997

decimal.**HAVE_THREADS**

该值为 `True`。已弃用，因为 Python 现在总是启用线程。

自 3.9 版本弃用.

decimal.**HAVE_CONTEXTVAR**

默认值为 `True`。如果 Python 编译版本 使用了 `--without-decimal-contextvar` 选项来配置，则 C 版本会使用线程局部而非协程局部上下文并且该值为 `False`。这在某些嵌套上下文场景中将会稍快一些。

Added in version 3.8.3.

9.4.5 舍入模式

decimal.**ROUND_CEILING**

舍入方向为 `Infinity`。

decimal.**ROUND_DOWN**

舍入方向为零。

decimal.**ROUND_FLOOR**

舍入方向为 `-Infinity`。

decimal.**ROUND_HALF_DOWN**

舍入到最接近的数，同样接近则舍入方向为零。

decimal.**ROUND_HALF_EVEN**

舍入到最接近的数，同样接近则舍入到最接近的偶数。

decimal.**ROUND_HALF_UP**

舍入到最接近的数，同样接近则舍入到零的反方向。

decimal.**ROUND_UP**

舍入到零的反方向。

decimal.**ROUND_05UP**

如果最后一位朝零的方向舍入后为 0 或 5 则舍入到零的反方向；否则舍入方向为零。

9.4.6 信号

信号代表在计算期间引发的条件。每个信号对应于一个上下文旗标和一个上下文陷阱启用器。

上下文旗标将在遇到特定条件时被设定。在完成计算之后，将为了获得信息而检测旗标（例如确定计算是否精确）。在检测旗标后，请确保在开始下一次计算之前清除所有旗标。

如果为信号设定了上下文的陷阱启用器，则条件会导致特定的 Python 异常被引发。举例来说，如果设定了*DivisionByZero* 陷阱，则当遇到此条件时就将引发*DivisionByZero* 异常。

class decimal.**Clamped**

修改一个指数以符合表示限制。

通常，限位将在一个指数值超出上下文的 `Emin` 和 `Emax` 限制时发生。在可能的情况下，会通过向系数添加零来将指数缩减至符合限制。

class decimal.**DecimalException**

其他信号的基类，并且也是*ArithmeticError* 的一个子类。

class decimal.**DivisionByZero**

非无限数被零除的信号。

可在除法、取余除法或对一个数执行负数次幂运算时发生。如果此信号未被陷阱捕获，则返回 `Infinity` 或 `-Infinity` 并由对计算的输入来确定正负符号。

class decimal.**Inexact**

表明发生了舍入且结果是不精确的。

有非零数位在舍入期间被丢弃的信号。舍入结果将被返回。此信号旗标或陷阱被用于检测结果不精确的情况。

class decimal.**InvalidOperation**

执行了一个无效的操作。

表明请求了一个无意义的运算。如果未被捕获，则返回 NaN。可能的原因包括:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**

数值的溢出。

表明在发生舍入之后指数值大于 Context.Emax。如果未被捕获，则结果将取决于舍入模式，或是向下舍入为最大的可表示有限数值，或是向上舍入为 Infinity。无论是哪种情况，都将发出*Inexact* 和*Rounded* 信号。

class decimal.**Rounded**

发生了舍入，但或许并没有信息丢失。

一旦舍入操作丢弃了数位就会发出此信号；即使被丢弃的数位是零（如将 5.00 舍入到 5.0 的情况）。如果未被捕获，则不加修改地返回结果。此信号用于检测有效位数的丢弃。

class decimal.**Subnormal**

在舍入之前指数值低于 Emin。

当操作结果是次标准数（即指数过小）时就会发出此信号。如果未被陷阱捕获，则不经修改过返回结果。

class decimal.**Underflow**

数字向下溢出导致结果舍入到零。

当一个次标准数结果通过舍入转为零时就会发出此信号。同时还将引发*Inexact* 和*Subnormal* 信号。

class decimal.**FloatOperation**

为 float 和 Decimal 的混合启用更严格的语义。

如果信号未被捕获（默认），则在*Decimal* 构造器、*create_decimal()* 和所有比较运算中允许 float 和 Decimal 的混合。转换和比较都是完全精确的。发生的任何混合运算都将通过在上下文旗标中设置*FloatOperation* 来静默地记录。通过*from_float()* 或*create_decimal_from_float()* 进行显式转换则不会设置旗标。

在其他情况下（即信号被捕获），则只静默执行相等性比较和显式转换。所有其他混合运算都将引发*FloatOperation*。

以下表格总结了信号的层级结构:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
```

(续下页)

(接上页)

```
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 浮点数说明

通过提升精度来解决舍入错误

使用 decimal 浮点数可以消除十进制表示错误（即能够精确地表示 `0.1` 这样的数）；然而，某些运算在非零数位超出了给定的精度时仍然可能导至舍入错误。

舍入错误的影响可能因接近相互抵销的加减运算被放大从而导致丢失有效位。Knuth 提供了两个指导性示例，其中出现了精度不足的浮点算术舍入，导致加法的交换律和分配律被打破：

```
# 来自 Seminumerical Algorithms, Section 4.2.2 的示例。
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

decimal 模块则可以通过充分地扩展精度来避免有效位的丢失：

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

特殊的值

decimal 模块的数字系统提供了一些特殊的值包括 `NaN`, `sNaN`, `-Infinity`, `Infinity`, 和两种零值，`+0` 和 `-0`。

无穷大可以使用 `Decimal('Infinity')` 来构建。它们也可以在不捕获*DivisionByZero* 信号捕获时通过除以零来产生。类似地，当不捕获*Overflow* 信号时，也可以通过舍入到超出最大可表示数字限制的方式产生无穷大的结果。

无穷大是有符号的（仿射）并可用于算术运算，它们会被当作极其巨大的不确定数字来处理。例如，无穷大加一个常量结果也将为无穷大。

某些运算没有确定的结果并将返回 `NaN`，或者如果捕获了*InvalidOperation* 信号，则会引发一个异常。例如，`0/0` 将返回 `NaN` 表示”not a number”。这样的 `NaN` 将静默产生，并且一旦产生就将在参与其他运算

时始终得到 NaN 的结果。这种行为对于偶尔缺少输入的各类计算都很有用处 --- 它允许在将特定结果标记为无效的同时让计算继续进行。

一种变体形式是 sNaN，它在每次运算后会发出信号而不是保持静默。当对于无效结果需要中断计算进行特别处理时这是一个很有用的返回值。

Python 中比较运算符的行为在涉及 NaN 时可能会令人有点惊讶。相等性检测在操作数中有静默型或信号型 NaN 时总是会返回*False*(即使是执行 Decimal('NaN')==Decimal('NaN'))，而不等性检测总是会返回*True*。当尝试使用 <, <=, > 或 >= 运算符中的任何一个来比较两个 Decimal 值时，如果运算数中有 NaN 则将引发*InvalidOperation* 信号，如果此信号未被捕获则将返回*False*。请注意通用十进制算术规范并未规定直接比较行为；这些涉及 NaN 的比较规则来自于 IEEE 854 标准 (见第 5.7 节表 3)。要确保严格符合标准，请改用*compare()* 和*compare_signal()* 方法。

有符号零值可以由向下溢出的运算产生。它们保留符号是为了让运算结果能以更高的精度传递。由于它们的大小为零，正零和负零会被视为相等，且它们的符号具有信息。

在这两个不相同但却相等的有符号零之外，还存在几种零的不同表示形式，它们的精度不同但值也都相等。这需要一些时间来逐渐适应。对于习惯了标准浮点表示形式的眼睛来说，以下运算返回等于零的值并不是显而易见的：

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 使用线程

getcontext() 函数会为每个线程访问不同的*Context* 对象。具有单独线程上下文意味着线程可以修改上下文 (例如 getcontext().prec=10) 而不影响其他线程。

类似的*setcontext()* 会为当前上下文的目标自动赋值。

如果在调用*setcontext()* 之前调用了*getcontext()*，则*getcontext()* 将自动创建一个新的上下文在当前线程中使用。

新的上下文拷贝自一个名为 *DefaultContext* 的原型上下文。要控制默认值以便每个线程在应用运行期间都使用相同的值，可以直接修改 *DefaultContext* 对象。这应当在任何线程启动 之前完成以使得调用*getcontext()* 的线程之间不会产生竞争条件。例如：

```
# 为所有将要启动的线程设置应用程序级默认值
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# 在此之后，即可启动线程
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 例程

以下是一些用作工具函数的例程，它们演示了使用*Decimal* 类的各种方式：

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """将 Decimal 值转换为以货币形式格式化的字符串。

    places:  小数点之后要保留的位数
    curr:    正负号之前可选的货币符号 (可以为空)
    sep:     可选的千位分隔符 (逗点, 句点, 空格或为空)
    dp:      小数点符号 (逗点或句点)
```

(续下页)

(接上页)

```
            仅在保留零位小数时设为空
    pos:      可选的正值符号: '+', 空格或为空
    neg:      可选的负值符号: '-', '(', 空格或为空
    trailneg:可选的末尾负值符号:  '-', ')', 空格或为空

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 个小数位 --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """计算 Pi 到当前精度。

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # 用于中间步骤的额外位数
    three = Decimal(3)      # 替换常规浮点数的 "three=3.0"
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # 单目取正值运算将应用新的精度
```

(续下页)

(接上页)

```
def exp(x):
    """返回 e 的 x 次方。 结果类型将与输入类型相匹配。

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """返回以弧度衡量的 x 的余弦。

    泰勒级数近似算法对较小的 x 值效果最好。
    对于较大的值，则先计算 x = x % (2 * pi)。

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """返回以弧度衡量的 x 的正弦。

    泰勒级数近似算法对较小的 x 值效果最好。
    对于较大的值，则先计算 x = x % (2 * pi)。

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
```

(续下页)

(接上页)

```
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal 常见问题

Q. 总是输入 `decimal.Decimal('1234.5')` 是否过于笨拙。在使用交互解释器时有没有最小化输入量的方式?

A. 有些用户会将构造器简写为一个字母:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. 在带有两个十进制位的定点数应用中，有些输入值具有许多位，需要被舍入。另一些数则不应具有多余位，需要验证有效性。这种情况应该用什么方法?

A. 用 *quantize()* 方法舍入到固定数目的十进制位。如果设置了 *Inexact* 陷阱，它也适用于验证有效性:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. 当我使用两个有效位的输入时，我要如何在一个应用中保持有效位不变?

A. 某些运算如与整数相加、相减和相乘将会自动保留固定的小数位数。其他运算，如相除和非整数相乘则会改变小数位数，需要再加上 *quantize()* 处理步骤:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
```

(续下页)

(接上页)

```
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)      # And quantize division
Decimal('0.03')
```

在开发定点数应用时，更方便的做法是定义处理*quantize()*步骤的函数:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
...
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                           # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. 表示同一个值有许多方式。数字 200, 200.000, 2E2 和 .02E+4 都具有相同的值但其精度不同。是否有办法将它们转换为一个可识别的规范值?

A. *normalize()* 方法可将所有相等的值映射为单一表示形式:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. 计算中的舍入是在什么时候发生的?

A. 是在计算 *之后* 发生的。decimal 设计规范认为数字应当被视为是精确的并且是不依赖于当前上下文而创建的。它们甚至可以具有比当前上下文更高的精确度。计算过程将使用精确的输入然后再对计算的 *结果* 应用舍入（或其他的上下文操作）:

```
>>> getcontext().prec = 5
>>> pi = Decimal('3.1415926535')    # 超过 5 个数位
>>> pi                              # 所有数位都将保留
Decimal('3.1415926535')
>>> pi + 0                          # 加法运算后将执行舍入
Decimal('3.1416')
>>> pi - Decimal('0.00005')         # 减去未舍入的数值，然后执行舍入
Decimal('3.1415')
>>> pi + 0 - Decimal('0.00005').    # 中间值将执行舍入
Decimal('3.1416')
```

Q. 有些十进制值总是被打印为指数表示形式。是否有办法得到一个非指数表示形式?

A. 对于某些值来说，指数表示法是表示系数中有效位的唯一方式。例如，将 5.0E+3 表示为 5000 可以让值保持恒定但是无法显示原本的两位有效位。

如果一个应用不必关心追踪有效位，则可以很容易地移除指数和末尾的零，丢弃有效位但让值保持不变:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. 是否有办法将一个普通浮点数转换为*Decimal*?

A. 是的，任何二进制浮点数都可以精确地表示为 Decimal 值，但完全精确的转换可能需要比平常感觉更高的精度:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. 在一个复杂的计算中，我怎样才能保证不会得到由精度不足和舍入异常所导致的虚假结果。

A. 使用 decimal 模块可以很容易地检测结果。最好的做法是使用更高的精度和不同的舍入模式重新进行计算。明显不同的结果表明存在精度不足、舍入模式问题、不符合条件的输入或是结果不稳定的算法。

Q. 我发现上下文精度的应用只针对运算结果而不针对输入。在混合使用不同精度的值时有什么需要注意的吗?

A. 是的。原则上所有值都会被视为精确值，在这些值上进行的算术运算也是如此。只有结果会被舍入。对于输入来说其好处是“所输入即所得”。而其缺点则是如果你忘记了输入没有被舍入，结果看起来可能会很奇怪：

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

解决办法是提高精度或使用单目加法运算对输入执行强制舍入：

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # 单目取正运算符将触发舍入
Decimal('1.23')
```

此外，还可以使用*Context.create_decimal()* 方法在创建输入时执行舍入：

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q. CPython 实现对于巨大数字是否足够快速?

A. 是的。在 CPython 和 PyPy3 实现中，decimal 模块的 C/CFFI 版本集成了高速 libmpdec 库用于实现任意精度正确舍入的十进制浮点算术[1]。`libmpdec` 会对中等大小的数字使用 Karatsuba 乘法 而对非常巨大的数字使用 数字原理变换。

上下文必须针对任意精度算术进行适配。`Emin` 和 `Emax` 应当总是被设为最大值，`clamp` 应当总是为 0 (默认值)。设置 `prec` 需要十分谨慎。requires some care.

进行大数字算术的最便捷方式同样是使用 `prec` 的最大值[2]:

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
Decimal('904625697166532776746648320380374280103671755200316906558262375061821325312')
```

对于不精确的结果，在 64 位平台上*MAX_PREC* 的值太大因而会导致可用内存不足:

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

在具有超量分配的系统上 (如 Linux)，一种更复杂的方式是根据可用的 RAM 大小来调整 `prec`。假设你有 8GB 的 RAM 并期望同时有 10 个操作数，每个最多使用 500MB:

[1] Added in version 3.3.

[2] 在 3.9 版本发生变更: 此方式现在适用于除了非整数乘方以外的所有精确结果。

```
>>> import sys
>>>
>>> # 字长 8 字节使用 500MB 的单个操作数的最大位数
>>> # 每字 19 个数位 (32 位构建版则为字长 4 字节每字 9 个数位):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # 检测此代码有效。
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # 以数字九填满可用的精度:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

总体而言（特别是在没有超量分配的系统上），如果期望所有计算都是精确的则推荐预估更严格的边界并设置*Inexact* 陷阱。

9.5 fractions --- 有理数

源代码 Lib/fractions.py

---

fractions 模块支持分数运算。

分数实例可以由一对整数，一个分数，或者一个字符串构建而成。

class fractions.**Fraction**(*numerator=0*, *denominator=1*)

class fractions.**Fraction**(*other_fraction*)

class fractions.**Fraction**(*float*)

class fractions.**Fraction**(*decimal*)

class fractions.**Fraction**(*string*)

第一个版本要求 *numerator* 和 *denominator* 是*numbers.Rational* 的实例，并返回一个值为 numerator/denominator 的新*Fraction* 实例。如果 *denominator* 是 0 则会引发*ZeroDivisionError*。第二个版本要求 *other_fraction* 是*numbers.Rational* 的实例，并返回具有相同值的*Fraction* 实例。接下来的两个版本接受*float* 或*decimal.Decimal* 实例，并返回具有完全相同值的*Fraction* 实例。请注意由于二进制浮点运算通常存在的问题 (参见 tut-fp-issues)，Fraction(1.1) 的参数并不完全等于 11/10，因此 Fraction(1.1) 也不会像人们所期望的那样返回 Fraction(11, 10)。(请参阅下面*limit_denominator()* 方法的文档。) 最后一个版本的构造器接受一个字符串或 unicode 实例。该实例的通常形式为:

```
[sign] numerator ['/' denominator]
```

其中的可选项 sign 可能为'+' 或'-' 且 numerator 和 denominator (如果存在) 是十进制数码的字符串 (可以如代码中的整数字面值一样使用下划线来分隔数码)。此外，*float* 构造器所接受的任何代表一个有限值的字符串也都为*Fraction* 构造器所接受。不论哪种形式的输入字符串也都可以带有开头和/或末尾空格符。这里是一些示例:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
```

(续下页)

(接上页)

```
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

Fraction 类继承自抽象基类*numbers.Rational*，并实现了该类的所有方法和操作。*Fraction* 实例是*hashable* 对象，并应当被视为不可变对象。此外，*Fraction* 还具有以下特征属性和方法:

在 3.2 版本发生变更: *Fraction* 构造器现在接受*float* 和*decimal.Decimal* 实例。

在 3.9 版本发生变更: 现在会使用*math.gcd()* 函数来正规化 *numerator* 和 *denominator*。*math.gcd()* 总是返回*int* 类型。在之前版本中，GCD 的类型取决于 *numerator* 和 *denominator* 的类型。

在 3.11 版本发生变更: 现在当使用字符串创建*Fraction* 实例时已允许使用下划线，遵循 **PEP 515** 规则。

在 3.11 版本发生变更: *Fraction* 现在实现了 `__int__` 以满足 `typing.SupportsInt` 实例检测。

在 3.12 版本发生变更: 允许字符串输入在斜杠两边添加空格: `Fraction('2 / 3')`。

在 3.12 版本发生变更: *Fraction* 实例现在支持浮点风格的格式化，使用 `"e"`, `"E"`, `"f"`, `"F"`, `"g"`, `"G"` 和 `"%"` 等表示类型。.

在 3.13 版本发生变更: 没有表示类型的*Fraction* 实例的格式化现在支持填充、对齐、正负号处理、最小宽度和分组。

numerator

最简分数形式的分子。

denominator

最简分数形式的分母。

as_integer_ratio()

返回由两个整数组成的元组，两数之比等于原 Fraction 的值且其分母为正数。

Added in version 3.8.

is_integer()

如果 Fraction 为整数则返回 `True`。

Added in version 3.12.

classmethod from_float(*flt*)

只接受*float* 或*numbers.Integral* 实例的替代性构造器。请注意 `Fraction.from_float(0.3)` 与 `Fraction(3, 10)` 的值是不同的。

> **备注**
>
> 从 Python 3.2 开始，在构造*Fraction* 实例时可以直接使用*float*。

classmethod from_decimal(*dec*)

只接受*decimal.Decimal* 或*numbers.Integral* 实例的替代性构造器。

> **备注**
>
> 从 Python 3.2 开始，在构造*Fraction* 实例时可以直接使用*decimal.Decimal* 实例。

limit_denominator(*max_denominator=1000000*)

找到并返回一个*Fraction* 使得其值最接近 `self` 并且分母不大于 max_denominator。此方法适用于找出给定浮点数的有理数近似值：

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

或是用来恢复被表示为一个浮点数的有理数：

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

返回最大的*int* `<= self`。此方法也可通过*math.floor()* 函数来使用：

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

返回最小的*int* `>= self`。此方法也可通过*math.ceil()* 函数来使用。

__round__()

__round__(*ndigits*)

第一个版本返回一个*int* 使得其值最接近 `self`，位值为二分之一时只对偶数舍入。第二个版本会将 `self` 舍入到最接近 `Fraction(1, 10**ndigits)` 的倍数（如果 `ndigits` 为负值则为逻辑运算），位值为二分之一时同样只对偶数舍入。此方法也可通过*round()* 函数来使用。

__format__(*format_spec*, */*)

通过*str.format()* 方法、*format()* 内置函数或 格式化字符串字面值提供对*Fraction* 实例格式化的支持。

如果 `format_spec` 格式说明字符串末尾不带表示类型 `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` 或 `'%'` 之一则格式化操作将遵循在格式说明微语言 中描述的有关填充、对齐、正负号处理、最小宽度和分组的一般规则。“替代形式” 旗标 `'#'` 也是受支持的：如果提供，将强制输出字符串始终包括一个显式的分母，即使被格式化的值恰好为整数也是如此。表示填充零值的旗标 `'0'` 是不被支持的。

如果 `format_spec` 格式说明字符串末尾带有表示类型 `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` 或 `'%'` 之一那么格式化操作将遵循在格式规格迷你语言 小节中针对*float* 类型所描述的规则。

这是一些例子:

```
>>> from fractions import Fraction
>>> format(Fraction(103993, 33102), '_')
'103_993/33_102'
>>> format(Fraction(1, 7), '.^+10')
'...+1/7...'
>>> format(Fraction(3, 1), '')
'3'
>>> format(Fraction(3, 1), '#')
'3/1'
>>> format(Fraction(1, 7), '.40g')
'0.1428571428571428571428571428571428571429'
>>> format(Fraction('1234567.855'), '_.2f')
'1_234_567.86'
>>> f"{Fraction(355, 113):*>20.6e}"
'********3.141593e+00'
>>> old_price, new_price = 499, 672
>>> "{:.2%} price increase".format(Fraction(new_price, old_price) - 1)
'34.67% price increase'
```

参见

numbers **模块**
构成数字塔的所有抽象基类。

9.6 random --- 生成伪随机数

源码: Lib/random.py

该模块实现了各种分布的伪随机数生成器。

对于整数，从范围中有统一的选择。对于序列，存在随机元素的统一选择、用于生成列表的随机排列的函数、以及用于随机抽样而无需替换的函数。

在实数轴上，有计算均匀、正态（高斯）、对数正态、负指数、伽马和贝塔分布的函数。为了生成角度分布，可以使用 von Mises 分布。

几乎所有模块函数都依赖于基本函数*random()*，它在左开右闭区间 `0.0 <= X < 1.0` 内均匀生成随机浮点数。Python 使用 Mersenne Twister 作为核心生成器。它产生 53 位精度的浮点数并且周期为 2**19937-1。其在 C 中的这个底层实现既快速又线程安全。Mersenne Twister 是目前经过最广泛测试的随机数生成器之一。但是，因为是完全确定性的，它不适用于所有目的，并且完全不适用于加密目的。

这个模块提供的函数实际上是*random.Random* 类的隐藏实例的绑定方法。你可以实例化自己的*Random* 类实例以获取不共享状态的生成器。

如果你想使用自己设计的不同的基本生成器那么也可以子类化*Random* 类：请参阅该类的文档了解详情。

random 模块还提供*SystemRandom* 类，它使用系统函数*os.urandom()* 从操作系统提供的源生成随机数。

警告

不应将此模块的伪随机生成器用于安全目的。有关安全性或加密用途，请参阅*secrets* 模块。

参见

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3--30 1998.

Complementary-Multiply-with-Carry recipe 用于兼容的替代性随机数发生器，具有长周期和相对简单的更新操作。

备注

全局随机数发生器和*Random* 实例是线程安全的。不过，在自由线程构建版中，对全局发生器或同一个*Random* 实例的并发调用可能导致竞争和糟糕的性能。请考虑改用每线程单独的*Random* 实例。

9.6.1 簿记功能

random.**seed**(*a=None*, *version=2*)

初始化随机数生成器。

如果 *a* 被省略或为 None，则使用当前系统时间。如果操作系统提供随机源，则使用它们而不是系统时间（有关可用性的详细信息，请参阅*os.urandom()* 函数）。

如果 *a* 是 int 类型，则直接使用。

对于版本 2（默认的），*str*、*bytes* 或*bytearray* 对象转换为*int* 并使用它的所有位。

对于版本 1（用于从旧版本的 Python 再现随机序列），用于*str* 和*bytes* 的算法生成更窄的种子范围。

在 3.2 版本发生变更: 已移至版本 2 方案，该方案使用字符串种子中的所有位。

在 3.11 版本发生变更: *seed* 必须是下列类型之一: None, *int*, *float*, *str*, *bytes* 或*bytearray*。

random.**getstate**()

返回捕获生成器当前内部状态的对象。这个对象可以传递给*setstate()* 来恢复状态。

random.**setstate**(*state*)

state 应该是从之前调用*getstate()* 获得的，并且*setstate()* 将生成器的内部状态恢复到*getstate()* 被调用时的状态。

9.6.2 用于字节数据的函数

random.**randbytes**(*n*)

生成 *n* 个随机字节。

此方法不可用于生成安全凭据。那应当使用*secrets.token_bytes()*。

Added in version 3.9.

9.6.3 整数用函数

random.**randrange**(*stop*)

random.**randrange**(*start*, *stop*[, *step*])

返回从 range(start, stop, step) 随机选择的一个元素。

这大致等价于 choice(range(start, stop, step)) 但是支持任意大的取值范围并针对常见场景进行了优化。

该位置参数的模式与*range()* 函数相匹配。

关键字参数不应被使用因为它们可能以预料之外的方式被解读。例如 `randrange(start=100)` 会被解读为 `randrange(0, 100, 1)`。

在 3.2 版本发生变更: *randrange()* 在生成均匀分布的值方面更为复杂。以前它使用了像 `int(random()*n)` 这样的形式，它可以产生稍微不均匀的分布。

在 3.12 版本发生变更: 不再支持非整数类型的自动转换。`randrange(10.0)` 和 `randrange(Fraction(10, 1))` 之类的调用现在将会引发*TypeError*。

`random.`**`randint`**(*a*, *b*)

返回随机整数 *N* 满足 `a <= N <= b`。相当于 `randrange(a, b+1)`。

`random.`**`getrandbits`**(*k*)

返回具有 *k* 个随机比特位的非负 Python 整数。此方法随 Mersenne Twister 生成器一起提供，其他一些生成器也可能将其作为 API 的可选部分提供。在可能的情况下，*getrandbits()* 会启用*randrange()* 来处理任意大的区间。

在 3.9 版本发生变更: 此方法现在接受零作为 *k* 的值。

9.6.4 序列用函数

`random.`**`choice`**(*seq*)

从非空序列 *seq* 返回一个随机元素。如果 *seq* 为空，则引发*IndexError*。

`random.`**`choices`**(*population*, *weights=None*, *, *cum_weights=None*, *k=1*)

从 *population* 中有重复地随机选取元素，返回大小为 *k* 的元素列表。如果 *population* 为空，则引发*IndexError*。

如果指定了 *weight* 序列，则根据相对权重进行选择。或者，如果给出 *cum_weights* 序列，则根据累积权重（可能使用*itertools.accumulate()* 计算）进行选择。例如，相对权重 `[10, 5, 30, 5]`相当于累积权重``[10, 15, 45, 50]。在内部，相对权重在进行选择之前会转换为累积权重，因此提供累积权重可以节省工作量。

如果既未指定 *weight* 也未指定 *cum_weights* ，则以相等的概率进行选择。如果提供了权重序列，则它必须与 *population* 序列的长度相同。一个*TypeError* 指定了 *weights* 和 *cum_weights* 。

weights 或 *cum_weights* 可使用*random()* 所返回的能与*float* 值进行相互运算的任何数字类型（包括整数、浮点数、分数但不包括 decimal）。权重值应当非负且为有限的数值。如果所有的权重值均为零则会引发*ValueError*。

对于给定的种子，具有相等加权的*choices()* 函数通常产生与重复调用*choice()* 不同的序列。*choices()* 使用的算法使用浮点运算来实现内部一致性和速度。*choice()* 使用的算法默认为重复选择的整数运算，以避免因舍入误差引起的小偏差。

Added in version 3.6.

在 3.9 版本发生变更: 如果所有权重均为负值则将引发*ValueError*。

`random.`**`shuffle`**(*x*)

就地将序列 *x* 随机打乱位置。

要改变一个不可变的序列并返回一个新的打乱列表，请使用 `sample(x, k=len(x))`。

请注意，即使对于小的 `len(x)`，*x* 的排列总数也可以快速增长，大于大多数随机数生成器的周期。这意味着长序列的大多数排列永远不会产生。例如，长度为 2080 的序列是可以在 Mersenne Twister 随机数生成器的周期内拟合的最大序列。

在 3.11 版本发生变更: 移除了可选的形参 *random*。

`random.`**`sample`**(*population*, *k*, *, *counts=None*)

返回从总体序列中选取的唯一元素的长度为 *k* 的列表。用于无重复的随机抽样。

返回包含来自总体的元素的新列表，同时保持原始总体不变。结果列表按选择顺序排列，因此所有子切片也将是有效的随机样本。这允许抽奖获奖者（样本）被划分为大奖和第二名获胜者（子切片）。

总体成员不必是*hashable* 或 unique 。如果总体包含重复，则每次出现都是样本中可能的选择。

重复的元素可以一个个地直接列出，或使用可选的仅限关键字形参 *counts* 来指定。例如，`sample(['red', 'blue'], counts=[4, 2], k=5)` 等价于 `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`。

要从一系列整数中选择样本，请使用*range()* 对象作为参数。对于从大量人群中采样，这种方法特别快速且节省空间：`sample(range(10000000), k=60)` 。

如果样本大小大于总体大小，则引发*ValueError* 。

在 3.9 版本发生变更: 增加了 *counts* 形参。

在 3.11 版本发生变更: *population* 必须为一个序列。不再支持将集合自动转换为列表。

9.6.5 离散分布

以下函数会生成离散分布。

`random.`**`binomialvariate`**(*n=1*, *p=0.5*)

二项式分布。返回 *n* 次独立试验在每次试验的成功率为 *p* 时的成功次数:

在数学上等价于:

```
sum(random() < p for i in range(n))
```

试验次数 *n* 应为一个非负整数。成功几率 *p* 的取值范围应为 `0.0 <= p <= 1.0`。结果是一个 `0 <= X <= n` 范围内的整数。

Added in version 3.12.

9.6.6 实值分布

以下函数生成特定的实值分布。如常用数学实践中所使用的那样，函数形参以分布方程中的相应变量命名，大多数这些方程都可以在任何统计学教材中找到。

`random.`**`random`**()

返回 `0.0 <= X < 1.0` 范围内的下一个随机浮点数。

`random.`**`uniform`**(*a*, *b*)

返回一个随机浮点数 *N* ，当 `a <= b` 时 `a <= N <= b` ，当 `b < a` 时 `b <= N <= a` 。

终点值 `b` 可能包括或不包括在该范围内，具体取决于表达式 `a + (b-a) * random()` 的浮点舍入结果。

`random.`**`triangular`**(*low*, *high*, *mode*)

返回一个随机浮点数 *N* ，使得 `low <= N <= high` 并在这些边界之间使用指定的 *mode* 。*low* 和 *high* 边界默认为零和一。*mode* 参数默认为边界之间的中点，给出对称分布。

`random.`**`betavariate`**(*alpha*, *beta*)

Beta 分布。参数的条件是 `alpha > 0` 和 `beta > 0`。返回值的范围介于 0 和 1 之间。

`random.`**`expovariate`**(*lambd=1.0*)

指数分布。*lambd* 是 1.0 除以所需的平均值，它应该是非零的。（该参数本应命名为 “lambda”，但这是 Python 中的保留字。）如果 *lambd* 为正，则返回值的范围为 0 到正无穷大；如果 *lambd* 为负，则返回值从负无穷大到 0。

在 3.12 版本发生变更: 添加了 `lambd` 的默认值。

`random.`**`gammavariate`**(*alpha*, *beta*)

Gamma 分布。(不是 gamma 函数！) shape 和 scale 形参，即 *alpha* 和 *beta*，必须为正值。(调用规范有变动并且有些源码会将'beta' 定义为逆向的 scale)。

概率分布函数是:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu=0.0*, *sigma=1.0*)

正态分布，也称高斯分布。*mu* 为平均值，而 *sigma* 为标准差。此函数要稍快于下面所定义的*normalvariate()* 函数。

多线程注意事项：当两个线程同时调用此方法时，它们有可能将获得相同的返回值。这可以通过三种办法来避免。1) 让每个线程使用不同的随机数生成器实例。2) 在所有调用外面加锁。3) 改用速度较慢但是线程安全的*normalvariate()* 函数。

在 3.11 版本发生变更: 现在 *mu* 和 *sigma* 均有默认参数。

random.**lognormvariate**(*mu*, *sigma*)

对数正态分布。如果你采用这个分布的自然对数，你将得到一个正态分布，平均值为 *mu* 和标准差为 *sigma* 。*mu* 可以是任何值，*sigma* 必须大于零。

random.**normalvariate**(*mu=0.0*, *sigma=1.0*)

正态分布。*mu* 是平均值，*sigma* 是标准差。

在 3.11 版本发生变更: 现在 *mu* 和 *sigma* 均有默认参数。

random.**vonmisesvariate**(*mu*, *kappa*)

冯 · 米塞斯分布。*mu* 是平均角度，以弧度表示，介于 0 和 2**pi* 之间，*kappa* 是浓度参数，必须大于或等于零。如果 *kappa* 等于零，则该分布在 0 到 2**pi* 的范围内减小到均匀的随机角度。

random.**paretovariate**(*alpha*)

帕累托分布。*alpha* 是形状参数。

random.**weibullvariate**(*alpha*, *beta*)

威布尔分布。*alpha* 是比例参数，*beta* 是形状参数。

9.6.7 替代生成器

class random.**Random**([*seed*])

该类实现了*random* 模块所用的默认伪随机数生成器。

在 3.11 版本发生变更: 之前 *seed* 可以是任何可哈希对象。现在它被限制为: None, *int*, *float*, *str*, *bytes* 或*bytearray*。

Random 的子类如果想要使用不同的基本生成器则应当重载下列方法：

seed(*a=None*, *version=2*)

在子类中重写此方法以自定义 Random 实例的*seed()* 行为。

getstate()

在子类中重写此方法以自定义 Random 实例的*getstate()* 行为。

setstate(*state*)

在子类中重写此方法以自定义 Random 实例的*setstate()* 行为。

random()

在子类中重写此方法以自定义 Random 实例的*random()* 行为。

作为可选项，自定义的生成器子类还可以提供下列方法：

getrandbits(*k*)

在子类中重写此方法以自定义 Random 实例的*getrandbits()* 行为。

class random.**SystemRandom**([*seed*])

使用*os.urandom()* 函数的类，用从操作系统提供的源生成随机数。这并非适用于所有系统。也不依赖于软件状态，序列不可重现。因此，*seed()* 方法没有效果而被忽略。*getstate()* 和*setstate()* 方法如果被调用则引发*NotImplementedError*。

9.6.8 关于再现性的说明

有时能够重现伪随机数生成器给出的序列是很有用处的。通过重用一个种子值，只要没有运行多线程，相同的序列就应当可在多次运行中重现。

大多数随机模块的算法和种子函数都会在 Python 版本中发生变化，但保证两个方面不会改变：

- 如果添加了新的播种方法，则将提供向后兼容的播种机。
- 当兼容的播种机被赋予相同的种子时，生成器的*random()* 方法将继续产生相同的序列。

9.6.9 例子

基本示例:

```
>>> random()                            # 随机浮点数:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # 随机浮点数:  2.5 <= x <= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # 到达的间隔平均为 5 秒
5.148957571865031

>>> randrange(10)                       # 从 0 至 9 区间内的整数
7

>>> randrange(0, 101, 2)                # 从 0 至 100 区间内的偶数
26

>>> choice(['win', 'lose', 'draw'])     # 从序列中取单个随机元素
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # 打乱列表
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # 不重复地取四个样本
[40, 10, 50, 30]
```

模拟:

```
>>> # 六次轮盘旋转 (可重复的带权重取样)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # 从一套扑克 52 张牌中不重复的抽取 20 张,
>>> # 并以十以上的大牌来确定牌的对比值:
>>> # 10、J、Q 或 K。
>>> deal = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> deal.count('tens') / 20
0.15

>>> # 估计一个被做了手脚的正面朝上概率为 60% 硬币
>>> # 在 7 次抛掷中得到 5 次及以上正面的概率。
>>> sum(binomialvariate(n=7, p=0.6) >= 5 for i in range(10_000)) / 10_000
0.4169

>>> # 5 个样本的中位数位于中间两个四分位区之内的概率
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
```

(续下页)

(接上页)

```
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

statistical bootstrapping 的示例，使用重新采样和替换来估计一个样本的均值的置信区间:

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

使用 重新采样排列测试 来确定统计学显著性或者使用 p-值 来观察药物与安慰剂的作用之间差异的示例:

```
# 来自 Dennis Shasha 与 Manda Wilson 所著 "Statistics is Easy" 的示例
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

多服务器队列的到达时间和服务交付模拟:

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # 每个服务器都可用的时刻
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)
```

(续下页)

(接上页)

```
print(f'Mean wait: {mean(waits):.1f}   Max wait: {max(waits):.1f}')
print('Quartiles:', [round(q, 1) for q in quantiles(waits)])
```

参见

Statistics for Hackers Jake Vanderplas 撰写的视频教程，使用一些基本概念进行统计分析，包括模拟、抽样、洗牌和交叉验证。

Economics Simulation 是 Peter Norvig 编写的市场模拟，它演示了对此模块所提供的许多工具和分布（gauss, uniform, sample, betavariate, choice, triangular 和 randrange）的高效运用。

A Concrete Introduction to Probability (using Python) 是 Peter Norvig 撰写的教程，其中涉及概率论基础、如何编写模拟以及如何使用 Python 进行数据分析等内容。

9.6.10 例程

这些例程演示了如何有效地使用 *itertools* 模块中的组合迭代器进行随机选取:

```
def random_product(*args, repeat=1):
    "从 itertools.product(*args, **kwds) 随机选取"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "从 itertools.permutations(iterable, r) 随机选取"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "从 itertools.combinations(iterable, r) 随机选取"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "可重复地选取 r 个元素。 对结果进行排序以与 iterable 相匹配。"
    # 结果将在 set(itertools.combinations_with_replacement(iterable, r)) 中。
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)
```

默认的 *random()* 返回在 *0.0 ≤ x < 1.0* 范围内 2^{-53} 的倍数。所有这些数值间隔相等并能精确表示为 Python 浮点数。但是在此间隔上有许多其他可表示浮点数是不可能的选择。例如，`0.05954861408025609` 就不是 2^{-53} 的整数倍。

以下规范程序采取了一种不同的方式。在间隔上的所有浮点数都是可能的选择。它们的尾数取值来自 $2^{52} \leq$ 尾数 $< 2^{53}$ 范围内整数的均匀分布。指数取值则来自几何分布，其中小于 *-53* 的指数的出现频率为下一个较大指数的一半。

```
from random import Random
from math import ldexp

class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
```

(续下页)

(接上页)

```
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

该类中所有的*实值分布* 都将使用新的方法:

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

该规范程序在概念上等效于在 *0.0 ≤ x < 1.0* 范围内对所有 2^{-1074} 的倍数进行选择的算法。所有这样的数字间隔都相等，但大多必须向下舍入为最接近的 Python 浮点数表示形式。（2^{-1074} 这个数值是等于 `math.ulp(0.0)` 的未经正规化的最小正浮点数。）

参见

生成伪随机浮点数值 为 Allen B. Downey 所撰写的描述如何生成相比通过*random()* 正常生成的数值更细粒度浮点数的论文。

9.6.11 命令行用法

Added in version 3.13.

`random` 模块可以在命令行中执行。

```
python -m random [-h] [-c CHOICE [CHOICE ...] | -i N | -f N] [input ...]
```

可以接受以下选项:

-h, --help
显示帮助信息并退出。

-c CHOICE [CHOICE ...]
--choice CHOICE [CHOICE ...]
使用*choice()* 打印一个随机选择的项。

-i <N>
--integer <N>
使用*randint()* 打印从 1 到 N 闭区间中的一个随机整数。

-f <N>
--float <N>
使用*uniform()* 打印从 0 到 N 闭区间中的一个随机浮点数。

如果未给出任何选项，输出将取决于输入:

- 字符串或多个字符串: 与*--choice* 相同。
- 整数: 与*--integer* 相同。
- 浮点数: 与*--float* 相同。

9.6.12 命令行示例

下面是一些 random 命令行接口的示例：

```
$ # 随机选择一项
$ python -m random egg bacon sausage spam "Lobster Thermidor aux crevettes with a Mornay␣
↪sauce"
Lobster Thermidor aux crevettes with a Mornay sauce

$ # 随机整数
$ python -m random 6
6

$ # 随机浮点数
$ python -m random 1.8
1.7080016272295635

$ # 带有显式的参数
$ python  -m random --choice egg bacon sausage spam "Lobster Thermidor aux crevettes with a␣
↪Mornay sauce"
egg

$ python -m random --integer 6
3

$ python -m random --float 1.8
1.5666339105010318

$ python -m random --integer 6
5

$ python -m random --float 6
3.1942323316565915
```

9.7 statistics --- 数字统计函数

Added in version 3.4.

源代码: Lib/statistics.py

该模块提供了用于计算数字 (*Real*-valued) 数据的数理统计量的函数。

此模块并不是诸如 NumPy, SciPy 等第三方库或者诸如 Minitab, SAS 和 Matlab 等针对专业统计学家的专有全功能统计软件包的竞品。此模块针对图形和科学计算器的水平。

除非明确注释，这些函数支持 *int* ， *float* ， *Decimal* 和 *Fraction* 。当前不支持同其他类型（是否在数字塔中）的行为。混合类型的集合也是未定义的，并且依赖于实现。如果你输入的数据由混合类型组成，你应该能够使用 *map()* 来确保一个一致的结果，比如：`map(float, input_data)` 。

某些数据集合类型使用 `NaN` (not a number) 值来代表缺失的数据。由于 NaN 具有特殊的比较语义，它们会在数据排序或计数等统计函数中产生怪异或未定义的行为。受影响的函数有 `median()`, `median_low()`, `median_high()`, `median_grouped()`, `mode()`, `multimode()` 和 `quantiles()`。`NaN` 值应当在调用这些函数之前被去除：

```
>>> from statistics import median
>>> from math import isnan
>>> from itertools import filterfalse

>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)  # 这将有令人惊讶的行为
[20.7, nan, 14.4, 18.3, 19.2, nan]
```

(续下页)

(接上页)

```
>>> median(data)  # 这个结果不符合预期
16.35

>>> sum(map(isnan, data))    # 缺失值的数量
2
>>> clean = list(filterfalse(isnan, data))  # 去除 NaN 值
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)  # 现在排序将符合预期
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)       # 现在这个结果有良好定义的
18.75
```

9.7.1 平均值以及对中心位置的评估

这些函数用于计算一个总体或样本的平均值或者典型值。

mean()	数据的算术平均数（“平均数”）。
fmean()	快速的浮点算术平均值，带有可选的权重设置。
geometric_mean()	数据的几何平均数
harmonic_mean()	数据的调和均值
kde()	估算数据的概率密度分布。
kde_random()	对由 kde() 生成的 PDF 进行随机采样。
median()	数据的中位数（中间值）
median_low()	数据的低中位数
median_high()	数据的高中位数
median_grouped()	分组数据的中位数（即第 50 个百分点的位置）。
mode()	离散的或标称的数据的单个众数（出现最多的值）。
multimode()	离散的或标称的数据的众数（出现最多的值）列表。
quantiles()	将数据以相等的概率分为多个间隔。

9.7.2 对分散程度的评估

这些函数用于计算总体或样本与典型值或平均值的偏离程度。

pstdev()	数据的总体标准差
pvariance()	数据的总体方差
stdev()	数据的样本标准差
variance()	数据的样本方差

9.7.3 对两个输入之间关系的统计

这些函数计算两个输入之间关系的统计值。

covariance()	两个变量的样本协方差。
correlation()	皮尔逊和斯皮尔曼相关系数。
linear_regression()	简单线性回归的斜率和截距。

9.7.4 函数细节

注释：这些函数不需要对提供给它们的数据进行排序。但是，为了方便阅读，大多数例子展示的是已排序的序列。

statistics.**mean**(*data*)

返回 *data* 的样本算术平均数，形式为序列或迭代器。

算术平均数是数据之和与数据点个数的商。通常称作“平均数”，尽管它指示诸多数学平均数之一。它是数据的中心位置的度量。

若 *data* 为空，将会引发*StatisticsError*。

一些用法示例：

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

备注

平均数会受到 异常值 的强烈影响因而不一定能作为数据点的典型样本。想获得对于 集中趋势 的更可靠的度量，可以参看*median()*，但其效率要低一些。

样本均值给出了一个无偏向的真实总体均值的估计，因此当平均抽取所有可能的样本，`mean(sample)` 收敛于整个总体的真实均值。如果 *data* 代表整个总体而不是样本，那么 `mean(data)` 等同于计算真实整体均值 μ。

statistics.**fmean**(*data*, *weights=None*)

将 *data* 转换成浮点数并且计算算术平均数。

此函数的运行速度比*mean()* 函数快并且它总是返回一个*float*。*data* 可以为序列或可迭代对象。如果输入数据集为空，则会引发*StatisticsError*。

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

支持可选的权重参数。例如，某位教授在为课程打分时可设置权重为测验 20%, 作业 20%, 期中考试 30%, 期末考试 30%:

```
>>> grades = [85, 92, 83, 91]
>>> weights = [0.20, 0.20, 0.30, 0.30]
>>> fmean(grades, weights)
87.6
```

如果提供了 *weights*，它必须与 *data* 的长度相同否则将引发*ValueError*。

Added in version 3.8.

在 3.11 版本发生变更: 添加了对 *weights* 的支持。

statistics.**geometric_mean**(*data*)

将 *data* 转换成浮点数并且计算几何平均数。

几何平均值使用值的乘积表示 数据的中心趋势或典型值（与使用它们的总和的算术平均值相反）。

如果输入数据集为空、包含零或包含负值则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

无需做出特殊努力即可获得准确的结果。（但是，将来或许会修改。）

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

Added in version 3.8.

statistics.**harmonic_mean**(*data*, *weights=None*)

返回包含实数的序列或可迭代对象 *data* 的调和平均值。如果 *weights* 被省略或为 None，则会假定为相等权重。

调和平均数是数据的倒数的算术平均值*mean()* 的倒数。例如，三个数值 *a*, *b* 和 *c* 的调和平均数将等于 3/(1/a + 1/b + 1/c)。如果其中一个值为零，则结果也将为零。

调和平均数是均值的一种，是对数据的中心位置的度量。它通常适用于求比率和比例（如速度）的均值。

假设一辆车在 40 km/hr 的速度下行驶了 10 km，然后又以 60 km/hr 的速度行驶了 10 km。车辆的平均速率是多少?

```
>>> harmonic_mean([40, 60])
48.0
```

假设一辆汽车以速度 40 公里/小时行驶了 5 公里，当道路变得畅通后，提速到 60 公里/小时行驶了行程中剩余的 30 km。请问其平均速度是多少?

```
>>> harmonic_mean([40, 60], weights=[5, 30])
56.0
```

如果 *data* 为空、任意元素小于零，或者加权汇总值不为正数则会引发*StatisticsError*。

当前算法在输入中遇到零时会提前退出。这意味着不会测试后续输入的有效性。（此行为将来可能会更改。）

Added in version 3.6.

在 3.10 版本发生变更: 添加了对 *weights* 的支持。

statistics.**kde**(*data*, *h*, *kernel='normal'*, *, *cumulative=False*)

核密度估计 (KDE): 基于离散的样本创建一个连续概率密度函数或累积分布函数。

其基本思路是使用 核函数 来平滑数据。以帮助根据一个样本来推断总体情况。

平滑等级是由被称为“带宽”的缩放形参 *h* 来控制的。较小的值将强调局部特性而较大的值将给出更平滑的结果。

kernel 确定样本数据点的相对权重。通常，对核形状的选择带来的影响没有对带宽平滑形参的选择那样大。

为每个样本点都给出一定权重的核包括 *normal* (*gauss*), *logistic* 和 *sigmoid*。

只为带宽范围内的样本点给出权重的核包括 *rectangular* (*uniform*), *triangular*, *parabolic* (*epanechnikov*), *quartic* (*biweight*), *triweight* 和 *cosine*。

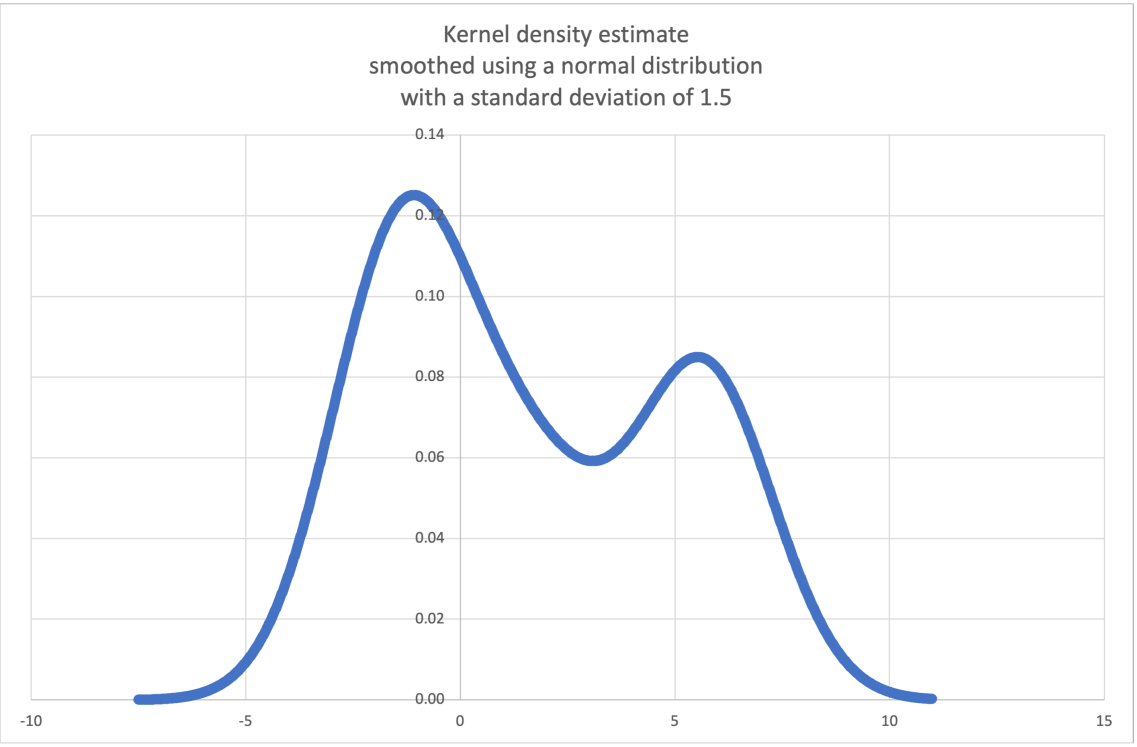
如果 *cumulative* 为真值，将返回一个累积分布函数。

如果 *data* 序列为空则会引发*StatisticsError*。

在 Wikipedia 提供的示例 中我们可以使用*kde()* 来生成并绘制从小样本中估算出的概率密度函数:

```
>>> sample = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> f_hat = kde(sample, h=1.5)
>>> xarr = [i/100 for i in range(-750, 1100)]
>>> yarr = [f_hat(x) for x in xarr]
```

xarr 和 yarr 中的点可被用来绘制一个 PDF 图形:



Added in version 3.13.

statistics.**kde_random**(*data*, *h*, *kernel='normal'*, *, *seed=None*)

返回一个函数，从 `kde(data, h, kernel)` 产生的估计概率密度函数中执行一次随机选择。

提供 *seed* 将允许可重现的选择。在未来版本中，这些值可能因更精确的反向 CDF 估计的实现而略微修改。seed 可以是一个整数、浮点数、字符串或字节串。

如果 *data* 序列为空则会引发*StatisticsError*。

继续*kde()* 的例子，我们可以使用*kde_random()* 从一个估计概率密度函数生成新的随机选择：

```
>>> data = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> rand = kde_random(data, h=1.5, seed=8675309)
>>> new_selections = [rand() for i in range(10)]
>>> [round(x, 1) for x in new_selections]
[0.7, 6.2, 1.2, 6.9, 7.0, 1.8, 2.5, -0.5, -1.8, 5.6]
```

Added in version 3.13.

statistics.**median**(*data*)

使用普通的“取中间两数平均值”方法返回数值数据的中位数（中间值）。如果 *data* 为空，则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

中位数是衡量中间位置的可靠方式，并且较少受到极端值的影响。当数据点的总数为奇数时，将返回中间数据点：

```
>>> median([1, 3, 5])
3
```

当数据点的总数为偶数时，中位数将通过对两个中间值求平均进行插值得出：

```
>>> median([1, 3, 5, 7])
4.0
```

这适用于当你的数据是离散的，并且你不介意中位数不是实际数据点的情况。

如果数据是有序的（支持排序操作）但不是数字（不支持加法），请考虑改用*median_low()* 或*median_high()*。

statistics.**median_low**(*data*)

返回数值数据的低中位数。如果 *data* 为空则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

低中位数一定是数据集的成员。当数据点总数为奇数时，将返回中间值。当其为偶数时，将返回两个中间值中较小的那个。

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

当你的数据是离散的，并且你希望中位数是一个实际数据点而非插值结果时可以使用低中位数。

statistics.**median_high**(*data*)

返回数据的高中位数。如果 *data* 为空则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

高中位数一定是数据集的成员。当数据点总数为奇数时，将返回中间值。当其为偶数时，将返回两个中间值中较大的那个。

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

当你的数据是离散的，并且你希望中位数是一个实际数据点而非插值结果时可以使用高中位数。

statistics.**median_grouped**(*data*, *interval=1.0*)

针对围绕连续的、固定宽度区间的中点进行了 分组或分档 的数值数据估算中位数。

data 可以是任意数值数据的可迭代对象，其中每个值都恰好为分档的中点。至少必须有一个值。

interval 是每个分档的宽度。

例如，人口信息可能被归纳为按 10 年划分的连续年龄分组，每个分组由各区间的 5 年中点来表示：

```
>>> from collections import Counter
>>> demographics = Counter({
...    25: 172,   # 20 至 30 岁
...    35: 484,   # 30 至 40 岁
...    45: 387,   # 40 至 50 岁
...    55:  22,   # 50 至 60 岁
...    65:   6,   # 60 至 70 岁
... })
...
```

第 50 个百分点位置（中位数）就是 1071 名成员中的第 536 人。此人属于 30 至 40 岁年龄分组。

常规的*median()* 函数会假定三十至四十岁年龄组中的每个人都正好是 35 岁。一个更站得住脚的假设则是该年龄组的 484 名成员均匀分布在 30 岁到 40 岁之间。为此，我们会使用*median_grouped()*：

```
>>> data = list(demographics.elements())
>>> median(data)
35
>>> round(median_grouped(data, interval=10), 1)
37.5
```

调用者有责任确保数据点之间以 *interval* 的精确倍数分隔。这对于获得正确结果至关重要。该函数不会检查这一前提条件。

输入可以是任何可在插值步骤中强制转换为浮点数的数值类型。

statistics.**mode**(*data*)

从离散或标称的 *data* 返回单个出现最多的数据点。此众数（如果存在）是最典型的值，并可用来度量中心的位置。

如果存在具有相同频率的多个众数，则返回在 *data* 中遇到的第一个。如果想要其中最小或最大的一个，请使用 `min(multimode(data))` 或 `max(multimode(data))`。如果输入的 *data* 为空，则会引发*StatisticsError*。

`mode` 将假定是离散数据并返回一个单一的值。这是通常的学校教学中标准的处理方式：

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

此众数的独特之处在于它是这个包中唯一还可应用于标称（非数字）数据的统计信息：

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

仅支持输入可哈希对象。要处理*set*类型，可将其转换为*frozenset*。要处理*list*类型，可将其转换为*tuple*。对于混合的或嵌套的输入，可使用这个仅依赖于相等性检测的速度较慢的二次方复杂度算法: `max(data, key=data.count)`。

在 3.8 版本发生变更: 现在会通过返回所遇到的第一个众数来处理多模数据集。之前它会在遇到超过一个的众数时引发*StatisticsError*。

statistics.**multimode**(*data*)

返回最频繁出现的值的列表，并按它们在 *data* 中首次出现的位置排序。如果存在多个众数则将返回一个以上的众数，或者如果 *data* 为空则将返回空列表：

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

Added in version 3.8.

statistics.**pstdev**(*data*, *mu=None*)

返回总体标准差（总体方差的平方根）。请参阅*pvariance()* 了解参数和其他细节。

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

返回非空序列或包含实数值的可迭代对象 *data* 的总体方差。方差或称相对于均值的二阶距，是对数据变化幅度（延展度或分散度）的度量。方差值较大表明数据的散布范围较大；方差值较小表明它紧密聚集于均值附近。

如果给出了可选的第二个参数 *mu*，它应为 *data* 的 *众数*均值。它也可以被用来计算一个非均值点的二阶距。如果该参数被省略或为 `None` (默认值)，则会自动进行算术均值计算。

使用此函数可根据所有数值来计算方差。要根据一个样本来估算方差，通常*variance()* 函数是更好的选择。

如果 *data* 为空则会引发*StatisticsError*。

示例：

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

如果你已经计算过数据的平均值，你可以将其作为可选的第二个参数 *mu* 传入以避免重复计算：

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

同样也支持使用 Decimal 和 Fraction 值：

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

备注

当调用时附带完整的总体数据时，这将给出总体方差 σ^2。而当调用时只附带一个样本时，这将给出偏置样本方差 s^2，也被称为带有 N 个自由度的方差。

如果你通过某种方式知道了真实的总体平均值 μ，则可以使用此函数来计算一个样本的方差，并将已知的总体平均值作为第二个参数。假设数据点是总体的一个随机样本，则结果将为总体方差的无偏估计值。

statistics.**stdev**(*data*, *xbar=None*)

返回样本标准差（样本方差的平方根）。请参阅*variance()* 了解参数和其他细节。

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

返回包含至少两个实数值的可迭代对象 *data* 的样本方差。方差或称相对于均值的二阶矩，是对数据变化幅度（延展度或分散度）的度量。方差值较大表明数据的散布范围较大；方差值较小表明它紧密聚集于均值附近。

如果给出了可选的第二个参数 *xbar*，它应为 *data* 的样本均值。如果该参数省略或为 None (默认值)，则会自动进行均值计算。

当你的数据是总体数据的样本时请使用此函数。要根据整个总体数据来计算方差，请参见*pvariance()*。

如果 *data* 包含的值少于两个则会引发*StatisticsError*。

示例：

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

如果你已经计算过数据的平均值，你可以将其作为可选的第二个参数 *xbar* 传入以避免重复计算：

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

此函数不会试图检查你所传入的 *xbar* 是否为真实的平均值。使用任意值作为 *xbar* 可能导致无效或不可能的结果。

同样也支持使用 Decimal 和 Fraction 值：

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

> **备注**
>
> 这是附带贝塞尔校正的样本方差 s^2，也称为具有 N-1 自由度的方差。假设数据点具有代表性（即为独立且均匀的分布），则结果应当是对总体方差的无偏估计。
>
> 如果你通过某种方式知道了真实的总体平均值 μ 则应当调用*pvariance()* 函数并将该值作为 *mu* 形参传入以得到一个样本的方差。

statistics.**quantiles**(*data*, ***, *n=4*, *method='exclusive'*)

将 *data* 分隔为具有相等概率的 *n* 个连续区间。返回分隔这些区间的 `n - 1` 个分隔点的列表。

将 *n* 设为 4 以使用四分位（默认值）。将 *n* 设为 10 以使用十分位。将 *n* 设为 100 以使用百分位，即给出 99 个分隔点来将 *data* 分隔为 100 个大小相等的组。如果 *n* 小于 1 则将引发*StatisticsError*。

data 可以是包含样本数据的任意可迭代对象。为了获得有意义的结果，*data* 中数据点的数量应当大于 *n*。如果连一个数据点都没有则会引发*StatisticsError*。

分隔点是通过对两个最接近的数据点进行线性插值得到的。例如，如果一个分隔点落在两个样本值 `100` 和 `112` 之间距离三分之一的位置，则分隔点的取值将为 `104`。

method 用于计算分位值，它会由于 *data* 是包含还是排除总体的最低和最高可能值而有所不同。

默认 *method* 是“唯一的”并且被用于在总体中数据采样这样可以有比样本中找到的更多的极端值。落在 *m* 个排序数据点的第 *i-th* 个以下的总体部分被计算为 `i / (m + 1)`。给定九个样本值，方法排序它们并且分配一下的百分位：10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% 。

将 *method* 设为"inclusive" 可用于描述总体数据或已明确知道包含有总体数据中最极端值的样本。*data* 中的最小值会被作为第 0 个百分位而最大值会被作为第 100 个百分位。总体数据里处于 *m* 个已排序数据点中 第 *i* 个以下的部分会以 `(i - 1) / (m - 1)` 来计算。给定 11 个样本值，该方法会对它们进行排序并赋予以下百分位: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%。

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

Added in version 3.8.

在 3.13 版本发生变更: 对于只有单个数据点的输入不会再引发异常。这允许分位点估计以每次一个样本点的方式建立并随着每个新数据点逐渐变得更为精细。

statistics.**covariance**(*x*, *y*, */*)

返回两个输入 *x* 和 *y* 的样本协方差。样本协方差是对两个输入的同步变化性的度量。

两个输入必须具有相同的长度（不少于两个元素），否则会引发*StatisticsError*。

示例：

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> z = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> covariance(x, z)
-7.5
>>> covariance(z, x)
-7.5
```

Added in version 3.10.

statistics.**correlation**(*x*, *y*, */*, ***, *method='linear'*)

返回两个输入的 皮尔逊相关系数。皮尔逊相关系数 *r* 的取值在 -1 到 +1 之间。它衡量线性相关的强度和方向。

如果 *method* 为"ranked"，则计算两个输入的 斯皮尔曼等级相关系数。数据将被替换为等级。同级的值将被平均因此相同的值将得到相同的等级。结果系数衡量的是单调关系的强度。

斯皮尔曼相关系数适用于有序数据或不满足皮尔逊相关系数的线性比例要求的连续数据。

两个输入必须具有相同的长度（不少于两个元素），并且不必为常量，否则会引发*StatisticsError*。

使用 开普勒行星运动定律 的示例:

```
>>> # 水星、金星、地球、火星、木星、土星、天王星和海王星
>>> orbital_period = [88, 225, 365, 687, 4331, 10_756, 30_687, 60_190]    # 天
>>> dist_from_sun = [58, 108, 150, 228, 778, 1_400, 2_900, 4_500] # 百万公里

>>> # 显示存在完美的单调关系
>>> correlation(orbital_period, dist_from_sun, method='ranked')
1.0

>>> # 表明存在不完美的线性关系
>>> round(correlation(orbital_period, dist_from_sun), 4)
0.9882

>>> # 体现开普勒第三定律：自转周期的
>>> # 平方和与太阳距离的立方之间存在
>>> # 线性对应关系
>>> period_squared = [p * p for p in orbital_period]
>>> dist_cubed = [d * d * d for d in dist_from_sun]
>>> round(correlation(period_squared, dist_cubed), 4)
1.0
```

Added in version 3.10.

在 3.12 版本发生变更: 增加了对斯皮尔曼等级相关系数的支持。

statistics.**linear_regression**(*x*, *y*, */*, ***, *proportional=False*)

返回使用普通最小二乘法估计得到的 简单线性回归 参数的斜率和截距。简单纯属回归通过此线性函数来描述自变量 *x* 和因变量 *y* 之间的关系。

*y = slope * x + intercept + noise*

其中 `slope` 和 `intercept` 是估计得到的回归参数，而 `noise` 代表不可由线性回归解释的数据变异性（它等于因变量的预测值和实际值之间的差异）。

两个输入必须具有相同的长度（不少于两个元素），并且自变量 *x* 不可为常量；否则会引发*StatisticsError*。

例如，我们可以使用 Monty Python 系列电影的发布日期 在假定出品方保持现有步调的情况下预测到 2019 年时产出的 Monty Python 电影的累计数量。

```
>>> year = [1971, 1975, 1979, 1982, 1983]
>>> films_total = [1, 2, 3, 4, 5]
>>> slope, intercept = linear_regression(year, films_total)
>>> round(slope * 2019 + intercept)
16
```

如果 *proportional* 为真值，则自变量 *x* 和因变量 *y* 将被视为成正比关系。数据会被拟合到一条通过原点的直线上。由于 *intercept* 将始终为 0.0，因此下层的线性函数会简化为:

*y = slope * x + noise*

继续*correlation()* 的例子，我们来看看基于大行星的模型是否能很好地预测矮行星的轨道距离:

```
>>> model = linear_regression(period_squared, dist_cubed, proportional=True)
>>> slope = model.slope

>>> # 矮行星：冥王星、阋神星、鸟神星、妊神星、谷神星
>>> orbital_periods = [90_560, 204_199, 111_845, 103_410, 1_680]  # days
>>> predicted_dist = [math.cbrt(slope * (p * p)) for p in orbital_periods]
>>> list(map(round, predicted_dist))
[5912, 10166, 6806, 6459, 414]

>>> [5_906, 10_152, 6_796, 6_450, 414]  # 以百万公里表示的实际距离
[5906, 10152, 6796, 6450, 414]
```

Added in version 3.10.

在 3.11 版本发生变更: 添加了对 *proportional* 的支持。

9.7.5 异常

只定义了一个异常：

exception statistics.**StatisticsError**

ValueError 的子类，表示统计相关的异常。

9.7.6 NormalDist 对象

NormalDist 工具可用于创建和操纵 随机变量 的正态分布。这个类将数据度量值的平均值和标准差作为单一实体来处理。

正态分布的概念来自于 中央极限定理 并且在统计学中有广泛的应用。

class statistics.**NormalDist**(*mu=0.0*, *sigma=1.0*)

返回一个新的 *NormalDist* 对象，其中 *mu* 代表 算术平均值 而 *sigma* 代表 标准差。

若 *sigma* 为负数，将会引发*StatisticsError*。

mean

一个只读特征属性，表示特定正态分布的 算术平均值。

median

一个只读特征属性，表示特定正态分布的 中位数。

mode

一个只读特征属性，表示特定正态分布的 众数。

stdev

一个只读特征属性，表示特定正态分布的 标准差。

variance

一个只读特征属性，表示特定正态分布的 方差。等于标准差的平方。

classmethod from_samples(*data*)

传入使用*fmean()* 和*stdev()* 基于 *data* 估算出的 *mu* 和 *sigma* 形参创建一个正态分布实例。

data 可以是任何*iterable* 并且应当包含能被转换为*float* 类型的值。如果 *data* 不包含至少两个元素，则会引发*StatisticsError*，因为估算中心值至少需要一个点而估算分散度至少需要两个点。

samples(*n*, ***, *seed=None*)

对于给定的平均值和标准差生成 *n* 个随机样本。返回一个由*float* 值组成的*list*。

当给定 *seed* 时，创建一个新的底层随机数生成器实例。这适用于创建可重现的结果，即使对于多线程上下文也有效。

在 3.13 版本发生变更.

切换为更快速的算法。要重新产生来自之前版本的样本，请使用*random.seed()* 和*random.gauss()*。

pdf(*x*)

使用 概率密度函数 (pdf)，计算一个随机变量 *X* 趋向于给定值 *x* 的相对可能性。在数学意义上，它是当 *dx* 趋向于零时比率 `P(x <= X < x+dx) / dx` 的极限。

相对可能性的计算方法是用一个狭窄区间内某个样本出现的概率除以区间的宽度（因此使用"density" 一词）。由于可能性是相对于其他点的，因此它的值可以大于 `1.0`。

cdf(*x*)

使用 累积分布函数 (cdf)，计算一个随机变量 *X* 小于等于 *x* 的概率。在数学上，它表示为 `P(X <= x)`。

inv_cdf(*p*)

计算反射累积分布函数，也称为 分位数函数 或 百分点 函数。在数学上，它表示为 `x : P(X <= x) = p`。

找出随机变量 *X* 的值 *x* 使得该变量小于等于该值的概率等于给定的概率 *p*。

overlap(*other*)

测量两个正态概率分布之间的一致性。返回介于 0.0 和 1.0 之间的值，给出 两个概率密度函数的重叠区域。

quantiles(*n=4*)

将指定正态分布划分为 *n* 个相等概率的连续分隔区。返回这些分隔区对应的 (n - 1) 个分隔点的列表。

将 *n* 设为 4 以使用四分位（默认值）。将 *n* 设为 10 以使用十分位。将 *n* 设为 100 以使用百分位，即给出 99 个分隔点来将正态分布分隔为 100 个大小相等的组。

zscore(*x*)

计算 标准分 即以高于或低于正态分布的平均值的标准差数值的形式来描述 *x*: `(x - mean) / stdev`.

Added in version 3.9.

NormalDist 的实例支持加上、减去、乘以或除以一个常量。这些运算被用于转换和缩放。例如：

```
>>> temperature_february = NormalDist(5, 2.5)             # 摄氏度
>>> temperature_february * (9/5) + 32                     # 华氏度
NormalDist(mu=41.0, sigma=4.5)
```

不允许一个常量除以*NormalDist* 的实例，因为结果将不是正态分布。

由于正态分布是由独立变量的累加效应产生的，因此允许表示为*NormalDist* 实例的 两组独立正态分布的随机变量相加和相减。例如：

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

Added in version 3.8.

9.7.7 例子和配方

经典概率问题

NormalDist 适合用来解决经典概率问题。

举例来说，如果 SAT 考试的历史数据 显示分数呈平均值为 1060 且标准差为 195 的正态分布，则可以确定考试分数处于 1100 和 1200 之间的学生的百分比舍入到最接近的整数应为：

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

求 SAT 分数的 四分位 和 十分位：

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

蒙特卡罗模拟输入

为了估算一个不易获得解析解的模型分布，*NormalDist* 可以生成用于 蒙特卡洛模拟 的输入样本：

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

近似二项分布

当样本量较大且成功试验的可能性接近 50% 时，正态分布可以被用来模拟 二项式分布 。

例如，一次开源会议有 750 名与会者和两个可分别容纳 500 人的会议厅。会上有一场关于 Python 的演讲和一场关于 Ruby 的演讲。在往届会议中，65% 的与会者更愿意去听关于 Python 的演讲。假定人群的偏好没有发生改变，那么 Python 演讲的会议厅不超出其容量上限的可能性是多少?

```
>>> n = 750             # 样本大小
>>> p = 0.65            # 对 Python 的偏好
>>> q = 1.0 - p         # 对 Ruby 的偏好
>>> k = 500             # 空间容量

>>> # 使用积累正态分布的近似解
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # 使用积累二项分布的精确解
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # 使用随机模拟的近似解
>>> from random import seed, binomialvariate
>>> seed(8675309)
>>> mean(binomialvariate(n, p) <= k for i in range(10_000))
0.8406
```

朴素贝叶斯分类器

在机器学习问题中也经常会出现正态分布。

维基百科上有一个 朴素贝叶斯分类器的良好样例。要处理的问题是根据对多个分布的特征测量值包括身高、体重和足部尺码来预测一个人的性别。

我们得到了由八个人的测量值组成的训练数据集。假定这些测量值是正态分布的，因此我们用*NormalDist* 来总结数据：

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

接下来，我们遇到一个特征测量值已知但性别未知的新人：

```
>>> ht = 6.0        # height
>>> wt = 130        # weight
>>> fs = 8          # foot size
```

从是男是女各 50% 的 先验概率 出发，我们通过将该先验概率乘以给定性别的特征度量值的可能性累积值来计算后验概率：

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

最终预测值应为最大后验概率值。这种算法被称为 maximum a posteriori 或 MAP：

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

CHAPTER 10

函数式编程模块

本章里描述的模块提供了函数和类，以支持函数式编程风格和在可调用对象上的通用操作。

本章包含以下模块的文档：

10.1 itertools --- 为高效循环创建迭代器的函数

本模块实现一系列*iterator*，这些迭代器受到 APL，Haskell 和 SML 的启发。为了适用于 Python，它们都被重新写过。

本模块标准化了一个快速、高效利用内存的核心工具集，这些工具本身或组合都很有用。它们一起形成了“迭代器代数”，这使得在纯 Python 中有可能创建简洁又高效的专用工具。

例如，SML 有一个制表工具：`tabulate(f)`，它可产生一个序列 `f(0), f(1), ...`。在 Python 中可以组合*`map()`* 和*`count()`* 实现：`map(f, count())`。

无穷迭代器：

迭代器	实参	结果	示例
`count()`	[start[, step]]	start, start+step, start+2*step, ...	`count(10) → 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') → A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, ... 重复无限次或 n 次	`repeat(10, 3) → 10 10 10`

根据最短输入序列长度停止的迭代器：

迭代器	实参	结果	示例
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) → 1 3 6 10 15`
batched()	p, n	(p0, p1, ..., p_n-1), ...	`batched('ABCDEFG', n=3) → ABC DEF G`
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') → A B C D E F`
chain.from_iterable	iterable -- 可迭代对象	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) → A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) → A C E F`
dropwhile()	predicate, seq	seq[n], seq[n+1], 从 predicate 未通过时开始	`dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8`
filterfalse()	predicate, seq	predicate(elem) 未通过的 seq 元素	`filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8`
groupby()	iterable[, key]	根据 key(v) 值分组的迭代器	`groupby(['A','B','DEF'], len) → (1, A B) (3, DEF)`
islice()	seq, [start,] stop [, step]	seq[start:stop:step] 中的元素	`islice('ABCDEFG', 2, None) → C D E F G`
pairwise()	iterable -- 可迭代对象	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') → AB BC CD DE EF FG`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000`
takewhile()	predicate, seq	seq[0], seq[1], 直到 predicate 未通过	`takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4`
tee()	it, n	it1, it2, ... itn 将一个迭代器拆分为 n 个迭代器	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-`

排列组合迭代器：

迭代器	实参	结果
product()	p, q, ... [repeat=1]	笛卡尔积，相当于嵌套的 for 循环
permutations()	p[, r]	长度 r 元组，所有可能的排列，无重复元素
combinations()	p, r	长度 r 元组，有序，无重复元素
combinations_with_replacement()	p, r	长度 r 元组，有序，元素可重复

例子	结果
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

10.1.1 Itertool 函数

下列函数都是构造并返回迭代器。有些会提供无限长度的流，所以它们应当只通过能截断流的函数或循环来访问。

itertools.**accumulate**(*iterable*[, *function*, *, *initial=None*])

创建一个返回累积汇总值或来自其他双目运算函数的累积结果的迭代器。

function 默认为加法运算。*function* 应当接受两个参数，即一个累积汇总值和一个来自 *iterable* 的值。

如果提供了 *initial* 值，将从该值开始累积并且输出将比输入可迭代对象多一个元素。

大致相当于：

```
def accumulate(iterable, function=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) → 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) → 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) → 1 2 6 24 120

    iterator = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(iterator)
        except StopIteration:
            return

    yield total
    for element in iterator:
        total = function(total, element)
        yield total
```

要计算运行最小值，则将 *function* 设为*min()*。对于运行最大值，则将 *function* 设为*max()*。或者对于运行乘积，则将 *function* 设为*operator.mul()*。对于构建 分期表，即累计利息并应用还款额：

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, max))                # 运行最大值
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
>>> list(accumulate(data, operator.mul))       # 运行乘积
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]

# 分期偿还利率 5% 总额 1000 的货款，每年还款 10 次，每次 90
>>> update = lambda balance, payment: round(balance * 1.05) - payment
>>> list(accumulate(repeat(90, 10), update, initial=1_000))
[1000, 960, 918, 874, 828, 779, 728, 674, 618, 559, 497]
```

参考一个类似函数*functools.reduce()*，它只返回一个最终累积值。

Added in version 3.2.

在 3.3 版本发生变更: 添加了可选的 *function* 形参。

在 3.8 版本发生变更: 添加了可选的 *initial* 形参。

itertools.**batched**(*iterable*, *n*, *, *strict=False*)

来自 *iterable* 的长度为 *n* 元组形式的批次数据。最后一个批次可能短于 *n*。

如果 *strict* 为真值，将在最终的批次短于 *n* 时引发*ValueError*。

循环处理输入可迭代对象并将数据积累为长度至多为 *n* 的元组。输入将被惰性地消耗，能填满一个批次即可。结果将在批次填满或输入可迭代对象被耗尽时产生：

```
>>> flattened_data = ['roses', 'red', 'violets', 'blue', 'sugar', 'sweet']
>>> unflattened = list(batched(flattened_data, 2))
>>> unflattened
[('roses', 'red'), ('violets', 'blue'), ('sugar', 'sweet')]
```

大致相当于：

```
def batched(iterable, n, *, strict=False):
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    iterator = iter(iterable)
    while batch := tuple(islice(iterator, n)):
        if strict and len(batch) != n:
            raise ValueError('batched(): incomplete batch')
        yield batch
```

Added in version 3.12.

在 3.13 版本发生变更: 增加了 *strict* 选项。

itertools.**chain**(**iterables*)

创建一个迭代器，它会从第一个可迭代对象返回元素直到将其耗尽，接着转到下一个可迭代对象，直到将所有可迭代对象都耗尽为止。这是将多个数据源合并为单个可迭代器。大致等价于:

```
def chain(*iterables):
    # chain('ABC', 'DEF') → A B C D E F
    for iterable in iterables:
        yield from iterable
```

classmethod chain.**from_iterable**(*iterable*)

构建类似*chain()* 迭代器的另一个选择。从一个单独的可迭代参数中得到链式输入，该参数是延迟计算的。大致相当于:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) → A B C D E F
    for iterable in iterables:
        yield from iterable
```

itertools.**combinations**(*iterable*, *r*)

返回由输入 *iterable* 中元素组成长度为 *r* 的子序列。

输出结果是*product()* 的子序列其中只保留属于 *iterable* 的子序列的条目。输出的长度由*math.comb()* 给出，该函数在 `0 ≤ r ≤ n` 时为计算 `n! / r! / (n - r)!` 而在 `r > n` 时为 0。

组合元组是根据输入的 *iterable* 的顺序以词典排序方式发出的。如果输入的 *iterable* 是已排序的，则输出的元组将按排序后的顺序产生。

元素是的唯一性是基于它们的位置，而不是它们的值。如果输入的元素都是唯一的，则将每个组合中将不会有重复的值。

大致相当于:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) → AB AC AD BC BD CD
    # combinations(range(4), 3) → 012 013 023 123

    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
```

(续下页)

(接上页)

```
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

itertools.**combinations_with_replacement**(*iterable*, *r*)

返回由输入 *iterable* 中元素组成的长度为 *r* 的子序列，允许每个元素可重复出现。

输出是*product()* 的子序列，其中仅保留也属于 *iterable* 的子序列的条目（可能有重复的元素）。当 `n > 0` 时返回的子序列数量为 `(n + r - 1)! / r! / (n - 1)!`。

组合元组是根据输入的 *iterable* 的顺序以词典排序方式发出的。如果输入的 *iterable* 是已排序的，则输出的元组将按已排序的顺序产生。

元素的唯一性是基于它们的位置，而不是它们的值。如果输入的元素都是唯一的，则生成的组合也将是唯一的。

大致相当于：

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) → AA AB AC BB BC CC

    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

Added in version 3.1.

itertools.**compress**(*data*, *selectors*)

创建一个迭代器，它返回来自 *data* 在 *selectors* 中对应元素为真值的元素。当 *data* 或 *selectors* 可迭代对象被耗尽时将停止。大致相当于:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) → A C E F
    return (datum for datum, selector in zip(data, selectors) if selector)
```

Added in version 3.1.

itertools.**count**(*start=0*, *step=1*)

创建一个迭代器，它返回从 *start* 开始的均匀间隔的值。可与*map()* 配合使用以生成连续的数据点或与*zip()* 配合使用以添加序列数字。大致相当于:

```
def count(start=0, step=1):
    # count(10) → 10 11 12 13 14 ...
    # count(2.5, 0.5) → 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

当对浮点数计数时，替换为乘法代码有时会有更高的精度，例如: `(start + step * i for i in count())`。

在 3.1 版本发生变更: 增加参数 *step* ，允许非整型。

itertools.**cycle**(*iterable*)

创建一个迭代器，它返回来自 *iterable* 中的元素并保存每个元素的拷贝。当 iterable 耗尽时，返回来自已保存拷贝中的元素。将无限重复进行。大致相当于:

```
def cycle(iterable):
    # cycle('ABCD') → A B C D A B C D A B C D ...

    saved = []
    for element in iterable:
        yield element
        saved.append(element)

    while saved:
        for element in saved:
            yield element
```

这个迭代工具可能需要很大的辅助存储（取决于 iterable 的长度）。

itertools.**dropwhile**(*predicate*, *iterable*)

创建一个迭代器，它将丢弃来自 *iterable* 中 *predicate* 为真值的元素然后返回每个元素。大致相当于:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8

    iterator = iter(iterable)
    for x in iterator:
        if not predicate(x):
            yield x
            break

    for x in iterator:
        yield x
```

请注意它将不产生 任何输出直到 predicate 首次变为假值，所以此迭代工具可能具有很长的启动时间。

itertools.**filterfalse**(*predicate*, *iterable*)

创建一个迭代器，它过滤来自 *iterable* 的元素从而只返回其中 *predicate* 返回假值的元素。如果 *predicate* 为 `None`，则返回本身为假值的条目。大致相当于:

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8

    if predicate is None:
        predicate = bool

    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

创建一个迭代器，返回 *iterable* 中连续的键和组。*key* 是一个计算元素键值函数。如果未指定或为 `None`，*key* 缺省为恒等函数（identity function），返回元素不变。一般来说，*iterable* 需用同一个键值函数预先排序。

groupby() 操作类似于 Unix 中的 `uniq`。当每次 *key* 函数产生的键值改变时，迭代器会分组或生成一个新组（这就是为什么通常需要使用同一个键值函数先对数据进行排序）。这种行为与 SQL 的 GROUP BY 操作不同，SQL 的操作会忽略输入的顺序将相同键值的元素分在同组中。

返回的组本身也是一个迭代器，它与*groupby()* 共享底层的可迭代对象。因为源是共享的，当*groupby()* 对象向后迭代时，前一个组将消失。因此如果稍后还需要返回结果，可保存为列表：

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # 将 group 迭代器以列表形式保存
    uniquekeys.append(k)
```

groupby() 大致相当于：

```
def groupby(iterable, key=None):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D

    keyfunc = (lambda x: x) if key is None else key
    iterator = iter(iterable)
    exhausted = False

    def _grouper(target_key):
        nonlocal curr_value, curr_key, exhausted
        yield curr_value
        for curr_value in iterator:
            curr_key = keyfunc(curr_value)
            if curr_key != target_key:
                return
            yield curr_value
        exhausted = True

    try:
        curr_value = next(iterator)
    except StopIteration:
        return
    curr_key = keyfunc(curr_value)

    while not exhausted:
        target_key = curr_key
        curr_group = _grouper(target_key)
        yield curr_key, curr_group
        if curr_key == target_key:
            for _ in curr_group:
                pass
```

itertools.**islice**(*iterable*, *stop*)

itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

创建一个迭代器，它返回 iterable 的选定元素。效果与序列切片类似但不支持负的 *start*, *stop* 或 *step* 值。

如果 *start* 为零或为 None，迭代将从零开始。在其他情况下，iterable 中的元素将被跳过直至到达 *start*。

如果 *stop* 为 None，迭代将持续进行直到输入被耗尽，如果能耗尽的话。在其他情况下，它将在指定位置停止。

如果 *step* 为 None，则步长默认为一。元素将被逐一返回除非 *step* 被设为大于一的数，此情况将导致部分条目被跳过。

大致相当于：

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
```

(续下页)

(接上页)

```
    # islice('ABCDEFG', 2, 4) → C D
    # islice('ABCDEFG', 2, None) → C D E F G
    # islice('ABCDEFG', 0, None, 2) → A C E G

    s = slice(*args)
    start = 0 if s.start is None else s.start
    stop = s.stop
    step = 1 if s.step is None else s.step
    if start < 0 or (stop is not None and stop < 0) or step <= 0:
        raise ValueError

    indices = count() if stop is None else range(max(start, stop))
    next_i = start
    for i, element in zip(indices, iterable):
        if i == next_i:
            yield element
            next_i += step
```

如果输入是一个迭代器，则完全消耗 *islice* 将使输入的迭代器向前执行 `max(start, stop)` 步而不管 *step* 值是多少。

`itertools.`**`pairwise`**(*iterable*)

返回从输入 *iterable* 中获取的连续重叠对。

输出迭代器中 2 元组的数量将比输入的数量少一个。如果输入可迭代对象中少于两个值则它将为空。

大致相当于：

```
def pairwise(iterable):
    # pairwise('ABCDEFG') → AB BC CD DE EF FG

    iterator = iter(iterable)
    a = next(iterator, None)

    for b in iterator:
        yield a, b
        a = b
```

Added in version 3.10.

`itertools.`**`permutations`**(*iterable*, *r=None*)

根据 *iterable* 返回连续的 *r* 长度 元素的排列。

如果 *r* 未指定或为 `None`，*r* 默认设置为 *iterable* 的长度，这种情况下，生成所有全长排列。

输出结果是*`product()`* 的子序列并已过滤掉其中的重复元素。输出的长度由*`math.perm()`* 给出，它在 `0 ≤ r ≤ n` 时为计算 `n! / (n - r)!` 而在 `r > n` 时则为零。

排列元组是根据输入的 *iterable* 的顺序以词典排序的形式发出的。如果输入的 *iterable* 是已排序的，则输出的元组将按已排序的顺序产生。

元素的唯一性是基于它们的位置，而不是它们的值。如果输入的元素都是唯一的，则在排列中就不会有重复的元素。

大致相当于：

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) → AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) → 012 021 102 120 201 210

    pool = tuple(iterable)
    n = len(pool)
```

(续下页)

(接上页)

```
    r = n if r is None else r
    if r > n:
        return

    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])

    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

itertools.**product**(**iterables*, *repeat=1*)

输入可迭代对象的 笛卡尔乘积。

大致相当于生成器表达式中的嵌套循环。例如，`product(A, B)` 和 `((x,y) for x in A for y in B)` 返回结果一样。

嵌套循环像里程表那样循环变动，每次迭代时将最右侧的元素向后迭代。这种模式形成了一种字典序，因此如果输入的可迭代对象是已排序的，笛卡尔积元组依次序发出。

要计算可迭代对象自身的笛卡尔积，将可选参数 *repeat* 设定为要重复的次数。例如，`product(A, repeat=4)` 和 `product(A, A, A, A)` 是一样的。

该函数大致相当于下面的代码，只不过实际实现方案不会在内存中创建中间结果。

```
def product(*iterables, repeat=1):
    # product('ABCD', 'xy') → Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) → 000 001 010 011 100 101 110 111

    if repeat < 0:
        raise ValueError('repeat argument cannot be negative')
    pools = [tuple(pool) for pool in iterables] * repeat

    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]

    for prod in result:
        yield tuple(prod)
```

在*product()* 运行之前，它会完全耗尽输入的可迭代对象，在内存中保留值的临时池以生成结果积。相应地，它只适用于有限的输入。

itertools.**repeat**(*object*[, *times*])

创建一个持续地返回 *object* 的迭代器。将会无限期地运行除非指定了 *times* 参数。

大致相当于：

```
def repeat(object, times=None):
    # repeat(10, 3) → 10 10 10
    if times is None:
```

(续下页)

(接上页)

```
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

repeat 的一个常见用途是向 *map* 或 *zip* 提供一个常量值的流:

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.**starmap**(*function*, *iterable*)

创建一个迭代器，它使用从 *iterable* 获取的参数来计算 *function*。当参数形参已被“预先 zip”为元组时可代替*map()* 来使用。

map() 和*starmap()* 之间的区别类似于 `function(a,b)` 和 `function(*c)` 之间的差异。大致相当于:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)

创建一个迭代器，它返回来自 *iterable* 的 *predicate* 为真值的元素。大致相当于:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,3,8]) --> 1 4
    for x in iterable:
        if not predicate(x):
            break
        yield x
```

请注意，第一个未能满足 predicate 条件的元素将从输入迭代器中消耗掉并且没有办法访问它。当应用程序想在 *takewhile* 运行到耗尽后进一步消耗输入迭代器时这可能会导致问题。要绕过这个问题，可以考虑改用 more-itertools before_and_after()。

itertools.**tee**(*iterable*, *n=2*)

从一个可迭代对象中返回 *n* 个独立的迭代器。

大致相当于:

```
def tee(iterable, n=2):
    if n < 0:
        raise ValueError
    if n == 0:
        return ()
    iterator = _tee(iterable)
    result = [iterator]
    for _ in range(n - 1):
        result.append(_tee(iterator))
    return tuple(result)

class _tee:

    def __init__(self, iterable):
        it = iter(iterable)
        if isinstance(it, _tee):
            self.iterator = it.iterator
            self.link = it.link
        else:
```

(续下页)

(接上页)

```
            self.iterator = it
            self.link = [None, None]

    def __iter__(self):
        return self

    def __next__(self):
        link = self.link
        if link[1] is None:
            link[0] = next(self.iterator)
            link[1] = [None, None]
        value, self.link = link
        return value
```

当输入的 *iterable* 已经是一个 tee 迭代器对象时，将会构造返回元组的所有成员就像它们已被上游的*tee()* 调用所产生那样。这个“展平步骤”允许嵌套的*tee()* 调用共享相同的下层数据链并仅使用一个更新步骤而非一个调用链。

展平的特征属性使得 tee 迭代器可被高效地查看：

```
def lookahead(tee_iterator):
    "返回下一个值而不向前获取输入"
    [forked_iterator] = tee(tee_iterator, 1)
    return next(forked_iterator)
```

```
>>> iterator = iter('abcdef')
>>> [iterator] = tee(iterator, 1)    # 使输入可被查看
>>> next(iterator)                   # 向前执行迭代器
'a'
>>> lookahead(iterator)              # 检查下一个值
'b'
>>> next(iterator)                   # 继续向前执行
'b'
```

tee 迭代器不是线程安全的。当同时使用由同一个*tee()* 调用所返回的迭代器时可能引发*RuntimeError*，即使原本的 *iterable* 是线程安全的。is threadsafe.

该迭代工具可能需要相当大的辅助存储空间（这取决于要保存多少临时数据）。通常，如果一个迭代器在另一个迭代器开始之前就要使用大部份或全部数据，使用*list()* 会比*tee()* 更快。

itertools.**zip_longest**(**iterables*, *fillvalue=None*)

创建一个迭代器，它聚合了来自 *iterables* 中每一项的对应元素。

如果 iterables 中每一项的长度不同，则缺失的值将以 *fillvalue* 填充。如果未指定，则 *fillvalue* 默认为 None。

迭代将持续进行直至其中最长的可迭代对象被耗尽。

大致相当于：

```
def zip_longest(*iterables, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

    iterators = list(map(iter, iterables))
    num_active = len(iterators)
    if not num_active:
        return

    while True:
        values = []
        for i, iterator in enumerate(iterators):
            try:
```

(续下页)

(接上页)

```
                value = next(iterator)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

如果 iterables 中的每一项可能有无限长度，则*zip_longest()* 函数应当用限制调用次数的代码进行包装（例如*islice()* 或*takewhile()* 等）。

10.1.2 itertools 配方

本节将展示如何使用现有的 itertools 作为基础构件来创建扩展的工具集。

这些 itertools 专题的主要目的是教学。各个专题显示了对单个工具的各种思维方式—例如，`chain.from_iterable` 被关联到展平的概念。这些专题还给出了有关这些工具的组合方式的想法—例如，`starmap()` 和 `repeat()` 应当如何一起工作。这些专题还显示了 itertools 与*operator* 和*collections* 模块以及内置迭代工具如 `map()`, `filter()`, `reversed()` 和 `enumerate()` 相互配合的使用模式。

这些例程的次要目的是作为一个孵化器使用。`accumulate()`, `compress()` 和 `pairwise()` 等迭代工具最初就是作为例程引入的。目前，`sliding_window()`, `iter_index()` 和 `sieve()` 例程正在被测试以确定它们是否堪当大任。

基本上所有这些配方和许许多多其他配方都可以通过 Python Package Index 上的 more-itertools 项目来安装:

```
python -m pip install more-itertools
```

许多例程提供了与底层工具集相当的高性能。更好的内存效率是通过每次只处理一个元素而不是将整个可迭代对象放入内存来保证的。代码量的精简是通过以 函数式风格 来链接工具来实现的。运行的早速度是通过选择使用“矢量化”构件来取代会导致较大解释器开销的 for 循环和生成器 来达成的。

```
from collections import Counter, deque
from contextlib import suppress
from functools import reduce
from math import comb, prod, sumprod, isqrt
from operator import itemgetter, getitem, mul, neg

def take(n, iterable):
    "Return first n items of the iterable as a list."
    return list(islice(iterable, n))

def prepend(value, iterable):
    "Prepend a single value in front of an iterable."
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterable)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def repeatfunc(function, times=None, *args):
    "Repeat calls to a function with specified arguments."
    if times is None:
        return starmap(function, repeat(args))
    return starmap(function, repeat(args, times))

def flatten(list_of_lists):
```

(续下页)

(接上页)

```
    "Flatten one level of nesting."
    return chain.from_iterable(list_of_lists)

def ncycles(iterable, n):
    "Returns the sequence elements n times."
    return chain.from_iterable(repeat(tuple(iterable), n))

def loops(n):
    "Loop n times. Like range(n) but without creating integers."
    # for _ in loops(100): ...
    return repeat(None, n)

def tail(n, iterable):
    "Return an iterator over the last n items."
    # tail(3, 'ABCDEFG') → E F G
    return iter(deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # 使用以速度 C 消耗迭代器的函数。
    if n is None:
        deque(iterator, maxlen=0)
    else:
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value."
    return next(islice(iterable, n, None), default)

def quantify(iterable, predicate=bool):
    "Given a predicate that returns True or False, count the True results."
    return sum(map(predicate, iterable))

def first_true(iterable, default=False, predicate=None):
    "Returns the first true value or the *default* if there is no true value."
    # first_true([a,b,c], x) → a or b or c or x
    # first_true([a,b], x, f) → a if f(a) else b if f(b) else x
    return next(filter(predicate, iterable), default)

def all_equal(iterable, key=None):
    "Returns True if all the elements are equal to each other."
    # all_equal('4□□□□', key=int) → True
    return len(take(2, groupby(iterable, key))) <= 1

def unique_justseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') → A B C D A B
    # unique_justseen('ABBcCAD', str.casefold) → A B c A D
    if key is None:
        return map(itemgetter(0), groupby(iterable))
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def unique_everseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') → A B C D
    # unique_everseen('ABBcCAD', str.casefold) → A B c D
    seen = set()
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen.add(element)
            yield element
```

(续下页)

(接上页)

```
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen.add(k)
                yield element

def unique(iterable, key=None, reverse=False):
   "Yield unique elements in sorted order. Supports unhashable inputs."
   # unique([[1, 2], [3, 4], [1, 2]]) → [1, 2] [3, 4]
   sequenced = sorted(iterable, key=key, reverse=reverse)
   return unique_justseen(sequenced, key=key)

def sliding_window(iterable, n):
    "Collect data into overlapping fixed-length chunks or blocks."
    # sliding_window('ABCDEFG', 4) → ABCD BCDE CDEF DEFG
    iterator = iter(iterable)
    window = deque(islice(iterator, n - 1), maxlen=n)
    for x in iterator:
        window.append(x)
        yield tuple(window)

def grouper(iterable, n, *, incomplete='fill', fillvalue=None):
    "Collect data into non-overlapping fixed-length chunks or blocks."
    # grouper('ABCDEFG', 3, fillvalue='x') → ABC DEF Gxx
    # grouper('ABCDEFG', 3, incomplete='strict') → ABC DEF ValueError
    # grouper('ABCDEFG', 3, incomplete='ignore') → ABC DEF
    iterators = [iter(iterable)] * n
    match incomplete:
        case 'fill':
            return zip_longest(*iterators, fillvalue=fillvalue)
        case 'strict':
            return zip(*iterators, strict=True)
        case 'ignore':
            return zip(*iterators)
        case _:
            raise ValueError('Expected fill, strict, or ignore')

def roundrobin(*iterables):
    "Visit input iterables in a cycle until each is exhausted."
    # roundrobin('ABC', 'D', 'EF') → A D E B F C
    # 此算法由 George Sakkis 首创
    iterators = map(iter, iterables)
    for num_active in range(len(iterables), 0, -1):
        iterators = cycle(islice(iterators, num_active))
        yield from map(next, iterators)

def subslices(seq):
    "Return all contiguous non-empty subslices of a sequence."
    # subslices('ABCD') → A AB ABC ABCD B BC BCD C CD D
    slices = starmap(slice, combinations(range(len(seq) + 1), 2))
    return map(getitem, repeat(seq), slices)

def iter_index(iterable, value, start=0, stop=None):
    "Return indices where a value occurs in a sequence or iterable."
    # iter_index('AABCADEAF', 'A') → 0 1 4 7
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
        iterator = islice(iterable, start, stop)
        for i, element in enumerate(iterator, start):
            if element is value or element == value:
```

(续下页)

(接上页)

```
                yield i
    else:
        stop = len(iterable) if stop is None else stop
        i = start
        with suppress(ValueError):
            while True:
                yield (i := seq_index(value, i, stop))
                i += 1

def iter_except(function, exception, first=None):
    "Convert a call-until-exception interface to an iterator interface."
    # iter_except(d.popitem, KeyError) → 非阻塞的字典迭代器
    with suppress(exception):
        if first is not None:
            yield first()
        while True:
            yield function()
```

下面的例程具有更数学化的风格:

```
def multinomial(*counts):
    "Number of distinct arrangements of a multiset."
    # Counter('abracadabra').values() → 5 2 2 1 1
    # multinomial(5, 2, 2, 1, 1) → 83160
    return prod(map(comb, accumulate(counts), counts))

def powerset(iterable):
    "Subsequences of the iterable from shortest to longest."
    # powerset([1,2,3]) → () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def sum_of_squares(iterable):
    "Add up the squares of the input values."
    # sum_of_squares([10, 20, 30]) → 1400
    return sumprod(*tee(iterable))

def reshape(matrix, columns):
    "Reshape a 2-D matrix to have a given number of columns."
    # reshape([(0, 1), (2, 3), (4, 5)], 3) →  (0, 1, 2), (3, 4, 5)
    return batched(chain.from_iterable(matrix), columns, strict=True)

def transpose(matrix):
    "Swap the rows and columns of a 2-D matrix."
    # transpose([(1, 2, 3), (11, 22, 33)]) → (1, 11) (2, 22) (3, 33)
    return zip(*matrix, strict=True)

def matmul(m1, m2):
    "Multiply two matrices."
    # matmul([(7, 5), (3, 5)], [(2, 5), (7, 9)]) → (49, 80), (41, 60)
    n = len(m2[0])
    return batched(starmap(sumprod, product(m1, transpose(m2))), n)

def convolve(signal, kernel):
    """Discrete linear convolution of two iterables.
    Equivalent to polynomial multiplication.

    Convolutions are mathematically commutative; however, the inputs are
    evaluated differently.  The signal is consumed lazily and can be
    infinite. The kernel is fully consumed before the calculations begin.
```

(续下页)

(接上页)

```
    Article:  https://betterexplained.com/articles/intuitive-convolution/
    Video:    https://www.youtube.com/watch?v=KuXjwB4LzSA
    """
    # convolve([1, -1, -20], [1, -3]) → 1 -4 -17 60
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) → Moving average (blur)
    # convolve(data, [1/2, 0, -1/2]) → 1st derivative estimate
    # convolve(data, [1, -2, 1]) → 2nd derivative estimate
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    padded_signal = chain(repeat(0, n-1), signal, repeat(0, n-1))
    windowed_signal = sliding_window(padded_signal, n)
    return map(sumprod, repeat(kernel), windowed_signal)

def polynomial_from_roots(roots):
    """Compute a polynomial's coefficients from its roots.

       (x - 5) (x + 4) (x - 3)  expands to:   x³ -4x² -17x + 60
    """
    # polynomial_from_roots([5, -4, 3]) → [1, -4, -17, 60]
    factors = zip(repeat(1), map(neg, roots))
    return list(reduce(convolve, factors, [1]))

def polynomial_eval(coefficients, x):
    """Evaluate a polynomial at a specific value.

    Computes with better numeric stability than Horner's method.
    """
    # Evaluate x³ -4x² -17x + 60 at x = 5
    # polynomial_eval([1, -4, -17, 60], x=5) → 0
    n = len(coefficients)
    if not n:
        return type(x)(0)
    powers = map(pow, repeat(x), reversed(range(n)))
    return sumprod(coefficients, powers)

def polynomial_derivative(coefficients):
    """Compute the first derivative of a polynomial.

       f(x)  =  x³ -4x² -17x + 60
       f'(x) = 3x² -8x  -17
    """
    # polynomial_derivative([1, -4, -17, 60]) → [3, -8, -17]
    n = len(coefficients)
    powers = reversed(range(1, n))
    return list(map(mul, coefficients, powers))

def sieve(n):
    "Primes less than n."
    # sieve(30) → 2 3 5 7 11 13 17 19 23 29
    if n > 2:
        yield 2
    data = bytearray((0, 1)) * (n // 2)
    for p in iter_index(data, 1, start=3, stop=isqrt(n) + 1):
        data[p*p : n : p+p] = bytes(len(range(p*p, n, p+p)))
    yield from iter_index(data, 1, start=3)

def factor(n):
    "Prime factors of n."
    # factor(99) → 3 3 11
    # factor(1_000_000_000_000_007) → 47 59 360620266859
    # factor(1_000_000_000_000_403) → 1000000000000403
```

(续下页)

(接上页)

```
    for prime in sieve(isqrt(n) + 1):
        while not n % prime:
            yield prime
            n //= prime
            if n == 1:
                return
    if n > 1:
        yield n

def is_prime(n):
    "Return True if n is prime."
    # is_prime(1_000_000_000_000_403) → True
    return n > 1 and next(factor(n)) == n

def totient(n):
    "Count of natural numbers up to n that are coprime to n."
    # https://mathworld.wolfram.com/TotientFunction.html
    # totient(12) → 4 because len([1, 5, 7, 11]) == 4
    for prime in set(factor(n)):
        n -= n // prime
    return n
```

10.2 functools ——高阶函数，以及可调用对象上的操作

源代码: Lib/functools.py

---

functools 模块应用于高阶函数，即参数或（和）返回值为其他函数的函数。通常来说，此模块的功能适用于所有可调用对象。

functools 模块定义了以下函数:

`@functools.cache(`*user_function*`)`

简单轻量级未绑定函数缓存。有时称为 "memoize"。

返回值与 `lru_cache(maxsize=None)` 相同，创建一个查找函数参数的字典的简单包装器。因为它不需要清除旧值，所以比带有大小限制的*lru_cache()* 更小更快。

例如:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # 不预先缓存结果，执行 11 次递归调用
3628800
>>> factorial(5)       # 只查找缓存结果值
120
>>> factorial(12)      # 执行两次新的递归调用，另外 10 次已缓存
479001600
```

该缓存是线程安全的因此被包装的函数可在多线程中使用。这意味着下层的数据结构将在并发更新期间保持一致性。

如果另一个线程在初始调用完成并被缓存之前执行了额外的调用则被包装的函数可能会被多次调用。

Added in version 3.9.

@functools.**cached_property**(*func*)

将一个类方法转换为特征属性，一次性计算该特征属性的值，然后将其缓存为实例生命周期内的普通属性。类似于*property()* 但增加了缓存功能。对于在其他情况下实际不可变的高计算资源消耗的实例特征属性来说该函数非常有用。

示例:

```
class DataSet:

    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

cached_property() 的设定与*property()* 有所不同。常规的 property 会阻止属性写入，除非定义了 setter。与之相反，*cached_property* 则允许写入。

cached_property 装饰器仅在执行查找且不存在同名属性时才会运行。当运行时，*cached_property* 会写入同名的属性。后续的属性读取和写入操作会优先于 *cached_property* 方法，其行为就像普通的属性一样。

缓存的值可通过删除该属性来清空。这允许 *cached_property* 方法再次运行。

cached_property 不能防止在多线程使用中可能出现的竞争条件。getter 函数可以在同一实例上多次运行，最后一次运行将设置缓存值。如果缓存的特征属性是幂等的或者对于在同一实例上多次运行是无害的，那就没有问题。如果需要进行同步，请在被装饰的 getter 函数内部或在缓存的特征属性访问外部实现必要的锁定操作。

注意，这个装饰器会影响 **PEP 412** 键共享字典的操作。这意味着相应的字典实例可能占用比通常时更多的空间。

而且，这个装饰器要求每个实例上的 __dict__ 是可变的映射。这意味着它将不适用于某些类型，例如元类（因为类型实例上的 __dict__ 属性是类命名空间的只读代理），以及那些指定了 __slots__ 但未包括 __dict__ 作为所定义的空位之一的类（因为这样的类根本没有提供 __dict__ 属性）。

如果可变的映射不可用或者如果想要节省空间的键共享，可以通过在*lru_cache()* 上堆叠*property()* 来实现类似*cached_property()* 的效果。请参阅 faq-cache-method-calls 了解这与*cached_property()* 之间区别的详情。

Added in version 3.8.

在 3.12 版本发生变更: 在 Python 3.12 之前，cached_property 包括了一个未写入文档的锁用来确保在多线程使用中 getter 函数对于每个实例保证只运行一次。但是，这个锁是针对特征属性的，不是针对实例的，这可能导致不可接受的高强度锁争用。在 Python 3.12+ 中这个锁已被移除。

functools.**cmp_to_key**(*func*)

将 (旧式的) 比较函数转换为新式的*key function*。在类似于*sorted()*，*min()*，*max()*，*heapq.nlargest()*，*heapq.nsmallest()*，*itertools.groupby()* 等函数的 *key* 参数中使用。此函数主要用作将 Python 2 程序转换至新版的转换工具，以保持对比较函数的兼容。

比较函数是任何接受两个参数，比较它们，并在结果为小于时返回负数，等于时返回零，大于时返回正数的可调用对象。键函数是接受一个参数并返回另一值的可调用对象，返回值在排序时被用作键。

示例:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # 感知语言区域的排序设置
```

有关排序示例和简要排序教程，请参阅 sortinghowto。

Added in version 3.2.

@functools.**lru_cache**(*user_function*)

@functools.**lru_cache**(*maxsize=128*, *typed=False*)

一个为函数提供缓存功能的装饰器，缓存 *maxsize* 组传入参数，在下次以相同参数调用时直接返回上一次的结果。用以节约高开销或 I/O 函数的调用时间。

该缓存是线程安全的因此被包装的函数可在多线程中使用。这意味着下层的数据结构将在并发更新期间保持一致性。

如果另一个线程在初始调用完成并被缓存之前执行了额外的调用则被包装的函数可能会被多次调用。

由于使用字典来缓存结果，因此传给该函数的位置和关键字参数必须为*hashable*。

不同的参数模式可能会被视为具有单独缓存项的不同调用。例如，`f(a=1, b=2)` 和 `f(b=2, a=1)` 因其关键字参数顺序不同而可能会具有两个单独的缓存项。

如果指定了 *user_function*，它必须是一个可调用对象。这允许 *lru_cache* 装饰器被直接应用于一个用户自定义函数，让 *maxsize* 保持其默认值 128:

```
@lru_cache
def count_vowels(sentence):
    return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')
```

如果 *maxsize* 设为 `None`，LRU 特性将被禁用且缓存可无限增长。

如果 *typed* 被设置为 true，不同类型的函数参数将被分别缓存。如果 *typed* 为 false，实现通常会将它们视为等价的调用，只缓存一个结果。(有些类型，如 *str* 和 *int*，即使 *typed* 为 false，也可能被分开缓存)。

注意，类型的特殊性只适用于函数的直接参数而不是它们的内容。标量参数 `Decimal(42)` 和 `Fraction(42)` 被视为具有不同结果的不同调用。相比之下，元组参数 `('answer', Decimal(42))` 和 `('answer', Fraction(42))` 被视为等同的。

被包装的函数配有一个 `cache_parameters()` 函数，它返回一个新的*dict* 用来显示 *maxsize* 和 *typed* 的值。这只是出于显示信息的目的。改变这些值没有有任何效果。

为了帮助衡量缓存的有效性以及调整 *maxsize* 形参，被包装的函数会带有一个 `cache_info()` 函数，它返回一个*named tuple* 以显示 *hits*, *misses*, *maxsize* 和 *currsize*。

该装饰器也提供了一个用于清理/使缓存失效的函数 `cache_clear()`。

原始的未经装饰的函数可以通过 `__wrapped__` 属性访问。它可以用于检查、绕过缓存，或使用不同的缓存再次装饰原始函数。

缓存会保持对参数的引用并返回值，直到它们结束生命期退出缓存或者直到缓存被清空。

如果一个方法被缓存，则 `self` 实例参数会被包括在缓存中。请参阅 faq-cache-method-calls

LRU (least recently used) 缓存 在最近的调用是即将到来的调用的最佳预测值时性能最好 (例如，新闻服务器上的最热门文章倾向于每天发生变化)。缓存的大小限制可确保缓存不会在长期运行的进程如 web 服务器上无限制地增长。

一般来说，LRU 缓存只应在你需要重复使用先前计算的值时使用。因此，缓存有附带影响的函数、每次调用都需要创建不同的可变对象的函数（如生成器和异步函数）或不纯的函数如 time() 或 random() 等是没有意义的。

静态 Web 内容的 LRU 缓存示例:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = f'https://peps.python.org/pep-{num:04d}'
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'
```

(续下页)

(接上页)

```
>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

以下是使用缓存通过 动态规划 计算 斐波那契数列 的例子。

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Added in version 3.2.

在 3.3 版本发生变更: 添加 *typed* 选项。

在 3.8 版本发生变更: 添加了 *user_function* 选项。

在 3.9 版本发生变更: 增加了 `cache_parameters()` 函数

`@functools.`**`total_ordering`**

给定一个声明一个或多个全比较排序方法的类，这个类装饰器实现剩余的方法。这减轻了指定所有可能的全比较操作的工作。

此类必须包含以下方法之一：`__lt__()` 、`__le__()`、`__gt__()` 或 `__ge__()`。另外，此类必须支持 `__eq__()` 方法。

例如：

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

备注

虽然此装饰器使得创建具有良好行为的完全有序类型变得非常容易，但它 确实是以执行速度更缓慢和派生比较方法的堆栈回溯更复杂为代价的。如果性能基准测试表明这是特定应用的瓶颈所在，则改为实现全部六个富比较方法应该会轻松提升速度。

> **备注**
>
> 这个装饰器不会尝试重写类 或其上级类中已经被声明的方法。这意味着如果某个上级类定义了比较运算符，则 *total_ordering* 将不会再次实现它，即使原方法是抽象方法。

Added in version 3.2.

在 3.4 版本发生变更: 现在已支持从未识别的类型的下层比较函数返回 `NotImplemented` 异常。

`functools.`**`partial`**(*func*, */*, **args*, ***keywords*)

返回一个新的*部分对象*，当被调用时其行为类似于 *func* 附带位置参数 *args* 和关键字参数 *keywords* 被调用。如果为调用提供了更多的参数，它们会被附加到 *args*。如果提供了额外的关键字参数，它们会扩展并重写 *keywords*。大致等价于:

```
def partial(func, /, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = {**keywords, **fkeywords}
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

partial() 会被“冻结了”一部分函数参数和/或关键字的部分函数应用所使用，从而得到一个具有简化签名的新对象。例如，*partial()* 可用来创建一个行为类似于*int()* 函数的可调用对象，其中 *base* 参数默认为二:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

class `functools.`**`partialmethod`**(*func*, */*, **args*, ***keywords*)

返回一个新的*partialmethod* 描述器，其行为类似*partial* 但它被设计用作方法定义而非直接用作可调用对象。

func 必须是一个*descriptor* 或可调用对象（同属两者的对象例如普通函数会被当作描述器来处理）。

当 *func* 是一个描述器（例如普通 Python 函数，*classmethod()*，*staticmethod()*，`abstractmethod()` 或其他*partialmethod* 的实例）时，对 `__get__` 的调用会被委托给底层的描述器，并会返回一个适当的*部分对象* 作为结果。

当 *func* 是一个非描述器类可调用对象时，则会动态创建一个适当的绑定方法。当用作方法时其行为类似普通 Python 函数：将会插入 *self* 参数作为第一个位置参数，其位置甚至会处于提供给*partialmethod* 构造器的 *args* 和 *keywords* 之前。

示例:

```
>>> class Cell:
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
```

(续下页)

(接上页)

```
False
>>> c.set_alive()
>>> c.alive
True
```

Added in version 3.4.

functools.**reduce**(*function*, *iterable*, [*initial*,]/)

将两个参数的 *function* 从左至右累积地应用到 *iterable* 的条目，以便将该可迭代对象缩减为单个值。例如，`reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` 就是计算 `((((1+2)+3)+4)+5)`。左边的参数 *x* 是累积的值而右边的参数 *y* 则是来自 *iterable* 的更新值。如果存在可选项 *initial*，它会被放在参与计算的可迭代对象的条目之前，并在可迭代对象为空时作为默认值。如果未给出 *initial* 并且 *iterable* 仅包含一个条目，则将返回第一项。

大致相当于：

```
initial_missing = object()

def reduce(function, iterable, initial=initial_missing, /):
    it = iter(iterable)
    if initial is initial_missing:
        value = next(it)
    else:
        value = initial
    for element in it:
        value = function(value, element)
    return value
```

请参阅*itertools.accumulate()* 了解有关可产生所有中间值的迭代器。

@functools.**singledispatch**

将一个函数转换为单分派 *generic function*。

要定义一个泛型函数，用装饰器 `@singledispatch` 来装饰它。当使用 `@singledispatch` 定义一个函数时，请注意调度发生在第一个参数的类型上：

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

要将重载的实现添加到函数中，请使用泛型函数的 `register()` 属性，它可以被用作装饰器。对于带有类型标注的函数，该装饰器将自动推断第一个参数的类型：

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

还可以使用*types.UnionType* 和*typing.Union*：

```
>>> @fun.register
... def _(arg: int | float, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> from typing import Union
>>> @fun.register
... def _(arg: Union[list, set], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
...
```

对于不使用类型标注的代码，可以将适当的类型参数显式地传给装饰器本身:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

对于在多项集类型 (例如 `list`) 上分派，但希望对多项集中的项设置类型提示 (例如 `list[int]`) 的代码，分派类型应当被显式地传给装饰器本身并将类型提示放在函数定义中:

```
>>> @fun.register(list)
... def _(arg: list[int], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

备注

当运行时函数将在一个列表的实例上分派而不管列表中包含的类型是什么，也就是说 `[1,2,3]` 将以与 `["foo", "bar", "baz"]` 相同的方式分派。在本例中提供的标注仅针对静态类型检查器而在运行时没有影响。

要启用注册*lambda* 和现有的函数，也可以使用 `register()` 属性的函数形式:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

`register()` 属性会返回未被装饰的函数。这将启用装饰器栈、封存，并为每个变量单独创建单元测试:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

在调用时，泛型函数会根据第一个参数的类型进行分派:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

在没有针对特定类型的已注册实现的情况下，会使用其方法解析顺序来查找更通用的实现。使用 `@singledispatch` 装饰的原始函数将为基本的 *object* 类型进行注册，这意味着它将在找不到更好的实现时被使用。

如果一个实现被注册到*abstract base class*，则基类的虚拟子类将被发送到该实现:

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

要检查泛型函数将为给定的类型选择哪个实现，请使用 `dispatch()` 属性:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # 注: 默认实现
<function fun at 0x103fe0000>
```

要访问所有已注册实现，请使用只读的 `registry` 属性:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Added in version 3.4.

在 3.7 版本发生变更: `register()` 属性现在支持使用类型标注。

在 3.11 版本发生变更: `register()` 属性现在支持将 *types.UnionType* 和 *typing.Union* 作为类型标注。

class functools.**singledispatchmethod**(*func*)

将一个方法转换为单分派 *generic function*。

要定义一个泛型方法，请用 `@singledispatchmethod` 装饰器来装饰它。当使用 `@singledispatchmethod` 定义一个函数时，请注意发送操作将针对第一个非 *self* 或非 *cls* 参数的类型上:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod` 支持与其他装饰器如*@classmethod* 相嵌套。请注意为了允许 `dispatcher.register`，`singledispatchmethod` 必须是最外层的装饰器。下面是一个 `Negator` 类包含绑定到类的 `neg` 方法，而不是一个类实例:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

同样的模式也可被用于其他类似的装饰器: *@staticmethod*, *@abstractmethod* 等等。

Added in version 3.8.

`functools.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

更新一个包装器函数以使其与被包装的函数相似。可选参数为指明原函数的哪些属性要被直接赋值给包装器函数的相匹配属性的元组以及包装器的哪些属性要使用原函数的相应属性来更新。这些参数的默认值是模块级常量 `WRAPPER_ASSIGNMENTS` (它将被赋值给包装器函数的 `__module__`, `__name__`, `__qualname__`, `__annotations__`, `__type_params__` 和 `__doc__`，即文档字符串) 以及 `WRAPPER_UPDATES` (它将更新包装器函数的 `__dict__`，即实例字典)。

为了允许出于内省和其他目的访问原始函数（例如绕过*lru_cache()* 之类的缓存装饰器），此函数会自动为 wrapper 添加一个指向被包装函数的 `__wrapped__` 属性。

此函数的主要目的是在*decorator* 函数中用来包装被装饰的函数并返回包装器。如果包装器函数未被更新，则被返回函数的元数据将反映包装器定义而不是原始函数定义，这通常没有什么用处。

update_wrapper() 可以与函数之外的可调用对象一同使用。在 *assigned* 或 *updated* 中命名的任何属性如果不存在于被包装对象则会被忽略（即该函数将不会尝试在包装器函数上设置它们）。如果包装器函数自身缺少在 *updated* 中命名的任何属性则仍将引发*AttributeError*。

在 3.2 版本发生变更: 现在 `__wrapped__` 属性会被自动添加。现在 `__annotations__` 属性默认会被拷贝。缺失的属性不会再触发*AttributeError*。

在 3.4 版本发生变更: `__wrapped__` 属性现在总是指向被包装的函数，即使该函数定义了 `__wrapped__` 属性。(参见 bpo-17482)

在 3.12 版本发生变更: 现在 __type_params__ 属性默认会被拷贝。

@functools.**wraps**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

这是一个便捷函数，用于在定义包装器函数时唤起*update_wrapper()* 作为函数装饰器。它等价于 `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`。例如:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

如果不使用这个装饰器工厂函数，则 example 函数的名称将变为 `'wrapper'`，并且 `example()` 原本的文档字符串将会丢失。

10.2.1 partial 对象

partial 对象是由*partial()* 创建的可调用对象。它们具有三个只读属性:

partial.**func**

一个可调用对象或函数。对*partial* 对象的调用将被转发给*func* 并附带新的参数和关键字。

partial.**args**

最左边的位置参数将放置在提供给*partial* 对象调用的位置参数之前。

partial.**keywords**

当调用*partial* 对象时将要提供的关键字参数。

partial 对象与 函数对象的类似之处在于它们都是可调用、可弱引用并可具有属性的。但两者也存在一些重要的区别。例如，__name__ 和 `function.__doc__` 属性不会被自动创建。此外，在类中定义的*partial* 对象的行为类似于静态方法并且不会在实例属性查找期间转换为绑定方法。

10.3 operator --- 标准运算符对应函数

源代码: Lib/operator.py

---

operator 模块提供了一套与 Python 的内置运算符对应的高效率函数。例如，`operator.add(x, y)` 与表达式 `x+y` 相同。许多函数名与特殊方法名相同，只是没有双下划线。为了向后兼容性，也保留了许多包含双下划线的函数。为了表述清楚，建议使用没有双下划线的函数。

函数包含的种类有：对象的比较运算、逻辑运算、数学运算以及序列运算。

对象比较函数适用于所有的对象，函数名根据它们对应的比较运算符命名。

operator.**lt**(*a*, *b*)

operator.**le**(*a*, *b*)

operator.**eq**(*a*, *b*)

operator.**ne**(*a*, *b*)

operator.**ge**(*a*, *b*)

operator.**gt**(*a*, *b*)

operator.**__lt__**(*a*, *b*)

operator.**__le__**(*a*, *b*)

operator.**__eq__**(*a*, *b*)

operator.**__ne__**(*a*, *b*)

operator.**__ge__**(*a*, *b*)

operator.**__gt__**(*a*, *b*)

在 *a* 和 *b* 之间进行全比较。具体的，`lt(a, b)` 与 `a < b` 相同，`le(a, b)` 与 `a <= b` 相同，`eq(a, b)` 与 `a == b` 相同，`ne(a, b)` 与 `a != b` 相同，`gt(a, b)` 与 `a > b` 相同，`ge(a, b)` 与 `a >= b` 相同。注意这些函数可以返回任何值，无论它是否可当作布尔值。关于全比较的更多信息请参考 comparisons 。

逻辑运算通常也适用于所有对象，并且支持真值检测、标识检测和布尔运算：

operator.**not_**(*obj*)

operator.**__not__**(*obj*)

返回 `not` *obj* 的结果。（请注意对象实例并没有 `__not__()` 方法；只有解释器核心可定义此操作。结果会受到 `__bool__()` 和 `__len__()` 方法的影响。）

operator.**truth**(*obj*)

如果 *obj* 为真值则返回*True*，否则返回*False*。这等价于使用*bool* 构造器。

operator.**is_**(*a*, *b*)

返回 `a is b`。检测对象标识。

operator.**is_not**(*a*, *b*)

返回 `a is not b`。检测对象标识。

数学和按位运算的种类是最多的：

operator.**abs**(*obj*)

operator.**__abs__**(*obj*)

返回 *obj* 的绝对值。

operator.**add**(*a*, *b*)

operator.**__add__**(*a*, *b*)

对于数字 *a* 和 *b*，返回 `a + b`。

operator.**and_**(*a*, *b*)

operator.**__and__**(*a*, *b*)

返回 *x* 和 *y* 按位与的结果。

operator.**floordiv**(*a*, *b*)

operator.**__floordiv__**(*a*, *b*)

返回 `a // b`。

operator.**index**(*a*)

operator.**__index__**(*a*)

返回 *a* 转换为整数的结果。等价于 `a.__index__()`。

在 3.10 版本发生变更: 结果总是为*int* 类型。在之前版本中，结果可能为 `int` 的子类的实例。

operator.**inv**(*obj*)

operator.**invert**(*obj*)

operator.**__inv__**(*obj*)

operator.**__invert__**(*obj*)
返回数字 *obj* 按位取反的结果。这等价于 `~obj`。

operator.**lshift**(*a*, *b*)
operator.**__lshift__**(*a*, *b*)
返回 *a* 左移 *b* 位的结果。

operator.**mod**(*a*, *b*)
operator.**__mod__**(*a*, *b*)
返回 `a % b`。

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a * b`。

operator.**matmul**(*a*, *b*)
operator.**__matmul__**(*a*, *b*)
返回 `a @ b`。

Added in version 3.5.

operator.**neg**(*obj*)
operator.**__neg__**(*obj*)
返回 *obj* 取负的结果 (`-obj`)。

operator.**or_**(*a*, *b*)
operator.**__or__**(*a*, *b*)
返回 *a* 和 *b* 按位或的结果。

operator.**pos**(*obj*)
operator.**__pos__**(*obj*)
返回 *obj* 取正的结果 (`+obj`)。

operator.**pow**(*a*, *b*)
operator.**__pow__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a ** b`。

operator.**rshift**(*a*, *b*)
operator.**__rshift__**(*a*, *b*)
返回 *a* 右移 *b* 位的结果。

operator.**sub**(*a*, *b*)
operator.**__sub__**(*a*, *b*)
返回 `a - b`。

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
返回 `a / b` 例如 2/3 将等于.66 而不是 0。这也被称为“真”除法。

operator.**xor**(*a*, *b*)
operator.**__xor__**(*a*, *b*)
返回 *a* 和 *b* 按位异或的结果。

适用于序列的操作（其中一些也适用于映射）包括:

operator.**concat**(*a*, *b*)
operator.**__concat__**(*a*, *b*)
对于序列 *a* 和 *b*，返回 `a + b`。

operator.**contains**(*a*, *b*)

operator.**__contains__**(*a*, *b*)

返回 `b in a` 检测的结果。请注意操作数是反序的。

operator.**countOf**(*a*, *b*)

返回 *b* 在 *a* 中的出现次数。

operator.**delitem**(*a*, *b*)

operator.**__delitem__**(*a*, *b*)

移除 *a* 中索引号为 *b* 的值。

operator.**getitem**(*a*, *b*)

operator.**__getitem__**(*a*, *b*)

返回 *a* 中索引为 *b* 的值。

operator.**indexOf**(*a*, *b*)

返回 *b* 在 *a* 中首次出现所在的索引号。

operator.**setitem**(*a*, *b*, *c*)

operator.**__setitem__**(*a*, *b*, *c*)

将 *a* 中索引号为 *b* 的值设为 *c*。

operator.**length_hint**(*obj*, *default=0*)

返回对象 *obj* 的估计长度。首先尝试返回其实际长度，再使用 `object.__length_hint__()` 得出估计值，最后返回默认值。

Added in version 3.4.

以下操作适用于可调用对象:

operator.**call**(*obj*, */*, **args*, ***kwargs*)

operator.**__call__**(*obj*, */*, **args*, ***kwargs*)

返回 `obj(*args, **kwargs)`。

Added in version 3.11.

operator 模块还定义了一些用于常规属性和条目查找的工具。这些工具适合用来编写快速字段提取器作为*map()*, *sorted()*, *itertools.groupby()* 或其他需要相应函数参数的函数的参数。

operator.**attrgetter**(*attr*)

operator.**attrgetter**(**attrs*)

返回一个可从操作数中获取 *attr* 的可调用对象。如果请求了一个以上的属性，则返回一个属性元组。属性名称还可包含点号。例如:

- 在 `f = attrgetter('name')` 之后，调用 `f(b)` 将返回 `b.name`。
- 在 `f = attrgetter('name', 'date')` 之后，调用 `f(b)` 将返回 `(b.name, b.date)`。
- 在 `f = attrgetter('name.first', 'name.last')` 之后，调用 `f(b)` 将返回 `(b.name.first, b.name.last)`。

等价于:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g
```

(续下页)

(接上页)

```
def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)

operator.**itemgetter**(**items*)

返回一个使用操作数的 __getitem__() 方法从操作数中获取 *item* 的可调用对象。如果指定了多个条目，则返回一个查找值的元组。例如：

- 在 `f = itemgetter(2)` 之后，调用 `f(r)` 将返回 `r[2]`。
- 在 `g = itemgetter(2, 5, 3)` 之后，调用 `g(r)` 将返回 `(r[2], r[5], r[3])`。

等价于：

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

条目可以是操作数的 __getitem__() 方法所接受的任何类型。字典接受任意*hashable* 值。列表、元组和字符串接受索引或切片对象：

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

使用*itemgetter()* 从元组的记录中提取特定字段的例子：

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*, */*, **args*, ***kwargs*)

返回一个在操作数上调用 *name* 方法的可调用对象。如果给出额外的参数和/或关键字参数，它们也将被传给该方法。例如：

- 在 `f = methodcaller('name')` 之后，调用 `f(b)` 将返回 `b.name()`。
- 在 `f = methodcaller('name', 'foo', bar=1)` 之后，调用 `f(b)` 将返回 `b.name('foo', bar=1)`。

等价于：

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
```

(续下页)

(接上页)

```
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 将运算符映射到函数

以下表格显示了抽象运算是如何对应于 Python 语法中的运算符和*operator* 模块中的函数的。

运算	语法	函数
加法	`a + b`	`add(a, b)`
字符串拼接	`seq1 + seq2`	`concat(seq1, seq2)`
包含测试	`obj in seq`	`contains(seq, obj)`
除法	`a / b`	`truediv(a, b)`
除法	`a // b`	`floordiv(a, b)`
按位与	`a & b`	`and_(a, b)`
按位异或	`a ^ b`	`xor(a, b)`
按位取反	`~ a`	`invert(a)`
按位或	`a | b`	`or_(a, b)`
取幂	`a ** b`	`pow(a, b)`
标识	`a is b`	`is_(a, b)`
标识	`a is not b`	`is_not(a, b)`
索引赋值	`obj[k] = v`	`setitem(obj, k, v)`
索引删除	`del obj[k]`	`delitem(obj, k)`
索引取值	`obj[k]`	`getitem(obj, k)`
左移	`a << b`	`lshift(a, b)`
取模	`a % b`	`mod(a, b)`
乘法	`a * b`	`mul(a, b)`
矩阵乘法	`a @ b`	`matmul(a, b)`
取反（算术）	`- a`	`neg(a)`
取反（逻辑）	`not a`	`not_(a)`
正数	`+ a`	`pos(a)`
右移	`a >> b`	`rshift(a, b)`
切片赋值	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
切片删除	`del seq[i:j]`	`delitem(seq, slice(i, j))`
切片取值	`seq[i:j]`	`getitem(seq, slice(i, j))`
字符串格式化	`s % obj`	`mod(s, obj)`
减法	`a - b`	`sub(a, b)`
真值测试	`obj`	`truth(obj)`
比较	`a < b`	`lt(a, b)`
比较	`a <= b`	`le(a, b)`
相等	`a == b`	`eq(a, b)`
不等	`a != b`	`ne(a, b)`
比较	`a >= b`	`ge(a, b)`
比较	`a > b`	`gt(a, b)`

10.3.2 原地运算符

许多运算都有“原地”版本。以下列出的是提供对原地运算符相比通常语法更底层访问的函数，例如*statement* `x += y` 相当于 `x = operator.iadd(x, y)`。换一种方式来讲就是 `z = operator.iadd(x, y)` 等价于语句块 `z = x; z += y`。

在这些例子中，请注意当调用一个原地方法时，运算和赋值是分成两个步骤来执行的。下面列出的原地函数只执行第一步即调用原地方法。第二步赋值则不加处理。

对于不可变的目标例如字符串、数字和元组，更新的值会被计算，但不会被再被赋值给输入变量：

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

对于可变的目标例如列表和字典，原地方法将执行更新，因此不需要后续赋值操作：

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)

operator.**__iadd__**(*a*, *b*)

a = iadd(a, b) 等价于 a += b。

operator.**iand**(*a*, *b*)

operator.**__iand__**(*a*, *b*)

a = iand(a, b) 等价于 a &= b。

operator.**iconcat**(*a*, *b*)

operator.**__iconcat__**(*a*, *b*)

a = iconcat(a, b) 等价于 a += b 其中 *a* 和 *b* 为序列。

operator.**ifloordiv**(*a*, *b*)

operator.**__ifloordiv__**(*a*, *b*)

a = ifloordiv(a, b) 等价于 a //= b。

operator.**ilshift**(*a*, *b*)

operator.**__ilshift__**(*a*, *b*)

a = ilshift(a, b) 等价于 a <<= b。

operator.**imod**(*a*, *b*)

operator.**__imod__**(*a*, *b*)

a = imod(a, b) 等价于 a %= b。

operator.**imul**(*a*, *b*)

operator.**__imul__**(*a*, *b*)

a = imul(a, b) 等价于 a *= b。

operator.**imatmul**(*a*, *b*)

operator.**__imatmul__**(*a*, *b*)

a = imatmul(a, b) 等价于 a @= b。

Added in version 3.5.

operator.**ior**(*a*, *b*)

operator.**__ior__**(*a*, *b*)

a = ior(a, b) 等价于 a |= b。

operator.**ipow**(*a*, *b*)

operator.**__ipow__**(*a*, *b*)

a = ipow(a, b) 等价于 a **= b。

operator.**irshift**(*a*, *b*)

operator.**__irshift__**(*a*, *b*)

a = irshift(a, b) 等价于 a >>= b。

operator.**isub**(*a*, *b*)

operator.**__isub__**(*a*, *b*)

a = isub(a, b) 等价于 a -= b。

operator.**itruediv**(*a*, *b*)

operator.**__itruediv__**(*a*, *b*)

a = itruediv(a, b) 等价于 a /= b。

operator.**ixor**(*a*, *b*)

operator.**__ixor__**(*a*, *b*)

a = ixor(a, b) 等价于 a ^= b。

CHAPTER 11

文件和目录访问

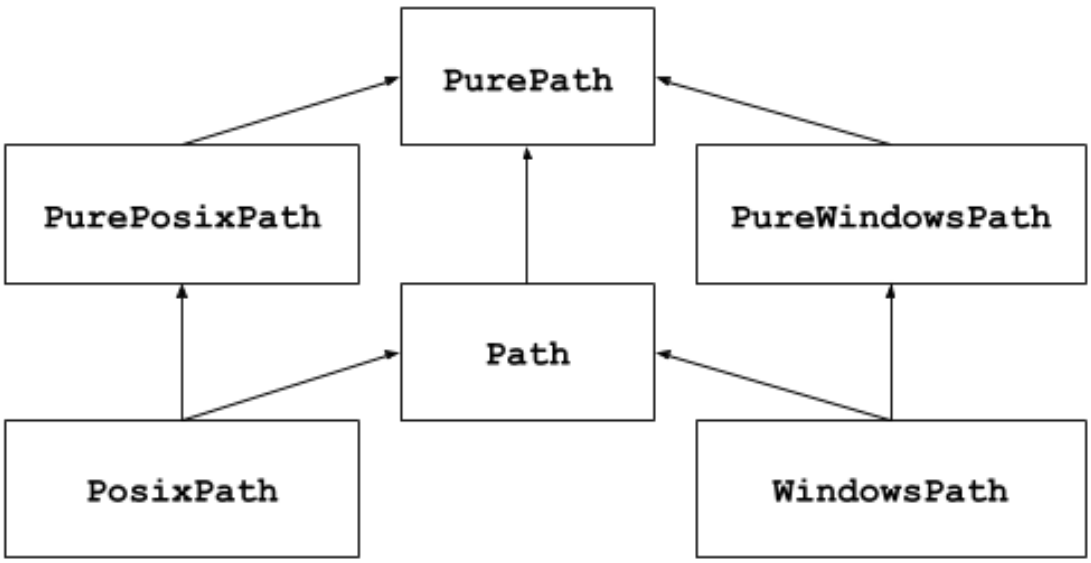
本章中描述的模块处理磁盘文件和目录。例如，有一些模块用于读取文件的属性，以可移植的方式操作路径以及创建临时文件。本章的完整模块列表如下：

11.1 `pathlib` --- 面向对象的文件系统路径

Added in version 3.4.

源代码: Lib/pathlib/

该模块提供表示文件系统路径的类，其语义适用于不同的操作系统。路径类被分为提供纯计算操作而没有 I/O 的纯路径，以及从纯路径继承而来但提供 I/O 操作的具体路径。



如果以前从未用过此模块，或不确定哪个类适合完成任务，那要用的可能就是*Path*。它在运行代码的平台上实例化为具体路径。

在一些用例中纯路径很有用，例如：

1. 如果你想要在 Unix 设备上操作 Windows 路径（或者相反）。你不应在 Unix 上实例化一个*WindowsPath*，但是你可以实例化*PureWindowsPath*。
2. 你只想操作路径但不想实际访问操作系统。在这种情况下，实例化一个纯路径是有用的，因为它们没有任何访问操作系统的操作。

参见

PEP 428：pathlib 模块 -- 面向对象的文件系统路径。

参见

对于底层的路径字符串操作，你也可以使用*os.path*模块。

11.1.1 基础使用

导入主类:

```
>>> from pathlib import Path
```

列出子目录:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

列出当前目录树下的所有 Python 源代码文件:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

在目录树中移动:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

查询路径的属性:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

打开一个文件:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 异常

exception pathlib.**UnsupportedOperation**

一个继承自*NotImplementedError* 的异常，当在路径对象上调用不受支持的操作时它将被引发。

Added in version 3.13.

11.1.3 纯路径

纯路径对象提供了不实际访问文件系统的路径处理操作。有三种方式来访问这些类，也是不同的风格：

class pathlib.**PurePath**(**pathsegments*)

一个通用的类，代表当前系统的路径风格（实例化为*PurePosixPath* 或者*PureWindowsPath*）：

```
>>> PurePath('setup.py')      # 在 Unix 机器上运行
PurePosixPath('setup.py')
```

pathsegments 的每个元素既可以是代表一个路径段的字符串，也可以是实现了*os.PathLike* 接口的对象，其中*__fspath__()* 方法返回一个字符串，例如另一个路径对象：

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

当 *pathsegments* 为空的时候，假定为当前目录：

```
>>> PurePath()
PurePosixPath('.')
```

如果某个段为绝对路径，则其前面的所有段都会被忽略 (类似*os.path.join()*)：

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

在 Windows 上，当遇到带根符号的路径段 (如 `r'\foo'`) 时驱动器将不会被重置：

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

假斜杠和单个点号会被消除，但双点号 (`'..'`) 和打头的双斜杠 (`'//'`) 不会，因为这会出于各种原因改变路径的实际含义 (例如符号链接、UNC 路径等)：

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('//foo/bar')
PurePosixPath('//foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

（一个很 naïve 的做法是让 `PurePosixPath('foo/../bar')` 等同于 `PurePosixPath('bar')`，如果 `foo` 是一个指向其他目录的符号链接那么这个做法就将出错）

纯路径对象实现了*os.PathLike* 接口，允许它们在任何接受此接口的地方使用。

在 3.6 版本发生变更: 添加了*os.PathLike* 接口支持。

class pathlib.**PurePosixPath**(**pathsegments*)

一个*PurePath* 的子类，路径风格不同于 Windows 文件系统：

```
>>> PurePosixPath('/etc/hosts')
PurePosixPath('/etc/hosts')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**PureWindowsPath**(**pathsegments*)

PurePath 的一个子类，此路径风格代表 Windows 文件系统路径，包括 UNC paths:

```
>>> PureWindowsPath('c:/', 'Users', 'Ximénez')
PureWindowsPath('c:/Users/Ximénez')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

pathsegments 参数的指定和*PurePath* 相同。

无论你正运行什么系统，你都可以实例化这些类，因为它们提供的操作不做任何系统调用。

通用性质

路径是不可变并且*hashable*。相同风格的路径可以排序和比较。这此特性会尊重对应风格的大小写转换语义。:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

不同风格的路径比较得到不等的结果并且无法被排序:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath'
```

运算符

斜杠操作符可以帮助创建子路径，如*os.path.join()*。如果参数为一个绝对路径，则之前的路径会被忽略。在 Windows 上，当参数为一个带根符号的相对路径 (如 `r'\foo'`) 时驱动器将不会被重置:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

文件对象可用于任何接受*os.PathLike* 接口实现的地方。

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

路径的字符串表示法为它自己原始的文件系统路径（以原生形式，例如在 Windows 下使用反斜杠）。你可以传递给任何需要字符串形式路径的函数。

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

类似地，在路径上调用*bytes*将原始文件系统路径作为字节对象给出，就像被*os.fsencode()*编码一样:

```
>>> bytes(p)
b'/etc'
```

备注

只推荐在 Unix 下调用*bytes*。在 Windows，unicode 形式是文件系统路径的规范表示法。

访问个别部分

为了访问路径独立的部分（组件），使用以下特征属性:

PurePath.**parts**

一个元组，可以访问路径的多个组件:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

（注意盘符和本地根目录是如何重组的）

方法和特征属性

纯路径提供以下方法和特征属性:

PurePath.**parser**

用于低层级路径解析与合并的*os.path*模块的实现: `posixpath` 或 `ntpath`。

Added in version 3.13.

PurePath.**drive**

一个表示驱动器盘符或命名的字符串，如果存在:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC 分享也被认作驱动器:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.**root**

一个表示（本地或全局）根的字符串，如果存在:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC 分享一样拥有根:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

如果路径以超过两个连续斜框打头，*PurePosixPath* 会合并它们:

```
>>> PurePosixPath('//etc').root
'//'
>>> PurePosixPath('///etc').root
'/'
>>> PurePosixPath('////etc').root
'/'
```

备注

此行为符合 *The Open Group Base Specifications Issue 6*, paragraph 4.11 Pathname Resolution:

"以连续两个斜杠打头的路径名可能会以具体实现所定义的方式被解读，但是两个以上的前缀斜杠则应当被当作一个斜杠来处理。"

PurePath.**anchor**

驱动器和根的联合:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**

提供访问此路径的逻辑祖先的不可变序列:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

在 3.10 版本发生变更: parents 序列现在支持切片 和负的索引值。

PurePath.**parent**

此路径的逻辑父路径:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

你不能超过一个 anchor 或空路径:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

备注

这是一个单纯的词法操作，因此有以下行为:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

如果你想要向上遍历任意文件系统路径，建议首先调用*Path.resolve()* 以便解析符号链接并消除 `".."` 部分。

PurePath.**name**

一个表示最后路径组件的字符串，排除了驱动器与根目录，如果存在的话:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC 驱动器名不被考虑:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**

最后一个路径组件中以点分割的后一部分（如果有）

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

这通常被称作文件扩展名。

PurePath.**suffixes**

由路径后缀组成的列表，经常被称作文件扩展名:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
```

(续下页)

(接上页)

```
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.**stem**

最后一个路径组件，除去后缀:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()

返回使用正斜杠（/）的路径字符串:

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**is_absolute**()

返回此路径是否为绝对路径。如果路径同时拥有驱动器符与根路径（如果风格允许）则将被认作绝对路径。

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_relative_to**(*other*)

返回此路径是否相对于 *other* 的路径。

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

此方法是基于字符串的；它不会访问文件系统也不会对".." 部分进行特殊处理。以下代码是等价的:

```
>>> u = PurePath('/usr')
>>> u == p or u in p.parents
False
```

Added in version 3.9.

Deprecated since version 3.12, will be removed in version 3.14: 传入附加参数的做法已被弃用；如果提供了附加参数，它们将与 *other* 合并。

PurePath.**is_reserved**()

在*PureWindowsPath*，如果路径是被 Windows 保留的则返回 `True`，否则 `False`。在*PurePosixPath*，总是返回 `False`。

在 3.13 版本发生变更: Windows 的含有一个冒号或以点或空格结尾的路径名是被认为是被保留的。UNC 路径也可能被保留。

Deprecated since version 3.13, will be removed in version 3.15: 此方法已被弃用；使用*os.path.isreserved()* 以鉴别在 Windows 下的保留路径。

PurePath.**joinpath**(**pathsegments*)

调用此方法等同于依次将路径与给定的每个 *pathsegments* 组合到一起:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**full_match**(*pattern*, *, *case_sensitive=None*)

将此路径与所提供的 glob 样式匹配。如果匹配成功，则返回 “True“，否则返回 “False“。例如:

```
>>> PurePath('a/b.py').full_match('a/*.py')
True
>>> PurePath('a/b.py').full_match('*.py')
False
>>> PurePath('/a/b/c.py').full_match('/a/**')
True
>>> PurePath('/a/b/c.py').full_match('**/*.py')
True
```

参见

模式语言 文档。

与其他方法一样，是否大小写敏感遵循平台的默认规则:

```
>>> PurePosixPath('b.py').full_match('*.PY')
False
>>> PureWindowsPath('b.py').full_match('*.PY')
True
```

将 *case_sensitive* 设为 `True` 或 `False` 来覆盖此行为。

Added in version 3.13.

PurePath.**match**(*pattern*, *, *case_sensitive=None*)

将此路径与所提供的非递归 glob 样式匹配。如果匹配成功，则返回 “True“，否则返回 “False“。

此方法与*full_match()* 类似，但不允许空模式 (将引发*ValueError*)，也不支持递归通配符”**”(将视为非递归的”*”)，并且如果提供了一个相对格式，则将从右开始匹配:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

在 3.12 版本发生变更: *pattern* 形参接受一个*path-like object*。

在 3.12 版本发生变更: 增加了 *case_sensitive* 形参。

PurePath.**relative_to**(*other*, *walk_up=False*)

计算此路径相对于 *other* 所表示路径的版本。如果不可计算，则引发*ValueError*:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR one path is relative and␣
↪the other is absolute.
```

当 *walk_up* 为（默认的）假值时，路径必须以 *other* 开始。当参数为真值时，可能会添加 `..` 条目以形成相对路径。在所有其他情况下，例如路径引用了不同的驱动器，则会引发*ValueError*。:

```
>>> p.relative_to('/usr', walk_up=True)
PurePosixPath('../etc/passwd')
>>> p.relative_to('foo', walk_up=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not on the same drive as 'foo' OR one path is relative and␣
↪the other is absolute.
```

警告

该函数是*PurePath* 的一部分并适用于字符串。它不会检查或访问下层的文件结构体。这可能会影响 *walk_up* 选项因为它假定路径中不存在符号链接；如果需要处理符号链接请先调用*resolve()*。

在 3.12 版本发生变更: 增加了 *walk_up* 形参数（原本的行为与 `walk_up=False` 相同）。

Deprecated since version 3.12, will be removed in version 3.14: 传入附加位置参数的做法已被弃用；如果提供的话，它们将与 *other* 合并。

PurePath.**with_name**(*name*)

返回一个新的路径并修改*name*。如果原本路径没有 name，ValueError 被抛出:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_stem**(*stem*)

返回一个带有修改后*stem* 的新路径。如果原路径没有名称，则会引发 ValueError:

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

Added in version 3.9.

PurePath.**with_suffix**(*suffix*)

返回一个新的路径并修改*suffix*。如果原本的路径没有后缀，新的 *suffix* 则被追加以代替。如果 *suffix* 是空字符串，则原本的后缀被移除:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

PurePath.**with_segments**(**pathsegments*)

通过组合给定的 *pathsegments* 创建一个相同类型的新路径对象。每当创建派生路径，如从*parent*和*relative_to()* 创建时都会调用此方法。子类可以覆盖此方法以便向派生路径传递信息，例如:

```
from pathlib import PurePosixPath

class MyPath(PurePosixPath):
    def __init__(self, *pathsegments, session_id):
        super().__init__(*pathsegments)
        self.session_id = session_id

    def with_segments(self, *pathsegments):
        return type(self)(*pathsegments, session_id=self.session_id)

etc = MyPath('/etc', session_id=42)
hosts = etc / 'hosts'
print(hosts.session_id)  # 42
```

Added in version 3.12.

11.1.4 具体路径

具体路径是纯路径的子类。除了后者提供的操作之外，它们还提供了对路径对象进行系统调用的方法。有三种方法可以实例化具体路径:

class pathlib.**Path**(**pathsegments*)

一个*PurePath* 的子类，此类以当前系统的路径风格表示路径（实例化为*PosixPath* 或*WindowsPath*）：

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**PosixPath**(**pathsegments*)

一个*Path* 和*PurePosixPath* 的子类，此类表示一个非 Windows 文件系统的具体路径:

```
>>> PosixPath('/etc/hosts')
PosixPath('/etc/hosts')
```

pathsegments 参数的指定和*PurePath* 相同。

在 3.13 版本发生变更: 在 Windows 上引发*UnsupportedOperation*。在之前版本中，则为引发*NotImplementedError*。

class pathlib.**WindowsPath**(**pathsegments*)

Path 和*PureWindowsPath* 的子类，从类表示一个 Windows 文件系统的具体路径:

```
>>> WindowsPath('c:/', 'Users', 'Ximénez')
WindowsPath('c:/Users/Ximénez')
```

pathsegments 参数的指定和*PurePath* 相同。

在 3.13 版本发生变更: 在非 Windows 平台上引发*UnsupportedOperation*。在之前版本中，则为引发*NotImplementedError*。

你只能实例化与当前系统风格相同的类（允许系统调用作用于不兼容的路径风格可能在应用程序中导致缺陷或失败）：

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
UnsupportedOperation: cannot instantiate 'WindowsPath' on your system
```

某些具体路径方法在一个系统调用失败时（例如由于路径不存在）可能引发*OSError*。

解析和生成 URI

具体路径对象可基于符合 **RFC 8089** 的’ 文件’ URI 来创建，并可用它来表示。

备注

文件 URI 不能在具有不同*文件系统编码格式* 的机器之间进行移植。

classmethod Path.**from_uri**(*uri*)

通过解析一个’ 文件’ URI 来返回新的路径对象。例如:

```
>>> p = Path.from_uri('file:///etc/hosts')
PosixPath('/etc/hosts')
```

在 Windows 上，可以基于 URI 来解析 DOS 设备和 UNC 路径:

```
>>> p = Path.from_uri('file:///c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file://server/share')
WindowsPath('//server/share')
```

某些变化形式也是受支持的:

```
>>> p = Path.from_uri('file:////server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file://///server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file:c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file:/c|/windows')
WindowsPath('c:/windows')
```

如果 URI 不是以 `file:` 开头，或者被解析的不是绝对路径则会引发*ValueError*。

Added in version 3.13.

`Path.`**`as_uri`**`()`

将路径表示为'file' URI。如果路径不是绝对路径则会引发*ValueError*。

```
>>> p = PosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = WindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

出于历史原因，该方法在*PurePath*对象中也可用。不过，由于它使用了*os.fsencode()*因此严格来说并不纯净。

扩展和计算路径

`classmethod` `Path.`**`home`**`()`

返回一个表示用户家目录的新路径对象（与带 ~ 构造的*os.path.expanduser()*所返回的相同）。如果无法解析家目录，则会引发*RuntimeError*。

```
>>> Path.home()
PosixPath('/home/antoine')
```

Added in version 3.5.

`Path.`**`expanduser`**`()`

返回带有扩展 ~ 和 ~user 构造的新路径，与*os.path.expanduser()*所返回的相同。如果无法解析家目录，则会引发*RuntimeError*。

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Added in version 3.5.

`classmethod` `Path.`**`cwd`**`()`

返回一个新的表示当前目录的路径对象（和*os.getcwd()*返回的相同）：

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

Path.**absolute**()

改为绝对路径，不会执行正规化或解析符号链接。返回一个新的路径对象:

```
>>> p = Path('tests')
>>> p
PosixPath('tests')
>>> p.absolute()
PosixPath('/home/antoine/pathlib/tests')
```

Path.**resolve**(*strict=False*)

将路径绝对化，解析任何符号链接。返回新的路径对象:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

”..” 组件也将被消除（只有这一种方法这么做）：

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

如果一个路径不存在或是遇到了符号链接循环，并且 *strict* 为 `True`，则会引发*OSError*。如果 *strict* 为 `False`，则会尽可能地解析路径并添加任何剩余部分而不会检查其是否存在。

在 3.6 版本发生变更: 增加了 *strict* 参数（3.6 之前的行为是默认采用 strict 模式）。

在 3.13 版本发生变更: 符号链接循环将像其他错误一样处理：在严格模式下会引发*OSError*，而在非严格模式下不会引发异常。在之前版本中，无论 *strict* 的值是什么都会引发*RuntimeError*。

Path.**readlink**()

返回符号链接所指向的路径（即*os.readlink()* 的返回值）：

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.readlink()
PosixPath('setup.py')
```

Added in version 3.9.

在 3.13 版本发生变更: 如果*os.readlink()* 不可用则会引发*UnsupportedOperation*。在之前版本中，则是引发*NotImplementedError*。

查询文件类型和状态

在 3.8 版本发生变更: 现在对于包含在 OS 层级上无法表示的字符的路径*exists()*, *is_dir()*, *is_file()*, *is_mount()*, *is_symlink()*, *is_block_device()*, *is_char_device()*, *is_fifo()*, *is_socket()* 将返回 `False` 而不是引发异常。

Path.**stat**(*, *follow_symlinks=True*)

返回一个*os.stat_result* 对象，其中包含有关此路径的信息，就像*os.stat()*。结果会在每次调用此方法时被查找。

此方法通常会跟随符号链接；要对 symlink 使用 stat 请添加参数 `follow_symlinks=False`，或者使用*lstat()*。

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

在 3.10 版本发生变更: 增加了 *follow_symlinks* 形参。

Path.**lstat**()

就和*Path.stat()* 一样，但是如果路径指向符号链接，则是返回符号链接而不是目标的信息。

Path.**exists**(*, *follow_symlinks=True*)

如果路径指向现有的文件或目录则返回 `True`。

此方法通常会跟随符号链接；要检查符号链接是否存在，请添加参数 `follow_symlinks=False`。

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

在 3.12 版本发生变更: 增加了 *follow_symlinks* 形参。

Path.**is_file**(*, *follow_symlinks=True*)

如果路径指向一个常规文件则返回 `True`，如果是指向其他种类的文件则返回 `False`。

当路径不存在或者是一个破损的符号链接时也会返回 `False`；其他错误（例如权限错误）被传播。

此方法通常会跟随符号链接；要排除符号链接，请添加参数 `follow_symlinks=False`。

在 3.13 版本发生变更: 增加了 *follow_symlinks* 形参。

Path.**is_dir**(*, *follow_symlinks=True*)

如果路径指向一个目录则返回 `True`，如果是指向其他种类的文件则返回 `False`。

当路径不存在或者是一个破损的符号链接时也会返回 `False`；其他错误（例如权限错误）被传播。

此方法通常会跟随符号链接；要排除指向目录的符号链接，请添加参数 `follow_symlinks=False`。

在 3.13 版本发生变更: 增加了 *follow_symlinks* 形参。

Path.**is_symlink**()

如果路径指向符号链接则返回 `True`，否则 `False`。

如果路径不存在也返回 `False`；其他错误（例如权限错误）被传播。

Path.**is_junction**()

如果路径是指向一个接合点则返回 `True`，如果是其他文件类型则返回 `False`。目前只有 Windows 支持接合点。

Added in version 3.12.

Path.**is_mount**()

如果路径是一个 挂载点: 在文件系统中被其他不同文件系统挂载的位置则返回 `True`。在 POSIX 上，此函数将检查 *path* 的上一级 `path/..` 是否位于和 *path* 不同的设备中，或者 `path/..` 和 *path* 是否指向位于相同设置的相同 i-node --- 这应当能检测所有 Unix 和 POSIX 变种上的挂载点。在 Windows 上，挂载点是被视为驱动器盘符的根目录 (例如 `c:\`)、UNC 共享目录 (例如 `\\server\share`) 或已挂载的文件系统目录。

Added in version 3.7.

在 3.12 版本发生变更: 添加了 Windows 支持。

Path.**is_socket**()

如果路径指向一个 Unix socket 文件（或者指向 Unix socket 文件的符号链接）则返回 `True`，如果指向其他类型的文件则返回 `False`。

当路径不存在或者是一个破损的符号链接时也会返回 `False`；其他错误（例如权限错误）被传播。

Path.**is_fifo**()

如果路径指向一个先进先出存储（或者指向先进先出存储的符号链接）则返回 `True`，指向其他类型的文件则返回 `False`。

当路径不存在或者是一个破损的符号链接时也会返回 `False`；其他错误（例如权限错误）被传播。

Path.**is_block_device**()

如果文件指向一个块设备（或者指向块设备的符号链接）则返回 `True`，指向其他类型的文件则返回 `False`。

当路径不存在或者是一个破损的符号链接时也会返回 `False`；其他错误（例如权限错误）被传播。

Path.**is_char_device**()

如果路径指向一个字符设备（或指向字符设备的符号链接）则返回 `True`，指向其他类型的文件则返回 `False`。

当路径不存在或者是一个破损的符号链接时也会返回 `False`；其他错误（例如权限错误）被传播。

Path.**samefile**(*other_path*)

返回此目录是否指向与可能是字符串或者另一个路径对象的 *other_path* 相同的文件。语义类似于*os.path.samefile()* 与*os.path.samestat()*。

如果两者都以同一原因无法访问，则抛出*OSError*。

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Added in version 3.5.

读写文件

Path.**open**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

打开路径指向的文件，就像内置的*open()* 函数所做的一样:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

Path.**read_text**(*encoding=None*, *errors=None*, *newline=None*)

以字符串形式返回路径指向的文件的解码后文本内容。

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

文件先被打开然后关闭。有和*open()* 一样的可选形参。

Added in version 3.5.

在 3.13 版本发生变更: 增加了 *newline* 形参。

Path.**read_bytes**()

以字节对象的形式返回路径指向的文件的二进制内容:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Added in version 3.5.

Path.**write_text**(*data*, *encoding=None*, *errors=None*, *newline=None*)

将文件以文本模式打开，写入 *data* 并关闭:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

同名的现有文件会被覆盖。可选形参的含义与*open()* 的相同。

Added in version 3.5.

在 3.10 版本发生变更: 增加了 *newline* 形参。

Path.**write_bytes**(*data*)

将文件以二进制模式打开，写入 *data* 并关闭:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

一个同名的现存文件将被覆盖。

Added in version 3.5.

读取目录

Path.**iterdir**()

当路径指向一个目录时，产生该路径下的对象的路径:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

子条目会以任意顺序产生，并且不包括特殊条目 `'.'` 和 `'..'`。如果迭代器创建之后有文件在目录中被移除或添加，是否要包括该文件所对应的路径对象并没有明确规定。

如果该路径不是一个目录或是无法访问，则会引发*OSError*。

Path.**glob**(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

解析相对于此路径的通配符 *pattern*，产生所有匹配的文件:

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
```

(续下页)

(接上页)

```
[PosixPath('docs/conf.py')]
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

参见

模式语言 文档。

在默认情况下，或当 *case_sensitive* 关键字参数被设为 `None` 时，该方法将使用特定平台的大小写规则匹配路径：通常，在 POSIX 上区分大小写，而在 Windows 上不区分大小写。将 *case_sensitive* 设为 `True` 或 `False` 可覆盖此行为。

在默认情况下，或是当 *recurse_symlinks* 关键字参数被设为 `False` 时，此方法将跟随符号链接但在扩展"`**`" 通配符时除外。将 *recurse_symlinks* 设为 `True` 将总是跟随符号链接。

引发一个审计事件 `pathlib.Path.glob` 并附带参数 `self`, `pattern`。

在 3.12 版本发生变更: 增加了 *case_sensitive* 形参。

在 3.13 版本发生变更: 增加了 *recurse_symlinks* 形参。

在 3.13 版本发生变更: *pattern* 形参接受一个*path-like object*。

在 3.13 版本发生变更: 任何因扫描文件系统而引发的`OSError` 异常都会被抑制。在之前的版本中，此类异常在许多情况下都会被抑制，但并非全部情况。

`Path.`**`rglob`**(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

递归地对给定的相对 *pattern* 执行 glob 通配。这类似于调用`Path.glob()` 时在 *pattern* 之前加上"`**/`"。

参见

模式语言 和`Path.glob()` 文档。

引发一个审计事件 `pathlib.Path.rglob` 并附带参数 `self`, `pattern`。

在 3.12 版本发生变更: 增加了 *case_sensitive* 形参。

在 3.13 版本发生变更: 增加了 *recurse_symlinks* 形参。

在 3.13 版本发生变更: *pattern* 形参接受一个*path-like object*。

`Path.`**`walk`**(*top_down=True*, *on_error=None*, *follow_symlinks=False*)

通过对目录树自上而下或自下而上的遍历来生成其中的文件名。

对于根位置为 *self* 的目录树中的每个目录（包括 *self* 但不包括'.' 和'..'），该方法会产生一个 3 元组 `(dirpath, dirnames, filenames)`。

dirpath 是指向当前正被遍历到的目录的`Path`，*dirnames* 是由表示 *dirpath* 中子目录名称的字符串组成的列表 (不包括 `'.'` 和 `'..'`)，*filenames* 是由表示 *dirpath* 中非目录文件名称的字符串组成的列表。要获取 *dirpath* 中文件或目录的完整路径 (以 *self* 开头)，可使用 `dirpath / name`。这些列表是否排序取决于具体的文件系统。

如果可选参数 *top_down* 为（默认的）真值，则会在所有子目录的三元组生成之前生成父目录的三元组（目录是自上而下遍历的）。如果 *top_down* 为假值，则会在所有子目录的三元组生成之后再生成父目录的三元组（目录是是自下而上遍历的）。无论 *top_down* 的值是什么，都会在遍历目录及其子目录的三元组之前提取子目录列表。

当 *top_down* 为真值时，调用方可以原地修改 *dirnames* 列表（例如，使用 `del` 或切片赋值），而 *Path.walk()* 只会向名称保留在 *dirnames* 中的子目录递归。这可被用于搜索剪枝，或强制应用特定的访问顺序，或者甚至是在重新恢复执行 *Path.walk()* 之前告知 *Path.walk()* 调用方所创建或重命名的目录。当 *top_down* 为假值时修改 *dirnames* 不会对 *Path.walk()* 的的行为造成影响，因为在 *dirnames* 被提供给调用方时 *dirnames* 中的目录已经被生成了。

在默认情况下，来自 *os.scandir()* 的错误将被忽略。如果指定了可选参数 *on_error*，则它应为一个可调用对象；调用它需要传入一个参数，即 *OSError* 的实例。该可调用对象能处理错误以继续执行遍历或是重新引发错误以停止遍历。请注意可以通过异常对象的 `filename` 属性来获取文件名。

在默认情况下，*Path.walk()* 不会跟踪符号链接，而是将其添加到 *filenames* 列表中。将 *follow_symlinks* 设为真值可解析符号链接并根据它们的目标将其放入 *dirnames* 和 *filenames* 中，从而（在受支持的系统上）访问符号链接所指向的目录。

备注

请注意将 *follow_symlinks* 设为真值会在链接指向自身的父目录时导致无限递归。*Path.walk()* 不会跟踪已它访问过的目录。

备注

Path.walk() 会假定在执行过程中它所遍历的目录没有被修改。例如，如果 *dirnames* 中的某个目录已被替换为符号链接并且 *follow_symlinks* 为假值，则 *Path.walk()* 仍会尝试进入该目录。为防止出现这种行为，请相应地移除 *dirnames* 中的目录。

备注

与 *os.walk()* 不同，当 *follow_symlinks* 为假值时 *Path.walk()* 会在 *filenames* 中列出指向目录的符号链接。

这个例子显示每个目录中所有文件使用的总字节数，忽略其中的 `__pycache__` 目录:

```
from pathlib import Path
for root, dirs, files in Path("cpython/Lib/concurrent").walk(on_error=print):
  print(
      root,
      "consumes",
      sum((root / file).stat().st_size for file in files),
      "bytes in",
      len(files),
      "non-directory files"
  )
  if '__pycache__' in dirs:
        dirs.remove('__pycache__')
```

下一个例子是 *shutil.rmtree()* 的一个简单实现。由于 *rmdir()* 不允许在目录为空之前删除该目录因此自下而上地遍历目录树是至关重要的:

```
# 删除可从目录 "top" 进入的所有东西。
# 小心：这很危险！举例来说，如果 top == Path('/'),
# 它可能会删除你的全部文件。
for root, dirs, files in top.walk(top_down=False):
    for name in files:
        (root / name).unlink()
    for name in dirs:
        (root / name).rmdir()
```

Added in version 3.12.

创建文件和目录

Path.**touch**(*mode=0o666*, *exist_ok=True*)

使用给定的路径创建文件。如果给出了 *mode*，它将与进程的 `umask` 值合并以确定文件模式和访问旗标。如果文件已存在，则当 *exist_ok* 为真值时函数将成功执行（并且其修改时间将更新为当前时间），在其他情况下则会引发*FileExistsError*。

> **参见**
>
> *open()*, *write_text()* 和*write_bytes()* 方法经常被用来创建文件。

Path.**mkdir**(*mode=0o777*, *parents=False*, *exist_ok=False*)

使用给定的路径新建目录。如果给出了 *mode*，它将与进程的 `umask` 值合并来决定文件模式和访问旗标。如果路径已存在，则会引发*FileExistsError*。

如果 *parents* 为真值，任何找不到的父目录都会伴随着此路径被创建；它们会以默认权限被创建，而不考虑 *mode* 设置（模仿 POSIX 的 `mkdir -p` 命令）。

如果 *parents* 为假值（默认），则找不到的父级目录会引发*FileNotFoundError*。

如果 *exist_ok* 为 false（默认），则在目标已存在的情况下抛出*FileExistsError*。

如果 *exist_ok* 为真值，则*FileExistsError* 将不会被引发除非给定的路径在文件系统中已存在并且不是目录（与 POSIX `mkdir -p` 命令的行为相同）。

在 3.5 版本发生变更: *exist_ok* 形参被加入。

Path.**symlink_to**(*target*, *target_is_directory=False*)

使该路径成为一个指向 *target* 的符号连接。

在 Windows，符号链接可以代表文件或者目录，并且不会动态适应目标。如果目标存在，则将创建相匹配的符号链接类型。在其他情况下，如果 *target_is_directory* 为真值则符号链接将创建为目录类型否则将创建为文件符号链接。在非 Windows 平台上，*target_is_directory* 将被忽略。

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

> **备注**
>
> 参数的顺序（link, target) 和*os.symlink()* 是相反的。

在 3.13 版本发生变更: 如果*os.symlink()* 不可用则会引发*UnsupportedOperation*。在之前版本中，则是引发*NotImplementedError*。

Path.**hardlink_to**(*target*)

将此路径设为一个指向与 *target* 相同文件的硬链接。

> **备注**
>
> 参数顺序 (link, target) 和*os.link()* 是相反的。

Added in version 3.10.

在 3.13 版本发生变更: 如果*os.link()* 不可用则会引发*UnsupportedOperation*。在之前版本中，则是引发*NotImplementedError*。

重命名与删除

Path.**rename**(*target*)

将此文件或目录重命名为给定的 *target*，并返回一个新的指向 *target* 的 Path 实例。在 Unix 上，如果 *target* 存在且为一个文件，那么如果用户具有相应权限则它将静默地替换。在 Windows 上，如果 *target* 存在，则会引发*FileExistsError*。*target* 可以是一个字符串或者另一个路径对象:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

目标路径可能为绝对或相对路径。相对路径将被解读为相对于当前工作目录，而 不是相对于 Path 对象的目录。

它根据*os.rename()* 实现并给出了同样的保证。

在 3.8 版本发生变更: 添加了返回值，将返回新的 Path 实例。

Path.**replace**(*target*)

将此文件或目录重命名为给定的 *target*，并返回一个新的指向 *target* 的 Path 实例。如果 *target* 指向一个现有文件或空目录，则它将被无条件地替换。

目标路径可能为绝对或相对路径。相对路径将被解读为相对于当前工作目录，而 不是相对于 Path 对象的目录。

在 3.8 版本发生变更: 添加了返回值，将返回新的 Path 实例。

Path.**unlink**(*missing_ok=False*)

移除此文件或符号链接。如果路径指向目录，则用*Path.rmdir()* 代替。

如果 *missing_ok* 为假值（默认），则如果路径不存在将会引发*FileNotFoundError*。

如果 *missing_ok* 为真值，则*FileNotFoundError* 异常将被忽略（和 POSIX rm -f 命令的行为相同）。

在 3.8 版本发生变更: 增加了 *missing_ok* 形参。

Path.**rmdir**()

移除此目录。此目录必须为空的。

访问权限与所有权

Path.**owner**(*, *follow_symlinks=True*)

返拥有此文件的用户名。如果文件的用户标识 (UID) 无法在系统数据库中找到则会引发*KeyError*。

此方法通常会跟随符号链接；要获取符号链接的所有者，请添加参数 follow_symlinks=False。

在 3.13 版本发生变更: 如果*pwd* 模块不可用则会引发*UnsupportedOperation*。在之前的版本中，则是引发*NotImplementedError*。

在 3.13 版本发生变更: 增加了 *follow_symlinks* 形参。

Path.**group**(*, *follow_symlinks=True*)

返回拥有此文件的用户组名。如果文件的用户组标识 (GID) 无法在系统数据库中找到则会引发*KeyError*。

此方法通常会跟随符号链接；要获取符号链接的用户组，请添加参数 follow_symlinks=False。

在 3.13 版本发生变更: 如果*grp* 模块不可用则会引发*UnsupportedOperation*。在较早的版本中，则是引发*NotImplementedError*。

在 3.13 版本发生变更: 增加了 *follow_symlinks* 形参。

Path.**chmod**(*mode*, *, *follow_symlinks=True*)

改变文件模式和权限，和*os.chmod()* 一样。

此方法通常会跟随符号链接。某些 Unix 变种支持改变 symlink 本身的权限；在这些平台上你可以添加参数 `follow_symlinks=False`，或者使用*lchmod()*。

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

在 3.10 版本发生变更: 增加了 *follow_symlinks* 形参。

Path.**lchmod**(*mode*)

就像*Path.chmod()* 但是如果路径指向符号链接则是修改符号链接的模式，而不是修改符号链接的目标。

11.1.5 模式语言

以下通配符在用于*full_match()*, *glob()* 和*rglob()* 的模式中是受支持的：

**** (整个分段)**
匹配任意数量的文件或目录分段，包括零个。

*** (整个分段)**
匹配一个文件或目录分段。

*** (分段的一部分)**
匹配任意数量的非分隔符型字符，包括零个。

?
匹配一个不是分隔符的字符。

[seq]
匹配在 *seq* 中的一个字符。

[!seq]
匹配不在 *seq* 中的一个字符。

对于字面值匹配，请将元字符用方括号括起来。例如，`"[?]"` 将匹配字符 `"?"`。

”`**`” 通配符将启用递归 glob。下面是几个例子：

模式	含意
”`**/*`”	任何具有至少一个分段的路径。
”`**/*.py`”	最后部分以”`.py`” 结尾的任意路径。
”`assets/**`”	以”`assets/`” 开头的任意路径。
”`assets/**/*`”	以”`assets/`” 开头，但不包括”`assets/`” 本身的任意路径。

备注

使用”`**`” 通配符执行 glob 操作将访问目录树中的每个目录。非常大的目录树可能要花费非常长的时间来搜索。

在 3.13 版本发生变更: 使用以"**" 结尾的模式执行 glob 操作将同时返回文件和目录。在之前的版本中，只有目录会被返回。

在*Path.glob()* 和*rglob()* 中，可以向模式添加一个末尾斜杠以只匹配目录。

在 3.11 版本发生变更: 使用以一个路径名称组件分隔符 (*sep* or *altsep*) 结尾的模式执行 glob 操作将只返回目录。

11.1.6 与 glob 模块的比较

Path.glob() 和*Path.rglob()* 所接受的模式和所生成的结果相比*glob* 模式的略有不同:

1. 以点号打点的文件在 pathlib 中没有特殊含义。这类似于向*glob.glob()* 传入 `include_hidden=True`。
2. "**" 模式组件在 pathlib 总是递归的。这类似于向*glob.glob()* 传入 `recursive=True`。
3. "**" 模式组件在 pathlib 中默认不会跟随符号链接。此行为在*glob.glob()* 中没有对应物，但你可以向*Path.glob()* 传入 `recurse_symlinks=True` 以获得兼容的行为。
4. 与所有*PurePath* 和*Path* 对象类似，从*Path.glob()* 和*Path.rglob()* 返回的值都不包括末尾斜杠。
5. 从 pathlib 的 `path.glob()` 和 `path.rglob()` 返回的值包括作为前缀的 *path*，这不同于 `glob.glob(root_dir=path)` 的结果。
6. 从 pathlib 的 `path.glob()` 和 `path.rglob()` 返回的值可能包括 *path* 本身，例如当对"**" 执行 glob 操作的时候，而 `glob.glob(root_dir=path)` 的结果绝不会包括与 *path* 对应的空字符串。

11.1.7 与 os 和 os.path 模块的比较

pathlib 使用*PurePath* 和*Path* 对象来实现路径操作，因此它被认为是 *面向对象的*。而在另一方面，*os* 和*os.path* 模块提供与低层级 `str` 和 `bytes` 对象配合使用的函数，它更接近于 *面向过程的*方式。某些用户认为面向对象的风格可读性更好。

os 和*os.path* 中的许多函数都支持 `bytes` 路径和*相对于目录描述符的路径*。这些特性在 pathlib 中均不可用。

Python 的 `str` 和 `bytes` 类型，以及*os* 和*os.path* 模块的各个部分都是用 C 编写的因而非常快速。pathlib 是用纯 Python 编写的因而往往较慢，但很少会慢到引人注意。

pathlib 的路径规范化比*os.path* 更有主见也更一致。例如，*os.path.abspath()* 会删除路径中的".." 段，因为当涉及符号链接时这可能会改变其含义；而*Path.absolute()* 则会保留这些段以提高安全性。

pathlib 的路径规范化可能使其不适合某些应用程序:

1. pathlib 会将 `Path("my_folder/")` 规范化为 `Path("my_folder")`，这改变了路径在提供给各种操作系统 API 和命令行工具时的含义。具体来说，缺少末尾分隔符可能会使路径被解析为文件或目录，而非只是目录。
2. pathlib 会将 `Path("./my_program")` 规范化为 `Path("my_program")`，这改变了路径在被用作可执行文件搜索路径，例如在 shell 中或者在产生子进程时的含义。具体来说，缺少路径中的分隔符可能会迫使其在 `PATH` 而不是在当前目录中查找。

由于这些差异的影响，pathlib 并不是*os.path* 的直接替代品。

相关工具

以下是一个映射了*os* 与*PurePath*/*Path* 对应相同的函数的表。

os 和*os.path*	*pathlib*
os.path.dirname()	*PurePath.parent*
os.path.basename()	*PurePath.name*
os.path.splitext()	*PurePath.stem*, *PurePath.suffix*

续下页

表 1 – 接上页

os 和 os.path	pathlib
os.path.join()	PurePath.joinpath()
os.path.isabs()	PurePath.is_absolute()
os.path.relpath()	PurePath.relative_to()[1]
os.path.expanduser()	Path.expanduser()[2]
os.path.realpath()	Path.resolve()
os.path.abspath()	Path.absolute()[3]
os.path.exists()	Path.exists()
os.path.isfile()	Path.is_file()
os.path.isdir()	Path.is_dir()
os.path.islink()	Path.is_symlink()
os.path.isjunction()	Path.is_junction()
os.path.ismount()	Path.is_mount()
os.path.samefile()	Path.samefile()
os.getcwd()	Path.cwd()
os.stat()	Path.stat()
os.lstat()	Path.lstat()
os.listdir()	Path.iterdir()
os.walk()	Path.walk()[4]
os.mkdir(), os.makedirs()	Path.mkdir()
os.link()	Path.hardlink_to()
os.symlink()	Path.symlink_to()
os.readlink()	Path.readlink()
os.rename()	Path.rename()
os.replace()	Path.replace()
os.remove(), os.unlink()	Path.unlink()
os.rmdir()	Path.rmdir()
os.chmod()	Path.chmod()
os.lchmod()	Path.lchmod()

备注

11.2 os.path --- 常用的路径操作

源代码: Lib/genericpath.py, Lib/posixpath.py (用于 POSIX) 和 Lib/ntpath.py (用于 Windows)。

此模块实现了一些有用的路径名称相关函数。要读取或写入文件请参见 *open()*，对于访问文件系统请参阅 *os* 模块。传给 path 形参的可以是字符串、字节串或者任何实现了 *os.PathLike* 协议的对象。

与 Unix 不同，Python 不会执行任何 自动路径扩展。当应用程序需要类似 shell 的路径扩展时，可以显式地唤起 *expanduser()* 和 *expandvars()* 这样的函数。(另请参阅 *glob* 模块。)

参见

pathlib 模块提供高级路径对象。

[1] *os.path.relpath()* 会调用 *abspath()* 以使路径变为绝对路径并移除".." 部分，而 *PurePath.relative_to()* 是一个词法操作，当其输入的锚点不同时（例如当一个路径为绝对路径而另一个为相对路径时）将引发 *ValueError*。

[2] 若无法解析家目录，*os.path.expanduser()* 会原封不动地返回输入的路径，但 *Path.expanduser()* 会抛出 *RuntimeError* 异常。

[3] *os.path.abspath()* 会移除".." 组件而不解析符号链接，这可能改变路径的含义，而 *Path.absolute()* 则会让路径中的任何".." 组件保持原样。

[4] *os.walk()* 在将路径分类为 *dirnames* 和 *filenames* 时总是会跟随符号链接，而 *Path.walk()* 当 *follow_symlinks* 为（默认的）假值时会将符号链接分类为 *filenames*。

备注

所有这些函数都仅接受字节或字符串对象作为其参数。如果返回路径或文件名，则结果是相同类型的对象。

备注

由于不同的操作系统具有不同的路径名称约定，因此标准库中有此模块的几个版本。*os.path* 模块始终是适合 Python 运行的操作系统的路径模块，因此可用于本地路径。但是，如果操作的路径总是以一种不同的格式显示，那么也可以分别导入和使用各个模块。它们都具有相同的接口：

- `posixpath` 用于 Unix 样式的路径
- `ntpath` 用于 Windows 路径

在 3.8 版本发生变更: *exists()*、*lexists()*、*isdir()*、*isfile()*、*islink()* 和*ismount()* 现在遇到系统层面上不可表示的字符或字节的路径时，会返回 `False`，而不是抛出异常。

`os.path.`**`abspath`**(*path*)

返回路径 *path* 的绝对路径（标准化的）。在大多数平台上，这等同于用 `normpath(join(os.getcwd(), path))` 的方式调用*normpath()* 函数。

在 3.6 版本发生变更: 接受一个*path-like object*。

`os.path.`**`basename`**(*path*)

返回路径 *path* 的基本名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第二个元素。请注意，此函数的结果与 Unix **basename** 程序不同。**basename** 在 `'/foo/bar/'` 上返回 `'bar'`，而*basename()* 函数返回一个空字符串 (`''`)。

在 3.6 版本发生变更: 接受一个*path-like object*。

`os.path.`**`commonpath`**(*paths*)

返回可迭代对象 *paths* 中每个路径名称的最长共同子路径。如果 *paths* 同时包含绝对和相对路径名称，如果 *paths* 位于不同驱动器，或者如果 *paths* 为空则会引发*ValueError*。不同于*commonprefix()*，此函数将返回一个有效的路径。

Added in version 3.5.

在 3.6 版本发生变更: 接受一个类路径对象 序列。

在 3.13 版本发生变更: 现在可以传入任意可迭代对象，而不只是序列。

`os.path.`**`commonprefix`**(*list*)

接受包含多个路径的 列表，返回所有路径的最长公共前缀（逐字符比较）。如果 列表为空，则返回空字符串 (`''`)。

备注

此函数是逐字符比较，因此可能返回无效路径。要获取有效路径，参见*commonpath()*。

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**dirname**(*path*)

返回路径 *path* 的目录名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第一个元素。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**exists**(*path*)

如果 *path* 指向一个已存在的路径或已打开的文件描述符，返回 True。对于失效的符号链接，返回 False。在某些平台上，如果使用*os.stat()* 查询到目标文件没有执行权限，即使 *path* 确实存在，本函数也可能返回 False。

在 3.3 版本发生变更: *path* 现在可以是一个整数：如果该整数是一个已打开的文件描述符，返回 True，否则返回 False。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**lexists**(*path*)

如果 *path* 指向一个已存在的路径，包括失效的符号链接则返回 True。在缺少*os.lstat()* 的平台上就等价于*exists()*。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**expanduser**(*path*)

在 Unix 和 Windows 上，将参数中开头部分的 ~ 或 ~user 替换为当前 *用户* 的家目录并返回。

在 Unix 上，开头的 ~ 会被环境变量 HOME 代替，如果变量未设置，则通过内置模块*pwd* 在 password 目录中查找当前用户的主目录。以 ~user 开头则直接在 password 目录中查找。

在 Windows 上，如果 USERPROFILE 已设置将会被使用，否则 HOMEPATH 和 HOMEDRIVE 将被组合起来使用。初始的 ~user 会通过检查当前用户的家目录中匹配 USERNAME 的最后一部分目录名并执行替换来处理。

如果展开路径失败，或者路径不是以波浪号开头，则路径将保持不变。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.8 版本发生变更: Windows 不再使用 HOME。

os.path.**expandvars**(*path*)

输入带有环境变量的路径作为参数，返回展开变量以后的路径。$name 或 ${name} 形式的子字符串被环境变量 *name* 的值替换。格式错误的变量名称和对不存在变量的引用保持不变。

在 Windows 上，除了 $name 和 ${name} 外，还可以展开 %name%。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**getatime**(*path*)

返回 *path* 的最后访问时间。返回值是一个浮点数，为纪元秒数（参见*time* 模块）。如果该文件不存在或不可访问，则抛出*OSError* 异常。

os.path.**getmtime**(*path*)

返回 *path* 的最后修改时间。返回值是一个浮点数，为纪元秒数（参见*time* 模块）。如果该文件不存在或不可访问，则抛出*OSError* 异常。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**getctime**(*path*)

返回 *path* 在系统中的 ctime，在有些系统（比如 Unix）上，它是元数据的最后修改时间，其他系统（比如 Windows）上，它是 *path* 的创建时间。返回值是一个数，为纪元秒数（参见*time* 模块）。如果该文件不存在或不可访问，则抛出*OSError* 异常。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**getsize**(*path*)

返回 *path* 的大小，以字节为单位。如果该文件不存在或不可访问，则抛出*OSError* 异常。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**isabs**(*path*)

如果 *path* 为绝对路径则返回 `True`。在 Unix 上，意味着它是以斜杠打头的，在 Windows 上它是以两个（反）斜杠，或是由驱动器号、冒号和（反斜杠）连在一起打头的。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.13 版本发生变更: 在 Windows 上，如果给定的路径是以一个单独（反）斜杠打头则返回 `False`。

os.path.**isfile**(*path*)

如果 *path* 是现有的 常规文件，则返回 `True`。本方法会跟踪符号链接，因此，对于同一路径，*islink()* 和 *isfile()* 都可能为 `True`。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**isdir**(*path*)

如果 *path* 是现有的 目录，则返回 `True`。本方法会跟踪符号链接，因此，对于同一路径，*islink()* 和 *isdir()* 都可能为 `True`。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**isjunction**(*path*)

Return `True` 如果 *path* 指向的现有 目录条目是一个连接点。则当连接点在当前平台不受支持时将总是返回 `False`。

Added in version 3.12.

os.path.**islink**(*path*)

如果 *path* 指向的现有 目录条目是一个符号链接，则返回 `True`。如果 Python 运行时不支持符号链接，则总是返回 `False`。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**ismount**(*path*)

如果路径 *path* 是 挂载点（文件系统中挂载其他文件系统的点），则返回 `True`。在 POSIX 上，该函数检查 *path* 的父目录 *path*`/..` 是否在与 *path* 不同的设备上，或者 *path*`/..` 和 *path* 是否指向同一设备上的同一 inode（这一检测挂载点的方法适用于所有 Unix 和 POSIX 变体）。本方法不能可靠地检测同一文件系统上的绑定挂载 (bind mount)。在 Windows 上，盘符和共享 UNC 始终是挂载点，对于任何其他路径，将调用 `GetVolumePathName` 来查看它是否与输入的路径不同。

在 3.4 版本发生变更: 增加了在 Windows 上检测非根挂载点的支持。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**isdevdrive**(*path*)

如果路径名 *path* 位于一个 Windows Dev 驱动器则返回 `True`。Dev Drive 针对开发者场景进行了优化，并为读写文件提供更快的性能。推荐用于源代码、临时构建目录、包缓存以及其他的 IO 密集型操作。

对于无效的路径可能引发错误，例如，没有可识别的驱动器的路径，但在不支持 Dev 驱动器的平台上将返回 `False`。请参阅 Windows 文档 了解有关启用并创建 Dev 驱动器的信息。

Added in version 3.12.

在 3.13 版本发生变更: 现在此函数在所有平台上可用，并且在不支持 Dev 驱动器的平台上将总是返回 `False`。

os.path.**isreserved**(*path*)

如果 *path* 为当前系统的保留路径名则返回 `True`。

在 Windows 上，保留文件名包括以一个空格或点号结尾的；包含冒号的 (即文件流如”name:stream”)，通配符 (即 `'*?"<>'`)，管道或 ASCII 控制符；以及 DOS 设备名称如”NUL”, ”CON”, ”CONIN$”, ”CONOUT$”, ”AUX”, ”PRN”, ”COM1” 和”LPT1”。

> **备注**
>
> 此函数在多数 Windows 系统上使用相似的保留路径规则。这些规则会随着时间的推移在不同的 Windows 发布版中发生改变。此函数可能会随着规则的更改广泛可用在未来的 Python 发布版中被更新。

Availability: Windows.

Added in version 3.13.

os.path.**join**(*path*, **paths*)

智能地合并一个或多个路径部分。返回值将是 *path* 和所有 **paths* 成员的拼接，其中每个非空部分后面都紧跟一个目录分隔符，最后一个除外。也就是说，如果最后一个部分为空或是以一个分隔符结束则结果将仅以一个分隔符结束。如果某个部分为绝对路径（在 Windows 上需要同时有驱动器号和根路径符号），则之前的所有部分会被忽略并从该绝对路径部分开始拼接。

在 Windows，当遇到绝对路径部分 (如 `r'\foo'`) 时驱动器号将不会被重置。如果某个部分位于不同驱动器或为绝对路径，则之前的所有部分会被忽略并且该驱动器号会被重置。请注意由于每个驱动器都有一个当前目录，因此 `os.path.join("c:", "foo")` 是代表驱动器 `C:` 上当前路径的相对路径 (`c:foo`)，而不是 `c:\foo`。

在 3.6 版本发生变更: 接受一个类路径对象 用于 *path* 和 *paths* 。

os.path.**normcase**(*path*)

规范路径的大小写。在 Windows 上，将路径中的所有字符都转换为小写，并将正斜杠转换为反斜杠。在其他操作系统上返回原路径。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**normpath**(*path*)

通过折叠多余的分隔符和对上级目录的引用来标准化路径名，所以 `A//B`、`A/B/`、`A/./B` 和 `A/foo/../B` 都会转换成 `A/B`。这个字符串操作可能会改变带有符号链接的路径的含义。在 Windows 上，本方法将正斜杠转换为反斜杠。要规范大小写，请使用*`normcase()`*。

> **备注**
>
> 在 POSIX 系统上，根据 IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution，如果一个路径名称以两个斜杠开始，则开始字符之后的第一个部分将以具体实现所定义的方式来解读，但是超过两个开始字符则将被视为单个字符。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**realpath**(*path*, *, *strict=False*)

返回指定文件名的规范路径，去除在路径中出现的任何符号链接（如果操作系统支持的话）。在 Windows 上，此函数也会将 MS-DOS (或称 8.3) 风格的名称如 `C:\\PROGRA~1` 解析为 `C:\\Program Files`。

如果一个路径不存在或是遇到了符号链接循环，并且 *strict* 为 `True`，则会引发*`OSError`*。如果 *strict* 为 `False` 则这些错误将被忽略，因此结果可能缺失或无法访问。

> **备注**
>
> 这个函数会模拟操作系统生成规范路径的过程，Windows 与 UNIX 的这个过程在处理链接和后续路径组成部分的交互方式上有所差异。
>
> 操作系统 API 会根据需要来规范化路径，因此通常不需要调用此函数。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.8 版本发生变更: 在 Windows 上现在可以正确解析符号链接和交接点 (junction point)。

在 3.10 版本发生变更: 增加了 *strict* 形参。

os.path.**relpath**(*path*, *start=os.curdir*)

返回从当前目录或可选的 *start* 目录至 *path* 的相对文件路径。这只是一个路径计算：不会访问文件系统来确认 *path* 或 *start* 是否存在或其性质。在 Windows 上，当 *path* 和 *start* 位于不同驱动器时将引发 `ValueError`。

start 默认为 `os.curdir`。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**samefile**(*path1*, *path2*)

如果两个路径都指向相同的文件或目录，则返回 `True`。这由设备号和 inode 号确定，在任一路径上调用 `os.stat()` 失败则抛出异常。

在 3.2 版本发生变更: 添加了对 Windows 的支持。

在 3.4 版本发生变更: Windows 现在使用与其他所有平台相同的实现。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**sameopenfile**(*fp1*, *fp2*)

如果文件描述符 *fp1* 和 *fp2* 指向相同文件，则返回 `True`。

在 3.2 版本发生变更: 添加了对 Windows 的支持。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**samestat**(*stat1*, *stat2*)

如果 stat 元组 *stat1* 和 *stat2* 指向相同文件，则返回 `True`。这些 stat 元组可能是由 `os.fstat()`、`os.lstat()` 或 `os.stat()` 返回的。本函数实现了 `samefile()` 和 `sameopenfile()` 底层所使用的比较过程。

在 3.4 版本发生变更: 添加了对 Windows 的支持。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**split**(*path*)

将路径 *path* 拆分为一对，即 `(head, tail)`，其中，*tail* 是路径的最后一部分，而 *head* 里是除最后部分外的所有内容。*tail* 部分不会包含斜杠，如果 *path* 以斜杠结尾，则 *tail* 将为空。如果 *path* 中没有斜杠，*head* 将为空。如果 *path* 为空，则 *head* 和 *tail* 均为空。*head* 末尾的斜杠会被去掉，除非它是根目录（即它仅包含一个或多个斜杠）。在所有情况下，`join(head, tail)` 指向的位置都与 *path* 相同（但字符串可能不同）。另请参见函数 `dirname()` 和 `basename()`。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**splitdrive**(*path*)

将路径 *path* 拆分为一对，即 `(drive, tail)`，其中 *drive* 是挂载点或空字符串。在没有驱动器概念的系统上，*drive* 将始终为空字符串。在所有情况下，`drive + tail` 都与 *path* 相同。

在 Windows 上，本方法将路径拆分为驱动器/UNC 根节点和相对路径。

如果路径 path 包含盘符，则 drive 将包含冒号之前的所有内容包括冒号本身:

```
>>> splitdrive("c:/dir")
("c:", "/dir")
```

如果路径包含 UNC 路径，则 drive 将包含主机名和 share:

```
>>> splitdrive("//host/computer/dir")
("//host/computer", "/dir")
```

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**splitroot**(*path*)

将路径名 *path* 拆分为一个 3 元组 `(drive, root, tail)` 其中 *drive* 是设置名或挂载点，*root* 是表示 drive 之后的分隔符的字符串，而 *tail* 则为 root 之后的所有内容。这此条目均可以为空字符串。在所有情况下，`drive + root + tail` 都与 *path* 相同。

在 POSIX 系统上，*drive* 将总是为空。*root* 可能为空（如果 *path* 是相对路径）、单个正斜杠（如果 *path* 是绝对路径）、或两个正斜杠（由基于 IEEE Std 1003.1-2017; 4.13 Pathname Resolution 的具体实现来定义。）例如:

```
>>> splitroot('/home/sam')
('', '/', 'home/sam')
>>> splitroot('//home/sam')
('', '//', 'home/sam')
>>> splitroot('///home/sam')
('', '/', '//home/sam')
```

在 Windows 上，*drive* 可能为空、以字母表示的驱动器名称、UNC share 或是设备名称。*root* 可能为空、单个正斜杠，或单个反斜杠。例如:

```
>>> splitroot('C:/Users/Sam')
('C:', '/', 'Users/Sam')
>>> splitroot('//Server/Share/Users/Sam')
('//Server/Share', '/', 'Users/Sam')
```

Added in version 3.12.

os.path.**splitext**(*path*)

将路径名称 *path* 拆分为 `(root, ext)` 对使得 `root + ext == path`，并且扩展名 *ext* 为空或以句点打头并最多只包含一个句点。

如果路径 path 不包含扩展名，则 *ext* 将为 `''`:

```
>>> splitext('bar')
('bar', '')
```

如果路径 path 包含扩展名，则 *ext* 将被设为该扩展名，包括打头的句点。请注意在其之前的句点将被忽略:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

path 中最后一部分如果以点号开头则会被视为 root 的一部分:

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

在 3.6 版本发生变更: 接受一个*path-like object*。

os.path.**supports_unicode_filenames**

如果（在文件系统限制下）允许将任意 Unicode 字符串用作文件名，则为 `True`。

11.3 `stat` --- 解释 `stat()` 的结果

源代码： Lib/stat.py

`stat` 模块定义了一些用于解读*`os.stat()`*, *`os.fstat()`* 和*`os.lstat()`* (如果它们存在) 输出结果的常量和函数。有关 `stat()`, `fstat()` 和 `lstat()` 调用的完整细节，请参阅你的系统文档。

在 3.4 版本发生变更: stat 模块是通过 C 实现来支持的。

stat 模块定义了以下函数来检测特定文件类型:

stat.**S_ISDIR**(*mode*)
如果 mode 来自一个目录则返回非零值。

stat.**S_ISCHR**(*mode*)
如果 mode 来自一个字符特殊设备文件则返回非零值。

stat.**S_ISBLK**(*mode*)
如果 mode 来自一个块特殊设备文件则返回非零值。

stat.**S_ISREG**(*mode*)
如果 mode 来自一个常规文件则返回非零值。

stat.**S_ISFIFO**(*mode*)
如果 mode 来自一个 FIFO (命名管道) 则返回非零值。

stat.**S_ISLNK**(*mode*)
如果 mode 来自一个符号链接则返回非零值。

stat.**S_ISSOCK**(*mode*)
如果 mode 来自一个套接字则返回非零值。

stat.**S_ISDOOR**(*mode*)
如果 mode 来自一个门则返回非零值。

Added in version 3.4.

stat.**S_ISPORT**(*mode*)
如果 mode 来自一个事件端口则返回非零值。

Added in version 3.4.

stat.**S_ISWHT**(*mode*)
如果 mode 来自一个白输出则返回非零值。

Added in version 3.4.

定义了两个附加函数用于对文件模式进行更一般化的操作:

stat.**S_IMODE**(*mode*)
返回文件模式中可由*os.chmod()* 进行设置的部分 --- 即文件的 permission 位，加上 sticky 位、set-group-id 以及 set-user-id 位（在支持这些部分的系统上）。

stat.**S_IFMT**(*mode*)
返回文件模式中描述文件类型的部分（供上面的 S_IS*() 函数使用）。

通常，你将使用 os.path.is*() 函数来检测文件的类型；这里提供的函数在你要对同一文件执行多项检测并且希望避免每项检测的 stat() 系统调用的开销时会很有用。这些函数也适用于检测有关未被*os.path* 处理的信息，如检测块和字符设备等。

示例:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''在根位于顶部的目录树中递归地下行，
       为每个常规文件调用回调函数'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.lstat(pathname).st_mode
```

(续下页)

(接上页)

```
        if S_ISDIR(mode):
            # 是个目录，递归进去
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # 是个文件，调用回调函数
            callback(pathname)
        else:
            # 未知文件类型，打印一条消息
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

另外还提供了一个附加的辅助函数用来将文件模式转换为人类易读的字符串：

stat.**filemode**(*mode*)

将文件模式转换为'-rwxrwxrwx' 形式的字符串。

Added in version 3.3.

在 3.4 版本发生变更: 此函数支持*S_IFDOOR*, *S_IFPORT* and *S_IFWHT*。

以下所有变量是一些简单的符号索引，用于访问*os.stat()*, *os.fstat()* 或*os.lstat()* 所返回的 10 条目元组。

stat.**ST_MODE**

inode 保护模式。

stat.**ST_INO**

Inode 号

stat.**ST_DEV**

Inode 所在的设备。

stat.**ST_NLINK**

Inode 拥有的链接数量。

stat.**ST_UID**

所有者的用户 ID。

stat.**ST_GID**

所有者的用户组 ID。

stat.**ST_SIZE**

以字节为单位的普通文件大小；对于某些特殊文件则是所等待的数据量。

stat.**ST_ATIME**

上次访问的时间。

stat.**ST_MTIME**

上次修改的时间。

stat.**ST_CTIME**

操作系统所报告的"ctime"。在某些系统上（例如 Unix）是元数据的最后修改时间，而在其他系统上（例如 Windows）则是创建时间（请参阅系统平台的文档了解相关细节）。

对于"文件大小"的解析可因文件类型的不同而变化。对于普通文件就是文件的字节数。对于大部分种类的 Unix（特别包括 Linux）的 FIFO 和套接字来说，"大小"则是指在调用*os.stat()*, *os.fstat()* 或*os.lstat()* 时等待读取的字节数；这在某些时候很有用处，特别是在一个非阻塞的打开后轮询这些

特殊文件中的一个时。其他字符和块设备的文件大小字段的含义还会有更多变化，具体取决于底层系统调用的实现方式。

以下变量定义了在*ST_MODE* 字段中使用的旗标。

使用上面的函数会比使用第一组旗标更容易移植：

stat.**S_IFSOCK**
套接字。

stat.**S_IFLNK**
符号链接。

stat.**S_IFREG**
普通文件。

stat.**S_IFBLK**
块设备。

stat.**S_IFDIR**
目录。

stat.**S_IFCHR**
字符设备。

stat.**S_IFIFO**
先进先出。

stat.**S_IFDOOR**
门。

Added in version 3.4.

stat.**S_IFPORT**
事件端口。

Added in version 3.4.

stat.**S_IFWHT**
白输出。

Added in version 3.4.

备注

S_IFDOOR, *S_IFPORT* or *S_IFWHT* 等文件类型在不受系统平台支持时会被定义为 0。

以下旗标还可以*os.chmod()* 的在 *mode* 参数中使用：

stat.**S_ISUID**
设置 UID 位。

stat.**S_ISGID**
设置分组 ID 位。这个位有几种特殊用途。对于目录它表示该目录将使用 BSD 语义：在其中创建的文件将从目录继承其分组 ID，而不是从创建进程的有效分组 ID 继承，并且在其中创建的目录也将设置*S_ISGID* 位。对于没有设置分组执行位 (*S_IXGRP*) 的文件，设置分组 ID 位表示强制性文件/记录锁定 (另请参见*S_ENFMT*)。

stat.**S_ISVTX**
固定位。当对目录设置该位时则意味着此目录中的文件只能由文件所有者、目录所有者或特权进程来重命名或删除。

stat.**S_IRWXU**
文件所有者权限的掩码。

stat.**S_IRUSR**
所有者具有读取权限。

stat.**S_IWUSR**
所有者具有写入权限。

stat.**S_IXUSR**
所有者具有执行权限。

stat.**S_IRWXG**
组权限的掩码。

stat.**S_IRGRP**
组具有读取权限。

stat.**S_IWGRP**
组具有写入权限。

stat.**S_IXGRP**
组具有执行权限。

stat.**S_IRWXO**
其他人（不在组中）的权限掩码。

stat.**S_IROTH**
其他人具有读取权限。

stat.**S_IWOTH**
其他人具有写入权限。

stat.**S_IXOTH**
其他人具有执行权限。

stat.**S_ENFMT**
System V 执行文件锁定。此旗标是与*S_ISGID* 共享的：文件/记录锁定会针对未设置分组执行位 (*S_IXGRP*) 的文件强制执行。

stat.**S_IREAD**
Unix V7 中*S_IRUSR* 的同义词。

stat.**S_IWRITE**
Unix V7 中*S_IWUSR* 的同义词。

stat.**S_IEXEC**
Unix V7 中*S_IXUSR* 的同义词。

以下旗标可以在*os.chflags()* 的 *flags* 参数中使用：

stat.**UF_SETTABLE**
所有用户可设置的旗标。

Added in version 3.13.

stat.**UF_NODUMP**
不要转储文件。

stat.**UF_IMMUTABLE**
文件不能被更改。

stat.**UF_APPEND**
文件只能被附加。

stat.**UF_OPAQUE**
当通过联合堆栈查看时，目录是不透明的。

stat.**UF_NOUNLINK**
文件不能重命名或删除。

stat.**UF_COMPRESSED**
文件是压缩存储的（macOS 10.6+）。

stat.**UF_TRACKED**
用于处理文档 ID (macOS)

Added in version 3.13.

stat.**UF_DATAVAULT**
文件需要赋予读取或写入权限 (macOS 10.13+)

Added in version 3.13.

stat.**UF_HIDDEN**
文件不可被显示在 GUI 中（macOS 10.5+）。

stat.**SF_SETTABLE**
所有超级用户可修改的旗标

Added in version 3.13.

stat.**SF_SUPPORTED**
所有超级用户支持的旗标

Availability: macOS

Added in version 3.13.

stat.**SF_SYNTHETIC**
所有超级用户只读的合成旗标

Availability: macOS

Added in version 3.13.

stat.**SF_ARCHIVED**
文件可能已存档。

stat.**SF_IMMUTABLE**
文件不能被更改。

stat.**SF_APPEND**
文件只能被附加。

stat.**SF_RESTRICTED**
文件需要赋予写入权限 (macOS 10.13+)

Added in version 3.13.

stat.**SF_NOUNLINK**
文件不能重命名或删除。

stat.**SF_SNAPSHOT**
文件有一个快照文件

stat.**SF_FIRMLINK**
文件是一个固定链接 (macOS 10.15+)

Added in version 3.13.

stat.**SF_DATALESS**

文件是一个无数据对象 (macOS 10.15+)

Added in version 3.13.

请参阅 *BSD 或 macOS 系统的指南页 *chflags(2)* 来了解详情。

在 Windows 上，以下文件属性常量可被用来检测*os.stat()* 所返回的 st_file_attributes 成员中的位。请参阅 Windows API 文档 了解有关这些常量含义的详情。

stat.**FILE_ATTRIBUTE_ARCHIVE**

stat.**FILE_ATTRIBUTE_COMPRESSED**

stat.**FILE_ATTRIBUTE_DEVICE**

stat.**FILE_ATTRIBUTE_DIRECTORY**

stat.**FILE_ATTRIBUTE_ENCRYPTED**

stat.**FILE_ATTRIBUTE_HIDDEN**

stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**

stat.**FILE_ATTRIBUTE_NORMAL**

stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**

stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**

stat.**FILE_ATTRIBUTE_OFFLINE**

stat.**FILE_ATTRIBUTE_READONLY**

stat.**FILE_ATTRIBUTE_REPARSE_POINT**

stat.**FILE_ATTRIBUTE_SPARSE_FILE**

stat.**FILE_ATTRIBUTE_SYSTEM**

stat.**FILE_ATTRIBUTE_TEMPORARY**

stat.**FILE_ATTRIBUTE_VIRTUAL**

Added in version 3.5.

在 Windows 上，以下常量可被用来与*os.lstat()* 所返回的 st_reparse_tag 成员进行比较。这些是最主要的常量，而不是详尽的清单。

stat.**IO_REPARSE_TAG_SYMLINK**

stat.**IO_REPARSE_TAG_MOUNT_POINT**

stat.**IO_REPARSE_TAG_APPEXECLINK**

Added in version 3.8.

11.4 filecmp --- 文件和目录比较

源代码: Lib/filecmp.py

---

filecmp 模块定义了用于比较文件及目录的函数，并且可以选取多种关于时间和准确性的折衷方案。对于文件的比较，另见*difflib* 模块。

filecmp 模块定义了如下函数：

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)

比较名为 *f1* 和 *f2* 的文件，如果它们似乎相等则返回 True ，否则返回 False 。

如果 *shallow* 为真值且两个文件的*os.stat()* 签名信息（文件类型、大小和修改时间）一致，则文件会被视为相同。

在其他情况下，如果文件大小或内容不同则它们会被视为不同。

需要注意，没有外部程序被该函数调用，这赋予了该函数可移植性与效率。

该函数会缓存过去的比较及其结果，且在文件的*os.stat()* 信息变化后缓存条目失效。所有的缓存可以通过使用*clear_cache()* 来清除。

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)

比较在两个目录 *dir1* 和 *dir2* 中，由 *common* 所确定名称的文件。

返回三组文件名列表：*match*, *mismatch*, *errors* 。*match* 含有相匹配的文件，*mismatch* 含有那些不匹配的，然后 *errors* 列出那些未被比较文件的名称。如果文件不存在于两目录中的任一个，或者用户缺少读取它们的权限，又或者因为其他的一些原因而无法比较，那么这些文件将会被列在 *errors* 中。

参数 *shallow* 具有同*filecmp.cmp()* 一致的含义与默认值。

例如，`cmpfiles('a', 'b', ['c', 'd/e'])` 将会比较 `a/c` 与 `b/c` 以及 `a/d/e` 与 `b/d/e` 。`'c'` 和 `'d/e'` 将会各自出现在返回的三个列表里的某一个列表中。

filecmp.**clear_cache**()

清除 filecmp 缓存。如果一个文件过快地修改，以至于超过底层文件系统记录修改时间的精度，那么该函数可能有助于比较该类文件。

Added in version 3.4.

11.4.1 dircmp 类

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*, *, *shallow=True*)

构造一个新的目录比较对象，用来比较目录 *a* 和 *b*。*ignore* 是要忽略的名称列表，且默认为*filecmp.DEFAULT_IGNORES*。*hide* 是要隐藏的名称列表，且默认为 `[os.curdir, os.pardir]`。

dircmp 类如*filecmp.cmp()* 所描述的那样默认使用 *shallow* 形参通过执行 *shallow* 比较来比较文件。

在 3.13 版本发生变更: 增加了 *shallow* 形参。

dircmp 类提供以下方法：

report()

将 *a* 与 *b* 之间的比较结果打印（到*sys.stdout* ）。

report_partial_closure()

打印 *a* 与 *b* 及共同直接子目录的比较结果。

report_full_closure()

打印 *a* 与 *b* 及共同子目录比较结果（递归地）。

dircmp 类提供了一些有趣的属性，用以得到关于参与比较的目录树的各种信息。

请注意通过 `__getattr__()` 钩子，所有的属性都将被惰性求值，因此如果只需使用那些计算简便的属性就不会有速度上的损失。

left

目录 *a* 。

right

目录 *b* 。

left_list

经 *hide* 和 *ignore* 过滤，目录 *a* 中的文件与子目录。

right_list

经 *hide* 和 *ignore* 过滤，目录 *b* 中的文件与子目录。

common

同时存在于目录 *a* 和 *b* 中的文件和子目录。

left_only

仅在目录 *a* 中的文件和子目录。

right_only

仅在目录 *b* 中的文件和子目录。

common_dirs

同时存在于目录 *a* 和 *b* 中的子目录。

common_files

同时存在于目录 *a* 和 *b* 中的文件。

common_funny

在目录 *a* 和 *b* 中类型不同的名字，或者那些*os.stat()* 报告错误的名字。

same_files

在目录 *a* 和 *b* 中，使用类的文件比较操作符判定相等的文件。

diff_files

在目录 *a* 和 *b* 中，根据类的文件比较操作符判定内容不等的文件。

funny_files

在目录 *a* 和 *b* 中无法比较的文件。

subdirs

一个将*common_dirs* 中的名称映射到*dircmp* 实例（或者 MyDirCmp 实例，如果该实例类型为*dircmp* 的子类 MyDirCmp 的话）的字典。

在 3.10 版本发生变更: 在之前版本中字典条目总是为*dircmp* 实例。现在条目将与 *self* 的类型相同，如果 *self* 为*dircmp* 的子类的话。

filecmp.**DEFAULT_IGNORES**

Added in version 3.4.

默认被*dircmp* 忽略的目录列表。

下面是一个简单的例子，使用 subdirs 属性递归搜索两个目录以显示公共差异文件:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.5 tempfile --- 生成临时文件和目录

源代码： Lib/tempfile.py

该模块可以创建临时文件和目录。它适用于所有受支持的平台。*TemporaryFile*, *NamedTemporaryFile*, *TemporaryDirectory* 和*SpooledTemporaryFile* 是提供自动清理功能的高层级接口并可用作上下文管理器。*mkstemp()* 和*mkdtemp()* 是需要执行手动清理的低层级函数。

所有由用户调用的函数和构造函数都带有参数，这些参数可以设置临时文件和临时目录的路径和名称。该模块生成的文件名包括一串随机字符，在公共的临时目录中，这些字符可以让创建文件更加安全。为了保持向后兼容性，参数的顺序有些奇怪。所以为了代码清晰，建议使用关键字参数。

这个模块定义了以下内容供用户调用:

tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

返回一个*file-like object* 作为临时存储区域。创建该文件使用了与`mkstemp()` 相同的安全规则。它将在关闭后立即销毁（包括垃圾回收机制关闭该对象时）。在 Unix 下，该文件在目录中的条目根本不创建，或者创建文件后立即就被删除了，其他平台不支持此功能。您的代码不应依赖使用此功能创建的临时文件名称，因为它在文件系统中的名称可能是可见的，也可能是不可见的。

结果对象可以用作*context manager* (参见例子)。上下文结束或文件对象销毁后会将临时文件从文件系统中移除。

mode 参数默认值为 `'w+b'`，所以创建的文件不用关闭，就可以读取或写入。因为用的是二进制模式，所以无论存的是什么数据，它在所有平台上都表现一致。*buffering*、*encoding*、*errors* 和 *newline* 的含义与`open()` 中的相同。

参数 *dir*、*prefix* 和 *suffix* 的含义和默认值都与它们在`mkstemp()` 中的相同。

在 POSIX 平台上，它返回的对象是真实的文件对象。在其他平台上，它是一个文件型对象，它的 `file` 属性是底层的真实文件对象。

在可用且有效时将使用`os.O_TMPFILE` 旗标（Linux 专属，需要 Linux 内核版本为 3.11 或更高）。

在 Posix 或 Cygwin 以外的平台上，TemporaryFile 是 NamedTemporaryFile 的别名。

引发一个审计事件 `tempfile.mkstemp` 并附带参数 `fullpath`。

在 3.5 版本发生变更: 在可以时现在将使用`os.O_TMPFILE` 旗标。

在 3.8 版本发生变更: 添加了 *errors* 参数。

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*, *delete_on_close=True*)

此函数的操作与`TemporaryFile()` 所做的完全相同，除了存在下列差异:

- 此函数将返回一个肯定具有在文件系统中的可见名称的文件。
- 为管理指定名称的文件，它将为`TemporaryFile()` 扩展 *delete* 和 *delete_on_close* 形参来确定指定名称的文件是否要被自动删除以及要如何执行删除。

返回的对象将总是一个*file-like object* 并且其 `file` 属性为底层的实际文件对象。这个文件型对象可在 `with` 语句中使用，就像普通的文件一样。该临时文件的文件名可从被返回的文件型对象的 `name` 属性中提取。在 Unix 上，不同于`TemporaryFile()`，其目录项不会在创建文件之后立即被取消链接。

如果 *delete* 为（默认的）真值且 *delete_on_close* 也为（默认的）真值，则文件将在关闭后立即被删除。如果 *delete* 为真值而 *delete_on_close* 为假值，则文件将在退出上下文管理器，或者当*file-like object* 被终结时才会被删除。在此情况下将不保证总是能删除文件（参见 `object.__del__()` 文档）。如果 *delete* 为假值，则 *delete_on_close* 的值将被忽略。

因此要使用该临时文件的名称在关闭文件之后重新打开它，那么注意在关闭时不要删除文件（将 *delete* 形参设为假值），或者如果该临时文件是在 `with` 语句中创建的，则要将 *delete_on_close* 形参设为假值。更推荐后一种方式因为它在上下文管理器退出时提供了自动清理协助。

临时文件仍然打开时使用其名称再次打开它的操作如下所示:

- 在 POSIX 上该文件总是可以被再次打开。the file can always be opened again.
- 在 Windows 上，要确保至少满足下列条件之一:
  - *delete* 为假值
  - 额外的打开将共享删除操作（例如调用`os.open()` 时附带了 `O_TEMPORARY` 旗标）
  - *delete* 为真值但 *delete_on_close* 为假值。注意，在此情况下没有共享删除操作的额外的打开（例如通过内置的`open()` 创建）必须在退出上下文管理器之前被关闭，否则在退出上下文管理器时的`os.unlink()` 调用将失败并引发`PermissionError`。

在 Windows 上，如果 *delete_on_close* 为假值，并且文件是在用户没有删除权限的目录中创建的，则退出上下文管理器时的`os.unlink()` 调用将失败并引发`PermissionError`。这在 *delete_on_close* 为真值时不会发生因为删除权限是由打开操作所请求的，如果未获得所请求的权限此操作将立即失败。

（只有）在 POSIX 上，一个用 SIGKILL 突然终止的进程无法自动删除它所创建的任何 NamedTemporaryFiles。

引发一个审计事件 tempfile.mkstemp 并附带参数 fullpath。

在 3.8 版本发生变更: 添加了 *errors* 参数。

在 3.12 版本发生变更: 增加了 *delete_on_close* 形参。

class tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

这个类执行的操作与*TemporaryFile()* 完全相同，但会将数据放入内存池直到文件大小超过 *max_size*，或者直到文件的*fileno()* 方法被调用，这时文件内容会被写入磁盘并如使用*TemporaryFile()* 时一样执行后续操作。

rollover()

结果文件有一个额外的方法 rollover()，它可以忽略文件大小将其立即写入到磁盘文件。

返回的对象是一个文件型对象，它的 _file 属性是*io.BytesIO* 或*io.TextIOWrapper* 对象（这取决于所指定的 *mode* 是二进制还是文本）或真实的文件对象，这取决于*rollover()* 是否已被调用。这个文件型对象可以像普通文件一样在 with 语句中使用。

在 3.3 版本发生变更: 现在 truncate 方法可接受一个 *size* 参数。

在 3.8 版本发生变更: 添加了 *errors* 参数。

在 3.11 版本发生变更: 完整实现*io.BufferedIOBase* 和*io.TextIOBase* 抽象基类（取决于二进制或文本 *mode* 是否已指定）。

class tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*, *ignore_cleanup_errors=False*, ***, *delete=True*)

这个类会使用与*mkdtemp()* 相同的规则安全地创建一个临时目录。结果对象可以被用作*context manager* (参见例子)。在完成上下文或销毁临时目录对象时，新创建的临时目录及其所有内容会从文件系统中被移除。

name

可以从所返回对象的 name 属性中提取目录名称。当返回的对象被用作*context manager* 时，这个 name 将被作为 with 语句中 as 子句的目标，如果存在该子句的话。

cleanup()

此目录可通过调用 cleanup() 方法来显式地清理。如果 *ignore_cleanup_errors* 为真值，则在显式或隐式清理（例如在 Windows 上*PermissionError* 移除打开的文件）期间出现的未处理异常将被忽略，并且剩余的可移除条目会被“尽可能”地删除。在其他情况下，错误将在任何上下文清理发生时被引发（如 cleanup() 调用，退出上下文管理器、对象被作为垃圾回收或解释器关闭等情况）。

delete 可被用于禁止在退出上下文时清理目录树。虽然在退出上下文时禁止此操作看起来可能很不常见，但这在进行调试或在你的清理行为需要以其他逻辑为条件时将会很有用处。

引发一个审计事件 tempfile.mkdtemp 并附带参数 fullpath。

Added in version 3.2.

在 3.10 版本发生变更: 添加了 *ignore_cleanup_errors* 形参。

在 3.12 版本发生变更: 增加了 *delete* 形参。

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

以最安全的方式创建一个临时文件。假设所在平台正确实现了*os.open()* 的*os.O_EXCL* 标志，则创建文件时不会有竞争的情况。该文件只能由创建者读写，如果所在平台用权限位来标记文件是否可执行，那么没有人有执行权。文件描述符不会过继给子进程。

与*TemporaryFile()* 不同，*mkstemp()* 用户用完临时文件后需要自行将其删除。

如果 *suffix* 不是 None 则文件名将以该后缀结尾，是 None 则没有后缀。*mkstemp()* 不会在文件名和后缀之间加点，如果需要加一个点号，请将其放在 *suffix* 的开头。

如果 *prefix* 不是 None，则文件名将以该前缀开头，是 None 则使用默认前缀。默认前缀是*gettempprefix()* 或*gettempprefixb()* 函数的返回值（自动调用合适的函数）。

如果 *dir* 不为 None，则在指定的目录创建文件，是 None 则使用默认目录。默认目录是从一个列表中选择出来的，这个列表不同平台不一样，但是用户可以设置 *TMPDIR*、*TEMP* 或 *TMP* 环境变量来设置目录的位置。因此，不能保证生成的临时文件路径很规范，比如，通过 os.popen() 将路径传递给外部命令时仍需要加引号。

如果 *suffix*、*prefix* 和 *dir* 中的任何一个不是 None，就要保证它们是同一数据类型。如果它们是 bytes，则返回的名称的类型就是 bytes 而不是 str。如果确实要用默认参数，但又想要返回值是 bytes 类型，请传入 suffix=b''。

如果指定了 *text* 且为真值，文件会以文本模式打开。否则，文件（默认）会以二进制模式打开。

mkstemp() 返回一个元组，元组中第一个元素是句柄，它是一个系统级句柄，指向一个打开的文件（等同于*os.open()* 的返回值），第二元素是该文件的绝对路径。

引发一个审计事件 tempfile.mkstemp 并附带参数 fullpath。

在 3.5 版本发生变更: 现在，*suffix*、*prefix* 和 *dir* 可以以 bytes 类型按顺序提供，以获得 bytes 类型的返回值。之前只允许使用 str。*suffix* 和 *prefix* 现在可以接受 None，并且默认为 None 以使用合适的默认值。

在 3.6 版本发生变更: *dir* 参数现在可接受一个路径类对象 (*path-like object*)。

tempfile.**mkdtemp**(*suffix=None*, *prefix=None*, *dir=None*)

以最安全的方式创建一个临时目录，创建该目录时不会有竞争的情况。该目录只能由创建者读取、写入和搜索。

mkdtemp() 用户用完临时目录后需要自行将其删除。

prefix、*suffix* 和 *dir* 的含义与它们在*mkstemp()* 中的相同。

mkdtemp() 返回新目录的绝对路径。

引发一个审计事件 tempfile.mkdtemp 并附带参数 fullpath。

在 3.5 版本发生变更: 现在，*suffix*、*prefix* 和 *dir* 可以以 bytes 类型按顺序提供，以获得 bytes 类型的返回值。之前只允许使用 str。*suffix* 和 *prefix* 现在可以接受 None，并且默认为 None 以使用合适的默认值。

在 3.6 版本发生变更: *dir* 参数现在可接受一个路径类对象 (*path-like object*)。

在 3.12 版本发生变更: *mkdtemp()* 现在将始终返回绝对路径，即使 *dir* 为相对路径。

tempfile.**gettempdir**()

返回放置临时文件的目录的名称。这个方法的返回值就是本模块所有函数的 *dir* 参数的默认值。

Python 搜索标准目录列表，以找到调用者可以在其中创建文件的目录。这个列表是：

1. TMPDIR 环境变量指向的目录。
2. TEMP 环境变量指向的目录。
3. TMP 环境变量指向的目录。
4. 与平台相关的位置：
   - 在 Windows 上，依次为 C:\TEMP、C:\TMP、\TEMP 和 \TMP。
   - 在所有其他平台上，依次为 /tmp、/var/tmp 和 /usr/tmp。
5. 不得已时，使用当前工作目录。

搜索的结果会缓存起来，参见下面*tempdir* 的描述。

在 3.10 版本发生变更: 总是返回一个字符串。在之前的版本中它会返回任意*tempdir* 值而不考虑它的类型，只要它不为 None。

tempfile.**gettempdirb**()

与*gettempdir()* 相同，但返回值为字节类型。

Added in version 3.5.

tempfile.**gettempprefix**()

返回用于创建临时文件的文件名前缀，它不包含目录部分。

tempfile.**gettempprefixb**()

与*gettempprefix()* 相同，但返回值为字节类型。

Added in version 3.5.

本模块使用一个全局变量来存储由*gettempdir()* 返回的临时文件使用的目录路径。它可被直接设置以覆盖选择过程，但不建议这样做。本模块中的所有函数都接受一个 *dir* 参数，它可被用于指定目录。这是不会通过改变全局 API 行为对其他无准备代码造成影响的推荐做法。

tempfile.**tempdir**

当设为 None 以外的值时，此变量会为本模块中定义的函数的 *dir* 参数定义默认值，包括确定其类型为字节串还是字符串。它不可以为*path-like object*。

如果在调用除*gettempprefix()* 外的上述任何函数时 tempdir 为 None (默认值) 则它会按照*gettempdir()* 中所描述的算法来初始化。

备注

请注意如果你将 tempdir 设为字节串值，会有一个麻烦的副作用: *mkstemp()* 和*mkdtemp()* 的全局默认返回类型会在没有显式提供字符串类型的 when no explicit prefix, suffix 或 dir 的时候被改为字节串。请不要编写预期或依赖于此入围的代码。这个笨拙行为是为了保持与历史实现的兼容性。

11.5.1 例子

以下是*tempfile* 模块典型用法的一些示例:

```
>>> import tempfile

# 创建一个临时文件并向其写入一些数据
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# 从文件读取数据
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# 关闭文件，它将被移除
>>> fp.close()

# 使用上下文管理器创建一个临时文件
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# 现在文件已被关闭并移除

# 使用上下文管理器创建一个临时文件
# 关闭该文件，再使用文件名再次打开该文件
>>> with tempfile.NamedTemporaryFile(delete_on_close=False) as fp:
...     fp.write(b'Hello world!')
...     fp.close()
```

(续下页)

(接上页)

```
... # 文件已被关闭，但未被移除
... # 使用文件名再次打开该文件
...     with open(fp.name, mode='rb') as f:
...         f.read()
b'Hello world!'
>>>
# 现在文件已被移除

# 使用上下文管理器创建一个临时目录
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# 目录及其内容已被移除
```

11.5.2 已弃用的函数和变量

创建临时文件有一种历史方法，首先使用*mktemp()* 函数生成一个文件名，然后使用该文件名创建文件。不幸的是，这是不安全的，因为在调用*mktemp()* 与随后尝试创建文件的进程之间的时间里，其他进程可能会使用该名称创建文件。解决方案是将两个步骤结合起来，立即创建文件。这个方案目前被*mkstemp()* 和上述其他函数所采用。

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

自 2.3 版本弃用: 使用*mkstemp()* 来代替。

返回一个绝对路径，这个路径指向的文件在调用本方法时不存在。*prefix*、*suffix* 和 *dir* 参数与*mkstemp()* 中的同名参数类似，不同之处在于不支持字节类型的文件名，不支持 `suffix=None` 和 `prefix=None`。

警告

使用此功能可能会在程序中引入安全漏洞。当你开始使用本方法返回的文件执行任何操作时，可能有人已经捷足先登了。*mktemp()* 的功能可以很轻松地用*NamedTemporaryFile()* 代替，当然需要传递 `delete=False` 参数:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.6 glob --- Unix 风格的路径名模式扩展

源代码: Lib/glob.py

---

glob 模块会按照 Unix shell 所使用的规则找出所有匹配特定模式的路径名称，但返回结果的顺序是不确定的。波浪号扩展不会生效，但 `*`, `?` 以及用 `[]` 表示的字符范围将被正确地匹配。这是通过配合使用*os.scandir()* 和*fnmatch.fnmatch()* 函数来实现的，而不是通过实际唤起子 shell。

请注意以点号 (`.`) 打头的文件只能用同样以点号打头的模式来匹配，这不同于*fnmatch.fnmatch()* 或*pathlib.Path.glob()*。(对于波浪号和 shell 变量扩展，请使用*os.path.expanduser()* 和*os.path.expandvars()*。)

对于字面值匹配，请将原字符用方括号括起来。例如，`'[?]'` 将匹配字符 `'?'`。

glob 模块定义了下列函数：

glob.**glob**(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

返回一个匹配 *pathname* 的可能为空的路径名列表，其中的元素必须为包含路径信息的字符串。*pathname* 可以是绝对路径 (如 `/usr/src/Python-1.5/Makefile`) 或相对路径 (如 `../../Tools/*/*.gif`)，并可包含 shell 风格的通配符。无效的符号链接也将包括在结果中 (如像在 shell 中一样)。结果是否排序取决于具体文件系统。如果某个符合条件的文件在调用此函数期间被移除或添加，是否包括该文件的路径是没有规定的。

如果 *root_dir* 不为 `None`，则它应当是指明要搜索的根目录的*path-like object*。它用在*glob()* 上与在调用它之前改变当前目录有相同的效果。如果 *pathname* 为相对路径，结果将包含相对于 *root_dir* 的路径。

本函数带有 *dir_fd* 参数，支持基于目录描述符的相对路径。

如果 *recursive* 为真值，则模式"`**`" 将匹配目录中的任何文件以及零个或多个目录、子目录和符号链接。如果模式加了一个*os.sep* 或*os.altsep* 则将不匹配文件。

如果 *include_hidden* 为真值，"`**`" 模式将匹配隐藏目录。

引发一个审计事件 `glob.glob` 并附带参数 `pathname`, `recursive`。

引发一个审计事件 `glob.glob/2` 并附带参数 `pathname`, `recursive`, `root_dir`, `dir_fd`。

> **备注**
>
> 在一个较大的目录树中使用"`**`" 模式可能会消耗非常多的时间。

> **备注**
>
> 如果 *pathname* 包含多个"`**`" 模式并且 *recursive* 为真值则此函数可能返回重复的路径名。

在 3.5 版本发生变更: 支持使用"`**`" 的递归 glob。

在 3.10 版本发生变更: 添加了 *root_dir* 和 *dir_fd* 形参。

在 3.11 版本发生变更: 增加了 *include_hidden* 形参。

glob.**iglob**(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

返回一个*iterator*，它会产生与*glob()* 相同的结果，但不会实际地同时保存它们。

引发一个审计事件 `glob.glob` 并附带参数 `pathname`, `recursive`。

引发一个审计事件 `glob.glob/2` 并附带参数 `pathname`, `recursive`, `root_dir`, `dir_fd`。

> **备注**
>
> 如果 *pathname* 包含多个"`**`" 模式并且 *recursive* 为真值则此函数可能返回重复的路径名。

在 3.5 版本发生变更: 支持使用"`**`" 的递归 glob。

在 3.10 版本发生变更: 添加了 *root_dir* 和 *dir_fd* 形参。

在 3.11 版本发生变更: 增加了 *include_hidden* 形参。

glob.**escape**(*pathname*)

转义所有特殊字符 (`'?'`, `'*'` 和 `'['`)。这适用于当你想要匹配可能带有特殊字符的任意字符串字面值的情况。在 drive/UNC 共享点中的特殊字符不会被转义，例如在 Windows 上 `escape('//?/c:/Quo vadis?.txt')` 将返回 `'//?/c:/Quo vadis[?].txt'`。

Added in version 3.4.

glob.**translate**(*pathname*, *, *recursive=False*, *include_hidden=False*, *seps=None*)

将给定的路径规格说明转换为一个正则表达式供*re.match()* 使用。路径规格说明可以包含 shell 风格的通配符。

例如:

```
>>> import glob, re
>>>
>>> regex = glob.translate('**/*.txt', recursive=True, include_hidden=True)
>>> regex
'(?s:(?:.+/)?[^/]*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foo/bar/baz.txt')
<re.Match object; span=(0, 15), match='foo/bar/baz.txt'>
```

路径分隔符与部件对该函数是有意义的，这与*fnmatch.translate()* 不同。在默认情况下通配符不会匹配路径分隔符，而 `*` 模式部件将精确匹配一个路径部件。

如果 *recursive* 为真值，则模式部件"`**`" 将匹配任意数量的路径部件。

如果 *include_hidden* 为真值，则通配符可以匹配以点号 (`.`) 打头的路径部件。

可以向 *seps* 参数提供一个由路径分隔符组成的序列。如果未给出，则将使用*os.sep* 和*altsep* (如果可用)。

参见

pathlib.PurePath.full_match() 和*pathlib.Path.glob()* 方法，它们将调用此函数来实现模式匹配和 glob 操作。

Added in version 3.13.

11.6.1 例子

考虑一个包含以下文件的目录: `1.gif`, `2.txt`, `card.gif` 以及一个子目录 `sub` 且其中只包含一个文件 `3.txt`。*glob()* 将产生如下结果。请注意路径的任何开头部件都将被保留。

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

如果目录包含以 `.` 打头的文件，它们默认将不会被匹配。例如，考虑一个包含 `card.gif` 和 `.card.gif` 的目录:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

参见

fnmatch 模块提供了 shell 风格的文件名（而非路径）扩展。

参见

pathlib 模块提供高级路径对象。

11.7 fnmatch --- Unix 文件名模式匹配

源代码: Lib/fnmatch.py

此模块提供了 Unix shell 风格的通配符，它们 并不等同于正则表达式（关于后者的文档参见*re* 模块）。shell 风格通配符所使用的特殊字符如下：

模式	含意
`*`	匹配所有
`?`	匹配任何单个字符
`[seq]`	匹配 *seq* 中的任何字符
`[!seq]`	匹配任何不在 *seq* 中的字符

对于字面值匹配，请将原字符用方括号括起来。例如，`'[?]'` 将匹配字符 `'?'`。

注意文件名分隔符 (Unix 上为 `'/'`) 不会被此模块特别对待。请参见*glob* 模块了解文件名扩展 (*glob* 使用*filter()* 来匹配文件名的各个部分)。类似地，以一个句点打头的文件名也不会被此模块特别对待，可以通过 `*` 和 `?` 模式来匹配。

除非另有说明，”文件名字符串” 和”模式字符串” 是指使用*str* 或 `ISO-8859-1` 编码的*bytes* 对象。请注意下的记录的函数不允许将 `bytes` 模式与 `str` 文件名混用，反之亦然。

最后，请注意 *maxsize* 为 32768 的*functools.lru_cache()* 将被用于缓存下列函数中（标注类型的）已编译正则表达式: *fnmatch()*, *fnmatchcase()*, *filter()*。

fnmatch.**fnmatch**(*name*, *pat*)

检测文件名字符串 *name* 是否匹配模式字符串 *pat*，返回 `True` 或 `False`。两个形参都会使用*os.path.normcase()* 进行大小写正规化。*fnmatchcase()* 可被用于执行大小写敏感的比较，无论这是否为所在操作系统的标准。can be used to perform a case-sensitive comparison, regardless of whether that's standard for the operating system.

这个例子将打印当前目录下带有扩展名 `.txt` 的所有文件名:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*name*, *pat*)

检测文件名字符串 *name* 是否匹配模式字符串 *pat*，返回 `True` 或 `False`；此比较是大小写敏感的并且不会应用*os.path.normcase()*。

fnmatch.**filter**(*names*, *pat*)

基于包含匹配模式字符串 *pat* 的文件名字符串 *names* 的*iterable* 构造一个列表。它等价于 `[n for n in names if fnmatch(n, pat)]`，但实现得更为高效。

fnmatch.**translate**(*pat*)

返回由 shell 风格的模式 *pat* 转换成正则表达式配合*re.match()* 使用。此模式预期为一个*str*。

示例:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

参见

模块*glob*
Unix shell 风格路径扩展。

11.8 linecache --- 随机访问文本行

源代码: Lib/linecache.py

---

linecache 模块允许从一个 Python 源文件中获取任意的行，并会尝试使用缓存进行内部优化，常应用于从单个文件读取多行的场合。此模块被*traceback* 模块用来提取源码行以便包含在格式化的回溯中。

tokenize.open() 函数被用于打开文件。此函数使用*tokenize.detect_encoding()* 来获取文件的编码格式；如果未指明编码格式，则默认编码为 UTF-8。

linecache 模块定义了下列函数:

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)

从名为 *filename* 的文件中获取 *lineno* 行，此函数绝不会引发异常 --- 出现错误时它将返回 `''` (所有找到的行都将包含换行符作为结束)。

如果找不到名为 *filename* 的文件，此函数会先在 *module_globals* 中检查 **PEP 302** `__loader__`。如果存在这样的加载器并且它定义了 `get_source` 方法，则由该方法来确定源行 (如果 `get_source()` 返回 `None`，则该函数返回 `''`)。最后，如果 *filename* 是一个相对路径文件名，则它会在模块搜索路径 `sys.path` 中按条目的相对位置进行查找。

linecache.**clearcache**()

清空缓存。如果你不再需要之前使用*getline()* 从文件读取的行即可使用此函数。

linecache.**checkcache**(*filename=None*)

检查缓存有效性。如果缓存中的文件在磁盘上发生了改变，而你需要更新后的版本即可使用此函数。如果省略了 *filename*，它会检查缓存中的所有条目。

linecache.**lazycache**(*filename*, *module_globals*)

捕获有关某个非基于文件的模块的足够细节信息，以允许稍后再通过*getline()* 来获取其中的行，即使当稍后调用时 *module_globals* 为 `None`。这可以避免在实际需要读取行之前执行 I/O，也不必始终保持模块全局变量。

Added in version 3.5.

示例:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.9 shutil --- 高层级文件操作

源代码： Lib/shutil.py

shutil 模块提供了一系列对文件和文件集合的高阶操作。特别是提供了一些支持文件拷贝和删除的函数。对于单个文件的操作，请参阅*os* 模块。

> **警告**
>
> 即便是高阶文件拷贝函数 (*shutil.copy()*, *shutil.copy2()*) 也无法拷贝所有的文件元数据。
>
> 在 POSIX 平台上，这意味着将丢失文件所有者和组以及 ACL 数据。在 Mac OS 上，资源钩子和其他元数据不被使用。这意味着将丢失这些资源并且文件类型和创建者代码将不正确。在 Windows 上，将不会拷贝文件所有者、ACL 和替代数据流。

11.9.1 目录和文件操作

shutil.**copyfileobj**(*fsrc*, *fdst*[, *length*])

将文件型对象 *fsrc* 的内容拷贝到文件型对象 *fdst*。如果给出了整数值 *length*，即为缓冲区大小。特别地，*length* 为负值表示拷贝数据时不对源数据进行分块循环处理；在默认情况下会分块读取数据以避免不受控制的内存消耗。请注意如果 *fsrc* 对象的当前文件位置不为 0，只有从当前文件位置到文件末尾的内容会被拷贝。

shutil.**copyfile**(*src*, *dst*, ***, *follow_symlinks=True*)

将名为 *src* 的文件的内容（不带元数据）拷贝到名为 *dst* 的文件并以尽可能高效的方式返回 *dst*。*src* 和 *dst* 均为数据型对象 或字符串形式的路径名。

dst 必须是完整的目标文件名；对于接受目标目录路径的拷贝请参见*copy()*。如果 *src* 和 *dst* 指定了同一个文件，则将引发*SameFileError*。

目标位置必须是可写的；否则将引发*OSError* 异常。如果 *dst* 已经存在，它将被替换。特殊文件如字符或块设备以及管道无法用此函数来拷贝。

如果 *follow_symlinks* 为假值且 *src* 为符号链接，则将创建一个新的符号链接而不是拷贝 *src* 所指向的文件。

引发一个审计事件 `shutil.copyfile` 并附带参数 `src`, `dst`。

在 3.3 版本发生变更: 曾经是引发*IOError* 而不是*OSError*。增加了 *follow_symlinks* 参数。现在是返回 *dst*。

在 3.4 版本发生变更: 引发*SameFileError* 而不是*Error*。由于前者是后者的子类，此改变是向后兼容的。

在 3.8 版本发生变更: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

exception shutil.**SameFileError**

此异常会在*copyfile()* 中的源和目标为同一文件时被引发。

Added in version 3.4.

shutil.**copymode**(*src*, *dst*, ***, *follow_symlinks=True*)

将权限位从 *src* 拷贝到 *dst*。文件的内容、所有者和分组将不受影响。*src* 和 *dst* 均为路径型对象 或字符串形式的路径名。如果 *follow_symlinks* 为假值，并且 *src* 和 *dst* 均为符号链接，则*copymode()* 将尝试修改 *dst* 本身的模式（而不是它所指向的文件）。此功能并不是在所有平台上均可用；请参阅*copystat()* 了解详情。如果*copymode()* 无法修改本机平台上的符号链接，而它被要求这样做，它将不做任何操作即返回。

引发一个审计事件 `shutil.copymode` 并附带参数 `src`, `dst`。

在 3.3 版本发生变更: 加入 *follow_symlinks* 参数。

shutil.**copystat**(*src*, *dst*, *, *follow_symlinks=True*)

将权限位、最近访问时间、最近修改时间和旗标从 *src* 拷贝到 *dst*。在 Linux 上，*copystat()* 还会在可能的情况下拷贝“扩展属性”。文件的内容、所有者和分组将不受影响。*src* 和 *dst* 均为路径型对象 或字符串形式的路径名。

如果 *follow_symlinks* 为假值，并且 *src* 和 *dst* 均指向符号链接，*copystat()* 将作用于符号链接本身而非该符号链接所指向的文件—从 *src* 符号链接读取信息，并将信息写入 *dst* 符号链接。

> **备注**
>
> 并非所有平台者提供检查和修改符号链接的功能。Python 本身可以告诉你哪些功能是在本机上可用的。
>
> - 如果 `os.chmod in os.supports_follow_symlinks` 为 `True`，则*copystat()* 可以修改符号链接的权限位。
> - 如果 `os.utime in os.supports_follow_symlinks` 为 `True`，则*copystat()* 可以修改符号链接的最近访问和修改时间。
> - 如果 `os.chflags in os.supports_follow_symlinks` 为 `True`，则*copystat()* 可以修改符号链接的旗标。(`os.chflags` 不是在所有平台上均可用。)
>
> 在此功能部分或全部不可用的平台上，当被要求修改一个符号链接时，*copystat()* 将尽量拷贝所有内容。*copystat()* 一定不会返回失败信息。
>
> 更多信息请参阅*os.supports_follow_symlinks*。

引发一个审计事件 `shutil.copystat` 并附带参数 `src`, `dst`。

在 3.3 版本发生变更: 添加了 *follow_symlinks* 参数并且支持 Linux 扩展属性。

shutil.**copy**(*src*, *dst*, *, *follow_symlinks=True*)

将文件 *src* 拷贝到文件或目录 *dst*。*src* 和 *dst* 应为路径类对象 或字符串。如果 *dst* 指定了一个目录，文件将使用 *src* 中的基准文件名拷贝到 *dst* 中。如果 *dst* 指定了一个已存在的文件，它将被替换。返回新创建文件所对应的路径。

如果 *follow_symlinks* 为假值且 *src* 为符号链接，则 *dst* 也将被创建为符号链接。如果 *follow_symlinks* 为真值且 *src* 为符号链接，*dst* 将成为 *src* 所指向的文件的一个副本。

copy() 会拷贝文件数据和文件的权限模式 (参见*os.chmod()*)。其他元数据，例如文件的创建和修改时间不会被保留。要保留所有原有的元数据，请改用*copy2()* 。

引发一个审计事件 `shutil.copyfile` 并附带参数 `src`, `dst`。

引发一个审计事件 `shutil.copymode` 并附带参数 `src`, `dst`。

在 3.3 版本发生变更: 添加了 *follow_symlinks* 参数。现在会返回新创建文件的路径。

在 3.8 版本发生变更: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

shutil.**copy2**(*src*, *dst*, *, *follow_symlinks=True*)

类似于*copy()*，区别在于*copy2()* 还会尝试保留文件的元数据。

当 *follow_symlinks* 为假值，并且 *src* 为符号链接时，*copy2()* 会尝试将来自 *src* 符号链接的所有元数据拷贝到新创建的 *dst* 符号链接。但是，此功能不是在所有平台上均可用。在此功能部分或全部不可用的平台上，*copy2()* 将尽量保留所有元数据，*copy2()* 一定不会由于无法保留文件元数据而引发异常。

copy2() 会使用*copystat()* 来拷贝文件元数据。请参阅*copystat()* 了解有关修改符号链接元数据的平台支持的更多信息。

引发一个审计事件 `shutil.copyfile` 并附带参数 `src`, `dst`。

引发一个审计事件 `shutil.copystat` 并附带参数 `src`, `dst`。

在 3.3 版本发生变更: 添加了 *follow_symlinks* 参数，还会尝试拷贝扩展文件系统属性（目前仅限 Linux）。现在会返回新创建文件的路径。

在 3.8 版本发生变更: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

shutil.**ignore_patterns**(**patterns*)

这个工厂函数会创建一个函数，它可被用作`copytree()` 的 *ignore* 可调用对象参数，以忽略那些匹配所提供的 glob 风格的 *patterns* 之一的文件和目录。参见以下示例。

shutil.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

递归地将以 *src* 为根起点的整个目录树拷贝到名为 *dst* 的目录并返回目标目录。所需的包含 *dst* 的中间目录在默认情况下也将被创建。

目录的权限和时间会通过`copystat()` 来拷贝，单个文件则会使用`copy2()` 来拷贝。

如果 *symlinks* 为真值，源目录树中的符号链接会在新目录树中表示为符号链接，并且原链接的元数据在平台允许的情况下也会被拷贝；如果为假值或省略，则会将被链接文件的内容和元数据拷贝到新目录树。

当 *symlinks* 为假值时，如果符号链接所指向的文件不存在，则会在拷贝进程的末尾将一个异常添加到`Error` 异常中的被引发错误列表。如果你希望屏蔽此异常则可以将可选的 *ignore_dangling_symlinks* 旗标设为真值。请注意此选项在不支持`os.symlink()` 的平台上将不起作用。

如果给出了 *ignore*，它必须是一个可调用对象，该对象将接受`copytree()` 所访问的目录以及`os.listdir()` 所返回的目录内容列表作为其参数。由于`copytree()` 是递归地被调用的，*ignore* 可调用对象对于每个被拷贝目录都将被调用一次。该可调用对象必须返回一个相对于当前目录的目录和文件名序列（即其第二个参数的子集）；随后这些名称将在拷贝进程中被忽略。`ignore_patterns()` 可被用于创建这种基于 glob 风格模式来忽略特定名称的可调用对象。

如果发生了（一个或多个）异常，将引发一个附带原因列表的`Error`。

如果给出了 *copy_function*，它必须是一个将被用来拷贝每个文件的可调用对象。它在被调用时会将源路径和目标路径作为参数传入。默认情况下，`copy2()` 将被使用，但任何支持同样签名（与`copy()` 一致）都可以使用。

如果 *dirs_exist_ok* 为（默认的）假值且 *dst* 已存在，则会引发`FileExistsError`。如果 *dirs_exist_ok* 为真值，则如果拷贝操作遇到已存在的目录时将继续执行，并且在 *dst* 目录树中的文件将被 *src* 目录树中对应的文件所覆盖。

引发一个审计事件 `shutil.copytree` 并附带参数 `src`, `dst`。

在 3.2 版本发生变更: 添加了 *copy_function* 参数以允许提供定制的拷贝函数。添加了 *ignore_dangling_symlinks* 参数以便在 *symlinks* 为假值时屏蔽目标不存在的符号链接。

在 3.3 版本发生变更: 当 *symlinks* 为假值时拷贝元数据。现在会返回 *dst*。

在 3.8 版本发生变更: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

在 3.8 版本发生变更: 增加了 *dirs_exist_ok* 形参。

shutil.**rmtree**(*path*, *ignore_errors=False*, *onerror=None*, ***, *onexc=None*, *dir_fd=None*)

删除一个完整的目录树；*path* 必须指向一个目录（但不能是一个目录的符号链接）。如果 *ignore_errors* 为真值，则删除失败导致的错误将被忽略；如果为假值或被省略，则此类错误将通过调用由 *onexc* 或 *onerror* 所指定的处理器来处理，或者如果此参数被省略，异常将被传播给调用方。

本函数支持基于目录描述符的相对路径。

备注

在支持必要的基于 fd 的函数的平台上，默认会使用`rmtree()` 的可防御符号链接攻击的版本。在其他平台上，`rmtree()` 较易遭受符号链接攻击：给定适当的时间和环境，攻击者可

以操纵文件系统中的符号链接来删除他们在其他情况下无法访问的文件。应用程序可以使用*rmtree.avoids_symlink_attacks* 函数属性来确定此类情况具体是哪一些。

如果提供了 *onexc*，它必须为接受三个形参的可调用对象: *function*, *path* 和 *excinfo*。

第一个形参 *function* 是引发异常的函数；它依赖于具体的平台和实现。第二个形参 *path* 将为传递给 *function* 的路径名称。第三个形参 *excinfo* 是被引发的异常。由 *onexc* 所引发的异常将不会被捕获。

已弃用的 *onerror* 与 *onexc* 类似，区别在于它接受的第三个形参是从*sys.exc_info()* 返回的元组。

引发一个审计事件 `shutil.rmtree` 并附带参数 `path`, `dir_fd`。

在 3.3 版本发生变更: 添加了一个防御符号链接攻击的版本，如果平台支持基于 fd 的函数就会被使用。

在 3.8 版本发生变更: 在 Windows 上将不会再在移除连接之前删除目录连接中的内容。

在 3.11 版本发生变更: 添加了 *dir_fd* 参数。

在 3.12 版本发生变更: 增加了 *onexc* 形参，弃用了 *onerror*。

在 3.13 版本发生变更: 现在 `rmtree()` 会忽略最高层级路径以外所有路径的*FileNotFoundError* 异常。*OSError* 和 `OSError` 的子类现在总是会被传播给调用方。

`rmtree.`**`avoids_symlink_attacks`**

指明当前平台和实现是否提供防御符号链接攻击的*rmtree()* 版本。目前它仅在平台支持基于 fd 的目录访问函数时才返回真值。

Added in version 3.3.

`shutil.`**`move`**(*src*, *dst*, *copy_function=copy2*)

递归地将一个文件或目录 (*src*) 移到另一位置并返回目标位置。

如果 *dst* 为已存在的目录或指向目录的符号链接，则 *src* 将被移到该目录中。目标路径在该目录中不能已存在。

如果 *dst* 已存在但不是一个目录，则它可能会被覆盖，具体取决于*os.rename()* 的语义。

如果目标是在当前文件系统中，则会使用*os.rename()*。在其他情况下，则使用 *copy_function* 将 *src* 拷贝至目标然后移除它。对于符号链接，则将创建一个指向 *src* 目标的新符号链接作为目标位置而 *src* 将被移除。

如果给出了 *copy_function*，则它必须为接受两个参数 *src* 和目标位置的可调用对象，并将在*os.rename()* 无法使用时被用来将 *src* 拷贝到目标位置。如果源是一个目录，则会调用*copytree()*，并向它传入 *copy_function*。默认的 *copy_function* 是*copy2()*。使用*copy()* 作为 *copy_function* 将允许在无法附带拷贝元数据时让移动操作成功执行，但其代价是不拷贝任何元数据。

引发一个审计事件 `shutil.move` 并附带参数 `src`, `dst`。

在 3.3 版本发生变更: 为异类文件系统添加了显式的符号链接处理，以便使它适应 GNU 的 **mv** 的行为。现在会返回 *dst*。

在 3.5 版本发生变更: 增加了 *copy_function* 关键字参数。

在 3.8 版本发生变更: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

在 3.9 版本发生变更: 接受一个*path-like object* 作为 *src* 和 *dst*。

`shutil.`**`disk_usage`**(*path*)

返回给定路径的磁盘使用统计数据，形式为一个*named tuple*，其中包含 *total*, *used* 和 *free* 属性，分别表示总计、已使用和未使用空间的字节数。*path* 可以是一个文件或是一个目录。

> **备注**
>
> 在 Unix 文件系统中，*path* 必须指向一个 **已挂载**文件系统分区中的路径。在这些平台上，CPython 不会尝试从未挂载的文件系统中获取磁盘使用信息。

Added in version 3.3.

在 3.8 版本发生变更: 在 Windows 上，*path* 现在可以是一个文件或目录。

Availability: Unix, Windows.

shutil.**chown**(*path*, *user=None*, *group=None*, *, *dir_fd=None*, *follow_symlinks=True*)

修改给定 *path* 的所有者 *user* 和/或 *group*。

user 可以是一个系统用户名或 uid；*group* 同样如此。要求至少有一个参数。

另请参阅下层的函数*os.chown()*。

引发一个审计事件 `shutil.chown` 并附带参数 `path`, `user`, `group`。

Availability: Unix.

Added in version 3.3.

在 3.13 版本发生变更: 增加了 *dir_fd* 和 *follow_symlinks* 形参。

shutil.**which**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

返回当给定的 *cmd* 被调用时将要运行的可执行文件的路径。如果没有 *cmd* 会被调用则返回 `None`。

mode 是一个传递给*os.access()* 的权限掩码，在默认情况下将确定文件是否存在并且为可执行文件。

path 是一个指明要查找的目录的"`PATH` 字符串"，由*os.pathsep* 分隔。当未指定 *path* 时，将从*os.environ* 读取 `PATH` 环境变量，如果其未被设置则将回退至*os.defpath*。

在 Windows 上，如果 *mode* 不包括 `os.X_OK` 则会将当前目录添加到 *path* 中。当 *mode* 包括 `os.X_OK` 时，则将通过 Windows API `NeedCurrentDirectoryForExePathW` 来确定当前目录是否应当添加到 *path* 中。要避免在当前工作目录下查找可执行文件：可设置 `NoDefaultCurrentDirectoryInExePath` 环境变量。

在 Windows 上，还会使用 `PATHEXT` 环境变量来处理可能尚未包括某个扩展的命令。举例来说，如果你调用 `shutil.which("python")`，*which()* 将搜索 `PATHEXT` 以获知应当在 *path* 的目录中查找 `python.exe`。例如，在 Windows 上:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

这也适用于当 *cmd* 是一个包含目录组成部分路径的情况:

```
>>> shutil.which("C:\\Python33\\python")
'C:\\Python33\\python.EXE'
```

Added in version 3.3.

在 3.8 版本发生变更: 现在可以接受*bytes* 类型。如果 *cmd* 的类型为*bytes*，结果的类型也将为*bytes*。

在 3.12 版本发生变更: 在 Windows 上，如果 *mode* 包括 `os.X_OK` 且 WinAPI `NeedCurrentDirectoryForExePathW(cmd)` 为假值则不会再将当前目录添加到搜索路径中，否则即使当前目录已经在搜索路径中仍会再次添加它；现在 `PATHEXT` 即使当 *cmd* 包括目录组成部分或以 `PATHEXT` 中的扩展名结束时仍然会被使用；并且没有扩展名的文件名现在也能被找到。

exception shutil.**Error**

此异常会收集在多文件操作期间所引发的异常。对于*copytree()*，此异常参数将是一个由三元组 (*srcname*, *dstname*, *exception*) 构成的列表。

依赖于具体平台的高效拷贝操作

从 Python 3.8 开始，所有涉及文件拷贝的函数 (*copyfile()*, *copy()*, *copy2()*, *copytree()* 以及*move()*) 将会使用平台专属的"fast-copy" 系统调用以便更高效地拷贝文件 (参见 bpo-33671)。"fast-copy" 意味着拷贝操作将发生于内核之中，避免像在"`outfd.write(infd.read())`" 中那样使用 Python 用户空间的缓冲区。

在 macOS 上将会使用 fcopyfile 来拷贝文件内容（不含元数据）。

在 Linux 上将会使用*os.sendfile()*。

在 Windows 上*shutil.copyfile()* 将会使用更大的默认缓冲区（1 MiB 而非 64 KiB）并且会使用基于*memoryview()* 的*shutil.copyfileobj()* 变种形式。

如果快速拷贝操作失败并且没有数据被写入目标文件，则 shutil 将在内部静默地回退到使用效率较低的*copyfileobj()* 函数。

在 3.8 版本发生变更.

copytree 示例

一个使用*ignore_patterns()* 辅助函数的例子:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

这将会拷贝除 `.pyc` 文件和以 `tmp` 打头的文件或目录以外的所有条目.

另一个使用 *ignore* 参数来添加记录调用的例子:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree 示例

这个例子演示了如何在 Windows 上删除一个目录树，其中部分文件设置了只读属性位。它会使用 onexc 回调函数来清除只读属性并再次尝试删除。任何后续的失败都将被传播。

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onexc=remove_readonly)
```

11.9.2 归档操作

Added in version 3.2.

在 3.5 版本发生变更: 添加了对 *xztar* 格式的支持。

本模块也提供了用于创建和读取压缩和归档文件的高层级工具。它们依赖于*zipfile* 和*tarfile* 模块。

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

创建一个归档文件（例如 zip 或 tar）并返回其名称。

base_name 是要创建的文件的名称，包括路径，去除任何格式专属的扩展名。

format 是归档格式：为"zip"（如果*zlib*模块可用），"tar"，"gztar"（如果*zlib*模块可用），"bztar"（如果*bz2*模块可用）或"xztar"（如果*lzma*模块可用）中的一个。

root_dir 是一个目录，它将作为归档文件的根目录，归档中的所有路径都将是它的相对路径；例如，我们通常会在创建归档之前用 chdir 命令切换到 *root_dir*。

base_dir 是我们要执行归档的起始目录；也就是说 *base_dir* 将成为归档中所有文件和目录共有的路径前缀。*base_dir* 必须相对于 *root_dir* 给出。请参阅*使用 base_dir 的归档程序示例* 了解如何同时使用 *base_dir* 和 *root_dir*。

root_dir 和 *base_dir* 默认均为当前目录。

如果 *dry_run* 为真值，则不会创建归档文件，但将要被执行的操作会被记录到 *logger*。

owner 和 *group* 将在创建 tar 归档文件时被使用。默认会使用当前的所有者和分组。

logger 必须是一个兼容 **PEP 282** 的对象，通常为*logging.Logger*的实例。

verbose 参数已不再使用并进入弃用状态。

引发一个*审计事件* `shutil.make_archive` 并附带参数 `base_name`, `format`, `root_dir`, `base_dir`。

> **备注**
>
> 此函数在通过*register_archive_format()*注册的自定义归档程序不支持 *root_dir* 参数时时不是线程安全的。在这种情况下它会临时改变进程的当前工作目录到 *root_dir* 来执行归档操作。

在 3.8 版本发生变更: 现在对于通过 `format="tar"` 创建的归档文件将使用新式的 pax (POSIX.1-2001) 格式而非旧式的 GNU 格式。

在 3.10.6 版本发生变更: 目前此函数在创建标准 `.zip` 和 tar 归档文件期间会确保是线程安全的。

shutil.**get_archive_formats**()

返回支持的归档格式列表。所返回序列中的每个元素为一个元组 `(name, description)`。

默认情况下*shutil*提供以下格式:

- *zip*: ZIP 文件（如果*zlib*模块可用）。
- *tar*: 未压缩的 tar 文件。对于新归档文件将使用 POSIX.1-2001 pax 格式。
- *gztar*: gzip 压缩的 tar 文件（如果*zlib*模块可用）。
- *bztar*: bzip2 压缩的 tar 文件（如果*bz2*模块可用）。
- *xztar*: xz 压缩的 tar 文件（如果*lzma*模块可用）。

你可以通过使用*register_archive_format()*注册新的格式或为任何现有格式提供你自己的归档器。

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])

为 *name* 格式注册一个归档器。

function 是将被用来解包归档文件的可调用对象。该可调用对象将接收要创建文件的 *base_name*，再加上要归档内容的 *base_dir* (其默认值为*os.curdir*)。更多参数会被作为关键字参数传入: *owner*, *group*, *dry_run* 和 *logger* (与向*make_archive()*传入的参数一致)。

如果 *function* 将自定义属性 `function.supports_root_dir` 设为 `True`，则会以关键字参数形式传递 *root_dir* 参数。否则进程的当前工作目录将在调用 *function* 之前被临时更改为 *root_dir*。在此情况下*make_archive()*将不是线程安全的。

如果给出了 *extra_args*，则其应为一个 `(name, value)` 对的序列，将在归档器可调用对象被使用时作为附加的关键字参数。

description 由 `get_archive_formats()` 使用，它将返回归档器的列表。默认值为一个空字符串。

在 3.12 版本发生变更: 增加了对支持 *root_dir* 参数的函数的支持。

`shutil.`**`unregister_archive_format`**(*name*)

从支持的格式中移除归档格式 *name*。

`shutil.`**`unpack_archive`**(*filename*[, *extract_dir*[, *format*[, *filter*]]])

解包一个归档文件。*filename* 是归档文件的完整路径。

extract_dir 是归档文件解包的目标目录名称。如果未提供，则将使用当前工作目录。

format 是归档格式：应为"zip"，"tar"，"gztar"，"bztar" 或"xztar" 之一。或者任何通过 `register_unpack_format()` 注册的其他格式。如果未提供，`unpack_archive()` 将使用归档文件的扩展名来检查是否注册了对应于该扩展名的解包器。在未找到任何解包器的情况下，将引发 `ValueError`。

仅限关键字参数 *filter* 将被传给下层的解包函数。对于 zip 文件，*filter* 将不被接受。对于 tar 文件，推荐将其设为 `'data'`，除非使用了 tar 专属的特征且为 UNIX 类文件系统。（请参阅解压缩过滤器了解详情。）`'data'` 将在 Python 3.14 中成为 tar 文件的默认过滤器。

引发一个审计事件 `shutil.unpack_archive` 并附带参数 `filename`, `extract_dir`, `format`。

> **警告**
>
> 绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能会在 *extract_dir* 参数所指定的路径之外创建文件，例如某些成员具有以"/" 打头的绝对路径文件名或是以两个点号".." 打头的文件名。

在 3.7 版本发生变更: 接受一个 *path-like object* 作为 *filename* 和 *extract_dir*。

在 3.12 版本发生变更: 增加了 *filter* 参数。

`shutil.`**`register_unpack_format`**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])

注册一个解包格式。*name* 为格式名称而 *extensions* 为对应于该格式的扩展名列表，例如 Zip 文件的扩展名为 `.zip`。

function 是将被用于解包归档的可调用对象。该可调用对象将接受:

- 归档的路径，为位置参数;
- 归档要提取到的目录，为位置参数;
- 可选的 *filter* 关键字参数，如果有提供给 `unpack_archive()` 的话;
- 额外的关键字参数，由 `(name, value)` 元组组成的序列 *extra_args* 指明。

可以提供 *description* 来描述该格式，它将被 `get_unpack_formats()` 返回。

`shutil.`**`unregister_unpack_format`**(*name*)

撤销注册一个解包格式。*name* 为格式的名称。

`shutil.`**`get_unpack_formats`**()

返回所有已注册的解包格式列表。所返回序列中的每个元素为一个元组 `(name, extensions, description)`。

默认情况下 `shutil` 提供以下格式:

- *zip*: ZIP 文件（只有在相应模块可用时才能解包压缩文件）。
- *tar*: 未压缩的 tar 文件。
- *gztar*: gzip 压缩的 tar 文件（如果 `zlib` 模块可用）。

- *bztar*: bzip2 压缩的 tar 文件（如果*bz2* 模块可用）。
- *xztar*: xz 压缩的 tar 文件（如果*lzma* 模块可用）。

你可以通过使用*register_unpack_format()* 注册新的格式或为任何现有格式提供你自己的解包器。

归档程序示例

在这个示例中，我们创建了一个 gzip 压缩的 tar 归档文件，其中包含用户的 .ssh 目录下的所有文件:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

结果归档文件中包含有:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

使用 *base_dir* 的归档程序示例

在这个例子中，与上面的例子 类似，我们演示了如何使用*make_archive()*，但这次是使用 *base_dir*。我们现在具有如下的目录结构:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
        │   └── please_add.txt
        └── do_not_add.txt
```

在最终的归档中，应当会包括 please_add.txt，但不应当包括 do_not_add.txt。因此我们使用以下代码:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/my_archive.tar'
```

列出结果归档中的文件我们将会得到:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.9.3 查询输出终端的尺寸

shutil.**get_terminal_size**(*fallback=(columns, lines)*)

获取终端窗口的尺寸。

对于两个维度中的每一个，会分别检查环境变量 COLUMNS 和 LINES。如果定义了这些变量并且其值为正整数，则将使用这些值。

如果未定义 COLUMNS 或 LINES，这是通常的情况，则连接到*sys.__stdout__* 的终端将通过唤起*os.get_terminal_size()* 被查询。

如果由于系统不支持查询，或是由于我们未连接到某个终端而导致查询终端尺寸不成功，则会使用在 fallback 形参中给出的值。fallback 默认为 (80, 24)，这是许多终端模拟器所使用的默认尺寸。

返回的值是一个*os.terminal_size* 类型的具名元组。

另请参阅: The Single UNIX Specification, Version 2, Other Environment Variables.

Added in version 3.3.

在 3.11 版本发生变更: 如果*os.get_terminal_size()* 返回零值则 fallback 值也将被使用。

参见

模块*os*
操作系统接口，包括处理比 Python 文件对象 更低级别文件的功能。

模块*io*
Python 的内置 I/O 库，包括抽象类和一些具体的类，如文件 I/O 。

内置函数*open()*
使用 Python 打开文件进行读写的标准方法。

CHAPTER 12

数据持久化

本章中描述的模块支持在磁盘上以持久形式存储 Python 数据。*pickle* 和*marshal* 模块可以将许多 Python 数据类型转换为字节流，然后从字节中重新创建对象。各种与 DBM 相关的模块支持一系列基于散列的文件格式，这些格式存储字符串到其他字符串的映射。

本章中描述的模块列表是：

12.1 pickle --- Python 对象序列化

源代码： Lib/pickle.py

模块*pickle* 实现了对一个 Python 对象结构的二进制序列化和反序列化。*"pickling"* 是将 Python 对象及其所拥有的层次结构转化为一个字节流的过程，而 *"unpickling"* 是相反的操作，会将（来自一个*binary file* 或者*bytes-like object* 的）字节流转化回一个对象层次结构。pickling（和 unpickling）也被称为“序列化”，“编组”[1] 或者“平面化”。而为了避免混乱，此处采用术语“封存 (pickling)”和“解封 (unpickling)”。

> **警告**
>
> pickle 模块 **并不安全**。你只应该对你信任的数据进行 unpickle 操作。
>
> 构建恶意的 pickle 数据来 **在解封时执行任意代码**是可能的。绝对不要对不信任来源的数据和可能被篡改过的数据进行解封。
>
> 请考虑使用*hmac* 来对数据进行签名，确保数据没有被篡改。
>
> 在你处理不信任数据时，更安全的序列化格式如*json* 可能更为适合。参见*与 json 模块的比较* 。

12.1.1 与其他 Python 模块间的关系

与 marshal 间的关系

Python 有一个更原始的序列化模块称为*marshal*，但一般地*pickle* 应该是序列化 Python 对象时的首选。*marshal* 存在主要是为了支持 Python 的 .pyc 文件.

pickle 模块与*marshal* 在如下几方面显著地不同：

[1] 不要把它与*marshal* 模块混淆。

- *pickle* 模块会跟踪已被序列化的对象，所以该对象之后再次被引用时不会再次被序列化。*marshal* 不会这么做。

  这隐含了递归对象和共享对象。递归对象指包含对自己的引用的对象。这种对象并不会被 marshal 接受，并且实际上尝试 marshal 递归对象会让你的 Python 解释器崩溃。对象共享发生在对象层级中存在多处引用同一对象时。*pickle* 只会存储这些对象一次，并确保其他的引用指向同一个主副本。共享对象将保持共享，这可能对可变对象非常重要。

- *marshal* 不能被用于序列化用户定义类及其实例。*pickle* 能够透明地存储并保存类实例，然而此时类定义必须能够从与被存储时相同的模块被引入。

- 同样用于序列化的*marshal* 格式不保证数据能移植到不同的 Python 版本中。因为它的主要任务是支持 `.pyc` 文件，必要时会以破坏向后兼容的方式更改这种序列化格式，为此 Python 的实现者保留了更改格式的权利。*pickle* 序列化格式可以在不同版本的 Python 中实现向后兼容，前提是选择了合适的 pickle 协议。如果你的数据要在 Python 2 与 Python 3 之间跨越传递，封存和解封的代码在 2 和 3 之间也是不同的。

与 `json` 模块的比较

在 pickle 协议和 JSON (JavaScript Object Notation) 之间有着本质上的差异:

- JSON 是一个文本序列化格式（它输出 unicode 文本，尽管在大多数时候它会接着以 `utf-8` 编码），而 pickle 是一个二进制序列化格式；
- JSON 是我们可以直观阅读的，而 pickle 不是；
- JSON 是可互操作的，在 Python 系统之外广泛使用，而 pickle 则是 Python 专用的；
- 默认情况下，JSON 只能表示 Python 内置类型的子集，不能表示自定义的类；但 pickle 可以表示大量的 Python 数据类型（可以合理使用 Python 的对象内省功能自动地表示大多数类型，复杂情况可以通过实现*specific object APIs* 来解决）。
- 不像 pickle，对一个不信任的 JSON 进行反序列化的操作本身不会造成任意代码执行漏洞。

> **参见**
>
> *json* 模块: 一个允许 JSON 序列化和反序列化的标准库模块

12.1.2 数据流格式

pickle 所使用的数据格式是 Python 专属的。这样做的好处是没有外部标准如 JSON（它无法表示指针共享）给该格式施加限制；但这也意味着非 Python 程序可能无法重新构建已封存的 Python 对象。

默认情况下，*pickle* 格式使用相对紧凑的二进制来存储。如果需要让文件更小，可以高效地压缩 由 pickle 封存的数据。

pickletools 模块包含了相应的工具用于分析*pickle* 生成的数据流。*pickletools* 源码中包含了对 pickle 协议使用的操作码的大量注释。

当前共有 6 种不同的协议可用于封存操作。使用的协议版本越高，读取所生成 pickle 对象所需的 Python 版本就要越新。

- v0 版协议是原始的“人类可读”协议，并且向后兼容早期版本的 Python。
- v1 版协议是较早的二进制格式，它也与早期版本的 Python 兼容。
- 第 2 版协议是在 Python 2.3 中引入的。它为新式类 提供了更高效的封存机制。请参考 **PEP 307** 了解第 2 版协议带来的改进的相关信息。
- v3 版协议是在 Python 3.0 中引入的。它显式地支持*bytes* 字节对象，不能使用 Python 2.x 解封。这是 Python 3.0-3.7 的默认协议。
- v4 版协议添加于 Python 3.4。它支持存储非常大的对象，能存储更多种类的对象，还包括一些针对数据格式的优化。它是 Python 3.8 使用的默认协议。有关第 4 版协议带来改进的信息，请参阅 **PEP 3154**。

- 第 5 版协议是在 Python 3.8 中加入的。它增加了对带外数据的支持，并可加速带内数据处理。请参阅 **PEP 574** 了解第 5 版协议所带来的改进的详情。

> **备注**
>
> 序列化是一种比持久化更底层的概念，虽然*pickle* 读取和写入的是文件对象，但它不处理持久对象的命名问题，也不处理对持久对象的并发访问（甚至更复杂）的问题。*pickle* 模块可以将复杂对象转换为字节流，也可以将字节流转换为具有相同内部结构的对象。处理这些字节流最常见的做法是将它们写入文件，但它们也可以通过网络发送或存储在数据库中。*shelve* 模块提供了一个简单的接口，用于在 DBM 类型的数据库文件上封存和解封对象。

12.1.3 模块接口

要序列化某个包含层次结构的对象，只需调用*dumps()* 函数即可。同样，要反序列化数据流，可以调用*loads()* 函数。但是，如果要对序列化和反序列化加以更多的控制，可以分别创建*Pickler* 或*Unpickler* 对象。

pickle 模块包含了以下常量：

pickle.**HIGHEST_PROTOCOL**

整数，可用的最高协议版本。此值可以作为 协议值传递给*dump()* 和*dumps()* 函数，以及*Pickler* 的构造函数。

pickle.**DEFAULT_PROTOCOL**

整数，用于 pickle 数据的默认协议版本。它可能小于*HIGHEST_PROTOCOL*。当前默认协议是 v4，它在 Python 3.4 中首次引入，与之前的版本不兼容。

在 3.0 版本发生变更: 默认协议版本是 3。

在 3.8 版本发生变更: 默认协议版本是 4。

pickle 模块提供了以下方法，让封存过程更加方便：

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

将对象 *obj* 封存以后的对象写入已打开的*file object* *file*。它等同于 `Pickler(file, protocol).dump(obj)`。

参数 *file*、*protocol*、*fix_imports* 和 *buffer_callback* 的含义与它们在*Pickler* 的构造函数中的含义相同。

在 3.8 版本发生变更: 加入了 *buffer_callback* 参数。

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

将 *obj* 封存以后的对象作为*bytes* 类型直接返回，而不是将其写入到文件。

参数 *protocol*、*fix_imports* 和 *buffer_callback* 的含义与它们在*Pickler* 的构造函数中的含义相同。

在 3.8 版本发生变更: 加入了 *buffer_callback* 参数。

pickle.**load**(*file*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

从已打开的*file object* 文件中读取封存后的对象，重建其中特定对象的层次结构并返回。它相当于 `Unpickler(file).load()`。

Pickle 协议版本是自动检测出来的，所以不需要参数来指定协议。封存对象以外的其他字节将被忽略。

参数 *file*、*fix_imports*、*encoding*、*errors*、*strict* 和 *buffers* 的含义与它们在*Unpickler* 的构造函数中的含义相同。

在 3.8 版本发生变更: 加入了 *buffers* 参数。

pickle.**loads**(*data*, */*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

重建并返回一个对象的封存表示形式 *data* 的对象层级结构。*data* 必须为*bytes-like object*。

Pickle 协议版本是自动检测出来的，所以不需要参数来指定协议。封存对象以外的其他字节将被忽略。

参数 *fix_imports*, *encoding*, *errors*, *strict* 和 *buffers* 的含义与在*Unpickler* 构造器中的含义相同。

在 3.8 版本发生变更: 加入了 *buffers* 参数。

pickle 模块定义了以下 3 个异常:

exception pickle.**PickleError**

其他 pickle 异常的共同基类。它继承自*Exception*。

exception pickle.**PicklingError**

当*Pickler* 遇到无法解封的对象时将引发的错误。它继承自*PickleError*。

参考*可以被封存/解封的对象* 来了解哪些对象可以被封存。

exception pickle.**UnpicklingError**

当解封对象出现问题时将引发的错误，例如数据损坏或违反安全规则。它继承自*PickleError*。

注意，解封时可能还会抛出其他异常，包括（但不限于）AttributeError、EOFError、ImportError 和 IndexError。

pickle 模块包含了 3 个类，*Pickler*、*Unpickler* 和*PickleBuffer*:

class pickle.**Pickler**(*file*, *protocol=None*, *, *fix_imports=True*, *buffer_callback=None*)

它接受一个二进制文件用于写入 pickle 数据流。

可选参数 *protocol* 是一个整数，告知 pickler 使用指定的协议，可选择的协议范围从 0 到*HIGHEST_PROTOCOL*。如果没有指定，这一参数默认值为*DEFAULT_PROTOCOL*。指定一个负数就相当于指定*HIGHEST_PROTOCOL*。

参数 *file* 必须有一个 write() 方法，该 write() 方法要能接收字节作为其唯一参数。因此，它可以是一个打开的磁盘文件（用于写入二进制内容），也可以是一个*io.BytesIO* 实例，也可以是满足这一接口的其他任何自定义对象。

如果 *fix_imports* 为 True 且 *protocol* 小于 3，pickle 将尝试将 Python 3 中的新名称映射到 Python 2 中的旧模块名称，因此 Python 2 也可以读取封存的数据流。

如果 *buffer_callback* 为 None (默认值)，缓冲区视图将作为 pickle 流的一部分被序列化到 *file* 中。

如果 *buffer_callback* 不为 None，那它可以用缓冲区视图调用任意次。如果某次调用返回了假值 (例如 None)，则给定的缓冲区是*带外的*；在其他情况下缓冲区是带内的，例如在 pickle 流内部。

如果 *buffer_callback* 不为 None 且 *protocol* 为 None 或小于 5 则将出错。

在 3.8 版本发生变更: 加入了 *buffer_callback* 参数。

dump(*obj*)

将 *obj* 封存后的内容写入已打开的文件对象，该文件对象已经在构造函数中指定。

persistent_id(*obj*)

默认无动作，该方法可被子类重写。

如果*persistent_id()* 返回 None，*obj* 会被照常 pickle。如果返回其他值，*Pickler* 会将这个函数的返回值作为 *obj* 的持久化 ID（Pickler 本应得到序列化数据流并将其写入文件，若此函数有返回值，则得到此函数的返回值并写入文件）。这个持久化 ID 的解释应当定义在*Unpickler.persistent_load()* 中（该方法定义还原对象的过程，并返回得到的对象）。注意，*persistent_id()* 的返回值本身不能拥有持久化 ID。

参阅*持久化外部对象* 获取详情和使用示例。

在 3.13 版本发生变更: 在 Pickler 的 C 实现中添加此方法的默认实现。

dispatch_table

pickler 对象的 dispatch 表是对 *reduction* 函数的注册，其类别可使用*copyreg.pickle()* 来声明。它本身是一个以类为键并以 reduction 函数为值的映射。一个 reduction 函数接受单个参数即其所关联的类并应当遵循与*__reduce__()* 方法相同的接口。

Pickler 对象默认并没有*dispatch_table* 属性，该对象默认使用*copyreg* 模块中定义的全局 dispatch 表。如果要为特定 Pickler 对象自定义序列化过程，可以将*dispatch_table* 属性设置

为类字典对象（dict-like object）。另外，如果*Pickler* 的子类设置了*dispatch_table* 属性，则该子类的实例会使用这个表作为默认的 dispatch 表。

参阅*Dispatch 表* 获取使用示例。

Added in version 3.3.

reducer_override(*obj*)

可以在*Pickler* 子类中定义的特殊 reducer。该方法的优先级高于*dispatch_table* 中的任何 reducer。它应当遵循与*__reduce__()* 方法相同的接口，也可以选择返回*NotImplemented* 以回退到使用*dispatch_table* 注册的 reducer 来封存 `obj`。

参阅*类型，函数和其他对象的自定义归约* 获取详细的示例。

Added in version 3.8.

fast

已弃用。设为 True 则启用快速模式。快速模式禁用了“备忘录” (memo) 的使用，即不生成多余的 PUT 操作码来加快封存过程。不应将其与自指 (self-referential) 对象一起使用，否则将导致*Pickler* 无限递归。

如果需要进一步提高 pickle 的压缩率，请使用*pickletools.optimize()*。

class pickle.**Unpickler**(*file*, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

它接受一个二进制文件用于读取 pickle 数据流。

Pickle 协议版本是自动检测出来的，所以不需要参数来指定协议。

参数 *file* 必须有三个方法，read() 方法接受一个整数参数，readinto() 方法接受一个缓冲区作为参数，readline() 方法不需要参数，这与*io.BufferedIOBase* 里定义的接口是相同的。因此 *file* 可以是一个磁盘上用于二进制读取的文件，也可以是一个*io.BytesIO* 实例，也可以是满足这一接口的其他任何自定义对象。

可选的参数是 *fix_imports*, *encoding* 和 *errors*，用于控制由 Python 2 生成的 pickle 流的兼容性。如果 *fix_imports* 为 True，则 pickle 将尝试将旧的 Python 2 名称映射到 Python 3 中对应的新名称。*encoding* 和 *errors* 参数告诉 pickle 如何解码 Python 2 存储的 8 位字符串实例；这两个参数默认分别为'ASCII' 和'strict'。*encoding* 参数可置为'bytes' 来将这些 8 位字符串实例读取为字节对象。读取 NumPy array 和 Python 2 存储的*datetime*、*date* 和*time* 实例时，请使用 `encoding='latin1'`。

如果 *buffers* 为 `None` (默认值)，则反序列化所需的所有数据都必须包含在 pickle 流中。这意味着当*Pickler* 被实例化 (或当*dump()* 或*dumps()* 被调用) 时 *buffer_callback* 参数为 `None`。

如果 *buffers* 不为 `None`，则每次 pickle 流引用一个*带外的* 缓冲区视图时，消耗的对象都应该是一个启用缓冲区的对象的可迭代对象。这样的缓冲区将按顺序提供给 Pickler 对象的 *buffer_callback*。

在 3.8 版本发生变更: 加入了 *buffers* 参数。

load()

从构造函数中指定的文件对象里读取封存好的对象，重建其中特定对象的层次结构并返回。封存对象以外的其他字节将被忽略。

persistent_load(*pid*)

默认抛出*UnpicklingError* 异常。

如果定义了此方法，*persistent_load()* 应当返回持久化 ID *pid* 所指定的对象。如果遇到无效的持久化 ID，则应当引发*UnpicklingError*。

参阅*持久化外部对象* 获取详情和使用示例。

在 3.13 版本发生变更: 在 `Unpickler` 的 C 实现中添加了此方法的默认实现。

find_class(*module*, *name*)

如有必要，导入 *module* 模块并返回其中名叫 *name* 的对象，其中 *module* 和 *name* 参数都是*str* 对象。注意，不要被这个函数的名字迷惑，*find_class()* 同样可以用来导入函数。

子类可以重写此方法，来控制加载对象的类型和加载对象的方式，从而尽可能降低安全风险。参阅*限制全局变量* 获取更详细的信息。

引发一个审计事件 pickle.find_class 并附带参数 module, name。

class pickle.**PickleBuffer**(*buffer*)

缓冲区的包装器 (wrapper)，缓冲区中包含着可封存的数据。*buffer* 必须是一个 buffer-providing 对象，比如*bytes-like object* 或多维数组。

PickleBuffer 本身就可以生成缓冲区对象，因此可以将其传递给需要缓冲区生成器的其他 API，比如*memoryview*。

PickleBuffer 对象只能用 pickle 版本 5 及以上协议进行序列化。它们符合带外序列化 的条件。

Added in version 3.8.

raw()

返回该缓冲区底层内存区域的*memoryview*。返回的对象是一维的、C 连续布局的 memoryview，格式为 B (无符号字节)。如果缓冲区既不是 C 连续布局也不是 Fortran 连续布局的，则抛出*BufferError* 异常。

release()

释放由 PickleBuffer 占用的底层缓冲区。

12.1.4 可以被封存/解封的对象

下列类型可以被封存：

- 内置常量 (None, True, False, Ellipsis 和*NotImplemented*)；
- 整数、浮点数、复数；
- 字符串、字节串、字节数组；
- 只包含可封存对象的元组、列表、集合和字典；
- 可在模块最高层级上访问的（内置与用户自定义的）函数（使用 def，而不是使用 lambda 定义）；
- 可在模块最高层级上访问的类；
- 这种类的实例调用*__getstate__()* 的结果是可 pickle 的（请参阅封存类实例 一节了解详情）。

尝试封存不能被封存的对象会抛出*PicklingError* 异常，异常发生时，可能有部分字节已经被写入指定文件中。尝试封存递归层级很深的对象时，可能会超出最大递归层级限制，此时会抛出*RecursionError* 异常，可以通过*sys.setrecursionlimit()* 调整递归层级，不过请谨慎使用这个函数，因为可能会导致解释器崩溃。

请注意（内置与用户自定义的）函数是按完整*qualified name*，而不是按值来封存的。[2] 这意味着只会封存函数名称，以及包含它的模块和类名称。函数的代码，以及函数的属性都不会被封存。因而定义它的模块在解封环境中必须可以被导入，并且模块必须包含所命名的对象，否则将会引发异常。[3]

类似地，类也是按完整限定名称来封存的，因此在解封环境中也会应用相同的限制。请注意类的代码或数据都不会被封存，因此在下面的示例中类属性 attr 不会在解封环境中被恢复：

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

这些限制决定了为什么可封存的函数和类必须在一个模块的最高层级上定义。

类似的，在封存类的实例时，类的代码和数据不会随它们一起被封存，只有实际数据会被封存。这样设计有其目的，在将来修复类中的错误或给类增加方法之后仍然可以载入较早版本创建的对象。如果你打算长期使用某些可能有许多版本的类的对象，那么在对象中设置一个版本号以便通过类的*__setstate__()* 方法进行适当的转换就是值得做的事情。

[2] 这就是为什么 lambda 函数不可以被封存：所有的匿名函数都有同一个名字：<lambda>。
[3] 抛出的异常有可能是*ImportError* 或*AttributeError*，也可能是其他异常。

12.1.5 封存类实例

在本节中，我们描述了可用于定义、自定义和控制如何封存和解封类实例的通用流程。

在大多数情况下，使一个实例可被封存不需要任何额外的代码。根据默认设置，pickle 将通过内省来获取实例的类及属性。当一个类实例被解封时，它的 `__init__()` 方法通常 *不会* 被唤起。默认的行为会先创建一个未初始化的实例然后恢复已保存的属性。下面的代码展示了这种行为的具体实现:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def restore(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

类可以改变默认行为，只需定义以下一种或几种特殊方法:

`object.`**`__getnewargs_ex__`**`()`

对于使用第 2 版或更高版协议的 pickle，实现了*__getnewargs_ex__()* 方法的类可以控制在解封时传给 `__new__()` 方法的参数。本方法必须返回一对 `(args, kwargs)` 用于构建对象，其中 *args* 是表示位置参数的 tuple，而 *kwargs* 是表示命名参数的 dict。它们会在解封时传递给 `__new__()` 方法。

如果类的 `__new__()` 方法只接受关键字参数，则应当实现这个方法。否则，为了兼容性，更推荐实现*__getnewargs__()* 方法。

在 3.6 版本发生变更: *__getnewargs_ex__()* 现在可用于第 2 和第 3 版协议。

`object.`**`__getnewargs__`**`()`

这个方法与上一个*__getnewargs_ex__()* 方法类似，但仅支持位置参数。它要求返回一个 tuple 类型的 `args`，用于解封时传递给 `__new__()` 方法。

如果定义了*__getnewargs_ex__()*，那么*__getnewargs__()* 就不会被调用。

在 3.6 版本发生变更: 在 Python 3.6 前，第 2、3 版协议会调用*__getnewargs__()*，更高版本协议会调用*__getnewargs_ex__()*。

`object.`**`__getstate__`**`()`

类还可以通过重写方法*__getstate__()* 来进一步影响它们的实例要如何被封存。该方法将被调用并且其返回的对象会被当作实例的内容来封存，而不是使用默认状态。这有几种情况:

- 对于没有实例 `__dict__` 以及没有 `__slots__` 的类，默认状态为 `None`。
- 对于具有实例 `__dict__` 而没有 `__slots__` 的类，默认状态为 `self.__dict__`。
- 对于具有实例 `__dict__` 和 `__slots__` 的类，默认状态为一个由两个字典: `self.__dict__`、以及将槽位名称映射到槽位值的字典所组成的元组。只有包含具体值的槽位才会被包括在后一个字典当中。
- 对于具有 `__slots__` 而没有实例 `__dict__` 的类，默认状态为一个第一项是 `None` 而第二项是上述将槽位名称映射到槽位值的字典的元组。

在 3.11 版本发生变更: 将 `__getstate__()` 方法的默认实现添加到*object* 类中。

`object.`**`__setstate__`**`(`*state*`)`

当解封时，如果类定义了*__setstate__()*，就会在已解封状态下调用它。此时不要求实例的 state 对象必须是 dict。没有定义此方法的话，先前封存的 state 对象必须是 dict，且该 dict 内容会在解封时赋给新实例的 `__dict__`。

> **备注**
>
> 如果*__reduce__()* 在封存时返回一个 `None` 值状态，那么在解封时将不会调用*__setstate__()* 方法。

请参阅处理有状态的对象 一节如何使用 *__getstate__()* 和 *__setstate__()* 方法的更多信息。

> **备注**
>
> 在解封时，某些方法比如 `__getattr__()`，`__getattribute__()` 或 `__setattr__()` 可能会在实例上被调用。对于这些方法依赖于某些内部的不变量为真值的情况，类型应当实现 `__new__()` 以建立这样的不变量，因为当解封一个实例时 `__init__()` 并不会被调用。

正如我们会看到的，pickle 并不会直接使用上述的方法。实际上，这些方法是拷贝协议的一部分，它实现了 *__reduce__()* 特殊方法。拷贝协议提供了统一的接口用于在封存和拷贝对象时获取所需的数据。[4]

在你的类中直接实现 *__reduce__()* 虽然功能很强但也容易出错。因此，类的设计者应当尽可能使用高层级的接口 (即 *__getnewargs_ex__()*, *__getstate__()* 和 *__setstate__()*)。不过，我们仍然会演示使用 `__reduce__()` 是唯一选项或是更高效的封存或是两者兼有的场景。

`object.`**`__reduce__`**`()`

该接口当前定义如下。*__reduce__()* 方法不带任何参数，并且应返回字符串或最好返回一个元组 (返回的对象通常称为 “reduce 值”)。

如果返回字符串，该字符串会被当做一个全局变量的名称。它应该是对象相对于其模块的本地名称，pickle 模块会搜索模块命名空间来确定对象所属的模块。这种行为常在单例模式使用。

如果返回的是元组，则应当包含 2 到 6 个元素，可选元素可以省略或设置为 `None`。每个元素代表的意义如下：

- 一个可调用对象，该对象会在创建对象的最初版本时调用。
- 可调用对象的参数，是一个元组。如果可调用对象不接受参数，必须提供一个空元组。
- 可选元素，用于表示对象的状态，将被传给前述的 *__setstate__()* 方法。如果对象没有此方法，则这个元素必须是字典类型，并会被添加至 `__dict__` 属性中。
- 可选项，一个返回连续条目的迭代器 (而不是序列)。这些条目将使用 `obj.append(item)` 或是使用 `obj.extend(list_of_items)` 批量地添加到对象中。这主要用于列表的子类，但也可以用于其他类，只要它们具有使用相应签名的 *append* 和 *extend* 方法。(具体是使用 `append()` 还是 `extend()` 取决于所使用的 pickle 协议版本以及要插入的条目数量，所以这两个方法都必须被支持。)
- 可选元素，一个返回连续键值对的迭代器 (而不是序列)。这些键值对将会以 `obj[key] = value` 的方式存储于对象中。该元素主要用于 dict 子类，也可以用于那些实现了 `__setitem__()` 的类。
- 可选元素，一个带有 `(obj, state)` 签名的可调用对象。该可调用对象允许用户以编程方式控制特定对象的状态更新行为，而不是使用 `obj` 的静态 *__setstate__()* 方法。如果此处不是 `None`，则此可调用对象的优先级高于 `obj` 的 *__setstate__()*。

  Added in version 3.8: 新增了元组的第 6 项，可选元素 `(obj, state)`。

`object.`**`__reduce_ex__`**`(`*protocol*`)`

作为替代选项，也可以实现 *__reduce_ex__()* 方法。此方法的唯一不同之处在于它应接受一个整型参数用于指定协议版本。如果定义了这个函数，则会覆盖 *__reduce__()* 的行为。此外，*__reduce__()* 方法会自动成为扩展版方法的同义词。这个函数主要用于为以前的 Python 版本提供向后兼容的 reduce 值。

持久化外部对象

为了获取对象持久化的利益，*pickle* 模块支持引用已封存数据流之外的对象。这样的对象是通过一个持久化 ID 来引用的，它应当是一个由字母数字类字符组成的字符串 (对于第 0 版协议)[5] 或是一个任意对象 (用于任意新版协议)。

[4] *copy* 模块使用这一协议实现浅层 (shallow) 和深层 (deep) 复制操作。

[5] 对于字符数字类字符的限制是由于持久化 ID 在协议版本 0 中是由分行符来分隔的。因此如果持久化 ID 中出现了任何形式的分行符，封存结果就将变得无法读取。

pickle 模块不提供对持久化 ID 的解析工作，它将解析工作分配给用户定义的方法，分别是 pickler 中的*persistent_id()* 方法和 unpickler 中的*persistent_load()* 方法。

要通过持久化 ID 将外部对象封存，必须在 pickler 中实现*persistent_id()* 方法，该方法接受需要被封存的对象作为参数，返回一个 None 或返回该对象的持久化 ID。如果返回 None，该对象会被按照默认方式封存为数据流。如果返回字符串形式的持久化 ID，则会封存这个字符串并加上一个标记，这样 unpickler 才能将其识别为持久化 ID。

要解封外部对象，Unpickler 必须实现*persistent_load()* 方法，接受一个持久化 ID 对象作为参数并返回一个引用的对象。

下面是一个全面的例子，展示了如何使用持久化 ID 来封存外部对象。

```
# 介绍如何使用持久性 ID 基于引用
# 对外部对象进行 pickle 的简单示例。

import pickle
import sqlite3
from collections import namedtuple

# 代表数据库中一条记录的简单类。
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # 我们不是将 MemoRecord 作为常规类实例进行 pickle,
        # 而是发出一个持久性 ID。
        if isinstance(obj, MemoRecord):
            # 这里，我们的持久性 ID 就是一个元组，包含标签和键，
            # 它指向数据库中的特定记录。
            return ("MemoRecord", obj.key)
        else:
            # 如果 obj 没有持久性 ID，则返回 None。 这意味着 obj
            # 需要被正常地 pickle。
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # 此方法会在遇到一个持久性 ID 时被唤起。
        # 在这里，pid 是 DBPickler 所返回的元组。
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # 从数据库提取被引用的记录并将其返回。
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # 如果你无法返回正确的对象则总是引发一个错误。
            # 否则，反 pickle 操作将认为持久性 ID 所引用的对象
            # 为 None。
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
```

(续下页)

(接上页)

```
    import pprint

    # 初始化并填充我们的数据库。
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # 提取要进行 pickle 的记录。
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # 使用我们自定义的 DBPickler 保存记录。
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # 更新一条记录，以确保有效。
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # 从 pickle 数据流加载记录。
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

Dispatch 表

如果想对某些类进行自定义封存，而又不想在类中增加用于封存的代码，就可以创建带有特殊 dispatch 表的 pickler。

copyreg 模块所管理的全局 dispatch 表可通过 `copyreg.dispatch_table` 来访问。因此，可以选择使用经过修改的 `copyreg.dispatch_table` 副本作为私有 dispatch 表。

例如

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

创建了一个带有自有 dispatch 表的*pickle.Pickler* 实例，它可以对 `SomeClass` 类进行特殊处理。另外，下列代码

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

完成同样的操作，但所有 MyPickler 的实例都会共享一个私有分发表。另一方面，代码

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

会修改由 *copyreg* 模块的所有用户共享的全局分发表。

处理有状态的对象

下面的例子展示了如何修改类的封存行为。下面的 TextReader 类会打开一个文本文件，每次调用其 readline() 方法时将返回行号和该行的内容。如果一个 TextReader 实例被封存，则 除了文件对象以外的所有属性都会被保存。当实际被解封时，该文件将被重新打开，并从最后的位置开始恢复读取。实现此行为需要使用 __setstate__() 和 __getstate__() 方法。

```
class TextReader:
    """打印文本文件的行内容和行号。"""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # 从 self.__dict__ 拷贝对象的状态，其中包含
        # 所有的实例属性。 总是使用 dict.copy() 方法
        # 以避免修改原始状态。
        state = self.__dict__.copy()
        # 移除不可 pickle 的条目。
        del state['file']
        return state

    def __setstate__(self, state):
        # 恢复实例属性 (即 filename 和 lineno)。
        self.__dict__.update(state)
        # 恢复之前所打开文件的状态。 为做到这点，我们需要
        # 重新打开它并从其中读取直到恢复行计数。
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # 最后，保存该文件。
        self.file = file
```

使用方法如下所示：

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 类型，函数和其他对象的自定义归约

Added in version 3.8.

有时，*dispatch_table* 可能不够灵活。特别是当我们想要基于对象类型以外的其他规则来对封存进行定制，或是当我们想要对函数和类的封存进行定制的时候。

对于那些情况，可以子类化*Pickler* 类并实现*reducer_override()* 方法。此方法可返回任意 reduction 元组 (参见*__reduce__()*)。它也可以选择返回*NotImplemented* 以回退至传统的行为。

如果同时定义了*dispatch_table* 和*reducer_override()*，则*reducer_override()* 方法具有优先权。

备注

出于性能理由，可能不会为以下对象调用*reducer_override()*: None, True, False, 以及*int*, *float*, *bytes*, *str*, *dict*, *set*, *frozenset*, *list* 和*tuple* 的具体实例。

以下是一个简单的例子，其中我们允许封存并重新构建一个给定的类:

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """针对 MyClass 的自定义缩减器。"""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # 对于任何其他对象，回退为正常缩减操作
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

12.1.7 外部缓冲区

Added in version 3.8.

在某些场景中，*pickle* 模块会被用来传输海量的数据。因此，最小化内存复制次数以保证性能和节省资源是很重要的。但是*pickle* 模块的正常运作会将图类对象结构转换为字节序列流，因此在本质上就要从封存流中来回复制数据。

如果 *provider* (待传输对象类型的实现) 和 *consumer* (通信系统的实现) 都支持 pickle 第 5 版或更高版本所提供的外部传输功能，则此约束可以被撤销。

提供方 API

需要 pickle 的大型数据对象必须实现专门用于协议 5 以上版本的 *__reduce_ex__()* 方法，该方法将为任何大型数据返回一个 *PickleBuffer* 实例（而不是 *bytes* 对象）。

PickleBuffer 对象会 表明底层缓冲区可被用于外部数据传输。那些对象仍将保持与 *pickle* 模块的正常用法兼容。但是，使用方也可以选择告知 *pickle* 它们将自行处理那些缓冲区。

使用方 API

当序列化一个对象图时，通信系统可以启用对所生成 *PickleBuffer* 对象的定制处理。

发送端需要传递 *buffer_callback* 参数到 *Pickler* (或是到 *dump()* 或 *dumps()* 函数)，该回调函数将在封存对象图时附带每个所生成的 *PickleBuffer* 被调用。由 *buffer_callback* 所累积的缓冲区的数据将不会被拷贝到 pickle 流，而是仅插入一个简单的标记。

接收端需要传递 *buffers* 参数到 *Unpickler* (或是到 *load()* 或 *loads()* 函数)，其值是一个由缓冲区组成的可迭代对象，它会被传递给 *buffer_callback*。该可迭代对象应当按其被传递给 *buffer_callback* 时的顺序产生缓冲区。这些缓冲区将提供对象重构造器所期望的数据，对这些数据的封存产生了原本的 *PickleBuffer* 对象。

在发送端和接受端之间，通信系统可以自由地实现它自己用于外部缓冲区的传输机制。潜在的优化包括使用共享内存或基于特定数据类型的压缩等。

示例

下面是一个小例子，在其中我们实现了一个 *bytearray* 的子类，能够用于外部缓冲区封存:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer 当 pickle 协议 <= 4 时是禁用的。
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # 获取原始缓冲区对象的句柄
            obj = m.obj
            if type(obj) is cls:
                # 原始缓冲区对象是一个 ZeroCopyByteArray,
                # 则将其原样返回。
                return obj
            else:
                return cls(obj)
```

重构造器 (_reconstruct 类方法) 会在缓冲区的提供对象具有正确类型时返回该对象。在此小示例中这是模拟零拷贝行为的便捷方式。

在使用方，我们可以按通常方式封存那些对象，它们在反序列化时将提供原始对象的一个副本:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: 生成了副本
```

但是如果我们传入 *buffer_callback* 然后在反序列化时给回累积的缓冲区，我们就能够取回原始对象:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: 没有生成副本
```

这个例子受限于*bytearray* 会自行分配内存这一事实：你无法基于另一个对象的内存创建*bytearray* 的实例。但是，第三方数据类型例如 NumPy 数组则没有这种限制，允许在单独进程或系统间传输时使用零拷贝的封存（或是尽可能少地拷贝）。

参见

PEP 574 -- 带有外部数据缓冲区的 pickle 协议 5

12.1.8 限制全局变量

默认情况下，解封将会导入在 pickle 数据中找到的任何类或函数。对于许多应用来说，此行为是不可接受的，因为它会允许解封器导入并唤起任意代码。只须考虑当这个手工构建的 pickle 数据流被加载时会做什么:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

在这个例子里，解封器导入*os.system()* 函数然后应用字符串参数"echo hello world"。虽然这个例子不具攻击性，但是不难想象别人能够通过此方式对你的系统造成损害。

出于这样的理由，你可能会希望通过定制*Unpickler.find_class()* 来控制要解封的对象。与其名称所提示的不同，*Unpickler.find_class()* 会在执行对任何全局对象（例如一个类或一个函数）的请求时被调用。因此可以完全禁止全局对象或是将它们限制在一个安全的子集中。

下面的例子是一个解封器，它只允许某一些安全的来自*builtins* 模块的类被加载:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # 只允许来自 builtins 的安全的类。
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # 禁止任何其他的类。
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """类似于 pickle.loads() 的辅助函数。"""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

我们这个解封器完成其功能的一个示例用法:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

正如我们这个例子所显示的，对于允许解封的对象你必须要保持谨慎。因此如果要保证安全，你可以考虑其他选择例如*xmlrpc.client* 中的编组 API 或是第三方解决方案。

12.1.9 性能

较新版本的 pickle 协议（第 2 版或更高）具有针对某些常见特性和内置类型的高效二进制编码格式。此外，*pickle* 模块还拥有一个以 C 编写的透明优化器。

12.1.10 例子

对于最简单的代码，请使用*dump()* 和*load()* 函数。

```
import pickle

# pickle 所支持的任意对象集。
data = {
    'a': [1, 2.0, 3+4j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # 使用最高版本可用协议对 'data' 字典进行 pickle。
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

以下示例读取之前封存的数据。

```
import pickle

with open('data.pickle', 'rb') as f:
    # 将自动检测所使用的协议版本，因此我们
    # 不需要指定它。
    data = pickle.load(f)
```

参见

模块*copyreg*
为扩展类型提供 pickle 接口所需的构造函数。

模块*pickletools*
用于处理和分析已封存数据的工具。

模块*shelve*
带索引的数据库，用于存放对象，使用了*pickle* 模块。

模块*copy*
浅层 (shallow) 和深层 (deep) 复制对象操作

模块*marshal*
高效地序列化内置类型的数据。

备注

12.2 copyreg --- 注册 pickle 支持函数

源代码: Lib/copyreg.py

---

copyreg 模块提供了可在封存特定对象时使用的一种定义函数方式。*pickle* 和*copy* 模块会在封存/拷贝特定对象时使用这些函数。此模块提供了非类对象构造器的相关配置信息。这样的构造器可以是工厂函数或类实例。

copyreg.**constructor**(*object*)

将 *object* 声明为一个有效的构造器。如果 *object* 是不可调用的（因而不是一个有效的构造器）则会引发*TypeError*。

copyreg.**pickle**(*type*, *function*, *constructor_ob=None*)

声明 *function* 应当被用作 *type* 类型的对象的 “归约” 函数。*function* 必须返回一个字符串或包含二至六个元素的元组。请参阅*dispatch_table* 了解有关 *function* 的接口的更多细节。

constructor_ob 形参是一个旧式特性并且现在会被忽略，但如果传入则它必须为一个可调用对象。

请注意一个 pickler 或*pickle.Pickler* 的子类的*dispatch_table* 属性也可以被用来声明约归函数。

12.2.1 示例

以下示例将会显示如何注册一个封存函数，以及如何来使用它：

```
>>> import copyreg, copy, pickle
>>> class C:
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 shelve --- Python 对象持久化

源代码: Lib/shelve.py

”Shelf” 是一种持久化的类似字典的对象。与”dbm” 数据库的区别在于 Shelf 中的值（不是键！）实际上可以为任意 Python 对象 --- 即*pickle* 模块能够处理的任何东西。这包括大部分类实例、递归数据类型，以及包含大量共享子对象的对象。键则为普通的字符串。

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

打开一个持久化字典。filename 指定下层数据库的基准文件名。作为附带效果，会为 filename 添加一个扩展名并且可能创建更多的文件。默认情况下，下层数据库会以读写模式打开。可选的 *flag* 形参具有与*dbm.open()* *flag* 形参相同的含义。

在默认情况下，会使用以*pickle.DEFAULT_PROTOCOL* 创建的 pickle 来序列化值。pickle 协议的版本可通过 *protocol* 形参来指定。

由于 Python 语义的限制，Shelf 对象无法确定一个可变的持久化字典条目在何时被修改。默认情况下 只有在被修改对象再赋值给 shelf 时才会写入该对象 (参见示例)。如果可选的 *writeback* 形参设为 `True`，则所有被访问的条目都将在内存中被缓存，并会在*sync()* 和*close()* 时被写入；这可以使得对持久化字典中可变条目的修改更方便，但是如果访问的条目很多，这会消耗大量内存作为缓存，并会使得关闭操作变得非常缓慢，因为所有被访问的条目都需要写回到字典（无法确定被访问的条目中哪个是可变的，也无法确定哪个被实际修改了）。

在 3.10 版本发生变更: *pickle.DEFAULT_PROTOCOL* 现在会被用作默认的 pickle 协议。

在 3.11 版本发生变更: 接受*path-like object* 作为文件名。

备注

请不要依赖于 Shelf 的自动关闭功能；当你不再需要时应当总是显式地调用*close()*，或者使用*shelve.open()* 作为上下文管理器:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

警告

由于*shelve* 模块需要*pickle* 的支持，因此从不可靠的来源载入 shelf 是不安全的。与 pickle 一样，载入 Shelf 时可以执行任意代码。

Shelf 对象支持字典所支持的大多数方法和运算（除了拷贝、构造器以及 `|` 和 `|=` 运算符）。这样就能方便地将基于字典的脚本转换为要求持久化存储的脚本。

额外支持的两个方法:

Shelf.**sync**()

如果 Shelf 打开时将 *writeback* 设为*True* 则写回缓存中的所有条目。如果可行还会清空缓存并将持久化字典同步到磁盘。此方法会在使用*close()* 关闭 Shelf 时自动被调用。

Shelf.**close**()

同步并关闭持久化 *dict* 对象。对已关闭 Shelf 的操作将失败并引发*ValueError*。

参见

持久化字典方案 使用了广泛支持的存储格式并具有原生字典的速度。

12.3.1 限制

- 可选择使用哪种数据库包 (例如*dbm.ndbm* 或*dbm.gnu*) 取决于支持哪种接口。因此使用*dbm* 直接打开数据库是不安全的。如果使用了*dbm*，数据库同样会（不幸地）受限于它 --- 这意味着存储在数据库中的（封存形式的）对象尺寸应当较小，并且在少数情况下键冲突有可能导致数据库拒绝更新。
- *shelve* 模块不支持对 Shelf 对象的 *并发读/写访问*。（多个同时读取访问则是安全的。）当一个程序打开一个 shelve 对象来写入时，不应再有其他程序同时打开它来读取或写入。Unix 文件锁定可被用来解决此问题，但这在不同 Unix 版本上会存在差异，并且需要有关所用数据库实现的细节知识。

- 在 macOS 上 *dbm.ndbm* 会在更新时静默地破坏数据库文件，这将导致在尝试读取该数据库时发生硬崩溃。

class shelve.**Shelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

collections.abc.MutableMapping 的一个子类，它会将封存的值保存在 *dict* 对象中。

在默认情况下，会使用以 *pickle.DEFAULT_PROTOCOL* 创建的 pickle 来序列化值。pickle 协议的版本可通过 *protocol* 形参来指定。请参阅 *pickle* 文档来查看 pickle 协议的相关讨论。

如果 *writeback* 形参为 `True`，对象将为所有访问过的条目保留缓存并在同步和关闭时将它们写回到 *dict*。这允许对可变的条目执行自然操作，但是会消耗更多内存并让同步和关闭花费更长时间。

keyencoding 形参是在下层字典被使用之前用于编码键的编码格式。

Shelf 对象还可以被用作上下文管理器，在这种情况下它将在 `with` 语句块结束时自动被关闭。

在 3.2 版本发生变更: 添加了 *keyencoding* 形参；之前，键总是使用 UTF-8 编码。

在 3.4 版本发生变更: 添加了上下文管理器支持。

在 3.10 版本发生变更: *pickle.DEFAULT_PROTOCOL* 现在会被用作默认的 pickle 协议。

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

Shelf 的一个子类，它对外公开了 `first()`, `next()`, `previous()`, `last()` 和 `set_location()` 方法。这在来自 pybsddb 的第三方模块 `bsddb` 中可用，但在其他数据库模块中不可用。传给构造器的 *dict* 对象必须支持这些方法。这一般是通过调用 `bsddb.hashopen()`, `bsddb.btopen()` 或 `bsddb.rnopen()` 中的一个来完成的。可选的 *protocol*, *writeback* 和 *keyencoding* 形参具有与 *Shelf* 类的对应形参相同的含义。

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Shelf 的一个子类，它接受一个 *filename* 而非字典类对象。下层文件将使用 *dbm.open()* 来打开。默认情况下，文件将以读写模式打开。可选的 *flag* 形参具有与 *open()* 函数相同的含义。可选的 *protocol* 和 *writeback* 形参具有与 *Shelf* 类相同的含义。

12.3.2 示例

对接口的总结如下 (`key` 为字符串，`data` 为任意对象):

```
import shelve

d = shelve.open(filename)  # 打开 -- 文件可能带有低层级库
                           # 所添加的后缀

d[key] = data              # 将 data 存储到 key 位置 (如果使用现有的 key
                           # 则会覆盖旧数据)
data = d[key]              # 获取 key 位置上 data 的拷贝 (如果 key 不存在
                           # 则会引发 KeyError)
del d[key]                 # 删除 key 位置上的 data (如果 key 不存在
                           # 则会引发 KeyError)

flag = key in d            # 如果 key 存在则为真值
klist = list(d.keys())     # 由全部现有 key 组成的列表 (会很慢!)

# 由于 d 打开时未设置 writeback=True, 需要注意:
d['xx'] = [0, 1, 2]        # 这将符合预期，但是...
d['xx'].append(3)          # *这将不符合预期! * -- d['xx'] 仍为 [0, 1, 2]!

# 由于 d 打开时未设置 writeback=True, 你需要小心地编码:
temp = d['xx']             # 提取副本
temp.append(5)             # 修改副本
d['xx'] = temp             # 将副本存储回去，以使其持久化

# 或者，d=shelve.open(filename,writeback=True) 将允许你
# 编码 d['xx'].append(5) 并使其符合预期，但是它也会
```

(续下页)

(接上页)

```
# 消耗更多内存并使 d.close() 操作变慢。

d.close()                    # 关闭它
```

参见

模块dbm
dbm 风格数据库的泛型接口。

模块pickle
shelve 所使用的对象序列化。

12.4 marshal --- 内部 Python 对象序列化

此模块包含一些能以二进制格式来读写 Python 值的函数。这种格式是 Python 专属的，但是独立于特定的机器架构（即你可以在一台 PC 上写入某个 Python 值，将文件传到一台 Mac 上并在那里读取它）。这种格式的细节有意不带文档说明；可可能在不同 Python 版本之间发生改变（但这种情况极少发生）。[1]

这不是一个通用的“持久化”模块。对于通用的持久化以及通过 RPC 调用传递 Python 对象，请参阅*pickle* 和*shelve* 等模块。提供*marshal* 模块主要是为了支持读写 `.pyc` 形式“伪编译”代码的 Python 模块。因此，Python 维护者保留在必要时以不向下兼容的方式修改 marshal 格式的权利。代码对象的格式在 Python 版本之间不保证兼容，即使格式的版本是相同的。在不正确的 Python 版本中反序列化代码对象是未定义的行为。如果你要序列化和反序列化 Python 对象，请改用*pickle* 模块——具有类似的性能，保证版本独立性，并且 pickle 还支持比 marshal 更丰富种类的对象。

警告

marshal 模块对于错误或恶意构建的数据来说是不安全的。永远不要 unmarshal 来自不受信任的或未经验证的来源的数据。

并非所有 Python 对象类型都受到支持；通常，此模块只能写入和读取不依赖于特定 Python 调用发起方式的对象值。下列类型是受支持的：布尔对象、整数、浮点数、复数、字符串、字节串、字节数组、元组、列表、集合、冻结集合、字典以及代码对象（如果 *allow_code* 为真值），需要了解的一点是元组、列表、集合、冻结集合和字典只在其所包含的值也受支持时才是受支持的。单例对象*None*, *Ellipsis* 和*StopIteration* 也可以执行 marshall 和 unmarshall。对于 *version* 小于 3 的格式，将无法写入递归的列表、集合以及字典（见下文）。

有些函数可以读/写文件，还有些函数可以操作字节类对象。

这个模块定义了以下函数：

marshal.**dump**(*value*, *file*, *version=version*, */*, ***, *allow_code=True*)

向打开的文件写入值。值必须为受支持的类型。文件必须为可写的*binary file*。

如果值具有（或其包含的对象具有）不受支持的类型，则会引发*ValueError* 异常 --- 但还是会向文件写入垃圾数据。对象将不能使用*load()* 正确地重新读取。代码对象仅在 *allow_code* 为真值时受到支持。

version 参数指明 dump 应当使用的数据格式（见下文）。

引发一个审计事件 marshal.dumps 并附带参数 value, version。

在 3.13 版本发生变更: 增加了 *allow_code* 形参。

[1] 此模块的名称来源于 Modula-3 (及其他语言) 的设计者所使用的术语，他们使用术语”marshal” 来表示以自包含的形式传输数据。严格地说，将数据从内部形式转换为外部形式 (例如用于 RPC 缓冲区) 称为”marshal” 而其逆过程则称为”unmarshal”。

marshal.**load**(*file*, /, *, *allow_code=True*)

从打开的文件读取一个值并返回它。如果没有读取到有效的值（例如由于数据具有来自不同 Python 版本的不兼容的 marshal 格式），则会引发*EOFError*, *ValueError* 或*TypeError*。代码对象仅在 *allow_code* 为真值时受到支持。文件必须为可读的*binary file*。

引发一个不带参数的审计事件 marshal.load。

> **备注**
>
> 如果通过*dump()* marshal 了一个包含不受支持类型的对象，*load()* 将为不可 marshal 的类型替换 None。

在 3.10 版本发生变更: 使用此调用为每个代码对象引发一个 code.__new__ 审计事件。现在它会为整个载入操作引发单个 marshal.load 事件。

在 3.13 版本发生变更: 增加了 *allow_code* 形参。

marshal.**dumps**(*value*, *version=version*, /, *, *allow_code=True*)

返回将通过 dump(value, file) 写入到文件的字节串对象。值必须是受支持的类型。如果值具有（或其包含的对象具有）不受支持的类型则会引发*ValueError* 异常。代码对象仅在 *allow_code* 为真值时受到支持。

version 参数指明 dumps 应当使用的数据类型（见下文）。

引发一个审计事件 marshal.dumps 并附带参数 value, version。

在 3.13 版本发生变更: 增加了 *allow_code* 形参。

marshal.**loads**(*bytes*, /, *, *allow_code=True*)

将*bytes-like object* 转换为一个值。如果找不到有效的值，则会引发*EOFError*, *ValueError* 或*TypeError*。代码对象仅在 *allow_code* 为真值时受支持。输入的额外字节串会被忽略。

引发一个审计事件 marshal.loads 并附带参数 bytes。

在 3.10 版本发生变更: 使用此调用为每个代码对象引发一个 code.__new__ 审计事件。现在它会为整个载入操作引发单个 marshal.loads 事件。

在 3.13 版本发生变更: 增加了 *allow_code* 形参。

此外，还定义了以下常量:

marshal.**version**

指明模块所使用的格式。第 0 版为历史格式，第 1 版为共享固化的字符串，第 2 版对浮点数使用二进制格式。第 3 版添加了对于对象实例化和递归的支持。目前使用的为第 4 版。

备注

12.5 dbm --- Unix ”数据库” 接口

源代码: Lib/dbm/__init__.py

---

dbm 是一个针对多种 DBM 数据库的泛用接口:

- *dbm.sqlite3*
- *dbm.gnu*
- *dbm.ndbm*

如果未安装这些模块中的任何一种，则将使用*dbm.dumb* 模块中慢速但简单的实现。还有一个适用于 Oracle Berkeley DB 的 第三方接口。

exception dbm.**error**

一个元组，其中包含每个受支持的模块可引发的异常，另外还有一个名为*dbm.error* 的特殊异常作为第一项 --- 后者最在引发*dbm.error* 时被使用。

dbm.**whichdb**(*filename*)

此函数会尝试猜测几种简单数据库模块中哪一个是可用的 --- *dbm.sqlite3*, *dbm.gnu*, *dbm.ndbm* 或*dbm.dumb* --- 并应当被用来打开给定的文件。

返回下列值中的一个:

- 如果文件因其不可读或不存在而无法打开则返回 None
- 如果文件格式无法猜测则返回空字符串 ('')
- 包含所需模块名称的字符串，如 'dbm.ndbm' 或 'dbm.gnu'

在 3.11 版本发生变更: *filename* 接受一个*path-like object*。

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)

打开一个数据库并返回相应的数据库对象。

参数

- **file** (*path-like object*) -- 要打开的数据库文件。如果数据库文件已存在，则使用*whichdb()* 来确定其类型和要使用的适当模块；如果文件不存在，则会使用上述可导入子模块中的第一个。
- **flag** (str) --
  - 'r' (default): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) -- The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

在 3.11 版本发生变更: *file* 接受一个*path-like object*。

open() 所返回的对象支持与*dict* 相同的基本功能；可以存储、获取和删除键及其对应的值，并可使用 in 运算符和 keys() 方法，以及 get() 和 setdefault() 方法。

键和值总是被存储为*bytes*。这意味着当使用字符串时它们会在被存储之前隐式地转换至默认编码格式。

这些对象也支持在 with 语句中使用，当语句结束时将自动关闭它们。

在 3.2 版本发生变更: 现在 get() 和 setdefault() 方法对所有*dbm* 后端均可用。

在 3.4 版本发生变更: 向*open()* 所返回的对象添加了对上下文管理协议的原生支持。

在 3.8 版本发生变更: 从只读数据库中删除键将引发数据库模块专属的异常而不是*KeyError*。

以下示例记录了一些主机名和对应的标题，随后将数据库的内容打印出来。:

```
import dbm

# 打开数据库，如有必要则创建它。
with dbm.open('cache', 'c') as db:

    # 记录一些值
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # 请注意现在键被作为字节串。
    assert db[b'www.python.org'] == b'Python Website'
```

(续下页)

(接上页)

```
    # 可以看到值现在被作为字节串。
    assert db['www.cnn.com'] == b'Cable News Network'

    # 常用的字典接口方法同样可用。
    print(db.get('python.org', b'not present'))

    # 存储非字符串的键或值将引发异常
    # (通常为 TypeError)。
    db['www.yahoo.com'] = 4

# 当离开 with 语句时 db 将被自动关闭。
```

参见

模块 *shelve*
存储非字符串数据的持久化模块。

以下部分描述了各个单独的子模块。

12.5.1 dbm.sqlite3 --- 针对 dbm 的 SQLite 后端

Added in version 3.13.

源代码: Lib/dbm/sqlite3.py

此模块使用标准库 *sqlite3* 模块来提供针对 *dbm* 模块的 SQLite 后端。这样由 *dbm.sqlite3* 创建的文件可通过 *sqlite3*，或任何其他 SQLite 浏览器，包括 SQLite CLI 打开。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅 *WebAssembly 平台* 了解详情。

dbm.sqlite3.**open**(*filename*, */*, *flag='r'*, *mode=0o666*)

打开一个 SQLite 数据库。返回的对象行为类似于 *mapping*，实现了 `close()` 方法，并通过 `with` 关键字支持“关闭”上下文管理器。

参数

- **filename** (*path-like object*) -- 要打开的数据库的路径。
- **flag** (`str`) --
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **mode** -- 文件的 Unix 文件访问模式 (默认值: 八进制数 `0o666`)，仅在需要创建数据为时使用。

12.5.2 dbm.gnu --- GNU 数据库管理器

源代码: Lib/dbm/gnu.py

---

dbm.gnu 模块提供了针对 GDBM (GNU dbm) 库的接口，类似于 *dbm.ndbm* 模块，但带有额外的功能如对崩溃的容忍。

备注

由 *dbm.gnu* 和 *dbm.ndbm* 创建的文件格式是不兼容的因而无法互换使用。

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或 *WebAssembly 平台* 上不受支持。

exception dbm.gnu.**error**

针对 *dbm.gnu* 专属错误例如 I/O 错误引发。*KeyError* 的引发则针对一般映射错误例如指定了不正确的键。

dbm.gnu.**open**(*filename*, *flag='r'*, *mode=0o666*, */*)

打开 GDBM 数据库并返回一个 gdbm 对象。

参数

- **filename** (*path-like object*) -- 要打开的数据库文件。
- **flag** (str) --
  - 'r' (default): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.

  可以添加下列额外字符来控制数据库的打开方式:
  - 'f': 以快速模式打开数据库。对数据库的写入将不是同步的。
  - 's': 同步模式。对数据库的修改将立即写入到文件。
  - 'u': 不锁定数据库。

  并非所有旗标都对所有 GDBM 版本可用。请参阅 *open_flags* 成员获取受支持旗标字符的列表。
- **mode** (int) -- The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

抛出

error -- 如果传入了不可用的 *flag* 参数。

在 3.11 版本发生变更: *filename* 接受一个*path-like object*。

dbm.gnu.**open_flags**

由 *open()* 的 *flag* 形参所支持的字符组成的字符串。

gdbm 对象的行为类似于*映射*，但不支持 items() 和 values() 方法。还提供了以下方法:

gdbm.**firstkey**()

可以使用此方法和 *nextkey()* 方法循环遍历数据库中的每个键。遍历的顺序是按照 GDBM 的内部哈希值，而不会根据键的值排序。此方法将返回起始的键。

gdbm.**nextkey**(*key*)

在遍历中返回 *key* 之后的的下一个键。以下代码将打印数据库 db 中的每个键，而不会在内存中创建一个包含所有键的列表:

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**()

如果你进行了大量删除操作并且想要缩减 GDBM 文件所使用的空间，此例程可将可重新组织数据库。除非使用此重组功能否则 gdbm 对象不会缩减数据库文件大小；在其他情况下，被删除的文件空间将会保留并在添加新的 (键, 值) 对时被重用。

gdbm.**sync**()

当以快速模式打开数据库时，此方法会将任何未写入数据强制写入磁盘。

gdbm.**close**()

关闭 GDBM 数据库。

gdbm.**clear**()

从 GDBM 数据库移除所有条目。

Added in version 3.13.

12.5.3 dbm.ndbm --- 新数据库管理器

源代码: Lib/dbm/ndbm.py

---

dbm.ndbm 模块提供了对 NDBM (New Database Manager) 库的接口。此模块可与" 经典" NDBM 接口或 GDBM 兼容接口配合使用。

备注

由*dbm.gnu* 和*dbm.ndbm* 创建的文件格式是不兼容的因而无法互换使用。

警告

作为 macOS 的组成部分提供的 NDBM 库对值的大小有一个未写入文档的限制，当存储的值大于此限制时可能会导致数据库文件损坏。读取这种已损坏的文件可能会导致硬崩溃（段错误）。

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或*WebAssembly 平台* 上不受支持。

exception dbm.ndbm.**error**

针对*dbm.ndbm* 专属错误例如 I/O 错误引发。*KeyError* 的引发则针对一般映射错误例如指定了不正确的键。

dbm.ndbm.**library**

所使用的 NDBM 实现库的名称。

dbm.ndbm.**open**(*filename*, *flag='r'*, *mode=0o666*, */*)

打开 NDBM 数据库并返回一个 ndbm 对象。

参数

- **filename** (*path-like object*) -- 数据库文件的基本名（不带 .dir 或 .pag 扩展名）。
- **flag** (str) --
  - 'r' (default): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.

- **mode** (int) -- The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

ndbm 对象的行为类似于映射，但不支持 items() 和 values() 方法。还提供了以下方法：

在 3.11 版本发生变更: 接受*path-like object* 作为文件名。

ndbm.**close**()

关闭 NDBM 数据库。

ndbm.**clear**()

从 NDBM 数据库移除所有条目。

Added in version 3.13.

12.5.4 dbm.dumb --- 便携式 DBM 实现

源代码: Lib/dbm/dumb.py

备注

dbm.dumb 模块的目的是在更健壮的模块不可用时作为*dbm* 模块的最终回退项。*dbm.dumb* 不是为高速运行而编写的，也不像其他数据库模块一样被经常使用。

---

dbm.dumb 模块提供了一个完全以 Python 编写的持久化*dict* 型接口。不同于其他*dbm* 后端，例如*dbm.gnu*，它不需要外部库。

dbm.dumb 模块定义了以下对象：

exception dbm.dumb.**error**

针对*dbm.dumb* 专属错误例如 I/O 错误引发。*KeyError* 的引发则针对一般映射例如指定了不正确的键。

dbm.dumb.**open**(*filename*, *flag='c'*, *mode=0o666*)

打开一个 dbm.dumb 数据库。返回的数据库对象的行为类似于*mapping*，并额外提供*sync()* 和*close()* 等方法。

参数

- **filename** -- 数据库文件的基本名（不带扩展名）。新数据库将会创建以下文件: - *filename*.dat - *filename*.dir
- **flag** (str) --
  - 'r': Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c' (default): Open database for reading and writing, creating it if it doesn't exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) -- The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

警告

当载入包含足够巨大/复杂条目的数据库时有可能导致 Python 解释器的崩溃，这是由于 Python AST 编译器有栈深度限制。

在 3.5 版本发生变更: *open()* 在 *flag* 为 `'n'` 时将总是创建一个新数据库。

在 3.8 版本发生变更: 如果 *flag* 为 `'r'` 则打开的数据库将为只读的。如果 *flag* 为 `'r'` 或 `'w'` 则当数据库不存在时不会自动创建它。

在 3.11 版本发生变更: *filename* 接受一个*path-like object*。

在*collections.abc.MutableMapping* 类所提供的方法之外，还提供了以下方法：

dumbdbm.**sync**()
: 同步磁盘上的目录和数据文件。此方法将被*shelve.Shelf.sync()* 方法调用。

dumbdbm.**close**()
: 关闭数据库。

12.6 sqlite3 --- SQLite 数据库的 DB-API 2.0 接口

源代码： Lib/sqlite3/ SQLite 是一个 C 语言库，它可以提供一种轻量级的基于磁盘的数据库，这种数据库不需要独立的服务器进程，也允许需要使用一种非标准的 SQL 查询语言来访问它。一些应用程序可以使用 SQLite 作为内部数据存储。可以用它来创建一个应用程序原型，然后再迁移到更大的数据库，比如 PostgreSQL 或 Oracle。

`sqlite3` 模块由 Gerhard Häring 编写。它提供了 **PEP 249** 所描述的符合 DB-API 2.0 规范的 SQL 接口，并要求使用 SQLite 3.15.2 或更新的版本。

本文档包括了四个主要部分：

- *教程* 将教你如何使用 `sqlite3` 模块。
- *参考* 描述了该模块定义的类与函数。
- *常用方案指引* 详细介绍了如何处理一些特定的任务。
- *说明* 提供了关于事务控制（transaction control）的更深一步的背景。

参见

https://www.sqlite.org
: SQLite 的主页；它的文档详细描述了它所支持的 SQL 方言的语法和可用的数据类型。

https://www.w3schools.com/sql/
: 学习 SQL 语法的教程、参考和例子。

PEP 249 - DB-API 2.0 规范
: PEP 由 Marc-André Lemburg 撰写。

12.6.1 教程

在本篇教程中，你将会使用 `sqlite3` 模块的基本功能创建一个存储 Monty Python 的电影作品信息的数据库。本篇教程假定您在阅读前对于数据库的基本概念有所了解，例如 cursors 与 transactions 。

首先，我们需要创建一个新的数据库并打开一个数据库连接以允许 `sqlite3` 通过它来动作。调用*sqlite3.connect()* 来创建与当前工作目录下 `tutorial.db` 数据库的连接，如果它不存在则会隐式地创建它：

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

上面的代码中，返回的*Connection* 对象 `con` 代表一个与在磁盘上的数据库（on-disk databse）的连接。

为了执行 SQL 语句并且从 SQL 查询中取得结果，我们需要使用游标 (cursor) 。在下面的代码中，我们调用函数*con.cursor()* 创建了一个游标 (*Cursor*)：

```
cur = con.cursor()
```

通过上面的操作，我们已经得到了与数据库的连接 (connection) 与游标 (cursor)，现在我们便可以在数据库中创建一张名为 movie 的表了，它包括电影名（title，在下方代码中对应“title”）、上映年份（release year，在下方代码中对应“year”）以及电影评分（review score，在下方代码中对应“score”）这三列。在本篇教程中，出于简洁的考虑，我们在创建表的 SQL 语句声明中只列出表头名 (column names)，而没有像一般的 SQL 语句那样同时声明数据列的对应数据类型——这一点得益于 SQLite 的 flexible typing 特性，它使得我们在使用 SQLite 时，指明数据类型这一项工作时可选的。如下面的代码所示，我们通过调用函数*cur.excute(...)* 执行创建表格的 CREATE TABLE 语句：

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

我们可以通过查询 SQLite 内置的 sqlite_matser 表以验证新表是否已经创建，本例中，此时该表应该已经包括了一条 movie 的表定义（更多内容请参考 The Schema Table）。下面的代码将通过调用函数*cur.excute(...)* 执行查询，把结果赋给 res，而后调用*res.fetchone()* 获取结果行：

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

我们可以看到表已被创建，因为查询结果返回了一个包含表名的*tuple*。如果我们在 sqlite_master 中查询一个不存在的表 spam，则 res.fetchone() 将返回 None：

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
>>> res.fetchone() is None
True
```

现在，让我们再次调用*cur.execute(...)* 去添加由 SQL 字面量 (literals) 提供的两行数据：

```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

INSERT 语句将隐式地创建一个事务 (transaction)，事务需要在将更改保存到数据库前提交（更多细节请参考事务控制）。我们通过在一个连接对象（本例中为 con）上调用*con.commit()* 提交事务：

```
con.commit()
```

我们可以通过执行一个 SELECT 查询以验证数据是否被正确地插入表中。下面的代码中，我们使用我们已经很熟悉的函数*cur.execute(...)* 将查询结果赋给 res，而后调用*res.fetchall()* 返回所有的结果行：

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

上面的代码中，结果是一个包含了两个元组 (tuple) 的列表 (*list*)，其中每一个元组代表一个数据行，每个数据行都包括该行的 score 值。

现在，让我们调用*cur.executemany(...)* 再插入三行数据：

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # 记得在执行 INSERT 之后提交事务。
```

请注意，占位符 (placeholders) `?` 是用来在查询中绑定数据 `data` 的。在绑定 Python 的值到 SQL 语句中时，请使用占位符取代格式化字符串 (string formatting) 以避免 SQL 注入攻击 （更多细节请参见*如何在 SQL 查询中使用占位符来绑定值*)。

同样的，我们可以通过执行 `SELECT` 查询验证新的数据行是否已经插入表中，这一次我们将迭代查询的结果：

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

如上可见，每一行都是包括 `(year,title)` 这两个元素的元组 (*tuple*)，它与我们查询中选中的数据列相匹配。

最后，让我们先通过调用 *con.close()* 关闭现存的与数据库的连接，而后打开一个新的连接、创建一个新的游标、执行一个新的查询以验证我们是否将数据库写入到了本地磁盘上：

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r}, released in {year}')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail', released in 1975
>>> new_con.close()
```

现在您已经成功地使用模块 `sqlite3` 创建了一个 SQLite 数据库，并且学会了以多种方式往其中插入数据与检索值。

参见

- 阅读*常用方案指引* 以获取更多信息：
  - *如何在 SQL 查询中使用占位符来绑定值*
  - *如何将自定义 Python 类型适配到 SQLite 值*
  - *如何将 SQLite 值转换为自定义 Python 类型*
  - *如何使用连接上下文管理器*
  - *如何创建并使用行工厂对象*
- 参阅*说明* 以获取关于事务控制的更深一步的背景。

12.6.2 参考

模块函数

sqlite3.**connect**(*database*, *timeout=5.0*, *detect_types=0*, *isolation_level='DEFERRED'*, *check_same_thread=True*, *factory=sqlite3.Connection*, *cached_statements=128*, *uri=False*, ***, *autocommit=sqlite3.LEGACY_TRANSACTION_CONTROL*)

打开一个与 SQLite 数据库的连接。

参数

- **database** (*path-like object*) -- 要撕开的数据库文件的路径。你可以传入 `":memory:"` 来创建一个 仅存在于内存中的 SQLite 数据库，并打开它的一个连接。

- **timeout** (float) -- 当一个表被锁定时连接在最终引发*OperationalError* 之前应该等待多少秒。如果另一个链接开启了一个事务来修改一个表，该表将被锁定直到该事务完成提交。默认值为五秒。
- **detect_types** (int) -- 控制是否以及如何查找要转换为 Python 类型的非*SQLite 原生支持的* 数据类型，将使用通过*register_converter()* 注册的转换器。将它设置为*PARSE_DECLTYPES* 和*PARSE_COLNAMES* 的任意组合 (使用 |，即按位或) 来启用此选项。如果这两个旗标都已设置则列名将优先于声明的类型。当为默认值 (0) 时，类型检测将被禁用。
- **isolation_level** (str | *None*) -- 控制旧式的事务处理行为。更多信息请参阅*Connection.isolation_level* 和*通过 isolation_level 属性进行事务控制*。可以为 "DEFERRED" (默认值)、"EXCLUSIVE" 或 "IMMEDIATE"；或者为 None 表示禁止隐式地开启事务。除非*Connection.autocommit* 设为*LEGACY_TRANSACTION_CONTROL* (默认值) 否则没有任何影响。
- **check_same_thread** (bool) -- 如果为 True (默认)，则*ProgrammingError* 将在数据库连接被它的创建者以外的线程使用时被引发。如果为 False，则连接可以在多个线程中被访问；写入操作需要由用户者进行序列化以避免数据损坏。请参阅*threadsafety* 了解详情。
- **factory** (Connection) -- 如果您不想使用默认的*Connection* 类创建连接，那么您可以通过传入一个自定义的*Connection* 类的子类给该参数以创建连接。
- **cached_statements** (int) -- 该参数指明 sqlite3 模块应该为该连接进行内部缓存的语句 (statements) 数量。默认情况下，它的值为 128。
- **uri** (bool) -- 如果将该参数的值设置为 True，参数 *database* 将会被解释为一个由文件路径与可选的查询字符串组成的 URI (Uniform Resource Identifier) 链接。链接的前缀协议部分 (schema part) 必需是 "file:"，后面的文件路径可以是相对路径或绝对路径。查询字符串允许向 SQLite 传递参数，以实现不同的*如何使用 SQLite URI*。
- **autocommit** (bool) -- 控制 **PEP 249** 事务处理行为。更多信息参见*Connection.autocommit* 和*通过 autocommit 属性进行事务控制*。*autocommit* 目前默认值为*LEGACY_TRANSACTION_CONTROL*。在未来的 Python 版本中默认值将变为 False。

返回类型
Connection

引发一个*审计事件* sqlite3.connect 并附带参数 database。

引发一个*审计事件* sqlite3.connect/handle 并附带参数 connection_handle。

在 3.4 版本发生变更: 增加了 *uri* 参数。

在 3.7 版本发生变更: *database* 现在可以是一个*path-like object* 对象了，而不仅仅是字符串。

在 3.10 版本发生变更: 增加了 sqlite3.connect/handle 审计事件。

在 3.12 版本发生变更: 增加了 *autocommit* 形参。

在 3.13 版本发生变更: 形参 *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, 和 *uri* 用作为位置参数做法已被弃用。它们将在 Python 3.15 中成为限仅关键字形参。

sqlite3.**complete_statement**(*statement*)

如果传入的字符串语句 (statement) 看起来像是包括一条或多条完整的 SQL 语句，那么该函数将返回 True 。请注意，除了检查未封闭的字符串字面 (unclosed string literals) 以及语句是否以分号结束外，它不会执行任何的语法检查 (syntactic verification) 与语法解析 (synatatic parsing) 。

例如:

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

该函数可能在这样的情形下非常有用：在通过命令行 (command-line) 输入数据时，可使用该函数判断输入文本是否可以构成一个完成的 SQL 语句，或者判断在调用函数*execute()* 前是否还需要额外的输入。

请参阅 Lib/sqlite3/__main__.py 中的 `runsource()` 了解实际使用情况。

sqlite3.**enable_callback_tracebacks**(*flag*, */*)

是否启用回调回溯 (callback tracebacks)。默认情况下，在 SQLite 中，您不会在用户定义的函数、聚合函数 (aggregates)、转换函数 (converters)、验证回调函数 (authorizer callbacks) 等中得到任何回溯信息。如果您想调试它们，您可以在将形式参数 *flag* 设置为 `True` 的情况下调用该函数。之后您便可以从*sys.stderr* 的回调中得到回溯信息。使用 `False` 将再次禁用该功能。

备注

用户自定义函数回调中的错误将被记录为不可引发的异常。请使用*不可引发的钩子处理器* 执行对失败回调的内省。

sqlite3.**register_adapter**(*type*, *adapter*, */*)

注册 *adapter* *callable* 以将 Python 类型 *type* 适配为一个 SQLite 类型。该适配器在调用时会传入一个 *type* 类型的 Python 对象作为其唯一参数，并且必须返回一个*SQLite 原生支持的类型* 的值。

sqlite3.**register_converter**(*typename*, *converter*, */*)

注册 *converter* *callable* 以将 *typename* 类型的 SQLite 对象转换为一个特定类型的 Python 对象。转换器会针对所有类型为 *typename* 的 SQLite 值唤起；它会传递一个*bytes* 对象并且应该返回一个所需的 Python 类型的对象。请参阅*connect()* 的 *detect_types* 形参了解有关类型检测工作方式的详情。

注：*typename* 以及您在查询中使用的类型名是不大小写敏感的。

模块常量

sqlite3.**LEGACY_TRANSACTION_CONTROL**

将*autocommit* 设为该常量以选择旧式（Python 3.12 之前）事务控制行为。更多信息请参阅*通过 isolation_level 属性进行事务控制*。

sqlite3.**PARSE_DECLTYPES**

将这个旗标值传递给*connect()* 的 *detect_types* 形参，以使用创建数据库表时为每列声明的类型的查找转换器函数。`sqlite3` 将使用声明类型的第一个单词作为转换字典键来查找转换函数。例如：

```
CREATE TABLE test(
   i integer primary key,  ! 将查找名为 "integer" 的转换器
   p point,                ! 将查找名为 "point" 的转换器
   n number(10)            ! 将查找名为 "number" 的转换器
)
```

此旗标可以使用 `|`（位或）运算符与*PARSE_COLNAMES* 组合。

备注

生成的字段 (例如 `MAX(p)`) 将作为*str* 返回。使用 `PARSE_COLNAMES` 为这样的查询设置类型。

sqlite3.**PARSE_COLNAMES**

将这个旗标值传递给*connect()* 的 *detect_types* 形参以使用类型名称来查找转换器函数，类型名称

解析自查询列名，将作为转换器字典键。查询列名必须包装在双引号 (") 中而类型名称必须包装在方括号 ([]) 中。

```
SELECT MAX(p) as "p [point]" FROM test;  ! 将查找转换器 "point"
```

此旗标可以使用 | （位或）运算符与*PARSE_DECLTYPES* 组合。

sqlite3.**SQLITE_OK**

sqlite3.**SQLITE_DENY**

sqlite3.**SQLITE_IGNORE**

应当由传给*Connection.set_authorizer()* 的 *authorizer_callback* *callable* 返回的旗标，用于指明是否:

- 访问被允许（SQLITE_OK）。
- SQL 语句伴异常的执行失败（SQLITE_DENY）。
- 该列应被视为 NULL（SQLITE_IGNORE）。

sqlite3.**apilevel**

指明所支持的 DB-API 级别的字符串常量。根据 DB-API 的需要设置。硬编码为 "2.0"。

sqlite3.**paramstyle**

指明 sqlite3 模块所预期的形参标记格式化类型。根据 DB-API 的需要设置。硬编码为 "qmark"。

备注

named DB-API 形参风格也受到支持。

sqlite3.**sqlite_version**

以字符串 表示的运行时 SQLite 库版本号。

sqlite3.**sqlite_version_info**

以整数 *tuple* 表示的运行时. SQLite 库版本号。

sqlite3.**threadsafety**

DB-API 2.0 所要求的整数常量，指明 sqlite3 模块支持的线程安全级别。该属性将基于编译下层 SQLite 库所使用的默认 线程模式 来设置。SQLite 的线程模式有:

1. **Single-thread**: 在此模式下，所有的互斥都被禁用并且 SQLite 同时在多个线程中使用将是不安全的。
2. **Multi-thread**: 在此模式下，只要单个数据库连接没有被同时用于两个或多个线程之中 SQLite 就可以安全地被多个线程所使用。
3. **Serialized**: 在序列化模式下，SQLite 可以安全地被多个线程所使用而没有额外的限制。

从 SQLite 线程模式到 DB-API 2.0 线程安全级别的映射关系如下:

SQLite 线程模式	thread-safety	SQLITE_THREA	DB-API 2.0 含义
single-thread	0	0	各个线程不能共享模块
multi-thread	1	2	线程可以共享模块，但不能共享连接
serialized	3	1	线程可以共享模块、连接和游标 Threads may share the module, connections and cursors

在 3.11 版本发生变更: 动态设置 *threadsafety* 而不是将其硬编码为 1。

sqlite3.**version**

此模块字符串 形式的版本号。这不是 SQLite 库的版本号。

Deprecated since version 3.12, will be removed in version 3.14: 这个常量原本是用于反映 pysqlite 包的版本号，它是一个用于对 sqlite3 进行上游修改的第三方库。如今它已不具任何意义或实用价值。

sqlite3.**version_info**

此模块整数 *tuple* 形式的版本号。这不是 SQLite 库的版本号。library.

Deprecated since version 3.12, will be removed in version 3.14: 这个常量原本是用于反映 pysqlite 包的版本号，它是一个用于对 sqlite3 进行上游修改的第三方库。如今它已不具任何意义或实用价值。

sqlite3.**SQLITE_DBCONFIG_DEFENSIVE**

sqlite3.**SQLITE_DBCONFIG_DQS_DDL**

sqlite3.**SQLITE_DBCONFIG_DQS_DML**

sqlite3.**SQLITE_DBCONFIG_ENABLE_FKEY**

sqlite3.**SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER**

sqlite3.**SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION**

sqlite3.**SQLITE_DBCONFIG_ENABLE_QPSG**

sqlite3.**SQLITE_DBCONFIG_ENABLE_TRIGGER**

sqlite3.**SQLITE_DBCONFIG_ENABLE_VIEW**

sqlite3.**SQLITE_DBCONFIG_LEGACY_ALTER_TABLE**

sqlite3.**SQLITE_DBCONFIG_LEGACY_FILE_FORMAT**

sqlite3.**SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE**

sqlite3.**SQLITE_DBCONFIG_RESET_DATABASE**

sqlite3.**SQLITE_DBCONFIG_TRIGGER_EQP**

sqlite3.**SQLITE_DBCONFIG_TRUSTED_SCHEMA**

sqlite3.**SQLITE_DBCONFIG_WRITABLE_SCHEMA**

这些常量被用于*Connection.setconfig()* 和*getconfig()* 方法。

这些常量的可用性会根据 Python 编译时使用的 SQLite 版本而发生变化。

Added in version 3.12.

参见

https://www.sqlite.org/c3ref/c_dbconfig_defensive.html
SQLite 文档：数据库连接配置选项

连接对象

class sqlite3.**Connection**

每个打开的 SQLite 数据库均以 Connection 对象来表示，这种对象是使用*sqlite3.connect()* 创建的。它们的主要目的是创建*Cursor* 对象，以及事务控制。

参见

- 如何使用连接快捷方法
- 如何使用连接上下文管理器

在 3.13 版本发生变更: 如果未在 Connection 对象被删除前调用*close()* 则会发出*ResourceWarning*。

SQLite 数据库连接对象有如下的属性和方法：

cursor(*factory=Cursor*)

创建并返回*Cursor* 对象。cursor 方法接受一个可选参数 *factory*。如果提供了这个参数，它必须是一个*callable* 并且返回*Cursor* 或其子类的实例。

blobopen(*table*, *column*, *row*, */*, ***, *readonly=False*, *name='main'*)

打开一个已有的 BLOB（二进制大型对象） *Blob* 句柄。

参数

- **table** (str) -- 二进制大对象 blob 所在表的名称。
- **column** (str) -- 二进制大对象 blob 所在表的列名。
- **row** (str) -- 二进制大对象 blob 所在的列名。
- **readonly** (bool) -- 如果 blob 应当不带写入权限打开则设为 True。默认为 False。
- **name** (str) -- 二进制大对象 blob 所在的数据库名。默认为 "main"。

抛出

OperationalError -- 当尝试打开 WITHOUT ROWID 的表中的某个 blob 时。

返回类型

Blob

> **备注**
>
> blob 的大小无法使用*Blob* 类来修改。可使用 SQL 函数 zeroblob 来创建固定大小的 blob。

Added in version 3.11.

commit()

向数据库提交任何待处理事务。如果*autocommit* 为 True，或者没有已开启的事务，则此方法不会做任何操作。如果 autocommit 为 False，则如果有一个待处理事务被此方法提交则会隐式地开启一个新事务。

rollback()

回滚到任何待处理事务的起始位置。如果*autocommit* 为 True，或者没有已开启的事务，则此方法不会做任何操作。如果:attr:*!autocommit* 为 False，则如果此方法回滚了一个待处理事务则会隐式地开启一个新事务。

close()

关闭数据库连接。如果*autocommit* 为 False，则任何待处理事务都会被隐式地回滚。如果 autocommit 为 True 或*LEGACY_TRANSACTION_CONTROL*，则不会执行隐式的事务控制。请确保在关闭之前*commit()* 以避免丢失待处理的更改。

execute(*sql*, *parameters=()*, */*)

创建一个新的*Cursor* 对象，并在其上使用给出的 *sql* 和 *parameters* 调用*execute()*。返回新的游标对象。

executemany(*sql*, *parameters*, */*)

创建一个新的*Cursor* 对象，并在其上使用给出的 *sql* 和 *parameters* 调用*executemany()*。返回新的游标对象。

executescript(*sql_script*, */*)

创建一个新的*Cursor* 对象，并在其上使用给出的 *sql_script* 调用*executescript()*。返回新的游标对象。

create_function(*name*, *narg*, *func*, ***, *deterministic=False*)

创建或移除用户定义的 SQL 函数。

参数

- **name** (str) -- SQL 函数的名称。
- **narg** (int) -- SQL 函数可接受的参数数量，如果是 -1，则该函数可以接受任意数量的参数。
- **func** (*callback* | None) -- 当该 SQL 函数被唤起时将会调用的*callable*。该可调用对象必须返回一个 *SQLite 原生支持的类型*。设为 None 将移除现有的 SQL 函数。
- **deterministic** (bool) -- 如为 True，创建的 SQL 函数将被标记为 deterministic，这允许 SQLite 执行额外的优化。

在 3.8 版本发生变更: 增加了 *deterministic* 形参。

示例:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
>>> con.close()
```

在 3.13 版本发生变更: 将 *name*, *narg* 和 *func* 作为关键字参数传入的做法已被弃用。这些形参将在 Python 3.15 中成为仅限位置形参。

create_aggregate(*name*, *n_arg*, *aggregate_class*)

创建或移除用户自定义的 SQL 聚合函数。

参数

- **name** (str) -- SQL 聚合函数的名称。
- **n_arg** (int) -- SQL 聚合函数可接受的参数数量。如为 -1，则可以接受任意数量的参数。
- **aggregate_class** (*class* | None) -- 一个类必须实现下列方法: * step(): 向聚合添加一行。* finalize(): 将聚合的最终结果作为一个 *SQLite 原生支持的类型* 返回。step() 方法需要接受的参数数量是由 *n_arg* 控制的。设为 None 将移除现有的 SQL 聚合函数。

示例:

```
class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.execute("CREATE TABLE test(i)")
cur.execute("INSERT INTO test(i) VALUES(1)")
cur.execute("INSERT INTO test(i) VALUES(2)")
cur.execute("SELECT mysum(i) FROM test")
print(cur.fetchone()[0])

con.close()
```

在 3.13 版本发生变更: 将 *name*, *n_arg* 和 *aggregate_class* 作为关键字参数传入的做法已被弃用。这些形参将在 Python 3.15 中成为仅限位置形参。

create_window_function(*name*, *num_params*, *aggregate_class*, /)

创建或移除用户定义的聚合窗口函数。

参数

- **name** (str) -- 要创建或移除的 SQL 聚合窗口函数的名称。
- **num_params** (int) -- SQL 聚合窗口函数可接受的参数数量。如为 `-1`，则可以接受任意数量的参数。
- **aggregate_class** (*class* | None) -- 一个必须实现下列方法的类: * `step()`: 向当前窗口添加一行。* `value()`: 返回聚合的当前值。* `inverse()`: 从当前窗口移除一行。* `finalize()`: 将聚合的最终结果作为一个 *SQLite 原生支持的类型* 返回。`step()` 和 `value()` 方法需要接受的参数数量是由 *num_params* 控制的。设为 `None` 将移除现有的 SQL 聚合窗口函数。

抛出

NotSupportedError -- 如果在早于 SQLite 3.25.0，不支持聚合窗口函数的版本上使用。

Added in version 3.11.

示例:

```
# 来自 https://www.sqlite.org/windowfunctions.html#udfwinfunc 的示例
class WindowSumInt:
    def __init__(self):
        self.count = 0

    def step(self, value):
        """添加一行到当前窗口。"""
        self.count += value

    def value(self):
        """返回聚合的当前值。"""
        return self.count

    def inverse(self, value):
        """从当前窗口移除一行。"""
        self.count -= value

    def finalize(self):
        """返回聚合的最终值。

        任何清理动作都应放在此处。
        """
        return self.count


con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE test(x, y)")
values = [
    ("a", 4),
    ("b", 5),
    ("c", 3),
    ("d", 8),
    ("e", 1),
]
cur.executemany("INSERT INTO test VALUES(?, ?)", values)
con.create_window_function("sumint", 1, WindowSumInt)
cur.execute("""
```

(续下页)

(接上页)

```
    SELECT x, sumint(y) OVER (
        ORDER BY x ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS sum_y
    FROM test ORDER BY x
""")
print(cur.fetchall())
con.close()
```

create_collation(*name*, *callable*, */*)

使用排序函数 *callable* 创建一个名为 *name* 的排序规则。*callable* 被传递给两个字符串 参数，并且它应该返回一个整数。

- 如果前者的排序高于后者则为 `1`
- 如果前者的排序低于于后者则为 `-1`
- 如果它们的顺序相同则为 `0`

下面的例子显示了一个反向排序的排序方法:

```
def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.execute("CREATE TABLE test(x)")
cur.executemany("INSERT INTO test(x) VALUES(?)", [("a",), ("b",)])
cur.execute("SELECT x FROM test ORDER BY x COLLATE reverse")
for row in cur:
    print(row)
con.close()
```

通过将 *callable* 设为 `None` 来移除一个排序规则函数。

在 3.11 版本发生变更: 排序规则的名称可以包含任意 Unicode 字符。在之前，只允许 ASCII 字符。

interrupt()

从其他的线程调用此方法以中止可能正在连接上执行的任何查询。被中止的查询将引发*OperationalError*。

set_authorizer(*authorizer_callback*)

注册*callable* *authorizer_callback* 用于在每次尝试访问数据库中表的某一列时被唤起。该回调应当返回*SQLITE_OK*、*SQLITE_DENY* 或*SQLITE_IGNORE* 中的一个以提示下层 SQLite 库应当如何处理对该列的访问。

该回调的第一个参数指明哪种操作将被授权。第二个和第三个参数根据第一个参数的具体值将为传给操作的参数或为 `None`。第四个参数如果适用则为数据库名称（"main", "temp" 等）。第五个参数是负责尝试访问的最内层触发器或视图的名称或者如果该尝试访问是直接来自输入的 SQL 代码的话则为 `None`。

请参阅 SQLite 文档了解第一个参数可能的值以及依赖于第一个参数的第二个和第三个参数的含义。所有必需的常量均在 `sqlite3` 模块中可用。

将 `None` 作为 *authorizer_callback* 传入将禁用授权回调。

在 3.11 版本发生变更: 增加对使用 `None` 禁用授权回调的支持。

在 3.13 版本发生变更: 将 *authorizer_callback* 作为关键字参数传入的做法已被弃用。该形参将在 Python 3.15 中成为仅限位置形参。

set_progress_handler(*progress_handler*, *n*)

注册*callable* *progress_handler* 以针对 SQLite 虚拟机的每 *n* 条指令被唤起。如果你想要在长时间运行的操作，例如更新 GUI 期间获得来自 SQLite 的调用这将很有用处。

如果你想清除任何先前安装的进度处理器，可在调用该方法时传入 `None` 作为 *progress_handler*。

从处理函数返回非零值将终止当前正在执行的查询并导致它引发*DatabaseError* 异常。

在 3.13 版本发生变更: 将 *progress_handler* 作为关键字参数传入的做法已被弃用。该形参将在 Python 3.15 中成为仅限位置形参。

set_trace_callback(*trace_callback*)

注册*callable* *trace_callback* 以针对 SQLite 后端实际执行的每条 SQL 语句被唤起。

传给该回调的唯一参数是被执行的语句 (作为*str*)。回调的返回值将被忽略。请注意后端不仅会运行传给*Cursor.execute()* 方法的语句。其他来源还包括 `sqlite3` 模块的事务管理 以及在当前数据库中定义的触发器的执行。

传入 `None` 作为 *trace_callback* 将禁用追踪回调。

> **备注**
>
> 在跟踪回调中产生的异常不会被传播。作为开发和调试的辅助手段，使用*enable_callback_tracebacks()* 来启用打印跟踪回调中产生的异常的回调。

Added in version 3.3.

在 3.13 版本发生变更: 将 *trace_callback* 作为关键字参数传入的做法已被弃用。该形参将在 Python 3.15 中成为仅限位置形参。

enable_load_extension(*enabled*, */*)

如果 *enabled* 为 `True` 则允许 SQLite 从共享库加载 SQLite 扩展；否则，不允许加载 SQLite 扩展。SQLite 扩展可以定义新的函数、聚合或全新的虚拟表实现。一个知名的扩展是与随同 SQLite 一起分发的全文搜索扩展。

> **备注**
>
> 在默认情况下 `sqlite3` 模块的构建没有附带可加载扩展支持，因为某些平台（主要是 macOS）上的 SQLite 库在编译时未启用此特性。要获得可加载扩展支持，你必须将 `--enable-loadable-sqlite-extensions` 选项传给 **configure**。

引发一个审计事件 `sqlite3.enable_load_extension` 并附带参数 `connection`, `enabled`。

Added in version 3.2.

在 3.10 版本发生变更: 增加了 `sqlite3.enable_load_extension` 审计事件。

```
con.enable_load_extension(True)

# 加载 fts (fulltext search) 扩展
con.execute("select load_extension('./fts3.so')")

# 你也可以使用 API 调用来加载该扩展:
# con.load_extension("./fts3.so")

# 禁止扩展再次加载
con.enable_load_extension(False)
```

(续下页)

(接上页)

```
# 来自 SQLite wiki 的示例
con.execute("CREATE VIRTUAL TABLE recipe USING fts3(name, ingredients)")
con.executescript("""
    INSERT INTO recipe (name, ingredients) VALUES('broccoli stew', 'broccoli␣
↪peppers cheese tomatoes');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin stew', 'pumpkin onions␣
↪garlic celery');
    INSERT INTO recipe (name, ingredients) VALUES('broccoli pie', 'broccoli cheese␣
↪onions flour');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin pie', 'pumpkin sugar␣
↪flour butter');
    """)
for row in con.execute("SELECT rowid, name, ingredients FROM recipe WHERE name␣
↪MATCH 'pie'"):
    print(row)
```

load_extension(*path*, */*, ***, *entrypoint=None*)

从共享库加载 SQLite 扩展。请在调用此方法前通过*enable_load_extension()* 来启用扩展加载。

参数

- **path** (str) -- SQLite 扩展的路径。
- **entrypoint** (str | *None*) -- 入口点名称。如果为 None (默认值)，SQLite 将自行生成入口点名称；请参阅 SQLite 文档 Loading an Extension 了解详情。

引发一个审计事件 sqlite3.load_extension 并附带参数 connection, path。

Added in version 3.2.

在 3.10 版本发生变更: 增加了 sqlite3.load_extension 审计事件。

在 3.12 版本发生变更: 增加了 *entrypoint* 形参。

iterdump(***, *filter=None*)

返回一个*iterator* 用来将数据库转储为 SQL 源代码。在保存内存数据库以便将来恢复时很有用处。类似于 **sqlite3** shell 中的 .dump 命令。

参数

filter (str | *None*) -- 可选的 LIKE 模式用于确定要转储的数据库对象，例如 prefix_%。如为 None (默认值)，则将包括所有数据库对象。

示例:

```
# 将文件 example.db 转换为 SQL 转储文件 dump.sql
con = sqlite3.connect('example.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

参见

如何处理非 UTF-8 文本编码格式

在 3.13 版本发生变更: 添加了 *filter* 形参。

backup(*target*, ***, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.250*)

创建 SQLite 数据库的备份。

即使数据库是通过其他客户端访问或通过同一连接并发访问也是有效的。

参数

- **target** (Connection) -- 用于保存备份的数据库连接。
- **pages** (int) -- 每次要拷贝的页数。如果小于等于 0，则一次性拷贝整个数据库。默认为 -1。
- **progress** (*callback* | None) -- 如果设为一个*callable*，它将针对每次备份迭代附带三个整数参数被唤起：上次迭代的状态 *status*，待拷贝的剩余页数 *remaining*，以及总页数 *total*。默认值为 None。
- **name** (str) -- 要备份的数据库名称。可能为代表主数据库的 "main" (默认值)，代表临时数据库的 "temp"，或者使用 ATTACH DATABASE SQL 语句所附加的自定义数据库名称。
- **sleep** (float) -- 连续尝试备份剩余页所要间隔的休眠秒数。

示例 1，将现有数据库拷贝至另一个数据库:

```
def progress(status, remaining, total):
    print(f'已复制 {total-remaining} 到 {total} 页……')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

示例 2，将现有数据库拷贝至一个临时副本:

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
dst.close()
src.close()
```

Added in version 3.7.

参见

如何处理非 UTF-8 文本编码格式

getlimit(*category*, */*)

获取一个连接的运行时限制。

参数

category (int) -- 要查询的 SQLite limit category。

返回类型

int

抛出

ProgrammingError -- 如果 *category* 不能被下层的 SQLite 库所识别。

示例，查询*Connection* con 上一条 SQL 语句的最大长度（默认值为 1000000000）：

```
>>> con.getlimit(sqlite3.SQLITE_LIMIT_SQL_LENGTH)
1000000000
```

Added in version 3.11.

setlimit(*category*, *limit*, /)

设置连接运行时限制。如果试图将限制提高到超出强制上界则会静默地截短到强制上界。无论限制值是否被修改，都将返回之前的限制值。

参数

- **category** (int) -- 要设置的 SQLite limit category。
- **limit** (int) -- 新的限制值。如为负值，当前限制将保持不变。

返回类型
int

抛出
ProgrammingError -- 如果 *category* 不能被下层的 SQLite 库所识别。

示例，将*Connection* con 上附加的数据库数量限制为 1（默认限制为 10）：

```
>>> con.setlimit(sqlite3.SQLITE_LIMIT_ATTACHED, 1)
10
>>> con.getlimit(sqlite3.SQLITE_LIMIT_ATTACHED)
1
```

Added in version 3.11.

getconfig(*op*, /)

查询一个布尔类型的连接配置选项。

参数
op (int) -- 一个*SQLITE_DBCONFIG 代码*。

返回类型
bool

Added in version 3.12.

setconfig(*op*, *enable=True*, /)

设置一个布尔类型的连接配置选项。

参数

- **op** (int) -- 一个*SQLITE_DBCONFIG 代码*。
- **enable** (bool) -- 如果该配置选项应当启用则为 True (默认值)；如果应当禁用则为 False。

Added in version 3.12.

serialize(*, *name='main'*)

将一个数据库序列化为*bytes* 对象。对于普通的磁盘数据库文件，序列化就是磁盘文件的一个副本。对于内存数据库或“临时”数据库，序列化就是当数据库备份到磁盘时要写入到磁盘的相同字节序列。

参数
name (str) -- 要序列化的数据库名称。默认为 "main"。

返回类型
bytes

> **备注**
>
> 此方法仅在下层 SQLite 库具有序列化 API 时可用。

Added in version 3.11.

deserialize(*data*, */*, ***, *name='main'*)

将一个已序列化的 数据库反序列化至*Connection*。此方法将导致数据库连接从 *name* 数据库断开，并基于包含在 *data* 中的序列化数据将 *name* 作为内存数据库重新打开。

参数

- **data** (bytes) -- 已序列化的数据库。
- **name** (str) -- 反序列化的目标数据库名称。默认为 "main"。

抛出

- ***OperationalError*** -- 如果当前数据库连接正在执行读取事务或备份操作。
- ***DatabaseError*** -- 如果 *data* 不包含有效的 SQLite 数据库。
- ***OverflowError*** -- 如果*len(data)* 大于 2**63 - 1。

> **备注**
>
> 此方法仅在下层的 SQLite 库具有反序列化 API 时可用。

Added in version 3.11.

autocommit

该属性控制符合 **PEP 249** 的事务行为。autocommit 有三个可用的值:

- False: 选择符合 **PEP 249** 的事务行为，即 sqlite3 将保证总是开启一个事务。使用*commit()* 和*rollback()* 来关闭事务。

  这是 autocommit 推荐的取值。

- True: 使用 SQLite 的 autocommit mode。在此模式下*commit()* 和*rollback()* 将没有任何效果。

- *LEGACY_TRANSACTION_CONTROL*: Python 3.12 之前 (不符合 **PEP 249**) 的事务控制。请参阅*isolation_level* 了解详情。

  这是 autocommit 当前的默认值。

将 autocommit 更改为 False 将开启一个新事务，而将其更改为 True 将提交任何待处理事务。

详情参见*通过 autocommit 属性进行事务控制*。

> **备注**
>
> 除非*autocommit* 为*LEGACY_TRANSACTION_CONTROL* 否则*isolation_level* 属性将不起作用。

Added in version 3.12.

in_transaction

这个只读属性对应于低层级的 SQLite autocommit mode。

如果一个事务处于活动状态（有未提交的更改）则为 True，否则为 False。

Added in version 3.2.

isolation_level

控制 sqlite3 的*旧式事务处理模式*。如果设为 None，则绝不会隐式地开启事务。如果设为 "DEFERRED"、"IMMEDIATE" 或 "EXCLUSIVE" 中的一个，对应于下层的 SQLite transaction behaviour，会执行*隐式事务管理*。

如果未被*connect()* 的 *isolation_level* 形参覆盖，则默认为 ""，这是 "DEFERRED" 的一个别名。

> **备注**
>
> 建议使用*autocommit* 来控制事务处理而不是使用 isolation_level。除非*autocommit* 设为*LEGACY_TRANSACTION_CONTROL* (默认值) 否则 isolation_level 将不起作用。

row_factory

针对从该连接创建的*Cursor* 对象的初始*row_factory*。为该属性赋值不会影响属于该连接的现有游标的 row_factory，只影响新的游标。默认为 None，表示将每一行作为*tuple* 返回。

详情参见*如何创建并使用行工厂对象*。

text_factory

一个接受*bytes* 形参并返回其文本表示形式的*callable*。该可调用对象将针对数据类型为 TEXT 的 SQLite 值被唤起。在默认情况下，该属性将被设为*str*。

请参阅*如何处理非 UTF-8 文本编码格式* 了解详情。

total_changes

返回自打开数据库连接以来已修改、插入或删除的数据库行的总数。

游标对象

一个代表被用于执行 SQL 语句，并管理获取操作的上下文的 database cursor 的 Cursor 对象。游标对象是使用*Connection.cursor()*，或是通过使用任何*连接快捷方法* 来创建的。

Cursor 对象属于*迭代器*，这意味着如果你通过*execute()* 来执行 SELECT 查询，你可以简单地迭代游标来获取结果行:

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.**Cursor**

Cursor 游标实例具有以下属性和方法。

execute(*sql*, *parameters=()*, */*)

执行一条 SQL 语句，可以选择使用*占位符* 来绑定 Python 值。

参数

- **sql** (*str*) -- 一条 SQL 语句。
- **parameters** (*dict* | *sequence*) -- 要绑定到 *sql* 中占位符的 Python 值。如果使用命名占位符则会使用 dict。如果使用非命名占位符则会使用 sequence。参见*如何在 SQL 查询中使用占位符来绑定值*。

抛出

ProgrammingError -- 如果 *sql* 包含多条 SQL 语句。

如果*autocommit* 为*LEGACY_TRANSACTION_CONTROL*，*isolation_level* 不为 None，*sql* 为一条 INSERT, UPDATE, DELETE 或 REPLACE 语句，并且没有开启事务，则会在执行 *sql* 之前隐式地开启事务。

Deprecated since version 3.12, will be removed in version 3.14: 如果使用了*命名占位符* 并且 *parameters* 是一个序列而非*dict* 则会发出*DeprecationWarning*。从 Python 3.14 起，将改为引发*ProgrammingError*。

使用*executescript()* 来执行多条 SQL 语句。statements.

executemany(*sql*, *parameters*, */*)

对于 *parameters* 中的每一项，重复执行*参数化的* DML (Data Manipulation Language) SQL 语句 *sql*。

使用与*execute()* 相同的隐式事务处理。

参数

- **sql** (*str*) -- 一条 SQL DML 语句。
- **parameters** (*iterable*) -- 一个用来绑定到 *sql* 中的占位符的形参的 iterable。参见*如何在 SQL 查询中使用占位符来绑定值*。

抛出

ProgrammingError -- 如果 *sql* 包含多条 SQL 语句，或者不属于 DML 语句。

示例:

```
rows = [
    ("row1",),
    ("row2",),
]
# cur 是一个 sqlite3.Cursor 对象
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

备注

任何结果行都将被丢弃，包括带有 RETURNING 子句 的 DML 语句。

Deprecated since version 3.12, will be removed in version 3.14: 如果使用了*命名占位符* 并且 *parameters* 中的每个条目都是序列而非*dict* 则会发出*DeprecationWarning*。从 Python 3.14 起，将改为引发*ProgrammingError*。

executescript(*sql_script*, */*)

执行 *sql_script* 中的 SQL 语句。如果*autocommit* 为*LEGACY_TRANSACTION_CONTROL* 并且存在待处理的事务，则首先隐式执行一条 COMMIT 语句。不会执行其他隐式事务控制；任何事务控制都必须添加至 *sql_script*。

sql_script 必须为*字符串*。

示例:

```
# cur 是一个 sqlite3.Cursor 对象
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

fetchone()

如果*row_factory* 为 None，则将下一行查询结果集作为*tuple* 返回。否则，将其传给指定的行工厂函数并返回函数结果。如果没有更多可用数据则返回 None。

fetchmany(*size=cursor.arraysize*)

将下一个多行查询结果集作为*list* 返回。如果没有更多可用行时则返回一个空列表。

每次调用要获取的行数是由 *size* 形参指定的。如果未指定 *size*，则由*arraysize* 确定要获取的行数。如果可用的行少于 *size*，则返回可用的行数。

请注意 *size* 形参会涉及到性能方面的考虑。为了获得优化的性能，通常最好是使用 arraysize 属性。如果使用 *size* 形参，则最好在从一个*fetchmany()* 调用到下一个调用之间保持相同的值。

fetchall()

将全部（剩余的）查询结果行作为*list* 返回。如果没有可用的行则返回空列表。请注意*arraysize* 属性可能会影响此操作的性能。

close()

立即关闭 cursor（而不是在当 __del__ 被调用的时候）。

从这一时刻起该 cursor 将不再可用，如果再尝试用该 cursor 执行任何操作将引发*ProgrammingError* 异常。

setinputsizes(*sizes*, /)

DB-API 要求的方法。在 sqlite3 不做任何事情。

setoutputsize(*size*, *column=None*, /)

DB-API 要求的方法。在 sqlite3 不做任何事情。

arraysize

用于控制*fetchmany()* 返回行数的可读取/写入属性。该属性的默认值为 1，表示每次调用将获取单独一行。

connection

提供属于该游标的 SQLite *Connection* 的只读属性。通过调用*con.cursor()* 创建的*Cursor* 对象将具有一个指向 *con* 的*connection* 属性:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
>>> con.close()
```

description

提供上一次查询的列名称的只读属性。为了与 Python DB API 保持兼容，它会为每个列返回一个 7 元组，每个元组的最后六个条目均为 None。

对于没有任何匹配行的 SELECT 语句同样会设置该属性。

lastrowid

提供上一次插入的行的行 ID 的只读属性。它只会在使用*execute()* 方法的 INSERT 或 REPLACE 语句成功后被更新。对于其他语句，则在*executemany()* 或*executescript()*，或者如果插入失败，lastrowid 的值将保持不变。lastrowid 的初始值为 None。

> **备注**
>
> 对 WITHOUT ROWID 表的插入不被记录。

在 3.6 版本发生变更: 增加了 REPLACE 语句的支持。

rowcount

提供 INSERT, UPDATE, DELETE 和 REPLACE 语句所修改行数的只读属性；对于其他语句则为 -1，包括 CTE (Common Table Expression) 查询。只有*execute()* 和*executemany()* 方法会在语句运行完成后更新此属性。这意味着任何结果行都必须按顺序被提取以使 rowcount 获得更新。

row_factory

控制从该 Cursor 获取的行的表示形式。如为 None，一行将表示为一个*tuple*。可设置形式包括*sqlite3.Row*；或者接受两个参数的*callable*，一个*Cursor* 对象和由行内所有值组成的 tuple，以及返回代表一个 SQLite 行的自定义对象。

默认为当 Cursor 被创建时设置的*Connection.row_factory*。对该属性赋值不会影响父连接的*Connection.row_factory*。

详情参见*如何创建并使用行工厂对象*。

Row 对象

class sqlite3.**Row**

一个被用作*Connection* 对象的高度优化的*row_factory* 的 Row 实例。它支持迭代、相等性检测、*len()* 以及基于列名称的*mapping* 访问和数字序列。

两个 Row 对象如果具有相同的列名称和值则比较结果相等。

详情参见*如何创建并使用行工厂对象*。

keys()

在一次查询之后，立即将由列名称组成的*list* 作为*字符串* 返回，它是*Cursor.description* 中每个元组的第一个成员。

在 3.5 版本发生变更: 添加了对切片操作的支持。

Blob 对象

class sqlite3.**Blob**

Added in version 3.11.

Blob 实例是可以读写 SQLite BLOB (Binary Large OBject) 数据的*file-like object*。调用*len(blob)* 可得到 blob 的大小（字节数）。请使用索引和*切片* 来直接访问 blob 数据。

将*Blob* 作为*context manager* 使用以确保使用结束后 blob 句柄自动关闭。

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE test(blob_col blob)")
con.execute("INSERT INTO test(blob_col) VALUES(zeroblob(13))")

# 写入到我们的 blob，使用两次 write 操作：
with con.blobopen("test", "blob_col", 1) as blob:
    blob.write(b"hello, ")
    blob.write(b"world.")
    # 修改我们的 blob 的开头和末尾字节
    blob[0] = ord("H")
    blob[-1] = ord("!")

# 读取我们的 blob 的内容
with con.blobopen("test", "blob_col", 1) as blob:
    greeting = blob.read()

print(greeting)  # 输出 "b'Hello, world!'"
con.close()
```

close()

关闭 blob。

从这一时刻起该 blob 将不再可用。如果再尝试用该 blob 执行任何操作将引发*Error* (或其子类) 异常。

read(*length=-1*, */*)

从 blob 的当前偏移位置读取 *length* 个字节的数据。如果到达了 blob 的末尾，则将返回 EOF (End of File) 之前的数据。当未指定 *length*，或指定负值时，*read()* 将读取至 blob 的末尾。

write(*data*, */*)

在 blob 的当前偏移位置上写入 *data*。此函数不能改变 blob 的长度。写入数据超出 blob 的末尾将引发*ValueError*。

tell()

返回 blob 的当前访问位置。

seek(*offset*, *origin=os.SEEK_SET*, */*)

将 Blob 的当前访问位置设为 *offset*。*origin* 参数默认为*os.SEEK_SET* (blob 的绝对位置)。*origin* 的其他值包括*os.SEEK_CUR* （相对于当前位置寻址) 和*os.SEEK_END* (相对于 blob 末尾寻址)。

PrepareProtocol 对象

class sqlite3.**PrepareProtocol**

PrepareProtocol 类型的唯一目的是作为 **PEP 246** 风格的适配协议让对象能够将自身适配为原生 *SQLite* 类型。

异常

异常层次是由 DB-API 2.0 (**PEP 249**) 定义的。

exception sqlite3.**Warning**

目前此异常不会被 sqlite3 模块引发，但可能会被使用 sqlite3 的应用程序引发，例如当一个用户自定义的函数在插入操作中截断了数据时。Warning 是*Exception* 的一个子类。

exception sqlite3.**Error**

本模块中其他异常的基类。使用它来捕捉所有的错误，只需一条 except 语句。Error 是*Exception* 的子类。

如果异常是产生于 SQLite 库的内部，则以下两个属性将被添加到该异常:

sqlite_errorcode

来自 SQLite API 的数字错误代码

Added in version 3.11.

sqlite_errorname

来自 SQLite API 的数字错误代码符号名称

Added in version 3.11.

exception sqlite3.**InterfaceError**

因错误使用低层级 SQLite C API 而引发的异常，换句话说，如果此异常被引发，则可能表明 sqlite3 模块中存在错误。InterfaceError 是*Error* 的一个子类。

exception sqlite3.**DatabaseError**

对与数据库有关的错误引发的异常。它作为几种数据库错误的基础异常。它只通过专门的子类隐式引发。DatabaseError 是*Error* 的一个子类。

exception sqlite3.**DataError**

由于处理的数据有问题而产生的异常，比如数字值超出范围，字符串太长。DataError 是*DatabaseError* 的子类。

exception sqlite3.**OperationalError**

与数据库操作有关的错误而引发的异常，不一定在程序员的控制之下。例如，数据库路径没有找到，或者一个事务无法被处理。OperationalError 是*DatabaseError* 的子类。

exception sqlite3.**IntegrityError**

当数据库的关系一致性受到影响时引发的异常。例如外键检查失败等。它是*DatabaseError* 的子类。

exception sqlite3.**InternalError**

当 SQLite 遇到一个内部错误时引发的异常。如果它被引发，可能表明运行中的 SQLite 库有问题。InternalError 是*DatabaseError* 的子类。

exception sqlite3.**ProgrammingError**

针对 sqlite3 API 编程错误引发的异常，例如向查询提供错误数量的绑定，或试图在已关闭的*Connection* 上执行操作。ProgrammingError 是*DatabaseError* 的一个子类。

exception sqlite3.**NotSupportedError**

在下层的 SQLite 库不支持某个方法或数据库 API 的情况下引发的异常。例如，在*create_function()* 中把 *deterministic* 设为 True，而下层的 SQLite 库不支持确定性函数的时候。NotSupportedError 是*DatabaseError* 的一个子类。

SQLite 与 Python 类型

SQLite 原生支持如下的类型：NULL，INTEGER，REAL，TEXT，BLOB。

因此可以将以下 Python 类型发送到 SQLite 而不会出现任何问题：

Python 类型	SQLite 类型
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

这是 SQLite 类型默认转换为 Python 类型的方式：

SQLite 类型	Python 类型
NULL	None
INTEGER	*int*
REAL	*float*
TEXT	取决于*text_factory*，默认为*str*
BLOB	*bytes*

sqlite3 模块的类型系统可通过两种方式来扩展：你可以通过对象适配器 将额外的 Python 类型保存在 SQLite 数据库中，你也可以让 sqlite3 模块通过转换器 将 SQLite 类型转换为不同的 Python 类型。types via.

默认适配器和转换器（已弃用）

备注

自 Python 3.12 起，默认适配器和转换器已被弃用。取而代之的是使用适配器和转换器范例程序，并根据您的需要定制它们。

弃用的默认适配器和转换器包括：

- 将*datetime.date* 对象转换为 ISO 8601 格式字符串 的适配器。
- 将*datetime.datetime* 对象转换为 ISO 8601 格式字符串的适配器。
- 从已声明的 ”date” 类型到*datetime.date* 对象的转换器。
- 将已声明的”timestamp” 类型转成*datetime.datetime* 对象的转换器。小数部分将截断至 6 位（微秒精度）。

备注

默认的” 时间戳” 转换器忽略了数据库中的 UTC 偏移，总是返回一个原生的*datetime.datetime* 对象。要在时间戳中保留 UTC 偏移，可以不使用转换器，或者用*register_converter()* 注册一个偏移感知的转换器。

自 3.12 版本弃用.

命令行接口

sqlite3 模块可以作为脚本被唤起，使用解释器的 -m 开关选项，以提供一个简单的 SQLite shell。参数签名如下:

```
python -m sqlite3 [-h] [-v] [filename] [sql]
```

输入 .quit 或 CTRL-D 退出 shell。

-h, --help
打印 CLI 帮助。

-v, --version
打印下层 SQLite 库版本。

Added in version 3.12.

12.6.3 常用方案指引

如何在 SQL 查询中使用占位符来绑定值

SQL 操作通常会需要使用来自 Python 变量的值。不过，请谨慎使用 Python 的字符串操作来拼装查询，因为这样易受 SQL injection attacks。例如，攻击者可以简单地添加结束单引号并注入 OR TRUE 来选择所有的行:

```
>>> # 绝不要这样做 -- 很不安全!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

请改用 DB-API 的形参替换。要将变量插入到查询字符串中，可在字符串中使用占位符，并通过将实际值作为游标的*execute()* 方法的第二个参数以由多个值组成的*tuple* 形式提供给查询来替换它们。

SQL 语句可以使用两种占位符之一：问号占位符（问号风格）或命名占位符（命名风格）。对于问号风格，*parameters* 要是一个长度必须与占位符的数量相匹配的*sequence*，否则将引发*ProgrammingError*。对于命名风格，*parameters* 必须是*dict* （或其子类）的实例，它必须包含与所有命名参数相对应的键；任何额外的条目都将被忽略。下面是一个同时使用这两种风格的示例:

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# 这是用于 executemany() 的名称风格:
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# 这是用于 SELECT 查询的问号风格:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
print(cur.fetchall())
con.close()
```

> **备注**
>
> **PEP 249** 数字占位符已经 不再被支持。如果使用，它们将被解读为命名占位符。

如何将自定义 Python 类型适配到 SQLite 值

SQLite 仅支持一个原生数据类型的有限集。要在 SQLite 数据库中存储自定义 Python 类型，请将它们 *适配到SQLite 原生可识别的 Python 类型* 之一。

有两种方式可将 Python 对象适配到 SQLite 类型：让你的对象自行适配，或是使用 *适配器可调用对象*。后者将优先于前者发挥作用。对于导出自定义类型的库，启用该类型的自行适配可能更为合理。而作为一名应用程序开发者，通过注册自定义适配器函数进行直接控制可能更为合理。

如何编写可适配对象

假设我们有一个代表笛卡尔坐标系中的坐标值对 `Point`，`x` 和 `y` 的类，该坐标值在数据库中将存储为一个文本字符串。这可以通过添加一个返回已适配值的 `__conform__(self, protocol)` 方法来实现。传给 *protocol* 的对象将为 *PrepareProtocol* 类型。

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.x};{self.y}"

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
con.close()
```

如何注册适配器可调用对象

另一种可能的方式是创建一个将 Python 对象转换为 SQLite 兼容类型的函数。随后可使用 *register_adapter()* 来注册该函数。

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return f"{point.x};{point.y}"

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
con.close()
```

如何将 SQLite 值转换为自定义 Python 类型

编写适配器使你可以将 *from* 自定义 Python 类型转换为 *to* SQLite 值。为了能将 *from* SQLite 值转换为 *to* 自定义 Python 类型，我们可使用 *converters*。

让我们回到 `Point` 类。我们以以分号分隔的字符串形式在 SQLite 中存储了 x 和 y 坐标值。

首先，我们将定义一个转换器函数，它接受这样的字符串作为形参并根据该参数构造一个 `Point` 对象。

备注

转换器函数 **总是**接受传入一个*bytes* 对象，无论下层的 SQLite 数据类型是什么。

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

我们现在需要告诉 `sqlite3` 何时应当转换一个给定的 SQLite 值。这是在连接到一个数据库时完成的，使用*connect()* 的 *detect_types* 形参。有三个选项:

- 隐式: 将 *detect_types* 设为*PARSE_DECLTYPES*
- 显式: 将 *detect_types* 设为*PARSE_COLNAMES*
- 同时: 将 *detect_types* 设为 `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`。列名的优先级高于声明的类型。

下面的示例演示了隐式和显式的方法:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
    return f"{point.x};{point.y}"

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# 注册适配器和转换器
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) 使用声明的类型来解析
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.execute("CREATE TABLE test(p point)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute("SELECT p FROM test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 2) 使用列名称来解析
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.execute("CREATE TABLE test(p)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute('SELECT p AS "p [point]" FROM test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

适配器和转换器范例程序

本小节显示了通用适配器和转换器的范例程序。

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """将 datetime.date 适配为 ISO 8601 日期。"""
    return val.isoformat()

def adapt_datetime_iso(val):
    """将 datetime.datetime 适配为不带时区的 ISO 8601 日期。"""
    return val.isoformat()

def adapt_datetime_epoch(val):
    """将 datetime.datetime 适配为 Unix 时间戳。"""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """将 ISO 8601 日期转换为 datetime.date 对象。"""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """将 ISO 8601 日期时间转换为 datetime.datetime 对象。"""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
    """将 Unix 纪元时间戳转换为 datetime.datetime 对象。"""
    return datetime.datetime.fromtimestamp(int(val))

sqlite3.register_converter("date", convert_date)
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

如何使用连接快捷方法

通过使用*Connection* 类的*execute()*, *executemany()* 与*executescript()* 方法，您可以简化您的代码，因为无需再显式创建（通常是多余的）*Cursor* 对象。此时*Cursor* 对象会被隐式创建并且由这些快捷方法返回。这样一来，您仅需在*Connection* 对象上调用一次方法就可以执行 SELECT 语句，并对其进行迭代。

```
# 创建并填充表。
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)

# 打印表内容
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")

# close() 不是一个快捷方法也不会被自动调用；
```

(续下页)

(接上页)

```
# 连接对象应当被手动关闭
con.close()
```

如何使用连接上下文管理器

Connection 对象可被用作上下文管理器以便在离开上下文管理器代码块时自动提交或回滚开启的事务。如果 with 语句体无异常地结束，事务将被提交。如果提交失败，或者如果 with 语句体引发了未捕获的异常，则事务将被回滚。如果*autocommit* 为 False，则会在提交或回滚后隐式地开启一个新事务。

如果在离开 with 语句体时没有开启的事务，或者如果*autocommit* 为 True，则上下文管理器将不做任何操作。

备注

上下文管理器既不会隐式开启新事务也不会关闭连接。如果你需要关闭上下文管理器，请考虑使用*contextlib.closing()*。

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# 成功，con.commit() 将在此后被自动调用
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() 会在 with 代码块结束时被自动调用并附带一个异常；
# 该异常仍会被引发并且必须被捕获
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# 被用作上下文管理器的连接对象只能提交或回滚事务，
# 因此连接对象必须被手动关闭
con.close()
```

如何使用 SQLite URI

一些有用的 URI 技巧包括:

- 以只读模式打开一个数据库:

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
>>> con.close()
```

- 如果一个数据库尚不存在则不会隐式地新建数据库；如果无法新建数据库则将引发*OperationalError*:

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

- 创建一个名为 shared 的内存数据库:

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)

con1.close()
con2.close()
```

关于此特性的更多信息，包括可用的形参列表，可以在 SQLite URI documentation 中找到。

如何创建并使用行工厂对象

在默认情况下，sqlite3 会以*tuple* 来表示每一行。如果 tuple 不适合你的需求，你可以使用*sqlite3.Row* 类或自定义的*row_factory*。

虽然 row_factory 同时作为*Cursor* 和*Connection* 的属性存在，但推荐设置*Connection.row_factory*，这样在该连接上创建的所有游标都将使用同一个行工厂对象。

Row 提供了针对列的序列方式和大小写不敏感的名称方式访问，具有优于 tuple 的最小化内存开销和性能影响。要使用 Row 作为行工厂对象，请将其赋值给 row_factory 属性:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = sqlite3.Row
```

现在查询将返回 Row 对象:

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
>>> row[0]          # 通过索引访问。
'Earth'
>>> row["name"]     # 通过名称访问。
'Earth'
>>> row["RADIUS"]   # 列名不区分大小写。
6378
>>> con.close()
```

备注

FROM 子句可以在 SELECT 语句中省略，像在上面的示例中那样。在这种情况下，SQLite 将返回单独的行，其中的列由表达式来定义，例如使用字面量并给出相应的别名 expr AS alias。

你可以创建自定义*row_factory* 用来返回*dict* 形式的行，将列名映射到相应的值。

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

使用它，现在查询将返回 dict 而不是 tuple:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
```

(续下页)

(接上页)

```
{'a': 1, 'b': 2}
>>> con.close()
```

以下行工厂函数将返回一个*named tuple*:

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` 可以像下面这样使用:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # 索引访问。
1
>>> row.b   # 属性访问。
2
>>> con.close()
```

经过一些调整，上面的范例程序可以被适配为使用*dataclass*，或任何其他自定义类，而不是*namedtuple*。

如何处理非 UTF-8 文本编码格式

在默认情况下，`sqlite3` 使用*str* 来适配 `TEXT` 数据类型的 SQLite 值。这对 UTF-8 编码的文本来说很适用，但对于其他编码格式和无效的 UTF-8 来说则可能出错。你可以使用自定义的*text_factory* 来处理这种情况。

由于 SQLite 的 flexible typing，遇到包含非 UTF-8 编码格式的 `TEXT` 数据类型甚至任意数据的表字段的情况并不少见。作为演示，让我们假定有一个使用 ISO-8859-2 (Latin-2) 编码的文本的数据库，例如一个捷克语-英语字典条目的表。假定我们现在有一个*Connection* 实例 `con` 已连接到这个数据库，我们将可以使用这个*text_factory* 来解码使用 Latin-2 编码的文本:

```
con.text_factory = lambda data: str(data, encoding="latin2")
```

对于存储在 `TEXT` 表字段中的无效 UTF-8 或任意数据，你可以使用以下技巧，借用自 unicode-howto:

```
con.text_factory = lambda data: str(data, errors="surrogateescape")
```

备注

`sqlite3` 模块 API 不支持包含替代符的字符串。

参见

unicode-howto

12.6.4 说明

事务控制

sqlite3 提供了多个方法来控制在何时以及怎样控制数据库事务的开启和关闭。推荐使用通过 *autocommit* 属性进行事务控制，而通过 *isolation_level* 属性进行事务控制则保留了 Python 3.12 之前的行为。

通过 autocommit 属性进行事务控制

控制事务行为的推荐方式是通过 *Connection.autocommit* 属性，最好是使用 *connect()* 的 *autocommit* 形参来设置该属性。

建议将 *autocommit* 设为 False，表示使用兼容 **PEP 249** 的事务控制。这意味着：

- sqlite3 会确保事务始终处于开启状态，因此 *connect()*、*Connection.commit()* 和 *Connection.rollback()* 将隐式地开启一个新事务（对于后两者，在关闭待处理事务后会立即执行）。开启事务时 sqlite3 会使用 BEGIN DEFERRED 语句。
- 事务应当显式地使用 commit() 执行提交。
- 事务应当显式地使用 rollback() 执行回滚。
- 如果数据库执行 *close()* 时有待处理的更改则会隐式地执行回滚。

将 *autocommit* 设为 True 以启用 SQLite 的 autocommit mode。在此模式下，*Connection.commit()* 和 *Connection.rollback()* 将没有任何作用。请注意 SQLite 的自动提交模式与兼容 **PEP 249** 的 *Connection.autocommit* 属性不同；请使用 *Connection.in_transaction* 查询底层的 SQLite 自动提交模式。

将 *autocommit* 设为 *LEGACY_TRANSACTION_CONTROL* 以将事务控制行为保留给 *Connection.isolation_level* 属性。更多信息参见通过 *isolation_level* 属性进行事务控制。

通过 isolation_level 属性进行事务控制

> **备注**
>
> 推荐的控制事务方式是通过 *autocommit* 属性。参见通过 *autocommit* 属性进行事务控制。

如果 *Connection.autocommit* 被设为 *LEGACY_TRANSACTION_CONTROL*（默认值），则事务行为由 *Connection.isolation_level* 属性控制。否则，isolation_level 将没有任何作用。

如果连接的属性 *isolation_level* 不为 None，新的事务会在 *execute()* 和 *executemany()* 执行 INSERT，UPDATE，DELETE 或 REPLACE 语句之前隐式地开启；对于其他语句，则不会执行隐式的事务处理。可分别使用 *commit()* 和 *rollback()* 方法提交和回滚未应用的事务。你可以通过 *isolation_level* 属性来选择下层的 SQLite transaction behaviour —也就是说，sqlite3 是否要隐式地执行以及执行何种类型的 BEGIN 语句

如果 *isolation_level* 被设为 None，则完全不会隐式地开启任何事务。这将使下层 SQLite 库处于自动提交模式，但也允许用户使用显式 SQL 语句执行他们自己的事务处理。下层 SQLite 库的自动提交模式可使用 *in_transaction* 属性来查询。

executescript() 方法会在执行给定的 SQL 脚本之前隐式地提交任何挂起的事务，无论 *isolation_level* 的值是什么。

在 3.6 版本发生变更: 在以前 sqlite3 会在 DDL 语句之前隐式地提交已开启的事务。现存则不会再这样做。

在 3.12 版本发生变更: 现在推荐的控制事务方式是通过 *autocommit* 属性。

CHAPTER 13

数据压缩和存档

本章中描述的模块支持 zlib、gzip、bzip2 和 lzma 数据压缩算法，以及创建 ZIP 和 tar 格式的归档文件。参见由 *shutil* 模块提供的*归档操作* 。

13.1 zlib --- 与 gzip 兼容的压缩

对于需要数据压缩的应用，此模块中的函数允许使用 zlib 库进行压缩和解压缩。zlib 库的项目主页是 https://www.zlib.net。已知此 Python 模块与 1.1.3 之前版本的 zlib 库存在不兼容；1.1.3 版则存在一个 安全缺陷，因此我们推荐使用 1.1.4 或更新的版本。

zlib 的函数有很多选项，一般需要按特定顺序使用。本文档没有覆盖全部的用法。更多详细信息请于 http://www.zlib.net/manual.html 参阅官方手册。

要读写 `.gz` 格式的文件，请参考 *gzip* 模块。

此模块中可用的异常和函数如下：

exception zlib.**error**

在压缩或解压缩过程中发生错误时的异常。

zlib.**adler32**(*data*[, *value*])

计算 *data* 的 Adler-32 校验值。(Adler-32 校验的可靠性与 CRC32 基本相当，但比计算 CRC32 更高效。) 计算的结果是一个 32 位的整数。参数 *value* 是校验时的起始值，其默认值为 1。借助参数 *value* 可为分段的输入计算校验值。此算法没有加密强度，不应用于身份验证和数字签名。此算法的目的仅为验证数据的正确性，不适合作为通用散列算法。

在 3.0 版本发生变更: 结果将总是不带符号的。

zlib.**compress**(*data*, */*, *level=-1*, *wbits=MAX_WBITS*)

压缩 *data* 中的字节，返回包含已压缩数据的字节串对象。*level* 是一个用于控制压缩级别的 `0` 到 `9` 之间的整数或 `-1`；`1` (Z_BEST_SPEED) 表示最快速度和最低压缩率，`9` (Z_BEST_COMPRESSION) 表示最慢速度和最高压缩率。`0` (Z_NO_COMPRESSION) 表示不压缩。默认值为 `-1` (Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 表示速度和压缩率折中的默认值 (目前相当于级别 6)。

参数 *wbits* 指定压缩数据时所使用的历史缓冲区的大小 (窗口大小)，并指定压缩输出是否包含头部或尾部。参数的默认值是 `15` (MAX_WBITS)。参数的值分为几个范围：

- +9 至 +15：窗口大小以二为底的对数。即这些值对应着 512 至 32768 的窗口大小。更大的值会提供更好的压缩，同时内存开销也会更大。压缩输出会包含 zlib 特定格式的头部和尾部。
- −9 至 −15：绝对值为窗口大小以二为底的对数。压缩输出仅包含压缩数据，没有头部和尾部。
- +25 至 +31 = 16 + (9 至 15)：后 4 个比特位为窗口大小以二为底的对数。压缩输出包含一个基本的 **gzip** 头部，并以校验和为尾部。

如果发生任何错误则将引发*error* 异常。

在 3.6 版本发生变更: 现在，*level* 可作为关键字参数。

在 3.11 版本发生变更: 现在可以用 *wbits* 形参来设置窗口位和压缩类型。

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

返回一个压缩对象，用来压缩内存中难以容下的数据流。

参数 *level* 为压缩等级，是整数，可取值为 `0` 到 `9` 或 `-1`。`1` (Z_BEST_SPEED) 表示最快速度和最低压缩率，`9` (Z_BEST_COMPRESSION) 表示最慢速度和最高压缩率。`0` (Z_NO_COMPRESSION) 表示不压缩。参数默认值为 `-1` (Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 是速度和压缩率之间的平衡 (一般相当于设压缩等级为 6)。

method 表示压缩算法。现在只支持 `DEFLATED` 这个算法。

wbits 形参控制历史缓冲区的大小（或称“窗口大小”），以及将要使用的头部和尾部格式。它的含义与*对 compress() 的描述* 相同。

参数 *memLevel* 指定内部压缩操作时所占用内存大小。参数取 `1` 到 `9`。更大的值占用更多的内存，同时速度也更快输出也更小。

参数 *strategy* 用于调节压缩算法。可取值为 `Z_DEFAULT_STRATEGY`、`Z_FILTERED`、`Z_HUFFMAN_ONLY`、`Z_RLE` (zlib 1.2.0.1) 或 `Z_FIXED` (zlib 1.2.2.2)。

参数 *zdict* 指定预定义的压缩字典。它是一个字节序列 (如*bytes* 对象)，其中包含用户认为要压缩的数据中可能频繁出现的子序列。频率高的子序列应当放在字典的尾部。

在 3.3 版本发生变更: 添加关键字参数 *zdict*。

zlib.**crc32**(*data*[, *value*])

计算 *data* 的 CRC (循环冗余校验) 值。计算的结果是一个 32 位的整数。参数 *value* 是校验时的起始值，其默认值为 0。借助参数 *value* 可为分段的输入计算校验值。此算法没有加密强度，不应用于身份验证和数字签名。此算法的目的仅为验证数据的正确性，不适合作为通用散列算法。

在 3.0 版本发生变更: 结果将总是不带符号的。

zlib.**decompress**(*data*, */*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

解压 *data* 中的字节，返回含有已解压内容的 bytes 对象。参数 *wbits* 取决于 *data* 的格式，具体参见下边的说明。*bufsize* 为输出缓冲区的起始大小。函数发生错误时抛出*error* 异常。

wbits 形参控制历史缓冲区的大小（或称“窗口大小”）以及所期望的头部和尾部格式。它类似于*compressobj()* 的形参，但可接受更大范围的值：

- +8 至 +15：窗口尺寸以二为底的对数。输入必须包含 zlib 头部和尾部。
- 0：根据 zlib 头部自动确定窗口大小。只从 zlib 1.2.3.5 版起受支持。
- −8 至 −15：使用 *wbits* 的绝对值作为窗口大小以二为底的对数。输入必须为原始数据流，没有头部和尾部。
- +24 至 +31 = 16 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数。输入必须包括 gzip 头部和尾部。
- +40 至 +47 = 32 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数，并且自动接受 zlib 或 gzip 格式。

当解压缩一个数据流时，窗口大小必须不小于用于压缩数据流的原始窗口大小；使用太小的值可能导致*error* 异常。默认 *wbits* 值对应于最大的窗口大小并且要求包括 zlib 头部和尾部。

bufsize 是用于存放解压数据的缓冲区初始大小。如果需要更大空间，缓冲区大小将按需增加，因此你不需要让这个值完全精确；对其进行调整仅会节省一点对 `malloc()` 的调用次数。

在 3.6 版本发生变更: *wbits* 和 *bufsize* 可用作关键字参数。

`zlib.`**`decompressobj`**(*wbits=MAX_WBITS*[, *zdict*])

返回一个解压对象，用来解压无法被一次性放入内存的数据流。

wbits 形参控制历史缓冲区的大小（或称“窗口大小”）以及所期望的头部和尾部格式。它的含义与*对 decompress() 的描述* 相同。

zdict 形参指定指定一个预定义的压缩字典。如果提供了此形参，它必须与产生将解压数据的压缩器所使用的字典相同。

> **备注**
>
> 如果 *zdict* 是一个可变对象 (例如*bytearray*)，则你不可在对*decompressobj()* 的调用和对解压器的 `decompress()` 方法的调用之间修改其内容。

在 3.3 版本发生变更: 增加了 *zdict* 形参。

压缩对象支持以下方法：

`Compress.`**`compress`**(*data*)

压缩 *data* 并返回 bytes 对象，这个对象含有 *data* 的部分或全部内容的已压缩数据。所得的对象必须拼接在上一次调用*compress()* 方法所得数据的后面。缓冲区中可能留存部分输入以供下一次调用。

`Compress.`**`flush`**([*mode*])

压缩所有缓冲区的数据并返回已压缩的数据。参数 *mode* 可以传入的常量为：`Z_NO_FLUSH`、`Z_PARTIAL_FLUSH`、`Z_SYNC_FLUSH`、`Z_FULL_FLUSH`、`Z_BLOCK` (zlib 1.2.3.4) 或 `Z_FINISH`。默认值为 `Z_FINISH`。`Z_FINISH` 关闭已压缩数据流并不允许再压缩其他数据，`Z_FINISH` 以外的值皆允许这个对象继续压缩数据。调用*flush()* 方法并将 *mode* 设为 `Z_FINISH` 后会无法再次调用*compress()*，此时只能删除这个对象。

`Compress.`**`copy`**()

返回此压缩对象的一个拷贝。它可以用来高效压缩一系列拥有相同前缀的数据。

在 3.8 版本发生变更: 添加了对压缩对象执行*copy.copy()* 和*copy.deepcopy()* 的支持。

解压缩对象支持以下方法：

`Decompress.`**`unused_data`**

一个 bytes 对象，其中包含压缩数据结束之后的任何字节数据。也就是说，它将为 `b""` 直到包含压缩数据的末尾字节可用。如果整个结果字节串都包含压缩数据，它将为一个空的 bytes 对象 `b""`。

`Decompress.`**`unconsumed_tail`**

一个 bytes 对象，其中包含未被上一次*decompress()* 调用所消耗的任何数据。此数据不能被 zlib 机制看到，因此你必须将其送回（可能要附带额外的数据拼接）到后续的*decompress()* 方法调用以获得正确的输出。

`Decompress.`**`eof`**

一个布尔值，指明是否已到达压缩数据流的末尾。

这使得区分正确构造的压缩数据流和不完整或被截断的流成为可能。

Added in version 3.3.

`Decompress.`**`decompress`**(*data*, *max_length=0*)

解压缩 *data* 并返回 bytes 对象，其中包含对应于 *string* 中至少一部分数据的解压缩数据。此数据应当被拼接到之前任何对*decompress()* 方法的调用所产生的输出。部分输入数据可能会被保留在内部缓冲区以供后续处理。

如果可选的形参 *max_length* 非零则返回值将不会长于 *max_length*。这可能意味着不是所有已压缩输入都能被处理；并且未被消耗的数据将被保存在 `unconsumed_tail` 属性中。如果要继续解压缩则这个字节串必须被传给对 `decompress()` 的后续调用。如果 *max_length* 为零则整个输入都会被解压缩，并且 `unconsumed_tail` 将为空。

在 3.6 版本发生变更: *max_length* 可用作关键字参数。

`Decompress.`**`flush`**`(`[*length*]`)`

所有挂起的输入会被处理，并且返回包含剩余未压缩输出的 bytes 对象。在调用 `flush()` 之后，`decompress()` 方法将无法被再次调用；唯一可行的操作是删除该对象。

可选的形参 *length* 设置输出缓冲区的初始大小。

`Decompress.`**`copy`**`()`

返回解压缩对象的一个拷贝。它可以用来在数据流的中途保存解压缩器的状态以便加快随机查找数据流后续位置的速度。

在 3.8 版本发生变更: 添加了对解压缩对象执行 `copy.copy()` 和 `copy.deepcopy()` 的支持。

通过下列常量可获取模块所使用的 zlib 库的版本信息：

`zlib.`**`ZLIB_VERSION`**

构建此模块时所用的 zlib 库的版本字符串。它的值可能与运行时所加载的 zlib 不同。运行时加载的 zlib 库的版本字符串为 `ZLIB_RUNTIME_VERSION`。

`zlib.`**`ZLIB_RUNTIME_VERSION`**

解释器所加载的 zlib 库的版本字符串。

Added in version 3.3.

参见

模块 *gzip*
读写 **gzip** 格式的文件。

http://www.zlib.net
zlib 库项目主页。

http://www.zlib.net/manual.html
zlib 库用户手册。提供了库的许多功能的解释和用法。

对于 gzip (解) 压缩成为瓶颈的情况，python-isal 软件包会使用最兼容的 API 来加快 (解) 压缩的速度。

13.2 `gzip` --- 对 `gzip` 文件的支持

源代码： Lib/gzip.py

此模块提供的简单接口帮助用户压缩和解压缩文件，功能类似于 GNU 应用程序 **gzip** 和 **gunzip**。

数据压缩由 `zlib` 模块提供。

`gzip` 模块提供 `GzipFile` 类和 `open()`、`compress()`、`decompress()` 几个便利的函数。`GzipFile` 类可以读写 **gzip** 格式的文件，还能自动压缩和解压缩数据，这让操作压缩文件如同操作普通的 *file object* 一样方便。

注意，此模块不支持部分可以被 **gzip** 和 **gunzip** 解压的格式，如利用 **compress** 或 **pack** 压缩所得的文件。

这个模块定义了以下内容：

gzip.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

以二进制方式或者文本方式打开一个 gzip 格式的压缩文件，返回一个*file object*。

filename 参数可以是一个实际的文件名（一个*str* 对象或者*bytes* 对象），或者是一个用来读写的已存在的文件对象。

mode 参数可以是二进制模式：`'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` or `'xb'`，或者是文本模式 `'rt'`, `'at'`, `'wt'`, or `'xt'`。默认值是 `'rb'`。

The *compresslevel* argument is an integer from 0 to 9, as for the *GzipFile* constructor.

对于二进制模式，这个函数等价于*GzipFile* 构造器：`GzipFile(filename, mode, compresslevel)`。在这个例子中，*encoding*, *errors* 和 *newline* 三个参数一定不要设置。

对于文本模式，将会创建一个*GzipFile* 对象，并将它封装到一个*io.TextIOWrapper* 实例中，这个实例默认了指定编码，错误抓获行为和行。

在 3.3 版本发生变更: 支持 *filename* 为一个文件对象，支持文本模式和 *encoding*, *errors* 和 *newline* 参数。

在 3.4 版本发生变更: 支持 `'x'`, `'xb'` 和 `'xt'` 三种模式。

在 3.6 版本发生变更: 接受一个*path-like object*。

exception gzip.**BadGzipFile**

针对无效 gzip 文件引发的异常。它继承自*OSError*。针对无效 gzip 文件也可能引发*EOFError* 和*zlib.error*。

Added in version 3.8.

class gzip.**GzipFile**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)

GzipFile 类的构造器，它模拟了*file object* 的大部分方法，但*truncate()* 方法除外。*fileobj* 和 *filename* 中至少有一个必须为非空值。

新的实例基于 *fileobj*，它可以是一个普通文件，一个*io.BytesIO* 对象，或者任何一个与文件相似的对象。当 *filename* 是一个文件对象时，它的默认值是 `None`。

当 *fileobj* 为 `None` 时，*filename* 参数只用于 **gzip** 文件头中，这个文件有可能包含未压缩文件的源文件名。如果文件可以被识别，默认 *fileobj* 的文件名；否则默认为空字符串，在这种情况下文件头将不包含源文件名。

mode 参数可以是 `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` 或 `'xb'` 中的一个，具体取决于文件将被读取还是被写入。如果可识别则默认为 *fileobj* 的模式；否则默认为 `'rb'`。在未来的 Python 发布版中将不再使用 *fileobj* 的模式。最好总是指定 *mode* 为写入模式。

需要注意的是，文件默认使用二进制模式打开。如果要以文本模式打开一个压缩文件，请使用*open()* 方法 (或者使用*io.TextIOWrapper* 包装*GzipFile*)。

compresslevel 参数是一个从 `0` 到 `9` 的整数，用于控制压缩等级；`1` 最快但压缩比例最小，`9` 最慢但压缩比例最大。`0` 不压缩。默认为 `9`。

可选的 *mtime* 参数是 gzip 所请求的时间戳。该时间为 Unix 格式，即距离 1970-01-01 00:00:00 UTC 的秒数。如果 *mtime* 被省略或为 `None`，则会使用当前时间。使用 *mtime* = 0 可生成不依赖于创建时间的压缩流。

有关在解压缩时设置的*mtime* 属性见下文。

调用*GzipFile* 对象的 `close()` 方法不会关闭 *fileobj*，因为你可能希望增加其它内容到已经缩的数据中。你还可以传入一个*io.BytesIO* 对象作为 *fileobj* 打开，并使用*io.BytesIO* 对象的*getvalue()* 方法提取所得到的内存缓冲区数据。

GzipFile 支持*io.BufferedIOBase* 接口，包括迭代和 `with` 语句。只有*truncate()* 方法未被实现。

GzipFile 还提供了以下的方法和属性:

peek(*n*)

读取 *n* 个未压缩字节而不前移文件指针位置。所返回的字节数有可能多于或少于所请求的。

> **备注**
>
> 调用*peek()* 并没有改变*GzipFile* 的文件指针，它可能改变潜在文件对象 (例如：*GzipFile* 使用 *fileobj* 参数进行初始化)。

Added in version 3.2.

mode

`'rb'` 表示可读而 `'wb'` 表示可写。

在 3.13 版本发生变更: 在之前版本中该值为整数 `1` 或 `2`。

mtime

当解压缩时，该属性将被设为最近读取标头的末尾时间戳。它是一个整数，保存从 Unix 纪元 (1970 -01-01 00:00:00 UTC) 开始的秒数。在读取任何标头之前的初始值为 `None`。

name

指向磁盘上 gzip 文件的路径，为*str* 或*bytes* 对象。等价于原始输入路径上*os.fspath()* 的输出，不带其他标准化、解析或扩展。

在 3.1 版本发生变更: 支持 `with` 语句，构造器参数 *mtime* 和*mtime* 属性。

在 3.2 版本发生变更: 添加了对零填充和不可搜索文件的支持。

在 3.3 版本发生变更: 实现*io.BufferedIOBase.read1()* 方法。

在 3.4 版本发生变更: 支持 `'x'` and `'xb'` 两种模式。

在 3.5 版本发生变更: 支持写入任意*bytes-like objects*。*read()* 方法可以接受 `None` 为参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

自 3.9 版本弃用: 打开*GzipFile* 用于写入而不指定 *mode* 参数的做法已被弃用。

在 3.12 版本发生变更: 移除 `filename` 属性，改用*name* 属性。

`gzip.`**compress**(*data*, *compresslevel=9*, ***, *mtime=None*)

压缩 *data*，返回一个包含压缩数据的*bytes* 对象。*compresslevel* 和 *mtime* 的含义与上文中*GzipFile* 构造器的相同。

Added in version 3.2.

在 3.8 版本发生变更: 添加了 *mtime* 形参用于可重复的输出。

在 3.11 版本发生变更: 速度的提升是通过一次性压缩所有数据代替流的方式来达成的。将 *mtime* 设为 `0` 的调用被委托给*zlib.compress()* 以加快速度。在此情况下输出可能包含一个 gzip 标头”OS” 字节值而不是下层 zlib 实现所提供的 255 ”unknown”。

在 3.13 版本发生变更: 当使用此函数时 gzip 标头 OS 字节会保证如在 3.10 和更早版本中一样被设为 255。

`gzip.`**decompress**(*data*)

解压缩 *data*，返回一个包含已解压数据的*bytes* 对象。此函数可以解压缩多成员的 gzip 数据（即多个 gzip 块拼接在一起）。当数据确定只包含一个成员时则 *wbits* 设为 31 的*zlib.decompress()* 函数更快一些。

Added in version 3.2.

在 3.11 版本发生变更: 通过一次性解压缩全部数据而不是通过流方式提高了速度。

13.2.1 用法示例

读取压缩文件示例：

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

创建 GZIP 文件示例：

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

使用 GZIP 压缩已有的文件示例：

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

使用 GZIP 压缩二进制字符串示例：

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

参见

模块 *zlib*
支持 **gzip** 格式所需要的基本压缩模块。

对于 gzip (解) 压缩成为瓶颈的情况，python-isal 软件包会使用最兼容的 API 来加快 (解) 压缩的速度。

13.2.2 命令行界面

gzip 模块提供了简单的命令行界面用于压缩和解压缩文件。

在执行后*gzip* 模块会保留输入文件。

在 3.8 版本发生变更: 添加一个带有用法说明的新命令行界面命令。默认情况下，当你要执行 CLI 时，默认压缩等级为 6。

命令行选项

file
如果未指定 *file*，则从*sys.stdin* 读取。

--fast
指明最快速的压缩方法（较低压缩率）。

--best
指明最慢速的压缩方法（最高压缩率）。

-d, --decompress
解压缩给定的文件。

-h, --help
显示帮助消息。

13.3 bz2 --- 对 bzip2 压缩算法的支持

源代码： Lib/bz2.py

此模块提供了使用 bzip2 压缩算法压缩和解压数据的一套完整的接口。

bz2 模块包含：

- 用于读写压缩文件的*open()* 函数和*BZ2File* 类。
- 用于增量压缩和解压的*BZ2Compressor* 和*BZ2Decompressor* 类。
- 用于一次性压缩和解压的*compress()* 和*decompress()* 函数。

13.3.1 文件压缩和解压

bz2.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

以二进制或文本模式打开 bzip2 压缩文件，返回一个*file object*。

和*BZ2File* 的构造函数类似，*filename* 参数可以是一个实际的文件名（*str* 或*bytes* 对象），或是已有的可供读取或写入的文件对象。

mode 参数可设为二进制模式的 `'r'`、`'rb'`、`'w'`、`'wb'`、`'x'`、`'xb'`、`'a'` 或 `'ab'`，或者文本模式的 `'rt'`、`'wt'`、`'xt'` 或 `'at'`。默认是 `'rb'`。

compresslevel 参数是 1 到 9 的整数，和*BZ2File* 的构造函数一样。

对于二进制模式，这个函数等价于*BZ2File* 构造器：`BZ2File(filename, mode, compresslevel=compresslevel)`。在这种情况下，不可提供 *encoding*, *errors* 和 *newline* 参数。

对于文本模式，将会创建一个*BZ2File* 对象，并将它包装到一个*io.TextIOWrapper* 实例中，此实例带有指定的编码格式、错误处理行为和行结束符。

Added in version 3.3.

在 3.4 版本发生变更: 添加了 `'x'` (单独创建) 模式。

在 3.6 版本发生变更: 接受一个*path-like object*。

class bz2.**BZ2File**(*filename*, *mode='r'*, ***, *compresslevel=9*)

用二进制模式打开 bzip2 压缩文件。

如果 *filename* 是一个*str* 或*bytes* 对象，则打开名称对应的文件目录。否则的话，*filename* 应当是一个*file object*，它将被用来读取或写入压缩数据。

mode 参数可以是表示读取的 `'r'` (默认值)，表示覆写的 `'w'`，表示单独创建的 `'x'`，或表示添加的 `'a'`。这些模式还可分别以 `'rb'`, `'wb'`, `'xb'` 和 `'ab'` 的等价形式给出。

如果 *filename* 是一个文件对象（而不是实际的文件名），则 `'w'` 模式并不会截断文件，而是会等价于 `'a'`。

如果 *mode* 为 `'w'` 或 `'a'`，则 *compresslevel* 可以是 `1` 到 `9` 之间的整数，用于指定压缩等级: `1` 产生最低压缩率，而 `9` (默认值) 产生最高压缩率。

如果 *mode* 为 `'r'`，则输入文件可以为多个压缩流的拼接。

BZ2File 提供了*io.BufferedIOBase* 所指定的所有成员，但*detach()* 和*truncate()* 除外。并支持迭代和 `with` 语句。

BZ2File 还提供了以下方法和属性：

peek([*n*])

返回缓冲的数据而不前移文件位置。至少将返回一个字节的数据（除非为 EOF）。实际返回的字节数不确定。

> **备注**
>
> 虽然调用*peek()* 不会改变*BZ2File* 的文件位置，但它可能改变下层文件对象的位置（举例来说如果*BZ2File* 是通过传入一个文件对象作为 *filename* 的话）。

Added in version 3.3.

fileno()

返回底层文件的文件描述符。

Added in version 3.3.

readable()

返回文件是否已被打开供读取。

Added in version 3.3.

seekable()

返回文件是否支持定位。

Added in version 3.3.

writable()

返回文件是否已被打开供写入。

Added in version 3.3.

read1(*size=-1*)

读取至多 *size* 个未压缩字节，将会避免多次从下层流读取。如果 size 为负值则读取至多为缓冲区数据大小。

如果文件位置为 EOF 则返回 `b''`。

Added in version 3.3.

readinto(*b*)

将字节数据读取到 *b*。

返回读取的字节数（0 表示 EOF）。

Added in version 3.3.

mode

`'rb'` 表示可读而 `'wb'` 表示可写。

Added in version 3.13.

name

bzip2 文件名。等价于下层*file object* 的*name* 属性。

Added in version 3.13.

在 3.1 版本发生变更: 添加了对 `with` 语句的支持。

在 3.3 版本发生变更: 添加了对 *filename* 使用*file object* 而非实际文件名的支持。

添加了 `'a'` (append) 模式，以及对读取多数据流文件的支持。

在 3.4 版本发生变更: 添加了 `'x'` (单独创建) 模式。

在 3.5 版本发生变更: *read()* 方法现在接受 `None` 作为参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.9 版本发生变更: *buffering* 形参已被移除。它自 Python 3.0 起即被忽略并弃用。请传入一个打开文件对象来控制文件的打开方式。

compresslevel 形参成为仅限关键字参数。

在 3.10 版本发生变更: 这个类在面对多个同时读取器和写入器时是线程安全的，就如它在*gzip* 和*lzma* 中的等价类所具有的特性一样。

13.3.2 增量压缩和解压

class bz2.**BZ2Compressor**(*compresslevel=9*)

创建一个新的压缩器对象。此对象可被用来执行增量数据压缩。对于一次性压缩，请改用*compress()* 函数。

如果给定 *compresslevel*，它必须为 1 至 9 之间的整数。默认值为 9。

compress(*data*)

向压缩器对象提供数据。在可能的情况下返回一段已压缩数据，否则返回空字节串。

当你已结束向压缩器提供数据时，请调用*flush()* 方法来完成压缩进程。

flush()

结束压缩进程，返回内部缓冲中剩余的压缩完成的数据。

调用此方法之后压缩器对象将不可再被使用。

class bz2.**BZ2Decompressor**

创建一个新的解压缩器对象。此对象可被用来执行增量数据解压缩。对于一次性解压缩，请改用*decompress()* 函数。

> **备注**
>
> 这个类不会透明地处理包含多个已压缩数据流的输入，这不同于*decompress()* 和*BZ2File*。如果你需要通过*BZ2Decompressor* 来解压缩多个数据流输入，你必须为每个数据流都使用新的解压缩器。

decompress(*data*, *max_length=-1*)

解压缩 *data* (一个*bytes-like object*)，返回字节串形式的解压缩数据。某些 *data* 可以在内部被缓冲，以便用于后续的*decompress()* 调用。返回的数据应当与之前任何*decompress()* 调用的输出进行拼接。

如果 *max_length* 为非负数，将返回至多 *max_length* 个字节的解压缩数据。如果达到此限制并且可以产生后续输出，则*needs_input* 属性将被设为 `False`。在这种情况下，下一次*decompress()* 调用提供的 *data* 可以为 `b''` 以获取更多的输出。

如果所有输入数据都已被解压缩并返回（或是因为它少于 *max_length* 个字节，或是因为 *max_length* 为负数），则*needs_input* 属性将被设为 `True`。

在到达数据流末尾之后再尝试解压缩数据会引发*EOFError*。在数据流末尾之后获取的任何数据都会被忽略并存储至*unused_data* 属性。

在 3.5 版本发生变更: 添加了 *max_length* 形参。

eof

若达到了数据流的末尾标记则为 `True`。

Added in version 3.3.

unused_data

在压缩数据流的末尾之后获取的数据。

如果在达到数据流末尾之前访问此属性，其值将为 `b''`。

needs_input

如果在要求新的未解压缩输入之前*decompress()* 方法可以提供更多的解压缩数据则为 `False`。

Added in version 3.5.

13.3.3 一次性压缩或解压缩

bz2.**compress**(*data*, *compresslevel=9*)

压缩 *data*，此参数为一个字节类对象。

如果给定 *compresslevel*，它必须为 1 至 9 之间的整数。默认值为 9。

对于增量压缩，请改用*BZ2Compressor*。

bz2.**decompress**(*data*)

解压缩 *data*，此参数为一个字节类对象。

如果 *data* 是多个压缩数据流的拼接，则解压缩所有数据流。

对于增量解压缩，请改用*BZ2Decompressor*。

在 3.3 版本发生变更: 支持了多数据流的输入。

13.3.4 用法示例

以下是*bz2* 模块典型用法的一些示例。

使用*compress()* 和*decompress()* 来显示往复式的压缩：

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

使用*BZ2Compressor* 进行增量压缩：

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

上面的示例使用了一个相当” 非随机” 的数据流（即 `b"z"` 块的数据流）。随机数据的压缩率通常很差，而有序、重复的数据通常会产生很高的压缩率。

用二进制模式写入和读取 bzip2 压缩文件：

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
...
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
...
>>> content == data  # Check equality to original object after round-trip
True
```

13.4 lzma --- 使用 LZMA 算法进行压缩

Added in version 3.3.

源代码： Lib/lzma.py

此模块提供了可以压缩和解压缩使用 LZMA 压缩算法的数据的类和便携函数。其中还包含支持 **xz** 工具所使用的 `.xz` 和旧式 `.lzma` 文件格式的文件接口，以及相应的原始压缩数据流。

此模块所提供了接口与*bz2* 模块的非常类似。请注意*LZMAFile* 和*bz2.BZ2File* 都 不是线程安全的，因此如果你需要在多个线程中使用单个*LZMAFile* 实例，则需要通过锁来保护它。

exception lzma.**LZMAError**

当在压缩或解压缩期间或是在初始化压缩器/解压缩器的状态期间发生错误时此异常会被引发。

13.4.1 读写压缩文件

lzma.**open**(*filename*, *mode='rb'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

以二进制或文本模式打开 LZMA 压缩文件，返回一个*file object*。

filename 参数可以是一个实际的文件名（以*str*, *bytes* 或*路径类* 对象的形式给出），在此情况下会打开指定名称的文件，或者可以是一个用于读写的现有文件对象。

mode 参数可以是二进制模式的 `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` 或 `"ab"`，或者文本模式的 `"rt"`, `"wt"`, `"xt"` 或 `"at"`。默认值为 `"rb"`。

当打开一个文件用于读取时，*format* 和 *filters* 参数具有与*LZMADecompressor* 的参数相同的含义。在此情况下，*check* 和 *preset* 参数不应被使用。

当打开一个文件用于写入的，*format*, *check*, *preset* 和 *filters* 参数具有与*LZMACompressor* 的参数相同的含义。

对于二进制模式，这个函数等价于*LZMAFile* 构造器: `LZMAFile(filename, mode, ...)`。在这种情况下，不可提供 *encoding*, *errors* 和 *newline* 参数。

对于文本模式，将会创建一个*LZMAFile* 对象，并将它包装到一个*io.TextIOWrapper* 实例中，此实例带有指定的编码格式、错误处理行为和行结束符。

在 3.4 版本发生变更: 增加了对 `"x"`, `"xb"` 和 `"xt"` 模式的支持。

在 3.6 版本发生变更: 接受一个*path-like object*。

class lzma.**LZMAFile**(*filename=None*, *mode='r'*, *, *format=None*, *check=-1*, *preset=None*, *filters=None*)

以二进制模式打开一个 LZMA 压缩文件。

LZMAFile 可以包装在一个已打开的*file object* 中，或者是在给定名称的文件上直接操作。*filename* 参数指定所包装的文件对象，或是要打开的文件名称（类型为*str*, *bytes* 或*路径类* 对象）。如果是包装现有的文件对象，被包装的文件在*LZMAFile* 被关闭时将不会被关闭。

mode 参数可以是表示读取的 "r" (默认值)，表示覆写的 "w"，表示单独创建的 "x"，或表示添加的 "a"。这些模式还可以分别以 "rb", "wb", "xb" 和 "ab" 的等价形式给出。

如果 *filename* 是一个文件对象（而不是实际的文件名），则 "w" 模式并不会截断文件，而会等价于 "a"。

当打开一个文件用于读取时，输入文件可以为多个独立压缩流的拼接。它们会被作为单个逻辑流被透明地解码。

当打开一个文件用于读取时，*format* 和 *filters* 参数具有与*LZMADecompressor* 的参数相同的含义。在此情况下，*check* 和 *preset* 参数不应被使用。

当打开一个文件用于写入的，*format*, *check*, *preset* 和 *filters* 参数具有与*LZMACompressor* 的参数相同的含义。

LZMAFile 支持*io.BufferedIOBase* 所指定的所有成员，但*detach()* 和*truncate()* 除外。并支持迭代和 with 语句。

还提供了下列方法和属性：

peek(*size=-1*)

返回缓冲的数据而不前移文件位置。至少将返回一个字节的数据，除非已经到达 EOF。实际返回的字节数不确定（会忽略 *size* 参数）。

> **备注**
>
> 虽然调用*peek()* 不会改变*LZMAFile* 的文件位置，但它可能改变下层文件对象的位置（举例来说如果*LZMAFile* 是通过传入一个文件对象作为 *filename* 的话）。

mode

'rb' 表示可读而 'wb' 表示可写。

Added in version 3.13.

name

lzma 文件名。等价于下层*file object* 的*name* 属性。

Added in version 3.13.

在 3.4 版本发生变更: 增加了对 "x" 和 "xb" 模式的支持。

在 3.5 版本发生变更: *read()* 方法现在接受 None 作为参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

13.4.2 在内存中压缩和解压缩数据

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

创建一个压缩器对象，此对象可被用来执行增量压缩。

压缩单个数据块的更便捷方式请参阅*compress()*。

format 参数指定应当使用哪种容器格式。可能的值有：

- **FORMAT_XZ: .xz 容器格式**。
  这是默认格式。

- **FORMAT_ALONE: 传统的 .lzma 容器格式。**
  这种格式相比 .xz 更为受限 -- 它不支持一致性检查或多重过滤器。
- **FORMAT_RAW: 原始数据流，不使用任何容器格式。**
  这个格式描述器不支持一致性检查，并且要求你必须指定一个自定义的过滤器链（用于压缩和解压缩）。此外，以这种方式压缩的数据不可使用 FORMAT_AUTO 来解压缩 (参见*LZMADecompressor*)。

check 参数指定要包含在压缩数据中的一致性检查类型。这种检查在解压缩时使用，以确保数据没有被破坏。可能的值是:

- CHECK_NONE: 没有一致性检查。这是 FORMAT_ALONE 和 FORMAT_RAW 的默认值（也是唯一可接受的值）。
- CHECK_CRC32: 32 位循环冗余检查。
- CHECK_CRC64: 64 位循环冗余检查。这是 FORMAT_XZ 的默认值。
- CHECK_SHA256: 256 位安全哈希算法。

如果指定的检查不受支持，则会引发*LZMAError*。

压缩设置可被指定为一个预设的压缩等级（通过 *preset* 参数）或以自定义过滤器链来详细设置（通过 *filters* 参数）。

preset 参数（如果提供）应当为一个 0 到 9 (包括边界) 之间的整数，可以选择与常数 PRESET_EXTREME 进行 OR 运算。如果 *preset* 和 *filters* 均未给出，则默认行为是使用 PRESET_DEFAULT (预设等级 6)。更高的预设等级会产生更小的输出，但会使得压缩过程更缓慢。

> **备注**
>
> 除了更加 CPU 密集，使用更高的预设等级来压缩还需要更多的内存（并产生需要更多内存来解压缩的输出）。例如使用预设等级 9 时，一个*LZMACompressor* 对象的开销可以高达 800 MiB。出于这样的原因，通常最好是保持使用默认预设等级。

filters 参数（如果提供）应当指定一个过滤器链。详情参见指定自定义的过滤器链。

compress(*data*)

压缩 *data* (一个*bytes* object)，返回包含针对输入的至少一部分已压缩数据的*bytes* 对象。一部 *data* 可能会被放入内部缓冲区，以便用于后续的*compress()* 和*flush()* 调用。返回的数据应当与之前任何*compress()* 调用的输出进行拼接。

flush()

结束压缩进程，返回包含保存在压缩器的内部缓冲区中的任意数据的*bytes* 对象。

调用此方法之后压缩器将不可再被使用。

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

创建一个压缩器对象，此对象可被用来执行增量解压缩。

一次性解压缩整个压缩数据流的更便捷方式请参阅*decompress()*。

format 参数指定应当被使用的容器格式。默认值为 FORMAT_AUTO，它可以解压缩 .xz 和 .lzma 文件。其他可能的值为 FORMAT_XZ, FORMAT_ALONE 和 FORMAT_RAW。

memlimit 参数指定解压缩器可以使用的内存上限（字节数）。当使用此参数时，如果不可能在给定内存上限之内解压缩输入数据则解压缩将失败并引发*LZMAError*。

filters 参数指定用于创建被解压缩数据流的过滤器链。此参数在 *format* 为 FORMAT_RAW 时要求提供，但对于其他格式不应使用。有关过滤器链的更多信息请参阅指定自定义的过滤器链。

> **备注**
>
> 这个类不会透明地处理包含多个已压缩数据流的输入，这不同于*decompress()* 和*LZMAFile*。要通过*LZMADecompressor* 来解压缩多个数据流输入，你必须为每个数据流都创建一个新的解压缩器。

decompress(*data*, *max_length=-1*)

解压缩 *data* (一个*bytes-like object*)，返回字节串形式的解压缩数据。某些 *data* 可以在内部被缓冲，以便用于后续的*decompress()* 调用。返回的数据应当与之前任何*decompress()* 调用的输出进行拼接。

如果 *max_length* 为非负数，将返回至多 *max_length* 个字节的解压缩数据。如果达到此限制并且可以产生后续输出，则*needs_input* 属性将被设为 `False`。在这种情况下，下一次*decompress()* 调用提供的 *data* 可以为 `b''` 以获取更多的输出。

如果所有输入数据都已被解压缩并返回（或是因为它少于 *max_length* 个字节，或是因为 *max_length* 为负数），则*needs_input* 属性将被设为 `True`。

在到达数据流末尾之后再尝试解压缩数据会引发*EOFError*。在数据流末尾之后获取的任何数据都会被忽略并存储至*unused_data* 属性。

在 3.5 版本发生变更: 添加了 *max_length* 形参。

check

输入流使用的一致性检查的 ID。这可能为 `CHECK_UNKNOWN` 直到已解压了足够的输入数据来确定它所使用的一致性检查。

eof

若达到了数据流的末尾标记则为 `True`。

unused_data

在压缩数据流的末尾之后获取的数据。

在达到数据流末尾之前，这个值将为 `b""`。

needs_input

如果在要求新的未解压缩输入之前*decompress()* 方法可以提供更多的解压缩数据则为 `False`。

Added in version 3.5.

`lzma`.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

压缩 *data* (一个*bytes* 对象)，返回包含压缩数据的*bytes* 对象。

参见上文的*LZMACompressor* 了解有关 *format*, *check*, *preset* 和 *filters* 参数的说明。

`lzma`.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

解压缩 *data* (一个*bytes* 对象)，返回包含解压缩数据的*bytes* 对象。

如果 *data* 是多个单独压缩数据流的拼接，则解压缩所有相应数据流，并返回结果的拼接。

参见上文的*LZMADecompressor* 了解有关 *format*, *memlimit* 和 *filters* 参数的说明。

13.4.3 杂项

`lzma`.**is_check_supported**(*check*)

如果本系统支持给定的一致性检查则返回 `True`。

`CHECK_NONE` 和 `CHECK_CRC32` 总是受支持。`CHECK_CRC64` 和 `CHECK_SHA256` 或许不可用，如果你正在使用基于受限制特性集编译的 **liblzma** 版本的话。

13.4.4 指定自定义的过滤器链

过滤器链描述符是由字典组成的序列，其中每个字典包含单个过滤器的 ID 和选项。每个字典必须包含键 "id"，并可能包含额外的键用来指定基于过滤器的选项。有效的过滤器 ID 如下：

- 压缩过滤器：
  - FILTER_LZMA1 (配合 FORMAT_ALONE 使用)
  - FILTER_LZMA2 (配合 FORMAT_XZ 和 FORMAT_RAW 使用)
- Delta 过滤器：
  - FILTER_DELTA
- Branch-Call-Jump (BCJ) 过滤器：
  - FILTER_X86
  - FILTER_IA64
  - FILTER_ARM
  - FILTER_ARMTHUMB
  - FILTER_POWERPC
  - FILTER_SPARC

一个过滤器链最多可由 4 个过滤器组成，并且不能为空。过滤器链中的最后一个过滤器必须为压缩过滤器，其他过滤器必须为 Delta 或 BCJ 过滤器。

压缩过滤器支持下列选项（指定为表示过滤器的字典中的附加条目）：

- preset: 压缩预设选项，用于作为未显式指定的选项的默认值的来源。
- dict_size: 以字节表示的字典大小。这应当在 4 KiB 和 1.5 GiB 之间（包含边界）。
- lc: 字面值上下文的比特数。
- lp: 字面值位置的比特数。总计值 lc + lp 必须不大于 4。
- pb: 位置的比特数；必须不大于 4。
- mode: MODE_FAST 或 MODE_NORMAL。
- nice_len: 对于一个匹配应当被视为“适宜长度”的值。这应当小于或等于 273。
- mf: 要使用的匹配查找器 -- MF_HC3, MF_HC4, MF_BT2, MF_BT3 或 MF_BT4。
- depth: 匹配查找器使用的最大查找深度。0 (默认值) 表示基于其他过滤器选项自动选择。

Delta 过滤器保存字节数据之间的差值，在特定环境下可产生更具重复性的输入。它支持一个 dist 选项，指明要减去的字节之间的差值大小。默认值为 1，即相邻字节之间的差值。

BCJ 过滤器主要作用于机器码。它们会转换机器码内的相对分支、调用和跳转以使用绝对寻址，其目标是提升冗余度以供压缩器利用。这些过滤器支持一个 start_offset 选项，指明应当被映射到输入数据开头的地址。默认值为 0。

13.4.5 例子

在已压缩的数据中读取:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

创建一个压缩文件:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

在内存中压缩文件:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

增量压缩:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# 拼接所有的结果部分:
result = b"".join([out1, out2, out3, out4])
```

写入已压缩数据到已打开的文件:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

使用自定义过滤器链创建一个已压缩文件:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile --- 操作 ZIP 归档文件

源代码: Lib/zipfile/

ZIP 文件格式是一个常用的归档与压缩标准。这个模块提供了创建、读取、写入、添加及列出 ZIP 文件的工具。任何对此模块的进阶使用都将需要理解此格式，其定义参见 PKZIP 应用程序笔记。

此模块目前不能处理分卷 ZIP 文件。它可以处理使用 ZIP64 扩展（超过 4 GB 的 ZIP 文件）的 ZIP 文件。它支持解密 ZIP 归档中的加密文件，但是目前不能创建一个加密的文件。解密非常慢，因为它是使用原生 Python 而不是 C 实现的。

这个模块定义了以下内容:

exception zipfile.**BadZipFile**

为损坏的 ZIP 文件抛出的错误。

Added in version 3.2.

exception zipfile.**BadZipfile**

BadZipFile 的别名，与旧版本 Python 保持兼容性。

自 3.2 版本弃用.

exception zipfile.**LargeZipFile**

当 ZIP 文件需要 ZIP64 功能但是未启用时会抛出此错误。

class zipfile.**ZipFile**

用于读写 ZIP 文件的类。欲了解构造函数的描述，参阅段落*ZipFile 对象*。

class zipfile.**Path**

实现了*pathlib.Path* 所提供接口的一个子集的类，包括完整的*importlib.resources.abc.Traversable* 接口。

Added in version 3.8.

class zipfile.**PyZipFile**

用于创建包含 Python 库的 ZIP 归档的类。

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)

用于表示档案内一个成员信息的类。此类的实例会由*ZipFile* 对象的*getinfo()* 和*infolist()* 方法返回。大多数*zipfile* 模块的用户都不必创建它们，只需使用此模块所创建的实例。*filename* 应当是档案成员的全名，*date_time* 应当是包含六个字段的描述最近修改时间的元组；这些字段的描述请参阅*ZipInfo 对象*。

在 3.13 版本发生变更: 增加了公有的 `compress_level` 属性来暴露之前被保护 `_compresslevel`。较旧的被保护名称可继续作为保持向下兼容性的特征属性使用。

zipfile.**is_zipfile**(*filename*)

根据文件的 Magic Number，如果 *filename* 是一个有效的 ZIP 文件则返回 `True`，否则返回 `False`。*filename* 也可能是一个文件或类文件对象。

在 3.1 版本发生变更: 支持文件或类文件对象。

zipfile.**ZIP_STORED**

未被压缩的归档成员的数字常数。

zipfile.**ZIP_DEFLATED**

常用的 ZIP 压缩方法的数字常数。需要*zlib* 模块。

zipfile.**ZIP_BZIP2**

BZIP2 压缩方法的数字常数。需要*bz2* 模块。

Added in version 3.3.

zipfile.**ZIP_LZMA**

LZMA 压缩方法的数字常数。需要*lzma* 模块。

Added in version 3.3.

备注

ZIP 文件格式规范包括自 2001 年以来对 bzip2 压缩的支持，以及自 2006 年以来对 LZMA 压缩的支持。但是，一些工具（包括较旧的 Python 版本）不支持这些压缩方法，并且可能拒绝完全处理 ZIP 文件，或者无法提取单个文件。

参见

PKZIP 应用程序笔记
Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

Info-ZIP 主页
有关 Info-ZIP 项目的 ZIP 存档程序和开发库的信息。

13.5.1 ZipFile 对象

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, ***, *strict_timestamps=True*, *metadata_encoding=None*)

打开一个 ZIP 文件，*file* 为一个指向文件的路径（字符串），一个类文件对象或者一个*path-like object*。

形参 *mode* 应当为 `'r'` 来读取一个存在的文件，`'w'` 来截断并写入新的文件，`'a'` 来添加到一个存在的文件，或者 `'x'` 来仅新建并写入新的文件。如果 *mode* 为 `'x'` 并且 *file* 指向已经存在的文件，则抛出*FileExistsError*。如果 *mode* 为 `'a'` 且 *file* 为已存在的文件，则格外的文件将被加入。如果 *file* 不指向 ZIP 文件，之后一个新的 ZIP 归档将被追加为此文件。这是为了将 ZIP 归档添加到另一个文件（例如 `python.exe`）。如果 *mode* 为 `'a'` 并且文件不存在，则会新建。如果 *mode* 为 `'r'` 或 `'a'`，则文件应当可定位。

compression 是在写入归档时要使用的 ZIP 压缩方法，应为*ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* 或*ZIP_LZMA*；不可识别的值将导致引发*NotImplementedError*。如果指定了*ZIP_DEFLATED*, *ZIP_BZIP2* 或*ZIP_LZMA* 但相应的模块 (*zlib*, *bz2* 或*lzma*) 不可用，则会引发*RuntimeError*。默认值为*ZIP_STORED*。

如果 *allowZip64* 为 `True` (默认值) 则当 zipfile 大于 4 GiB 时 zipfile 将创建使用 ZIP64 扩展的 ZIP 文件。如果该参数为 `false` 则当 ZIP 文件需要 ZIP64 扩展时*zipfile* 将引发异常。

compresslevel 形参控制在将文件写入归档时要使用的压缩等级。当使用*ZIP_STORED* 或*ZIP_LZMA* 时无压缩效果。当使用*ZIP_DEFLATED* 时接受整数 `0` 至 `9` (更多信息参见*zlib*)。当使用*ZIP_BZIP2* 时接受整数 `1` 至 `9` (更多信息参见*bz2*)。

strict_timestamps 参数在设为 `False` 时允许压缩早于 1980-01-01 的文件，代价时会将时间戳设为 1980-01-01。类似的行为也会对晚于 2107-12-31 的文件发生，时间戳也会被设为该上限值。

当 mode 为 `'r'` 时，可以将 *metadata_encoding* 设为某个编解码器的名称，它将被用来解码元数据如成员名称和 ZIP 注释等等。

如果创建文件时使用 `'w'`, `'x'` 或 `'a'` 模式并且未向归档添加任何文件就执行了*closed*，则会将适当的空归档 ZIP 结构写入文件。

ZipFile 也是一个上下文管理器，因此支持 `with` 语句。在这个示例中，*myzip* 将在 `with` 语句块执行完成之后被关闭 --- 即使是发生了异常:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

备注

metadata_encoding 是用于 ZipFile 的实例级设置。目前无法在成员层级上设置此选项。

该属性是对旧式实现的变通处理，它产生的归档文件名会使用当前语言区域编码格式或代码页（主要是在 Windows 上）。根据.ZIP 标准，元数据的编码格式可以通过归档文件标头中的一个旗标指定为 IBM 代码页（默认）或 UTF-8。该旗标优先于 *metadata_encoding*，后者是一个 Python 专属的扩展。

在 3.2 版本发生变更: 添加了将*ZipFile* 用作上下文管理器的功能。

在 3.3 版本发生变更: 添加了对*bzip2* 和*lzma* 压缩的支持。

在 3.4 版本发生变更: 默认启用 ZIP64 扩展。

在 3.5 版本发生变更: 添加了对不可查找数据流的支持。并添加了对 `'x'` 模式的支持。

在 3.6 版本发生变更: 在此之前，对于不可识别的压缩值将引发普通的*RuntimeError*。

在 3.6.2 版本发生变更: *file* 形参接受一个*path-like object*。

在 3.7 版本发生变更: 添加了 *compresslevel* 形参。

在 3.8 版本发生变更: *strict_timestamps* 仅限关键字形参。

在 3.11 版本发生变更: 增加了对指定成员名称编码格式的支持以便在 ZIP 文件的目录和文件标头中读取元数据。

ZipFile.**close**()

关闭归档文件。你必须在退出程序之前调用*close()* 否则将不会写入关键记录数据。

ZipFile.**getinfo**(*name*)

返回一个*ZipInfo* 对象，其中包含有关归档成员 *name* 的信息。针对一个目前并不包含于归档中的名称调用*getinfo()* 将会引发*KeyError*。

ZipFile.**infolist**()

返回一个列表，其中包含每个归档成员的*ZipInfo* 对象。如果是打开一个现有归档则这些对象的排列顺序与它们对应条目在磁盘上的实际 ZIP 文件中的顺序一致。

ZipFile.**namelist**()

返回按名称排序的归档成员列表。

ZipFile.**open**(*name*, *mode='r'*, *pwd=None*, ***, *force_zip64=False*)

以二进制文件型对象的形式访问一个归档成员。*name* 可以是归档内某个文件的名称或是某个*ZipInfo* 对象。如果包括了 *mode* 形参，则它必须为 `'r'` (默认值) 或 `'w'`。*pwd* 是用于解密*bytes* 对象形式的已加密 ZIP 文件的密码。

open() 也是一个上下文管理器，因此支持 `with` 语句:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

如果 *mode* 为 `'r'` 则文件型对象 (`ZipExtFile`) 将为只读并且提供下列方法: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, *__iter__()*, *__next__()*。这些对象可独立于 ZipFile 进行操作。

如果 `mode='w'` 则返回一个可写入的文件句柄，它将支持*write()* 方法。当一个可写入的文件句柄被打开时，尝试读写 ZIP 文件中的其他文件将会引发*ValueError*。

在两种情况下该文件型对象还具有属性 `name`，它等价于归档内文件的名称，以及 `mode`，它根据输入模式的不同可能为 `'rb'` 或 `'wb'`。

当写入一个文件时，如果文件大小不能预先确定但是可能超过 2 GiB，可传入 `force_zip64=True` 以确保标头格式能够支持超大文件。如果文件大小可以预先确定，则在构造*ZipInfo* 对象时应设置*file_size*，并将其用作 *name* 形参。

备注

open(), *read()* 和*extract()* 方法可接受文件名或*ZipInfo* 对象。当尝试读取一个包含重复名称成员的 ZIP 文件时你将发现此功能很有好处。

在 3.6 版本发生变更: 移除了对 `mode='U'` 的支持。请使用*io.TextIOWrapper* 以在*universal newlines* 模式中读取已压缩的文本文件。

在 3.6 版本发生变更: 现在*ZipFile.open()* 可以被用来配合 `mode='w'` 选项将文件写入归档。

在 3.6 版本发生变更: 在已关闭的 ZipFile 上调用*open()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

在 3.13 版本发生变更: 为可写文件型对象增加了属性 `name` 和 `mode`。可读文件型对象的 `mode` 属性值由 `'r'` 改为 `'rb'`。

ZipFile.**extract**(*member*, *path=None*, *pwd=None*)

从归档中提取一个成员放入当前工作目录；*member* 必须是一个成员的完整名称或*ZipInfo* 对象。成员的文件信息会尽可能精确地被提取。*path* 指定一个要放入的不同目录。*member* 可以是一个文件名或*ZipInfo* 对象。*pwd* 是*bytes* 对象形式的用于解密已加密文件的密码。

返回所创建的经正规化的路径（对应于目录或新文件）。

备注

如果一个成员文件名为绝对路径，则将去掉驱动器/UNC 共享点和前导的（反）斜杠，例如：`///foo/bar` 在 Unix 上将变为 `foo/bar`，而 `C:\foo\bar` 在 Windows 上将变为 `foo\bar`。并且一个成员文件名中的所有 `".."` 都将被移除，例如：`../../foo../../ba..r` 将变为 `foo../ba..r`。在 Windows 上非法字符 (`:`, `<`, `>`, `|`, `"`, `?`, and `*`) 会被替换为下划线 (`_`)。

在 3.6 版本发生变更: 在已关闭的 ZipFile 上调用*extract()* 将引发*ValueError*。在之前的版本中则将引发*RuntimeError*。

在 3.6.2 版本发生变更: *path* 形参接受一个*path-like object*。

ZipFile.**extractall**(*path=None*, *members=None*, *pwd=None*)

从归档中提取出所有成员放入当前工作目录。*path* 指定一个要放入的不同目录。*members* 为可选项且必须为*namelist()* 所返回列表的一个子集。*pwd* 是*bytes* 对象形式的用于解密已加密文件的密码。

警告

绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能在 *path* 之外创建文件，例如某些成员具有以 `"/"` 开始的文件名或带有两个点号 `".."` 的文件名。此模块会尝试防止这种情况。参见*extract()* 的注释。

在 3.6 版本发生变更: 在已关闭的 ZipFile 上调用*extractall()* 将引发*ValueError*。在之前的版本中则将引发*RuntimeError*。

在 3.6.2 版本发生变更: *path* 形参接受一个*path-like object*。

ZipFile.**printdir**()

将归档的目录表打印到 `sys.stdout`。

ZipFile.**setpassword**(*pwd*)

将 *pwd* (一个*bytes* 对象) 设为用于提取已加密文件的默认密码。

ZipFile.**read**(*name*, *pwd=None*)

返回归档中文件 *name* 的字节数据。*name* 是归档中文件的名称，或是一个*ZipInfo* 对象。归档必须以读取或追加模式打开。*pwd* 为*bytes* 对象形式的用于解密已加密文件的密码，并且如果指定了该参数则它将覆盖通过*setpassword()* 设置的默认密码。在使用*ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* 以外的压缩方法的 ZipFile 上调用*read()* 将引发*NotImplementedError*。如果相应的压缩模块不可用也会引发错误。

在 3.6 版本发生变更: 在已关闭的 ZipFile 上调用*read()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

ZipFile.**testzip**()

读取归档中的所有文件并检查它们的 CRC 和文件头。返回第一个已损坏文件的名称，在其他情况下则返回 `None`。

在 3.6 版本发生变更: 在已关闭的 ZipFile 上调用*testzip()* 将引发*ValueError*。在之前的版本中则将引发*RuntimeError*。

ZipFile.**write**(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)

将名为 *filename* 的文件写入归档，给予的归档名为 *arcname* (默认情况下将与 *filename* 一致，但是不带驱动器盘符并会移除开头的路径分隔符)。*compress_type* 如果给出，它将覆盖作为构造器 *compression* 形参对于新条目所给出的值。类似地，*compresslevel* 如果给出也将覆盖构造器。归档必须使用 `'w'`, `'x'` 或 `'a'` 模式打开。

备注

ZIP 文件标准在历史上并未指定元数据编码格式，但是强烈建议使用 CP437（原始 IBM PC 编码格式）来实现互操作性。最近的版本允许（仅）使用 UTF-8。在这个模块中，如果成员名称包含任何非 ASCII 字符则将自动使用 UTF-8 来写入它们。不可能用 ASCII 或 UTF-8 以外的任何其他编码格式来写入成员名称。

备注

归档名称应当是基于归档根目录的相对路径，也就是说，它们不应以路径分隔符开头。

备注

如果 `arcname` (或 `filename`，如果 `arcname` 未给出) 包含一个空字节，则归档中该文件的名称将在空字节位置被截断。

备注

文件名开头有一个斜杠可能导致存档文件无法在 Windows 系统上的某些 zip 程序中打开。

在 3.6 版本发生变更: 在使用 `'r'` 模式创建的 ZipFile 或已关闭的 ZipFile 上调用 *write()* 将引发 *ValueError*。在之前的版本中则会引发 *RuntimeError*。

ZipFile.**writestr**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)

将一个文件写入归档。内容为 *data*，它可以是一个 *str* 或 *bytes* 的实例；如果是 *str*，则会先使用 UTF-8 进行编码。*zinfo_or_arcname* 可以是它在归档中将被给予的名称，或者是 *ZipInfo* 的实例。如果它是一个实例，则至少必须给定文件名、日期和时间。如果它是一个名称，则日期和时间会被设为当前日期和时间。归档必须以 `'w'`, `'x'` 或 `'a'` 模式打开。

如果给定了 *compress_type*，它将会覆盖作为新条目构造器的 *compression* 形参或在 *zinfo_or_arcname* (如果是一个 *ZipInfo* 实例) 中所给出的值。类似地，如果给定了 *compresslevel*，它将会覆盖构造器。

备注

当传入一个 *ZipInfo* 实例作为 *zinfo_or_arcname* 形参时，所使用的压缩方法将为在给定的 *ZipInfo* 实例的 *compress_type* 成员中指定的方法。默认情况下，*ZipInfo* 构造器将将此成员设为 *ZIP_STORED*。

在 3.2 版本发生变更: *compress_type* 参数。

在 3.6 版本发生变更: 在使用 `'r'` 模式创建的 ZipFile 或已关闭的 ZipFile 上调用 *writestr()* 将引发 *ValueError*。在之前的版本中则会引发 *RuntimeError*。

ZipFile.**mkdir**(*zinfo_or_directory*, *mode=511*)

在归档文件内创建一个目录。如果 *zinfo_or_directory* 是一个字符串，则会在归档文件中以 *mode* 参数指定的模式创建目录。但是，如果 *zinfo_or_directory* 是一个 *ZipInfo* 实例则 *mode* 参数将被忽略。

归档文件必须以 'w', 'x' 或 'a' 模式打开。

Added in version 3.11.

以下数据属性也是可用的:

ZipFile.**filename**

ZIP 文件的名称。

ZipFile.**debug**

要使用的调试输出等级。这可以设为从 0 (默认无输出) 到 3 (最多输出) 的值。调试信息会被写入 sys.stdout。

ZipFile.**comment**

关联到 ZIP 文件的*bytes* 对象形式的说明。如果将说明赋给以 'w', 'x' 或 'a' 模式创建的*ZipFile* 实例，它的长度不应超过 65535 字节。超过此长度的说明将被截断。

13.5.2 Path 对象

class zipfile.**Path**(*root*, *at*='')

根据 root zipfile (它可以是一个*ZipFile* 实例或适合传给*ZipFile* 构造器的 file) 构造一个 Path 对象。

at 指定此 Path 在 zipfile 中的位置，例如'dir/file.txt', 'dir/' 或''。默认为空字符串，即指定跟目录。

> **备注**
>
> *Path* 类不会对 ZIP 归档内的文件名做无害化处理。不同于*ZipFile.extract()* 和*ZipFile.extractall()* 方法，对文件名做验证或无害化处理以防止路径遍历漏洞（例如包含".." 或绝对路径的文件名）是调用方的责任。当处理不受信任的归档时，请考虑使用*os.path.abspath()* 来解析文件名并使用*os.path.commonpath()* 来检查目标路径。

Path 对象会公开*pathlib.Path* 对象的下列特性:

Path 对象可以使用 / 运算符或 joinpath 来进行遍历。

Path.**name**

最终的路径组成部分。

Path.**open**(*mode='r'*, ***, *pwd*, ****)

在当前路径上唤起*ZipFile.open()*。允许通过支持的模式打开用于读取或写入文本或二进制数据: 'r', 'w', 'rb', 'wb'。当以文本模式打开时位置和关键字参数会被传给*io.TextIOWrapper*，在其他情况下则会被忽略。pwd 是要传给*ZipFile.open()* 的 pwd 形参。

在 3.9 版本发生变更: 增加了对以文本和二进制模式打开的支持。现在默认为文本模式。

在 3.11.2 版本发生变更: encoding 形参可以作为位置参数来提供而不会引起*TypeError*。这种情况在 3.9 中是会发生的。需要与未打补丁的 3.10 和 3.11 版保持兼容的代码必须将所有*io.TextIOWrapper* 参数，包括 encoding 作为关键字参数传入。

Path.**iterdir**()

枚举当前目录的子目录。

Path.**is_dir**()

如果当前上下文引用了一个目录则返回 True。

Path.**is_file**()

如果当前上下文引用了一个文件则返回 True。

Path.**is_symlink**()

如果当前上下文引用了一个符号链接则返回 True。

Added in version 3.12.

在 3.13 版本发生变更: 在之前版本中，is_symlink 将无条件地返回 False。

Path.**exists**()

如果当前上下文引用了 zip 文件内的一个文件或目录则返回 True。

Path.**suffix**

最终组件末尾的以点号分隔的部分，如果存在的话。这通常被称为文件扩展名。

Added in version 3.11: 添加了*Path.suffix* 特征属性。

Path.**stem**

路径的末尾部分，不带文件后缀。

Added in version 3.11: 添加了*Path.stem* 特征属性。

Path.**suffixes**

由路径后缀组成的列表，通常被称为文件扩展名。

Added in version 3.11: 添加了*Path.suffixes* 特征属性。

Path.**read_text**(*, **)

读取当前文件为 unicode 文本。位置和关键字参数会被传递给*io.TextIOWrapper* (buffer 除外，它将由上下文确定)。

在 3.11.2 版本发生变更: encoding 形参可以作为位置参数来提供而不会引起*TypeError*。这种情况在 3.9 中是会发生的。需要与未打补丁的 3.10 和 3.11 版保持兼容的代码必须将所有*io.TextIOWrapper* 参数，包括 encoding 作为关键字参数传入。

Path.**read_bytes**()

读取当前文件为字节串。

Path.**joinpath**(*other*)

返回一个新的 Path 对象，其中合并了每个 *other* 参数。以下代码是等价的:

```
>>> Path(...).joinpath('child').joinpath('grandchild')
>>> Path(...).joinpath('child', 'grandchild')
>>> Path(...) / 'child' / 'grandchild'
```

在 3.10 版本发生变更: 在 3.10 之前，joinpath 未被写入文档并且只接受一个形参。

zipp 项目向较旧版本的 Python 提供了最新路径对象功能的向下移植。为尽早应用这些改变请使用 zipp.Path 来替代 zipfile.Path。

13.5.3 PyZipFile 对象

PyZipFile 构造器接受与*ZipFile* 构造器相同的形参，以及一个额外的形参 *optimize*。

class zipfile.**PyZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)

在 3.2 版本发生变更: 增加了 *optimize* 形参。

在 3.4 版本发生变更: 默认启用 ZIP64 扩展。

实例在*ZipFile* 对象所具有的方法以外还附加了一个方法:

writepy(*pathname*, *basename=''*, *filterfunc=None*)

查找 *.py 文件并将相应的文件添加到归档。

如果*PyZipFile* 的 *optimize* 形参未给定或为 -1，则相应的文件为 *.pyc 文件，并在必要时进行编译。

如果*PyZipFile* 的 *optimize* 形参为 0, 1 或 2，则限具有相应优化级别 (参见*compile()*) 的文件会被添加到归档，并在必要时进行编译。

如果 *pathname* 是文件，则文件名必须以 .py 为后缀，并且只有 (相应的 *.pyc) 文件会被添加到最高层级 (不带路径信息)。如果 *pathname* 不是以 .py 为后缀的文件，则将引发*RuntimeError*。如果它是目录，并且该目录不是一个包目录，则所有的 *.pyc 文件会被添加到最高层级。如果目录是一个包目录，则所有的 *.pyc 会被添加到包名所表示的文件路径下，并且如果有任何子目录为包目录，则会以排好的顺序递归地添加这些目录。

basename 仅限在内部使用。

如果给定 *filterfunc*，则它必须是一个接受单个字符串参数的函数。在将其添加到归档之前它将被传入每个路径（包括每个单独的完整路径）。如果 *filterfunc* 返回假值，则路径将不会被添加，而如果它是一个目录则其内容将被忽略。例如，如果我们的测试文件全都位于 test 目录或以字符串 test_ 打头，则我们可以使用一个 *filterfunc* 来排除它们:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
...
>>> zf.writepy('myprog', filterfunc=notests)
```

writepy() 方法会产生带有这样一些文件名的归档:

```
string.pyc                      # 最高层级名称
test/__init__.pyc               # 包目录
test/testall.pyc                # 模块 test.testall
test/bogus/__init__.pyc         # 子包目录
test/bogus/myfile.pyc           # 子模块 test.bogus.myfile
```

在 3.4 版本发生变更: 增加了 *filterfunc* 形参。

在 3.6.2 版本发生变更: *pathname* 形参接受一个*path-like object*。

在 3.7 版本发生变更: 递归排序目录条目。

13.5.4 ZipInfo 对象

ZipInfo 类的实例会通过*getinfo()* 和*ZipFile* 对象的*infolist()* 方法返回。每个对象将存储关于 ZIP 归档的一个成员的信息。

有一个类方法可以为文件系统文件创建*ZipInfo* 实例:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*, *, *strict_timestamps=True*)

为文件系统中的文件构造一个*ZipInfo* 实例，并准备将其添加到一个 zip 文件。

filename 应为文件系统中某个文件或目录的路径。

如果指定了 *arcname*，它会被用作归档中的名称。如果未指定 *arcname*，则所用名称与 *filename* 相同，但将去除任何驱动器盘符和打头的路径分隔符。

strict_timestamps 参数在设为 False 时允许压缩早于 1980-01-01 的文件，代价时会将时间戳设为 1980-01-01。类似的行为也会对晚于 2107-12-31 的文件发生，时间戳也会被设为该上限值。

Added in version 3.6.

在 3.6.2 版本发生变更: *filename* 形参接受一个*path-like object*。

在 3.8 版本发生变更: 增加了 *strict_timestamps* 仅限关键字形参。

实例具有下列方法和属性:

ZipInfo.**is_dir**()

如果此归档成员是一个目录则返回 True。

这会使用条目的名称：目录应当总是以 / 结尾。

Added in version 3.6.

ZipInfo.**filename**
归档中的文件名称。

ZipInfo.**date_time**
上次修改存档成员的时间和日期。这是六个值的元组：

索引	值
0	Year (>= 1980)
1	月（1 为基数）
2	月份中的日期（1 为基数）
3	小时（0 为基数）
4	分钟（0 为基数）
5	秒（0 为基数）

备注

ZIP 文件格式不支持 1980 年以前的时间戳。

ZipInfo.**compress_type**
归档成员的压缩类型。

ZipInfo.**comment**
bytes 对象形式的单个归档成员的注释。

ZipInfo.**extra**
扩展字段数据。PKZIP Application Note 包含一些保存于该*bytes* 对象中的内部结构的注释。

ZipInfo.**create_system**
创建 ZIP 归档所用的系统。

ZipInfo.**create_version**
创建 ZIP 归档所用的 PKZIP 版本。

ZipInfo.**extract_version**
需要用来提取归档的 PKZIP 版本。

ZipInfo.**reserved**
必须为零。

ZipInfo.**flag_bits**
ZIP 标志位。

ZipInfo.**volume**
文件头的分卷号。

ZipInfo.**internal_attr**
内部属性。

ZipInfo.**external_attr**
外部文件属性。

ZipInfo.**header_offset**
文件头的字节偏移量。

ZipInfo.**CRC**
未压缩文件的 CRC-32。

ZipInfo.**compress_size**
已压缩数据的大小。

ZipInfo.**file_size**
未压缩文件的大小。

13.5.5 命令行接口

zipfile 模块提供了简单的命令行接口用于与 ZIP 归档的交互。

如果你想要创建一个新的 ZIP 归档，请在*-c* 选项后指定其名称然后列出应当被包含的文件名:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

传入一个目录也是可接受的:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

如果你想要将一个 ZIP 归档提取到指定的目录，请使用*-e* 选项:

```
$ python -m zipfile -e monty.zip target-dir/
```

要获取一个 ZIP 归档中的文件列表，请使用*-l* 选项:

```
$ python -m zipfile -l monty.zip
```

命令行选项

-l <zipfile>

--list <zipfile>
列出一个 zipfile 中的文件名。

-c <zipfile> <source1> ... <sourceN>

--create <zipfile> <source1> ... <sourceN>
基于源文件创建 zipfile。

-e <zipfile> <output_dir>

--extract <zipfile> <output_dir>
将 zipfile 提取到目标目录中。

-t <zipfile>

--test <zipfile>
检测 zipfile 是否有效。

--metadata-encoding <encoding>
为*-l*, *-e* 和*-t* 指定成员名称的编码格式。

Added in version 3.11.

13.5.6 解压缩的障碍

zipfile 模块的提取操作可能会由于下面列出的障碍而失败。

由于文件本身

解压缩可能由于不正确的密码 / CRC 校验和 / ZIP 格式或不受支持的压缩 / 解密方法而失败。

文件系统限制

超出特定文件系统上的限制可能会导致解压缩失败。例如目录条目所允许的字符、文件名的长度、路径名的长度、单个文件的大小以及文件的数量等等。

资源限制

缺乏内存或磁盘空间将会导致解压缩失败。例如，作用于 zipfile 库的解压缩炸弹 (即 ZIP bomb) 就可能造成磁盘空间耗尽。

中断

在解压缩期间中断执行，例如按下 ctrl-C 或杀死解压缩进程可能会导致归档文件的解压缩不完整。

提取的默认行为

不了解提取的默认行为可能导致不符合期望的解压缩结果。例如，当提取相同归档两次时，它会不经询问地覆盖文件。

13.6 `tarfile` --- 读写 tar 归档文件

源代码: Lib/tarfile.py

---

tarfile 模块可以用来读写 tar 归档，包括使用 gzip, bz2 和 lzma 压缩的归档。请使用*zipfile* 模块来读写 `.zip` 文件，或者使用*shutil* 的高层级函数。

一些事实和数字:

- 读写*gzip*, *bz2* 和*lzma* 解压的归档要求相应的模块可用。
- 支持读取 / 写入 POSIX.1-1988 (ustar) 格式。
- 对 GNU tar 格式的读/写支持，包括 *longname* 和 *longlink* 扩展，对所有种类 *sparse* 扩展的只读支持，包括 sparse 文件的恢复。
- 对 POSIX.1-2001 (pax) 格式的读/写支持。
- 处理目录、正常文件、硬链接、符号链接、fifo 管道、字符设备和块设备，并且能够获取和恢复文件信息例如时间戳、访问权限和所有者等。

在 3.3 版本发生变更: 添加了对*lzma* 压缩的支持。

在 3.12 版本发生变更: 归档文件使用过滤器 来提取，这将可以限制令人惊讶/危险的特性，或确认它们符合预期并且归档文档受到完全信任。在默认情况下，归档文档将受到完全信任，但此默认选项已被弃用并计划在 Python 3.14 中改变。

`tarfile.`**`open`**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

针对路径名 *name* 返回*TarFile* 对象。有关*TarFile* 对象以及所允许的关键字参数的详细信息请参阅*TarFile 对象*。

mode 必须是 `'filemode[:compression]'` 形式的字符串，其默认值为 `'r'`。以下是模式组合的完整列表:

模式	action
`'r' or 'r:*'`	打开和读取使用透明压缩（推荐）。
`'r:'`	打开和读取不使用压缩。
`'r:gz'`	打开和读取使用 gzip 压缩。
`'r:bz2'`	打开和读取使用 bzip2 压缩。
`'r:xz'`	打开和读取使用 lzma 压缩。
`'x'` 或 `'x:'`	单独创建一个 tarfile 而不带压缩。如果它已经存在则会引发*FileExistsError*异常。
`'x:gz'`	使用 gzip 压缩创建一个 tarfile。如果它已经存在则会引发*FileExistsError*异常。
`'x:bz2'`	使用 bzip2 压缩创建一个 tarfile。如果它已经存在则会引发*FileExistsError*异常。
`'x:xz'`	使用 lzma 压缩创建一个 tarfile。如果它已经存在则会引发*FileExistsError*异常。
`'a' or 'a:'`	打开以便在没有压缩的情况下追加。如果文件不存在，则创建该文件。
`'w' or 'w:'`	打开用于未压缩的写入。
`'w:gz'`	打开用于 gzip 压缩的写入。
`'w:bz2'`	打开用于 bzip2 压缩的写入。
`'w:xz'`	打开用于 lzma 压缩的写入。

请注意 `'a:gz'`, `'a:bz2'` 或 `'a:xz'` 是不可能的组合。如果 *mode* 不适用于打开特定（压缩的）文件用于读取，则会引发*ReadError*。请使用 *mode* `'r'` 来避免这种情况。如果某种压缩方法不受支持，则会引发*CompressionError*。

如果指定了 *fileobj*，它会被用作对应于 *name* 的以二进制模式打开的*file object* 的替代。它会被设定为处在位置 0。

对于 `'w:gz'`, `'x:gz'`, `'w|gz'`, `'w:bz2'`, `'x:bz2'`, `'w|bz2'` 等模式，*tarfile.open()* 接受关键字参数 *compresslevel* (默认值为 9) 用于指定文件的压缩等级。

对于 `'w:xz'` 和 `'x:xz'` 模式，*tarfile.open()* 接受关键字参数 *preset* 来指定文件的压缩等级。

针对特殊的目的，还存在第二种 *mode* 格式: `'filemode|[compression]'`。*tarfile.open()* 将返回一个将其数据作为数据块流来处理的*TarFile* 对象。对此文件将不能执行随机查找。如果给定了 *fileobj*，它可以是任何具有*read()* 或*write()* 方法（由 *mode* 确定）的对象。*bufsize* 指定块大小，默认为 `20 * 512` 字节。可与此格式组合使用的有 `sys.stdin.buffer`、套接字*file object* 或磁盘设备等。但是，这样的*TarFile* 对象存在不允许随机访问的限制，参见例子。当前可用的模式有:

模式	动作
`'r|*'`	打开 tar 块的 流以进行透明压缩读取。
`'r|'`	打开一个未压缩的 tar 块的 *stream* 用于读取。
`'r|gz'`	打开一个 gzip 压缩的 *stream* 用于读取。
`'r|bz2'`	打开一个 bzip2 压缩的 *stream* 用于读取。
`'r|xz'`	打开一个 lzma 压缩 *stream* 用于读取。
`'w|'`	打开一个未压缩的 *stream* 用于写入。
`'w|gz'`	打开一个 gzip 压缩的 *stream* 用于写入。
`'w|bz2'`	打开一个 bzip2 压缩的 *stream* 用于写入。
`'w|xz'`	打开一个 lzma 压缩的 *stream* 用于写入。

在 3.5 版本发生变更: 添加了 `'x'` (单独创建) 模式。

在 3.6 版本发生变更: *name* 形参接受一个*path-like object*。

在 3.12 版本发生变更: *compresslevel* 关键字参数也适用于流式数据。

class tarfile.**TarFile**

用于读取和写入 tar 归档的类。请不要直接使用这个类：而要使用*tarfile.open()*。参见*TarFile 对象*。

tarfile.**is_tarfile**(*name*)

如果 *name* 是一个*tarfile* 能读取的 tar 归档文件则返回*True*。*name* 可以为*str*，文件或文件型对象。

在 3.9 版本发生变更: 支持文件或类文件对象。

tarfile 模块定义了以下异常:

exception tarfile.**TarError**

所有*tarfile* 异常的基类。

exception tarfile.**ReadError**

当一个不能被*tarfile* 模块处理或者因某种原因而无效的 tar 归档被打开时将被引发。

exception tarfile.**CompressionError**

当一个压缩方法不受支持或者当数据无法被正确解码时将被引发。

exception tarfile.**StreamError**

当达到流式*TarFile* 对象的典型限制时将被引发。

exception tarfile.**ExtractError**

当使用*TarFile.extract()* 时针对 *non-fatal* 所引发的异常，但是仅限*TarFile.errorlevel*== 2。

exception tarfile.**HeaderError**

如果获取的缓冲区无效则会由*TarInfo.frombuf()* 引发的异常。

exception tarfile.**FilterError**

被过滤器*拒绝* 的成员的基类。

tarinfo

关于过滤器拒绝提取的成员的信息，为*TarInfo* 类型。

exception tarfile.**AbsolutePathError**

在拒绝提取具有绝对路径的成员时引发。

exception tarfile.**OutsideDestinationError**

在拒绝提取目标目录以外的成员时引发。

exception tarfile.**SpecialFileError**

在拒绝提取特殊文件（例如设备或管道）时引发。

exception tarfile.**AbsoluteLinkError**

在拒绝提取具有绝对路径的符号链接时引发。

exception tarfile.**LinkOutsideDestinationError**

在拒绝提取指向目标目录以外的符号链接时引发。

以下常量在模块层级上可用:

tarfile.**ENCODING**

默认的字符编码格式: 在 Windows 上为 'utf-8'，其他系统上则为*sys.getfilesystemencoding()* 所返回的值。

tarfile.**REGTYPE**

tarfile.**AREGTYPE**

常规文件*type*。

tarfile.**LNKTYPE**

（tar 文件中的）链接*type*。

tarfile.**SYMTYPE**
符号链接*type*。

tarfile.**CHRTYPE**
字符特殊设备*type*。

tarfile.**BLKTYPE**
块特殊设备*type*。

tarfile.**DIRTYPE**
目录*type*。

tarfile.**FIFOTYPE**
FIFO 特殊设备*type*。

tarfile.**CONTTYPE**
连续文件*type*。

tarfile.**GNUTYPE_LONGNAME**
GNU tar 长名称*type*。

tarfile.**GNUTYPE_LONGLINK**
GNU tar 长链接*type*。

tarfile.**GNUTYPE_SPARSE**
A GNU tar 离散文件*type*。

以下常量各自定义了一个*tarfile* 模块能够创建的 tar 归档格式。相关细节请参阅*受支持的 tar 格式* 小节。

tarfile.**USTAR_FORMAT**
POSIX.1-1988 (ustar) 格式。

tarfile.**GNU_FORMAT**
GNU tar 格式。

tarfile.**PAX_FORMAT**
POSIX.1-2001 (pax) 格式。

tarfile.**DEFAULT_FORMAT**
用于创建归档的默认格式。目前为*PAX_FORMAT*。

在 3.8 版本发生变更: 新归档的默认格式已更改为*PAX_FORMAT* 而不再是*GNU_FORMAT*。

参见

模块*zipfile*
zipfile 标准模块的文档。

归档操作
标准*shutil* 模块所提供的高层级归档工具的文档。

GNU tar manual, Basic Tar Format
针对 tar 归档文件的文档，包含 GNU tar 扩展。

13.6.1 TarFile 对象

TarFile 对象提供了一个 tar 归档的接口。一个 tar 归档就是数据块的序列。一个归档成员（被保存文件）是由一个标头块加多个数据块组成的。一个文件可以在一个 tar 归档中多次被保存。每个归档成员都由一个*TarInfo* 对象来代表，详情参见*TarInfo 对象*。

TarFile 对象可在 with 语句中作为上下文管理器使用。当语句块结束时它将自动被关闭。请注意在发生异常事件时被打开用于写入的归档将不会被终结；只有内部使用的文件对象将被关闭。相关用例请参见例子。

Added in version 3.2: 添加了对上下文管理器协议的支持。

class tarfile.**TarFile**(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=1*, *stream=False*)

下列所有参数都是可选项并且也可作为实例属性来访问。

name 是归档的路径名。*name* 可以是一个*path-like object*。如果给定了 *fileobj* 则它可以被省略。在此情况下，如果对象存在 name 属性则将使用它。

mode 可以为 'r' 表示从现有归档读取，'a' 表示将数据追加到现有文件，'w' 表示创建新文件覆盖现有文件，或者 'x' 表示仅在文件不存在时创建新文件。

如果给定了 *fileobj*，它会被用于读取或写入数据。如果可以被确定，则 *mode* 会被 *fileobj* 的模式所覆盖。*fileobj* 的使用将从位置 0 开始。

> **备注**
>
> 当*TarFile* 被关闭时，*fileobj* 不会被关闭。

format 控制用于写入的归档格式。它必须为在模块层级定义的常量*USTAR_FORMAT*, *GNU_FORMAT* 或*PAX_FORMAT* 中的一个。当读取时，格式将被自动检测，即使单个归档中存在不同的格式。

tarinfo 参数可以被用来将默认的*TarInfo* 类替换为另一个。

如果 *dereference* 为*False*，则会将符号链接和硬链接添加到归档中。如果为*True*，则会将目标文件的内容添加到归档中。在不支持符号链接的系统上参数将不起作用。

如果 *ignore_zeros* 为*False*，则会将空的数据块当作归档的末尾来处理。如果为*True*，则会跳过空的（和无效的）数据块并尝试获取尽可能多的成员。此参数仅适用于读取拼接的或损坏的归档。

debug 可设为从 0 (无调试消息) 到 3 (全部调试消息)。消息会被写入到 sys.stderr。

errorlevel 控制如何处理解压错误，参见相应的属性。

encoding 和 *errors* 参数定义了读取或写入归档所使用的字符编码格式以及要如何处理转换错误。默认设置将适用于大多数用户。要深入了解详情可参阅*Unicode 问题* 小节。

可选的 *pax_headers* 参数是字符串的字典，如果 *format* 为*PAX_FORMAT* 它将被作为 pax 全局标头被添加。

如果 *stream* 被设为*True* 则在读取时有关归档中文件的归档信息不会被缓存，以节省内存消耗。

在 3.2 版本发生变更: 使用 'surrogateescape' 作为 *errors* 参数的默认值。

在 3.5 版本发生变更: 添加了 'x' (单独创建) 模式。

在 3.6 版本发生变更: *name* 形参接受一个*path-like object*。

在 3.13 版本发生变更: 增加了 *stream* 形参。

classmethod TarFile.**open**(...)

作为替代的构造器。*tarfile.open()* 函数实际上是这个类方法的快捷方式。

TarFile.**getmember**(*name*)

返回成员 *name* 的*TarInfo* 对象。如果 *name* 在归档中找不到，则会引发*KeyError*。

> **备注**
>
> 如果一个成员在归档中出现超过一次，它的最后一次出现会被视为是最新的版本。

TarFile.**getmembers**()

以*TarInfo* 对象列表的形式返回归档的成员。列表的顺序与归档中成员的顺序一致。

TarFile.**getnames**()

以名称列表的形式返回成员。它的顺序与*getmembers()* 所返回列表的顺序一致。

TarFile.**list**(*verbose=True*, ***, *members=None*)

将内容清单打印到 `sys.stdout`。如果 *verbose* 为*False*，则将只打印成员名称。如果为*True*，则输出将类似于 **`ls -l`** 的输出效果。如果给定了可选的 *members*，它必须为*getmembers()* 所返回的列表的一个子集。

在 3.5 版本发生变更: 添加了 *members* 形参。

TarFile.**next**()

当*TarFile* 被打开用于读取时，以*TarInfo* 对象的形式返回归档的下一个成员。如果不再有可用对象则返回*None*。

TarFile.**extractall**(*path='.'*, *members=None*, ***, *numeric_owner=False*, *filter=None*)

将归档中的所有成员提取到当前工作目录或 *path* 目录。如果给定了可选的 *members*，则它必须为*getmembers()* 所返回的列表的一个子集。字典信息例如所有者、修改时间和权限会在所有成员提取完毕后被设置。这样做是为了避免两个问题：目录的修改时间会在每当在其中创建文件时被重置。并且如果目录的权限不允许写入，提取文件到目录的操作将失败。

如果 *numeric_owner* 为*True*，则将使用来自 tarfile 的 uid 和 gid 数值来设置被提取文件的所有者/用户组。在其他情况下，则会使用来自 tarfile 的名称值。

filter 参数指明在提取之前要如何修改或拒绝 `members`。请参阅解压缩过滤器 了解详情。建议应根据你需要支持的 *tar* 特征显式地设置该参数。

> **警告**
>
> 绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能在 *path* 之外创建文件，例如某些成员具有以 `"/"` 开始的绝对路径文件名或带有两个点号 `".."` 的文件名。
>
> 设置 `filter='data'` 来防止最危险的安全问题，并请参阅解压缩过滤器 一节了解详情。section for details.

在 3.5 版本发生变更: 添加了 *numeric_owner* 形参。

在 3.6 版本发生变更: *path* 形参接受一个*path-like object*。

在 3.12 版本发生变更: 添加了 *filter* 形参。

TarFile.**extract**(*member*, *path=''*, *set_attrs=True*, ***, *numeric_owner=False*, *filter=None*)

从归档中提取出一个成员放入当前工作目录，将使用其完整名称。成员的文件信息会尽可能精确地被提取。*member* 可以是一个文件名或*TarInfo* 对象。你可以使用 *path* 指定一个不同的目录。*path* 可以是一个*path-like object*。将会设置文件属性 (owner, mtime, mode) 除非 *set_attrs* 为假值。

numeric_owner 和 *filter* 参数与*extractall()* 中的相同。

> **备注**
>
> *extract()* 方法不会处理某些提取问题。在大多数情况下你应当考虑使用*extractall()* 方法。

> **警告**
>
> 查看*extractall()* 的警告信息。
>
> 设置 `filter='data'` 来防止最危险的安全问题，并请参阅解压缩过滤器 一节了解详情。section for details.

在 3.2 版本发生变更: 添加了 *set_attrs* 形参。

在 3.5 版本发生变更: 添加了 *numeric_owner* 形参。

在 3.6 版本发生变更: *path* 形参接受一个*path-like object*。

在 3.12 版本发生变更: 添加了 *filter* 形参。

TarFile.**extractfile**(*member*)

将归档中的一个成员提取为文件对象。*member* 可以是一个文件名或*TarInfo* 对象。如果 *member* 是一个常规文件或链接，则会返回一个*io.BufferedReader* 对象。对于所有其他现有成员，则都将返回*None*。如果 *member* 未在归档中出现，则会引发*KeyError*。

在 3.3 版本发生变更: 返回一个*io.BufferedReader* 对象。

在 3.13 版本发生变更: 返回的*io.BufferedReader* 对象具有 `mode` 属性并且总是会等于 `'rb'`。

TarFile.**errorlevel**: *int*

如果 *errorlevel* 为 `0`，则在使用*TarFile.extract()* 和*TarFile.extractall()* 时错误会被忽略。不过，当 *debug* 大于 0 时它们将会作为错误消息在调试输出中出现。如果 *errorlevel* 为 "1"(默认值)*，则所有 **fatal* 错误都会作为*OSError* 或*FilterError* 异常被引发。如果为 `2`，则所有 *non-fatal* 错误也会作为*TarError* 异常被引发。

某些异常，如参数类型错误或数据损坏导致的异常，总是会被触发。

自定义提取过滤器 应针对 *fatal* 错误引发*FilterError*，针对 *non-fatal* 错误引发*ExtractError*。

请注意，当出现异常时，存档可能会被部分提取。用需要户负责进行清理。

TarFile.**extraction_filter**

Added in version 3.12.

被用作*extract()* 和*extractall()* 的 *filter* 参数的默认值的提取过滤器。

该属性可以为 `None` 或是一个可调用对象。与*extract()* 的 *filter* 参数不同，该属性不允许使用字符串名称。

如果 `extraction_filter` 为 `None` (默认值)，则不带 *filter* 参数调用提取方法将引发 `DeprecationWarning`，并回退至*fully_trusted* 过滤器，其危险行为与之前版本的 Python 一致。

在 Python 3.14+ 中，保持 `extraction_filter=None` 将导致提取方法默认使用*data* 过滤器。

该属性可在实例上设置或在子类中覆盖。也可以在 `TarFile` 类本身上设置它以设置一个全局默认值，不过，由于它会影响 *tarfile* 的所有使用，最好的做法是只在最高层级应和程序或站点配置 中这样做。要以这种方式设置全局默认值，需要将一个过滤器函数包装在*staticmethod()* 中以防止 `self` 参数的注入。

TarFile.**add**(*name*, *arcname=None*, *recursive=True*, *, *filter=None*)

将文件 *name* 添加到归档。*name* 可以为任意类型的文件（目录、fifo、符号链接等等）。如果给出 *arcname* 则它将为归档中的文件指定一个替代名称。默认情况下会递归地添加目录。这可以通过将 *recursive* 设为*False* 来避免。递归操作会按排序顺序添加条目。如果给定了 *filter*，它应当为一个接受*TarInfo* 对象并返回已修改*TarInfo* 对象的函数。如果它返回*None* 则*TarInfo* 对象将从归档中被排除。具体示例参见例子。

在 3.2 版本发生变更: 添加了 *filter* 形参。

在 3.7 版本发生变更: 递归操作按排序顺序添加条目。

TarFile.**addfile**(*tarinfo*, *fileobj=None*)

将*TarInfo* 对象 *tarinfo* 添加到归档中。如果 *tarinfo* 代表一个大小不为零的常规文件，则 *fileobj* 参数应为一个*binary file*，且会从中读取 `tarinfo.size` 个字节并添加到归档中。你可以直接创建*TarInfo* 对象，或者也可以使用*gettarinfo()*。

在 3.13 版本发生变更: 对于大小不为零的常规文件必须给出 *fileobj*。

TarFile.**gettarinfo**(*name=None*, *arcname=None*, *fileobj=None*)

基于*os.stat()* 的结果或者现有文件的相同数据创建一个*TarInfo*。文件或者是命名为 *name*，或者是使用文件描述符指定为一个*file object* *fileobj*。*name* 可以是一个*path-like object*。如果给定了 *arcname*，则它将为归档中的文件指定一个替代名称，在其他情况下，名称将从 *fileobj* 的*name* 属性或 *name* 参数获取。名称应当是一个文本字符串。

你可以在使用*addfile()* 添加*TarInfo* 的某些属性之前修改它们。如果文件对象不是从文件开头进行定位的普通文件对象，*size* 之类的属性就可能需要修改。例如*GzipFile* 之类的文件就属于这种情况。*name* 也可以被修改，在这种情况下 *arcname* 可以是一个占位字符串。

在 3.6 版本发生变更: *name* 形参接受一个*path-like object*。

TarFile.**close**()

关闭*TarFile*。在写入模式下，会向归档添加两个表示结束的零数据块。

TarFile.**pax_headers**: ***dict***

一个包含 pax 全局标头的键值对的字典。

13.6.2 TarInfo 对象

TarInfo 对象代表*TarFile* 中的一个文件。除了会存储所有必要的文件属性（例如文件类型、大小、时间、权限、所有者等），它还提供了一些确定文件类型的有用方法。此对象 *并不* 包含文件数据本身。

TarInfo 对象可通过*TarFile* 的方法*getmember()*, *getmembers()* 和*gettarinfo()* 返回。

修改*getmember()* 或*getmembers()* 返回的对象会影响在上的所有后续操作。对于不想要这样的场景，你可以使用*copy.copy()* 或调用*replace()* 方法一次性创建修改后的副本。

部分属性可以设为 None 以表示一些元数据未被使用或未知。不同的*TarInfo* 方法会以不同的方式处理 None:

- *extract()* 或*extractall()* 方法会忽略相应的元数据，让其保持默认设置。
- *addfile()* 将会失败。
- *list()* 将打印一个占位字符串。

class tarfile.**TarInfo**(*name=''*)

创建一个*TarInfo* 对象。

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)

基于字符串缓冲区 *buf* 创建并返回一个*TarInfo* 对象。

如果缓冲区无效则会引发*HeaderError*。

classmethod TarInfo.**fromtarfile**(*tarfile*)

从*TarFile* 对象 *tarfile* 读取下一个成员并将其作为*TarInfo* 对象返回。

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)

基于*TarInfo* 对象创建一个字符串缓冲区。有关参数的信息请参见*TarFile* 类的构造器。

在 3.2 版本发生变更: 使用 'surrogateescape' 作为 *errors* 参数的默认值。

TarInfo 对象具有以下公有数据属性:

TarInfo.**name**: ***str***

归档成员的名称。

TarInfo.**size**: ***int***

以字节表示的大小。

TarInfo.**mtime**: ***int*** | ***float***

以*Unix 纪元* 秒数表示的最近修改时间，与*os.stat_result.st_mtime* 相同。

在 3.12 版本发生变更: 对于*extract()* 和*extractall()* 可设为 None，以使解压缩操作跳过应用此属性。

TarInfo.**mode**: *int*

权限比特位，与*os.chmod()* 相同。

在 3.12 版本发生变更: 对于*extract()* 和*extractall()* 可设为 None，以使解压缩操作跳过应用此属性。

TarInfo.**type**

文件类型。*type* 通常为以下常量之一: *REGTYPE*, *AREGTYPE*, *LNKTYPE*, *SYMTYPE*, *DIRTYPE*, *FIFOTYPE*, *CONTTYPE*, *CHRTYPE*, *BLKTYPE*, *GNUTYPE_SPARSE*。要更方便地确定一个*TarInfo* 对象的类型，请使用下述的 is*() 方法。

TarInfo.**linkname**: *str*

目标文件名的名称，该属性仅在类型为*LNKTYPE* 和*SYMTYPE* 的*TarInfo* 对象中存在。

对于符号链接 (SYMTYPE)，*linkname* 是相对于包含链接的目录的。对于硬链接 (LNKTYPE)，*linkname* 则是相对于存档根目录的。

TarInfo.**uid**: *int*

最初保存该成员的用户的用户 ID。

在 3.12 版本发生变更: 对于*extract()* 和*extractall()* 可设为 None，以使解压缩操作跳过应用此属性。

TarInfo.**gid**: *int*

最初保存该成员的用户的分组 ID。

在 3.12 版本发生变更: 对于*extract()* 和*extractall()* 可设为 None，以使解压缩操作跳过应用此属性。

TarInfo.**uname**: *str*

用户名。

在 3.12 版本发生变更: 对于*extract()* 和*extractall()* 可设为 None，以使解压缩操作跳过应用此属性。

TarInfo.**gname**: *str*

分组名。

在 3.12 版本发生变更: 对于*extract()* 和*extractall()* 可设为 None，以使解压缩操作跳过应用此属性。

TarInfo.**chksum**: *int*

标头校验和。

TarInfo.**devmajor**: *int*

设备主编号。

TarInfo.**devminor**: *int*

设备次编号。

TarInfo.**offset**: *int*

tar 标头从这里开始。

TarInfo.**offset_data**: *int*

文件的数据从这里开始。

TarInfo.**sparse**

离散的成员信息。

TarInfo.**pax_headers**: *dict*

一个包含所关联的 pax 扩展标头的键值对的字典。

TarInfo.**replace**(*name=...*, *mtime=...*, *mode=...*, *linkname=...*, *uid=...*, *gid=...*, *uname=...*, *gname=...*, *deep=True*)

Added in version 3.12.

返回修改了给定属性的 TarInfo 对象的 新副本。例如，要返回组名设为 'staff' 的 TarInfo，请使用:

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

在默认情况下，将执行深拷贝。如果 *deep* 为假值，则执行浅拷贝，即 pax_headers 及任何自定义属性都与原始 TarInfo 对象共享。

TarInfo 对象还提供了一些便捷查询方法:

TarInfo.**isfile**()

如果*TarInfo* 对象为普通文件则返回*True*。

TarInfo.**isreg**()

与*isfile()* 相同。

TarInfo.**isdir**()

如果为目录则返回*True*。

TarInfo.**issym**()

如果为符号链接则返回*True*。

TarInfo.**islnk**()

如果为硬链接则返回*True*。

TarInfo.**ischr**()

如果为字符设备则返回*True*。

TarInfo.**isblk**()

如果为块设备则返回*True*。

TarInfo.**isfifo**()

如果为 FIFO 则返回*True*。.

TarInfo.**isdev**()

如果为字符设备、块设备或 FIFO 之一则返回*True*。

13.6.3 解压缩过滤器

Added in version 3.12.

tar 格式的设计旨在捕捉类 UNIX 文件系统的所有细节，这使其功能非常强大。不幸的是，这些特性也使得很容易创建在解压缩时产生意想不到的 -- 甚至可能是恶意的 -- 影响的 tar 文件。举例来说，解压缩 tar 文件时可以通过各种方式覆盖任意文件（例如通过使用绝对路径、.. 路径组件或影响后续成员的符号链接等）。

在大多数情况下，并不需要全部的功能。因此，*tarfile* 支持提取过滤器：一种限制功能的机制，从而避免一些安全问题。

参见

PEP 706
包含设计背后进一步的动机和理由。

TarFile.extract() 或*extractall()* 的 *filter* 参数可以是:

- 字符串 'fully_trusted': 尊重归档文件中指定的所有元数据。如果用户完全信任该归档，或实现了自己的复杂验证则应使用此过滤器。

- 字符串 'tar': 尊重大多数 *tar* 专属的特性（即类 UNIX 文件系统的功能），但阻止极有可能令人惊讶的或恶意的功能。详情参见*tar_filter()*。
- 字符串 'data': 忽略或阻止大多数类 UNIX 文件系统专属的特性。用于提取跨平台数据归档文件。详情参见*data_filter()*。
- None (默认): 使用*TarFile.extraction_filter*。

  如果这也为 None (默认值)，则引发 DeprecationWarning，并回退为 'fully_trusted' 过滤器，其危险行为与之前版本的 Python 一致。

  在 Python 3.14 中，'data' 过滤器将变成默认选项。也可以提前切换，参见*TarFile.extraction_filter*。
- 该可调用对象将针每个被提取的成员执行调用并附带一个*TarInfo* 来描述该成员以及被提取归档文件的目标路径（即供所有成员使用的相同路径）：

  ```
  filter(member: TarInfo, path: str, /) -> TarInfo | None
  ```

  该可调用对象会在提取每个成员之前被调用，因此它能够将磁盘的当前状态考虑在内。它可以：
  - 返回一个*TarInfo* 对象，该对象将被用来代替归档文件中的元数据，或者
  - 返回 None，在这种情况下该成员将被跳过，或者
  - 根据*errorlevel* 的值引发一个异常以中止操作或跳过成员。请注意当提取操作中止时，*extractall()* 可能会保留部分已提取的归档文件。它不会尝试执行清理。

默认的命名过滤器

预定义的命名过滤器可作为函数使用，因此它们可在自定义过滤器中被重用：

tarfile.**fully_trusted_filter**(*member*, *path*)

不加修改地返回 *member*。

实现 'fully_trusted' 过滤器。

tarfile.**tar_filter**(*member*, *path*)

实现 'tar' 过滤器。

- 从文件名中去除开头的斜杠 (/ 和*os.sep*)。
- 拒绝 提取具有绝对路径的文件（针对名称在去除斜杠后仍为绝对路径的情况，例如 Windows 上 C:/foo 这样的路径）。这会引发*AbsolutePathError*。
- 拒绝 提取具有位于目标以外的绝对路径（跟随符号链接之后）的文件。这会引发*OutsideDestinationError*。
- 清空高模式位 (setuid, setgid, sticky) 和 group/other 写入位 (*S_IWGRP* | *S_IWOTH*)。

返回修改后的 TarInfo 成员。

tarfile.**data_filter**(*member*, *path*)

实现 'data' 过滤器。在 tar_filter 的所具有的功能之外：

- 拒绝 提取链接到绝对路径的链接（不论是硬链接还是软链接），或链接到目标之外的链接。

  这会引发*AbsoluteLinkError* 或*LinkOutsideDestinationError*。

  请注意即使在不支持符号链接的平台上此类文件也会被拒绝。
- 拒绝 提取设备文件（包括管道）。这会引发*SpecialFileError*。
- 用于常规文件，包括硬链接：
  - 设置所有者读写权限 (*S_IRUSR* | *S_IWUSR*)。
  - 如果所有者没有 group 和 other 可执行权限 (*S_IXGRP* | *S_IXOTH*) 则移除它 (*S_IXUSR*)。
- 对于其他文件（目录），将 mode 设为 None，以便提取方法跳过应用权限位。

- 将用户和组信息 (`uid`, `gid`, `uname`, `gname`) 设为 `None`，以使得提取方法跳过对它的设置。

返回修改后的 `TarInfo` 成员。

过滤器错误

当过滤器拒绝提取文件时，它将引发一个适当的异常，即*`FilterError`* 的子类。如果*`TarFile.errorlevel`* 为 1 或更大的值则提取将中止。如果 `errorlevel=0` 则会记录错误并跳过该成员，但提取仍会继续。

进一步核验的提示

即使 `filter='data'`，*tarfile* 也不适合在没有事先检查的情况下提取不受信任的文件。除其他问题外，预定义的过滤器不能防止拒绝服务攻击。用户应当进行额外的检查。

以下是一份不完整的考虑事项列表：

- 提取到新的临时目录 以避免滥用已存在的链接等问题，并使得提取失败后更容易清理。
- 在处理不受信任的数据时，使用外部（例如操作系统层级）的磁盘、内存和 CPU 使用限制。
- 根据允许字符列表检查文件名（来过滤控制字符、易混淆字符、外来路径分隔符等）。
- 检查文件名是否有预期的扩展名（不鼓励使用在“点击”时会被执行的文件，或像 Windows 特殊设备名称这样没有扩展名的文件）。
- 限制提取文件的数量、提取数据的总大小、文件名长度（包括符号链接长度）以及单个文件的大小。
- 检查在不区分大小写的文件系统上会被屏蔽的文件。

还需要注意：

- Tar 文件可能包含同一文件的多个版本。较晚的版本会覆盖任何较早的版本。这一功能对于更新磁带归档来说至关重要，但也可能被恶意滥用。
- *tarfile* 无法为“实时”数据的问题提供保护，例如在提取（或归档）过程中攻击者对目标（或源）目录进行了改动。

支持较早的 Python 版本

提取过滤器是在 Python 3.12 中增加的，但可能会作为安全更新向下移植到较老的版本。要检查该特性是否可用，请使用 `hasattr(tarfile, 'data_filter')` 而不是检查 Python 版本。

下面的例子演示了如何支持带有和没有有该功能的 Python 版本。请注意设置 `extraction_filter` 会影响任何后续的操作。

- 完全受信任的归档:

```
my_tarfile.extraction_filter = (lambda member, path: member)
my_tarfile.extractall()
```

- 如果可用则使用 `'data'` 过滤器；如果此特性不可用，则恢复为 Python 3.11 的行为 (`'fully_trusted'`):

```
my_tarfile.extraction_filter = getattr(tarfile, 'data_filter',
                                       (lambda member, path: member))
my_tarfile.extractall()
```

- 使用 `'data'` 过滤器；如果不可用则 *fail*:

```
my_tarfile.extractall(filter=tarfile.data_filter)
```

或者:

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- 使用 `'data'` 过滤器；如果不可用则 *warn*:

```
if hasattr(tarfile, 'data_filter'):
    my_tarfile.extractall(filter='data')
else:
    # 当不再需要时移除这个
    warn_the_user('Extracting may be unsafe; consider updating Python')
    my_tarfile.extractall()
```

有状态的提取过滤器示例

tarfile 的提取方法接受一个简单的 *filter* 可调用对象，而自定义过滤器则可以是具有内部状态的更复杂对象。将其写成为下文管理器可能会很有用处，即以这样的方式使用:

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

例如，这种过滤器可以写成:

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0

    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member

    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

13.6.4 命令行接口

Added in version 3.4.

`tarfile` 模块提供了简单的命令行接口以便与 tar 归档进行交互。

如果你想要创建一个新的 tar 归档，请在`-c` 选项后指定其名称然后列出应当被包含的文件名:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

传入一个字典也是可接受的:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

如果你想要将一个 tar 归档提取到指定的目录，请使用`-e` 选项:

```
$ python -m tarfile -e monty.tar
```

你也可以通过传入目录名称将一个 tar 归档提取到不同的目录:

```
$ python -m tarfile -e monty.tar  other-dir/
```

要获取一个 tar 归档中文件的列表，请使用`-l` 选项:

```
$ python -m tarfile -l monty.tar
```

命令行选项

-l <tarfile>

--list <tarfile>

列出一个 tarfile 中的文件名。

-c <tarfile> <source1> ... <sourceN>

--create <tarfile> <source1> ... <sourceN>

基于源文件创建 tarfile。

-e <tarfile> [<output_dir>]

--extract <tarfile> [<output_dir>]

如果未指定 *output_dir* 则会将 tarfile 提取到当前目录。

-t <tarfile>

--test <tarfile>

检测 tarfile 是否有效。

-v, --verbose

更详细地输出结果。

--filter <filtername>

为 `--extract` 指定 *filter*。详情参见解压缩过滤器。只接受字符串名称 (包括 `fully_trusted`, `tar` 和 `data`)。

13.6.5 例子

如何将整个 tar 归档提取到当前工作目录:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall(filter='data')
tar.close()
```

如何通过*`TarFile.extractall()`* 使用生成器函数而非列表来提取一个 tar 归档的子集:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

如何基于一个文件名列表创建未压缩的 tar 归档:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

使用 `with` 语句的同一个示例:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

如何读取一个 gzip 压缩的 tar 归档并显示一些成员信息:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

如何创建一个归档并使用 *TarFile.add()* 中的 *filter* 形参来重置用户信息:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.6 受支持的 tar 格式

通过 *tarfile* 模块可以创建三种 tar 格式:

- The POSIX.1-1988 ustar 格式 (*USTAR_FORMAT*)。它支持最多 256 个字符的文件名长度和最多 100 个字符的链接名长度。文件大小上限为 8 GiB。这是一种老旧但广受支持的格式。
- GNU tar 格式 (*GNU_FORMAT*)。它支持长文件名和链接名、大于 8 GiB 的文件以及稀疏文件。它是 GNU/Linux 系统上的事实标准。*tarfile* 完全支持针对长名称的 GNU tar 扩展，稀疏文件支持则限制为只读。
- POSIX.1-2001 pax 格式 (*PAX_FORMAT*)。它是几乎无限制的最灵活格式。它支持长文件名和链接名，大文件以及使用可移植方式存储路径名。现代的 tar 实现，包括 GNU tar, bsdtar/libarchive 和 star，都完全支持扩展的 *pax* 特性；某些老旧或不再维护的库可能不支持，但应当会将 *pax* 归档视为广受支持的 *ustar* 格式。它是当前新建归档的默认格式。

  它扩展了现有的 *ustar* 格式，包括用于无法以其他方式存储的附加标头。存在两种形式的 pax 标头：扩展标头只影响后续的文件标头，全局标头则适用于完整归档并会影响所有后续的文件。为了便于移植，在 pax 标头中的所有数据均以 *UTF-8* 编码。

还有一些 tar 格式的其他变种，它们可以被读取但不能被创建:

- 古老的 V7 格式。这是来自 Unix 第七版的第一个 tar 格式，它只存储常规文件和目录。名称长度不能超过 100 个字符，并且没有用户/分组名信息。某些归档在带有非 ASCII 字符字段的情况下会产生计算错误的标头校验和。
- SunOS tar 扩展格式。此格式是 POSIX.1-2001 pax 格式的一个变种，但并不保持兼容。

13.6.7 Unicode 问题

最初 tar 格式被设计用来在磁带机上生成备份，主要关注于保存文件系统信息。现在 tar 归档通常用于文件分发和在网络上交换归档。最初格式（它是所有其他格式的基础）的一个问题是它没有支持不同字符编码格式的概念。例如，一个在 *UTF-8* 系统上创建的普通 tar 归档如果包含非 *ASCII* 字符则将无法在 *Latin-1* 系统上被正确读取。文本元数据（例如文件名，链接名，用户/分组名）将变为损坏状态。不幸的是，没有什么办法能够自动检测一个归档的编码格式。pax 格式被设计用来解决这个问题。它使用通用字符编码格式 *UTF-8* 来存储非 ASCII 元数据。

在 *tarfile* 中字符转换的细节由 *TarFile* 类的 *encoding* 和 *errors* 关键字参数控制。

encoding 定义了用于归档中元数据的字符编码格式。默认值为*sys.getfilesystemencoding()* 或是回退选项 'ascii'。根据归档是被读取还是被写入，元数据必须被解码或编码。如果没有正确设置 *encoding*，转换可能会失败。

errors 参数定义了不能被转换的字符将如何处理。可能的取值在错误处理方案 小节列出。默认方案为 'surrogateescape'，它也被 Python 用于文件系统调用，参见文件名，命令行参数，以及环境变量。。

对于*PAX_FORMAT* 归档（默认格式），*encoding* 通常是不必要的，因为所有元数据都使用 *UTF-8* 来存储。*encoding* 仅在解码二进制 pax 标头或存储带有替代字符的字符串等少数场景下会被使用。

CHAPTER 14

文件格式

本章中描述的模块解析各种不是标记语言且与电子邮件无关的杂项文件格式。

14.1 csv --- CSV 文件读写

源代码： Lib/csv.py

CSV (Comma Separated Values) 格式是电子表格和数据库中最常见的输入、输出文件格式。在 **RFC 4180** 规范推出的很多年前，CSV 格式就已经被开始使用了，由于当时并没有合理的标准，不同应用程序读写的数据会存在细微的差别。这种差别让处理多个来源的 CSV 文件变得困难。但尽管分隔符会变化，此类文件的大致格式是相似的，所以编写一个单独的模块以高效处理此类数据，将程序员从读写数据的繁琐细节中解放出来是有可能的。

csv 模块实现了 CSV 格式表单数据的读写。其提供了诸如“以兼容 Excel 的方式输出数据文件”或“读取 Excel 程序输出的数据文件”的功能，程序员无需知道 Excel 所采用 CSV 格式的细节。此模块同样可以用于定义其他应用程序可用的 CSV 格式或定义特定需求的 CSV 格式。

csv 模块中的*reader* 类和*writer* 类可用于读写序列化的数据。也可使用*DictReader* 类和*DictWriter* 类以字典的形式读写数据。

> **参见**
>
> **该实现在“Python 增强提议” - PEP *305* (CSV 文件 API) 中被提出**
> 《Python 增强提议》提出了对 Python 的这一补充。

14.1.1 模块内容

csv 模块定义了以下函数：

csv.**reader**(*csvfile*, *dialect='excel'*, ***fmtparams*)

返回一个*reader 对象*，该对象将处理给定 *csvfile* 中的行。csvfile 必须是一个包含字符串的可迭代对象，使用 reader 所定义的 csv 格式。csvfile 通常是一个文件型对象或列表。如果 *csvfile* 是一个文件对象，则打开它时应设置 `newline=''`.[1] 给定可选 *dialect* 形参将被用于定义一组专属于特定 CSV 变

[1] 如果没有指定 `newline=''`，则嵌入引号中的换行符将无法正确解析，并且在写入时，使用 `\r\n` 换行的平台会有多余的 `\r` 写入。由于 csv 模块会执行自己的（通用）换行符处理，因此指定 `newline=''` 应该总是安全的。

种的形参。它可以是*Dialect* 类的子类的实例，或是*list_dialects()* 函数所返回的字符串之一。另一个可选关键字形参 *fmtparams* 可被用来覆盖当前变种中的单个格式形参。有关变种和格式设置形参的完整细节，请参阅变种与格式参数 一节。

csv 文件的每一行都读取为一个由字符串组成的列表。除非指定了 QUOTE_NONNUMERIC 格式选项（在这种情况下，未加引号的字段会转换为浮点数），否则不会执行自动数据类型转换。

一个简短的用法示例:

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, *dialect='excel'*, ***fmtparams*)

返回一个 writer 对象，该对象负责将用户的数据在给定的文件型对象上转换为带分隔符的字符串。*csvfile* 可以是任何具有*write()* 方法的对象。如果 *csvfile* 是一个文件对象，则打开它时应使用 newline=''[Page 569, 1]。可以给出可选的 *dialect* 形参用来定义一组特定 CSV 变种专属的形参。它可以是*Dialect* 类的某个子类的实例或是*list_dialects()* 函数所返回的字符串之一。还可以给出另一个可选的 *fmtparams* 关键字参数来覆盖当前变种中的单个格式化形参。有关各个变种和格式化形参的完整细节，请参阅变种与格式参数 部分。为了尽量简化与实现 DB API 的模块之间的接口，可以将*None* 作为空字符串写入。虽然这个转换是不可逆的，但它可以简化 SQL NULL 数据值到 CSV 文件的转储而无需预处理从 cursor.fetch* 调用返回的数据。在被写入之前所有其他非字符串数据都会先用*str()* 来转换为字符串。

一个简短的用法示例:

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*[, ***fmtparams*]])

将 *dialect* 与 *name* 关联起来。*name* 必须是字符串。变种的指定可以通过传入一个*Dialect* 的子类，或通过 *fmtparams* 关键字参数，或是两者同时传入，此时关键字参数会覆盖 dialect 形参。有关变种和格式化形参的完整细节，请参阅变种与格式参数 部分。

csv.**unregister_dialect**(*name*)

从变种注册表中删除 *name* 对应的变种。如果 *name* 不是已注册的变种名称，则抛出*Error* 异常。

csv.**get_dialect**(*name*)

返回 *name* 对应的变种。如果 *name* 不是已注册的变种名称，则抛出*Error* 异常。该函数返回的是不可变的*Dialect* 对象。

csv.**list_dialects**()

返回所有已注册变种的名称。

csv.**field_size_limit**([*new_limit*])

返回解析器当前允许的最大字段大小。如果指定了 *new_limit*，则它将成为新的最大字段大小。

csv 模块定义了以下类:

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

创建一个对象，该对象在操作上类似于常规 reader，但是将每行中的信息映射到一个*dict*，该 dict 的键由 *fieldnames* 可选参数给出。

fieldnames 形参是一个*sequence*。如果省略 *fieldnames*，则文件 *f* 第一行中的值将用作字段名并将从结果中去除。如果提供了 *fieldnames*，它们将被使用而第一行将包括在结果中。无论字段名是如何确定的，字典都将保留其原始顺序。

如果某一行中的字段多于字段名，则剩余数据会被放入一个列表，并与 *restkey* 所指定的字段名 (默认为 None) 一起保存。如果某个非空白行的字段少于字段名，则缺失的值会使用 *restval* 的值来填充 (默认为 None)。

所有其他可选或关键字参数都传递给底层的*reader* 实例。

如果传给 *fieldnames* 的参数是一个迭代器，它将被强制转换为*list*。

在 3.6 版本发生变更: 返回的行现在的类型是 OrderedDict。

在 3.8 版本发生变更: 现在，返回的行是*dict* 类型。

一个简短的用法示例:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)

创建一个对象，该对象在操作上类似常规 writer，但会将字典映射到输出行。*fieldnames* 形参是一个由键组成的*序列*，它指定字典中要传给*writerow()* 方法并写入文件 *f* 的值的顺序。如果字典没有 *fieldnames* 中的键，则可选的 *restval* 形参将指明要写入的值。如果传递给*writerow()* 方法包含的键在 *fieldnames* 中找不到，则可选的 *extrasaction* 形参将指明要执行的操作。如果将其设为默认值 'raise'，则会引发*ValueError*。如果将其设为 'ignore'，则字典中额外的值将被忽略。任何其他可选或关键字参数都将被传递给下层的*writer* 实例。

注意，与*DictReader* 类不同，*DictWriter* 类的 *fieldnames* 参数不是可选参数。

如果传给 *fieldnames* 的参数是一个迭代器，它将被强制转换为*list*。

一个简短的用法示例:

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**

Dialect 类是一个容器类，其属性包含有如何处理双引号、空白符、分隔符等的信息。由于缺少严格的 CSV 规格描述，不同的应用程序会产生略有差别的 CSV 数据。*Dialect* 实例定义了*reader* 和*writer* 实例将具有怎样的行为。

所有可用的*Dialect* 名称会由*list_dialects()* 返回，并且它们可由特定的*reader* 和*writer* 类通过它们的初始化函数 (__init__) 来注册，例如:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
```

class csv.**excel**

excel 类定义了 Excel 生成的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel'。

class csv.**excel_tab**

excel_tab 类定义了 Excel 生成的、制表符分隔的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel-tab'。

class csv.**unix_dialect**

unix_dialect 类定义了在 UNIX 系统上生成的 CSV 文件的常规属性，即使用 '\n' 作为换行符，且所有字段都有引号包围。它在变种注册表中的名称是 'unix'。

Added in version 3.2.

class csv.**Sniffer**

Sniffer 类用于推断 CSV 文件的格式。

Sniffer 类提供了两个方法：

sniff(*sample*, *delimiters=None*)

分析给定的 *sample* 并返回一个*Dialect* 子类，该子类中包含了分析出的格式参数。如果给出可选的 *delimiters* 参数，则该参数会被解释为字符串，该字符串包含了可能的有效定界符。

has_header(*sample*)

分析 sample 文本（假定为 CSV 格式），如果发现其首行为一组列标题则返回*True*。在检查每一列时，将考虑是否满足两个关键标准之一来估计 sample 是否包含标题:

- 第二至第 n 行包含数字值
- 第二至第 n 行包含字符串值，其中至少有一个值的长度与该列预期标题的长度不同。

会对第一行之后的二十行进行采样；如果有超过一半的列 + 行符合标准，则返回*True*。

> **备注**
>
> 此方法是一个粗略的启发式方式，有可能产生错误的真值和假值。

使用*Sniffer* 的示例：

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... 在此处理 CSV 文件内容 ...
```

csv 模块定义了以下常量：

csv.**QUOTE_ALL**

指示*writer* 对象给所有字段加上引号。

csv.**QUOTE_MINIMAL**

指示*writer* 对象仅为包含特殊字符（例如 定界符、引号字符或 行结束符中的任何字符）的字段加上引号。

csv.**QUOTE_NONNUMERIC**

指示*writer* 对象为所有非数字字段加上引号。

指示*reader* 将所有未加引号的字段转换为 *float* 类型。

csv.**QUOTE_NONE**

指示*writer*对象不使用引号引出字段。当定界符出现在输出数据中时，其前面应该有转义符。如果未设置转义符，则遇到任何需要转义的字符时，writer 都会抛出*Error*异常。

指示*reader*对象不对引号字符执行特殊处理。

csv.**QUOTE_NOTNULL**

指示*writer*对象为所有不为 None 的字段加引号。这类似于*QUOTE_ALL*，区别是如果一个字段值为 None 则会写入一个（不带引号的）空字符串。

指示*reader*对象将（不带引号的）空字段解读为 None 并在其他情况下采取与*QUOTE_ALL*相同的行为。

Added in version 3.12.

csv.**QUOTE_STRINGS**

指示*writer*对象总是为字符串字段加引号。这类似于*QUOTE_NONNUMERIC*，区别是如果一个字段值为 None 则会写入一个（不带引号的）空字符串。

指示*reader*对象将(不带引号的)空字符串解读为 None 并在其他情况下采取与*QUOTE_NONNUMERIC*相同的行为。

Added in version 3.12.

*csv*模块定义了以下异常：

exception csv.**Error**

该异常可能由任何发生错误的函数抛出。

14.1.2 变种与格式参数

为了更容易地指定输入和输出记录的格式，特定的多个格式化形参将组合成为不同的 dialect。特定的 dialect 是*Dialect*类的一个子类，它包含多个用于描述 CSV 文件的格式的属性。当创建*reader*或*writer*对象时，程序员可以指定一个字符串或*Dialect*类的子类作为 dialect 形参。作为对 *dialect* 形参的补充或替代，程序员还可以指定单独的格式化形参，它们的名称与*Dialect*类所定义的以下属性相同。

Dialect 类支持以下属性：

Dialect.**delimiter**

一个用于分隔字段的单字符，默认为 ','。

Dialect.**doublequote**

控制出现在字段中的*引号字符*本身应如何被引出。当该属性为*True*时，双写引号字符。如果该属性为*False*，则在*引号字符*的前面放置*转义符*。默认值为*True*。

在输出时，如果 *doublequote* 是*False*，且*转义符*未指定，且在字段中发现*引号字符*时，会抛出*Error*异常。

Dialect.**escapechar**

一个用于 writer 的单字符，用来在 *quoting* 设置为*QUOTE_NONE*的情况下转义*定界符*，在 *doublequote* 设置为*False*的情况下转义*引号字符*。在读取时，*escapechar* 去除了其后所跟字符的任何特殊含义。该属性默认为*None*，表示禁用转义。

在 3.11 版本发生变更: 不允许空的 *escapechar*。

Dialect.**lineterminator**

放在*writer*产生的行的结尾，默认为 '\r\n'。

> **备注**
>
> *reader*经过硬编码，会识别 '\r' 或 '\n' 作为行尾，并忽略 *lineterminator*。未来可能会更改这一行为。

Dialect.**quotechar**

一个单字符，用于包住含有特殊字符的字段，特殊字符如 定界符或 引号字符或换行符。默认为 '"'。

在 3.11 版本发生变更: 不允许空的 *quotechar*。

Dialect.**quoting**

控制 writer 何时生成引号以及 reader 何时识别引号。它可以设为任意*QUOTE_** 常量 并且默认为*QUOTE_MINIMAL*。

Dialect.**skipinitialspace**

当为*True* 时，紧接在 *delimiter* 之后空格会被忽略。默认值为*False*。

Dialect.**strict**

如果为 True，则在输入错误的 CSV 时抛出*Error* 异常。默认值为 False。

14.1.3 Reader 对象

Reader 对象（*DictReader* 实例和*reader()* 函数返回的对象）具有以下公开方法：

csvreader.**__next__**()

返回 reader 的可迭代对象的下一行，它可以是一个列表（如果对象是由*reader()* 返回）或字典（如果是一个*DictReader* 实例），根据当前*Dialect* 来解析。通常你应当以 next(reader) 的形式来调用它。

Reader 对象具有以下公开属性：

csvreader.**dialect**

变种描述，只读，供解析器使用。

csvreader.**line_num**

源迭代器已经读取了的行数。它与返回的记录数不同，因为记录可能跨越多行。

DictReader 对象具有以下公开属性：

DictReader.**fieldnames**

字段名称。如果在创建对象时未传入字段名称，则首次访问时或从文件中读取第一条记录时会初始化此属性。

14.1.4 Writer 对象

writer 对象 (*DictWriter* 实例和*writer()* 函数所返回的对象) 具有以下公共方法。对于*writer* 对象 *row* 必须是输出字符串或数字的可迭代对象的数字，而对于*DictWriter* 对象则是一个将文件名映射到字符串或数字 (会先将其传给*str()*) 的字典。请注意在写入复数时会用圆括号括起来。这可能会给其他读取 CSV 文件的程序带来一些问题 (假定它们确实支持复数)。

csvwriter.**writerow**(*row*)

将 *row* 形参写入到 writer 的文件对象，根据当前*Dialect* 进行格式化。返回对下层文件对象的 *write* 方法的调用的返回值。

在 3.5 版本发生变更: 开始支持任意类型的迭代器。

csvwriter.**writerows**(*rows*)

将 *rows** （即能迭代出多个上述 **row* 对象的迭代器）中的所有元素写入 writer 的文件对象，并根据当前设置的变种进行格式化。

Writer 对象具有以下公开属性：

csvwriter.**dialect**

变种描述，只读，供 writer 使用。

DictWriter 对象具有以下公开方法：

DictWriter.**writeheader**()

在 writer 的文件对象中，写入一行字段名称（字段名称在构造函数中指定），并根据当前设置的变种进行格式化。本方法的返回值就是内部使用的*csvwriter.writerow()* 方法的返回值。

Added in version 3.2.

在 3.8 版本发生变更: 现在*writeheader()* 也返回其内部使用的*csvwriter.writerow()* 方法的返回值。

14.1.5 例子

读取 CSV 文件最简单的一个例子:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

读取其他格式的文件:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

相应最简单的写入示例是:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

由于*open()* 被用来打开 CSV 文件供读取，因此在默认情况下将使用系统默认编码格式 (参见*locale.getencoding()*) 把文件解码至 unicode。要使用其他编码格式来解码文件，请使用 open 的 `encoding` 参数:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

这同样适用于写入非系统默认编码的内容：打开输出文件时，指定 encoding 参数。

注册一个新的变种:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

Reader 的更高级用法——捕获并报告错误:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
```

(续下页)

(接上页)

```
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

尽管该模块不直接支持解析字符串，但仍可如下轻松完成:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

备注

14.2 configparser --- 配置文件解析器

源代码: Lib/configparser.py

此模块提供了它实现一种基本配置语言*ConfigParser* 类，这种语言所提供的结构与 Microsoft Windows INI 文件的类似。你可以使用这种语言来编写能够由最终用户来自定义的 Python 程序。

备注

这个库 *并不能够* 解析或写入在 Windows Registry 扩展版本 INI 语法中所使用的值-类型前缀。

参见

模块*tomllib*
TOML 是一种具有良好规范的针对应用程序配置文件的格式。它被专门设计作为 INI 改进版本。

模块*shlex*
支持创建类似 Unix shell 的同样可被用于应用程序配置文件的迷你语言。

模块*json*
json 模块实现了 JavaScript 语法的一个子集，它有时被用于配置，但是不支持注释。

14.2.1 快速起步

让我们准备一个非常基本的配置文件，它看起来是这样的:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[forge.example]
User = hg

[topsecret.server.example]
Port = 50022
ForwardX11 = no
```

INI 文件的结构描述见*以下章节*。总的来说，这种文件由多个节组成，每个节包含多个带有值的键。*configparser* 类可以读取和写入这种文件。让我们先通过程序方式来创建上述的配置文件。

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['forge.example'] = {}
>>> config['forge.example']['User'] = 'hg'
>>> config['topsecret.server.example'] = {}
>>> topsecret = config['topsecret.server.example']
>>> topsecret['Port'] = '50022'     # 更改解析器
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

如你所见，我们可以把配置解析器当作一个字典来处理。两者确实存在差异，将在后文说明，但是其行为非常接近于字典所具有一般行为。

现在我们已经创建并保存了一个配置文件，让我们再将它读取出来并探究其中包含的数据。

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['forge.example', 'topsecret.server.example']
>>> 'forge.example' in config
True
>>> 'python.org' in config
False
>>> config['forge.example']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.example']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['forge.example']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['forge.example']['ForwardX11']
'yes'
```

正如我们在上面所看到的，相关的 API 相当直观。唯一有些神奇的地方是 `DEFAULT` 小节，它为所有其他小节提供了默认值[1]。还要注意小节中的键大小写不敏感并且会存储为小写形式[1]。

将多个配置读入单个 *ConfigParser* 是可能的，其中最近添加的配置具有最高优先级。任何冲突的键都会从更近的配置获取并且先前存在的键会被保留。下面的例子读入一个 `override.ini` 文件，它将覆盖任何来自 `example.ini` 文件的冲突的键。

```
[DEFAULT]
ServerAliveInterval = -1
```

[1] 配置解析器允许重度定制。如果你有兴趣改变脚注说明中所介绍的行为，请参阅*Customizing Parser Behaviour* 一节。

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

此行为等价于一次*ConfigParser.read()* 调用并向 *filenames* 形参传入多个文件。

14.2.2 支持的数据类型

配置解析器并不会猜测配置文件中值的类型，而总是将它们在内部存储为字符串。这意味着如果你需要其他数据类型，你应当自己来转换:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

由于这种任务十分常用，配置解析器提供了一系列便捷的获取方法来处理整数、浮点数和布尔值。最后一个类型的处理最为有趣，因为简单地将值传给 `bool()` 是没有用的，`bool('False')` 仍然会是 `True`。为解决这个问题配置解析器还提供了*getboolean()*。这个方法对大小写不敏感并可识别 `'yes'`/`'no'`，`'on'`/`'off'`，`'true'`/`'false'` 和 `'1'`/`'0'`[Page 577, 1] 等布尔值。例如:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['forge.example'].getboolean('ForwardX11')
True
>>> config.getboolean('forge.example', 'Compression')
True
```

除了*getboolean()*，配置解析器还提供了同类的*getint()* 和*getfloat()* 方法。你可以注册你自己的转换器并或是定制已提供的转换器。[Page 577, 1]

14.2.3 回退值

与字典类似，你可以使用某一节的*get()* 方法来提供回退值:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

请注意默认值会优先于回退值。例如，在我们的示例中 `'CompressionLevel'` 键仅在 `'DEFAULT'` 小节中被指定。如果我们尝试从 `'topsecret.server.example'` 小节获取它，我们将总是会得到默认值，即使我们指定了一个回退值:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

还需要注意的一点是解析器层级的*get()* 方法提供了自定义的更复杂接口，它被继续维护用于向下兼容。当使用此方法时，可以通过 `fallback` 仅限关键字参数提供一个回退值:

```
>>> config.get('forge.example', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

同样的 `fallback` 参数也可在*getint()*, *getfloat()* 和*getboolean()* 方法中使用，例如:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 受支持的 INI 文件结构

配置文件是由小节组成的，每个小节都有一个 `[section]` 标头，加上多个由特定字符串 (默认为 = 或 :[Page 577, 1]) 分隔的键/值条目。在默认情况下，小节名对大小写敏感而键对大小写不敏感[Page 577, 1]。键和值开头和末尾的空格会被移除。在解释器配置允许时值可以被省略[Page 577, 1]，在此情况下键/值分隔符也可以被省略。值还可以跨越多行，只要值的其他行带有比第一行更深的缩进。依据解析器的具体模式，空白行可能会被视为多行值的组成部分或是被忽略。

在默认情况下，有效的节名称可以是不包含'\n' 的任意字符串。要改变此设定，请参阅*ConfigParser.SECTCRE*。

如果解析器通过 `allow_unnamed_section=True` 被配置为允许未命名的最高层级小节则第一个小节的名称可以省略。在这种情况下，键/值可以通过*UNNAMED_SECTION* 来获取例如 `config[UNNAMED_SECTION]`。

配置文件可以包含注释，要带有指定字符前缀 (默认为 `#` 和 `;`[Page 577, 1])。注释可以单独出现于原本的空白行，并可使用缩进。[Page 577, 1]

例如:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
```

(续下页)

(接上页)

```
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 未命名小节

第一（或唯一）小节的名称可以省略并且其值可通过*UNNAMED_SECTION* 属性来获取。

```
>>> config = """
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> unnamed = configparser.ConfigParser(allow_unnamed_section=True)
>>> unnamed.read_string(config)
>>> unnamed.get(configparser.UNNAMED_SECTION, 'option')
'value'
```

14.2.6 值的插值

在核心功能之上，*ConfigParser* 还支持插值。这意味着值可以在被 `get()` 调用返回之前进行预处理。

class configparser.**BasicInterpolation**

默认实现由*ConfigParser* 来使用。它允许值包含引用了相同小节中其他值或者特殊的默认小节中的值的格式字符串[Page 577, 1]。额外的默认值可以在初始化时提供。

例如:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# use a %% to escape the % sign (% is the only character that needs to be escaped):
gain: 80%%
```

在上面的例子里，*ConfigParser* 的 *interpolation* 设为 `BasicInterpolation()`，这会将 `%(home_dir)s` 求解为 `home_dir` 的值(在这里是 `/Users`)。`%(my_dir)s` 的将被实际求解为 `/Users/lumberjack`。所有插值都是按需进行的，这样引用链中使用的键不必以任何特定顺序在配置文件中指明。

当 `interpolation` 设为 `None` 时，解析器会简单地返回 `%(my_dir)s/Pictures` 作为 `my_pictures` 的值，并返回 `%(home_dir)s/lumberjack` 作为 `my_dir` 的值。

class configparser.**ExtendedInterpolation**

一个用于插值的替代处理程序实现了更高级的语法，它被用于 `zc.buildout` 中的实例。扩展插值使用 `${section:option}` 来表示来自外部小节的值。插值可以跨越多个层级。为了方便使用，`section:` 部分可被省略，插值会默认作用于当前小节（可能会从特殊小节获取默认值）。

例如，上面使用基本插值描述的配置，使用扩展插值将是这个样子:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# use a $$ to escape the $ sign ($ is the only character that needs to be escaped):
cost: $$80
```

来自其他小节的值也可以被获取:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.7 映射协议访问

Added in version 3.2.

映射协议访问这个通用名称是指允许以字典的方式来使用自定义对象的功能。在*configparser* 中，映射接口的实现使用了 `parser['section']['option']` 标记法。

`parser['section']` 专门为解析器中的小节数据返回一个代理。这意味着其中的值不会被拷贝，而是在需要时从原始解析器中获取。更为重要的是，当值在小节代理上被修改时，它们其实是在原始解析器中发生了改变。

configparser 对象的行为会尽可能地接近真正的字典。映射接口是完整而且遵循*MutableMapping* ABC 规范的。但是，还是有一些差异应当被纳入考虑:

- 默认情况下，小节中的所有键是以大小写不敏感的方式来访问的[Page 577, 1]。例如 `for option in parser["section"]` 只会产生 `optionxform` 形式的选项键名称。也就是说默认使用小写字母键名。与此同时，对于一个包含键 `'a'` 的小节，以下两个表达式均将返回 `True`:

```
"a" in parser["section"]
"A" in parser["section"]
```

- 所有小节也包括 `DEFAULTSECT`，这意味着对一个小节执行 `.clear()` 可能无法使得该小节显示为空。这是因为默认值是无法从小节中被删除的（因为从技术上说它们并不在那里）。如果它们在小节中被覆盖，删除将导致默认值重新变为可见。尝试删除默认值将会引发*KeyError*。
- `DEFAULTSECT` 无法从解析器中被移除:
  - 尝试删除将引发*ValueError*，
  - `parser.clear()` 会保留其原状，
  - `parser.popitem()` 绝不会将其返回。
- `parser.get(section, option, **kwargs)` - 第二个参数 **并非**回退值。但是请注意小节层级的 `get()` 方法可同时兼容映射协议和经典配置解析器 API。

- `parser.items()` 兼容映射协议（返回 *section_name*, *section_proxy* 对的列表，包括 DEFAULTSECT）。但是，此方法也可以附带参数被唤起: `parser.items(section, raw, vars)`。这种调用形式返回指定 `section` 的 *option*, *value* 对的列表，将展开所有插值（除非提供了 `raw=True` 选项）。

映射协议是在现有的传统 API 之上实现的，以便重写原始接口的子类仍然具有符合预期的有效映射。

14.2.8 定制解析器行为

INI 格式的变种数量几乎和使用此格式的应用一样多。*configparser* 花费了很大力气来为尽量大范围的可用 INI 样式提供支持。默认的可用功能主要由历史状况来确定，你很可能会想要定制某些特性。

改变特定配置解析器行为的最常见方式是使用 `__init__()` 选项：

- *defaults*，默认值: `None`

  此选项接受一个键值对的字典，它将被首先放入 `DEFAULT` 小节。这实现了一种优雅的方式来支持简洁的配置文件，它不必指定与已记录的默认值相同的值。

  提示：如果你想要为特定的节指定默认值，请在读取实际文件之前使用*read_dict()*。

- *dict_type*，默认值: *dict*

  此选项主要影响映射协议的行为和写入配置文件的外观。使用标准字典时，每个小节是按照它们被加入解析器的顺序保存的。在小节内的选项也是如此。

  还有其他替换的字典类型可以使用，例如在写回数据时对小节和选项进行排序。

  请注意：存在其他方式只用一次操作来添加键值对的集合。当你在这些操作中使用一个常规字典时，键将按顺序进行排列。例如:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*，默认值: `False`

  已知某些配置文件会包括不带值的设置，但其在其他方面均符合*configparser* 所支持的语法。构造器的 *allow_no_value* 形参可用于指明应当接受这样的值:

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)
```

(续下页)

(接上页)

```
>>> # 有值的设置像之前一样处理:
>>> config["mysqld"]["user"]
'mysql'

>>> # 没有值的设置将为 None:
>>> config["mysqld"]["skip-bdb"]

>>> # 未指定的设置仍将引发错误:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*，默认值: `('=', ':')`

  分隔符是用于在小节内分隔键和值的子字符串。在一行中首次出现的分隔子字符串会被视为一个分隔符。这意味着值可以包含分隔符（但键不可以）。

  另请参见*ConfigParser.write()* 的 *space_around_delimiters* 参数。

- *comment_prefixes*，默认值: `('#', ';')`

- *inline_comment_prefixes*，默认值: `None`

  注释前缀是配置文件中用于标示一条有效注释的开头的字符串。*comment_prefixes* 仅用在被视为空白的行（可以缩进）之前而 *inline_comment_prefixes* 可用在每个有效值之后（例如小节名称、选项以及空白的行）。默认情况下禁用行内注释，并且 `'#'` 和 `';'` 都被用作完整行注释的前缀。

  在 3.2 版本发生变更: 在之前的*configparser* 版本中行为匹配 `comment_prefixes=('#',';')` 和 `inline_comment_prefixes=(';',)`。

  请注意配置解析器不支持对注释前缀的转义，因此使用 *inline_comment_prefixes* 可能妨碍用户将被用作注释前缀的字符指定为可选值。当有疑问时，请避免设置 *inline_comment_prefixes*。在许多情况下，在多行值的一行开头存储注释前缀字符的唯一方式是进行前缀插值，例如:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

(续下页)

(接上页)

```
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*，默认值: `True`

  当设为 `True` 时，解析器在从单一源读取 (使用*read_file()*, *read_string()* 或*read_dict()*) 期间将不允许任何节或选项出现重复。推荐在新的应用中使用严格解析器。

  在 3.2 版本发生变更: 在之前的*configparser* 版本中行为匹配 `strict=False`。

- *empty_lines_in_values*，默认值: `True`

  在配置解析器中，值可以包含多行，只要它们的缩进级别低于它们所对应的键。默认情况下解析器还会将空行视为值的一部分。于此同时，键本身也可以任意缩进以提升可读性。因此，当配置文件变得非常庞大而复杂时，用户很容易失去对文件结构的掌控。例如:

  ```
  [Section]
  key = multiline
    value with a gotcha

   this = is still a part of the multiline value of 'key'
  ```

  在用户查看时这可能会特别有问题，如果她是使用比例字体来编辑文件的话。这就是为什么当你的应用不需要带有空行的值时，你应该考虑禁用它们。这将使得空行每次都会作为键之间的分隔。在上面的示例中，空行产生了两个键，`key` 和 `this`。

- *default_section*，默认值: `configparser.DEFAULTSECT` (即: `"DEFAULT"`)

  允许设置一个保存默认值的特殊节在其他节或插值等目的中使用的惯例是这个库所拥有的一个强大概念，使得用户能够创建复杂的声明性配置。这种特殊节通常称为 `"DEFAULT"` 但也可以被定制为指向任何其他有效的节名称。一些典型的值包括: `"general"` 或 `"common"`。所提供的名称在从任意节读取的时候被用于识别默认的节，而且也会在将配置写回文件时被使用。它的当前值可以使用 `parser_instance.default_section` 属性来获取，并且可以在运行时被修改（即将文件从一种格式转换为另一种格式）。

- *interpolation*，默认值: `configparser.BasicInterpolation`

  插值行为可以用通过提供 *interpolation* 参数提供自定义处理程序的方式来定制。`None` 可用来完全禁用插值，`ExtendedInterpolation()` 提供了一种更高级的变体形式，它的设计受到了 `zc.buildout` 的启发。有关该主题的更多信息请参见专门的文档章节。*RawConfigParser* 具有默认的值 `None`。

- *converters*，默认值: 不设置

  配置解析器提供了可选的值获取方法用来执行类型转换。默认情况下实现了*getint()*, *getfloat()* 和*getboolean()*。如果还需要其他获取方法，用户可以在子类中定义它们，或者传入一个字典，其中每个键都是一个转换器的名称而每个值都是一个实现了特定转换的可调用对象。例如，传入 `{'decimal': decimal.Decimal}` 将对解释器对象和所有节代理添加 `getdecimal()`。换句话说，可以同时编写 `parser_instance.getdecimal('section', 'key', fallback=0)` 和 `parser_instance['section'].getdecimal('key', 0)`。

  如果转换器需要访问解析器的状态，可以在配置解析器子类上作为一个方法来实现。如果该方法的名称是以 `get` 打头的，它将在所有节代理上以兼容字典的形式提供（参见上面的 `getdecimal()` 示例）。

更多高级定制选项可通过重写这些解析器属性的默认值来达成。默认值是在类中定义的，因此它们可以通过子类或属性赋值来重写。

ConfigParser.**BOOLEAN_STATES**

默认情况下当使用*getboolean()*时，配置解析器会将下列值视为 `True`: `'1'`, `'yes'`, `'true'`, `'on'` 而将下列值视为 `False`: `'0'`, `'no'`, `'false'`, `'off'`。你可以通过指定一个自定义的字符串键及其对应的布尔值字典来覆盖此行为。例如:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

其他典型的布尔值对包括 `accept`/`reject` 或 `enabled`/`disabled`。

ConfigParser.**optionxform**(*option*)

这个方法会转换每次 read, get, 或 set 操作的选项名称。默认会将名称转换为小写形式。这也意味着当一个配置文件被写入时，所有键都将为小写形式。如果此行为不合适则要重写此方法。例如:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

备注

optionxform 函数会将选项名称转换为规范形式。这应该是一个幂等函数：如果名称已经为规范形式，则应不加修改地将其返回。

ConfigParser.**SECTCRE**

一个已编译正则表达式会被用来解析节标头。默认将 `[section]` 匹配到名称 `"section"`。空格会被视为节名称的一部分，因此 `[ larch ]` 将被读取为一个名称为 `" larch "` 的节。如果此行为不合适则要覆盖此属性。例如:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
```

(续下页)

(接上页)

```
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

备注

虽然 ConfigParser 对象也使用 `OPTCRE` 属性来识别选项行，但并不推荐重写它，因为这会与构造器选项 *allow_no_value* 和 *delimiters* 产生冲突。

14.2.9 旧式 API 示例

主要出于向下兼容性的考虑，*configparser* 还提供了一种采用显式 `get`/`set` 方法的旧式 API。虽然以下介绍的方法存在有效的用例，但对于新项目仍建议采用映射协议访问。旧式 API 在多数时候都更复杂、更底层并且完全违反直觉。

一个写入配置文件的示例:

```
import configparser

config = configparser.RawConfigParser()

# 请注意在使用 RawConfigParser 的 set 函数时，
# 你可以在内部为键赋非字符串值，但当你试图
# 写入文件或在非原始模式下获取它时将会报错。
# 使用映射协议或 ConfigParser 的 set() 设置值时
# 不允许执行这样的赋值。
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# 将我们的配置文件写入 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

一个再次读取配置文件的示例:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() 在值不为浮点数时将引发异常
# getint() 和 getboolean() 对其相应类型也是如此
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)
```

(续下页)

(接上页)

```
# 请注意下面的输出不会执行 '%(bar)s' 或 '%(baz)s' 插值。
# 这是因为我们是使用 RawConfigParser()。
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

要获取插值，请使用*ConfigParser*:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# 如果你想要在一个单独的 get 操作中禁用插值
# 可将 get() 的可选参数 *raw* 设为 True。
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# 可选参数 *vars* 是一个字典，其中的元素将在
# 插值中优先使用。
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# 可选参数 *fallback* 可被用来提供一个回退值
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# 直接使用 print(cfg.get('Section1', 'monster')) 将会引发 NoOptionError
# 但我们还可以使用：

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

默认值在两种类型的 ConfigParser 中均可用。它们将在当某个选项未在别处定义时被用于插值。

```
import configparser

# 'bar' 和 'baz' 分别默认为 'Life' 和 'hard' 的新实例
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))     # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))     # -> "Life is hard!"
```

14.2.10 ConfigParser 对象

class configparser.**ConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

主配置解析器。当给定 *defaults* 时，它会被初始化为包含固有默认值的字典。当给定 *dict_type* 时，它将被用来创建包含节、节中的选项以及默认值的字典。

当给定 *delimiters* 时，它会被用作分隔键与值的子字符串的集合。当给定 *comment_prefixes* 时，它将被用作在否则为空行的注释的前缀子字符串的集合。注释可以被缩进。当给定 *inline_comment_prefixes* 时，它将被用作非空行的注释的前缀子字符串的集合。

当 *strict* 为 `True` (默认值) 时，解析器在从单个源（文件、字符串或字典）读取时将不允许任何节或选项出现重复，否则会引发*DuplicateSectionError* 或*DuplicateOptionError*。当 *empty_lines_in_values* 为 `False` (默认值: `True`) 时，每个空行均表示一个选项的结束。在其他情况下，一个多行选项内部的空行会被保留为值的一部分。当 *allow_no_value* 为 `True` (默认值: `False`) 时，将接受没有值的选项；此种选项的值将为 `None` 并且它们会以不带末尾分隔符的形式被序列化。

当给出 *default_section* 时，它指定了为其他部分和插值目的而保存默认值的特殊部分的名称 (通常命名为 `"DEFAULT"`)。该值可通过使用 `default_section` 实例属性在运行时被读取或修改值。这不会对已解析的配置文件进行重新求值，但会在将解析的设置写入新的配置文件时使用。

插值行为可通过给出 *interpolation* 参数提供自定义处理程序的方式来定制。`None` 可用来完全禁用插值，`ExtendedInterpolation()` 提供了一种更高级的变体形式，它的设计受到了 `zc.buildout` 的启发。有关该主题的更多信息请参见专门的文档章节。

插值中使用的所有选项名称将像任何其他选项名称引用一样通过*optionxform()* 方法来传递。例如，使用*optionxform()* 的默认实现（它会将选项名称转换为小写形式）时，值 `foo %(bar)s` 和 `foo %(BAR)s` 是等价的。

当给出 *converters* 时，它应当是一个字典，其中每个键代表一个类型转换器的名称而每个值则为实现从字符串到目标数据类型的转换的可调用对象。每个转换器会获得其在解析器对象和节代理上对应的 `get*()` 方法。

当 *allow_unnamed_section* 为 `True` 为 (默认值: `False`)，第一个节名称可被省略。参见"未命名节"一节。

将多个配置读入单个*ConfigParser* 是可能的，其中最近添加的配置具有最高优先级。任何冲突的键都会从更近的配置获取并且先前存在的键会被保留。下面的例子读入一个 `override.ini` 文件，它将覆盖任何来自 `example.ini` 文件的冲突的键。

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

在 3.1 版本发生变更: 默认的 *dict_type* 为*collections.OrderedDict*。

在 3.2 版本发生变更: 添加了 *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* 以及 *interpolation*。

在 3.5 版本发生变更: 添加了 *converters* 参数。

在 3.7 版本发生变更: *defaults* 参数将在*read_dict()* 时被读取，提供全解析器范围内一致的行为: 非字符串类型的键和值会被隐式地转换为字符串。

在 3.8 版本发生变更: 默认的 *dict_type* 为*dict*，因为它现在会保留插入顺序。

在 3.13 版本发生变更: 当 *allow_no_value* 为 `True` 且没有值的键带有一个缩进的行时将会引发*MultilineContinuationError*。

在 3.13 版本发生变更: 增加了 *allow_unnamed_section* 参数。

defaults()

返回包含实例范围内默认值的字典。

sections()

返回可用节的列表；*default section* 不包括在该列表中。

add_section(*section*)

向实例添加一个名为 *section* 的节。如果给定名称的节已存在，将会引发*DuplicateSectionError*。如果传入了 *default section* 名称，则会引发*ValueError*。节名称必须为字符串；如果不是则会引发*TypeError*。

在 3.2 版本发生变更: 非字符串的节名称将引发*TypeError*。

has_section(*section*)

指明相应名称的 *section* 是否存在于配置中。*default section* 不包含在内。

options(*section*)

返回指定 *section* 中可用选项的列表。

has_option(*section*, *option*)

如果给定的 *section* 存在并且包含给定的 *option* 则返回*True*；否则返回*False*。如果指定的 *section* 为*None* 或空字符串，则会使用 DEFAULT。

read(*filenames*, *encoding=None*)

尝试读取并解析一个包含文件名的可迭代对象，返回一个被成功解析的文件名列表。

如果 *filenames* 为字符串、*bytes* 对象或*path-like object*，它会被当作单个文件来处理。如果 *filenames* 中名称对应的某个文件无法被打开，该文件将被忽略。这样的设计使得你可以指定包含多个潜在配置文件位置的可迭代对象（例如当前目录、用户家目录以及某个系统级目录），存在于该可迭代对象中的所有配置文件都将被读取。

如果名称对应的文件全都不存在，则*ConfigParser* 实例将包含一个空数据集。一个要求从文件加载初始值的应用应当在调用*read()* 来获取任何可选文件之前使用*read_file()* 来加载所要求的一个或多个文件:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

在 3.2 版本发生变更: 增加了 *encoding* 形参。在之前版本中，所有文件都将使用*open()* 的默认编码格式来读取。

在 3.6.1 版本发生变更: *filenames* 形参接受一个*path-like object*。

在 3.7 版本发生变更: *filenames* 形参接受一个*bytes* 对象。

read_file(*f*, *source=None*)

从 *f* 读取并解析配置数据，它必须是一个产生 Unicode 字符串的可迭代对象（例如以文本模式打开的文件）。

可选参数 *source* 指定要读取的文件名称。如果未给出并且 *f* 具有 `name` 属性，则该属性会被用作 *source*；默认值为 `'<???>'`。

Added in version 3.2: 替代 `readfp()`。

read_string(*string*, *source='<string>'*)

从字符串中解析配置数据。

可选参数 *source* 指定一个所传入字符串的上下文专属名称。如果未给出，则会使用 `'<string>'`。这通常应为一个文件系统路径或 URL。

Added in version 3.2.

read_dict(*dictionary*, *source='<dict>'*)

从任意一个提供了类似于字典的 `items()` 方法的对象加载配置。键为节名称，值为包含节中所出现的键和值的字典。如果所用的字典类型会保留顺序，则节和其中的键将按顺序加入。值会被自动转换为字符串。

可选参数 *source* 指定一个所传入字典的上下文专属名称。如果未给出，则会使用 `<dict>`。

此方法可被用于在解析器之间拷贝状态。

Added in version 3.2.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

获取指定名称的 *section* 的一个 *option* 的值。如果提供了 *vars*，则它必须为一个字典。*option* 的查找顺序为 *vars** (如果有提供)、**section* 以及 *DEFAULTSECT*。如果未找到该键并且提供了 *fallback*，则它会被用作回退值。可以提供 `None` 作为 *fallback* 值。

所有 `'%'` 插值会在返回值中被展开，除非 *raw* 参数为真值。插值键所使用的值会按与选项相同的方式来查找。

在 3.2 版本发生变更: *raw*, *vars* 和 *fallback* 都是仅限关键字参数，以防止用户试图使用第三个参数作业为 *fallback* 回退值（特别是在使用映射协议的时候）。

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

将在指定 *section* 中的 *option* 强制转换为整数的便捷方法。参见`get()` 获取对于 *raw*, *vars* 和 *fallback* 的解释。

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

将在指定 *section* 中的 *option* 强制转换为浮点数的便捷方法。参见`get()` 获取对于 *raw*, *vars* 和 *fallback* 的解释。

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

将在指定 *section* 中的 *option* 强制转换为布尔值的便捷方法。请注意选项所接受的值为 `'1'`, `'yes'`, `'true'` 和 `'on'`，它们会使得此方法返回 `True`，以及 `'0'`, `'no'`, `'false'` 和 `'off'`，它们会使得此方法返回 `False`。这些字符串值会以对大小写不敏感的方式被检测。任何其他值都将导致引发`ValueError`。参见`get()` 获取对于 *raw*, *vars* 和 *fallback* 的解释。

items(*raw=False*, *vars=None*)

items(*section*, *raw=False*, *vars=None*)

当未给出 *section* 时，将返回由 *section_name*, *section_proxy* 对组成的列表，包括 DEFAULTSECT。

在其他情况下，将返回给定的 *section* 中的 option 的 *name*, *value* 对组成的列表。可选参数具有与`get()` 方法的参数相同的含义。

在 3.8 版本发生变更: *vars* 中的条目将不在结果中出现。之前的行为混淆了实际的解析器选项和为插值提供的变量。

set(*section*, *option*, *value*)

如果给定的节存在，则将所给出的选项设为指定的值；在其他情况下将引发`NoSectionError`。*option* 和 *value* 必须为字符串；如果不是则将引发`TypeError`。

write(*fileobject*, *space_around_delimiters=True*)

将配置的表示形式写入指定的*file object*，该对象必须以文本模式打开（接受字符串）。此表示形式可由将来的`read()` 调用进行解析。如果 *space_around_delimiters* 为真值，键和值之前的分隔符两边将加上空格。

备注

原始配置文件中的注释在写回配置时不会被保留。具体哪些会被当作注释，取决于为 *comment_prefix* 和 *inline_comment_prefix* 所指定的值。

remove_option(*section*, *option*)

将指定的 *option* 从指定的 *section* 中移除。如果指定的节不存在则会引发*NoSectionError*。如果要移除的选项存在则返回*True*；在其他情况下将返回*False*。

remove_section(*section*)

从配置中移除指定的 *section*。如果指定的节确实存在则返回 `True`。在其他情况下将返回 `False`。

optionxform(*option*)

将选项名 *option* 转换为输入文件中的形式或客户端代码所传入的应当在内部结构中使用的形式。默认实现将返回 *option* 的小写形式版本；子类可以重写此行为，或者客户端代码也可以在实例上设置一个具有此名称的属性来影响此行为。

你不需要子类化解析器来使用此方法，你也可以在一个实例上设置它，或使用一个接受字符串参数并返回字符串的函数。例如将它设为 `str` 将使得选项名称变得大小写敏感:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

请注意当读取配置文件时，选项名称两边的空格将在调用*optionxform()*之前被去除。

configparser.**UNNAMED_SECTION**

一个代表用于引用未命名小节的小节名称的特殊对象 (参见未命名小节)。

configparser.**MAX_INTERPOLATION_DEPTH**

当 *raw* 形参为假值时*get()*所采用的递归插值的最大深度。这只在使用默认的 *interpolation* 时会起作用。

14.2.11 RawConfigParser 对象

class configparser.**RawConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

旧式*ConfigParser*。它默认禁用插值并且允许通过不安全的 `add_section` 和 `set` 方法以及旧式 `defaults=` 关键字参数处理来设置非字符串的节名、选项名和值。

在 3.2 版本发生变更: 添加了 *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* 以及 *interpolation*。

在 3.5 版本发生变更: 添加了 *converters* 参数。

在 3.8 版本发生变更: 默认的 *dict_type* 为*dict*，因为它现在会保留插入顺序。

在 3.13 版本发生变更: 增加了 *allow_unnamed_section* 参数。

> **备注**
>
> 考虑改用*ConfigParser*，它会检查内部保存的值的类型。如果你不想要插值，你可以使用 `ConfigParser(interpolation=None)`。

add_section(*section*)

向实例添加一个名为 *section* 的节。如果给定名称的节已存在，将会引发*DuplicateSectionError*。如果传入了 *default section* 名称，则会引发*ValueError*。

不检查 *section* 以允许用户创建以非字符串命名的节。此行为已不受支持并可能导致内部错误。

set(*section*, *option*, *value*)

如果给定的节存在，则将给定的选项设为指定的值；在其他情况下将引发*NoSectionError*。虽然可能使用*RawConfigParser*（或使用*ConfigParser* 并将 *raw* 形参设为真值）以便实现非字符串值的 *internal* 存储，但是完整功能（包括插值和输出到文件）只能使用字符串值来实现。

此方法允许用户在内部将非字符串值赋给键。此行为已不受支持并会在尝试写入到文件或在非原始模式下获取数据时导致错误。**请使用映射协议 API**，它不允许出现这样的赋值。

14.2.12 异常

exception configparser.**Error**

所有其他*configparser* 异常的基类。

exception configparser.**NoSectionError**

当找不到指定节时引发的异常。

exception configparser.**DuplicateSectionError**

当调用*add_section()* 时传入已存在的节名称，或者在严格解析器中当单个输入文件、字符串或字典内出现重复的节时引发的异常。

在 3.2 版本发生变更: 向 `__init__()` 添加了可选的 *source* 和 *lineno* 属性和形参。

exception configparser.**DuplicateOptionError**

当单个选项在从单个文件、字符串或字典读取时出现两次时引发的异常。这会捕获拼写错误和大小写敏感相关的错误，例如一个字典可能包含两个键分别代表同一个大小写不敏感的配置键。

exception configparser.**NoOptionError**

当指定的选项未在指定的节中被找到时引发的异常。

exception configparser.**InterpolationError**

当执行字符串插值发生问题时所引发的异常的基类。

exception configparser.**InterpolationDepthError**

当字符串插值由于迭代次数超出*MAX_INTERPOLATION_DEPTH* 而无法完成所引发的异常。为*InterpolationError* 的子类。

exception configparser.**InterpolationMissingOptionError**

当从某个值引用的选项并不存在时引发的异常。为*InterpolationError* 的子类。

exception configparser.**InterpolationSyntaxError**

当将要执行替换的源文本不符合要求的语法时引发的异常。为*InterpolationError* 的子类。

exception configparser.**MissingSectionHeaderError**

当尝试解析一个不带节标头的文件时引发的异常。

exception configparser.**ParsingError**

当尝试解析一个文件而发生错误时引发的异常。

在 3.12 版本发生变更: `filename` 属性和 `__init__()` 构造器参数已被移除。它们自 3.2 起可以使用名称 `source` 来访问。

exception configparser.**MultilineContinuationError**

当没有对应值的键带有一个缩进的行时将引发的异常。

Added in version 3.13.

备注

14.3 tomllib --- 解析 TOML 文件

Added in version 3.11.

源代码： Lib/tomllib

此模块提供了一个解析 TOML 1.0.0 (Tom's Obvious Minimal Language, https://toml.io) 的接口。该模块不支持写入 TOML。

> **参见**
>
> Tomli-W 包 是一个 TOML 写入器，它可以与此模块一起使用，提供了与标准库用户熟悉的*marshal* 和*pickle* 模块类似的写入 API。

> **参见**
>
> TOML Kit 包 一个是兼具读取和写入功能的保留样式的 TOML 库。它是用于编辑现有 TOML 文件的本模块的推荐替代品。

这个模块定义了以下函数：

tomllib.**load**(*fp*, */*, ***, *parse_float=float*)

读取一个 TOML 文件。第一个参数应该是一个可读的二进制文件对象。返回*dict*。使用*转换表* 将 TOML 类型转换为 Python。

对每个要解析的 TOML 浮点数字符串调用 *parse_float*。默认情况下，这相当于 `float(num_str)`。这可以用于为 TOML 浮点数使用另一种数据类型或解析器（例如：*decimal.Decimal*）。可调用对象不能返回*dict* 或*list*，否则将引发*ValueError*。

对无效的 TOML 文档将引发*TOMLDecodeError*。

tomllib.**loads**(*s*, */*, ***, *parse_float=float*)

从*str* 对象中加载 TOML。返回*dict*。使用*转换表* 将 TOML 类型转换为 Python 类型。参数 *parse_float* 与*load()* 中的意义相同。

对无效的 TOML 文档将引发*TOMLDecodeError*。

有以下几种异常：

exception tomllib.**TOMLDecodeError**

ValueError 的子类

14.3.1 例子

解析 TOML 文件:

```
import tomllib

with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)
```

解析 TOML 字符串:

```
import tomllib

toml_str = """
python-version = "3.11.0"
python-implementation = "CPython"
"""

data = tomllib.loads(toml_str)
```

14.3.2 转换表

TOML	Python
TOML 文档	dict
string	str
integer	int
float	float（可用 *parse_float* 配置）
boolean	bool
offset date-time	datetime.datetime（`tzinfo` 属性设置为 `datetime.timezone` 的实例）
local date-time	datetime.datetime （`tzinfo` 属性设置为 `None`）
local date	datetime.date
local time	datetime.time
array	list
table	dict
内联表	dict
表数组	字典列表

14.4 `netrc` --- netrc 文件处理

源代码: Lib/netrc.py

---

netrc 类解析并封装了 Unix 的 **ftp** 程序和其他 FTP 客户端所使用的 netrc 文件格式。

class netrc.**netrc**([*file*])

netrc 的实例或其子类的实例会被用来封装来自 netrc 文件的数据。如果有初始化参数，它将指明要解析的文件。如果未给出参数，则位于用户家目录的 `.netrc` 文件 -- 即*os.path.expanduser()* 所确定的文件 -- 将会被读取。在其他情况下，则将引发*FileNotFoundError* 异常。解析错误将引发*NetrcParseError* 并附带诊断信息，包括文件名、行号以及终止令牌。如果在 POSIX 系统上未指明参数，则当 `.netrc` 文件中有密码时，如果文件归属或权限不安全（归属的用户不是运行进程的用户，或者可供任何其他用户读取或写入）将引发*NetrcParseError*。这实现了与 ftp 和其他使用 `.netrc` 的程序同等的安全行为。

在 3.4 版本发生变更: 添加了 POSIX 权限检查。

在 3.7 版本发生变更: 当未将 *file* 作为参数传入时会使用*os.path.expanduser()* 来查找 `.netrc` 文件的位置。

在 3.10 版本发生变更: *netrc* 会在使用语言区域专属的编码格式之前先尝试 UTF-8 编码格式。netrc 文件中的条目不再需要包括所有凭据。缺失的凭据值默认将为空字符串。所有的凭据及其值现在可以包含任意字符，如空格和非 ASCII 字符。如果登录名为 anonymous，它将不会触发安全检查。

exception netrc.**NetrcParseError**

当源代码文本中出现语法错误时由*netrc* 类所引发的异常。该异常的实例提供了三个有用的属性:

msg

错误的解释性文本。

filename

源代码文件名。

lineno

发现错误所在的行号。

14.4.1 netrc 对象

netrc 实例具有下列方法:

netrc.**authenticators**(*host*)

针对 *host* 的身份验证者返回一个 3 元组 (login, account, password)。如果 netrc 文件不包含针对给定主机的条目，则返回关联到'default' 条目的元组。如果匹配的主机或默认条目均不可用，则返回 None。

netrc.**__repr__**()

将类数据以 netrc 文件的格式转储为一个字符串。(这会丢弃注释并可能重排条目顺序。)

netrc 的实例具有一些公共实例变量:

netrc.**hosts**

将主机名映射到 (login, account, password) 元组的字典。如果存在'default' 条目，则会表示为使用该名称的伪主机。

netrc.**macros**

将宏名称映射到字符串列表的字典。

14.5 plistlib --- 生成与解析 Apple .plist 文件

源代码: Lib/plistlib.py

此模块提供了可读写 Apple "property list" 文件的接口，它主要用于 macOS 和 iOS 系统。此模块同时支持二进制和 XML plist 文件。

property list (.plist) 文件格式是一种简单的序列化格式，它支持一些基本对象类型，例如字典、列表、数字和字符串等。通常使用一个字典作为最高层级对象。

要写入和解析 plist 文件，请使用*dump()* 和*load()* 函数。

要以字节串或字符串对象形式操作 plist 数据，请使用*dumps()* 和*loads()*。

值可以为字符串、整数、浮点数、布尔值、元组、列表、字典（但只允许用字符串作为键）、*bytes*、*bytearray* 或*datetime.datetime* 对象。

在 3.4 版本发生变更: 新版 API，旧版 API 已被弃用。添加了对二进制 plist 格式的支持。

在 3.8 版本发生变更: 添加了在二进制 plist 中读写*UID* 令牌的支持，例如用于 NSKeyedArchiver 和 NSKeyedUnarchiver。

在 3.9 版本发生变更: 旧 API 已被移除。

参见

PList 指南页面
针对该文件格式的 Apple 文档。

这个模块定义了以下函数:

plistlib.**load**(*fp*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

读取 plist 文件。*fp* 应当可读并且为二进制文件对象。返回已解包的根对象（通常是一个字典）。

fmt 为文件的格式，有效的值如下:

- *None*: 自动检测文件格式
- *FMT_XML*: XML 文件格式
- *FMT_BINARY*: 二进制 plist 格式

dict_type 为字典用来从 plist 文件读取的类型。

当 *aware_datetime* 为真值时，类型为 `datetime.datetime` 的字段将被创建为感知型对象，其 `tzinfo` 将设为 *datetime.UTC*。

FMT_XML 格式的 XML 数据会使用来自 *xml.parsers.expat* 的 Expat 解析器 -- 请参阅其文档了解错误格式 XML 可能引发的异常。未知元素将被 plist 解析器直接略过。

当文件无法被解析时解析器将引发 *InvalidFileException*。

Added in version 3.4.

在 3.13 版本发生变更: 增加了仅限关键字形参 *aware_datetime*。

`plistlib.`**`loads`**(*data*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

从一个字节串或字符串对象加载 plist。请参阅 *load()* 获取相应关键字参数的说明。

Added in version 3.4.

在 3.13 版本发生变更: 当 *fmt* 等于 *FMT_XML* 时 *data* 可以为字符串。

`plistlib.`**`dump`**(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

将 *value* 写入 plist 文件。*fp* 应当是一个可写的二进制文件对象。

fmt 参数指定 plist 文件的格式，可以是以下值之一:

- *FMT_XML*: XML 格式的 plist 文件
- *FMT_BINARY*: 二进制格式的 plist 文件

当 *sort_keys* 为真值（默认）时字典的键将经过排序再写入 plist，否则将按字典的迭代顺序写入。

当 *skipkeys* 为假值（默认）时该函数将在字典的键不为字符串时引发 *TypeError*，否则将跳过这样的键。

当 *aware_datetime* 为真值并且有任何类型为 `datetime.datetime` 的字段被设为感知型对象，它将在写入之前转换为 UTC 时区。

如果对象是不受支持的类型或者是包含不受支持类型的对象的容器则将引发 *TypeError*。

对于无法在（二进制）plist 文件中表示的整数值，将会引发 *OverflowError*。

Added in version 3.4.

在 3.13 版本发生变更: 增加了仅限关键字形参 *aware_datetime*。

`plistlib.`**`dumps`**(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

将 *value* 以 plist 格式字节串对象的形式返回。参阅 *dump()* 的文档获取此函数的关键字参数的说明。

Added in version 3.4.

可以使用以下的类:

`class` `plistlib.`**`UID`**(*data*)

包装一个 *int*。该类将在读取或写入 NSKeyedArchiver 编码的数据时被使用，其中包含 UID（参见 PList 指南）。

`data`

UID 的整数值。它必须在 `0 <= data < 2**64` 范围内。

Added in version 3.8.

可以使用以下的常量:

`plistlib.`**`FMT_XML`**

用于 plist 文件的 XML 格式。

Added in version 3.4.

plistlib.**FMT_BINARY**

用于 plist 文件的二进制格式。

Added in version 3.4.

此模块定义了下列异常：

exception plistlib.**InvalidFileException**

当文件无法被解析时将被引发。

Added in version 3.4.

14.5.1 例子

生成一个 plist:

```
import datetime
import plistlib

pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.now()
)
print(plistlib.dumps(pl).decode())
```

解析一个 plist:

```
import plistlib

plist = b"""<plist version="1.0">
<dict>
    <key>foo</key>
    <string>bar</string>
</dict>
</plist>"""
pl = plistlib.loads(plist)
print(pl["foo"])
```

CHAPTER 15

加密服务

本章中介绍的模块实现了多种加密性质的算法。它们可在安装时选择使用。以下是内容概要：

15.1 hashlib --- 安全哈希与消息摘要

源码： Lib/hashlib.py

本模块实现了一个针对不同哈希算法的通用接口。包括了 FIPS 安全哈希算法 SHA224, SHA256, SHA384, SHA512, (定义见 the FIPS 180-4 standard), SHA-3 系列 (定义见 the FIPS 202 standard) 以及旧式算法 SHA1 (formerly part of FIPS) 和 MD5 算法 (定义见 internet **RFC 1321**)。

> **备注**
>
> 如果你想找到 adler32 或 crc32 哈希函数，它们在`zlib`模块中。

15.1.1 哈希算法

每种类型的 *hash* 都有一个构造器方法。它们都返回一个具有相同简单接口的哈希对象。例如，使用`sha256()`创建一个 SHA-256 哈希对象。你可以使用`update`方法向这个对象输入字节类对象 (通常是`bytes`)。在任何时候你都可以使用`digest()`或`hexdigest()`方法获得到目前为止输入这个对象的拼接数据的 *digest*。

为了允许多线程，当在其构造器或`.update`方法中计算一次性提供超过 2047 字节数据的哈希时将会释放 Python *GIL*。

本模块中总是存在的哈希算法构造器有`sha1()`，`sha224()`，`sha256()`，`sha384()`，`sha512()`，`sha3_224()`，`sha3_256()`，`sha3_384()`，`sha3_512()`，`shake_128()`，`shake_256()`，`blake2b()`和`blake2s()`。`md5()`通常也是可用的，但在你使用稀有的"FIPS 兼容" Python 编译版时它可能会缺失或被屏蔽。这些构造器对应于`algorithms_guaranteed`。

如果你的 Python 分发版的`hashlib`是基于提供了其他算法的 OpenSSL 编译版上链接的那么还可能存在一些附加的算法。其他算法在所有安装版上不保证全都可用并且仅可通过`new()`使用名称来访问。参见`algorithms_available`。

> **警告**
>
> 一些算法具有已知的碰撞弱点（包括 MD5 和 SHA1）。请参阅本文档末尾的 Attacks on cryptographic hash algorithms 和*hashlib-seealso* 小节。

Added in version 3.6: 增加了 SHA3 (Keccak) 和 SHAKE 构造器*sha3_224()*, *sha3_256()*, *sha3_384()*, *sha3_512()*, *shake_128()*, *shake_256()*。并增加了*blake2b()* 和*blake2s()*。在 3.9 版本发生变更: 所有 hashlib 的构造器都接受仅限关键字参数 *usedforsecurity* 且其默认值为 `True`。设为假值即允许在受限的环境中使用不安全且阻塞的哈希算法。`False` 表示此哈希算法不可用于安全场景，例如用作非加密的单向压缩函数。

在 3.9 版本发生变更: 现在 hashlib 会在 OpenSSL 有提供的情况下使用 SHA3 和 SHAKE。

在 3.12 版本发生变更: 在所链接的 OpenSSL 未提供 MD5, SHA1, SHA2 或 SHA3 算法的情况下我们将回退至来自 HACL* project 的已验证的实现。

15.1.2 用法

要获取字节串 `b"Nobody inspects the spammish repetition"` 的摘要:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\x0fK\
↪x94\x06'
>>> m.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

更简要的写法:

```
>>> hashlib.sha256(b"Nobody inspects the spammish repetition").hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

15.1.3 构造器

`hashlib.`**`new`**(*name*, [*data*,]*, *usedforsecurity=True*)

接受想要的算法对应的字符串 *name* 作为其第一个形参的泛型构造器。它还允许访问上面列出的哈希算法以及你的 OpenSSL 库可能提供的任何其他算法。

使用*new()* 并附带一个算法名称:

```
>>> h = hashlib.new('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

`hashlib.`**`md5`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha1`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha224`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha256`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha384`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha512`**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_224**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_512**([*data*,]*, *usedforsecurity=True*)

这些带命名的构造器速度相比向*new()* 传入算法名称更快。

15.1.4 属性

在 hashlib 中提供了下列常量模块属性:

hashlib.**algorithms_guaranteed**

一个集合，其中包含此模块在所有平台上都保证支持的哈希算法的名称。请注意'md5' 也在此清单中，虽然某些上游厂商提供了一个怪异的排除了此算法的"FIPS 兼容" Python 编译版本。

Added in version 3.2.

hashlib.**algorithms_available**

一个集合，其中包含在所运行的 Python 解释器上可用的哈希算法的名称。将这些名称传给*new()* 时将可被识别。*algorithms_guaranteed* 将总是它的一个子集。同样的算法在此集合中可能以不同的名称出现多次（这是 OpenSSL 的原因）。

Added in version 3.2.

15.1.5 哈希对象

下列值会以构造器所返回的哈希对象的常量属性的形式被提供:

hash.**digest_size**

以字节表示的结果哈希对象的大小。

hash.**block_size**

以字节表示的哈希算法的内部块大小。

hash 对象具有以下属性:

hash.**name**

此哈希对象的规范名称，总是为小写形式并且总是可以作为*new()* 的形参用来创建另一个此类型的哈希对象。

在 3.4 版本发生变更: 该属性名称自被引入起即存在于 CPython 中，但在 Python 3.4 之前并未正式指明，因此可能不存在于某些平台上。

哈希对象具有下列方法:

hash.**update**(*data*)

用*bytes-like object* 来更新哈希对象。重复调用相当于单次调用并传入所有参数的拼接结果: `m.update(a); m.update(b)` 等价于 `m.update(a+b)`。

hash.**digest**()

返回当前已传给*update()* 方法的数据摘要。这是一个大小为*digest_size* 的字节串对象，字节串中可包含 0 至 255 的完整取值范围。

hash.**hexdigest**()

类似于*digest()* 但摘要会以两倍长度字符串对象的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

hash.**copy**()

返回哈希对象的副本（"克隆"）。这可被用来高效地计算共享相同初始子串的数据的摘要。

15.1.6 SHAKE 可变长度摘要

hashlib.**shake_128**([*data*,]*, *usedforsecurity=True*)

hashlib.**shake_256**([*data*,]*, *usedforsecurity=True*)

shake_128() 和*shake_256()* 算法提供安全的 length_in_bits//2 至 128 或 256 位可变长度摘要。为此，它们的摘要需指定一个长度。SHAKE 算法不限制最大长度。

shake.**digest**(*length*)

返回当前已传给*update()* 方法的数据摘要。这是一个大小为 *length* 的字节串对象，其中可包含 0 至 255 完整范围内的字节值。

shake.**hexdigest**(*length*)

类似于*digest()* 但摘要会以两倍长度字符串对象的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

用法示例:

```
>>> h = hashlib.shake_256(b'Nobody inspects the spammish repetition')
>>> h.hexdigest(20)
'44709d6fcb83d92a76dcb0b668c98e1b1d3dafe7'
```

15.1.7 文件哈希

hashlib 模块提供了一个辅助函数用于文件或文件型对象的高效哈希操作。

hashlib.**file_digest**(*fileobj*, *digest*, */*)

返回一个根据文件对象进行更新的摘要对象。

fileobj 必须是一个以二进制模式打开用于读取的文件型对象。它接受来自内置*open()*, *BytesIO* 实例, *socket.socket.makefile()* 创建的 SocketIO 及其他类似的文件对象。*fileobj* 必须以阻塞模式打开，否则可能引发*BlockingIOError*。

此函数可能会绕过 Python 的 I/O 并直接使用来自*fileno()* 的文件描述符。*fileobj* 在此函数返回或引发异常之后必须被假定为已处于未知状态。应当由调用方来负责关闭 *fileobj*。

digest 必须是一个 *str* 形式的哈希算法名称、哈希构造器或返回哈希对象的可调用对象。

示例:

```
>>> import io, hashlib, hmac
>>> with open(hashlib.__file__, "rb") as f:
...     digest = hashlib.file_digest(f, "sha256")
...
>>> digest.hexdigest()
'...'
```

```
>>> buf = io.BytesIO(b"somedata")
>>> mac1 = hmac.HMAC(b"key", digestmod=hashlib.sha512)
>>> digest = hashlib.file_digest(buf, lambda: mac1)
```

```
>>> digest is mac1
True
>>> mac2 = hmac.HMAC(b"key", b"somedata", digestmod=hashlib.sha512)
>>> mac1.digest() == mac2.digest()
True
```

Added in version 3.11.

在 3.13.3 (unreleased) 版本发生变更: 现在如果文件是以阻塞模式打开则会引发*BlockingIOError*。在之前版本中，会向摘要添加伪装的空字节。

15.1.8 密钥派生

密钥派生和密钥延展算法被设计用于安全密码哈希。`sha1(password)` 这样的简单算法无法防御暴力攻击。好的密码哈希函数必须可以微调、放慢步调，并且包含 加盐。

hashlib.**pbkdf2_hmac**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

此函数提供 PKCS#5 基于密码的密钥派生函数 2。它使用 HMAC 作为伪随机函数。

字符串 *hash_name* 是要求用于 HMAC 的哈希摘要算法的名称，例如'sha1' 或'sha256'。*password* 和 *salt* 会以字节串缓冲区的形式被解析。应用和库应当将 *password* 限制在合理长度 (例如 1024)。*salt* 应当为适当来源例如`os.urandom()` 的大约 16 个或更多的字节串数据。

iterations 的数值应当基于哈希算法和机器算力来选择。在 2022 年，建议选择进行数万次的 SHA-256 迭代。对于为何以及如何选择最适合你的应用程序的迭代次数的理由，请参阅 NIST-SP-800-132 的 *Appendix A.2.2*。其中 stackexchange pbkdf2 迭代问题 的解答提供的详细的说明。

dklen 是以字节数表示的派生密钥长度。如果 *dklen* 为 `None` 则会使用哈希算法 *hash_name* 的摘要长度，例如对 SHA-512 来说是 64。

```
>>> from hashlib import pbkdf2_hmac
>>> our_app_iters = 500_000  # Application specific, read above.
>>> dk = pbkdf2_hmac('sha256', b'password', b'bad salt' * 2, our_app_iters)
>>> dk.hex()
'15530bba69924174860db778f2c6f8104d3aaf9d26241840c8c4a641c8d000a9'
```

此函数只有在 Python 附带 OpenSSL 编译时才可用。

Added in version 3.4.

在 3.12 版本发生变更: 现在此函数只有在 Python 附带 OpenSSL 构建时才可用。慢速的纯 Python 实现已被移除。

hashlib.**scrypt**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

此函数提供基于密码加密的密钥派生函数，其定义参见 **RFC 7914**。

password 和 *salt* 必须为字节类对象。应用和库应当将 *password* 限制在合理长度 (例如 1024)。*salt* 应当为适当来源例如`os.urandom()` 的大约 16 个或更多的字节串数据。

n 是 CPU/内存开销因子，*r* 是块大小，*p* 是并行化因子而 *maxmem* 是内存上限（OpenSSL 1.1.0 默认为 32 MiB）。*dklen* 是以字节数表示的派生密钥长度。

Added in version 3.6.

15.1.9 BLAKE2

BLAKE2 是在 **RFC 7693** 中定义的加密哈希函数，它有两种形式:

- **BLAKE2b**，针对 64 位平台进行优化，并会生成长度介于 1 和 64 字节之间任意大小的摘要。
- **BLAKE2s**，针对 8 至 32 位平台进行优化，并会生成长度介于 1 和 32 字节之间任意大小的摘要。

BLAKE2 支持 **keyed mode** (HMAC 的更快速更简单的替代), **salted hashing**, **personalization** 和 **tree hashing**.

此模块的哈希对象遵循标准库`hashlib` 对象的 API。

创建哈希对象

新哈希对象可通过调用构造器函数来创建:

hashlib.**blake2b**(*data=b''*, ***, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

hashlib.**blake2s**(*data=b''*, ***, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

这些函数返回用于计算 BLAKE2b 或 BLAKE2s 的相应的哈希对象。它们接受下列可选通用形参:

- *data*: 要哈希的初始数据块，它必须为*bytes-like object*。它只能作为位置参数传入。
- *digest_size*: 以字节数表示的输出摘要大小。
- *key*: 用于密钥哈希的密钥（对于 BLAKE2b 最长 64 字节，对于 BLAKE2s 最长 32 字节）。
- *salt*: 用于随机哈希的盐值（对于 BLAKE2b 最长 16 字节，对于 BLAKE2s 最长 8 字节）。
- *person*: 个性化字符串（对于 BLAKE2b 最长 16 字节，对于 BLAKE2s 最长 8 字节）。

下表显示了常规参数的限制（以字节为单位）：

Hash	目标长度	长度（键）	长度（盐）	长度（个人）
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

备注

BLAKE2 规格描述为盐值和个性化形参定义了固定的长度，但是为了方便起见，此实现接受指定在长度以内的任意大小的字节串。如果形参长度小于指定值，它将以零值进行填充，因此举例来说，`b'salt'` 和 `b'salt\x00'` 为相同的值 (*key* 的情况则并非如此。)

如下面的模块*constants* 所描述，这些是可用的大小取值。

构造器函数还接受下列树形哈希形参:

- *fanout*: 扇出值 (0 至 255，如无限制即为 0，连续模式下为 1)。
- *depth*: 树的最大深度 (1 至 255，如无限制则为 255，连续模式下为 1)。
- *leaf_size*: 叶子的最大字节长度 (0 至 `2**32-1`，如无限制或在连续模式下则为 0)。
- *node_offset*: 节点的偏移量 (对于 BLAKE2b 为 0 至 `2**64-1`，对于 BLAKE2s 为 0 至 `2**48-1`，对于最多边的第一个叶子或在连续模式下则为 0)。
- *node_depth*: 节点深度 (0 至 255，对于叶子或在连续模式下则为 0)。
- *inner_size*: 内部摘要大小 (对于 BLAKE2b 为 0 至 64，对于 BLAKE2s 为 0 至 32，连续模式下则为 0)。
- *last_node*: 一个指明所处理的节点是否为最后一个 (在连续模式下为 `False`) 的布尔值。

请参阅 BLAKE2 规格描述 第 2.10 节获取有关树形哈希的完整介绍。

常量

`blake2b.`**`SALT_SIZE`**

`blake2s.`**`SALT_SIZE`**

盐值长度（构造器所接受的最大长度）。

`blake2b.`**`PERSON_SIZE`**

`blake2s.`**`PERSON_SIZE`**

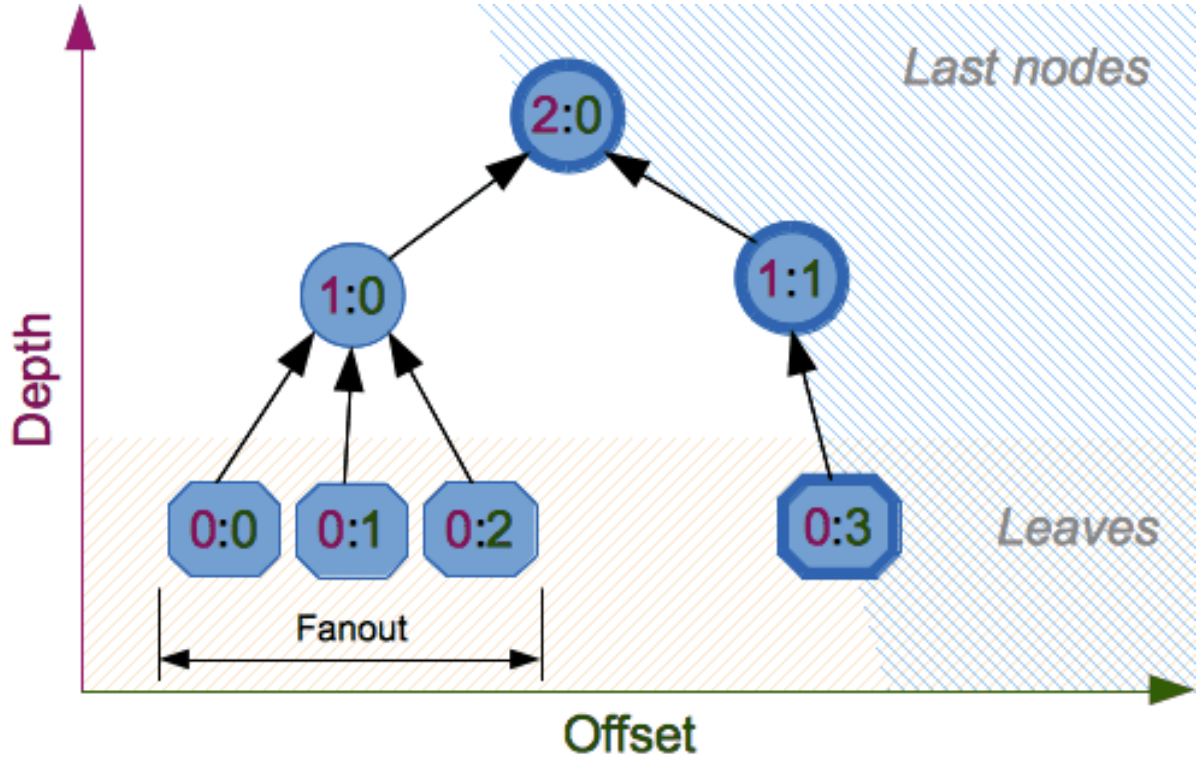
个性化字符串长度（构造器所接受的最大长度）。

`blake2b.`**`MAX_KEY_SIZE`**

`blake2s.`**`MAX_KEY_SIZE`**

最大密钥长度。

`blake2b.`**`MAX_DIGEST_SIZE`**



blake2s.**MAX_DIGEST_SIZE**

哈希函数可输出的最大摘要长度。

例子

简单哈希

要计算某个数据的哈希值，你应该首先通过调用适当的构造器函数 (*blake2b()* 或 *blake2s()*) 来构造一个哈希对象，然后通过在该对象上调用 *update()* 来更新目标数据，最后再通过调用 *digest()* (或针对十六进制编码字符串的 *hexdigest()*) 来获取该对象的摘要。

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3
↪'
```

作为快捷方式，你可以直接以位置参数的形式向构造器传入第一个数据块来直接更新:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3
↪'
```

你可以多次调用 *hash.update()* 至你所想要的任意次数以迭代地更新哈希值:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
```

(续下页)

(接上页)

```
...     h.update(item)
...
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3
↪'
```

使用不同的摘要大小

BLAKE2 具有可配置的摘要大小，对于 BLAKE2b 最多 64 字节，对于 BLAKE2s 最多 32 字节。例如，要使用 BLAKE2b 来替代 SHA-1 而不改变输出大小，我们可以让 BLAKE2b 产生 20 个字节的摘要:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

不同摘要大小的哈希对象具有完全不同的输出（较短哈希值并非较长哈希值的前缀）；即使输出长度相同，BLAKE2b 和 BLAKE2s 也会产生不同的输出:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

密钥哈希

带密钥的哈希运算可被用于身份验证，作为 基于哈希的消息验证代码 (HMAC) 的一种更快速更简单的替代。BLAKE2 可被安全地用于前缀 MAC 模式，这是由于它从 BLAKE 继承而来的不可区分特性。

这个例子演示了如何使用密钥 `b'pseudorandom key'` 来为 `b'message data'` 获取一个（十六进制编码的）128 位验证代码:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

作为实际的例子，一个 Web 应用可为发送给用户的 cookies 进行对称签名，并在之后对其进行验证以确保它们没有被篡改:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
```

(续下页)

(接上页)

```
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

即使存在原生的密钥哈希模式，BLAKE2 也同样可在*hmac* 模块的 HMAC 构造过程中使用:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

随机哈希

用户可通过设置 *salt* 形参来为哈希函数引入随机化。随机哈希适用于防止对数字签名中使用的哈希函数进行碰撞攻击。

> 随机哈希被设计用来处理当一方（消息准备者）要生成由另一方（消息签名者）进行签名的全部或部分消息的情况。如果消息准备者能够找到加密哈希函数的碰撞现象（即两条消息产生相同的哈希值），则他们就可以准备将产生相同哈希值和数字签名但却具有不同结果的有意义的消息版本（例如向某个账户转入 $1,000,000 而不是 $10）。加密哈希函数的设计都是以防碰撞性能为其主要目标之一的，但是当前针对加密哈希函数的集中攻击可能导致特定加密哈希函数所提供的防碰撞性能低于预期。随机哈希为签名者提供了额外的保护，可以降低准备者在数字签名生成过程中使得两条或更多条消息最终产生相同哈希值的可能性 --- 即使为特定哈希函数找到碰撞现象是可行的。但是，当消息的所有部分均由签名者准备时，使用随机哈希可能降低数字签名所提供的安全性。
>
> (NIST SP-800-106 " 数字签名的随机哈希")

在 BLAKE2 中，盐值会在初始化期间作为对哈希函数的一次性输入而不是对每个压缩函数的输入来处理。

> **警告**
>
> 使用 BLAKE2 或任何其他通用加密哈希函数，例如 SHA-256 进行 *加盐哈希* (或纯哈希) 并不适用于对密码的哈希。请参阅 BLAKE2 FAQ 了解更多信息。

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
```

(续下页)

(接上页)

```
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

个性化

出于不同的目的强制让哈希函数为相同的输入生成不同的摘要有时也是有用的。正如 Skein 哈希函数的作者所言:

> 我们建议所有应用设计者慎重考虑这种做法；我们已看到有许多协议在协议的某一部分中计算出来的哈希值在另一个完全不同的部分中也可以被使用，因为两次哈希计算是针对类似或相关的数据进行的，这样攻击者可以强制应用为相同的输入生成哈希值。个性化协议中所使用的每个哈希函数将有效地阻止这种类型的攻击。
>
> (Skein 哈希函数族, p. 21)

BLAKE2 可通过向 *person* 参数传入字节串来进行个性化:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

个性化配合密钥模式也可被用来从单个密钥派生出多个不同密钥。

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

树形模式

以下是对包含两个叶子节点的最小树进行哈希的例子:

```
  10
 /  \
00  01
```

这个例子使用 64 字节内部摘要，返回 32 字节最终摘要:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
```

(续下页)

(接上页)

```
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

开发人员

BLAKE2 是由 *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn* 和 *Christian Winnerlein* 基于 *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier* 和 *Raphael C.-W. Phan* 所创造的 SHA-3 入围方案 BLAKE 进行设计的。

它使用的核心算法来自由 *Daniel J. Bernstein* 所设计的 ChaCha 加密。

stdlib 实现是基于 pyblake2 模块的。它由 *Dmitry Chestnykh* 在 *Samuel Neves* 所编写的 C 实现的基础上编写。此文档拷贝自 pyblake2 并由 *Dmitry Chestnykh* 撰写。

C 代码由 *Christian Heimes* 针对 Python 进行了部分的重写。

以下公共领域贡献同时适用于 C 哈希函数实现、扩展代码和本文档:

> 在法律许可的范围内，作者已将此软件的全部版权以及关联和邻接权利贡献到全球公共领域。此软件的发布不附带任何担保。
>
> 你应该已收到此软件附带的 CC0 公共领域专属证书的副本。如果没有，请参阅 https://creativecommons.org/publicdomain/zero/1.0/。

根据创意分享公共领域贡献 1.0 通用规范，下列人士为此项目的开发提供了帮助或对公共领域的修改作出了贡献:

- *Alexandr Sokolovskiy*

参见

模块 *hmac*
使用哈希运算来生成消息验证代码的模块。

模块 *base64*
针对非二进制环境对二进制哈希值进行编辑的另一种方式。

https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf
有关安全哈希算法的 FIPS 180-4 发布版。

https://csrc.nist.gov/pubs/fips/202/final
关于 SHA-3 标准的 FIPS 202 公告。

https://www.blake2.net/
BLAKE2 官方网站

https://en.wikipedia.org/wiki/Cryptographic_hash_function
包含关于哪些算法存在已知问题以及对其使用所造成的影响的信息的 Wikipedia 文章。

https://www.ietf.org/rfc/rfc8018.txt
PKCS #5: 基于密码的加密规范描述 2.1 版

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf
NIST 对基于密码的密钥派生的建议。

15.2 hmac --- 用于消息验证的密钥哈希

源代码: Lib/hmac.py

此模块实现了 HMAC 算法，算法的描述参见 **RFC 2104**。

hmac.**new**(*key*, *msg=None*, *digestmod*)

返回一个新的 hmac 对象。*key* 是一个指定密钥的 bytes 或 bytearray 对象。如果提供了 *msg*，将会调用 `update(msg)` 方法。*digestmod* 为 HMAC 对象所用的摘要名称、摘要构造器或模块。它可以是适用于*hashlib.new()* 的任何名称。虽然该参数位置靠后，但它却是必须的。

在 3.4 版本发生变更: 形参 *key* 可以为 bytes 或 bytearray 对象。形参 *msg* 可以为*hashlib* 所支持的任意类型。形参 *digestmod* 可以为某种哈希算法的名称。

在 3.8 版本发生变更: *digestmod* 参数现在是必须的。请将其作为关键字参数传入以避免当你没有初始 *msg* 时将导致的麻烦。

hmac.**digest**(*key*, *msg*, *digest*)

基于给定密钥 *key* 和 *digest* 返回 *msg* 的摘要。此函数等价于 `HMAC(key, msg, digest).digest()`，但使用了优化的 C 或内联实现，对放入内存的消息能处理得更快。形参 *key*, *msg* 和 *digest* 具有与*new()* 中相同的含义。

作为 CPython 的实现细节，优化的 C 实现仅当 *digest* 为字符串并且是一个 OpenSSL 所支持的摘要算法的名称时才会被使用。

Added in version 3.7.

HMAC 对象具有下列方法:

HMAC.**update**(*msg*)

用 *msg* 来更新 hmac 对象。重复调用相当于单次调用并传入所有参数的拼接结果: `m.update(a); m.update(b)` 等价于 `m.update(a + b)`。

在 3.4 版本发生变更: 形参 *msg* 可以为*hashlib* 所支持的任何类型。

HMAC.**digest**()

返回当前已传给*update()* 方法的字节串数据的摘要。这个字节串数据的长度将与传给构造器的摘要的长度 *digest_size* 相同。它可以包含非 ASCII 的字节，包括 NUL 字节。

> **警告**
>
> 在验证例程运行期间将*digest()* 的输出与外部提供的摘要进行比较时，建议使用*compare_digest()* 函数而不是 == 运算符以减少面对定时攻击的弱点。

HMAC.**hexdigest**()

类似于*digest()*但摘要会以两倍长度字符串的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

> **警告**
>
> 在验证例程运行期间将*hexdigest()*的输出与外部提供的摘要进行比较时，建议使用*compare_digest()*函数而不是 == 运算符以减少面对定时攻击的弱点。

HMAC.**copy**()

返回 hmac 对象的副本（“克隆）。这可被用来高效地计算共享相同初始子串的数据的摘要。

hash 对象具有以下属性：

HMAC.**digest_size**

以字节表示的结果 HMAC 摘要的大小。

HMAC.**block_size**

以字节表示的哈希算法的内部块大小。

Added in version 3.4.

HMAC.**name**

HMAC 的规范名称，总是为小写形式，例如 `hmac-md5`。

Added in version 3.4.

在 3.10 版本发生变更: 移除了未写入文档的属性 `HMAC.digest_cons`, `HMAC.inner` 和 `HMAC.outer`。

这个模块还提供了下列辅助函数:

hmac.**compare_digest**(*a*, *b*)

返回 `a == b`。此函数使用一种经专门设计的方式通过避免基于内容的短路行为来防止定时分析，使得它适合处理密码。*a* 和 *b* 必须为相同的类型：或者是*str*（仅限 ASCII 字符，如*HMAC.hexdigest()*的返回值），或者是*bytes-like object*。

> **备注**
>
> 如果 *a* 和 *b* 具有不同的长度，或者如果发生了错误，定时攻击在理论上可以获取有关 *a* 和 *b* 的类型和长度信息—但不能获取它们的值。

Added in version 3.3.

在 3.10 版本发生变更: 此函数在可能的情况下会在内部使用 OpenSSL 的 `CRYPTO_memcmp()`。

> **参见**
>
> **模块*hashlib***
> 提供安全哈希函数的 Python 模块。

15.3 `secrets` --- 生成管理密码的安全随机数

Added in version 3.6.

源代码: Lib/secrets.py

---

*secrets*模块用于生成高度加密的随机数，适于管理密码、账户验证、安全凭据及机密数据。

最好用*secrets*替代*random*模块的默认伪随机数生成器，该生成器适用于建模和模拟，不宜用于安全与加密。

参见

PEP 506

15.3.1 随机数

*secrets*模块是操作系统提供的最安全地随机性来源。

class secrets.**SystemRandom**

用操作系统提供的最高质量源生成随机数的类。详见*random.SystemRandom*。

secrets.**choice**(*seq*)

返回一个从非空序列中随机选取的元素。

secrets.**randbelow**(*exclusive_upper_bound*)

返回 [0, *exclusive_upper_bound*) 范围内的随机整数。

secrets.**randbits**(*k*)

返回有 *k* 个随机比特位的非负数。

15.3.2 生成 Token

*secrets*模块提供了生成安全 Token 的函数，适用于密码重置、密保 URL 等应用场景。

secrets.**token_bytes**([*nbytes=None*])

返回含 *nbytes* 个字节的随机字节字符串。如果未提供 *nbytes*，或 *nbytes* 为 None，则使用合理的默认值。

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])

返回十六进制随机文本字符串。字符串有 *nbytes* 个随机字节，每个字节转换为两个十六进制数码。未提供 *nbytes* 或为 None 时，则使用合理的默认值。

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])

返回安全的 URL 随机文本字符串，包含 *nbytes* 个随机字节。文本用 Base64 编码，平均来说，每个字节对应 1.3 个结果字符。未提供 *nbytes* 或为 None 时，则使用合理的默认值。

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

Token 应当使用多少个字节？

为了在面对 暴力攻击 时保证安全，Token 的随机性必须足够高。随着计算机推衍能力的不断提升，随机性的安全标准也要不断提高。比如 2015 年，32 字节（256 位）的随机性对于*secrets*模块的典型用例就已经足够了。

要自行管理 Token 长度的用户，可以通过为 token_* 函数指定*int*参数显式指定 Token 要使用多大的随机性。该参数以字节数表示随机性大小。

反之，如果未提供参数，或参数为 None，则 token_* 函数将使用合理的默认值。

> **备注**
>
> 该默认值随时可能会改变，比如，版本更新的时候。

15.3.3 其他功能

secrets.**compare_digest**(*a*, *b*)

如果字符串或字节型对象 *a* 与 *b* 相等则返回 `True`，否则返回 `False`，使用了“常数时间比较”来降低 定时攻击 的风险。请参阅*hmac.compare_digest()* 了解更多细节。

15.3.4 应用技巧与最佳实践

本节展示了一些使用*secrets* 管理基本安全级别的应用技巧和最佳实践。

生成长度为八个字符的字母数字密码：

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

> **备注**
>
> 应用程序不应该 **以可恢复的格式存储密码**，无论是纯文本的还是加密的。它们应当使用高加密强度的单向（不可逆）哈希函数加盐并执行哈希运算。

生成长度为十个字符的字母数字密码，包含至少一个小写字母，至少一个大写字母以及至少三个数字：

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

生成 XKCD 风格的密码串：

```
import secrets
# 在标准 Linux 系统中，使用方便的字典文件。
# 其他系统平台可能需要提供它们专用的词列表。
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

生成临时密保 URL，包含密码恢复应用的安全 Token：

```
import secrets
url = 'https://example.com/reset=' + secrets.token_urlsafe()
```

CHAPTER 16

通用操作系统服务

本章中描述的各模块提供了在（几乎）所有的操作系统上可用的操作系统特性的接口，例如文件和时钟。这些接口通常以 Unix 或 C 接口为参考对象，不过在大多数其他系统上也可用。这里有一个概述：

16.1 os --- 多种操作系统接口

源代码： Lib/os.py

本模块提供了一种使用与操作系统相关的功能的便捷式途径。如果你只是想读写一个文件，请参阅*open()*，如果你想操作文件路径，请参阅*os.path* 模块，如果你想读取通过命令行给出的所有文件中的所有行，请参阅*fileinput* 模块。为了创建临时文件和目录，请参阅*tempfile* 模块，对于高级文件和目录处理，请参阅*shutil* 模块。

关于这些函数的适用性的说明：

- Python 中所有依赖于操作系统的内置模块的设计都是这样，只要不同的操作系统某一相同的功能可用，它就使用相同的接口。例如，函数 `os.stat(path)` 以相同的格式返回关于 *path* 的状态信息（该格式源于 POSIX 接口）。
- 特定于某一操作系统的扩展通过操作*os* 模块也是可用的，但是使用它们当然是对可移植性的一种威胁。
- 所有接受路径或文件名的函数都同时支持字节串和字符串对象，并在返回路径或文件名时使用相应类型的对象作为结果。
- 在 VxWorks 系统上，os.popen, os.fork, os.execv 和 os.spawn*p* 都未支持。
- 在 WebAssembly 平台、Android 和 iOS 上，*os* 模块的很大一部分都不可用或具有不同的行为。与进程相关联的 API (例如*fork()*, *execve()*) 和资源 (例如*nice()*) 都不可用。其他诸如*getuid()* 和*getpid()* 等则为模拟或空盒。WebAssembly 平台还缺少对信号 (例如*kill()*, *wait()*) 的支持。

备注

如果使用无效或无法访问的文件名与路径，或者其他类型正确但操作系统不接受的参数，此模块的所有函数都抛出*OSError* （或者它的子类）。

exception os.**error**

内建的*OSError* 异常的一个别名。

os.**name**

导入的依赖特定操作系统的模块的名称。以下名称目前已注册: `'posix'`, `'nt'`, `'java'`.

> **参见**
>
> *sys.platform* 具有更细的粒度。*os.uname()* 将给出基于不同系统的版本信息。
>
> *platform* 模块对系统的标识有更详细的检查。

16.1.1 文件名，命令行参数，以及环境变量。

在 Python 中，使用字符串类型表示文件名、命令行参数和环境变量。在某些系统上，在将这些字符串传递给操作系统之前，必须将这些字符串解码为字节。Python 使用*filesystem encoding and error handler* 来执行此转换（请参阅*sys.getfilesystemencoding()* ）。

filesystem encoding and error handler 是在 Python 启动时通过 `PyConfig_Read()` 函数来配置的：请参阅 `PyConfig` 的 `filesystem_encoding` 和 `filesystem_errors` 等成员。

在 3.1 版本发生变更: 在某些系统上，使用文件系统编码格式进行转换可能会失败。在这种情况下，Python 会使用*surrogateescape* 编码错误处理器，这意味着不可解码的字节在解码时会被 Unicode 字符 U+DC*xx* 替换，并且这些字节在编码时又会再次被转换为原始字节。

文件系统编码器 必须保证能成功解码所有 128 以内的字节。如果不能保证，API 函数可能触发*UnicodeError* 。

另请参见*locale encoding*。

16.1.2 Python UTF-8 模式

Added in version 3.7: 有关更多详细信息，请参阅 **PEP 540** 。

Python UTF-8 模式会忽略*locale encoding* 并强制使用 UTF-8 编码。

- 用 UTF-8 作为文件系统编码。
- *sys.getfilesystemencoding()* 返回 `'utf-8'`。
- *locale.getpreferredencoding()* 返回 `'utf-8'` (*do_setlocale* 参数不起作用)。
- *sys.stdin*, *sys.stdout* 和*sys.stderr* 都将 UTF-8 用作它们的文本编码，并且为*sys.stdin* 和*sys.stdout* 启用 `surrogateescape` 错误处理器 (*sys.stderr* 会继续使用 `backslashreplace` 如同在默认的局部感知模式下一样)
- 在 Unix 上，*os.device_encoding()* 返回 `'utf-8'` 而不是设备的编码格式。

请注意 UTF-8 模式下的标准流设置可以被 `PYTHONIOENCODING` 所覆盖（在默认的区域感知模式下也同样如此）。

作为低层级 API 发生改变的结果，其他高层级 API 也会表现出不同的默认行为:

- 命令行参数，环境变量和文件名会使用 UTF-8 编码来解码为文本。
- *os.fsdecode()* 和*os.fsencode()* 使用 UTF-8 编码格式。
- *open()*, *io.open()* 和*codecs.open()* 默认会使用 UTF-8 编码格式。但是，它们默认仍将使用 strict 错误处理器因此试图在文本模式下打开二进制文件可能会引发异常而不是产生无意义的数据。

如果在 Python 启动时 LC_CTYPE 区域设为 `C` 或 `POSIX` ，则启用*Python UTF-8 模式* (参见 `PyConfig_Read()` 函数)。

它可以通过命令行选项 `-X utf8` 和环境变量 `PYTHONUTF8`，来启用或禁用。

如果没有设置 PYTHONUTF8 环境变量，那么解释器默认使用当前的地区设置，除非当前地区识别为基于 ASCII 的传统地区（如 PYTHONCOERCECLOCALE 所述），并且 locale coercion 被禁用或失败。在这种传统地区，除非显式指明不要如此，解释器将默认启用 UTF-8 模式。

Python UTF-8 模式只能在 Python 启动时启用。其值可以从*sys.flags.utf8_mode* 读取。

另请参阅 在 Windows 中的 UTF-8 模式和*filesystem encoding and error handler*。

参见

PEP 686
Python 3.15 将把*Python UTF-8 模式* 设为默认值。

16.1.3 进程参数

这些函数和数据项提供了操作当前进程和用户的信息。

os.**ctermid**()

返回与进程控制终端对应的文件名。

Availability: Unix, not WASI.

os.**environ**

一个*mapping* 对象，其中键值是代表进程环境的字符串。例如，environ['HOME'] 是你的主目录（在某些平台上）的路径名，相当于 C 中的 getenv("HOME")。

这个映射是在第一次导入*os* 模块时捕获的，通常作为 Python 启动时处理 site.py 的一部分。除了通过直接修改*os.environ* 之外，在此之后对环境所做的更改不会反映在*os.environ* 中。

该映射除了可以用于查询环境外，还能用于修改环境。当该映射被修改时，将自动调用*putenv()*。

在 Unix 系统上，键和值会使用*sys.getfilesystemencoding()* 和 'surrogateescape' 的错误处理。如果你想使用其他的编码，使用*environb*。

在 Windows 上，这些键会被转换为大写形式。这也会在获取、设备或删除条目时被应用。例如，environ['monty'] = 'python' 会将键 'MONTY' 映射到值 'python'。

备注

直接调用*putenv()* 并不会影响*os.environ*，所以推荐直接修改*os.environ*。

备注

在某些平台上，包括 FreeBSD 和 macOS 等，设置 environ 可能会导致内存泄漏。请参阅有关 putenv() 的系统文档。

可以删除映射中的元素来删除对应的环境变量。当从*os.environ* 删除元素时，以及调用 pop() 或 clear() 之一时，将自动调用*unsetenv()*。

在 3.9 版本发生变更: 已更新并支持了 **PEP 584** 的合并 (|) 和更新 (|=) 运算符。

os.**environb**

environ 的字节版本：一个*mapping* 对象，其中键值都是*bytes* 对象，代表进程环境。*environ* 和*environb* 是同步的（修改*environb* 会更新*environ*，反之亦然）。

environb 仅在*supports_bytes_environ* 为 True 时可用。

Added in version 3.2.

在 3.9 版本发生变更: 已更新并支持了 **PEP 584** 的合并 (|) 和更新 (|=) 运算符。

os.**chdir**(*path*)

os.**fchdir**(*fd*)

os.**getcwd**()

以上函数请参阅文件和目录 。

os.**fsencode**(*filename*)

将类似路径形式的 *filename* 编码为*filesystem encoding and error handler*；原样返回*bytes*。

fsdecode() 是此函数的逆向函数。

Added in version 3.2.

在 3.6 版本发生变更: 增加对实现了*os.PathLike* 接口的对象的支持。

os.**fsdecode**(*filename*)

根据*filesystem encoding and error handler* 来解码类似路径形式的 *filename*；原样返回*str*。

fsencode() 是此函数的逆向函数。

Added in version 3.2.

在 3.6 版本发生变更: 增加对实现了*os.PathLike* 接口的对象的支持。

os.**fspath**(*path*)

返回路径的文件系统表示。

如果传入的是*str* 或*bytes* 类型的字符串，将原样返回。否则*__fspath__()* 将被调用，如果得到的是一个*str* 或*bytes* 类型的对象，那就返回这个值。其他所有情况则会抛出*TypeError* 异常。

Added in version 3.6.

class os.**PathLike**

某些对象用于表示文件系统中的路径（如*pathlib.PurePath* 对象），本类是这些对象的抽象基类。

Added in version 3.6.

abstractmethod **__fspath__**()

返回当前对象的文件系统表示。

这个方法只应该返回一个*str* 字符串或*bytes* 字节串，请优先选择*str* 字符串。

os.**getenv**(*key*, *default=None*)

如果环境变量 *key* 存在则将其值作为字符串返回，如果不存在则返回 *default*。*key* 是一个字符串。请注意由于*getenv()* 使用了*os.environ*，因此*getenv()* 的映射同样也会在导入时被捕获，并且该函数可能无法反映未来的环境变化。

在 Unix 系统上，键和值会使用*sys.getfilesystemencoding()* 和 'surrogateescape' 错误处理进行解码。如果你想使用其他的编码，使用*os.getenvb()*。

Availability: Unix, Windows.

os.**getenvb**(*key*, *default=None*)

如果环境变量 *key* 存在则将其值作为字节串返回，如果不存在则返回 *default*。*key* 必须为字节串。请注意由于*getenvb()* 使用了*os.environb*，因此*getenvb()* 的映射同样也会在导入时被捕获，并且该函数可能无法反映未来的环境变化。

getenvb() 仅在*supports_bytes_environ* 为 True 时可用。

Availability: Unix.

Added in version 3.2.

os.**get_exec_path**(*env=None*)

返回将用于搜索可执行文件的目录列表，与在外壳程序中启动一个进程时相似。指定的 *env* 应为用于搜索 PATH 的环境变量字典。默认情况下，当 *env* 为 None 时，将会使用*environ* 。

Added in version 3.2.

os.**getegid**()

返回当前进程的有效组 ID。对应当前进程执行文件的"set id" 位。

Availability: Unix, not WASI.

os.**geteuid**()

返回当前进程的有效用户 ID。

Availability: Unix, not WASI.

os.**getgid**()

返回当前进程的实际组 ID。

Availability: Unix.

该函数在 WASI 上为空代码段，请参阅*WebAssembly 平台* 了解详情。

os.**getgrouplist**(*user*, *group*, /)

返回 *user* 所属的组 ID 列表。如果 *group* 不在该列表中，它将被包括在内；通常，*group* 将会被指定为来自 *user* 的密码记录文件的组 ID 字段，因为在其他情况下该组 ID 有可能会被略去。

Availability: Unix, not WASI.

Added in version 3.3.

os.**getgroups**()

返回当前进程关联的附加组 ID 列表

Availability: Unix, not WASI.

> **备注**
>
> 在 macOS 上，*getgroups()* 的行为与其他 Unix 平台有所不同。如果 Python 解释器是以 `10.5` 或更早版本作为部署目标的，则*getgroups()* 会返回与当前用户进程相关联的有效组 ID 列表；该列表受限于系统预定义的条目数量，通常为 16，并且在适当的权限下还可通过调用*setgroups()* 来修改。如果所用的部署目标版本大于 `10.5`，则*getgroups()* 会返回与进程的有效用户 ID 相关联的当前组访问列表；组访问列表可能会在进程的生命周期之内发生改变，它不会受对*setgroups()* 的调用影响，且其长度也不会被限制为 16。部署目标值 `MACOSX_DEPLOYMENT_TARGET` 可以通过*sysconfig.get_config_var()* 来获取。

os.**getlogin**()

返回通过控制终端进程进行登录的用户名。在多数情况下，使用*getpass.getuser()* 会更有效，因为后者会通过检查环境变量 `LOGNAME` 或 `USERNAME` 来查找用户，再由 `pwd.getpwuid(os.getuid())[0]` 来获取当前用户 ID 的登录名。

Availability: Unix, Windows, not WASI.

os.**getpgid**(*pid*)

根据进程 id *pid* 返回进程的组 ID 列表。如果 *pid* 为 0，则返回当前进程的进程组 ID 列表

Availability: Unix, not WASI.

os.**getpgrp**()

返回当时进程组的 ID

Availability: Unix, not WASI.

os.**getpid**()

返回当前进程 ID

该函数在 WASI 上为空代码段，请参阅*WebAssembly 平台* 了解详情。

os.**getppid**()

返回父进程 ID。当父进程已经结束，在 Unix 中返回的 ID 是初始进程 (1) 中的一个，在 Windows 中仍然是同一个进程 ID，该进程 ID 有可能已经被进行进程所占用。

Availability: Unix, Windows, not WASI.

在 3.2 版本发生变更: 添加 WIndows 的支持。

os.**getpriority**(*which*, *who*)

获取程序调度优先级。*which* 参数值可以是*PRIO_PROCESS*，*PRIO_PGRP*，或*PRIO_USER* 中的一个，*who* 是相对于 *which* (*PRIO_PROCESS* 的进程标识符，*PRIO_PGRP* 的进程组标识符和*PRIO_USER* 的用户 ID)。当 *who* 为 0 时（分别）表示调用的进程，调用进程的进程组或调用进程所属的真实用户 ID。

Availability: Unix, not WASI.

Added in version 3.3.

os.**PRIO_PROCESS**

os.**PRIO_PGRP**

os.**PRIO_USER**

函数*getpriority()* 和*setpriority()* 的参数。

Availability: Unix, not WASI.

Added in version 3.3.

os.**PRIO_DARWIN_THREAD**

os.**PRIO_DARWIN_PROCESS**

os.**PRIO_DARWIN_BG**

os.**PRIO_DARWIN_NONUI**

函数*getpriority()* 和*setpriority()* 的参数。

Availability: macOS

Added in version 3.12.

os.**getresuid**()

返回一个由 (ruid, euid, suid) 所组成的元组，分别表示当前进程的真实用户 ID，有效用户 ID 和暂存用户 ID。

Availability: Unix, not WASI.

Added in version 3.2.

os.**getresgid**()

返回一个由 (rgid, egid, sgid) 所组成的元组，分别表示当前进程的真实组 ID，有效组 ID 和暂存组 ID。

Availability: Unix, not WASI.

Added in version 3.2.

os.**getuid**()

返回当前进程的真实用户 ID。

Availability: Unix.

该函数在 WASI 上为空代码段，请参阅*WebAssembly 平台* 了解详情。

os.**initgroups**(*username*, *gid*, */*)

调用系统 initgroups()，使用指定用户所在的所有值来初始化组访问列表，包括指定的组 ID。

Availability: Unix, not WASI, not Android.

Added in version 3.2.

os.**putenv**(*key*, *value*, /)

将名为 *key* 的环境变量值设置为 *value*。该变量名修改会影响由*os.system()*，*popen()* ，*fork()* 和*execv()* 发起的子进程。

对*os.environ* 中的项目的赋值会自动转化为对*putenv()* 的相应调用；然而，对*putenv()* 的调用并不更新*os.environ* ，所以实际上最好是赋值到*os.environ* 的项目。这也适用于*getenv()* 和*getenvb()* ，它们分别使用*os.environ* 和*os.environb* 在它们的实现中。

备注

在某些平台上，包括 FreeBSD 和 macOS 等，设置 environ 可能会导致内存泄漏。请参阅有关 putenv() 的系统文档。

引发一个审计事件 os.putenv 并附带参数 key, value。

在 3.9 版本发生变更: 该函数现在总是可用。

os.**setegid**(*egid*, /)

设置当前进程的有效组 ID。

Availability: Unix, not WASI, not Android.

os.**seteuid**(*euid*, /)

设置当前进程的有效用户 ID。

Availability: Unix, not WASI, not Android.

os.**setgid**(*gid*, /)

设置当前进程的组 ID。

Availability: Unix, not WASI, not Android.

os.**setgroups**(*groups*, /)

将 *group* 参数值设置为与当进程相关联的附加组 ID 列表。*group* 参数必须为一个序列，每个元素应为每个组的数字 ID。该操作通常只适用于超级用户。

Availability: Unix, not WASI.

备注

在 macOS 中，*groups* 的长度不能超过系统定义的最大有效组 ID 数量，通常为 16。对于未返回与调用 setgroups() 产生的相同组列表的情况，请参阅*getgroups()* 的文档。

os.**setns**(*fd*, *nstype=0*)

将当前线程与 Linux 命名空间重新关联。详情参见 *setns(2)* 和 *namespaces(7)* 手册页面。

如果 *fd* 是指向一个 /proc/*pid*/ns/ 链接，setns() 会将调用线程与该链接所关联的命名空间重新关联起来，并且 *nstype* 可以设为某个*CLONE_NEW** 常量 以便对操作施加约束 (0 表示没有任何约束)。

自 Linux 5.8 起，*fd* 可以是通过*pidfd_open()* 获取的 PID 文件描述符。在这种情况下，setns() 会将调用线程重新关联到与 *fd* 引用的线程相同的一个或多个命名空间。位掩码 *nstype* 通常会结合一个或多个*CLONE_NEW** 常量 ，例如 setns(fd, os.CLONE_NEWUTS | os.CLONE_NEWPID)，其施加的任何约束限制仍然保留，调用者在未指定的命名空间中的成员资格保持不变。

fd 可以是任何带有*fileno()* 方法的对象，或是一个原始文件描述符。

此示例将线程与 init 进程的网络命名空间进行了重新关联:

```
fd = os.open("/proc/1/ns/net", os.O_RDONLY)
os.setns(fd, os.CLONE_NEWNET)
os.close(fd)
```

Availability: Linux >= 3.0 with glibc >= 2.14.

Added in version 3.12.

参见

unshare() 函数。

os.**setpgrp**()

在系统调用 `setpgrp()` 和 `setpgrp(0, 0)` 中择一调用，具体取决于何种实现版本可用（如果任一实现存在的话）。请参阅 Unix 手册以了解语义。

Availability: Unix, not WASI.

os.**setpgid**(*pid*, *pgrp*, /)

使用系统调用 `setpgid()` 将 *pid* 对应进程的组 ID 设置为 *pgrp*。请参阅 Unix 手册以了解语义。

Availability: Unix, not WASI.

os.**setpriority**(*which*, *who*, *priority*)

设置程序调度优先级。*which* 的值为*PRIO_PROCESS*, *PRIO_PGRP* 或*PRIO_USER* 之一，而 *who* 会相对于 *which* (*PRIO_PROCESS* 的进程标识符, *PRIO_PGRP* 的进程组标识符和*PRIO_USER* 的用户 ID) 被解析。*who* 值为零 (分别) 表示调用进程，调用进程的进程组或调用进程的真实用户 ID。*priority* 是范围在 -20 至 19 的值。默认优先级为 0；较小的优先级数值会更优先被调度。

Availability: Unix, not WASI.

Added in version 3.3.

os.**setregid**(*rgid*, *egid*, /)

设置当前进程的真实和有效组 ID。

Availability: Unix, not WASI, not Android.

os.**setresgid**(*rgid*, *egid*, *sgid*, /)

设置当前进程的真实，有效和暂存组 ID。

Availability: Unix, not WASI, not Android.

Added in version 3.2.

os.**setresuid**(*ruid*, *euid*, *suid*, /)

设置当前进程的真实，有效和暂存用户 ID。

Availability: Unix, not WASI, not Android.

Added in version 3.2.

os.**setreuid**(*ruid*, *euid*, /)

设置当前进程的真实和有效用户 ID。

Availability: Unix, not WASI, not Android.

os.**getsid**(*pid*, /)

调用系统调用 `getsid()`。其语义请参见 Unix 手册。

Availability: Unix, not WASI.

os.**setsid**()

调用系统调用 `setsid()`。其语义请参见 Unix 手册。

Availability: Unix, not WASI.

os.**setuid**(*uid*, */*)

设置当前进程的用户 ID。

Availability: Unix, not WASI, not Android.

os.**strerror**(*code*, */*)

根据 *code* 中的错误码返回错误消息。如果 `strerror()` 返回 `NULL`，说明给出的是未知错误码，则会引发*ValueError*。

os.**supports_bytes_environ**

如果操作系统上原生环境类型是字节型则为 `True` (例如在 Windows 上为 `False`)。

Added in version 3.2.

os.**umask**(*mask*, */*)

设定当前数值掩码并返回之前的掩码。

该函数在 WASI 上为空代码段，请参阅*WebAssembly 平台* 了解详情。

os.**uname**()

返回当前操作系统的识别信息。返回值是一个有 5 个属性的对象：

- `sysname` - 操作系统名
- `nodename` - 机器在网络上的名称（需要先设定）
- `release` - 操作系统发行信息
- `version` - 操作系统版本信息
- `machine` - 硬件标识符

为了向后兼容，该对象也是可迭代的，像是一个按照 `sysname`，`nodename`，`release`，`version`，和 `machine` 顺序组成的元组。

有些系统会将 `nodename` 截短为 8 个字符或截短至前缀部分；获取主机名的一个更好方式是*socket.gethostname()* 或甚至可以用 `socket.gethostbyaddr(socket.gethostname())`。

在 macOS, iOS 和 Android 上，这将返回 *内核* 名称和版本 (即在 macOS 和 iOS 上为 `'Darwin'`; 在 Android 上为 `'Linux'`)。*platform.uname()* 可被用来在 iOS 和 Android 上获取面向用户的操作系统名称和版本。

Availability: Unix.

在 3.3 版本发生变更: 返回结果的类型由元组变成一个类似元组的对象，同时具有命名的属性。

os.**unsetenv**(*key*, */*)

取消设置（删除）名为 *key* 的环境变量。变量名的改变会影响由*os.system()*，*popen()*，*fork()* 和*execv()* 触发的子进程。

删除*os.environ* 中的项目会自动转化为对*unsetenv()* 的相应调用；然而，对*unsetenv()* 的调用并不更新*os.environ*，所以实际上最好是删除*os.environ* 的项目。

引发一个*审计事件* `os.unsetenv` 并附带参数 `key`。

在 3.9 版本发生变更: 该函数现在总是可用，并且在 Windows 上也可用。

os.**unshare**(*flags*)

拆分进程执行上下文的部分内容，并将其移入新创建的命名空间中。请参阅 *unshare(2)* 手册页了解详情。*flags* 参数是一个位掩码，它组合了零个或多个*CLONE_* 常量*，用于指定执行上下文中的哪些部分应从现有关联中解除共享并移动到新的命名空间。如果 *flags* 参数为 `0`，则不会对调用方进程的执行上下文进行任何更改。

Availability: Linux >= 2.6.16.

Added in version 3.12.

> **参见**
>
> *setns()* 函数。

unshare() 函数的旗标，如果实现支持。访问 Linux 手册中的 *unshare(2)* 以获取关于实际影响和可用性的信息。

os.**CLONE_FILES**

os.**CLONE_FS**

os.**CLONE_NEWCGROUP**

os.**CLONE_NEWIPC**

os.**CLONE_NEWNET**

os.**CLONE_NEWNS**

os.**CLONE_NEWPID**

os.**CLONE_NEWTIME**

os.**CLONE_NEWUSER**

os.**CLONE_NEWUTS**

os.**CLONE_SIGHAND**

os.**CLONE_SYSVSEM**

os.**CLONE_THREAD**

os.**CLONE_VM**

16.1.4 创建文件对象

这些函数创建新的*file objects* 。（参见*open()* 以获取打开文件描述符的相关信息。）

os.**fdopen**(*fd*, **args*, ***kwargs*)

返回打开文件描述符 *fd* 对应文件的对象。类似内建*open()* 函数，二者接受同样的参数。不同之处在于*fdopen()* 第一个参数应该为整数。

16.1.5 文件描述符操作

这些函数对文件描述符所引用的 I/O 流进行操作。

文件描述符是一些小的整数，对应于当前进程所打开的文件。例如，标准输入的文件描述符通常是 0，标准输出是 1，标准错误是 2。之后被进程打开的文件的文件描述符会被依次指定为 3，4，5 等。“文件描述符”这个词有点误导性，在 Unix 平台中套接字和管道也被文件描述符所引用。

当需要时，可以用*fileno()* 可以获得*file object* 所对应的文件描述符。需要注意的是，直接使用文件描述符会绕过文件对象的方法，会忽略如数据内部缓冲等情况。

os.**close**(*fd*)

关闭文件描述符 *fd*。

> **备注**
>
> 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要关闭由内建函数*open()*、*popen()* 或*fdopen()* 返回的” 文件对象”，则应使用其相应的*close()* 方法。

os.**closerange**(*fd_low*, *fd_high*, */*)

关闭从 *fd_low* （包括）到 *fd_high* （排除）间的文件描述符，并忽略错误。类似（但快于）：

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**copy_file_range**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

从文件描述符 *src* 自偏移量 *offset_src* 起的位置拷贝 *count* 个字节到文件描述符 *dst* 自偏移量 *offset_dst* 起的位置。如果 *offset_src* 为 None，则从当前位置读取 *src*；相应地 *offset_dst* 也是如此。

在早于 5.3 版的 Linux 内核中，*src* 和 *dst* 指向的文件必须位于相同的文件系统中，否则会引发 *OSError* 并将 *errno* 设为 *errno.EXDEV*。

执行这种拷贝无需付出将数据从内核传输到用户空间再返回内核的额外耗费。此外，某些文件系统还可以实现进一步的优化，例如使用引用链接（即两个或多个共享指向相同写入时复制磁盘块的指针的 inode；支持的文件系统包括 btrfs 和 XFS）和服务器端拷贝（对于 NFS）。

此函数在两个文件描述符之间拷贝字节数据。文本选项，如编码格式和行结束符等将被忽略。

返回值是复制的字节的数目。这可能低于需求的数目。

> **备注**
>
> 在 Linux 上，*os.copy_file_range()* 不应被用于从特殊的文件系统如 procfs 和 sysfs 复制特定范围的伪文件。因为已知的 Linux 内核问题它将总是不复制任何字节并返回 0 就像文件是空的一样。

Availability: Linux >= 4.5 with glibc >= 2.27.

Added in version 3.8.

os.**device_encoding**(*fd*)

如果连接到终端，则返回一个与 *fd* 关联的设备描述字符，否则返回 *None*。

在 Unix 上，如果启用了 *Python UTF-8 模式*，则返回 'UTF-8' 而不是设备的编码格式。

在 3.10 版本发生变更: 在 Unix 上，该函数现在实现了 Python UTF-8 模式。

os.**dup**(*fd*, */*)

返回一个文件描述符 *fd* 的副本。该文件描述符的副本是 *不可继承的*。

在 Windows 中，当复制一个标准流（0: stdin, 1: stdout, 2: stderr）时，新的文件描述符是 *可继承的*。

Availability: not WASI.

在 3.4 版本发生变更: 新的文件描述符现在是不可继承的。

os.**dup2**(*fd*, *fd2*, *inheritable=True*)

把文件描述符 *fd* 复制为 *fd2*，必要时先关闭后者。返回 *fd2*。新的文件描述符默认是 *可继承的*，除非在 *inheritable* 为 False 时，是不可继承的。

Availability: not WASI.

在 3.4 版本发生变更: 添加可选参数 *inheritable*。

在 3.7 版本发生变更: 成功时返回 *fd2*，以过去的版本中，总是返回 None。

os.**fchmod**(*fd*, *mode*)

将 *fd* 指定文件的权限状态修改为 *mode*。可以参考 *chmod()* 中列出 *mode* 的可用值。从 Python 3.3 开始，这相当于 os.chmod(fd, mode)。

引发一个 *审计事件* os.chmod 并附带参数 path, mode, dir_fd。

Availability: Unix, Windows.

此函数在 WASI 是受限的，请参阅 *WebAssembly 平台* 了解详情。

在 3.13 版本发生变更: 增加 Windows 上的支持。

os.**fchown**(*fd*, *uid*, *gid*)

分别将 *fd* 指定文件的所有者和组 ID 修改为 *uid* 和 *gid* 的值。若不想变更其中的某个 ID，可将相应值设为 -1。参考*chown()*。从 Python 3.3 开始，这相当于 `os.chown(fd, uid, gid)`。

引发一个审计事件 `os.chown` 并附带参数 `path`, `uid`, `gid`, `dir_fd`。

Availability: Unix.

此函数在 WASI 是受限的，请参阅*WebAssembly 平台* 了解详情。

os.**fdatasync**(*fd*)

强制将文件描述符 *fd* 指定文件写入磁盘。不强制更新元数据。

Availability: Unix.

备注

该功能在 MacOS 中不可用。

os.**fpathconf**(*fd*, *name*, */*)

返回与打开的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 `pathconf_names` 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 `pathconf_names`，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

从 Python 3.3 起，此功能等价于 `os.pathconf(fd, name)`。

Availability: Unix.

os.**fstat**(*fd*)

获取文件描述符 *fd* 的状态. 返回一个*stat_result* 对象。

从 Python 3.3 起，此功能等价于 `os.stat(fd)`。

参见

stat() 函数。

os.**fstatvfs**(*fd*, */*)

返回文件系统的信息，该文件系统是文件描述符 *fd* 指向的文件所在的文件系统，与*statvfs()* 一样。从 Python 3.3 开始，它等效于 `os.statvfs(fd)`。

Availability: Unix.

os.**fsync**(*fd*)

强制将文件描述符 *fd* 指向的文件写入磁盘。在 Unix 上，这将调用原生 `fsync()` 函数；在 Windows 上，则是 MS `_commit()` 函数。

如果要写入的是缓冲区内的 Python 文件对象 *f*，请先执行 `f.flush()`，然后执行 `os.fsync(f.fileno())`，以确保与 *f* 关联的所有内部缓冲区都写入磁盘。

Availability: Unix, Windows.

os.**ftruncate**(*fd*, *length*, /)

截断文件描述符 *fd* 指向的文件，以使其最大为 *length* 字节。从 Python 3.3 开始，它等效于 `os.truncate(fd, length)`。

引发一个审计事件 `os.truncate` 并附带参数 `fd`, `length`。

Availability: Unix, Windows.

在 3.5 版本发生变更: 添加了 Windows 支持

os.**get_blocking**(*fd*, /)

获取文件描述符的阻塞模式：如果设置了*O_NONBLOCK* 标志位，返回 `False`，如果该标志位被清除，返回 `True`。

参见*set_blocking()* 和*socket.socket.setblocking()*。

Availability: Unix, Windows.

此函数在 WASI 是受限的，请参阅*WebAssembly 平台* 了解详情。

在 Windows 上，此函数仅限于管道。

Added in version 3.5.

在 3.12 版本发生变更: 增加了在 Windows 上对于管道的支持。

os.**grantpt**(*fd*, /)

允许访问与文件描述符 *fd* 所指向的主伪终端设备相关联的从伪终端设备。文件描述符 *fd* 在失败时不会被关闭。

调用 C 标准库函数 `grantpt()`。

Availability: Unix, not WASI.

Added in version 3.13.

os.**isatty**(*fd*, /)

如果文件描述符 *fd* 打开且已连接至 tty 设备（或类 tty 设备），返回 `True`，否则返回 `False`。

os.**lockf**(*fd*, *cmd*, *len*, /)

在打开的文件描述符上，使用、测试或删除 POSIX 锁。*fd* 是一个打开的文件描述符。*cmd* 指定要进行的操作，它们是*F_LOCK*、*F_TLOCK*、*F_ULOCK* 或*F_TEST* 中的一个。*len* 指定哪部分文件需要锁定。

引发一个审计事件 `os.lockf` 并附带参数 `fd`, `cmd`, `len`。

Availability: Unix.

Added in version 3.3.

os.**F_LOCK**

os.**F_TLOCK**

os.**F_ULOCK**

os.**F_TEST**

标志位，用于指定*lockf()* 进行哪一种操作。

Availability: Unix.

Added in version 3.3.

os.**login_tty**(*fd*, /)

准备 tty 设置 fd 为新登录会话的文件描述符。设置调用方进程为会话主进程；设置该 tty 为主控 tty，调用方进程使用其 stdin, stdout 和 stderr；关闭 fd。

Availability: Unix, not WASI.

Added in version 3.11.

os.**lseek**(*fd*, *pos*, *whence*, */*)

将文件描述符 *fd* 的当前位置设为位置 *pos*，经 *whence* 修正，并返回相对于文件开头的以字节为单位的新位置。*whence* 的有效值为：

- *SEEK_SET* 或 0 -- 相对于文件开头设置 *pos*
- *SEEK_CUR* 或 1 -- 相对于当前文件位置设置 *pos*
- *SEEK_END* 或 2 -- 相对于文件末尾设置 *pos*
- *SEEK_HOLE* -- 将 *pos* 设置为相对于 *pos* 的下一个数据位置
- *SEEK_DATA* -- 将 *pos* 设为相对于 *pos* 的下一个数据空位

在 3.3 版本发生变更: 增加对 SEEK_HOLE 和 SEEK_DATA 的支持。

os.**SEEK_SET**

os.**SEEK_CUR**

os.**SEEK_END**

传给*lseek()* 函数和*文件型对象* 上*seek()* 方法的形参，用于调整文件位置指示器。

SEEK_SET
相对于文件的开头调整文件位置。

SEEK_CUR
相对于当前文件位置调整文件位置。

SEEK_END
相对于文件的末尾调整文件位置。

它们的值分别为 0, 1 和 2。

os.**SEEK_HOLE**

os.**SEEK_DATA**

传给*lseek()* 函数和*文件型对象* 上*seek()* 方法的形参，用于查找文件数据和稀疏分配的文件中的空洞。

SEEK_DATA
相对于查找位置调整到下一个包含数据的位置的文件偏移量。

SEEK_HOLE
相对于查找位置调整到下一个包含空洞的位置的文件偏移量。空洞被定义为零值的序列。

> **备注**
>
> 这些操作只对支持它们的文件系统具有意义。

Availability: Linux >= 3.1, macOS, Unix

Added in version 3.3.

os.**open**(*path*, *flags*, *mode=0o777*, *, *dir_fd=None*)

打开文件 *path*，根据 *flags* 设置各种标志位，并根据 *mode* 设置其权限状态。当计算 *mode* 时，会首先根据当前 umask 值将部分权限去除。本方法返回新文件的描述符。新的文件描述符是*不可继承* 的。

有关 flag 和 mode 取值的说明，请参见 C 运行时文档。标志位常量（如*O_RDONLY* 和*O_WRONLY*）在*os* 模块中定义。特别地，在 Windows 上需要添加*O_BINARY* 才能以二进制模式打开文件。

本函数带有 *dir_fd* 参数，支持*基于目录描述符的相对路径*。

引发一个*审计事件* open 并附带参数 path, mode, flags。

在 3.4 版本发生变更: 新的文件描述符现在是不可继承的。

> **备注**
>
> 本函数适用于底层的 I/O。常规用途请使用内置函数*open()*，该函数的 `read()` 和 `write()` 方法（及其他方法）会返回文件对象。要将文件描述符包装在文件对象中，请使用*fdopen()*。

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

在 3.6 版本发生变更: 接受一个*path-like object*。

以下常量是*open()* 函数 *flags* 参数的选项。可以用按位或运算符 `|` 将它们组合使用。部分常量并非在所有平台上都可用。有关其可用性和用法的说明，请参阅 *open(2)* 手册（Unix 上）或 MSDN （Windows 上）。

os.O_RDONLY

os.O_WRONLY

os.O_RDWR

os.O_APPEND

os.O_CREAT

os.O_EXCL

os.O_TRUNC

上述常量在 Unix 和 Windows 上均可用。

os.O_DSYNC

os.O_RSYNC

os.O_SYNC

os.O_NDELAY

os.O_NONBLOCK

os.O_NOCTTY

os.O_CLOEXEC

这个常数仅在 Unix 系统中可用。

在 3.3 版本发生变更: 增加*O_CLOEXEC* 常量。

os.O_BINARY

os.O_NOINHERIT

os.O_SHORT_LIVED

os.O_TEMPORARY

os.O_RANDOM

os.O_SEQUENTIAL

os.O_TEXT

这个常数仅在 Windows 系统中可用。

os.O_EVTONLY

os.O_FSYNC

os.O_SYMLINK

os.O_NOFOLLOW_ANY

以上常量仅适用于 macOS。

在 3.10 版本发生变更: 加入*O_EVTONLY* 、*O_FSYNC* 、*O_SYMLINK* 和*O_NOFOLLOW_ANY* 常量。

os.O_ASYNC

os.O_DIRECT

os.O_DIRECTORY

os.**O_NOFOLLOW**

os.**O_NOATIME**

os.**O_PATH**

os.**O_TMPFILE**

os.**O_SHLOCK**

os.**O_EXLOCK**

上述常量是扩展常量，如果 C 库未定义它们，则不存在。

在 3.4 版本发生变更: 在支持的系统上增加*O_PATH*。增加*O_TMPFILE*，仅在 Linux Kernel 3.11 或更高版本可用。

os.**openpty**()

打开一对新的伪终端，返回一对文件描述符（主，从），分别为 pty 和 tty。新的文件描述符是不可继承的。对于（稍微）轻量一些的方法，请使用*pty*模块。

Availability: Unix, not WASI.

在 3.4 版本发生变更: 新的文件描述符不再可继承。

os.**pipe**()

创建一个管道，返回一对分别用于读取和写入的文件描述符 (r, w)。新的文件描述符是不可继承的。

Availability: Unix, Windows.

在 3.4 版本发生变更: 新的文件描述符不再可继承。

os.**pipe2**(*flags*, /)

创建带有 *flags* 标志位的管道。可通过对以下一个或多个值进行“或”运算来构造这些 *flags*: *O_NONBLOCK*、*O_CLOEXEC*。返回一对分别用于读取和写入的文件描述符 (r, w)。

Availability: Unix, not WASI.

Added in version 3.3.

os.**posix_fallocate**(*fd*, *offset*, *len*, /)

确保为 *fd* 指向的文件分配了足够的磁盘空间，该空间从偏移量 *offset* 开始，到 *len* 字节为止。

Availability: Unix.

Added in version 3.3.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*, /)

声明即将以特定模式访问数据，使内核可以提前进行优化。数据范围是从 *fd* 所指向文件的 *offset* 开始，持续 *len* 个字节。*advice* 的取值是如下之一: *POSIX_FADV_NORMAL*, *POSIX_FADV_SEQUENTIAL*, *POSIX_FADV_RANDOM*, *POSIX_FADV_NOREUSE*, *POSIX_FADV_WILLNEED* 或*POSIX_FADV_DONTNEED*。

Availability: Unix.

Added in version 3.3.

os.**POSIX_FADV_NORMAL**

os.**POSIX_FADV_SEQUENTIAL**

os.**POSIX_FADV_RANDOM**

os.**POSIX_FADV_NOREUSE**

os.**POSIX_FADV_WILLNEED**

os.**POSIX_FADV_DONTNEED**

用于*posix_fadvise()*的 *advice* 参数的标志位，指定可能使用的访问模式。

Availability: Unix.

Added in version 3.3.

os.**pread**(*fd*, *n*, *offset*, /)

从文件描述符 *fd* 所指向文件的偏移位置 *offset* 开始，读取至多 *n* 个字节，而保持文件偏移量不变。

返回所读取字节的字节串 (bytestring)。如果到达了 *fd* 指向的文件末尾，则返回空字节对象。

Availability: Unix.

Added in version 3.3.

os.**posix_openpt**(*oflag*, /)

打开并返回一个代表主要伪终端设备的文件描述符。

调用 C 标准库函数 `posix_openpt()`。*oflag* 参数用于设置文件状态旗标和文件访问模式，如你所用系统的 `posix_openpt()` 帮助页中所指明的那样。

返回的文件描述符是非不可继承的。如果所在系统上*O_CLOEXEC* 值是可用的，它将被添加到 *oflag*。

Availability: Unix, not WASI.

Added in version 3.13.

os.**preadv**(*fd*, *buffers*, *offset*, *flags=0*, /)

从文件描述符 *fd* 所指向文件的偏移位置 *offset* 开始，将数据读取至可变字节类对象 缓冲区 *buffers* 中，保持文件偏移量不变。将数据依次存放到每个缓冲区中，填满一个后继续存放到序列中的下一个缓冲区，来保存其余数据。

flags 参数可以由零个或多个标志位进行按位或运算来得到：

- *RWF_HIPRI*
- *RWF_NOWAIT*

返回实际读取的字节总数，该总数可以小于所有对象的总容量。

操作系统可能对允许使用的缓冲区数量有限制（使用*sysconf()* 获取 `'SC_IOV_MAX'` 值）。

本方法结合了*os.readv()* 和*os.pread()* 的功能。

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

使用旗标需要 Linux >= 4.6。

Added in version 3.7.

os.**RWF_NOWAIT**

不要等待无法立即获得的数据。如果指定了此标志，那么当需要从后备存储器中读取数据，或等待文件锁时，系统调用将立即返回。

如果成功读取了数据，将返回读取的字节数。如果未读取到数据，将返回 `-1` 并将 errno 设为*errno.EAGAIN*。

Availability: Linux >= 4.14.

Added in version 3.7.

os.**RWF_HIPRI**

高优先级读/写。允许基于块的文件系统对设备进行轮询，这样可以降低延迟，但可能会占用更多资源。

目前在 Linux 上，此功能仅在使用*O_DIRECT* 标志打开的文件描述符上可用。

Availability: Linux >= 4.6.

Added in version 3.7.

os.**ptsname**(*fd*, /)

返回与文件描述符 *fd* 所指向的主伪终端设备相关联的从伪终端设备。文件描述符 *fd* 在失败时不会被关闭。

如果可重入 C 标准库函数 `ptsname_r()` 可用则调用它；在其他情况下，则调用 C 标准库函数 `ptsname()`，该函数不保证线程安全。

Availability: Unix, not WASI.

Added in version 3.13.

os.**pwrite**(*fd*, *str*, *offset*, /)

将 *str* 中的字节串 (bytestring) 写入文件描述符 *fd* 的偏移位置 *offset* 处，保持文件偏移量不变。

返回实际写入的字节数。

Availability: Unix.

Added in version 3.3.

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*, /)

将 *buffers* 内容写入文件描述符 *fd* 的偏移位置 *offset* 处，保持文件偏移位置不变。*buffers* 必须是由字节型对象 组成的序列。缓冲区以数组顺序处理。先写入第一个缓冲区的全部内容再写入第二个，依此类推。

flags 参数可以由零个或多个标志位进行按位或运算来得到：

- *RWF_DSYNC*
- *RWF_SYNC*
- *RWF_APPEND*

返回实际写入的字节总数。

操作系统可能对允许使用的缓冲区数量有限制（使用*sysconf()* 获取 'SC_IOV_MAX' 值）。

本方法结合了*os.writev()* 和*os.pwrite()* 的功能。

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

使用旗标需要 Linux >= 4.6。

Added in version 3.7.

os.**RWF_DSYNC**

提供预写功能，等效于带*O_DSYNC* 标志的*os.open()* 。本标志只作用于通过系统调用写入的数据。

Availability: Linux >= 4.7.

Added in version 3.7.

os.**RWF_SYNC**

提供预写功能，等效于带*O_SYNC* 标志的*os.open()* 。本标志只作用于通过系统调用写入的数据。

Availability: Linux >= 4.7.

Added in version 3.7.

os.**RWF_APPEND**

提供预写功能，等效于带*O_APPEND* 标志的*os.open()* 。本标志只对*os.pwritev()* 有意义，只作用于通过系统调用写入的数据。参数 *offset* 对写入操作无效；数据总是会添加到文件的末尾。但如果 *offset* 参数为 -1，则会刷新当前文件的 *offset* 。

Availability: Linux >= 4.16.

Added in version 3.10.

os.**read**(*fd*, *n*, /)

从文件描述符 *fd* 中读取至多 *n* 个字节。

返回所读取字节的字节串 (bytestring)。如果到达了 *fd* 指向的文件末尾，则返回空字节对象。

> **备注**
>
> 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要读取由内建函数*open()*、*popen()*、*fdopen()* 或*sys.stdin* 返回的" 文件对象"，则应使用其相应的 `read()` 或 `readline()` 方法。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

`os.`**`sendfile`**(*out_fd*, *in_fd*, *offset*, *count*)

`os.`**`sendfile`**(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)

将文件描述符 *in_fd* 中的 *count* 字节复制到文件描述符 *out_fd* 的偏移位置 *offset* 处。返回复制的字节数，如果到达 EOF，返回 `0`。

定义了*sendfile()* 的所有平台均支持第一种函数用法。

在 Linux 上，将 *offset* 设置为 `None`，则从 *in_fd* 的当前位置开始读取，并更新 *in_fd* 的位置。

第二种情况可以被用于 macOS 和 FreeBSD，其中 *headers* 和 *trailers* 是任意的缓冲区序列，它们会在写入来自 *in_fd* 的数据之前被写入。它的返回内容与第一种情况相同。

在 macOS 和 FreeBSD 上，传入 `0` 值作为 *count* 将指定持续发送直至到达 *in_fd* 的末尾。

所有平台都支持将套接字作为 *out_fd* 文件描述符，有些平台也支持其他类型（如常规文件或管道）。

跨平台应用程序不应使用 *headers*、*trailers* 和 *flags* 参数。

Availability: Unix, not WASI.

> **备注**
>
> 有关*sendfile()* 的高级封装，参见*socket.socket.sendfile()*。

Added in version 3.3.

在 3.9 版本发生变更: *out* 和 *in* 参数被重命名为 *out_fd* 和 *in_fd*。

`os.`**`SF_NODISKIO`**

`os.`**`SF_MNOWAIT`**

`os.`**`SF_SYNC`**

sendfile() 函数的参数（假设当前实现支持这些参数）。

Availability: Unix, not WASI.

Added in version 3.3.

`os.`**`SF_NOCACHE`**

传给*sendfile()* 函数的形参，如果具体实现支持的话。数据不会缓存在虚拟内存中并将随即被释放。

Availability: Unix, not WASI.

Added in version 3.11.

`os.`**`set_blocking`**(*fd*, *blocking*, */*)

设置指定文件描述符的阻塞模式：如果 blocking 为 `False`，则为该描述符设置*O_NONBLOCK* 标志位，反之则清除该标志位。

参见*get_blocking()* 和*socket.socket.setblocking()*。

Availability: Unix, Windows.

此函数在 WASI 是受限的，请参阅*WebAssembly 平台* 了解详情。

在 Windows 上，此函数仅限于管道。

Added in version 3.5.

在 3.12 版本发生变更: 增加了在 Windows 上对于管道的支持。

os.**splice**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

从文件描述符 *src* 偏移量 *offset_src* 开始，将 *count* 个字节传输到文件描述符 *dst*。至少有一个文件描述符必须指向一个管道。如果 *offset_src* 为 None，则 *src* 将从当前位置开始读取；相应地 *offset_dst* 也是如此。与指向管道的文件描述符相关的偏移量必须为 None。*src* 和 *dst* 指向的文件必须位于相同文件系统中，否则会引发*OSError* 并将*errno* 设为*errno.EXDEV*。

此复制的完成没有额外的从内核到用户空间再回到内核的数据转移花费。另外，一些文件系统可能实现额外的优化。完成复制就如同打开两个二进制文件一样。

调用成功后，返回拼接到管道的字节数或从管道拼接出来的字节数。返回值为 0 意味着输入结束。如果 *src* 指向一个管道，则意味着没有数据需要传输，而且由于没有写入程序连到管道的写入端，所以将不会阻塞。

Availability: Linux >= 2.6.17 with glibc >= 2.5

Added in version 3.10.

os.**SPLICE_F_MOVE**

os.**SPLICE_F_NONBLOCK**

os.**SPLICE_F_MORE**

Added in version 3.10.

os.**readv**(*fd*, *buffers*, */*)

从文件描述符 *fd* 将数据读取至多个可变的字节类对象 缓冲区 *buffers* 中。将数据依次存放到每个缓冲区中，填满一个后继续存放到序列中的下一个缓冲区，来保存其余数据。

返回实际读取的字节总数，该总数可以小于所有对象的总容量。

操作系统可能对允许使用的缓冲区数量有限制（使用*sysconf()* 获取 'SC_IOV_MAX' 值）。

Availability: Unix.

Added in version 3.3.

os.**tcgetpgrp**(*fd*, */*)

返回与 *fd* 指定的终端相关联的进程组（*fd* 是由*os.open()* 返回的已打开的文件描述符）。

Availability: Unix, not WASI.

os.**tcsetpgrp**(*fd*, *pg*, */*)

设置与 *fd* 指定的终端相关联的进程组为 *pg*（*fd* 是由*os.open()* 返回的已打开的文件描述符）。

Availability: Unix, not WASI.

os.**ttyname**(*fd*, */*)

返回一个字符串，该字符串表示与文件描述符 *fd* 关联的终端。如果 *fd* 没有与终端关联，则抛出异常。

Availability: Unix.

os.**unlockpt**(*fd*, */*)

解锁与文件描述符 *fd* 所指向的主伪终端设备相关联的从伪终端设备。文件描述符 *fd* 在失败时不会被关闭。

调用 C 标准库函数 unlockpt()。

Availability: Unix, not WASI.

Added in version 3.13.

os.**write**(*fd*, *str*, /)

将 *str* 中的字节串 (bytestring) 写入文件描述符 *fd*。

返回实际写入的字节数。

> **备注**
>
> 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要写入由内建函数*open()*、*popen()*、*fdopen()*、*sys.stdout* 或*sys.stderr* 返回的”文件对象”，则应使用其相应的 `write()` 方法。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

os.**writev**(*fd*, *buffers*, /)

将缓冲区 *buffers* 的内容写入文件描述符 *fd*。缓冲区 *buffers* 必须是由*字节类对象* 组成的序列。缓冲区以数组顺序处理。先写入第一个缓冲区的全部内容，再写入第二个缓冲区，照此继续。

返回实际写入的字节总数。

操作系统可能对允许使用的缓冲区数量有限制（使用*sysconf()* 获取 `'SC_IOV_MAX'` 值）。

Availability: Unix.

Added in version 3.3.

查询终端的尺寸

Added in version 3.3.

os.**get_terminal_size**(*fd=STDOUT_FILENO*, /)

返回终端窗口的尺寸，格式为 `(columns, lines)`，它是类型为*terminal_size* 的元组。

可选参数 `fd` （默认为 `STDOUT_FILENO` 或标准输出）指定应查询的文件描述符。

如果文件描述符未连接到终端，则抛出*OSError* 异常。

shutil.get_terminal_size() 是供常规使用的高阶函数，`os.get_terminal_size` 是其底层的实现。

Availability: Unix, Windows.

class os.**terminal_size**

元组的子类，存储终端窗口尺寸 `(columns, lines)`。

columns

终端窗口的宽度，单位为字符。

lines

终端窗口的高度，单位为字符。

文件描述符的继承

Added in version 3.4.

每个文件描述符都有一个”inheritable”（可继承）标志位，该标志位控制了文件描述符是否可以由子进程继承。从 Python 3.4 开始，由 Python 创建的文件描述符默认是不可继承的。

在 UNIX 上，执行新程序时，不可继承的文件描述符在子进程中是关闭的，其他文件描述符将被继承。

在 Windows 上，不可继承的句柄和文件描述符在子进程中是关闭的，但标准流（文件描述符 0、1 和 2 即标准输入、标准输出和标准错误）是始终继承的。如果使用*spawn** 函数，所有可继承的句柄和文件描述符都将被继承。如果使用*subprocess* 模块，将关闭除标准流以外的所有文件描述符，并且仅当 *close_fds* 参数为 `False` 时才继承可继承的句柄。

在 WebAssembly 平台上，文件描述符无法被修改。

os.**get_inheritable**(*fd*, /)

获取指定文件描述符的“可继承”标志位（为布尔值）。

os.**set_inheritable**(*fd*, *inheritable*, /)

设置指定文件描述符的“可继承”标志位。

os.**get_handle_inheritable**(*handle*, /)

获取指定句柄的“可继承”标志位（为布尔值）。

Availability: Windows.

os.**set_handle_inheritable**(*handle*, *inheritable*, /)

设置指定句柄的“可继承”标志位。

Availability: Windows.

16.1.6 文件和目录

在某些 Unix 平台上，许多函数支持以下一项或多项功能：

- **指定文件描述符为参数：**通常在*os* 模块中提供给函数的 *path* 参数必须是表示文件路径的字符串，但是，某些函数现在可以接受其 *path* 参数为打开文件描述符，该函数将对描述符指向的文件进行操作。（对于 POSIX 系统，Python 将调用以 `f` 开头的函数变体（如调用 `fchdir` 而不是 `chdir`）。）

  可以用*os.supports_fd* 检查某个函数在你的平台上是否支持将 *path* 参数指定为文件描述符。如果不支持，使用该功能将抛出*NotImplementedError* 异常。

  如果该函数还支持 *dir_fd* 或 *follow_symlinks* 参数，那么用文件描述符作为 *path* 后就不能再指定上述参数了。

- **基于目录描述符的相对路径：**如果 *dir_fd* 不是 `None`，它就应该是一个指向目录的文件描述符，这时待操作的 path 应该是相对路径，相对路径是相对于前述目录的。如果 path 是绝对路径，则 *dir_fd* 将被忽略。（对于 POSIX 系统，Python 将调用该函数的变体，变体以 `at` 结尾，可能以 `f` 开头（如调用 `faccessat` 而不是 `access`）。

  可以用*os.supports_dir_fd* 检查某个函数在你的平台上是否支持 *dir_fd*。如果不支持，使用该功能将抛出*NotImplementedError* 异常。

- **不跟踪符号链接：**如果 *follow_symlinks* 为 `False`，并且待操作路径的最后一个元素是符号链接，则该函数将在符号链接本身而不是链接所指向的文件上操作。（对于 POSIX 系统，Python 将调用该函数的 `l...` 变体。）

  可以用*os.supports_follow_symlinks* 检查某个函数在你的平台上是否支持 *follow_symlinks*。如果不支持，使用该功能将抛出*NotImplementedError* 异常。

os.**access**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

使用实际用户 ID/用户组 ID 测试对 *path* 的访问。请注意，大多数测试操作将使用有效用户 ID/用户组 ID，因此可以在 suid/sgid 环境中运用此例程，来测试调用用户是否具有对 *path* 的指定访问权限。*mode* 为*F_OK* 时用于测试 *path* 是否存在，也可以对*R_OK*、*W_OK* 和*X_OK* 中的一个或多个进行“或”运算来测试指定权限。允许访问则返回*True*，否则返回*False*。更多信息请参见 Unix 手册页 *access(2)*。

本函数支持指定基于目录描述符的相对路径 和不跟踪符号链接。

如果 *effective_ids* 为 `True`，*access()* 将使用有效用户 ID/用户组 ID 而非实际用户 ID/用户组 ID 进行访问检查。您的平台可能不支持 *effective_ids*，您可以使用*os.supports_effective_ids* 检查它是否可用。如果不可用，使用它时会抛出*NotImplementedError* 异常。

> **备注**

使用*access()* 来检查用户是否具有某项权限（如打开文件的权限），然后再使用*open()* 打开文件，这样做存在一个安全漏洞，因为用户可能会在检查和打开文件之间的时间里做其他操作。推荐使用*EAFP* 技术。如:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

最好写成:

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

备注

即使*access()* 指示 I/O 操作会成功，但实际操作仍可能失败，尤其是对网络文件系统的操作，其权限语义可能超出常规的 POSIX 权限位模型。

在 3.3 版本发生变更: 添加 *dir_fd*、*effective_ids* 和 *follow_symlinks* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**F_OK**

os.**R_OK**

os.**W_OK**

os.**X_OK**

作为*access()* 的 *mode* 参数的可选值，分别测试 *path* 的存在性、可读性、可写性和可执行性。

os.**chdir**(*path*)

将当前工作目录更改为 *path*。

本函数支持指定文件描述符为参数。其中，描述符必须指向打开的目录，不能是打开的文件。

本函数可以抛出*OSError* 及其子类的异常，如*FileNotFoundError*、*PermissionError* 和*NotADirectoryError* 异常。

引发一个审计事件 os.chdir 并附带参数 path。

在 3.3 版本发生变更: 在某些平台上新增支持将 *path* 参数指定为文件描述符。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**chflags**(*path*, *flags*, *, *follow_symlinks=True*)

将 *path* 的 flags 设置为其他由数字表示的 *flags*。*flags* 可以用以下值按位或组合起来（以下值在*stat* 模块中定义）:

- *stat.UF_NODUMP*
- *stat.UF_IMMUTABLE*
- *stat.UF_APPEND*
- *stat.UF_OPAQUE*
- *stat.UF_NOUNLINK*
- *stat.UF_COMPRESSED*
- *stat.UF_HIDDEN*

- *stat.SF_ARCHIVED*
- *stat.SF_IMMUTABLE*
- *stat.SF_APPEND*
- *stat.SF_NOUNLINK*
- *stat.SF_SNAPSHOT*

本函数支持不跟踪符号链接。

引发一个审计事件 `os.chflags` 并附带参数 `path`, `flags`。

Availability: Unix, not WASI.

在 3.3 版本发生变更: 增加了 *follow_symlinks* 形参。

在 3.6 版本发生变更: 接受一个*path-like object*。

`os.`**`chmod`**(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

将 *path* 的 mode 更改为其他由数字表示的 *mode*。*mode* 可以用以下值之一，也可以将它们按位或组合起来（以下值在*stat* 模块中定义）：

- *stat.S_ISUID*
- *stat.S_ISGID*
- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

本函数支持指定文件描述符、指定基于目录描述符的相对路径 和不跟踪符号链接。

备注

尽管 Windows 支持*chmod()*，但你只能用它设置文件的只读旗标（通过 `stat.S_IWRITE` 和 `stat.S_IREAD` 常量或对应的整数值）。所有其他旗标位都会被忽略。在 Windows 上 *follow_symlinks* 的默认值为 `False`。

此函数在 WASI 是受限的，请参阅*WebAssembly 平台* 了解详情。

引发一个审计事件 `os.chmod` 并附带参数 `path`, `mode`, `dir_fd`。

在 3.3 版本发生变更: 添加了指定 *path* 为文件描述符的支持，以及 *dir_fd* 和 *follow_symlinks* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.13 版本发生变更: 在 Windows 上增加了对文件描述符和 *follow_symlinks* 参数的支持。

`os.`**`chown`**(*path*, *uid*, *gid*, ***, *dir_fd=None*, *follow_symlinks=True*)

将 *path* 的用户和组 ID 分别修改为数字形式的 *uid* 和 *gid*。若要使其中某个 ID 保持不变，请将其置为 -1。

本函数支持指定文件描述符、指定基于目录描述符的相对路径 和不跟踪符号链接。

参见更高阶的函数*shutil.chown()*，除了数字 ID 之外，它也接受名称。

引发一个审计事件 `os.chown` 并附带参数 `path`, `uid`, `gid`, `dir_fd`。

Availability: Unix.

此函数在 WASI 是受限的，请参阅*WebAssembly 平台* 了解详情。

在 3.3 版本发生变更: 添加了指定 *path* 为文件描述符的支持，以及 *dir_fd* 和 *follow_symlinks* 参数。

在 3.6 版本发生变更: 支持类路径对象。

`os.`**`chroot`**(*path*)

将当前进程的根目录更改为 *path*。

Availability: Unix, not WASI, not Android.

在 3.6 版本发生变更: 接受一个*path-like object*。

`os.`**`fchdir`**(*fd*)

将当前工作目录更改为文件描述符 *fd* 指向的目录。fd 必须指向打开的目录而非文件。从 Python 3.3 开始，它等效于 `os.chdir(fd)`。

引发一个审计事件 `os.chdir` 并附带参数 `path`。

Availability: Unix.

`os.`**`getcwd`**()

返回表示当前工作目录的字符串。

`os.`**`getcwdb`**()

返回表示当前工作目录的字节串 (bytestring)。

在 3.8 版本发生变更: 在 Windows 上，本函数现在会使用 UTF-8 编码格式而不是 ANSI 代码页：请参看 **PEP 529** 了解具体原因。该函数在 Windows 上不再被弃用。

`os.`**`lchflags`**(*path*, *flags*)

将 *path* 的 flags 设置为其他由数字表示的 *flags*，与*chflags()* 类似，但不跟踪符号链接。从 Python 3.3 开始，它等效于 `os.chflags(path, flags, follow_symlinks=False)`。

引发一个审计事件 `os.chflags` 并附带参数 `path`, `flags`。

Availability: Unix, not WASI.

在 3.6 版本发生变更: 接受一个*path-like object*。

`os.`**`lchmod`**(*path*, *mode*)

将 *path* 的权限状态修改为 *mode*。如果 path 是符号链接，则影响符号链接本身而非链接目标。可以参考*chmod()* 中列出 *mode* 的可用值。从 Python 3.3 开始，它等效于 `os.chmod(path, mode, follow_symlinks=False)`。

`lchmod()` 不是 POSIX 的一部分，但 Unix 实现如果支持修改符号链接的模式则可能包含它。

引发一个审计事件 `os.chmod` 并附带参数 `path`, `mode`, `dir_fd`。

Availability: Unix, Windows, not Linux, FreeBSD >= 1.3, NetBSD >= 1.3, not OpenBSD

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.13 版本发生变更: 增加 Windows 上的支持。

os.**lchown**(*path*, *uid*, *gid*)

将 *path* 的用户和组 ID 分别修改为数字形式的 *uid* 和 *gid*，本函数不跟踪符号链接。从 Python 3.3 开始，它等效于 `os.chown(path, uid, gid, follow_symlinks=False)`。

引发一个审计事件 `os.chown` 并附带参数 `path`, `uid`, `gid`, `dir_fd`。

Availability: Unix.

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**link**(*src*, *dst*, **, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True*)

创建一个指向 *src* 的硬链接，名为 *dst*。

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径，支持不跟踪符号链接。

引发一个审计事件 `os.link` 并附带参数 `src`、`dst`、`src_dir_fd`、`dst_dir_fd`。

Availability: Unix, Windows.

在 3.2 版本发生变更: 添加了对 Windows 的支持。

在 3.3 版本发生变更: 增加了 *src_dir_fd*, *dst_dir_fd* 和 *follow_symlinks* 形参。

在 3.6 版本发生变更: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**listdir**(*path='.'*)

返回一个包含由 *path* 指定目录中条目名称组成的列表。该列表按任意顺序排列，并且不包括特殊条目 `'.'` 和 `'..'`，即使它们存在于目录中。如果有文件在调用此函数期间在被移除或添加到目录中，是否要包括该文件的名称并没有规定。

path 可以是类路径对象。如果 *path* 是（直接传入或通过*PathLike* 接口间接传入）`bytes` 类型，则返回的文件名也将是 `bytes` 类型，其他情况下是 `str` 类型。

本函数也支持指定文件描述符为参数，其中描述符必须指向目录。

引发一个审计事件 `os.listdir` 并附带参数 `path`。

备注

要将 `str` 类型的文件名编码为 `bytes`，请使用*fsencode()*。

参见

scandir() 函数返回目录内文件名的同时，也返回文件属性信息，它在某些具体情况下能提供更好的性能。

在 3.2 版本发生变更: *path* 变为可选参数。

在 3.3 版本发生变更: 新增支持将 *path* 参数指定为打开的文件描述符。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**listdrives**()

返回一个包括 Windows 系统上驱动名称的列表。

驱动器名称通常的形式如 `'C:\\'`。并非每个驱动器名都会关联到特定的卷，有些驱动器名可能出于各种原因而无法访问，包括权限、网络连接或介质丢失等。本函数不会测试可访问性。

如果在收集驱动器名时发生错误则可能引发*OSError*。

引发一个不带参数的审计事件 `os.listdrives`。

Availability: Windows

Added in version 3.12.

os.**listmounts**(*volume*)

返回一个包含 Windows 系统上指向卷的加载点的列表。

volume 必须表示为 GUID 路径，如*os.listvolumes()* 所返回的值。卷可能被挂载到多个位置也可能根本未挂载。在后一种情况下，该列表将为空。此函数不会返回没有关联到卷的挂载点。

此函数返回的挂载点将为绝对路径，并可能比驱动器名称更长。

如果卷未被识别或者如果在获取路径时发生错误则会引发*OSError*。

引发一个审计事件 `os.listmounts` 并附带参数 `volume`。

Availability: Windows

Added in version 3.12.

os.**listvolumes**()

返回一个包含系统中的卷的列表。

卷通常被表示为一个 GUID 路径如 `\\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}\`。文件通常可通过 GUID 路径来访问，如果权限允许的话。但是，用户往往并不熟悉这种路径，所以此函数的推荐用法是使用*os.listmounts()* 来获取加载点。

如果在收集卷时发生错误则可能引发*OSError*。

引发一个不带参数的审计事件 `os.listvolumes`。

Availability: Windows

Added in version 3.12.

os.**lstat**(*path*, *, *dir_fd=None*)

在给定的路径上执行 `lstat()` 系统调用的等价物。类似于*stat()*，但不会跟随符号链接。返回一个*stat_result* 对象。

在不支持符号链接的平台上，本函数是*stat()* 的别名。

从 Python 3.3 起，此功能等价于 `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`。

本函数支持基于目录描述符的相对路径。

参见

stat() 函数。

在 3.2 版本发生变更: 添加对 Windows 6.0 (Vista) 符号链接的支持。

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.8 版本发生变更: 目前在 Windows 上，遇到表示另一个路径的重解析点（即名称代理，包括符号链接和目录结点），本函数将打开它。其他种类的重解析点由*stat()* 交由操作系统解析。

os.**mkdir**(*path*, *mode=0o777*, *, *dir_fd=None*)

创建一个名为 *path* 的目录，应用以数字表示的权限模式 *mode*。

如果目录已经存在，*FileExistsError* 会被提出。如果路径中的父目录不存在，则会引发*FileNotFoundError* 。

某些系统会忽略 *mode*。如果没有忽略它，那么将首先从它中减去当前的 umask 值。如果除最后 9 位（即 *mode* 八进制的最后 3 位）之外，还设置了其他位，则其他位的含义取决于各个平台。在某些平台上，它们会被忽略，应显式调用*chmod()* 进行设置。

在 Windows 系统中，*mode* 值 `0o700` 被专门用于对新目录实施访问控制，使其只有当前用户和管理员才能访问。其他值的 *mode* 将被忽略。

本函数支持基于目录描述符的相对路径。

如果需要创建临时目录，请参阅 *tempfile* 模块中的 *tempfile.mkdtemp()* 函数。

引发一个审计事件 `os.mkdir` 并附带参数 `path`, `mode`, `dir_fd`。

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 接受一个 *path-like object*。

在 3.13 版本发生变更: Windows 系统现在使用 *mode* 值 “0o700“

`os.`**`makedirs`**(*name*, *mode=0o777*, *exist_ok=False*)

递归目录创建函数。与 *mkdir()* 类似，但会自动创建到达最后一级目录所需要的中间目录。

mode 形参会被传递给 *mkdir()* 用来创建最后一级目录；请参阅 *mkdir() 的说明* 了解其解读方式。要设置任何新建父目录的权限你可以在唤起 *makedirs()* 之前设置掩码。现有父目录的文件权限不会被更改。

如果 *exist_ok* 为 `False` (默认值)，则如果目标目录已存在将会引发 *FileExistsError*。

> **备注**
>
> 如果要创建的路径元素包含 *pardir* (如 UNIX 系统中的”..”) *makedirs()* 将无法明确目标。

本函数能正确处理 UNC 路径。

引发一个审计事件 `os.mkdir` 并附带参数 `path`, `mode`, `dir_fd`。

在 3.2 版本发生变更: 增加了 *exist_ok* 形参。

在 3.4.1 版本发生变更: 在 Python 3.4.1 以前，如果 *exist_ok* 为 `True`，且目录已存在，且 *mode* 与现有目录的权限不匹配，*makedirs()* 仍会抛出错误。由于无法安全地实现此行为，因此在 Python 3.4.1 中将该行为删除。请参阅 bpo-21082。

在 3.6 版本发生变更: 接受一个 *path-like object*。

在 3.7 版本发生变更: *mode* 参数不会再影响新创建的中间目录的文件权限位。

`os.`**`mkfifo`**(*path*, *mode=0o666*, ***, *dir_fd=None*)

创建一个名为 *path* 的 FIFO（命名管道，一种先进先出队列），具有以数字表示的权限状态 *mode*。将从 mode 中首先减去当前的 umask 值。

本函数支持基于目录描述符的相对路径。

FIFO 是可以像常规文件一样访问的管道。FIFO 如果没有被删除（如使用 *os.unlink()*），会一直存在。通常，FIFO 用作“客户端”和“服务器”进程之间的汇合点：服务器打开 FIFO 进行读取，而客户端打开 FIFO 进行写入。请注意，*mkfifo()* 不会打开 FIFO --- 它只是创建汇合点。

Availability: Unix, not WASI.

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 接受一个 *path-like object*。

`os.`**`mknod`**(*path*, *mode=0o600*, *device=0*, ***, *dir_fd=None*)

创建一个名为 *path* 的文件系统节点（文件，设备专用文件或命名管道）。*mode* 指定权限和节点类型，方法是将权限与下列节点类型 `stat.S_IFREG`、`stat.S_IFCHR`、`stat.S_IFBLK` 和 `stat.S_IFIFO` 之一（按位或）组合（这些常量可以在 *stat* 模块中找到）。对于 `stat.S_IFCHR` 和 `stat.S_IFBLK`，*device* 参数指定了新创建的设备专用文件（可能会用到 *os.makedev()*），否则该参数将被忽略。

本函数支持基于目录描述符的相对路径。

Availability: Unix, not WASI.

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**major**(*device*, /)

根据原始设备编号提取设备主编号（通常为来自 stat 的 st_dev 或 st_rdev 字段）。

os.**minor**(*device*, /)

根据原始设备编号提取设备次编号（通常为来自 stat 的 st_dev 或 st_rdev 字段）。

os.**makedev**(*major*, *minor*, /)

将主设备号和次设备号组合成原始设备号。

os.**pathconf**(*path*, *name*)

返回所给名称的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 pathconf_names 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 pathconf_names，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

本函数支持指定文件描述符为参数。

Availability: Unix.

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**pathconf_names**

字典，表示映射关系，为*pathconf()* 和*fpathconf()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

os.**readlink**(*path*, *, *dir_fd=None*)

返回一个字符串，为符号链接指向的实际路径。其结果可以是绝对或相对路径。如果是相对路径，则可用 os.path.join(os.path.dirname(path), result) 转换为绝对路径。

如果 *path* 是字符串对象（直接传入或通过*PathLike* 接口间接传入），则结果也将是字符串对象，且此类调用可能会引发 UnicodeDecodeError。如果 *path* 是字节对象（直接传入或间接传入），则结果将会是字节对象。

本函数支持基于目录描述符的相对路径。

当尝试解析的路径可能含有链接时，请改用*realpath()* 以正确处理递归和平台差异。

Availability: Unix, Windows.

在 3.2 版本发生变更: 添加对 Windows 6.0 (Vista) 符号链接的支持。

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 在 Unix 上可以接受一个类路径对象。

在 3.8 版本发生变更: 在 Windows 上接受类路径对象 和字节对象。

增加了对目录链接的支持，且返回值改为了 “替换路径” 的形式（通常带有 \\?\ 前缀），而不是先前那样返回可选的”print name” 字段。

os.**remove**(*path*, *, *dir_fd=None*)

移除（删除）文件 *path*。如果 *path* 是目录，则会引发*OSError*。请使用*rmdir()* 来移除目录。如果文件不存在，则会引发*FileNotFoundError*。

本函数支持基于目录描述符的相对路径。

在 Windows 上，尝试删除正在使用的文件会抛出异常。而在 Unix 上，虽然该文件的条目会被删除，但分配给文件的存储空间仍然不可用，直到原始文件不再使用为止。

本函数在语义上与*unlink()* 相同。

引发一个审计事件 os.remove 并附带参数 path, dir_fd。

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**removedirs**(*name*)

递归删除目录。工作方式类似于*rmdir()*，不同之处在于，如果成功删除了末尾一级目录，*removedirs()* 会尝试依次删除 *path* 中提到的每个父目录，直到抛出错误为止（但该错误会被忽略，因为这通常表示父目录不是空目录）。例如，os.removedirs('foo/bar/baz') 将首先删除目录 'foo/bar/baz'，然后如果 'foo/bar' 和 'foo' 为空，则继续删除它们。如果无法成功删除末尾一级目录，则抛出*OSError* 异常。

引发一个审计事件 os.remove 并附带参数 path, dir_fd。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

将文件或目录 *src* 重命名为 *dst*。如果 *dst* 已存在，则下列情况下将会操作失败，并抛出*OSError* 的子类:

在 Windows 上，如果 *dst* 存在则总是会引发*FileExistsError*。如果 *src* 和 *dst* 是在不同的文件系统上则此操作可能会失败。请使用*shutil.move()* 以支持移动到不同的文件系统。

在 Unix 上，如果 *src* 是文件而 *dst* 是目录，将抛出*IsADirectoryError* 异常，反之则抛出*NotADirectoryError* 异常。如果两者都是目录且 *dst* 为空，则 *dst* 将被静默替换。如果 *dst* 是非空目录，则抛出*OSError* 异常。如果两者都是文件，则在用户具有权限的情况下，将对 *dst* 进行静默替换。如果 *src* 和 *dst* 在不同的文件系统上，则本操作在某些 Unix 分支上可能会失败。如果成功，重命名操作将是一个原子操作（这是 POSIX 的要求）。

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径。

如果需要在不同平台上都能替换目标，请使用*replace()*。

引发一个审计事件 os.rename 并附带参数 src, dst, src_dir_fd, dst_dir_fd。

在 3.3 版本发生变更: 增加了 *src_dir_fd* 和 *dst_dir_fd* 形参。

在 3.6 版本发生变更: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**renames**(*old*, *new*)

递归重命名目录或文件。工作方式类似*rename()*，除了会首先创建新路径所需的中间目录。重命名后，将调用*removedirs()* 删除旧路径中不需要的目录。

> **备注**
>
> 如果用户没有权限删除末级的目录或文件，则本函数可能会无法建立新的目录结构。

引发一个审计事件 os.rename 并附带参数 src, dst, src_dir_fd, dst_dir_fd。

在 3.6 版本发生变更: 接受一个类路径对象 作为 *old* 和 *new*。

os.**replace**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

将文件或目录 *src* 重命名为 *dst*。如果 *dst* 是非空目录，将抛出*OSError* 异常。如果 *dst* 已存在且为文件，则在用户具有权限的情况下，将对其进行静默替换。如果 *src* 和 *dst* 在不同的文件系统上，本操作可能会失败。如果成功，重命名操作将是一个原子操作（这是 POSIX 的要求）。

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径。

引发一个审计事件 os.rename 并附带参数 src, dst, src_dir_fd, dst_dir_fd。

Added in version 3.3.

在 3.6 版本发生变更: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**rmdir**(*path*, *, *dir_fd=None*)

移除（删除）目录 *path*。如果目录不存在或不为空，则会分别引发*FileNotFoundError* 或*OSError*。要移除整个目录树，可以使用*shutil.rmtree()*。

本函数支持基于目录描述符的相对路径。

引发一个审计事件 os.rmdir 并附带参数 path, dir_fd。

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**scandir**(*path='.'*)

返回一个*os.DirEntry* 对象的迭代器，它们对应于由 *path* 指定目录中的条目。这些条目会以任意顺序生成，并且不包括特殊条目 '.' 和 '..'。如果有文件在迭代器创建之后在目录中被移除或添加，是否要包括该文件对应的条目并没有规定。

如果需要文件类型或文件属性信息，使用*scandir()* 代替*listdir()* 可以大大提高这部分代码的性能，因为如果操作系统在扫描目录时返回的是*os.DirEntry* 对象，则该对象包含了这些信息。所有*os.DirEntry* 的方法都可能执行一次系统调用，但是*is_dir()* 和*is_file()* 通常只在有符号链接时才执行一次系统调用。*os.DirEntry.stat()* 在 Unix 上始终需要一次系统调用，而在 Windows 上只在有符号链接时才需要。

path 可以是类路径对象。如果 *path* 是（直接传入或通过*PathLike* 接口间接传入的）bytes 类型，那么每个*os.DirEntry* 的*name* 和*path* 属性将是 bytes 类型，其他情况下是 str 类型。

本函数也支持指定文件描述符为参数，其中描述符必须指向目录。

引发一个审计事件 os.scandir 并附带参数 path。

scandir() 迭代器支持上下文管理 协议，并具有以下方法:

scandir.**close**()

关闭迭代器并释放占用的资源。

当迭代器迭代完毕，或垃圾回收，或迭代过程出错时，将自动调用本方法。但仍建议显式调用它或使用 with 语句。

Added in version 3.6.

下面的例子演示了*scandir()* 的简单用法，用来显示给定 *path* 中所有不以 '.' 开头的文件（不包括目录）。entry.is_file() 通常不会增加一次额外的系统调用:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

备注

在基于 Unix 的系统上，*scandir()* 使用系统的 opendir() 和 readdir() 函数。在 Windows 上，它使用 Win32 FindFirstFileW 和 FindNextFileW 函数。

Added in version 3.5.

在 3.6 版本发生变更: 增加了对*context manager* 协议和*close()* 方法的支持。如果*scandir()* 迭代器没有耗尽也没有被显式地关闭则将在其析构器中发出*ResourceWarning*。

本函数接受一个类路径对象。

在 3.7 版本发生变更: 在 Unix 上新增支持指定文件描述符为参数。

class os.**DirEntry**

由*scandir()* 生成的对象，用于显示目录内某个条目的文件路径和其他文件属性。

scandir() 将在不进行额外系统调用的情况下，提供尽可能多的此类信息。每次进行 stat() 或 lstat() 系统调用时，os.DirEntry 对象会将结果缓存下来。

os.DirEntry 实例不适合存储在长期存在的数据结构中，如果你知道文件元数据已更改，或者自调用*scandir()* 以来已经经过了很长时间，请调用 os.stat(entry.path) 来获取最新信息。

因为 os.DirEntry 方法可以进行系统调用，所以它也可能抛出*OSError* 异常。如需精确定位错误，可以逐个调用 os.DirEntry 中的方法来捕获*OSError*，并适当处理。

为了能直接用作类路径对象，os.DirEntry 实现了*PathLike* 接口。

os.DirEntry 实例所包含的属性和方法如下：

name

本条目的基本文件名，是根据*scandir()* 的 *path* 参数得出的相对路径。

如果*scandir()* 的 *path* 参数是 bytes 类型，则*name* 属性也是 bytes 类型，否则为 str。使用*fsdecode()* 解码 byte 类型的文件名。

path

本条目的完整路径：等效于 os.path.join(scandir_path, entry.name)，其中 *scandir_path* 就是*scandir()* 的 *path* 参数。仅当*scandir()* 的 *path* 参数为绝对路径时，本路径才是绝对路径。如果*scandir()* 的 *path* 参数是文件描述符，则*path* 属性与上述*name* 属性相同。

如果*scandir()* 的 *path* 参数是 bytes 类型，则*path* 属性也是 bytes 类型，否则为 str。使用*fsdecode()* 解码 byte 类型的文件名。

inode()

返回本条目的索引节点号 (inode number)。

这一结果是缓存在 os.DirEntry 对象中的，请调用 os.stat(entry.path, follow_symlinks=False).st_ino 来获取最新信息。

一开始没有缓存时，在 Windows 上需要一次系统调用，但在 Unix 上不需要。

is_dir(*, *follow_symlinks=True*)

如果本条目是目录，或是指向目录的符号链接，则返回 True。如果本条目是文件，或指向任何其他类型的文件，或该目录不再存在，则返回 False。

如果 *follow_symlinks* 是 False，那么仅当本条目为目录时返回 True（不跟踪符号链接），如果本条目是任何类型的文件，或该文件不再存在，则返回 False。

这一结果是缓存在 os.DirEntry 对象中的，且 *follow_symlinks* 为 True 和 False 时的缓存是分开的。请调用*os.stat()* 和*stat.S_ISDIR()* 来获取最新信息。

一开始没有缓存时，大多数情况下不需要系统调用。特别是对于非符号链接，Windows 和 Unix 都不需要系统调用，除非某些 Unix 文件系统（如网络文件系统）返回了 dirent.d_type == DT_UNKNOWN。如果本条目是符号链接，则需要一次系统调用来跟踪它（除非 *follow_symlinks* 为 False）。

本方法可能抛出*OSError* 异常，如*PermissionError* 异常，但*FileNotFoundError* 异常会被内部捕获且不会抛出。

is_file(*, *follow_symlinks=True*)

如果本条目是文件，或是指向文件的符号链接，则返回 True。如果本条目是目录，或指向目录，或指向其他非文件条目，或该文件不再存在，则返回 False。

如果 *follow_symlinks* 是 False，那么仅当本条目为文件时返回 True（不跟踪符号链接），如果本条目是目录或其他非文件条目，或该文件不再存在，则返回 False。

这一结果是缓存在 os.DirEntry 对象中的。缓存、系统调用、异常抛出都与*is_dir()* 一致。

is_symlink()

如果本条目是符号链接（即使是断开的链接），返回 `True`。如果是目录或任何类型的文件，或本条目不再存在，返回 `False`。

这一结果是缓存在 `os.DirEntry` 对象中的，请调用*os.path.islink()* 来获取最新信息。

一开始没有缓存时，大多数情况下不需要系统调用。其实 Windows 和 Unix 都不需要系统调用，除非某些 Unix 文件系统（如网络文件系统）返回了 `dirent.d_type == DT_UNKNOWN`。

本方法可能抛出*OSError* 异常，如*PermissionError* 异常，但*FileNotFoundError* 异常会被内部捕获且不会抛出。

is_junction()

如果本条目是接合点（即使已断开）则返回 `True`；如果条目指向常规目录、任何种类的文件、符号链接或者已不存在则返回 `False`。

结果是缓存在 `os.DirEntry` 对象中的。调用*os.path.isjunction()* 来获取更新信息。

Added in version 3.12.

stat(*, *follow_symlinks=True*)

返回本条目对应的*stat_result* 对象。本方法默认会跟踪符号链接，要获取符号链接本身的 stat，请添加 `follow_symlinks=False` 参数。

在 Unix 上，本方法需要一次系统调用。在 Windows 上，仅在 *follow_symlinks* 为 `True` 且该条目是一个重解析点（如符号链接或目录结点）时，才需要一次系统调用。

在 Windows 上，*stat_result* 的 `st_ino`、`st_dev` 和 `st_nlink` 属性总是为零。请调用*os.stat()* 以获得这些属性。

这一结果是缓存在 `os.DirEntry` 对象中的，且 *follow_symlinks* 为 `True` 和 `False` 时的缓存是分开的。请调用*os.stat()* 来获取最新信息。

请注意 `os.DirEntry` 和*pathlib.Path* 的几个属性和方法之间存在很好的对应关系。具体来说，`name` 属性具有相同的含义，`is_dir()`, `is_file()`, `is_symlink()`, `is_junction()` 和 `stat()` 方法也是如此。

Added in version 3.5.

在 3.6 版本发生变更: 添加了对*PathLike* 接口的支持。在 Windows 上添加了对*bytes* 类型路径的支持。

在 3.12 版本发生变更: 统计结果的 `st_ctime` 属性在 Windows 上已被弃用。文件创建时间可通过 `st_birthtime` 来访问，在未来 `st_ctime` 可能会改为返回零或元数据的修改时间，如果可用的话。

`os.`**stat**(*path*, *, *dir_fd=None*, *follow_symlinks=True*)

获取文件或文件描述符的状态。在所给路径上执行等效于 `stat()` 系统调用的操作。*path* 可以是字符串类型，或（直接传入或通过*PathLike* 接口间接传入的）bytes 类型，或打开的文件描述符。返回一个*stat_result* 对象。

本函数默认会跟踪符号链接，要获取符号链接本身的 stat，请添加 `follow_symlinks=False` 参数，或使用*lstat()*。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

在 Windows 上，传入 `follow_symlinks=False` 将禁用所有名称代理重解析点，其中包括符号链接和目录结点。其他类型的重解析点将直接打开，比如不像链接的或系统无法跟踪的重解析点。当多个链接形成一个链时，本方法可能会返回原始链接的 stat，无法完整遍历到非链接的对象。在这种情况下，要获取最终路径的 stat，请使用*os.path.realpath()* 函数尽可能地解析路径，并在解析结果上调用*lstat()*。这不适用于空链接或交接点，否则会抛出异常。

示例:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
```

(续下页)

(接上页)

```
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

参见

fstat() 和*lstat()* 函数。

在 3.3 版本发生变更: 增加了 *dir_fd* 和 *follow_symlinks* 形参，用于指定一个文件描述符而不是路径。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.8 版本发生变更: 在 Windows 上，本方法将跟踪系统能解析的所有重解析点，并且传入 `follow_symlinks=False` 会停止跟踪所有名称代理重解析点。现在，如果操作系统遇到无法跟踪的重解析点，*stat* 将返回原始路径的信息，就像已指定 `follow_symlinks=False` 一样，而不会抛出异常。

class os.**stat_result**

对象的属性大致对应于 `stat` 结构体的成员。它将被用作*os.stat()*, *os.fstat()* 和*os.lstat()* 的输出结果。

属性:

st_mode

文件模式：包括文件类型和文件模式位（即权限位）。

st_ino

与平台有关，但如果不为零，则根据 `st_dev` 值唯一地标识文件。通常:

- 在 Unix 上该值表示索引节点号 (inode number)。
- 在 Windows 上该值表示 文件索引号 。

st_dev

该文件所在设备的标识符。

st_nlink

硬链接的数量。

st_uid

文件所有者的用户 ID。

st_gid

文件所有者的用户组 ID。

st_size

文件大小（以字节为单位），文件可以是常规文件或符号链接。符号链接的大小是它包含的路径的长度，不包括末尾的空字节。

时间戳:

st_atime

最近的访问时间，以秒为单位。

st_mtime

最近的修改时间，以秒为单位。

st_ctime

以秒数表示的元数据最近更改的时间。

在 3.12 版本发生变更: `st_ctime` 在 Windows 上已被弃用。请使用 `st_birthtime` 获取文件创建时间。在未来，`st_ctime` 将包含最近的元数据修改时间，与其他平台一样。

st_atime_ns

最近的访问时间，以纳秒表示，为整数。

Added in version 3.3.

st_mtime_ns

最近的修改时间，以纳秒表示，为整数。

Added in version 3.3.

st_ctime_ns

最近的元数据修改时间，表示为一个以纳秒为单位的整数。

Added in version 3.3.

在 3.12 版本发生变更: `st_ctime_ns` 在 Windows 上已被弃用。请使用 `st_birthtime_ns` 获取文件创建时间。在未来，`st_ctime` 将包含最近的元数据修改时间，与其他平台一样。

st_birthtime

以秒为单位的文件创建时间。该属性并不总是可用的，并可能引发*AttributeError*。

在 3.12 版本发生变更: 目前 `st_birthtime` 已在 Windows 上可用。

st_birthtime_ns

表示为一个以纳秒为单位的整数的文件创建时间。该属性并不总是可用，并可能引发*AttributeError*。

Added in version 3.12.

备注

st_atime, *st_mtime*, *st_ctime* 和*st_birthtime* 等属性的确切含义和精度依赖于操作系统和文件系统。例如，在使用 FAT32 文件系统的 Windows 系统上，*st_mtime* 的精度为 2 秒，而*st_atime* 的精度只有 1 天。请参阅你的操作系统文档了解详情。

类似地，尽管*st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* 和*st_birthtime_ns* 始终以纳秒表示，但许多系统并不提供纳秒级精度。在确实提供纳秒级精度的系统上，用于存储*st_atime*, *st_mtime*, *st_ctime* 和*st_birthtime* 的浮点数对象无法保留所有精度，因此不是完全准确的。如果你需要准确的时间戳你应始终使用*st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* 和*st_birthtime_ns*。

在某些 Unix 系统上（如 Linux 上），以下属性可能也可用：

st_blocks

为文件分配的字节块数，每块 512 字节。文件是稀疏文件时，它可能小于*st_size*/512。

st_blksize

“首选的”块大小，用于提高文件系统 I/O 效率。写入文件时块大小太小可能会导致读取-修改-重写效率低下。

st_rdev

设备类型（如果是 inode 设备）。

st_flags

用户定义的文件标志位。

在其他 Unix 系统上（如 FreeBSD 上），以下属性可能可用（但仅当 root 使用它们时才被填充）：

st_gen

文件生成号。

在 Solaris 及其衍生版本上，以下属性可能也可用：

st_fstype

文件所在文件系统的类型的唯一标识，为字符串。

在 macOS 系统上，以下属性可能也可用：

st_rsize

文件的实际大小。

st_creator

文件的创建者。

st_type

文件类型。

在 Windows 系统上，以下属性也可用：

st_file_attributes

Windows 文件属性：由 `GetFileInformationByHandle()` 返回的 `BY_HANDLE_FILE_INFORMATION` 结构体的 `dwFileAttributes` 成员。参见*stat* 模块中的 `FILE_ATTRIBUTE_* <stat.FILE_ATTRIBUTE_ARCHIVE>` 常量。

Added in version 3.5.

st_reparse_tag

当*st_file_attributes* 存在*FILE_ATTRIBUTE_REPARSE_POINT* 集合时，本字段将包含标识重解析点的类型的标签。请参阅*stat* 模块中的*IO_REPARSE_TAG_** 常量。

标准模块*stat* 定义了一些可用于从 `stat` 结构体中提取信息的函数和常量。（在 Windows 上，某些项填充了虚拟值。）

为了向下兼容，*stat_result* 实例还可以作为至少包含 10 个整数的元组来访问以提供 `stat` 结构体中最重要（且可移植）的成员，其顺序为*st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*。某些实现还可能在末尾添加更多条目。为了与旧版 Python 兼容，以元组形式访问*stat_result* 将始终返回整数。

在 3.5 版本发生变更: 在 Windows 上，如果可用，会返回文件索引作为*st_ino* 的值。

在 3.7 版本发生变更: 在 Solaris 及其衍生版本上添加了*st_fstype* 成员。

在 3.8 版本发生变更: 在 Windows 上添加了*st_reparse_tag* 成员。

在 3.8 版本发生变更: 在 Windows 上，*st_mode* 成员现在可以根据需要将特殊文件标识为 `S_IFCHR`、`S_IFIFO` 或 `S_IFBLK`。

在 3.12 版本发生变更: 在 Windows 上，*st_ctime* 已被弃用。最终，它将包含元数据的最后修改时间，以与其他平台保持一致，但目前仍包含创建时间。请使用*st_birthtime* 来获取创建时间。

在 Windows 上，现在*st_ino* 最多可为 128 比特位，具体取决于文件系统。在之前它不会超过 64 比特位，更长的文件标识符会被强制缩减。

在 Windows 上，*st_rdev* 将不再返回值。在之前它将包含与*st_dev* 相同的值，这是不正确的。

在 Windows 上增加了*st_birthtime* 成员。

`os.`**`statvfs`**(*path*)

在给定的路径上执行 `statvfs()` 系统调用。返回值是一个对象，其属性描述了所给路径上的文件系统，并且与 `statvfs` 结构体的成员相对应，即: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`。

为 `f_flag` 属性位定义了两个模块级常量：如果存在 `ST_RDONLY` 位，则文件系统以只读挂载；如果存在 `ST_NOSUID` 位，则文件系统禁用或不支持 setuid/setgid 位。

为基于 GNU/glibc 的系统还定义了额外的模块级常量。它们是 ST_NODEV （禁止访问设备专用文件），ST_NOEXEC （禁止执行程序），ST_SYNCHRONOUS （写入后立即同步），ST_MANDLOCK （允许文件系统上的强制锁定），ST_WRITE （写入文件/目录/符号链接），ST_APPEND （仅追加文件），ST_IMMUTABLE （不可变文件），ST_NOATIME （不更新访问时间），ST_NODIRATIME （不更新目录访问时间），ST_RELATIME （相对于 mtime/ctime 更新访问时间）。

本函数支持指定文件描述符为参数。

Availability: Unix.

在 3.2 版本发生变更: 添加了 ST_RDONLY 和 ST_NOSUID 常量。

在 3.3 版本发生变更: 新增支持将 *path* 参数指定为打开的文件描述符。

在 3.4 版本发生变更: 添加了 ST_NODEV、ST_NOEXEC、ST_SYNCHRONOUS、ST_MANDLOCK、ST_WRITE、ST_APPEND、ST_IMMUTABLE、ST_NOATIME、ST_NODIRATIME 和 ST_RELATIME 常量。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.7 版本发生变更: 增加了 f_fsid 属性。

os.**supports_dir_fd**

一个*set* 对象，指示*os* 模块中的哪些函数接受一个打开的文件描述符作为 *dir_fd* 参数。不同平台提供的功能不同，且 Python 用于实现 *dir_fd* 参数的底层函数并非在 Python 支持的所有平台上都可用。考虑到一致性，支持 *dir_fd* 的函数始终允许指定描述符，但如果在底层不支持时调用了该函数，则会抛出异常。（在所有平台上始终支持将 *dir_fd* 指定为 None。）

要检查某个函数是否接受打开的文件描述符作为 *dir_fd* 参数，请在 supports_dir_fd 前使用 in 运算符。例如，如果*os.stat()* 在当前平台上接受打开的文件描述符作为 *dir_fd* 参数，则此表达式的计算结果为 True:

```
os.stat in os.supports_dir_fd
```

目前 *dir_fd* 参数仅在 Unix 平台上有效，在 Windows 上均无效。

Added in version 3.3.

os.**supports_effective_ids**

一个*set* 对象，指示*os.access()* 是否允许在当前平台上将其 *effective_ids* 参数指定为 True。（所有平台都支持将 *effective_ids* 指定为 False。）如果当前平台支持，则集合将包含*os.access()*，否则集合为空。

如果当前平台上的*os.access()* 支持 effective_ids=True，则此表达式的计算结果为 True:

```
os.access in os.supports_effective_ids
```

目前仅 Unix 平台支持 *effective_ids*，Windows 不支持。

Added in version 3.3.

os.**supports_fd**

一个*set* 对象，指示在当前平台上*os* 模块中的哪些函数接受一个打开的文件描述符作为 *path* 参数。不同平台提供的功能不同，且 Python 所使用到的底层函数（用于实现接受描述符作为 *path*）并非在 Python 支持的所有平台上都可用。

要判断某个函数是否接受打开的文件描述符作为 *path* 参数，请在 supports_fd 前使用 in 运算符。例如，如果*os.chdir()* 在当前平台上接受打开的文件描述符作为 *path* 参数，则此表达式的计算结果为 True:

```
os.chdir in os.supports_fd
```

Added in version 3.3.

os.**supports_follow_symlinks**

一个*set* 对象，指示在当前平台上*os* 模块中的哪些函数的 *follow_symlinks* 参数可指定为 `False`。不同平台提供的功能不同，且 Python 用于实现 *follow_symlinks* 的底层函数并非在 Python 支持的所有平台上都可用。考虑到一致性，支持 *follow_symlinks* 的函数始终允许将其指定为 `False`，但如果在底层不支持时调用了该函数，则会抛出异常。（在所有平台上始终支持将 *follow_symlinks* 指定为 `True`。）

要检查某个函数的 *follow_symlinks* 参数是否可以指定为 `False`，请在 `supports_follow_symlinks` 前使用 `in` 运算符。例如，如果在当前平台上调用*os.stat()* 时可以指定 `follow_symlinks=False`，则此表达式的计算结果为 `True`:

```
os.stat in os.supports_follow_symlinks
```

Added in version 3.3.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)

创建一个指向 *src* 的符号链接，名为 *dst*。

在 Windows 上，符号链接可以表示文件或目录两种类型，并且不会动态改变类型。如果目标存在，则新建链接的类型将与目标一致。否则，如果 *target_is_directory* 为 `True`，则符号链接将创建为目录链接，为 `False` （默认）将创建为文件链接。在非 Windows 平台上，*target_is_directory* 被忽略。

本函数支持基于目录描述符的相对路径。

备注

在 Windows 10 或更高版本上，如果启用了开发人员模式，非特权帐户可以创建符号链接。如果开发人员模式不可用/未启用，则需要 *SeCreateSymbolicLinkPrivilege* 权限，或者该进程必须以管理员身份运行。

当本函数由非特权账户调用时，抛出*OSError* 异常。

引发一个审计事件 `os.symlink` 并附带参数 `src`, `dst`, `dir_fd`。

Availability: Unix, Windows.

此函数在 WASI 是受限的，请参阅*WebAssembly 平台* 了解详情。

在 3.2 版本发生变更: 添加对 Windows 6.0 (Vista) 符号链接的支持。

在 3.3 版本发生变更: 增加了 *dir_fd* 形参，现在将在非 Windows 平台上允许 *target_is_directory*。

在 3.6 版本发生变更: 接受一个类路径对象 作为 *src* 和 *dst*。

在 3.8 版本发生变更: 针对启用了开发人员模式的 Windows，添加了非特权账户创建符号链接的支持。

os.**sync**()

强制将所有内容写入磁盘。

Availability: Unix.

Added in version 3.3.

os.**truncate**(*path*, *length*)

截断 *path* 对应的文件，以使其最大为 *length* 字节。

本函数支持指定文件描述符为参数。

引发一个审计事件 `os.truncate` 并附带参数 `path`, `length`。

Availability: Unix, Windows.

Added in version 3.3.

在 3.5 版本发生变更: 添加了 Windows 支持

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**unlink**(*path*, *, *dir_fd=None*)

移除（删除）文件 *path*。该函数在语义上与`remove()` 相同，`unlink` 是其传统的 Unix 名称。请参阅`remove()` 的文档以获取更多信息。

引发一个审计事件 `os.remove` 并附带参数 `path`, `dir_fd`。

在 3.3 版本发生变更: 添加了 *dir_fd* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**utime**(*path*, *times=None*, *, [*ns*,]*dir_fd=None*, *follow_symlinks=True*)

设置文件 *path* 的访问时间和修改时间。

`utime()` 有 *times* 和 *ns* 两个可选参数，它们指定了设置给 *path* 的时间，用法如下：

- 如果指定 *ns*，它必须是一个 `(atime_ns, mtime_ns)` 形式的二元组，其中每个成员都是一个表示纳秒的整数。
- 如果 *times* 不为 `None`，则它必须是 `(atime, mtime)` 形式的二元组，其中每个成员都是一个表示秒的 int 或 float。
- 如果 *times* 为 `None` 且未指定 *ns*，则相当于指定 `ns=(atime_ns, mtime_ns)`，其中两个时间均为当前时间。

同时为 *times* 和 *ns* 指定元组会出错。

请注意你在此处设置的确切时间可能不会被后续的`stat()` 调用所返回，具体取决于你的操作系统记录访问和修改时间的分辨率；请参阅`stat()`。保留准确时间的最佳方式是使用来自于将 *ns* 形参设为`utime()` 的`os.stat()` 结果对象的 *st_atime_ns* 和 *st_mtime_ns* 字段。

本函数支持指定文件描述符、指定基于目录描述符的相对路径 和不跟踪符号链接。

引发一个审计事件 `os.utime` 并附带参数 `path`, `times`, `ns`, `dir_fd`。

在 3.3 版本发生变更: 新增支持将 *path* 参数指定为打开的文件描述符，以及支持 *dir_fd*、*follow_symlinks* 和 *ns* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**walk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

生成目录树中的文件名，方式是按上-> 下或下-> 上顺序浏览目录树。对于以 *top* 为根的目录树中的每个目录（包括 *top* 本身），它都会生成一个三元组 `(dirpath, dirnames, filenames)`。

dirpath 是一个字符串，表示目录的路径。*dirnames* 是由 *dirpath* 中的子目录名称组成的列表 (包括指向目录的符号链接，不包括 `'.'` 和 `'..'`)。*filenames* 是由 *dirpath* 中非目录文件名称组成的列表。请注意列表中的名称不包含路径部分。要获取 *dirpath* 中文件或目录的完整路径 (以 *top* 打头，请执行 `os.path.join(dirpath, name)`。列表是否排序取决于具体文件系统。如果有文件在列表生成期间被移除或添加到 *dirpath*，是否要包括该文件的名称并没有规定。

如果可选参数 *topdown* 为 `True` 或未指定，则在所有子目录的三元组之前生成父目录的三元组（目录是自上而下生成的）。如果 *topdown* 为 `False`，则在所有子目录的三元组生成之后再生成父目录的三元组（目录是自下而上生成的）。无论 *topdown* 为何值，在生成目录及其子目录的元组之前，都将检索全部子目录列表。

当 *topdown* 为 `True` 时，调用者可以就地修改 *dirnames* 列表（也许用到了 `del` 或切片），而`walk()` 将仅仅递归到仍保留在 *dirnames* 中的子目录内。这可用于减少搜索、加入特定的访问顺序，甚至可在继续`walk()` 之前告知`walk()` 由调用者新建或重命名的目录的信息。当 *topdown* 为 `False` 时，修改 *dirnames* 对 walk 的行为没有影响，因为在自下而上模式中，*dirnames* 中的目录是在 *dirpath* 本身之前生成的。

默认将忽略`scandir()` 调用中的错误。如果指定了可选参数 *onerror*，它应该是一个函数。出错时它会被调用，参数是一个`OSError` 实例。它可以报告错误然后继续遍历，或者抛出异常然后中止遍历。注意，可以从异常对象的 `filename` 属性中获取出错的文件名。

`walk()` 默认不会递归进指向目录的符号链接。可以在支持符号链接的系统上将 *followlinks* 设置为 `True`，以访问符号链接指向的目录。

> **备注**
>
> 注意，如果链接指向自身的父目录，则将 *followlinks* 设置为 `True` 可能导致无限递归。*walk()* 不会记录它已经访问过的目录。

> **备注**
>
> 如果传入的是相对路径，请不要在恢复*walk()* 之间更改当前工作目录。*walk()* 不会更改当前目录，并假定其调用者也不会更改当前目录。

下面的例子显示了起始目录下的每个子目录内非目录文件所占用的字节数，例外情况是它不会在任何 `__pycache__` 子目录内进行查找:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # 不访问 __pycache__ 目录
```

在下一个示例（*shutil.rmtree()* 的简单实现）中，必须使树自下而上遍历，因为*rmdir()* 只允许在目录为空时删除目录:

```
# 删除可从 "top" 指定的目录进入的所有东西，
# 假定其中没有符号连接。
# 注意：这很危险！举例来说，如果 top == '/'，
# 它可能删除你所有的硬盘文件。
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
os.rmdir(top)
```

引发一个审计事件 `os.walk` 并附带参数 `top`, `topdown`, `onerror`, `followlinks`。

在 3.5 版本发生变更: 现在，本函数调用的是*os.scandir()* 而不是*os.listdir()*，从而减少了调用*os.stat()* 的次数而变得更快。

在 3.6 版本发生变更: 接受一个*path-like object*。

`os.`**`fwalk`**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

本方法的行为与*walk()* 完全一样，除了它产生的是 4 元组 `(dirpath, dirnames, filenames, dirfd)`，并且它支持 `dir_fd`。

dirpath、*dirnames* 和 *filenames* 与*walk()* 输出的相同，*dirfd* 是指向目录 *dirpath* 的文件描述符。

本函数始终支持基于目录描述符的相对路径 和不跟踪符号链接。但是请注意，与其他函数不同，*fwalk()* 的 *follow_symlinks* 的默认值为 `False`。

> **备注**
>
> 由于*fwalk()* 会生成文件描述符，而它们仅在下一个迭代步骤前有效，因此如果要将描述符保留更久，则应复制它们（比如使用*dup()*）。

下面的例子显示了起始目录下的每个子目录内非目录文件所占用的字节数，例外情况是它不会在任何 __pycache__ 子目录内进行查找:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/xml'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # 不访问 __pycache__ 目录
```

在下一个示例中，必须使树自下而上遍历，因为*rmdir()* 只允许在目录为空时删除目录:

```
# 删除可从 "top" 指定的目录进入的所有东西，
# 假定其中没有符号链接。
# 注意：这很危险！举例来说，如果 top == '/'，
# 它会删除你所有的磁盘文件。
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

引发一个审计事件 `os.fwalk` 并附带参数 `top`, `topdown`, `onerror`, `follow_symlinks`, `dir_fd`。

Availability: Unix.

Added in version 3.3.

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.7 版本发生变更: 添加了对*bytes* 类型路径的支持。

os.memfd_create(*name*[, *flags=os.MFD_CLOEXEC*])

创建一个匿名文件，返回指向该文件的文件描述符。*flags* 必须是系统上可用的 `os.MFD_*` 常量之一（或将它们按位“或”组合起来）。新文件描述符默认是不可继承的。

name 提供的名称会被用作文件名，并且 `/proc/self/fd/` 目录中相应符号链接的目标将显示为该名称。显示的名称始终以 `memfd:` 为前缀，并且仅用于调试目的。名称不会影响文件描述符的行为，因此多个文件可以有相同的名称，不会有副作用。

Availability: Linux >= 3.17 with glibc >= 2.27.

Added in version 3.8.

os.MFD_CLOEXEC

os.MFD_ALLOW_SEALING

os.MFD_HUGETLB

os.MFD_HUGE_SHIFT

os.MFD_HUGE_MASK

os.MFD_HUGE_64KB

os.MFD_HUGE_512KB

os.MFD_HUGE_1MB

os.MFD_HUGE_2MB

os.MFD_HUGE_8MB

os.MFD_HUGE_16MB

os.MFD_HUGE_32MB

os.MFD_HUGE_256MB

os.MFD_HUGE_512MB

os.**MFD_HUGE_1GB**

os.**MFD_HUGE_2GB**

os.**MFD_HUGE_16GB**

以上标志位可以传递给*memfd_create()*。

Availability: Linux >= 3.17 with glibc >= 2.27

MFD_HUGE* 旗标仅从 Linux 4.14 开始可用。

Added in version 3.8.

os.**eventfd**(*initval*[, *flags=os.EFD_CLOEXEC*])

创建并返回一个事件文件描述符。此文件描述符支持缓冲区大小为 8 的原生*read()* 和*write()* 操作、*select()* 、*poll()* 等类似操作。更多信息请参阅 man 文档 *eventfd(2)*。默认情况下，新的文件描述符是*non-inheritable*。

initval 是事件计数哭喊的初始值。该初始值必须是一个 32 位无符号整数。请注意虽然事件计数器是一个最大值为 2^{64}-2 的无符号 64 位整数但初始值仍被限制为 32 位无符号整数。

flags 可由*EFD_CLOEXEC* 、*EFD_NONBLOCK* 和*EFD_SEMAPHORE* 组合而成。

如果设置了*EFD_SEMAPHORE*，并且事件计数器非零，那么*eventfd_read()* 将返回 1 并将计数器递减 1。

如果未设置*EFD_SEMAPHORE*，并且事件计数器非零，那么*eventfd_read()* 返回当前的事件计数器值，并将计数器重置为零。

如果事件计数器为 0，并且未设置*EFD_NONBLOCK*，那么*eventfd_read()* 会阻塞。

eventfd_write() 会递增事件计数器。如果写操作会让计数器的增量大于 2^{64}-2，则写入会被阻止。

示例:

```
import os

# 取值从 '1' 开始的信号量
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # 获取信号量
    v = os.eventfd_read(fd)
    try:
        do_work()
    finally:
        # 释放信号量
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.10.

os.**eventfd_read**(*fd*)

从一个*eventfd()* 文件描述符中读取数据，并返回一个 64 位无符号整数。该函数不会校验 *fd* 是否为*eventfd()*。

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**eventfd_write**(*fd*, *value*)

向一个*eventfd()* 文件描述符加入数据。*value* 必须是一个 64 位无符号整数。本函数不会校验 *fd* 是否为*eventfd()*。

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_CLOEXEC**

为新的*eventfd()* 文件描述符设置 close-on-exec 标志。

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_NONBLOCK**

为新的*eventfd()* 文件描述符设置*O_NONBLOCK* 状态标志。

Availability: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_SEMAPHORE**

为从*eventfd()* 文件描述符进行读取提供类似信号量的语法。在读取时内部计数器将递减一。

Availability: Linux >= 2.6.30

Added in version 3.10.

计时器文件描述符

Added in version 3.13.

这些函数提供了对 Linux 的 计时器文件描述符 API 的支持。当然，它们仅在 Linux 上可用。

os.**timerfd_create**(*clockid*, */*, ***, *flags=0*)

创建并返回一个计时器文件描述符 (*timerfd*)。

由*timerfd_create()* 返回的文件描述符可支持：

- *read()*
- *select()*
- *poll()*

文件描述符的*read()* 方法被调用时可附带大小为 8 的缓冲区。如果计时器已到期一次或多次，*read()* 将返回以主机端序表示的到期次数，它可通过 `int.from_bytes(x, byteorder=sys.byteorder)` 转换为*int*。

select() 和*poll()* 可被用于等待直至计时器到期并且文件描述符为可读状态。

clockid 必须是一个有效的*时钟 ID*，如*time* 模块中所定义的：

- *time.CLOCK_REALTIME*
- *time.CLOCK_MONOTONIC*
- *time.CLOCK_BOOTTIME* (自 Linux 3.15 起用于 timerfd_create)

如果 *clockid* 为*time.CLOCK_REALTIME*，则将使用一个可设置的系统级实时时钟。如果系统时钟发生改变，计时器设置将需要被更新。要在系统时钟发生改变时取消计时器，请参阅*TFD_TIMER_CANCEL_ON_SET*。

如果 *clockid* 为*time.CLOCK_MONOTONIC*，将使用一个不可设置的单调递增时钟。即使系统时钟发生改变，计时器设置也不会受影响。

如果 *clockid* 为*time.CLOCK_BOOTTIME*，将与*time.CLOCK_MONOTONIC* 相似，不同之处在于它还包括任何系统挂起的时间。

这个文件描述符的行为可以通过指定 *flags* 值来改变。可以使用以下任意变量，可以使用按位 OR (|运算符) 来进行组合：

- *TFD_NONBLOCK*
- *TFD_CLOEXEC*

如果未将*TFD_NONBLOCK* 设为一个旗标，*read()* 将会阻塞直至计时器到期。如果它被设为旗标，则*read()* 不会阻塞，但是如果自从上次对 read 的调用以来尚未有一次到期，*read()* 将引发*OSError* 并将 errno 设为*errno.EAGAIN*。

TFD_CLOEXEC 总是会由 Python 自动设置。

该文件描述符在其不再需要时必须通过*os.close()* 来关闭，否则文件描述符将被泄漏。

参见

timerfd_create(2) 帮助页。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_settime**(*fd*, */*, ***, *flags=flags*, *initial=0.0*, *interval=0.0*)

修改一个计时器文件描述符的内部计时器。此函数将操作与*timerfd_settime_ns()* 相同的间隔计时器。

fd 必须是一个有效的计时器文件描述符。

该计时器的行为可以通过指定 *flags* 值来改变。可以使用以下任意变量，可以使用按位 OR (| 运算符) 来进行组合：

- *TFD_TIMER_ABSTIME*
- *TFD_TIMER_CANCEL_ON_SET*

该计时器可通过将 *initial* 设为零 (0) 来禁用。如果 *initial* 大于零，计时器将被启用。如果 *initial* 小于零，它将引发*OSError* 异常并将 errno 设为*errno.EINVAL*

在默认情况下计时器将在经过 *initial* 秒后启动。(如果 *initial* 为零，计时器将立即启动。)

但是，如果设置了*TFD_TIMER_ABSTIME* 旗标，计时器将在计时器时钟 (由*timerfd_create()* 中的 *clockid* 设置) 达到 *initial* 秒时启动。

计时器的间隔时间是通过 *interval* *float* 设置的。如果 *interval* 为零，计时器将只在初始到期时启动一次。如果 *interval* 大于零，计时器将从上一次到期后每经过 *interval* 秒启动一次。如果 *interval* 小于零，它将引发*OSError* 并将 errno 设为*errno.EINVAL*

如果*TFD_TIMER_CANCEL_ON_SET* 旗标与*TFD_TIMER_ABSTIME* 一起被设置并且用于此计时器的时钟为*time.CLOCK_REALTIME*，则当实时计时器发生不连续改变时计时器将被标记为可取消的。对描述符的读取将被取消并设置 ECANCELED 错误。

Linux 将以 UTC 来管理系统时钟。夏令时调整是仅通过修改时差完成的而不会导致不连续的系统时钟改变。

下列事件会导致不连续的系统时钟变化：

- settimeofday
- clock_settime
- 通过 date 命令设置系统日期和时间

根据此函数执行之前的计时器状态，返回一个二元组 (next_expiration, interval)。

参见

timerfd_create(2), *timerfd_settime(2)*, *settimeofday(2)*, *clock_settime(2)* 以及 *date(1)*。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_settime_ns**(*fd*, /, *, *flags=0*, *initial=0*, *interval=0*)

与*timerfd_settime()* 类似，但使用纳秒形式的时间。此函数将操作与*timerfd_settime()* 相同的间隔计时器。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_gettime**(*fd*, /)

返回一个浮点数形式的二元组 (next_expiration, interval)。

next_expiration 表示距离定时器下一次启动的相对时间，无论是否设置了*TFD_TIMER_ABSTIME* 旗标。

interval 表示计时器的间隔。如果为零，该计时器将只启动一次，即在经过了 next_expiration 秒之后。

参见

timerfd_gettime(2)

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_gettime_ns**(*fd*, /)

与*timerfd_gettime()* 类似，但返回以纳秒为单位的时间。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_NONBLOCK**

一个用于*timerfd_create()* 函数的旗标，它将为新的计时器文件描述符设置*O_NONBLOCK* 状态旗标。如果未将*TFD_NONBLOCK* 设为旗标，*read()* 将会阻塞。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_CLOEXEC**

一个用于*timerfd_create()* 函数的旗标，如果将*TFD_CLOEXEC* 设为旗标，将为新的文件描述符设置 close-on-exec 旗标。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_ABSTIME**

一个用于*timerfd_settime()* 和*timerfd_settime_ns()* 函数的旗标。如果设置了此旗标，*initial* 将被解读为计时器时钟上的一个绝对数值（即自 Unix 纪元开始的 UTC 秒数或纳秒数）。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_CANCEL_ON_SET**

一个配合*TFD_TIMER_ABSTIME* 用于*timerfd_settime()* 和*timerfd_settime_ns()* 函数的旗标。当下层时钟发生不连续改变时计时器将被取消。

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

Linux 扩展属性

Added in version 3.3.

这些函数仅在 Linux 上可用。

os.**getxattr**(*path*, *attribute*, *, *follow_symlinks=True*)

返回 *path* 的扩展文件系统属性 *attribute* 的值。*attribute* 可以是 bytes 或 str（直接传入或通过 *PathLike* 接口间接传入）。如果是 str，则使用文件系统编码来编码字符串。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

引发一个审计事件 os.getxattr 并附带参数 path, attribute。

在 3.6 版本发生变更: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**listxattr**(*path=None*, *, *follow_symlinks=True*)

返回一个列表，包含 *path* 的所有扩展文件系统属性。列表中的属性都表示为字符串，它们是根据文件系统编码解码出来的。如果 *path* 为 None，则 *listxattr()* 将检查当前目录。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

引发一个审计事件 os.listxattr 并附带参数 path。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)

移除 *path* 中的扩展文件系统属性 *attribute*。*attribute* 应为字节串或字符串类型（通过 *PathLike* 接口直接或间接得到）。若为字符串类型，则用*filesystem encoding and error handler* 进行编码。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

引发一个审计事件 os.removexattr 并附带参数 path, attribute。

在 3.6 版本发生变更: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)

将 *path* 的文件系统扩展属性 *attribute* 设为 *value*。*attribute* 必须是一个字节串或字符串，不含 NUL（通过 *PathLike* 接口直接或间接得到）。若为字符串，将用*filesystem encoding and error handler* 进行编码。*flags* 可以是 *XATTR_REPLACE* 或 *XATTR_CREATE*。如果给出 *XATTR_REPLACE* 而属性不存在，则会触发 ENODATA。如果给出了 *XATTR_CREATE* 而属性已存在，则不会创建属性并将触发 EEXISTS。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

> **备注**
>
> Linux kernel 2.6.39 以下版本的一个 bug 导致在某些文件系统上，flags 参数会被忽略。

引发一个审计事件 os.setxattr 并附带参数 path, attribute, value, flags。

在 3.6 版本发生变更: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**XATTR_SIZE_MAX**

一条扩展属性的值的最大大小。在当前的 Linux 上是 64 KiB。

os.**XATTR_CREATE**

这是 *setxattr()* 的 flags 参数的可取值，它表示该操作必须创建一个属性。

os.**XATTR_REPLACE**

这是 *setxattr()* 的 flags 参数的可取值，它表示该操作必须替换现有属性。

16.1.7 进程管理

下列函数可用于创建和管理进程。

所有*exec**函数都接受一个参数列表，用来给新程序加载到它的进程中。在所有情况下，传递给新程序的第一个参数是程序本身的名称，而不是用户在命令行上输入的参数。对于 C 程序员来说，这就是传递给 `main()` 函数的 `argv[0]`。例如，`os.execv('/bin/echo', ['foo', 'bar'])` 只会在标准输出上打印 `bar`，而 `foo` 会被忽略。

`os.abort()`

发送 `SIGABRT` 信号到当前进程。在 Unix 上，默认行为是生成一个核心转储。在 Windows 上，该进程立即返回退出代码 `3`。请注意，使用*signal.signal()*可以为 `SIGABRT` 注册 Python 信号处理程序，而调用本函数将不会调用按前述方法注册的程序。

`os.add_dll_directory(path)`

将路径添加到 DLL 搜索路径。

当需要解析扩展模块的依赖时（扩展模块本身通过*sys.path*解析），会使用该搜索路径，*ctypes*也会使用该搜索路径。

要移除目录，可以在返回的对象上调用 **close()**，也可以在 `with` 语句内使用本方法。

参阅 Microsoft 文档 获取如何加载 DLL 的信息。

引发一个审计事件 `os.add_dll_directory` 并附带参数 `path`。

Availability: Windows.

Added in version 3.8: 早期版本的 CPython 解析 DLL 时用的是当前进程的默认行为。这会导致不一致，比如不是每次都会去搜索 `PATH` 和当前工作目录，且系统函数（如 `AddDllDirectory`）失效。

在 3.8 中，DLL 的两种主要加载方式现在可以显式覆盖进程的行为，以确保一致性。请参阅 移植说明了解如何更新你的库。

`os.execl(path, arg0, arg1, ...)`

`os.execle(path, arg0, arg1, ..., env)`

`os.execlp(file, arg0, arg1, ...)`

`os.execlpe(file, arg0, arg1, ..., env)`

`os.execv(path, args)`

`os.execve(path, args, env)`

`os.execvp(file, args)`

`os.execvpe(file, args, env)`

这些函数都将执行一个新程序，以替换当前进程。它们没有返回值。在 Unix 上，新程序会加载到当前进程中，且进程号与调用者相同。过程中的错误会被报告为*OSError*异常。

当前进程会被立即替换。打开的文件对象和描述符都不会刷新，因此如果这些文件上可能缓冲了数据，则应在调用*exec**函数之前使用 `sys.stdout.flush()` 或*os.fsync()*刷新它们。

*exec**函数的"l" 和"v" 变体的不同在于命令行参数的传递方式。如果在编写代码时形参数量是固定的，则"l" 变体可能是最方便的；单个形参简单地作为传给 `execl*()` 函数的额外形参即可。当形参数量可变时则"v" 变体更为好用，参数将以列表或元组的形式作为 *args* 形参传入。在这两种情况下，传给子进程的参数应当以要运行的命令名称开头，但这不是强制性的。

在结尾位置包括"p" 的变体形式 (*execlp()*, *execlpe()*, *execvp()* 和*execvpe()*) 将使用 `PATH` 环境变量来定位程序 *file*。当环境被替换时 (使用某个*exec*e* 变体形式，将在下一段中讨论)，将使用新环境作为 `PATH` 变量的来源。其他的变体形式*execl()*, *execle()*, *execv()* 和*execve()* 将不使用 `PATH` 变量来定位可执行程序；*path* 必须包含正确的绝对或相对路径。相对路径必须包括至少一个斜杠，即使是在 Windows 上，因为简单名称将不会被解析。

对于*execle()*、*execlpe()*、*execve()* 和*execvpe()*（都以"e" 结尾），*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量）。而函数*execl()*、*execlp()*、*execv()* 和*execvp()* 会将当前进程的环境变量过继给新进程。

某些平台上的*execve()* 可以将 *path* 指定为打开的文件描述符。当前平台可能不支持此功能，可以使用*os.supports_fd* 检查它是否支持。如果不可用，则使用它会抛出*NotImplementedError* 异常。

引发一个审计事件 os.exec 并附带参数 path, args, env。

Availability: Unix, Windows, not WASI, not Android, not iOS.

在 3.3 版本发生变更: 新增支持将*execve()* 的 *path* 参数指定为打开的文件描述符。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.**_exit**(*n*)

以状态码 *n* 退出进程，不会调用清理处理程序，不会刷新 stdio，等等。

> **备注**
>
> 退出的标准方式是使用*sys.exit(n)*。_exit() 通常只应在*fork()* 所生成的子进程中使用。

以下是已定义的退出代码，可以用于*_exit()*，尽管它们不是必需的。这些退出代码通常用于 Python 编写的系统程序，例如邮件服务器的外部命令传递程序。

> **备注**
>
> 其中部分退出代码在部分 Unix 平台上可能不可用，因为平台间存在差异。如果底层平台定义了这些常量，那上层也会定义。

os.**EX_OK**

表示没有发生错误的退出码。在某些平台上可能会从 EXIT_SUCCESS 定义的值中选取。通常其值为零。

Availability: Unix, Windows.

os.**EX_USAGE**

退出代码，表示命令使用不正确，如给出的参数数量有误。

Availability: Unix, not WASI.

os.**EX_DATAERR**

退出代码，表示输入数据不正确。

Availability: Unix, not WASI.

os.**EX_NOINPUT**

退出代码，表示某个输入文件不存在或不可读。

Availability: Unix, not WASI.

os.**EX_NOUSER**

退出代码，表示指定的用户不存在。

Availability: Unix, not WASI.

os.**EX_NOHOST**

退出代码，表示指定的主机不存在。

Availability: Unix, not WASI.

os.**EX_UNAVAILABLE**

退出代码，表示所需的服务不可用。

Availability: Unix, not WASI.

os.**EX_SOFTWARE**

退出代码，表示检测到内部软件错误。

Availability: Unix, not WASI.

os.**EX_OSERR**

退出代码，表示检测到操作系统错误，例如无法 fork 或创建管道。

Availability: Unix, not WASI.

os.**EX_OSFILE**

退出代码，表示某些系统文件不存在、无法打开或发生其他错误。

Availability: Unix, not WASI.

os.**EX_CANTCREAT**

退出代码，表示无法创建用户指定的输出文件。

Availability: Unix, not WASI.

os.**EX_IOERR**

退出代码，表示对某些文件进行读写时发生错误。

Availability: Unix, not WASI.

os.**EX_TEMPFAIL**

退出代码，表示发生了暂时性故障。它可能并非意味着真正的错误，例如在可重试的情况下无法建立网络连接。

Availability: Unix, not WASI.

os.**EX_PROTOCOL**

退出代码，表示协议交换是非法的、无效的或无法解读的。

Availability: Unix, not WASI.

os.**EX_NOPERM**

退出代码，表示没有足够的权限执行该操作（但不适用于文件系统问题）。

Availability: Unix, not WASI.

os.**EX_CONFIG**

退出代码，表示发生某种配置错误。

Availability: Unix, not WASI.

os.**EX_NOTFOUND**

退出代码，表示的内容类似于“找不到条目”。

Availability: Unix, not WASI.

os.**fork**()

Fork 出一个子进程。在子进程中返回 `0`，在父进程中返回子进程的进程号。如果发生错误，则抛出 *OSError* 异常。

注意，当从线程中使用 `fork()` 时，某些平台（包括 FreeBSD <= 6.3 和 Cygwin）存在已知问题。

引发一个不带参数的审计事件 `os.fork`。

> **警告**
>
> 如果你在调用 “fork()“ 的应用程序中使用 TLS 套接字，请参阅 *ssl* 文档中的警告信息。

> **警告**
>
> 在 macOS 上将此函数与高层级的系统 API 混用是不安全的，包括*urllib.request*。

在 3.8 版本发生变更: 不再支持在子解释器中调用 `fork()` （将抛出*RuntimeError* 异常）。

在 3.12 版本发生变更: 如果 Python 能够检测到你的进程有多个线程，则*os.fork()* 现在会引发*DeprecationWarning*。

在可以检测时，我们选择将此显示为警告，以便更好地告知开发人员 POSIX 平台明确指出不支持的设计问题。在 POSIX 平台上即使在 *看起来* 可行的代码中，将线程与*os.fork()* 混用也是不安全的。当父进程中存在线程时 CPython 运行时本身总是会在子进程中执行不安全的 API 调用 (如 `malloc` 和 `free`)。

使用 macOS 的用户或使用 glibc 中可找到的典型实现以外的 libc 或 malloc 实现的用户在运行此类代码时更容易发生死锁现象。

请参阅 有关 fork 与线程不兼容的讨论 了解我们为何向开发者公开这个长期存在的平台不兼容性问题的技术细节。

Availability: POSIX, not WASI, not Android, not iOS.

`os.`**`forkpty`**`()`

Fork 出一个子进程，使用新的伪终端作为子进程的控制终端。返回一对 `(pid, fd)`，其中 *pid* 在子进程中为 `0`，这是父进程中新子进程的进程号，而 *fd* 是伪终端主设备的文件描述符。对于更便于移植的方法，请使用*pty* 模块。如果发生错误，则抛出*OSError* 异常。

引发一个不带参数的审计事件 `os.forkpty`。

> **警告**
>
> 在 macOS 上将此函数与高层级的系统 API 混用是不安全的，包括*urllib.request*。

在 3.8 版本发生变更: 不再支持在子解释器中调用 `forkpty()` （将抛出*RuntimeError* 异常）。

在 3.12 版本发生变更: 现在如果 Python 能够检测到你的进程有多个线程，此函数将引发*DeprecationWarning*。请参阅有关*os.fork()* 的更详细解释。

Availability: Unix, not WASI, not Android, not iOS.

`os.`**`kill`**`(`*pid*, *sig*, */*`)`

将信号 *sig* 发送至进程 *pid*。特定平台上可用的信号常量定义在*signal* 模块中。

Windows: *signal.CTRL_C_EVENT* 和*signal.CTRL_BREAK_EVENT* 信号是只能被发送给共享同一个控制台窗口的控制台进程例如某些子进程的特殊信号。任何其他的 *sig* 值都将导致进程被 TerminateProcess API 无条件地杀掉，且退出码将被设为 *sig*。

另请参阅*signal.pthread_kill()*。

引发一个审计事件 `os.kill` 并附带参数 `pid`, `sig`。

Availability: Unix, Windows, not WASI, not iOS.

在 3.2 版本发生变更: 添加了对 Windows 的支持。

`os.`**`killpg`**`(`*pgid*, *sig*, */*`)`

将信号 *sig* 发送给进程组 *pgid*。

引发一个审计事件 `os.killpg` 并附带参数 `pgid`, `sig`。

Availability: Unix, not WASI, not iOS.

os.**nice**(*increment*, /)

将进程的优先级（nice 值）增加 *increment*，返回新的 nice 值。

Availability: Unix, not WASI.

os.**pidfd_open**(*pid*, *flags=0*)

返回一个指向设置了 *flags* 的进程 *pid* 的文件描述符。该描述符可用于执行无需竞争和信号的进程管理。

更多详细信息请参阅 *pidfd_open(2)* 手册页。

Availability: Linux >= 5.3, Android >= *build-time* API level 31

Added in version 3.9.

os.**PIDFD_NONBLOCK**

该旗标表示文件描述符将是非阻塞的。如果文件描述符所引用的进程尚未终止，那么尝试使用 *waitid(2)* 等待文件描述符将立即返回错误*EAGAIN* 而不是阻塞。

Availability: Linux >= 5.10

Added in version 3.12.

os.**plock**(*op*, /)

将程序段锁定到内存中。*op* 的值（定义在 <sys/lock.h> 中）决定了哪些段被锁定。

Availability: Unix, not WASI, not iOS.

os.**popen**(*cmd*, *mode='r'*, *buffering=-1*)

打开一个通往或接受命令 *cmd* 的管道。返回值是连接到该管道的已打开文件对象，它可读取还是可写入取决于其 *mode* 是 'r' (默认) 还是 'w'。*buffering* 参数与内置*open()* 函数相应的参数含义相同。返回的文件对象只能读写文本字符串而不是字节串。

如果子进程成功退出，则 close 方法返回*None*。如果发生错误，则返回子进程的返回码。在 POSIX 系统上，如果返回码为正，则它就是进程返回值左移一个字节后的值。如果返回码为负，则进程是被信号终止的，返回码取反后就是该信号。（例如，如果子进程被终止，则返回值可能是 -signal.SIGKILL。）在 Windows 系统上，返回值包含子进程的返回码（有符号整数）。

在 Unix 上，*waitstatus_to_exitcode()* 可以将 close 方法的返回值（即退出状态，不能是 None）转换为退出码。在 Windows 上，close 方法的结果直接就是退出码（或 None ）。

本方法是使用*subprocess.Popen* 实现的，如需更强大的方法来管理和沟通子进程，请参阅该类的文档。

Availability: not WASI, not Android, not iOS.

备注

Python UTF-8 模型 影响 *cmd* 和管道内容所使用的编码格式。

popen() 是针对*subprocess.Popen* 的简单包装器。请使用*subprocess.Popen* 或*subprocess.run()* 来控制编码格式等选项。

os.**posix_spawn**(*path*, *argv*, *env*, *, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

包装 posix_spawn() C 库 API 以供 Python 使用。

大多数用户应使用*subprocess.run()* 代替*posix_spawn()*。

位置参数 *path*, *args* 和 *env* 与*execve()* 类似。*env* 允许为 None，在此情况下将使用当前进程的环境。

path 形参是可执行文件的路径，*path* 中应当包含目录。使用*posix_spawnp()* 可传入不带目录的可执行文件。

file_actions 参数可以是由元组组成的序列，序列描述了对子进程中指定文件描述符采取的操作，这些操作会在 C 库实现的 `fork()` 和 `exec()` 步骤间完成。每个元组的第一个元素必须是下面列出的三个类型指示符之一，用于描述元组剩余的元素：

`os.`**`POSIX_SPAWN_OPEN`**

(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

执行 `os.dup2(os.open(path, flags, mode), fd)`。

`os.`**`POSIX_SPAWN_CLOSE`**

(`os.POSIX_SPAWN_CLOSE`, *fd*)

执行 `os.close(fd)`。

`os.`**`POSIX_SPAWN_DUP2`**

(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

执行 `os.dup2(fd, new_fd)`。

`os.`**`POSIX_SPAWN_CLOSEFROM`**

(`os.POSIX_SPAWN_CLOSEFROM`, *fd*)

执行 `os.closerange(fd, INF)`。

这些元组对应于 C 库 `posix_spawn_file_actions_addopen()`, `posix_spawn_file_actions_addclose()`, `posix_spawn_file_actions_adddup2()` 和 `posix_spawn_file_actions_addclosefrom_np()` API 调用，用于为 `posix_spawn()` 调用本身做准备。

setpgroup 参数将把子进程的进程组设置为指定值。如果指定值为 0，则子进程的进程组 ID 将与其进程 ID 相同。如果未设置 *setpgroup* 的值，则子进程将继承父进程的进程组 ID。本参数对应于 C 库的 `POSIX_SPAWN_SETPGROUP` 旗标。

如果 *resetids* 参数为 `True` 则它会将子进程的有效 UID 和 GID 重置为父进程的实际 UID 和 GID。如果该参数为 `False`，则子进程会保留父进程的有效 UID 和 GID。无论哪种情况，如果在可执行文件上启用了设置用户 ID 和设置组 ID 权限位，它们的效果将覆盖有效 UID 和 GID 的设置。本参数对应于 C 库的 `POSIX_SPAWN_RESETIDS` 旗标。

如果 *setsid* 参数为 `True`，它将为 `posix_spawn` 新建一个会话 ID。*setsid* 需要 `POSIX_SPAWN_SETSID` 或 `POSIX_SPAWN_SETSID_NP` 旗标。否则，将会引发*NotImplementedError*。

setsigmask 参数会将信号掩码设置为指定的信号集合。如果未使用该参数，则子进程将继承父进程的信号掩码。本参数对应于 C 库的 `POSIX_SPAWN_SETSIGMASK` 旗标。

sigdef 参数会将集合中所有信号的操作全部重置为默认。本参数对应于 C 库的 `POSIX_SPAWN_SETSIGDEF` 旗标。

scheduler 参数必须是一个元组，其中包含（可选的）调度器策略以及携带了调度器参数的*sched_param* 实例。在调度器策略所在位置的值为 `None` 表示未提供该值。本参数是 C 库的 `POSIX_SPAWN_SETSCHEDPARAM` 和 `POSIX_SPAWN_SETSCHEDULER` 旗标的组合。

引发一个审计事件 `os.posix_spawn` 并附带参数 `path`, `argv`, `env`。

Added in version 3.8.

在 3.13 版本发生变更: *env* 形参可接受 `None`。`os.POSIX_SPAWN_CLOSEFROM` 在具有 `posix_spawn_file_actions_addclosefrom_np()` 的平台上可用。

Availability: Unix, not WASI, not Android, not iOS.

`os.`**`posix_spawnp`**(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

包装 `posix_spawnp()` C 库 API 以供 Python 使用。

与*posix_spawn()* 相似，但是系统会在 `PATH` 环境变量指定的目录列表中搜索可执行文件 *executable*（与 `execvp(3)` 相同）。

引发一个审计事件 `os.posix_spawn` 并附带参数 `path`, `argv`, `env`。

Added in version 3.8.

Availability: POSIX, not WASI, not Android, not iOS.

参见*posix_spawn()* 文档。

os.**register_at_fork**(*, *before=None*, *after_in_parent=None*, *after_in_child=None*)

注册可调用对象，在使用*os.fork()* 或类似的进程克隆 API 派生新的子进程时，这些对象会运行。参数是可选的，且为仅关键字 (Keyword-only) 参数。每个参数指定一个不同的调用点。

- *before* 是一个函数，在 fork 子进程前调用。
- *after_in_parent* 是一个函数，在 fork 子进程后从父进程调用。
- *after_in_child* 是一个函数，从子进程中调用。

只有希望控制权回到 Python 解释器时，才进行这些调用。典型的子进程 启动时不会触发它们，因为子进程不会重新进入解释器。

在注册的函数中，用于 fork 前运行的函数将按与注册相反的顺序调用。用于 fork 后（从父进程或子进程）运行的函数按注册顺序调用。

注意，第三方 C 代码的 `fork()` 调用可能不会调用这些函数，除非它显式调用了 `PyOS_BeforeFork()`、`PyOS_AfterFork_Parent()` 和 `PyOS_AfterFork_Child()`。

函数注册后无法注销。

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.7.

os.**spawnl**(*mode*, *path*, *...*)

os.**spawnle**(*mode*, *path*, *...*, *env*)

os.**spawnlp**(*mode*, *file*, *...*)

os.**spawnlpe**(*mode*, *file*, *...*, *env*)

os.**spawnv**(*mode*, *path*, *args*)

os.**spawnve**(*mode*, *path*, *args*, *env*)

os.**spawnvp**(*mode*, *file*, *args*)

os.**spawnvpe**(*mode*, *file*, *args*, *env*)

在新进程中执行程序 *path*。

（注意，*subprocess* 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用这些函数更好。尤其应当检查使用 *subprocess* 模块替换旧函数 部分。）

mode 为*P_NOWAIT* 时，本函数返回新进程的进程号。*mode* 为*P_WAIT* 时，如果进程正常退出，返回退出代码，如果被终止，返回 `-signal`，其中 *signal* 是终止进程的信号。在 Windows 上，进程号实际上是进程句柄，因此可以与*waitpid()* 函数一起使用。

注意在 VxWorks 上，新进程被终止时，本函数不会返回 `-signal`，而是会抛出 OSError 异常。

*spawn** 函数的"l" 和"v" 变体的不同在于命令行参数的传递方式。如果在编写代码时形参数量是固定的，则"l" 变体可能是最方便的；单个形参简单地作为传给 `spawnl*()` 函数的额外形参即可。当形参数量可变时"v" 变体更为好用，参数将以列表或元组的形式作为 *args* 形参传入。在这两种情况下，传给子进程的参数应当以要运行的命令名称开头。

结尾包含第二个"p" 的变体（*spawnlp()*、*spawnlpe()*、*spawnvp()* 和*spawnvpe()*）将使用 `PATH` 环境变量来查找程序 *file*。当环境被替换时（使用下一段讨论的*spawn*e* 变体之一），`PATH` 变量将来自于新环境。其他变体*spawnl()*、*spawnle()*、*spawnv()* 和*spawnve()* 不使用 `PATH` 变量来查找程序，因此 *path* 必须包含正确的绝对或相对路径。

对于*spawnle()*、*spawnlpe()*、*spawnve()* 和*spawnvpe()* （都以"e" 结尾），*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量）。而函数*spawnl()*、*spawnlp()*、*spawnv()* 和*spawnvp()* 会将当前进程的环境变量过继给新进程。注意，*env* 字典中的键和值必须是字符串。无效的键或值将导致函数出错，返回值为 `127`。

例如，以下对*spawnlp()* 和*spawnvpe()* 的调用是等效的:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

引发一个审计事件 `os.spawn` 并附带参数 `mode`, `path`, `args`, `env`。

Availability: Unix, Windows, not WASI, not Android, not iOS.

spawnlp(), *spawnlpe()*, *spawnvp()* 和*spawnvpe()* 在 Windows 上不可用。*spawnle()* 和*spawnve()* 在 Windows 上不是线程安全的；我们建议你用*subprocess* 模块来代替。

在 3.6 版本发生变更: 接受一个*path-like object*。

os.P_NOWAIT

os.P_NOWAITO

传给*spawn** 函数族的 *mode* 形参可能的值。如果给出这些值中的某一个，则*spawn** 函数将在新进程创建后立即返回，并将进程 ID 作为返回值。

Availability: Unix, Windows.

os.P_WAIT

传给*spawn** 函数族的 *mode* 形参可能的值。如果作为 *mode* 给出，则*spawn** 函数将在新进程运行完毕后才返回，并在进程运行成功时返回退出码，或者如果进程被信号杀掉则返回 `-signal`。

Availability: Unix, Windows.

os.P_DETACH

os.P_OVERLAY

*spawn** 系列函数的 *mode* 参数的可取值。它们比上面列出的值可移植性差。*P_DETACH* 与*P_NOWAIT* 相似，但是新进程会与父进程的控制台脱离。使用*P_OVERLAY* 则会替换当前进程，*spawn** 函数将不会返回。

Availability: Windows.

os.startfile(*path*[, *operation*][, *arguments*][, *cwd*][, *show_cmd*])

使用已关联的应用程序打开文件。

当未指定 *operation* 时，这类似于在 Windows 资源管理器中双击文件，或在交互式命令行中将文件名作为 **start** 命令的参数：通过扩展名所关联的应用程序（如果有的话）来打开文件。

当指定其他 *operation* 时，它必须是一个对该文件执行操作的“命令动词”。Microsoft 文档中记录的常用动词有 `'open'`, `'print'` 和 `'edit'` (用于文件) 以及 `'explore'` 和 `'find'` (用于目录)。

在启动某个应用程序时，*arguments* 将作为一个字符串传入。若是打开某个文档，此参数可能没什么效果。

默认工作目录是继承而来的，但可以通过 *cwd* 参数进行覆盖。且应为绝对路径。相对路径 *path* 将据此参数进行解析。

使用 *show_cmd* 覆盖默认的窗口样式。这是否生效则取决于被启动的应用程序。其值应为 Win32 `ShellExecute()` 函数所支持的整数形式。

在关联的应用程序启动后，*startfile()* 就会立即返回。没有提供等待应用程序关闭的选项，也没有办法获得应用程序的退出状态。*path* 形参是基于当前目录或 *cwd* 的相对路径。如果要使用绝对路径，请确保第一个字符不为斜杠 (`'/'`)。请用*pathlib* 或*os.path.normpath()* 函数来保证路径已按照 Win32 的要求进行了正确的编码。

为了减少解释器的启动开销，Win32 `ShellExecute()` 函数将不会被解析直到该函数首次被调用。如果该函数无法被解析，则将引发*NotImplementedError* 异常。

引发一个审计事件 `os.startfile` 并附带参数 `path`, `operation`。

引发一个审计事件 `os.startfile/2` 并附带参数 `path`, `operation`, `arguments`, `cwd`, `show_cmd`。

Availability: Windows.

在 3.10 版本发生变更: 加入了 *arguments*、*cwd* 和 *show_cmd* 参数，以及 `os.startfile/2` 审计事件。

`os.`**`system`**(*command*)

在子外壳程序中执行此命令（一个字符串）。这是通过调用标准 C 函数 `system()` 来实现的，并受到同样的限制。对 *sys.stdin* 的更改等不会反映在所执行命令的环境中。如果 *command* 生成了任何输出，它将被发送到解释器的标准输出流。C 标准没有指明这个 C 函数返回值的含义，因此这个 Python 函数的返回值取决于具体系统。

在 Unix 上，返回值为进程的退出状态，以针对 *wait()* 而指定的格式进行编码。

在 Windows 上，返回值是运行 *command* 后系统 Shell 返回的值。该 Shell 由 Windows 环境变量 `COMSPEC`: 给出：通常是 **`cmd.exe`**，它会返回命令的退出状态。在使用非原生 Shell 的系统上，请查阅 Shell 的文档。

subprocess 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用本函数更好。参阅 *subprocess* 文档中的 *使用 subprocess 模块替换旧函数* 部分以获取有用的帮助。

在 Unix 上，*waitstatus_to_exitcode()* 可以将返回值（即退出状态）转换为退出码。在 Windows 上，返回值就是退出码。

引发一个*审计事件* `os.system` 并附带参数 `command`。

Availability: Unix, Windows, not WASI, not Android, not iOS.

`os.`**`times`**()

返回当前的全局进程时间。返回值是一个有 5 个属性的对象：

- `user` - 用户时间
- `system` - 系统时间
- `children_user` - 所有子进程的用户时间
- `children_system` - 所有子进程的系统时间
- `elapsed` - 从过去的固定时间点起，经过的真实时间

为了向后兼容，该对象的行为也类似于五元组，按照 `user`，`system`，`children_user`，`children_system` 和 `elapsed` 顺序组成。

在 Unix 上请参阅 Unix 手册页 *times(2)* 和 times(3) 而在 Windows 上请参阅 GetProcessTimes MSDN。在 Windows 上，只有 `user` 和 `system` 是已知的；其他属性均为零。

Availability: Unix, Windows.

在 3.3 版本发生变更: 返回结果的类型由元组变成一个类似元组的对象，同时具有命名的属性。

`os.`**`wait`**()

等待子进程执行完毕，返回一个元组，包含其 pid 和退出状态指示：一个 16 位数字，其低字节是终止该进程的信号编号，高字节是退出状态码（信号编号为零的情况下），如果生成了核心文件，则低字节的高位会置位。

如果不存在可被等待的子进程，则将引发 *ChildProcessError*。

可以使用 *waitstatus_to_exitcode()* 来将退出状态转换为退出码。

Availability: Unix, not WASI, not Android, not iOS.

> **参见**
>
> 下面列出的其他 `wait*()` 函数可被用于等待特定子进程完成并具有更多的选项。*waitpid()* 是其中唯一在 Windows 上也可用的。

`os.`**`waitid`**(*idtype*, *id*, *options*, */*)

等待一个子进程完成。

idtype 可以为*P_PID*, *P_PGID*, *P_ALL* 或 (在 Linux 上) *P_PIDFD*。对于 *id* 的解读由它来决定；请参阅它们各自的说明。

options 是多个旗标的 OR 组合。要求至少有*WEXITED*, *WSTOPPED* 或*WCONTINUED* 中的一个；*WNOHANG* 和*WNOWAIT* 是附加的可选旗标。

返回值是一个代表 `siginfo_t` 结构体所包含数据的对象，具有以下属性:

- `si_pid` (进程 ID)
- `si_uid` (子进程的实际用户 ID)
- `si_signo` (始终为*SIGCHLD*)
- `si_status` (退出状态或信号编号，具体取决于 `si_code`)
- `si_code` (可能的值参见*CLD_EXITED*)

如果指定了*WNOHANG* 而在所请求的状态中没有匹配的子进程，则将返回 `None`。在其他情况下，如果没有匹配的子进程可被等待，则将引发*ChildProcessError*。

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

在 3.13 版本发生变更: 现在该函数在 macOS 同样可用。

`os.`**`waitpid`**(*pid*, *options*, */*)

本函数的细节在 Unix 和 Windows 上有不同之处。

在 Unix 上：等待进程号为 *pid* 的子进程执行完毕，返回一个元组，内含其进程 ID 和退出状态指示(编码与*wait()* 相同)。调用的语义受整数 *options* 的影响，常规操作下该值应为 `0`。

如果 *pid* 大于 `0`，则*waitpid()* 会获取该指定进程的状态信息。如果 *pid* 为 `0`，则获取当前进程所在进程组中的所有子进程的状态。如果 *pid* 为 `-1`，则获取当前进程的子进程状态。如果 *pid* 小于 `-1`，则获取进程组 `-pid` （*pid* 的绝对值）中所有进程的状态。

options 是多个旗标的 OR 组合。如果它包含了*WNOHANG* 并且在所请求的状态中没有匹配的子进程，则将返回 `(0, 0)`。在其他情况下，如果没有匹配的子进程可以被等待，则将引发*ChildProcessError*。其他的可用选项还有*WUNTRACED* 和*WCONTINUED*。

在 Windows 上：等待句柄为 *pid* 的进程执行完毕，返回一个元组，内含 *pid* 以及左移 8 位后的退出状态码（移位简化了跨平台使用本函数)。小于或等于 `0` 的 *pid* 在 Windows 上没有特殊含义，且会抛出异常。整数值 *options* 无效。*pid* 可以指向任何 ID 已知的进程，不一定是子进程。调用*spawn** 函数时传入*P_NOWAIT* 将返回合适的进程句柄。

可以使用*waitstatus_to_exitcode()* 来将退出状态转换为退出码。

Availability: Unix, Windows, not WASI, not Android, not iOS.

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

`os.`**`wait3`**(*options*)

与*waitpid()* 类似，区别在于没有给出进程 id 参数并且返回一个 3 元组，其中包括子进程的 id、退出状态指示以及资源使用信息。请参阅*resource.getrusage()* 了解有关资源使用信息的详情。*options* 参数与提供给*waitpid()* 和*wait4()* 的相同。

可以使用*waitstatus_to_exitcode()* 来将退出状态转换为退出码。

Availability: Unix, not WASI, not Android, not iOS.

`os.`**`wait4`**(*pid*, *options*)

与*waitpid()* 类似，区别在于它是返回一个 3 元组，其中包括子进程的 id、退出状态指示以及资源使用信息。请参阅*resource.getrusage()* 了解有关资源使用信息的详情。*wait4()* 的参数与提供给*waitpid()* 的相同。

可以使用*waitstatus_to_exitcode()* 来将退出状态转换为退出码。

Availability: Unix, not WASI, not Android, not iOS.

os.**P_PID**

os.**P_PGID**

os.**P_ALL**

os.**P_PIDFD**

这些是*waitid()* 中 *idtype* 可取的值。它们会影响 *id* 的解读方式:

- P_PID - 等待 PID 为 *id* 的子进程。
- P_PGID - 等待进程组 ID 为 *id* 的任何子进程。
- P_ALL - 等待任何子进程；*id* 会被忽略。
- P_PIDFD - 等待以文件描述符 *id* 作为标识的子进程（使用 a process file descriptor created with *pidfd_open()* 创建的进程文件描述符）。

Availability: Unix, not WASI, not Android, not iOS.

> **备注**
>
> P_PIDFD 仅在 Linux >= 5.4 时可用。

Added in version 3.3.

Added in version 3.9: P_PIDFD 常量。

os.**WCONTINUED**

如果子进程自它们上次被报告之后从作业控制停止位置继续执行，则*waitpid()*, *wait3()*, *wait4()* 和*waitid()* 的这个 选项旗标将导致子进程被报告。

Availability: Unix, not WASI, not Android, not iOS.

os.**WEXITED**

waitid() 的这个 选项旗标将导致已终结的子进程被报告。

其他 wait* 函数总是会报告已终结的子进程，所以此选项对它们不可用。

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

os.**WSTOPPED**

这个*waitid()* 的 选项旗标将导致被信号发送所停止的子进程被报告。

这个选项对于其他 wait* 函数不可用。

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

os.**WUNTRACED**

如果子进程自它们上次被停止之后再次被停止但它们的当前状态还未被报告，则*waitpid()*, *wait3()* 和*wait4()* 的这个 选项旗标也将导致子进程被报告。

这个选项对于*waitid()* 不可用。

Availability: Unix, not WASI, not Android, not iOS.

os.**WNOHANG**

如果没有任何子进程状态是立即可用的，则这个 选项旗标将导致*waitpid()*, *wait3()*, *wait4()* 和*waitid()* 立即返回。

Availability: Unix, not WASI, not Android, not iOS.

os.**WNOWAIT**

这个 选项旗标将导致*waitid()* 以可等待的状态离开子进程，这样后续的 wait*() 调用可被用来再次获取子进程状态信息。

这个选项对于其他 wait* 函数不可用。

Availability: Unix, not WASI, not Android, not iOS.

os.**CLD_EXITED**

os.**CLD_KILLED**

os.**CLD_DUMPED**

os.**CLD_TRAPPED**

os.**CLD_STOPPED**

os.**CLD_CONTINUED**

这些是由*waitid()* 所返回的结果中 si_code 可能的取值。

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

在 3.9 版本发生变更: 添加了*CLD_KILLED* 和*CLD_STOPPED* 值。

os.**waitstatus_to_exitcode**(*status*)

将等待状态转换为退出码。

在 Unix 上：

- 如果进程正常退出（当 WIFEXITED(status) 为真值），则返回进程退出状态 (返回 WEXITSTATUS(status)): 结果值大于等于 0。
- 如果进程被信号终止（当 WIFSIGNALED(status) 为真值），则返回 -signum 其中 *signum* 为导致进程终止的信号数值 (返回 -WTERMSIG(status)): 结果值小于 0。
- 否则将抛出*ValueError* 异常。

在 Windows 上，返回 *status* 右移 8 位的结果。

在 Unix 上，如果进程正被追踪或*waitpid()* 附带*WUNTRACED* 选项被调用，则调用者必须先检查 WIFSTOPPED(status) 是否为真值。如果 WIFSTOPPED(status) 为真值则此函数不可被调用。

参见

WIFEXITED(), *WEXITSTATUS()*, *WIFSIGNALED()*, *WTERMSIG()*, *WIFSTOPPED()*, *WSTOPSIG()* 函数。

Availability: Unix, Windows, not WASI, not Android, not iOS.

Added in version 3.9.

下列函数采用进程状态码作为参数，状态码由*system()*、*wait()* 或*waitpid()* 返回。它们可用于确定进程上发生的操作。

os.**WCOREDUMP**(*status*, /)

如果为该进程生成了核心转储，返回 True，否则返回 False。

此函数应当仅在*WIFSIGNALED()* 为真值时使用。

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFCONTINUED**(*status*)

如果一个已停止的子进程通过传送*SIGCONT* 获得恢复（如果该进程是从任务控制停止后再继续的）则返回 True，否则返回 False。

参见*WCONTINUED* 选项。

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFSTOPPED**(*status*)

如果进程是通过传送一个信号来停止的则返回 True，否则返回 False。

WIFSTOPPED() 只有在当*waitpid()* 调用是通过使用*WUNTRACED* 选项来完成或者当该进程正被追踪时 (参见 *ptrace(2)*) 才返回 True。

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFSIGNALED**(*status*)

如果进程是通过一个信号来终止的则返回 True ，否则返回 False。

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFEXITED**(*status*)

如果进程正常终止退出则返回 True，也就是说通过调用 exit() 或 _exit()，或者通过从 main() 返回；在其他情况下则返回 False。

Availability: Unix, not WASI, not Android, not iOS.

os.**WEXITSTATUS**(*status*)

返回进程退出状态。

此函数应当仅在*WIFEXITED()* 为真值时使用。

Availability: Unix, not WASI, not Android, not iOS.

os.**WSTOPSIG**(*status*)

返回导致进程停止的信号。

此函数应当仅在*WIFSTOPPED()* 为真值时使用。

Availability: Unix, not WASI, not Android, not iOS.

os.**WTERMSIG**(*status*)

返回导致进程终止的信号的编号。

此函数应当仅在*WIFSIGNALED()* 为真值时使用。

Availability: Unix, not WASI, not Android, not iOS.

16.1.8 调度器接口

这些函数控制操作系统如何为进程分配 CPU 时间。它们仅在某些 Unix 平台上可用。更多细节信息请查阅你所用 Unix 的指南页面。

Added in version 3.3.

以下调度策略如果被操作系统支持就会对外公开。

os.**SCHED_OTHER**

默认调度策略。

os.**SCHED_BATCH**

用于 CPU 密集型进程的调度策略，它会尽量为计算机中的其余任务保留交互性。

os.**SCHED_IDLE**

用于极低优先级的后台任务的调度策略。

os.**SCHED_SPORADIC**

用于偶发型服务程序的调度策略。

os.**SCHED_FIFO**

先进先出的调度策略。

os.**SCHED_RR**

循环式的调度策略。

os.**SCHED_RESET_ON_FORK**

此旗标可与任何其他调度策略进行 OR 运算。当带有此旗标的进程设置分叉时，其子进程的调度策略和优先级会被重置为默认值。

class os.**sched_param**(*sched_priority*)

这个类表示在*sched_setparam()*, *sched_setscheduler()* 和*sched_getparam()* 中使用的可修改调度形参。它属于不可变对象。

目前它只有一个可能的形参：

sched_priority

一个调度策略的调度优先级。

os.**sched_get_priority_min**(*policy*)

获取 *policy* 的最低优先级数值。*policy* 是以上调度策略常量之一。

os.**sched_get_priority_max**(*policy*)

获取 *policy* 的最高优先级数值。*policy* 是以上调度策略常量之一。

os.**sched_setscheduler**(*pid*, *policy*, *param*, /)

设置 PID 为 *pid* 的进程的调度策略。*pid* 为 0 指的是调用本方法的进程。*policy* 是以上调度策略常量之一。*param* 是一个*sched_param* 实例。

os.**sched_getscheduler**(*pid*, /)

返回 PID 为 *pid* 的进程的调度策略。*pid* 为 0 指的是调用本方法的进程。返回的结果是以上调度策略常量之一。

os.**sched_setparam**(*pid*, *param*, /)

设置 PID 为 *pid* 的进程的调度参数。*pid* 为 0 表示调用方过程。*param* 是一个*sched_param* 实例。

os.**sched_getparam**(*pid*, /)

返回 PID 为 *pid* 的进程的调度参数为一个*sched_param* 实例。*pid* 为 0 指的是调用本方法的进程。

os.**sched_rr_get_interval**(*pid*, /)

返回 PID 为 *pid* 的进程在时间片轮转调度下的时间片长度（单位为秒）。*pid* 为 0 指的是调用本方法的进程。

os.**sched_yield**()

主动让出 CPU。请参阅 *sched_yield(2)* 了解详情。

os.**sched_setaffinity**(*pid*, *mask*, /)

将 PID 为 *pid* 的进程（为零则为当前进程）限制到一组 CPU 上。*mask* 是整数的可迭代对象，表示应将进程限制在其中的一组 CPU。

os.**sched_getaffinity**(*pid*, /)

返回 PID 为 *pid* 的进程被限制到的那一组 CPU。

如果 *pid* 为零，则返回当前进程的调用方线程被限制到的那一组 CPU。

另请参阅*process_cpu_count()* 函数。

16.1.9 其他系统信息

os.**confstr**(*name*, /)

返回字符串格式的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 `confstr_names` 字典的键中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 指定的配置值未定义，返回 `None`。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 `confstr_names`，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

Availability: Unix.

os.**confstr_names**

字典，表示映射关系，为*confstr()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

os.**cpu_count**()

返回 **系统**中逻辑 CPU 的数量。如果无法确定则返回 None。

process_cpu_count() 函数可被用于获取 **当前进程**的调用方线程可以使用的逻辑 CPU 数量。

Added in version 3.4.

在 3.13 版本发生变更: 如果给出了 -X cpu_count 或设置了 PYTHON_CPU_COUNT，则*cpu_count()* 将返回被覆盖的值 *n*。

os.**getloadavg**()

返回系统运行队列中最近 1、5 和 15 分钟内的平均进程数。无法获得平均负载则抛出*OSError* 异常。

Availability: Unix.

os.**process_cpu_count**()

获取 **当前进程**的调用方线程可以使用的逻辑 CPU 数量。如果无法确定则返回 None。根据实际 CPU 的关联性它可能会小于*cpu_count()*。

cpu_count() 函数可被用于获取 **系统**中逻辑 CPU 的数量。

如果给出了 -X cpu_count 或设置了 PYTHON_CPU_COUNT，则*process_cpu_count()* 将返回被覆盖的值 *n*。

另请参阅*sched_getaffinity()* 函数。

Added in version 3.13.

os.**sysconf**(*name*, /)

返回整数格式的系统配置信息。如果 *name* 指定的配置值未定义，返回 -1。对*confstr()* 的 *name* 参数的注释在此处也适用。当前已知的配置名称在 sysconf_names 字典中提供。

Availability: Unix.

os.**sysconf_names**

字典，表示映射关系，为*sysconf()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

在 3.11 版本发生变更: 添加 'SC_MINSIGSTKSZ' 名称。

以下数据值用于支持对路径本身的操作。所有平台都有定义。

对路径的高级操作在*os.path* 模块中定义。

os.**curdir**

操作系统用来表示当前目录的常量字符串。在 Windows 和 POSIX 上是 '.'。在*os.path* 中也可用。

os.**pardir**

操作系统用来表示父目录的常量字符串。在 Windows 和 POSIX 上是 '..'。在*os.path* 中也可用。

os.**sep**

操作系统用来分隔路径不同部分的字符。在 POSIX 上是 '/'，在 Windows 上是是 '\\'。注意，仅了解它不足以能解析或连接路径，请使用*os.path.split()* 和*os.path.join()*，但它有时是有用的。在*os.path* 中也可用。

os.**altsep**

操作系统用来分隔路径不同部分的替代字符。如果仅存在一个分隔符，则为 None。在 sep 是反斜杠的 Windows 系统上，该值被设为 '/'。在*os.path* 中也可用。

os.**extsep**

分隔基本文件名与扩展名的字符，如 os.py 中的 '.'。在*os.path* 中也可用。

os.**pathsep**

操作系统通常用于分隔搜索路径（如 PATH）中不同部分的字符，如 POSIX 上是 ':'，Windows 上是 ';'。在*os.path* 中也可用。

os.**defpath**

在环境变量没有 'PATH' 键的情况下，*exec*p** and *spawn*p** 使用的默认搜索路径。在*os.path* 中也可用。

os.**linesep**

当前平台用于分隔（或终止）行的字符串。它可以是单个字符，如 POSIX 上是 '\n'，也可以是多个字符，如 Windows 上是 '\r\n'。在写入以文本模式（默认模式）打开的文件时，请不要使用 *os.linesep* 作为行终止符，请在所有平台上都使用一个 '\n' 代替。

os.**devnull**

空设备的文件路径。如 POSIX 上为 '/dev/null'，Windows 上为 'nul'。在*os.path* 中也可用。

os.**RTLD_LAZY**

os.**RTLD_NOW**

os.**RTLD_GLOBAL**

os.**RTLD_LOCAL**

os.**RTLD_NODELETE**

os.**RTLD_NOLOAD**

os.**RTLD_DEEPBIND**

setdlopenflags() 和*getdlopenflags()* 函数所使用的标志。请参阅 Unix 手册页 *dlopen(3)* 获取不同标志的含义。

Added in version 3.3.

16.1.10 随机数

os.**getrandom**(*size*, *flags=0*)

获得最多为 *size* 的随机字节。本函数返回的字节数可能少于请求的字节数。

这些字节可用于为用户空间的随机数生成器提供种子，或用于加密目的。

getrandom() 依赖于从设备驱动程序和其他环境噪声源收集的熵。不必要地读取大量数据将对使用 /dev/random 和 /dev/urandom 设备的其他用户产生负面影响。

flags 参数是一个位掩码，它可以包含零个或多个下列值的或运算结果：*os.GRND_RANDOM* 和*GRND_NONBLOCK*。

另请参阅 Linux getrandom() 手册页。

Availability: Linux >= 3.17.

Added in version 3.6.

os.**urandom**(*size*, */*)

返回大小为 *size* 的字节串，它是适合加密使用的随机字节。

本函数从系统指定的随机源获取随机字节。对于加密应用程序，返回的数据应有足够的不可预测性，尽管其确切的品质取决于操作系统的实现。

在 Linux 上，如果 getrandom() 系统调用可用，它将以阻塞模式使用：阻塞直到系统的 urandom 熵池初始化完毕（内核收集了 128 位熵）。原理请参阅 **PEP 524**。在 Linux 上，*getrandom()* 可以以

非阻塞模式（使用*GRND_NONBLOCK* 标志）获取随机字节，或者轮询直到系统的 urandom 熵池初始化完毕。

在类 Unix 系统上，随机字节是从 /dev/urandom 设备读取的。如果 /dev/urandom 设备不可用或不可读，则抛出*NotImplementedError* 异常。

在 Windows 上，它将使用 BCryptGenRandom()。

> **参见**
>
> *secrets* 模块提供了更高级的功能。所在平台会提供随机数生成器，有关其易于使用的接口，请参阅*random.SystemRandom*。

在 3.5 版本发生变更: 在 Linux 3.17 和更高版本上，现在使用 getrandom() 系统调用（如果可用）。在 OpenBSD 5.6 和更高版本上，现在使用 getentropy() C 函数。这些函数避免了使用内部文件描述符。

在 3.5.2 版本发生变更: 在 Linux 上，如果 getrandom() 系统调用阻塞（urandom 熵池尚未初始化完毕），则退回一步读取 /dev/urandom。

在 3.6 版本发生变更: 在 Linux 上，getrandom() 现在以阻塞模式使用，以提高安全性。

在 3.11 版本发生变更: 在 Windows 上，将使用 BCryptGenRandom() 而不是已被弃用的 CryptGenRandom()。

os.**GRND_NONBLOCK**

默认情况下，从 /dev/random 读取时，如果没有可用的随机字节，则*getrandom()* 会阻塞；从 /dev/urandom 读取时，如果熵池尚未初始化，则会阻塞。

如果设置了*GRND_NONBLOCK* 标志，则这些情况下*getrandom()* 不会阻塞，而是立即抛出*BlockingIOError* 异常。

Added in version 3.6.

os.**GRND_RANDOM**

如果设置了此标志位，那么将从 /dev/random 池而不是 /dev/urandom 池中提取随机字节。

Added in version 3.6.

16.2 io --- 处理流的核心工具

源代码: Lib/io.py

16.2.1 概述

io 模块提供了 Python 用于处理各种 I/O 类型的主要工具。三种主要的 I/O 类型分别为: *文本 I/O*, *二进制 I/O* 和 *原始 I/O*。这些是泛型类型，有很多种后端存储可以用在他们上面。一个隶属于任何这些类型的具体对象被称作*file object*。其他同类的术语还有 *流*和 *类文件对象*。

独立于其类别，每个具体流对象也将具有各种功能：它可以是只读，只写或读写。它还可以允许任意随机访问（向前或向后寻找任何位置），或仅允许顺序访问（例如在套接字或管道的情况下）。

所有流对提供给它们的数据类型都很敏感。例如将*str* 对象提供给二进制流的 write() 方法将引发*TypeError*。将*bytes* 对象提供给文本流的 write() 方法也是如此。

在 3.3 版本发生变更: 由于*IOError* 现在是*OSError* 的别名，因此用于引发*IOError* 的操作现在会引发*OSError* 。

文本 I/O

文本 I/O 预期并生成*str* 对象。这意味着，无论何时后台存储是由字节组成的（例如在文件的情况下），数据的编码和解码都是透明的，并且可以选择转换特定于平台的换行符。

创建文本流的最简单方式是通过*open()*，并可选择指定编码格式:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

内存中文本流也可以作为*StringIO* 对象使用:

```
f = io.StringIO("some initial text data")
```

TextIOBase 的文档中详细描述了文本流的 API

二进制 I/O

二进制 I/O（也称为缓冲 I/O）预期*bytes-like objects* 并生成*bytes* 对象。不执行编码、解码或换行转换。这种类型的流可以用于所有类型的非文本数据，并且还可以在需要手动控制文本数据的处理时使用。

创建二进制流的最简单方式是通过*open()* 并在模式字符串中使用 'b':

```
f = open("myfile.jpg", "rb")
```

内存中二进制流也可以作为*BytesIO* 对象使用:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

BufferedIOBase 的文档中详细描述了二进制流 API。

其他库模块可以提供额外的方式来创建文本或二进制流。参见*socket.socket.makefile()* 的示例。

原始 I/O

原始 I/O（也称为 *非缓冲 I/O*）通常用作二进制和文本流的低级构建块。用户代码直接操作原始流的用法非常罕见。不过，可以通过在禁用缓冲的情况下以二进制模式打开文件来创建原始流:

```
f = open("myfile.jpg", "rb", buffering=0)
```

RawIOBase 的文档中详细描述了原始流的 API

16.2.2 文本编码格式

TextIOWrapper 和*open()* 的默认编码格式取决于语言区域的设置 (*locale.getencoding()*)。

但是，很多开发者在打开以 UTF-8 编码的文本文件 (例如 JSON, TOML, Markdown 等等...) 时会忘记指定编码格式，因为大多数 Unix 平台默认使用 UTF-8 语言区域。这会导致各种错误因为大多数 Windows 用户的语言区域编码格式并不是 UTF-8。例如:

```
# 当文件中有非 ASCII 字符时可能无法在 Windows 下使用。
with open("README.md") as f:
    long_description = f.read()
```

为此，强烈建议你在打开文本文件时显式地指定编码格式。如果你想要使用 UTF-8，就传入 `encoding="utf-8"`。要使用当前语言区域的编码格式，`encoding="locale"` 自 Python 3.10 开始已被支持。

参见

Python UTF-8 模式
Python UTF-8 模式可以用来将默认编码格式由语言区域所确定的编码格式改为 UTF-8。

PEP 686
Python 3.15 将把*Python UTF-8 模式* 设为默认值。

选择性的 EncodingWarning

Added in version 3.10: 请参阅 **PEP 597** 了解详情。

要找出哪里使用了默认语言区域的编码格式，你可以启用 `-X warn_default_encoding` 命令行选项或设置 `PYTHONWARNDEFAULTENCODING` 环境变量，这将在使用默认编码格式时发出*EncodingWarning*。

如果你提供了使用*open()* 或*TextIOWrapper* 的 API 并将 `encoding=None` 作为形参传入，你可以使用*text_encoding()* 以便 API 的调用方在没有传入 `encoding` 的时候将发出*EncodingWarning*。但是，对于新的 API 请考虑默认就使用 UTF-8 (即 `encoding="utf-8"`)。

16.2.3 高阶模块接口

io.DEFAULT_BUFFER_SIZE
包含模块缓冲 I/O 类使用的默认缓冲区大小的 int。在可能的情况下*open()* 将使用文件的 blksize (由*os.stat()* 获得)。

io.open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)
这是内置的*open()* 函数的别名。

此函数将引发一个*审计事件* `open` 并附带参数 *path*, *mode* 和 *flags*。*mode* 和 *flags* 参数可能在原始调用的基础上修改或推断得到。

io.open_code(*path*)
以 `'rb'` 模式打开提供的文件。如果目的是将文件内容做为可执行代码，则应使用此函数。

path 应当为*str* 类型并且是一个绝对路径。

该函数的行为可通过先期调用 `PyFile_SetOpenCodeHook()` 来重写。不过，假如 *path* 为*str* 类型并且是一个绝对路径，`open_code(path)` 的行为应当总是与 `open(path, 'rb')` 一致。重写行为的目的是为了给文件附加额外的验证或预处理。

Added in version 3.8.

io.text_encoding(*encoding*, *stacklevel=2*, */*)
这是一个针对使用*open()* 或*TextIOWrapper* 的可调用对象的辅助函数并且具有 `encoding=None` 形参。

如果 *encoding* 不为 `None` 则将其返回。还其他情况下，它将根据是否启用了*UTF-8 模式* 返回 `"locale"` 或 `"utf-8"`。

如果*sys.flags.warn_default_encoding* 为真值且 *encoding* 为 `None` 则此函数会发出*EncodingWarning*。*stacklevel* 指明警告在哪里发出。例如:

```
def read_text(path, encoding=None):
    encoding = io.text_encoding(encoding)  # stacklevel=2
    with open(path, encoding) as f:
        return f.read()
```

在这个例子中，将为 `read_text()` 的调用方发出*EncodingWarning*。

请参阅*文本编码格式* 了解更多信息。

Added in version 3.10.

在 3.11 版本发生变更: 当启用 UTF-8 模式且 *encoding* 为 `None` 时*text_encoding()* 将返回"utf-8"。

exception io.BlockingIOError
这是内置的*BlockingIOError* 异常的兼容性别名。

exception io.**UnsupportedOperation**

在流上调用不支持的操作时引发的继承*OSError* 和*ValueError* 的异常。

参见

sys
包含标准 IO 流: *sys.stdin*, *sys.stdout* 和*sys.stderr* 。

16.2.4 类的层次结构

I/O 流被安排为按类的层次结构实现。首先是抽象基类 (ABC)，用于指定流的各种类别，然后是提供标准流实现的具体类。

备注

这些抽象基类还提供了一些方法的默认实现，以帮助实现具体的流类。例如，*BufferedIOBase* 提供了 `readinto()` 和 `readline()` 的未优化实现。

I/O 层次结构的顶部是抽象基类*IOBase* 。它定义了流的基本接口。但是请注意，对流的读取和写入之间没有分离。如果实现不支持指定的操作，则会引发*UnsupportedOperation* 。

抽象基类*RawIOBase* 是*IOBase* 的子类。它负责将字节读取和写入流中。*RawIOBase* 的子类*FileIO* 提供计算机文件系统中文件的接口。

抽象基类*BufferedIOBase* 扩展了*IOBase*。它能处理原始二进制流 (*RawIOBase*) 上的缓冲。它的子类*BufferedWriter*, *BufferedReader* 和*BufferedRWPair* 分别缓冲可写、可读以及同时可读写的原始二进制流。*BufferedRandom* 提供了带缓冲的可随机访问接口。*BufferedIOBase* 的另一子类*BytesIO* 是内存中字节流。

抽象基类*TextIOBase* 继承了*IOBase* 。它处理可表示文本的流，并处理字符串的编码和解码。类*TextIOWrapper* 继承了*TextIOBase* ，是原始缓冲流（*BufferedIOBase* ）的缓冲文本接口。最后，*StringIO* 是文本的内存流。

参数名不是规范的一部分，只有*open()* 的参数才用作关键字参数。

下表总结了抽象基类提供的*io* 模块:

抽象基类	继承	抽象方法	Mixin 方法和属性
IOBase		`fileno`, `seek`, 和 `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable` 和 `writelines`
RawIOBase	*IOBase*	`readinto` 和 `write`	继承*IOBase* 方法, `read`, 和 `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1`, 和 `write`	继承*IOBase* 方法, `readinto`, 和 `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline`, 和 `write`	继承*IOBase* 方法, `encoding`, `errors`, 和 `newlines`

I/O 基类

class io.**IOBase**

所有 I/O 类的抽象基类。

此类为许多方法提供了空的抽象实现，派生类可以有选择地重写。默认实现代表一个无法读取、写入或查找的文件。

尽管*IOBase* 并未声明 `read()` 或 `write()`，因为它们的签名会有所不同，但是实现和客户端应该将这些方法视为接口的一部分。此外，当调用不支持的操作时实现可能会引发*ValueError* (或*UnsupportedOperation*)。

从文件读取或写入文件的二进制数据的基本类型为*bytes* 。其他*bytes-like objects* 也可以作为方法参数。文本 I/O 类使用*str* 数据。

请注意，在关闭的流上调用任何方法（甚至查询）都是未定义的（undefined）。在这种情况下，实现可能会引发*ValueError* 。

IOBase （及其子类）支持迭代器协议，这意味着可以迭代*IOBase* 对象以产生流中的行。根据流是二进制流（产生字节）还是文本流（产生字符串），行的定义略有不同。请参见下面的*readline()* 。

IOBase 也是一个上下文管理器，因此支持 `with` 语句。在这个示例中，*file* 将在 `with` 语句块执行完成之后被关闭 --- 即使是发生了异常:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase 提供以下数据属性和方法:

close()

刷新并关闭此流。如果文件已经关闭，则此方法无效。文件关闭后，对文件的任何操作（例如读取或写入）都会引发*ValueError* 。

为方便起见，允许多次调用此方法。但是，只有第一个调用才会生效。

closed

如果流已关闭，则返回 True。

fileno()

返回流的底层文件描述符（整数）---如果存在。如果 IO 对象不使用文件描述符，则会引发*OSError* 。

flush()

刷新流的写入缓冲区（如果适用）。这对只读和非阻塞流不起作用。

isatty()

如果流是交互式的（即连接到终端/tty 设备），则返回 `True` 。

readable()

如果可以读取流则返回 `True`。如果返回 `False`，则 `read()` 将引发*OSError*。

readline(*size=-1*, */*)

从流中读取并返回一行。如果指定了 *size*，将至多读取 *size* 个字节。

对于二进制文件行结束符总是 `b'\n'`；对于文本文件，可以用将 *newline* 参数传给*open()* 的方式来选择要识别的行结束符。

readlines(*hint=-1*, */*)

从流中读取并返回包含多行的列表。可以指定 *hint* 来控制要读取的行数：如果（以字节/字符数表示的）所有行的总大小超出了 *hint* 则将不会读取更多的行。

`0` 或更小的 *hint* 值以及 `None`，会被视为没有 hint。

请注意，在不调用 `file.readlines()` 的情况下使用 `for line in file: ...` 来遍历文件对象已经成为可能。

seek(*offset*, *whence=os.SEEK_SET*, /)

将流位置修改到给定的字节 *offset*，它将相对于 *whence* 所指明的位置进行解析，并返回新的绝对位置。*whence* 的可用值有:

- *os.SEEK_SET* 或 0 -- 流的开头（默认值；*offset* 应为零或正值
- *os.SEEK_CUR* 或 1 -- 当前流位置；*offset* 可以为负值
- *os.SEEK_END* 或 2 -- 流的末尾；*offset* 通常为负值

Added in version 3.1: SEEK_* 常量。

Added in version 3.3: 某些操作系统还可支持其他的值，如*os.SEEK_HOLE* 或*os.SEEK_DATA*。特定文件的可用值还会取决于它是以文本还是二进制模式打开。

seekable()

如果流支持随机访问则返回 True。如为 False，则*seek()*, *tell()* 和*truncate()* 将引发*OSError*。

tell()

返回当前流的位置。

truncate(*size=None*, /)

将流的大小调整为给定的 *size* 个字节（如果未指定 *size* 则调整至当前位置）。当前的流位置不变。这个调整操作可扩展或减小当前文件大小。在扩展的情况下，新文件区域的内容取决于具体平台（在大多数系统上，额外的字节会填充为零）。返回新的文件大小。

在 3.5 版本发生变更: 现在 Windows 在扩展时将文件填充为零。

writable()

如果流支持写入则返回 True。如为 False，则 write() 和*truncate()* 将引发*OSError*。

writelines(*lines*, /)

将行列表写入到流。不会添加行分隔符，因此通常所提供的每一行都带有末尾行分隔符。

__del__()

为对象销毁进行准备。*IOBase* 提供了此方法的默认实现，该实现会调用实例的*close()* 方法。

class io.**RawIOBase**

原始二进制流的基类。它继承自*IOBase*。

原始二进制流通常会提供对下层 OS 设备或 API 的低层级访问，而不是尝试将其封装到高层级的基元中（此功能是在更高层级的缓冲二进制流和文本流中实现的，将在下文中描述）。

RawIOBase 在*IOBase* 的现有成员以外还提供了下列方法:

read(*size=-1*, /)

从对象中读取 *size* 个字节并将其返回。作为一个便捷选项，如果 *size* 未指定或为 -1，则返回所有字节直到 EOF。在其他情况下，仅会执行一次系统调用。如果操作系统调用返回字节数少于 *size* 则此方法也可能返回少于 *size* 个字节。

如果返回 0 个字节而 *size* 不为零 0，这表明到达文件末尾。如果处于非阻塞模式并且没有更多字节可用，则返回 None。

默认实现会转至*readall()* 和*readinto()*。

readall()

从流中读取并返回所有字节直到 EOF，如有必要将对流执行多次调用。

readinto(*b*, /)

将字节数据读入预先分配的可写*bytes-like object* *b*，并返回所读取的字节数。例如，*b* 可以是一个*bytearray*。如果对象处理非阻塞模式并且没有更多字节可用，则返回 None。

write(*b*, /)

将给定的*bytes-like object* *b* 写入到下层的原始流，并返回所写入的字节数。这可以少于 *b* 的总字节数，具体取决于下层原始流的设定，特别是如果它处于非阻塞模式的话。如果原始流设为非阻塞并且不能真正向其写入单个字节时则返回 None。调用者可以在此方法返回后释放或改变 *b*，因此该实现应该仅在方法调用期间访问 *b*。

class io.**BufferedIOBase**

支持某种缓冲的二进制流的基类。它继承自*IOBase*。

与*RawIOBase* 的主要差别在于*read()*, *readinto()* 和*write()* 等方法将（分别）尝试按照要求读取尽可能多的输入或是耗尽所有给定的输出，其代价是可能会执行一次以上的系统调用。

除此之外，那些方法还可能引发*BlockingIOError*，如果下层的原始数据流处于非阻塞模式并且无法接受或给出足够数据的话；不同于对应的*RawIOBase* 方法，它们将永远不会返回 None。

并且，*read()* 方法也没有转向*readinto()* 的默认实现。

典型的*BufferedIOBase* 实现不应当继承自*RawIOBase* 实现，而要包装一个该实现，正如*BufferedWriter* 和*BufferedReader* 所做的那样。

BufferedIOBase 在*IOBase* 的现有成员以外还提供或重写了下列数据属性和方法:

raw

由*BufferedIOBase* 处理的下层原始流 (*RawIOBase* 的实例)。它不是*BufferedIOBase* API 的组成部分并且不存在于某些实现中。

detach()

从缓冲区分离出下层原始流并将其返回。

在原始流被分离之后，缓冲区将处于不可用的状态。

某些缓冲区例如*BytesIO* 并无可从此方法返回的单独原始流的概念。它们将会引发*UnsupportedOperation*。

Added in version 3.1.

read(*size=-1*, /)

读取并返回最多 *size* 个字节。如果此参数被省略、为 None 或为负值，则读取并返回所有数据直到 EOF。如果流已经到达 EOF 则返回一个空的*bytes* 对象。

如果此参数为正值，并且下层原始流不可交互，则可能发起多个原始读取以满足字节计数（直至先遇到 EOF）。但对于可交互原始流，则将至多发起一个原始读取，并且简短的结果并不意味着已到达 EOF。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

read1(*size=-1*, /)

通过至多一次对下层流的*read()* (或*readinto()*) 方法的调用读取并返回至多 *size* 个字节。这适用于在*BufferedIOBase* 对象之上实现你自己的缓冲区的情况。

如果 *size* 为 -1 (默认值)，则返回任意数量的字节（多于零字节，除非已到达 EOF）。

readinto(*b*, /)

将字节数据读入预先分配的可写*bytes-like object* *b* 并返回所读取的字节数。例如，*b* 可以是一个*bytearray*。

类似于*read()*，可能对下层原始流发起多次读取，除非后者为交互式。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

readinto1(*b*, /)

将字节数据读入预先分配的可写*bytes-like object* *b*，其中至多使用一次对下层原始流*read()* (或*readinto()*) 方法的调用。返回所读取的字节数。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

Added in version 3.5.

write(*b*, /)

写入给定的*bytes-like object* *b*，并返回写入的字节数 (总是等于 *b* 的字节长度，因为如果写入失败则会引发*OSError*)。根据具体实现的不同，这些字节可能被实际写入下层流，或是出于运行效率和冗余等考虑而暂存于缓冲区。

当处于非阻塞模式时，如果需要将数据写入原始流但它无法在不阻塞的情况下接受所有数据则将引发*BlockingIOError*。

调用者可能会在此方法返回后释放或改变 *b*，因此该实现应当仅在方法调用期间访问 *b*。

原始文件 I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

代码一个包含字节数据的 OS 层级文件的原始二进制流。它继承自*RawIOBase*。

name 可以是以下两项之一:

- 代表将被打开的文件路径的字符串或*bytes* 对象。在此情况下 closefd 必须为 `True` (默认值) 否则将会引发异常。
- 代表一个现有 OS 层级文件描述符的号码的整数，作为结果的*FileIO* 对象将可访问该文件。当 FileIO 对象被关闭时此 fd 也将被关闭，除非 *closefd* 设为 `False`。

mode 可以为 `'r'`, `'w'`, `'x'` 或 `'a'` 分别表示读取（默认模式）、写入、独占新建或添加。如果以写入或添加模式打开的文件不存在将自动新建；当以写入模式打开时文件将先清空。以新建模式打开时如果文件已存在则将引发*FileExistsError*。以新建模式打开文件也意味着要写入，因此该模式的行为与 `'w'` 类似。在模式中附带 `'+'` 将允许同时读取和写入。

该类的*read()* (当附带为正值的参数调用时), *readinto()* 和*write()* 方法将只执行一次系统调用。

可以通过传入一个可调用对象作为 *opener* 来使用自定义文件打开器。然后通过调用 *opener* 并传入 (*name*, *flags*) 来获取文件对象所对应的下层文件描述符。*opener* 必须返回一个打开文件描述符（传入*os.open* 作为 *opener* 的结果在功能上将与传入 `None` 类似）。

新创建的文件是*不可继承的*。

有关 opener 参数的示例，请参见内置函数*open()* 。

在 3.3 版本发生变更: 增加了 *opener* 参数。增加了 `'x'` 模式。

在 3.4 版本发生变更: 文件现在禁止继承。

FileIO 在继承自*RawIOBase* 和*IOBase* 的现有成员以外还提供了以下数据属性和方法:

mode

构造函数中给定的模式。

name

文件名。当构造函数中没有给定名称时，这是文件的文件描述符。

缓冲流

相比原始 I/O，缓冲 I/O 流提供了针对 I/O 设备的更高层级接口。

class io.**BytesIO**(*initial_bytes=b''*)

一个使用内存字节缓冲区的二进制流。它继承自*BufferedIOBase*。当*close()* 方法被调用时缓冲区将被丢弃。

可选参数 *initial_bytes* 是一个包含初始数据的*bytes-like object*。

BytesIO 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重写了下列方法:

getbuffer()

返回一个对应于缓冲区内容的可读写视图而不必拷贝其数据。此外，改变视图将透明地更新缓冲区内容:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

备注

只要视图保持存在，*BytesIO* 对象就无法被改变大小或关闭。

Added in version 3.2.

getvalue()

返回包含整个缓冲区内容的*bytes*。

read1(*size=-1*, */*)

在*BytesIO* 中，这与*read()* 相同。

在 3.7 版本发生变更: *size* 参数现在是可选的。

readinto1(*b*, */*)

在*BytesIO* 中，这与*readinto()* 相同。

Added in version 3.5.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

一个提供对可读、不可定位的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流。它继承自*BufferedIOBase*。

当从此对象读取数据时，可能会从下层原始流请求更大量的数据，并存放到内部缓冲区中。接下来可以在后续读取时直接返回缓冲数据。

根据给定的可读 *raw* 流和 *buffer_size* 创建*BufferedReader* 的构造器。如果省略 *buffer_size*，则会使用*DEFAULT_BUFFER_SIZE*。

BufferedReader 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重写了下列方法:

peek(*size=0*, */*)

从流返回字节数据而不前移位置。完成此调用将至多读取一次原始流。返回的字节数量可能少于或多于请求的数量。

read(*size=-1*, */*)

读取并返回 *size* 个字节，如果 *size* 未给定或为负值，则读取至 EOF 或是在非阻塞模式下读取调用将会阻塞。

read1(*size=-1*, */*)

在原始流上通过单次调用读取并返回至多 *size* 个字节。如果至少缓冲了一个字节，则只返回缓冲的字节。在其他情况下，将执行一次原始流读取。

在 3.7 版本发生变更: *size* 参数现在是可选的。

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

一个提供对可写、不可定位的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流。它继承自*BufferedIOBase*。

当写入到此对象时，数据通常会被放入到内部缓冲区中。缓冲区将在满足某些条件的情况下被写到下层的*RawIOBase* 对象，包括:

- 当缓冲区对于所有挂起数据而言太小时；
- 当*flush()* 被调用时；
- 当*seek()* 被请求时（针对*BufferedRandom* 对象）；

- 当*BufferedWriter*对象被关闭或销毁时。

该构造器会为给定的可写 *raw* 流创建一个*BufferedWriter*。如果未给定 *buffer_size*，则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedWriter 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重写了下列方法:

flush()

将缓冲区中保存的字节数据强制放入原始流。如果原始流发生阻塞则应当引发*BlockingIOError*。

write(*b*, /)

写入*bytes-like object* *b* 并返回写入的字节数。当处于非阻塞模式时，如果缓冲区需要被写入但原始流发生阻塞则将引发*BlockingIOError*。

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

一个提供对不可定位的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流。它继承自*BufferedReader* 和*BufferedWriter*。

该构造器会为在第一个参数中给定的可查找原始流创建一个读取器和写入器。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedRandom 能做到*BufferedReader* 或*BufferedWriter* 所能做的任何事。此外，还会确保实现*seek()* 和*tell()*。

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*, /)

一个提供对两个不可定位的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流 --- 一个可读，另一个可写。它继承自*BufferedIOBase*。

reader 和 *writer* 分别是可读和可写的*RawIOBase* 对象。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedRWPair 实现了*BufferedIOBase* 的所有方法，但*detach()* 除外，调用该方法将引发*UnsupportedOperation*。

> **警告**
>
> *BufferedRWPair* 不会尝试同步访问其下层的原始流。你不应当将传给它与读取器和写入器相同的对象；而要改用*BufferedRandom*。

文本 I/O

class io.**TextIOBase**

文本流的基类。该类提供了基于字符和行的流 I/O 的接口。它继承自*IOBase*。

TextIOBase 在来自*IOBase* 的成员以外还提供或重写了以下数据属性和方法:

encoding

用于将流的字节串解码为字符串以及将字符串编码为字节串的编码格式名称。

errors

解码器或编码器的错误设置。

newlines

一个字符串、字符串元组或者 None，表示目前已经转写的新行。根据具体实现和初始构造器旗标的不同，此属性或许会不可用。

buffer

由*TextIOBase* 处理的下层二进制缓冲区（为一个*BufferedIOBase* 的实例）。它不是*TextIOBase* API 的组成部分并且不存在于某些实现中。

detach()

从*TextIOBase* 分离出下层二进制缓冲区并将其返回。

在下层缓冲区被分离后，*TextIOBase* 将处于不可用的状态。

某些*TextIOBase* 的实现，例如*StringIO* 可能并无下层缓冲区的概念，因此调用此方法将引发*UnsupportedOperation*。

Added in version 3.1.

read(*size=-1*, */*)

从流中读取至多 *size* 个字符并以单个*str* 的形式返回。如果 *size* 为负值或 None，则读取至 EOF。

readline(*size=-1*, */*)

读取至换行符或 EOF 并返回单个*str*。如果流已经到达 EOF，则将返回一个空字符串。, an empty string is returned.

如果指定了 *size* ，最多将读取 *size* 个字符。

seek(*offset*, *whence=SEEK_SET*, */*)

将流位置改为给定的 *offset*。具体行为取决于 *whence* 形参。*whence* 的默认值为 SEEK_SET。

- SEEK_SET 或 0: 从流的起始位置开始查找（默认值）；*offset* 必须为*TextIOBase.tell()* 所返回的数值或为零。任何其他 *offset* 值都将导致未定义的行为。
- SEEK_CUR 或 1: ”查找” 到当前位置；*offset* 必须为零，表示无操作（所有其他值均不受支持）。
- SEEK_END 或 2: 查找到流的末尾；*offset* 必须为零（所有其他值均不受支持）。

以不透明数字形式返回新的绝对位置。

Added in version 3.1: SEEK_* 常量。

tell()

以不透明数字形式返回当前流的位置。该数字通常并不代表下层二进制存储中对应的字节数。

write(*s*, */*)

将字符串 *s* 写入到流并返回写入的字符数。

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

一个提供对*BufferedIOBase* 缓冲二进制流的高层级访问的缓冲文本流。它继承自*TextIOBase*。

encoding 给出将被用于编码或解码流的编码格式。该参数值默认为*locale.getencoding()*。可以使用 encoding="locale" 来显式地指定当前语言区域的编码格式。详情参见文本编码格式。

errors 是一个可选的字符串，它指明编码格式和编码格式错误的处理方式。传入 'strict' 将在出现编码格式错误时引发*ValueError* (默认值 None 具有相同的效果)，传入 'ignore' 将忽略错误。(请注意忽略编码格式错误会导致数据丢失。) 'replace' 会在出现错误数据时插入一个替换标记 (例如 '?')。'backslashreplace' 将把错误数据替换为一个反斜杠转义序列。在写入时，还可以使用 'xmlcharrefreplace' (替换为适当的 XML 字符引用) 或 'namereplace' (替换为 \N{...} 转义序列)。任何其他通过*codecs.register_error()* 注册的错误处理方式名称也可以被接受。

newline 控制行结束符处理方式。它可以为 None, '', '\n', '\r' 和 '\r\n'。其工作原理如下:

- 当从流读取输入时，如果 *newline* 为 None，则将启用*universal newlines* 模式。输入中的行结束符可以为 '\n', '\r' 或 '\r\n'，在返回给调用者之前它们会被统一转写为 '\n'。如果 *newline* 为 ''，也会启用通用换行模式，但行结束符会不加转写即返回给调用者。如果 *newline* 具有任何其他合法的值，则输入行将仅由给定的字符串结束，并且行结束符会不加转写即返回给调用者。
- 将输出写入流时，如果 *newline* 为 None，则写入的任何 '\n' 字符都将转换为系统默认行分隔符*os.linesep*。如果 *newline* 是 '' 或 '\n'，则不进行翻译。如果 *newline* 是任何其他合法值，则写入的任何 '\n' 字符将被转换为给定的字符串。

如果 *line_buffering* 为 `True`，则当一个写入调用包含换行或回车符时将会应用*flush()*。

如果 *write_through* 为 `True`，则对*write()* 的调用会确保不被缓冲：在*TextIOWrapper* 对象上写入的任何数据会立即交给其下层的 *buffer* 来处理。

在 3.3 版本发生变更: 已添加 *write_through* 参数

在 3.3 版本发生变更: 默认的 *encoding* 现在将为 `locale.getpreferredencoding(False)` 而非 `locale.getpreferredencoding()`。不要使用*locale.setlocale()* 来临时改变区域编码格式，要使用当前区域编码格式而不是用户的首选编码格式。

在 3.10 版本发生变更: *encoding* 参数现在支持 `"locale"` 作为编码格式名称。

TextIOWrapper 在继承自*TextIOBase* 和*IOBase* 的现有成员以外还提供了以下数据属性和方法:

line_buffering

是否启用行缓冲。

write_through

写入是否要立即传给下层的二进制缓冲。

Added in version 3.7.

reconfigure(*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=None*, *write_through=None*)

使用 *encoding*, *errors*, *newline*, *line_buffering* 和 *write_through* 的新设置来重新配置此文本流。

未指定的形参将保留当前设定，例外情况是当指定了 *encoding* 但未指定 *errors* 时将会使用 `errors='strict'`。

如果已经有数据从流中被读取则将无法再改变编码格式或行结束符。另一方面，在写入数据之后再改变编码格式则是可以的。

此方法会在设置新的形参之前执行隐式的流刷新。

Added in version 3.7.

在 3.11 版本发生变更: 此方法支持 `encoding="locale"` 选项。

seek(*cookie*, *whence=os.SEEK_SET*, */*)

设置流位置。以*int* 的形式返回新的流位置。

支持四种操作，由下列参数组合给出:

- `seek(0, SEEK_SET)`: 回退到流的开头。
- `seek(cookie, SEEK_SET)`: 恢复之前的位置；*cookie* **必须是**由*tell()* 返回的数字。
- `seek(0, SEEK_END)`: 快进到流的末尾。
- `seek(0, SEEK_CUR)`: 保持当前流位置不变。

任何其他参数组合均无效，并可能引发异常。

> **参见**
>
> *os.SEEK_SET*, *os.SEEK_CUR* 和*os.SEEK_END*。

tell()

以不透明数字的形式返回流位置。`tell()` 的返回值可以作为*seek()* 的输入，以恢复之前的流位置。

class io.**StringIO**(*initial_value=''*, *newline='\n'*)

一个使用内存文本缓冲区的文本流。它继承自*TextIOBase*。

当*close()* 方法被调用时将会丢弃文本缓冲区。

缓冲区的初始值可通过提供 *initial_value* 来设置。如果启用了换行符转写，换行符将以与*write()* 相同的方式进行编码。流将被定位到缓冲区的起点，这模拟了以 `w+` 模式打开一个现有文件的操作，

使其准备好从头开始立即写入或是将要覆盖初始值的写入。要模拟以 `a+` 模式打开一个文件准备好追加内容，请使用 `f.seek(0, io.SEEK_END)` 来将流重新定位到缓冲区的末尾。

newline 参数的规则与 *TextIOWrapper* 所用的一致，不同之处在于当将输出写入到流时，如果 *newline* 为 `None`，则在所有平台上换行符都会被写入为 `\n`。

StringIO 在继承自 *TextIOBase* 和 *IOBase* 的现有成员以外还提供了以下方法:

getvalue()

返回一个包含缓冲区全部内容的 *str*。换行符会以与 *read()* 相同的方式被编码，但是流位置不会改变。

用法示例:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# 提取文件内容 -- 这将为
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# 关闭对象并丢弃内存缓冲区 --
# 现在 .getvalue() 将引发一个异常。
output.close()
```

class io.**IncrementalNewlineDecoder**

用于在 *universal newlines* 模式下解码换行符的辅助编解码器。它继承自 *codecs.IncrementalDecoder*。

16.2.5 性能

本节讨论所提供的具体 I/O 实现的性能。

二进制 I/O

即使在用户请求单个字节时，也只读取和写入大块数据。通过该方法，缓冲 I/O 隐藏了操作系统调用和执行无缓冲 I/O 例程时的任何低效性。增益取决于操作系统和执行的 I/O 类型。例如，在某些现代操作系统上（例如 Linux），无缓冲磁盘 I/O 可以与缓冲 I/O 一样快。但最重要的是，无论平台和支持设备如何，缓冲 I/O 都能提供可预测的性能。因此，对于二进制数据，应首选使用缓冲的 I/O 而不是未缓冲的 I/O 。

文本 I/O

二进制存储（如文件）上的文本 I/O 比同一存储上的二进制 I/O 慢得多，因为它需要使用字符编解码器在 unicode 和二进制数据之间进行转换。在处理大量文本数据（如大型日志文件）时这种情况会非常明显。此外，由于使用了重构算法因而 *tell()* 和 *seek()* 的速度都相当慢。

StringIO 是原生的内存 Unicode 容器，速度与 *BytesIO* 相似。

多线程

FileIO 对象在它们封装的操作系统调用 (如 Unix 下的 *read(2)*) 是线程安全的情况下也是线程安全的。

二进制缓冲对象（例如 *BufferedReader*, *BufferedWriter*, *BufferedRandom* 和 *BufferedRWPair*）使用锁来保护其内部结构；因此，可以安全地一次从多个线程中调用它们。

TextIOWrapper 对象不再是线程安全的。

可重入性

二进制缓冲对象（*BufferedReader*，*BufferedWriter*，*BufferedRandom* 和 *BufferedRWPair* 的实例）不是可重入的。虽然在正常情况下不会发生可重入调用，但仍可能会在 *signal* 处理程序执行 I/O 时产生。如果线程尝试重入已经访问的缓冲对象，则会引发 *RuntimeError*。注意，这并不禁止其他线程进入缓冲对象。

上面的代码将显式地扩展到文本文件，因为 *open()* 函数将把缓冲的对象包装在 *TextIOWrapper* 中。这包括标准流因而也会影响内置的 *print()* 函数。

16.3 time --- 时间的访问和转换

该模块提供了各种与时间相关的函数。相关功能还可以参阅 *datetime* 和 *calendar* 模块。

尽管所有平台皆可使用此模块，但模块内的函数并非所有平台都可用。此模块中定义的大多数函数的实现都是调用其所在平台的 C 语言库的同名函数。因为这些函数的语义可能因平台而异，所以使用时最好查阅对应平台的相关文档。

下面是一些术语和惯例的解释.

- *epoch* 是起始的时间点，即 `time.gmtime(0)` 的返回值。这在所有平台上都是 1970-01-01, 00:00:00 (UTC)。
- 术语 *纪元秒数* 是指自 epoch （纪元）时间点以来经过的总秒数，通常不包括 闰秒。在所有符合 POSIX 标准的平台上，闰秒都不会记录在总秒数中。
- 此模块中的函数可能无法处理 *epoch* 之前或遥远未来的日期和时间。“遥远未来”的分界点是由 C 库确定的；对于 32 位系统，它通常是在 2038 年。
- 函数 *strptime()* 在接收到 `%y` 格式代码时可以解析使用 2 位数表示的年份。当解析 2 位数年份时，函数会按照 POSIX 和 ISO C 标准进行年份转换：数值 69--99 被映射为 1969--1999；数值 0--68 被映射为 2000--2068。
- UTC 即 Coordinated Universal Time，它取代 Greenwich Mean Time 即 GMT 作为国际时间计量的基准。UTC 是个错误的缩写但它遵循了更早的语言中立的时间标准命名方案如 UT0, UT1 和 UT2。
- DST 是夏令时（Daylight Saving Time）的缩写，在一年的某一段时间中将当地时间调整（通常）一小时。DST 的规则非常神奇（由当地法律确定），并且每年的起止时间都不同。C 语言库中有一个表格，记录了各地的夏令时规则（实际上，为了灵活性，C 语言库通常是从某个系统文件中读取这张表）。从这个角度而言，这张表是夏令时规则的唯一权威真理。
- 由于平台限制，各种实时函数的精度可能低于其值或参数所要求（或给定）的精度。例如，在大多数 Unix 系统上，时钟频率仅为每秒 50 或 100 次。
- 另一方面，*time()* 和 *sleep()* 的精度优于它们的 Unix 等价物：时间表示为浮点数，*time()* 返回可用的最准确时间 (如有可能将使用 Unix `gettimeofday()`)，并且 *sleep()* 将接受带有非零小数部分的时间 (如有可能将使用 Unix `select()` 来实现此功能)。
- 时间值由 *gmtime()*, *localtime()* 和 *strptime()* 返回，并被 *asctime()*, *mktime()* 和 *strftime()* 接受，是一个 9 个整数的序列。*gmtime()*，*localtime()* 和 *strptime()* 的返回值还提供各个字段的属性名称。

  请参阅 *struct_time* 以获取这些对象的描述。

  在 3.3 版本发生变更: 当平台支持相应的 `struct tm` 成员时 *struct_time* 类型将被扩展以提供 *tm_gmtoff* 和 *tm_zone* 属性。

  在 3.6 版本发生变更: *struct_time* 的属性 *tm_gmtoff* 和 *tm_zone* 现在可在所有平台上使用。
- 使用以下函数在时间表示之间进行转换：

从	到	使用
自纪元以来的秒数	UTC 的*struct_time*	*gmtime()*
自纪元以来的秒数	本地时间的*struct_time*	*localtime()*
UTC 的*struct_time*	自纪元以来的秒数	*calendar.timegm()*
本地时间的*struct_time*	自纪元以来的秒数	*mktime()*

16.3.1 函数

time.**asctime**([*t*])

转换由*gmtime()* 或*localtime()* 所返回的*struct_time* 或相应的表示时间的元组为以下形式的字符串: `'Sun Jun 20 23:21:05 1993'`。日期字段的长度为两个字符，如果日期只有一个数字则会以空格填充，例如: `'Wed Jun 9 04:26:40 1993'`。

如果未提供 *t*，则会使用*localtime()* 所返回的当前时间。*asctime()* 不会使用区域设置信息。

> **备注**
>
> 与同名的 C 函数不同，*asctime()* 不添加尾随换行符。

time.**pthread_getcpuclockid**(*thread_id*)

返回指定的 *thread_id* 的特定于线程的 CPU 时间时钟的 *clk_id* 。

使用*threading.Thread* 对象的*threading.get_ident()* 或*ident* 属性为 *thread_id* 获取合适的值。

> **警告**
>
> 传递无效的或过期的 *thread_id* 可能会导致未定义的行为，例如段错误。

Availability: Unix

请参阅 *pthread_getcpuclockid(3)* 的手册页面了解更多信息。

Added in version 3.7.

time.**clock_getres**(*clk_id*)

返回指定时钟 *clk_id* 的分辨率（精度）。有关 *clk_id* 的可接受值列表，请参阅*Clock ID 常量* 。

Availability: Unix.

Added in version 3.3.

time.**clock_gettime**(*clk_id*) → *float*

返回指定 *clk_id* 时钟的时间。有关 *clk_id* 的可接受值列表，请参阅*Clock ID 常量* 。

使用*clock_gettime_ns()* 以避免*float* 类型导致的精度损失。

Availability: Unix.

Added in version 3.3.

time.**clock_gettime_ns**(*clk_id*) → *int*

与*clock_gettime()* 相似，但返回时间为纳秒。

Availability: Unix.

Added in version 3.7.

time.**clock_settime**(*clk_id*, *time:* float)

设置指定 *clk_id* 时钟的时间。目前，*CLOCK_REALTIME* 是 *clk_id* 唯一可接受的值。

使用*clock_settime_ns()* 以避免*float* 类型导致的精度损失。

Availability: Unix, not Android, not iOS.

Added in version 3.3.

time.**clock_settime_ns**(*clk_id*, *time:* int)

与*clock_settime()* 相似，但设置时间为纳秒。

Availability: Unix, not Android, not iOS.

Added in version 3.7.

time.**ctime**([*secs*])

将以距离*epoch* 的秒数表示的时间转换为以下形式的字符串: `'Sun Jun 20 23:21:05 1993'` 代表本地时间。日期字段的长度为两个字符且如果日期只有一位数字则会以空格填充，例如: `'Wed Jun 9 04:26:40 1993'`。

如果 *secs* 未提供或为*None*，则使用*time()* 所返回的当前时间。`ctime(secs)` 等价于 `asctime(localtime(secs))`。*ctime()* 不会使用区域设置信息。

time.**get_clock_info**(*name*)

获取有关指定时钟的信息作为命名空间对象。支持的时钟名称和读取其值的相应函数是:

- `'monotonic'`: *time.monotonic()*
- `'perf_counter'`: *time.perf_counter()*
- `'process_time'`: *time.process_time()*
- `'thread_time'`: *time.thread_time()*
- `'time'`: *time.time()*

结果具有以下属性:

- *adjustable*： 如果时钟可以自动更改（例如通过 NTP 守护程序）或由系统管理员手动更改，则为 `True`，否则为 `False`。
- *implementation*： 用于获取时钟值的基础 C 函数的名称。有关可能的值，请参阅*Clock ID 常量*。
- *monotonic*： 如果时钟不能倒退，则为 `True`，否则为 `False`。
- *resolution*： 以秒为单位的时钟分辨率（*float*）

Added in version 3.3.

time.**gmtime**([*secs*])

将以自*epoch* 开始的秒数表示的时间转换为 UTC 的*struct_time*，其中 dst 旗标始终为零。如果未提供 *secs* 或为*None*，则使用*time()* 所返回的当前时间。一秒以内的小数将被忽略。有关*struct_time* 对象的说明请参见上文。有关此函数的逆操作请参阅*calendar.timegm()*。

time.**localtime**([*secs*])

与*gmtime()* 相似但转换为当地时间。如果未提供 *secs* 或为*None*，则使用由*time()* 返回的当前时间。当 DST 适用于给定时间时，dst 标志设置为 `1`。

localtime() 可能会引发*OverflowError*，如果时间戳超出平台 C `localtime()` 或 `gmtime()` 函数支持的范围，并会在 `localtime()` 或 `gmtime()` 失败时引发*OSError*。这通常被限制在 1970 至 2038 年之间。

time.**mktime**(*t*)

这是*localtime()* 的反函数。它的参数是*struct_time* 或者完整的 9 元组（因为需要 dst 标志；如果它是未知的则使用 `-1` 作为 dst 标志），它表示 *local* 的时间，而不是 UTC。它返回一个浮点数，以便与*time()* 兼容。如果输入值不能表示为有效时间，则*OverflowError* 或*ValueError* 将被引发（这取决于 Python 或底层 C 库是否捕获到无效值）。它可以生成时间的最早日期取决于平台。

time.**monotonic**() → *float*

（以小数表示的秒为单位）返回一个单调时钟的值，即不能倒退的时钟。该时钟不受系统时钟更新的影响。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

时钟：

- 在 Windows 上，调用 `QueryPerformanceCounter()` 和 `QueryPerformanceFrequency()`。
- 在 macOS 上，调用 `mach_absolute_time()` 和 `mach_timebase_info()`。
- 在 HP-UX 上，调用 `gethrtime()`。
- 如果可能则调用 `clock_gettime(CLOCK_HIGHRES)`。
- 在其他情况下，调用 `clock_gettime(CLOCK_MONOTONIC)`。

使用*monotonic_ns()* 以避免*float* 类型导致的精度损失。

Added in version 3.3.

在 3.5 版本发生变更: 该功能现在始终可用且始终在系统范围内。

在 3.10 版本发生变更: 在 macOS 上，现在这个函数作用于全系统。

time.**monotonic_ns**() → *int*

与*monotonic()* 相似，但是返回时间为纳秒数。

Added in version 3.7.

time.**perf_counter**() → *float*

（以小数表示的秒为单位）返回一个性能计数器的值，即用于测量较短持续时间的具有最高有效精度的时钟。它会包括睡眠状态所消耗的时间并且作用于全系统范围。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

在 CPython 中，使用与*time.monotonic()* 相同的单调时钟，即无法回退的时钟。

使用*perf_counter_ns()* 以避免*float* 类型导致的精度损失。

Added in version 3.3.

在 3.10 版本发生变更: 在 Windows 上，现在这个函数作用于全系统。

在 3.13 版本发生变更: 使用与*time.monotonic()* 相同的时钟。

time.**perf_counter_ns**() → *int*

与*perf_counter()* 相似，但是返回时间为纳秒。

Added in version 3.7.

time.**process_time**() → *float*

（以小数表示的秒为单位）返回当前进程的系统和用户 CPU 时间的总计值。它不包括睡眠状态所消耗的时间。根据定义它只作用于进程范围。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

使用*process_time_ns()* 以避免*float* 类型导致的精度损失。

Added in version 3.3.

time.**process_time_ns**() → *int*

与*process_time()* 相似，但是返回时间为纳秒。

Added in version 3.7.

time.**sleep**(*secs*)

调用方线程暂停执行给定的秒数。该参数可以为浮点数以指定一个更精确的休眠时间。

如果休眠被信号打断并且信号处理器未引发异常，休眠将基于重新计算的时延重新开始。

暂停时间有可能比请求的要长出一段不确定的时间，因为会受系统中的其他活动排期影响。

Windows 实现

在 Windows 上，如果 *secs* 为零，线程会将其时间片的剩余部分让渡给任何其他准备要运行的线程。如果没有其他准备要运行的线程，该函数将立即返回，而线程将继续执行。在 Windows 8.1 及更新版本中的实现使用了提供精度为 100 纳秒的 高分辨率定时器。如果 *secs* 为零，则会使用 `Sleep(0)`。

Unix 实现

- 如果可能则使用 `clock_nanosleep()` (精度: 1 纳秒);
- 或者如果可能则使用 `nanosleep()` (精度: 1 纳秒);
- 或者使用 `select()` (精度: 1 微秒).

备注

要模拟“无操作”，请使用 `pass` 而非 `time.sleep(0)`。

要主动让出 CPU，请指定一个实时计划任务策略 并改用 *os.sched_yield()*。

引发一个审计事件 `time.sleep` 并附带参数 `secs`。

在 3.5 版本发生变更: 现在，即使该睡眠过程被信号中断，该函数也会保证调用它的线程至少会睡眠 *secs* 秒。信号处理例程抛出异常的情况除外。（欲了解我们做出这次改变的原因，请参见 **PEP 475** ）

在 3.11 版本发生变更: 在 Unix 上，现在将在可能的情况下使用 `clock_nanosleep()` 和 `nanosleep()` 函数。在 Windows 上，现在将使用可等待的计时器。

在 3.13 版本发生变更: 引发一个审计事件。

`time.`**`strftime`**(*format*[, *t*])

转换一个元组或 *struct_time* 表示的由 *gmtime()* 或 *localtime()* 返回的时间到由 *format* 参数指定的字符串。如果未提供 *t* ，则使用由 *localtime()* 返回的当前时间。*format* 必须是一个字符串。如果 *t* 中的任何字段超出允许范围，则引发 *ValueError* 。

0 是时间元组中任何位置的合法参数；如果它通常是非法的，则该值被强制改为正确的值。

以下指令可以嵌入 *format* 字符串中。它们显示时没有可选的字段宽度和精度规范，并被 *strftime()* 结果中的指示字符替换：

指令	含意	备注
%a	本地化的缩写星期中每日的名称。	
%A	本地化的星期中每日的完整名称。	
%b	本地化的月缩写名称。	
%B	本地化的月完整名称。	
%c	本地化的适当日期和时间表示。	
%d	十进制数 [01,31] 表示的月中日。	
%f	**十进制表示的的微秒数** [000000,999999].	(1)
%H	十进制数 [00,23] 表示的小时（24 小时制）。	
%I	十进制数 [01,12] 表示的小时（12 小时制）。	
%j	十进制数 [001,366] 表示的年中日。	
%m	十进制数 [01,12] 表示的月。	
%M	十进制数 [00,59] 表示的分钟。	
%p	本地化的 AM 或 PM 。	(2)
%S	十进制数 [00,61] 表示的秒。	(3)
%U	十进制数 [00,53] 表示的一年中的周数（星期日作为一周的第一天）。在第一个星期日之前的新年中的所有日子都被认为是在第 0 周。	(4)
%u	以十进制数 [1, 7] 表示的日期值（星期一为 1；星期日为 7）。	
%w	十进制数 [0(星期日),6] 表示的周中日。	
%W	十进制数 [00,53] 表示的一年中的周数（星期一作为一周的第一天）。在第一个星期一之前的新年中的所有日子被认为是在第 0 周。	(4)
%x	本地化的适当日期表示。	
%X	本地化的适当时间表示。	
%y	十进制数 [00,99] 表示的没有世纪的年份。	
%Y	十进制数表示的带世纪的年份。	
%z	时区偏移以格式 +HHMM 或 -HHMM 形式的 UTC/GMT 的正或负时差指示，其中 H 表示十进制小时数字，M 表示小数分钟数字 [-23:59, +23:59] 。[1]	
%Z	时区名称（如果不存在时区，则不包含字符）。已弃用。Page 696, 1	
%G	ISO 8601 年份（类似于 %Y 但遵循针对 ISO 8601 日历年份的规则）。此年份从包含日历年份的第一个星期六的星期开始。	
%V	ISO 8601 星期序号（以十进制数 [01,53] 表示）。每年的第一个星期是包含该年的第一个星期六的星期。每星期的第一天为星期一。	
%%	字面的 '%' 字符。	

注释：

(1) `%f` 格式指示符只应用于*strptime()*，而不应用于*strftime()*。不过，请参看*datetime.datetime.strptime()* 和*datetime.datetime.strftime()*，在这里 `%f` 格式指示符应用于微秒数。

(2) 当与*strptime()* 函数一起使用时，如果使用 `%I` 指令来解析小时，`%p` 指令只影响输出小时字段。

(3) 范围真的是 `0` 到 `61`；值 `60` 在表示 leap seconds 的时间戳中有效，并且由于历史原因支持值 `61`。

(4) 当与*strptime()* 函数一起使用时，`%U` 和 `%W` 仅用于指定星期几和年份的计算。

下面是一个示例，一个与 **RFC 2822** Internet 电子邮件标准以兼容的日期格式。[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

某些平台可能支持其他指令，但只有此处列出的指令具有 ANSI C 标准化的含义。要查看平台支持的完整格式代码集，请参阅 *strftime(3)* 文档。

在某些平台上，可选的字段宽度和精度规范可以按照以下顺序紧跟在指令的初始 `'%'` 之后；这也不可移植。字段宽度通常为 2，除了 `%j`，它是 3。

`time.`**`strptime`**(*string*[, *format*])

根据格式解析表示时间的字符串。返回值为一个被*gmtime()* 或*localtime()* 返回的*struct_time*。

format 参数使用与*strftime()* 相同的指令。它默认为匹配*ctime()* 所返回的格式 `"%a %b %d %H:%M:%S %Y"` 。如果 *string* 不能根据 *format* 来解析，或者解析后它有多余的数据，则会引发*ValueError*。当无法推断出更准确的值时，用于填充任何缺失数据的默认值是 `(1900, 1, 1, 0, 0, 0, 0, 1, -1)`。*string* 和 *format* 都必须为字符串。

例如：

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

支持 `%Z` 指令是基于 `tzname` 中包含的值以及 `daylight` 是否为真。因此，它是特定于平台的，除了识别始终已知的 UTC 和 GMT （并且被认为是非夏令时时区）。

仅支持文档中指定的指令。因为每个平台都实现了 `strftime()`，它有时会提供比列出的指令更多的指令。但是 `strptime()` 独立于任何平台，因此不一定支持所有未记录为支持的可用指令。

`class` `time.`**`struct_time`**

返回的时间值序列的类型为*gmtime()*、*localtime()* 和*strptime()*。它是一个带有*named tuple*接口的对象：可以通过索引和属性名访问值。存在以下值：

[1] 现在不推荐使用 `%Z`，但是所有 ANSI C 库都不支持扩展为首选小时/分钟偏移量的 `%z` 转义符。此外，严格的 1982 年原始 **RFC 822** 标准要求两位数的年份（`%y` 而不是 `%Y`），但是实际在 2000 年之前很久就转移到了 4 位数年。之后，**RFC 822** 已经废弃了，4 位数的年份首先被推荐 **RFC 1123**，然后被 **RFC 2822** 强制执行。

索引	属性	值
0	**tm_year**	（例如，1993）
1	**tm_mon**	range [1, 12]
2	**tm_mday**	range [1, 31]
3	**tm_hour**	range [0, 23]
4	**tm_min**	range [0, 59]
5	**tm_sec**	range [0, 61]；参见*strftime()* 中的*注释 (2)*
6	**tm_wday**	取值范围 [0, 6]；周一为 0
7	**tm_yday**	range [1, 366]
8	**tm_isdst**	0, 1 或 -1；如下所示
N/A	**tm_zone**	时区名称的缩写
N/A	**tm_gmtoff**	以秒为单位的 UTC 以东偏离

请注意，与 C 结构不同，月份值是 [1,12] 的范围，而不是 [0,11] 。

在调用*mktime()* 时，*tm_isdst* 可以在夏令时生效时设置为 1，而在夏令时不生效时设置为 0。值-1 表示这是未知的，并且通常会导致填写正确的状态。

当一个长度不正确的元组被传递给期望*struct_time* 的函数，或者具有错误类型的元素时，会引发*TypeError* 。

time.**time**() → *float*

返回以浮点数表示的从*epoch* 开始的秒数形式的时间。对 leap seconds 的处理取决于具体平台。在 Windows 和大多数 Unix 系统中，闰秒不会被计入从*epoch* 开始的秒数形式的时间中。这通常被称为 Unix 时间。

请注意，即使时间总是作为浮点数返回，但并非所有系统都提供高于 1 秒的精度。虽然此函数通常返回非递减值，但如果在两次调用之间设置了系统时钟，则它可以返回比先前调用更低的值。

返回的数字*time()* 可以通过将其传递给*gmtime()* 函数或转换为 UTC 中更常见的时间格式（即年、月、日、小时等）或通过将它传递给*localtime()* 函数获得本地时间。在这两种情况下都返回一个*struct_time* 对象，日历日期组件可以从中作为属性访问。

时钟：

- 在 Windows 上，调用 `GetSystemTimeAsFileTime()`。
- 如果可能则调用 `clock_gettime(CLOCK_REALTIME)`。
- 在其他情况下，调用 `gettimeofday()`。

使用*time_ns()* 以避免*float* 类型导致的精度损失。

time.**time_ns**() → *int*

与*time()* 相似，但返回时间为用整数表示的自*epoch* 以来所经过的纳秒数。

Added in version 3.7.

time.**thread_time**() → *float*

（以小数表示的秒为单位）返回当前线程的系统和用户 CPU 时间的总计值。它不包括睡眠状态所消耗的时间。根据定义它只作用于线程范围。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

使用*thread_time_ns()* 以避免*float* 类型导致的精度损失。

Availability: Linux, Unix, Windows.

支持 CLOCK_THREAD_CPUTIME_ID 的 Unix 系统。

Added in version 3.7.

time.**thread_time_ns**() → *int*

与*thread_time()* 相似，但返回纳秒时间。

Added in version 3.7.

time.**tzset**()

重置库例程使用的时间转换规则。环境变量 TZ 指定如何完成。它还将设置变量 tzname （来自 TZ 环境变量），timezone （UTC 的西部非 DST 秒），altzone （UTC 以西的 DST 秒）和 daylight （如果此时区没有任何夏令时规则则为 0，如果有夏令时适用的时间，无论过去、现在或未来，则为非零）。

Availability: Unix.

备注

虽然在很多情况下，更改 TZ 环境变量而不调用*tzset()* 可能会影响函数的输出，例如*localtime()* ，不应该依赖此行为。

TZ 不应该包含空格。

TZ 环境变量的标准格式是（为了清晰起见，添加了空格）：

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

组件的位置是：

std 和 dst
三个或更多字母数字，给出时区缩写。这些将传到 time.tzname

offset
偏移量的形式为：± hh[:mm[:ss]] 。这表示添加到达 UTC 的本地时间的值。如果前面有'-'，则时区位于本初子午线的东边；否则，在它是西边。如果 dst 之后没有偏移，则假设夏令时比标准时间提前一小时。

start[/time], end[/time]
指示何时更改为 DST 和从 DST 返回。开始日期和结束日期的格式为以下之一：

Jn
Julian 日 *n* （1 <= *n* <= 365）。闰日不计算在内，因此在所有年份中，2 月 28 日是第 59 天，3 月 1 日是第 60 天。

n
从零开始的 Julian 日（0 <= *n* <= 365）。闰日计入，可以引用 2 月 29 日。

Mm.n.d
一年中 *m* 月的第 *n* 周（1 <= *n* <= 5，1 <= *m* <= 12，第 5 周表示“可能在 *m* 月第 4 周或第 5 周出现的最后第 *d* 日”）的第 *d* 天（0 <= *d* <= 6）。第 1 周是第 *d* 天发生的第一周。第 0 天是星期天。

`time` 的格式与 `offset` 的格式相同，但不允许使用前导符号（'-' 或'+'）。如果没有给出时间，则默认值为 02:00:00。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

在许多 Unix 系统（包括 *BSD，Linux，Solaris 和 Darwin 上），使用系统的区域信息（*tzfile(5)*）数据库来指定时区规则会更方便。为此，将 `TZ` 环境变量设置为所需时区数据文件的路径，相对于系统'zoneinfo' 时区数据库的根目录，通常位于 `/usr/share/zoneinfo`。例如，`'US/Eastern'`、`'Australia/Melbourne'`、`'Egypt'` 或 `'Europe/Amsterdam'`。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Clock ID 常量

这些常量用作*clock_getres()* 和*clock_gettime()* 的参数。

`time.`**CLOCK_BOOTTIME**
与*CLOCK_MONOTONIC* 相同，除了它还包括系统暂停的任何时间。

这允许应用程序获得一个暂停感知的单调时钟，而不必处理*CLOCK_REALTIME* 的复杂性，如果使用 `settimeofday()` 或类似的时间更改时间可能会有不连续性。

Availability: Linux >= 2.6.39.

Added in version 3.7.

`time.`**CLOCK_HIGHRES**
Solaris OS 有一个 `CLOCK_HIGHRES` 计时器，试图使用最佳硬件源，并可能提供接近纳秒的分辨率。`CLOCK_HIGHRES` 是不可调节的高分辨率时钟。

Availability: Solaris.

Added in version 3.3.

`time.`**CLOCK_MONOTONIC**
无法设置的时钟，表示自某些未指定的起点以来的单调时间。

Availability: Unix.

Added in version 3.3.

`time.`**CLOCK_MONOTONIC_RAW**
类似于*CLOCK_MONOTONIC*，但可以访问不受 NTP 调整影响的原始硬件时间。

Availability: Linux >= 2.6.28, macOS >= 10.12.

Added in version 3.3.

time.**CLOCK_MONOTONIC_RAW_APPROX**

类似于*CLOCK_MONOTONIC_RAW*，但在上下文切换时将读取由系统缓存的值因此会不够精确。

Availability: macOS >= 10.12.

Added in version 3.13.

time.**CLOCK_PROCESS_CPUTIME_ID**

来自 CPU 的高分辨率每进程计时器。

Availability: Unix.

Added in version 3.3.

time.**CLOCK_PROF**

来自 CPU 的高分辨率每进程计时器。

Availability: FreeBSD, NetBSD >= 7, OpenBSD.

Added in version 3.7.

time.**CLOCK_TAI**

国际原子时间

该系统必须有一个当前闰秒表以便能给出正确的回答。PTP 或 NTP 软件可以用来维护闰秒表。

Availability: Linux.

Added in version 3.9.

time.**CLOCK_THREAD_CPUTIME_ID**

特定于线程的 CPU 时钟。

Availability: Unix.

Added in version 3.3.

time.**CLOCK_UPTIME**

该时间的绝对值是系统运行且未暂停的时间，提供准确的正常运行时间测量，包括绝对值和间隔值。

Availability: FreeBSD, OpenBSD >= 5.5.

Added in version 3.7.

time.**CLOCK_UPTIME_RAW**

单调递增的时钟，记录从一个任意起点开始的时间，不受频率或时间调整的影响，并且当系统休眠时将不会递增。

Availability: macOS >= 10.12.

Added in version 3.8.

time.**CLOCK_UPTIME_RAW_APPROX**

类似于*CLOCK_UPTIME_RAW*，但该值在上下文切换时将由系统缓存因此会不够精确。

Availability: macOS >= 10.12.

Added in version 3.13.

以下常量是唯一可以发送到*clock_settime()*的参数。

time.**CLOCK_REALTIME**

系统范围的实时时钟。设置此时钟需要适当的权限。

Availability: Unix.

Added in version 3.3.

16.3.3 时区常量

time.**altzone**

本地 DST 时区的偏移量，以 UTC 为单位的秒数，如果已定义。如果当地 DST 时区在 UTC 以东（如在西欧，包括英国），则是负数。只有当 daylight 非零时才使用它。见下面的注释。

time.**daylight**

如果定义了 DST 时区，则为非零。见下面的注释。

time.**timezone**

本地（非 DST）时区的偏移量，UTC 以西的秒数（西欧大部分地区为负，美国为正，英国为零）。见下面的注释。

time.**tzname**

两个字符串的元组：第一个是本地非 DST 时区的名称，第二个是本地 DST 时区的名称。如果未定义 DST 时区，则不应使用第二个字符串。见下面的注释。

备注

对于上述时区常量 (*altzone*, *daylight*, *timezone* 和 *tzname*)，该值由当模块加载或 *tzset()* 最后一次被调用时生效的时区规则确定并且对于已过去的时间可能不正确。建议使用来自 *localtime()* 结果的 *tm_gmtoff* 和 *tm_zone* 来获取时区信息。

参见

模块 *datetime*
更多面向对象的日期和时间接口。

模块 *locale*
国际化服务。区域设置会影响 *strftime()* 和 *strptime()* 中许多格式说明符的解析。

模块 *calendar*
一般日历相关功能。这个模块的 timegm() 是函数 *gmtime()* 的反函数。

备注

16.4 logging --- Python 的日志记录工具

源代码： Lib/logging/__init__.py

Important

此页面仅包含 API 参考信息。教程信息和更多高级用法的讨论，请参阅

- 基础教程
- 进阶教程
- 日志记录操作手册

这个模块为应用与库实现了灵活的事件日志系统的函数与类。

使用标准库提供的 logging API 最主要的好处是，所有的 Python 模块都可能参与日志输出，包括你自己的日志消息和第三方模块的日志消息。

这是一个惯例用法的简单示例:

```
# myapp.py
import logging
import mylib
logger = logging.getLogger(__name__)

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logger.info('Started')
    mylib.do_something()
    logger.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging
logger = logging.getLogger(__name__)

def do_something():
    logger.info('Doing something')
```

如果你运行 *myapp.py*，你应该在 *myapp.log* 中看到：

```
INFO:__main__:Started
INFO:mylib:Doing something
INFO:__main__:Finished
```

这种惯常用法的一个关键特性在于大部分代码都是简单地通过 `getLogger(__name__)` 创建一个模块级别的日志记录器，并使用该日志记录器来完成任何需要的日志记录。这样既简洁明了，又能根据需要对下游代码进行细粒度的控制。记录到模块级日志记录器的消息会被转发给更高级别模块的日志记录器的处理器，一直到最高层级的日志记录器既根日志记录器；这种方式被称为分级日志记录。

要使日志记录有用，就需要对其进行配置：为每个日志记录器设置级别和目标，还可能改变特定模块的日志记录方式，通常是基于命令行参数或应用配置来实现。在大多数情况下，如上文所述，只有根日志记录器需要如此配置，因为所有在模块层级上的低级别日志记录器最终都会将消息转发给它的处理器。*basicConfig()* 提供了一种配置根日志记录器的快捷方式，它可以处理多种应用场景。

这个模块提供许多强大而灵活的功能。如果对 logging 不太熟悉，掌握它最好的方式就是查看它对应的教程（**详见右侧的链接**）。

该模块定义的基础类，以及它们的属性和方法都在下面的小节中列出。

- 记录器暴露了应用程序代码直接使用的接口。
- 处理器将日志记录（由记录器创建）发送到适当的目标。
- 过滤器提供了更细粒度的功能，用于确定要输出的日志记录。
- 格式器指定最终输出中日志记录的样式。

16.4.1 记录器对象

记录器有以下的属性和方法。注意 永远不要直接实例化记录器，应当通过模块级别的函数 `logging.getLogger(name)`。多次使用相同的名字调用*getLogger()* 会一直返回相同的 Logger 对象的引用。

`name` 一般是以句点分割的层级值，如 `foo.bar.baz` (尽管也可以是简单形式，例如 `foo`)。层级结构列表中位于下方的日志记录器是列表中较高位置的日志记录器的子级。例如，假定有一个名叫 `foo` 的日志记录器，则名字为 `foo.bar`, `foo.bar.baz` 和 `foo.bam` 的日志记录器都是 `foo` 的子级。而且，所有日志记录器都是根日志记录器的子级。日志记录器名称的层级结构类似于 Python 包的层级结构，如果你使用建议的构造 `logging.getLogger(__name__)` 以每个模块为基础来组织你的日志记录器则将与后者完全一致。这是因为在一个模块中，`__name__` 是该模块在 Python 包命名空间中的名字。

class logging.**Logger**

name

这是日志记录器的名称，也是传给*getLogger()* 用以获取日志记录器的值。

> **备注**
>
> 该属性应当被视为是只读的。

level

该日志记录器的阈值，由*setLevel()* 方法设置。

> **备注**
>
> 请不要直接设置该值——应当始终使用*setLevel()*，它会检查传入的级别。

parent

此日志记录器的父日志记录器。它可能会根据命名空间层次结构中更高日志记录器的实例化而发生变化。

> **备注**
>
> 该值应被视为只读。

propagate

如果这个属性为真，记录到这个记录器的事件除了会发送到此记录器的所有处理程序外，还会传递给更高级别（祖先）记录器的处理器，此外任何关联到这个记录器的处理器。消息会直接传递给祖先记录器的处理器——不考虑祖先记录器的级别和过滤器。

如果为假，记录消息将不会传递给当前记录器的祖先记录器的处理器。

举例说明：如果名为 `A.B.C` 的记录器的传播属性求值为真，则任何通过调用诸如 `logging.getLogger('A.B.C').error(...)` 之类的方法记录到 `A.B.C` 的事件，在第一次被传递到 `A.B.C` 上附加的处理器后，将 [取决于传递该记录器的级别和过滤器设置] 依次传递给附加到名为 `A.B`，`A` 的记录器和根记录器的所有处理器。如果 `A.B.C`、`A.B`、`A` 组成的链中，任一记录器的 `propagate` 属性设置为假，那么这将是最后一个其处理器会收到事件的记录器，此后传播在该点停止。

构造器将这个属性初始化为 `True`。

> **备注**
>
> 如果你将一个处理器附加到一个记录器 和其一个或多个祖先记录器，它可能发出多次相同的记录。通常，您不需要将一个处理器附加到一个以上的记录器上——如果您将它附加到记录器层次结构中最高的适当记录器上，则它将看到所有后代记录器记录的所有事件，前提是它们的传播设置保留为 `True`。一种常见的方案是仅将处理器附加到根记录器，通过传播来处理其余部分。

handlers

直接连接到此记录器的处理程序列表实例。

> **备注**
>
> 该属性应被视为只读；通常通过*addHandler()* 和*removeHandler()* 方法进行更改，它们使用锁来确保线程安全的操作。

disabled

该属性禁用对任何事件的处理。初始化程序将其设置为 `False`，只有日志配置代码才能更改。

> **备注**
>
> 该属性应当被视为是只读的。

setLevel(*level*)

给记录器设置阈值为 *level*。日志等级小于 *level* 会被忽略。严重性为 *level* 或更高的日志消息将由该记录器的任何一个或多个处理器发出，除非将处理器的级别设置为比 *level* 更高的级别。

创建记录器时，级别默认设置为*NOTSET*（当记录器是根记录器时，将处理所有消息；如果记录器不是根记录器，则将委托给父级）。请注意，根记录器的默认级别为*WARNING*。

委派给父级的意思是如果一个记录器的级别设置为 NOTSET，将遍历其祖先记录器，直到找到级别不是 NOTSET 的记录器，或者到根记录器为止。

如果发现某个父级的级别不是 NOTSET，那么该父级的级别将被视为发起搜索的记录器的有效级别，并用于确定如何处理日志事件。

如果搜索到达根记录器，并且其级别为 NOTSET，则将处理所有消息。否则，将使用根记录器的级别作为有效级别。

参见日志级别 级别列表。

在 3.2 版本发生变更: 现在 *level* 参数可以接受形如'INFO' 的级别字符串表示形式，以代替形如*INFO* 的整数常量。但是请注意，级别在内部存储为整数，并且*getEffectiveLevel()* 和*isEnabledFor()* 等方法的传入/返回值也为整数。

isEnabledFor(*level*)

指示此记录器是否将处理级别为 *level* 的消息。此方法首先检查由 `logging.disable(level)` 设置的模块级的级别，然后检查由*getEffectiveLevel()* 确定的记录器的有效级别。

getEffectiveLevel()

指示此记录器的有效级别。如果通过*setLevel()* 设置了除*NOTSET* 以外的值，则返回该值。否则，将层次结构遍历到根，直到找到除*NOTSET* 以外的其他值，然后返回该值。返回的值是一个整数，通常为*logging.DEBUG*、*logging.INFO* 等等。

getChild(*suffix*)

返回由后缀确定的该记录器的后代记录器。因此，`logging.getLogger('abc').getChild('def.ghi')` 与 `logging.getLogger('abc.def.ghi')` 将返回相同的记录器。这是一个便捷方法，当使用如 `__name__` 而不是字符串字面值命名父记录器时很有用。

Added in version 3.2.

getChildren()

返回由该日志记录器的直接下级日志记录器组成的集合。举例来说 `logging.getLogger().getChildren()` 将返回包含名为 `foo` 和 `bar` 的日志记录器的集合，但名为 `foo.bar` 的日志记录器则不会包括在集合中。类似地，`logging.getLogger('foo').getChildren()` 将返回包括名为 `foo.bar` 的日志记录器的集合，但不会包括名为 `foo.bar.baz` 的日志记录器。

Added in version 3.12.

debug(*msg*, **args*, ***kwargs*)

在此记录器上记录*DEBUG* 级别的消息。*msg* 是消息格式字符串，而 *args* 是用于字符串格式化操作合并到 *msg* 的参数。（请注意，这意味着您可以在格式字符串中使用关键字以及单个字典参数。）当未提供 *args* 时，不会对 *msg* 执行%格式化操作。

在 *kwargs* 中会检查四个关键字参数：*exc_info*，*stack_info*，*stacklevel* 和 *extra*。

如果 *exc_info* 的求值结果不为 false，则它将异常信息添加到日志消息中。如果提供了一个异常元组（按照*sys.exc_info()* 返回的格式）或一个异常实例，则它将被使用；否则，调用*sys.exc_info()* 以获取异常信息。

第二个可选关键字参数是 *stack_info*，默认为 `False`。如果为 True，则将堆栈信息添加到日志消息中，包括实际的日志调用。请注意，这与通过指定 *exc_info* 显示的堆栈信息不同：前者是从堆栈底部到当前线程中的日志记录调用的堆栈帧，而后者是在搜索异常处理程序时，跟踪异常而打开的堆栈帧的信息。

您可以独立于 *exc_info* 来指定 *stack_info*，例如，即使在未引发任何异常的情况下，也可以显示如何到达代码中的特定点。堆栈帧在标题行之后打印：

```
Stack (most recent call last):
```

这模仿了显示异常帧时所使用的 `Traceback (most recent call last):` 。

第三个可选关键字参数是 *stacklevel* ，默认为 `1` 。如果大于 1 ，则在为日志记录事件创建的*LogRecord* 中计算行号和函数名时，将跳过相应数量的堆栈帧。可以在记录帮助器时使用它，以便记录的函数名称，文件名和行号不是帮助器的函数/方法的信息，而是其调用方的信息。此参数是*warnings* 模块中的同名等效参数。

第四个关键字参数 *extra* 被用于传递一个字典，该字典将被用来将为日志记录事件创建的*LogRecord* 中的 `__dict__` 填充为用户自定义的属性。然后你将可以随意使用这些自定义的属性。举例来说，它们可以被加入到已记录的日志消息中。例如:

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

输出类似于

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

在 The keys in the dictionary passed in *extra* 传入的字典的键不应与日志系统所使用的键相冲突。(请参阅*LogRecord 属性* 一节了解有关日志系统所使用的键的更多信息。)

如果在已记录的消息中使用这些属性，则需要格外小心。例如，在上面的示例中，*Formatter* 已设置了格式字符串，其在*LogRecord* 的属性字典中键值为 “clientip” 和 “user”。如果缺少这些内容，则将不会记录该消息，因为会引发字符串格式化异常。因此，在这种情况下，您始终需要使用 *extra* 字典传递这些键。

尽管这可能很烦人，但此功能旨在用于特殊情况，例如在多个上下文中执行相同代码的多线程服务器，并且出现的有趣条件取决于此上下文（例如在上面的示例中就是远程客户端 IP 地址和已验证用户名)。在这种情况下，很可能将专门的*Formatter* 与特定的*Handler* 一起使用。

如果没有处理器附加到这个记录器（或它的任何上级，会将相关的*Logger.propagate* 属性也纳入考虑)，消息将被发送到在*lastResort* 上设置的处理器。

在 3.2 版本发生变更: 增加了 *stack_info* 参数。

在 3.5 版本发生变更: *exc_info* 参数现在可以接受异常实例。

在 3.8 版本发生变更: 增加了 *stacklevel* 参数。

info(*msg*, **args*, ***kwargs*)

在此记录器上记录*INFO* 级别的消息。参数解释同*debug()*。

warning(*msg*, **args*, ***kwargs*)

在此记录器上记录*WARNING* 级别的消息。参数解释同*debug()*。

> **备注**
>
> 有一个功能上与 `warning` 一致的方法 `warn`。由于 `warn` 已被弃用，请不要使用它——改为使用 `warning`。

error(*msg*, **args*, ***kwargs*)

在此记录器上记录*ERROR*级别的消息。参数解释同*debug()*。

critical(*msg*, **args*, ***kwargs*)

在此记录器上记录*CRITICAL*级别的消息。参数解释同*debug()*。

log(*level*, *msg*, **args*, ***kwargs*)

在此记录器上记录 *level* 整数代表的级别的消息。参数解释同*debug()*。

exception(*msg*, **args*, ***kwargs*)

在此记录器上记录*ERROR*级别的消息。参数解释同*debug()*。异常信息将添加到日志消息中。仅应从异常处理程序中调用此方法。

addFilter(*filter*)

将指定的过滤器 *filter* 添加到此记录器。

removeFilter(*filter*)

从此记录器中删除指定的过滤器 *filter*。

filter(*record*)

将此记录器的过滤器应用于记录，如果记录能被处理则返回 `True`。过滤器会被依次使用，直到其中一个返回假值为止。如果它们都不返回假值，则记录将被处理（传递给处理器）。如果返回任一为假值，则不会对该记录做进一步处理。

addHandler(*hdlr*)

将指定的处理器 *hdlr* 添加到此记录器。

removeHandler(*hdlr*)

从此记录器中删除指定的处理器 *hdlr*。

findCaller(*stack_info=False*, *stacklevel=1*)

查找调用源的文件名和行号，以文件名，行号，函数名称和堆栈信息 4 元素元组的形式返回。堆栈信息将返回 `None`，除非 *stack_info* 为 `True`。

stacklevel 参数用于调用*debug()* 和其他 API。如果大于 1，则多余部分将用于跳过堆栈帧，然后再确定要返回的值。当从帮助器/包装器代码调用日志记录 API 时，这通常很有用，以便事件日志中的信息不是来自帮助器/包装器代码，而是来自调用它的代码。

handle(*record*)

通过将记录传递给与此记录器及其祖先关联的所有处理器来处理（直到某个 *propagate* 值为 false）。此方法用于从套接字接收的未序列化的以及在本地创建的记录。使用*filter()* 进行记录器级别过滤。

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)

这是一种工厂方法，可以在子类中对其进行重写以创建专门的*LogRecord*实例。

hasHandlers()

检查此记录器是否配置了任何处理器。通过在此记录器及其记录器层次结构中的父级中查找处理器完成此操作。如果找到处理器则返回 `True`，否则返回 `False`。只要找到 “propagate” 属性设置为假值的记录器，该方法就会停止搜索层次结构——其将是最后一个检查处理器是否存在的记录器。

Added in version 3.2.

在 3.7 版本发生变更: 现在可以对处理器进行序列化和反序列化。

16.4.2 日志级别

日志记录级别的数值在下表中给出。如果你想要定义自己的级别，并且需要它们具有相对于预定义级别的特定值，那么这你可能对以下内容感兴趣。如果你定义具有相同数值的级别，它将覆盖预定义的值；预定义的名称将失效。

级别	数值	何种含义 / 何时使用
logging.**NOTSET**	0	当在日志记录器上设置时，表示将查询上级日志记录器以确定生效的级别。如果仍被解析为 NOTSET，则会记录所有事件。在处理器上设置时，所有事件都将被处理。
logging.**DEBUG**	10	详细的信息，通常只有试图诊断问题的开发人员才会感兴趣。
logging.**INFO**	20	确认程序按预期运行。
logging.**WARNING**	30	表明发生了意外情况，或近期有可能发生问题（例如‘磁盘空间不足’）。软件仍会按预期工作。
logging.**ERROR**	40	由于严重的问题，程序的某些功能已经不能正常执行
logging.**CRITICAL**	50	严重的错误，表明程序已不能继续执行

16.4.3 处理器对象

处理器具有以下属性和方法。请注意*Handler* 不可直接实例化；该类是被作为更有用的子类的基类。不过，子类中的 __init__() 方法需要调用*Handler.__init__()*。

class logging.**Handler**

__init__(*level=NOTSET*)

初始化*Handler* 实例时，需要设置它的级别，将过滤列表置为空，并且创建锁（通过*createLock()*）来序列化对 I/O 的访问。

createLock()

初始化一个线程锁，用来序列化对底层的 I/O 功能的访问，底层的 I/O 功能可能不是线程安全的。

acquire()

获取由*createLock()* 创建的线程锁。

release()

释放由*acquire()* 获取的线程锁。

setLevel(*level*)

给处理器设置阈值为 *level* 。日志级别小于 *level* 将被忽略。创建处理器时，日志级别被设置为*NOTSET* （所有的消息都会被处理）。

参见日志级别 级别列表。

在 3.2 版本发生变更: *level* 形参现在接受像'INFO' 这样的字符串形式的级别表达方式，也可以使用像*INFO* 这样的整数常量。

setFormatter(*fmt*)

将处理器的格式设为 *fmt*。*fmt* 参数必须为*Formatter* 实例或 None。

addFilter(*filter*)

将指定的过滤器 *filter* 添加到此处理器。

removeFilter(*filter*)

从此处理器中删除指定的过滤器 *filter* 。

filter(*record*)

将此处理器的过滤器应用于记录，在要处理记录时返回 `True` 。依次查询过滤器，直到其中一个返回假值为止。如果它们都不返回假值，则将发出记录。如果返回一个假值，则处理器将不会发出记录。

flush()

确保所有日志记录从缓存输出。此版本不执行任何操作，并且应由子类实现。

close()

清理处理器使用的所有资源。此版本没有输出但会从内部处理器映射中移除处理器，该映射被用来按名称查找处理器。

子类应当通过重写*close()* 方法确保它会被调用。

handle(*record*)

经已添加到处理器的过滤器过滤后，有条件地发出指定的日志记录。用获取/释放 I/O 线程锁包装了记录的实际发出行为。

handleError(*record*)

此方法应当在*emit()* 调用期间遇到异常时从处理器中调用。如果模块级属性*raiseExceptions* 为 `False`，则异常将被静默地忽略。这是大多数情况下日志系统所需要的——大多数用户不会关心日志系统中的错误，他们对应用程序错误更感兴趣。但是，你可以根据需要将其替换为自定义处理器。指定的记录是发生异常时正在处理的记录。(*raiseExceptions* 的默认值是 `True`，因为这在开发过程中更有用处)。

format(*record*)

如果设置了格式器则用其对记录进行格式化。否则，使用模块的默认格式器。

emit(*record*)

执行实际记录给定日志记录所需的操作。这个版本应由子类实现，因此这里直接引发*NotImplementedError* 异常。

> **警告**
>
> 此方法会在获得一个处理器层级的锁之后被调用，在此方法返回之后锁将被释放。当你重写此方法时，请注意在调用任何可能执行锁定操作的日志记录 API 的其他部分的方法时务必小心谨慎，因为这可能会导致死锁。具体来说:
>
> - 日志记录配置 API 会获取模块层级锁，然后还会在配置处理器时获取处理器层级锁。
> - 许多日志记录 API 都会锁定模块级锁。如果这样的 API 在此方法中被调用，则它可能会在另一个线程执行配置调用时导致死锁，因为那个线程将试图在处理器级锁 *之前* 获取模块级锁，而这个线程将试图在处理器级锁 *之后* 获取模块级锁（因为在此方法中，处理器级锁已经被获取了）。

有关作为标准随附的处理器列表，请参见*logging.handlers*。

16.4.4 格式器对象

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, ***, *defaults=None*)

负责将一个*LogRecord* 转换为可供人类或外部系统解读的输出字符串。

参数

- **fmt** (*str*) -- 用于日志记录整体输出的给定 *style* 形式的格式字符串。可用的映射键将从*LogRecord* 对象的*LogRecord 属性* 中提取。如果未指定，则将使用 `'%(message)s'`，即已记录的日志消息。

- **datefmt** (str) -- 用于日志记录输出的日期/时间部分的给定 *style* 形式的格式字符串。如果未指定，则将使用 *formatTime()* 中描述的默认值。
- **style** (str) -- 可以是 '%', '{' 或 '$' 之一并决定格式字符串将如何与数据合并: 使用 *printf 风格的字符串格式化* (%), *str.format()* ({) 或 *string.Template* ($) 之一。这将只应用于 *fmt* 和 *datefmt* (例如 '%(message)s' 或 '{message}')，而不会应用于传给日志记录方法的实际日志消息。但是，也存在 其他方式可以为日志消息使用 { 和 $ 格式化。
- **validate** (bool) -- 如果为 True (默认值)，则不正确或不匹配的 *fmt* 和 *style* 将引发 *ValueError*；例如 logging.Formatter('%(asctime)s - %(message)s', style='{')。
- **defaults** (dict[str, Any]) -- 一个由在自定义字段中使用的默认值组成的字典。例如 logging.Formatter('%(ip)s %(message)s', defaults={"ip": None})

在 3.2 版本发生变更: 增加了 *style* 形参。

在 3.8 版本发生变更: 增加了 *validate* 形参。

在 3.10 版本发生变更: 增加了 *defaults* 形参。

format(*record*)

记录的属性字典被用作字符串格式化操作的操作数。返回结果字符串。在格式化该字典之前，会执行几个预备步骤。记录的 *message* 属性是用 *msg % args* 来计算的。如果格式化字符串包含 '(asctime)'，则会调用 *formatTime()* 来格式化事件时间。如果有异常信息，则使用 *formatException()* 将其格式化并添加到消息中。请注意已格式化的异常信息会缓存在 *exc_text* 属性中。这很有用因为异常信息可以被 pickle 并通过网络发送，但是如果你有不止一个对异常信息进行定制的 *Formatter* 子类则应当小心。在这种情况下，你必须在一个格式化器完成格式化后清空缓存的值 (通过将 *exc_text* 属性设为 None)，以便下一个处理事件的格式化器不会使用缓存的值，而是重新计算它。

如果栈信息可用，它将被添加在异常信息之后，如有必要请使用 *formatStack()* 来转换它。

formatTime(*record*, *datefmt=None*)

此方法应由想要使用格式化时间的格式器中的 *format()* 调用。可以在格式器中重写此方法以提供任何特定要求，但是基本行为如下：如果指定了 *datefmt* （字符串），则将其用于 *time.strftime()* 来格式化记录的创建时间。否则，使用格式'%Y-%m-%d %H:%M:%S,uuu'，其中 uuu 部分是毫秒值，其他字母根据 *time.strftime()* 文档。这种时间格式的示例为 2003-01-23 00:29:50,411。返回结果字符串。

此函数使用一个用户可配置函数将创建时间转换为元组。默认情况下，使用 *time.localtime()*；要为特定格式化程序实例更改此项，请将实例的 converter 属性设为具有与 *time.localtime()* 或 *time.gmtime()* 相同签名的函数。要为所有格式化程序更改此项，例如当你希望所有日志时间都显示为 GMT，请在 Formatter 类中设置 converter 属性。

在 3.3 版本发生变更: 在之前版本中，默认格式是被硬编码的，例如这个例子: 2010-09-06 22:38:15,292 其中逗号之前的部分由 strptime 格式字符串 ('%Y-%m-%d %H:%M:%S') 处理，而逗号之后的部分为毫秒值。因为 strptime 没有表示毫秒的占位符，毫秒值使用了另外的格式字符串来添加 '%s,%03d' --- 这两个格式字符串代码都是硬编码在该方法中的。经过修改，这些字符串被定义为类层级的属性，当需要时可以在实例层级上被重写。属性的名称为 default_time_format (用于 strptime 格式字符串) 和 default_msec_format (用于添加毫秒值)。

在 3.9 版本发生变更: default_msec_format 可以为 None。

formatException(*exc_info*)

将指定的异常信息（由 *sys.exc_info()* 返回的标准异常元组）格式化为字符串。默认实现只是使用了 *traceback.print_exception()*。结果字符串将被返回。

formatStack(*stack_info*)

将指定的堆栈信息（由 *traceback.print_stack()* 返回的字符串，但移除末尾的换行符）格式化为字符串。默认实现只是返回输入值。

class logging.**BufferingFormatter**(*linefmt=None*)

适合用来在你想要格式化多条记录时进行子类化的格式化器。你可以传入一个*Formatter*实例用来格式化每一行（每一行对应一条记录）。如果未被指定，则会使用默认的格式化器（仅输出事件消息）作为行格式化器。

formatHeader(*records*)

为 *records* 列表返回一个标头。基本实现只是返回空字符串。如果你想要指明特定行为则需要重写此方法，例如显示记录条数、标题或分隔行等。

formatFooter(*records*)

为 *records* 列表返回一个结束标记。基本实现只是返回空字符串。如果你想要指明特定行为则需要重写此方法，例如显示记录条数或分隔行等。

format(*records*)

为 *records* 列表返回已格式化文本。基本实现在没有记录时只是返回空字符串；在其他情况下，它将返回标头、使用行格式化器执行格式化的每行记录以及结束标记。

16.4.5 过滤器对象

Filters 可被 Handlers 和 Loggers 用来实现比按层级提供更复杂的过滤操作。基本过滤器类只允许低于日志记录器层级结构中低于特定层级的事件。例如，一个用'A.B' 初始化的过滤器将允许'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' 等日志记录器所记录的事件。但'A.BB', 'B.A.B' 等则不允许。如果用空字符串初始化，则所有事件都会通过。

class logging.**Filter**(*name=''*)

返回一个*Filter*类的实例。如果指定了 *name*，则它将被用来为日志记录器命名，该类及其子类将通过该过滤器允许指定事件通过。如果 *name* 为空字符串，则允许所有事件通过。

filter(*record*)

指定的记录是否会被写入日志？否则返回假值，是则返回真值。过滤器可以原地修改日志记录或者返回完全不同的记录实例并在该事件未来的任何处理过程中用它来替代原始日志记录。

请注意关联到处理器的过滤器会在事件由处理器发出之前被查询，而关联到日志记录器的过滤器则会在有事件被记录的的任何时候（使用*debug()*, *info()* 等等）在将事件发送给处理器之前被查询。这意味着由后代日志记录器生成的事件将不会被父代日志记录器的过滤器设置所过滤，除非该过滤器也已被应用于后代日志记录器。

你实际上不需要子类化 Filter：你可以传入任何一个包含有相同语义的 filter 方法的实例。

在 3.2 版本发生变更: 你不需要创建专门的 Filter 类，或使用具有 filter 方法的其他类：你可以使用一个函数（或其他可调用对象）作为过滤器。过滤逻辑将检查过滤器对象是否具有 filter 属性：如果有，就会将它当作是 Filter 并调用它的*filter()*方法。在其他情况下，则会将它当作是可调用对象并将记录作为唯一的形参进行调用。返回值应当与*filter()*的返回值相一致。

在 3.12 版本发生变更: 现在你可以从过滤器返回一个*LogRecord*实例来替代日志记录而不是原地修改它。这允许附加到特定*Handler*的过滤器在日志记录发出之前修改它，而不会对其他处理器产生附带影响。

尽管过滤器主要被用来构造比层级更复杂的规则以过滤记录，但它们可以查看由它们关联的处理器或记录器所处理的每条记录：当你想要执行统计特定记录器或处理器共处理了多少条记录，或是在所处理的*LogRecord*中添加、修改或移除属性这样的任务时该特性将很有用处。显然改变 LogRecord 时需要相当小心，但将上下文信息注入日志确实是被允许的 (参见 filters-contextual)。

16.4.6 LogRecord 属性

*LogRecord*实例是每当有日志被记录时由*Logger*自动创建的，并且可通过*makeLogRecord()*手动创建（例如根据从网络接收的已封存事件创建）。

class logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

包含与被记录的事件相关的所有信息。

主要信息是在 *msg* 中 *args* 传递的，它们使用 msg % args 组合到一起以创建记录的 message 属性。

参数

- **name** (str) -- 用于记录此 LogRecord 所表示事件的记录器名称。请注意 LogRecord 中的记录器名称将始终为该值，即使它可能是由附加到不同（上级）日志记录器的处理器所发出的。
- **level** (int) -- 日志记录事件的数字层级 (如 10 表示 DEBUG, 20 表示 INFO 等等)。请注意这会转换为 LogRecord 的 两个属性: levelno 表示数字值而 levelname 表示对应的层级名。
- **pathname** (str) -- 日志记录调用所在源文件的完整路径字符串。
- **lineno** (int) -- 记录调用所在源文件中的行号。
- **msg** (Any) -- 事件描述消息，这可以是一个带有% 形式可变数据占位符的格式字符串，或是任意对象 (参见 arbitrary-object-messages)。
- **args** (tuple | dict[str, Any]) -- 要合并到 *msg* 参数以获得事件描述的可变数据。
- **exc_info** (tuple[type[BaseException], BaseException, types.TracebackType] | None) -- 包含当前异常信息的异常元组，就如sys.exc_info() 所返回的，或者如果没有可用异常信息则为 None。
- **func** (str | None) -- 唤起日志记录调用的函数或方法名称。
- **sinfo** (str | None) -- 一个文本字符串，表示当前线程中从堆栈底部直到日志记录调用的堆栈信息。

getMessage()

在将LogRecord 实例与任何用户提供的参数合并之后，返回此实例的消息。如果用户提供给日志记录调用的消息参数不是字符串，则会在其上调用str() 以将它转换为字符串。此方法允许将用户定义的类用作消息，类的 __str__ 方法可以返回要使用的实际格式字符串。

在 3.2 版本发生变更: 通过提供用于创建记录的工厂方法已使得LogRecord 的创建更易于配置。该工厂方法可使用getLogRecordFactory() 和setLogRecordFactory() （在此可查看工厂方法的签名）来设置。

在创建时可使用此功能将你自己的值注入LogRecord。你可以使用以下模式:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

通过此模式，多个工厂方法可以被链接起来，并且只要它们不重写彼此的属性或是在无意中覆盖了上面列出的标准属性，就不会发生意外。

16.4.7 LogRecord 属性

LogRecord 具有许多属性，它们大多数来自于传递给构造器的形参。（请注意 LogRecord 构造器形参与 LogRecord 属性的名称并不总是完全彼此对应的。）这些属性可被用于将来自记录的数据合并到格式字符串中。下面的表格（按字母顺序）列出了属性名称、它们的含义以及相应的%-style 格式字符串内占位符。

如果是使用 {}-格式化（str.format()），你可以将 {attrname} 用作格式字符串内的占位符。如果是使用 $-格式化（string.Template），则会使用 ${attrname} 的形式。当然在这两种情况下，都应当将 attrname 替换为你想要使用的实际属性名称。

在 {}-格式化的情况下，你可以在属性名称之后放置指定的格式化旗标，并用冒号来分隔。例如：占位符 {msecs:03.0f} 会将毫秒值 4 格式化为 004。有参看str.format() 文档了解你可以使用的选项的详情。

属性名称	格式	描述
args	此属性不需要用户进行格式化。	合并到 msg 以产生 message 的包含参数的元组，或是其中的值将被用于合并的字典（当只有一个参数且其类型为字典时）。
asctime	%(asctime)s	表示人类易读的*LogRecord* 生成时间。默认形式为'2003-07-08 16:49:45,896'（逗号之后的数字为时间的毫秒部分）。
created	%(created)f	当*LogRecord* 被创建的时间（即*time.time_ns()* / 1e9 所返回的值）。
exc_info	此属性不需要用户进行格式化。	异常元组（例如 sys.exc_info）或者如未发生异常则为 None。
filename	%(filename)s	pathname 的文件名部分。
funcName	%(funcName)s	函数名包括调用日志记录.
levelname	%(levelname)s	消息文本记录级别（'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'）。
levelno	%(levelno)s	消息数字的记录级别 (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*).
lineno	%(lineno)d	发出日志记录调用所在的源行号（如果可用）。
message	%(message)s	记入日志的消息，即 msg % args 的结果。这是在唤起*Formatter.format()* 时设置的。
module	%(module)s	模块 (filename 的名称部分)。
msecs	%(msecs)d	*LogRecord* 被创建的时间的毫秒部分。
msg	此属性不需要用户进行格式化。	在原始日志记录调用中传入的格式字符串。与 args 合并以产生 message，或是一个任意对象 (参见 arbitrary-object-messages)。
name	%(name)s	用于记录调用的日志记录器名称。
pathname	%(pathname)s	发出日志记录调用的源文件的完整路径名（如果可用）。
process	%(process)d	进程 ID（如果可用）
processName	%(processName)s	进程名（如果可用）
relativeCreated	%(relativeCreat	以毫秒数表示的 LogRecord 被创建的时间，即相对于 logging 模块被加载时间的差值。
stack_info	此属性不需要用户进行格式化。	当前线程中从堆栈底部起向上直到包括日志记录调用并引发创建当前记录堆栈帧创建的堆栈帧信息（如果可用）。
thread	%(thread)d	线程 ID（如果可用）
threadName	%(threadName)s	线程名（如果可用）
taskName	%(taskName)s	*asyncio.Task* 名称（如果可用）。

在 3.1 版本发生变更: 添加了 *processName*

在 3.12 版本发生变更: 添加了 *taskName*。

16.4.8 LoggerAdapter 对象

LoggerAdapter 实例会被用来方便地将上下文信息传入日志记录调用。要获取用法示例，请参阅 添加上下文信息到你的日志记录输出部分。

class logging.**LoggerAdapter**(*logger*, *extra*, *merge_extra=False*)

返回使用一个下层*Logger* 实例、一个字典型对象 (*extra*) 和一个指明单独日志调用的 *extra* 参数是否要与*LoggerAdapter* 的 extra 值合并的布尔值 (*merge_extra*) 进行初始化的*LoggerAdapter* 的实例。其默认行为将忽略单独日志调用的 *extra* 参数并只使用*LoggerAdapter* 实例中的相应参数值。

process(*msg*, *kwargs*)

修改传递给日志记录调用的消息和/或关键字参数以便插入上下文信息。此实现接受以 *extra* 形式传给构造器的对象并使用'extra' 键名将其加入 *kwargs*。返回值为一个 (*msg*, *kwargs*) 元组，其包含（可能经过修改的）传入参数。

manager

在 *logger* 中委托给下层的 manager。

_log

在 *logger* 中委托给下层的 `_log()` 方法。

在上述方法之外，*LoggerAdapter* 还支持*Logger* 的下列方法: *debug()*，*info()*，*warning()*，*error()*，*exception()*，*critical()*，*log()*，*isEnabledFor()*，*getEffectiveLevel()*，*setLevel()* 以及*hasHandlers()*。这些方法具有与它们在*Logger* 中的对应方法相同的签名，因此你可以互换使用这两种类型的实例。

在 3.2 版本发生变更: *isEnabledFor()*，*getEffectiveLevel()*，*setLevel()* 和*hasHandlers()* 方法已被添加到*LoggerAdapter*。这些方法会委托给下层的日志记录器。

在 3.6 版本发生变更: 增加了 `manager` 属性和 `_log()` 方法，它们会委托给下层的日志记录器并允许适配器嵌套。

在 3.13 版本发生变更: 增加了 *merge_extra* 参数。

16.4.9 线程安全

logging 模块的目标是使客户端不必执行任何特殊操作即可确保线程安全。它通过使用线程锁来达成这个目标；用一个锁来序列化对模块共享数据的访问，并且每个处理程序也会创建一个锁来序列化对其下层 I/O 的访问。

如果你要使用*signal* 模块来实现异步信号处理程序，则可能无法在这些处理程序中使用 logging。这是因为*threading* 模块中的锁实现并非总是可重入的，所以无法从此类信号处理程序唤起。

16.4.10 模块级函数

在上述的类之外，还有一些模块级的函数。

`logging.`**`getLogger`**(*name=None*)

返回一个由 name 指定名称的日志记录器，或者如果 name 为 `None` 则返回层级结构中的根日志记录器。如果指定了 name，它通常是以点号分隔的带层级结构的名称如 *'a'*, *'a.b'* 或 *'a.b.c.d'*。这些名称的选择完全取决于使用 logging 的开发者，不过就如在记录器对象 中提到的那样建议使用 `__name__`，除非你有不这样做的特别理由。

所有用给定的 name 对该函数的调用都将返回相同的日志记录器实例。这意味着日志记录器实例不需要在应用的各部分间传递。

`logging.`**`getLoggerClass`**()

返回标准的*Logger* 类，或是最近传给*setLoggerClass()* 的类。此函数可以从一个新的类定义中调用，以确保安装自定义的*Logger* 类不会撤销其他代码已经应用的自定义操作。例如:

```
class MyLogger(logging.getLoggerClass()):
    # ... 在这里重写行为
```

`logging.`**`getLogRecordFactory`**()

返回一个被用来创建*LogRecord* 的可调用对象。

Added in version 3.2: 此函数与*setLogRecordFactory()* 一起提供，以允许开发者对表示日志记录事件的*LogRecord* 的构造有更好的控制。

请参阅*setLogRecordFactory()* 了解有关如何调用该工厂方法的更多信息。

`logging.`**`debug`**(*msg*, **args*, ***kwargs*)

这是在根日志记录器上调用*Logger.debug()* 的便捷函数。其参数的处理方式与该方法中的描述完全一致。

唯一的区别在于如果根日志记录器没有处理器，则在根日志记录器上调用 `debug` 之前会先调用*basicConfig()*。

对于非常简短的脚本或 `logging` 功能的快速演示，`debug` 和其他模块级函数可能会很方便。不过，大多数程序都会想要仔细和显式地控制日志记录配置，所以应当更倾向于创建一个模块级的日志记录器并在其上调用*Logger.debug()* (或其他特定级别的方法)，如本文档的开头所描述的那样。

logging.**info**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条*INFO*级别的消息。其他参数与行为均与*debug()*的相同。

logging.**warning**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条*WARNING*级别的消息。其他参数与行为均与*debug()*的相同。

> **备注**
>
> 有一个已过时方法 warn 其功能与 warning 一致。由于 warn 已被弃用，请不要使用它——而是改用 warning。

logging.**error**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条*ERROR*级别的消息。其他参数与行为均与*debug()*的相同。

logging.**critical**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条*CRITICAL*级别的消息。其他参数与行为均与*debug()*的相同。

logging.**exception**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条*ERROR*级别的消息。其他参数与行为均与*debug()*的相同。异常信息会被添加到日志记录消息中。此函数应当仅从异常处理器中调用。

logging.**log**(*level*, *msg*, **args*, ***kwargs*)

在根日志记录器上记录一条 *level* 级别的消息。其他参数与行为均与*debug()*相同。

logging.**disable**(*level=CRITICAL*)

为所有日志记录器提供重写的级别 *level*，其优先级高于日志记录器自己的级别。当需要临时限制整个应用程序中的日志记录输出时，此功能会很有用。它的效果是禁用所有重要程度为 *level* 及以下的日志记录调用，因此如果你附带 INFO 值调用它，则所有 INFO 和 DEBUG 事件就会被丢弃，而重要程度为 WARNING 以及上的事件将根据日志记录器的当前有效级别来处理。如果 `logging.disable(logging.NOTSET)` 被调用，它将移除这个重写的级别，因此日志记录输出会再次取决于单个日志记录器的有效级别。

请注意如果你定义了任何高于 CRITICAL 的自定义日志级别（并不建议这样做），你就将无法沿用 *level* 形参的默认值，而必须显式地提供适当的值。

在 3.7 版本发生变更: *level* 形参默认级别为 CRITICAL。请参阅 bpo-28524 了解此项改变的更多细节。

logging.**addLevelName**(*level*, *levelName*)

在一个内部字典中关联级别 *level* 与文本 *levelName*，该字典会被用来将数字级别映射为文本表示形式，例如在*Formatter*格式化消息的时候。此函数也可被用来定义你自己的级别。唯一的限制是自定义的所有级别必须使用此函数来注册，级别值必须为正整数并且其应随严重程度而递增。

> **备注**
>
> 如果你考虑要定义你自己的级别，请参阅 custom-levels 部分。

logging.**getLevelNamesMapping**()

返回一个级别名到其对应日志记录级别的映射。例如，字符串"CRITICAL" 将映射到*CRITICAL*。所返回的映射是从每个对此函数的调用的内部映射拷贝的。

Added in version 3.11.

logging.**getLevelName**(*level*)

返回日志记录级别 *level* 的字符串表示。

如果 *level* 为预定义的级别*CRITICAL*, *ERROR*, *WARNING*, *INFO* 或*DEBUG* 之一则你会得到相应的字符串。如果你使用*addLevelName()*将级别关联到名称则返回你为 *level* 所关联的名称。如果传入了与已定义级别相对应的数字值，则返回对应的字符串表示。

level 形参也接受级别的字符串表示例如'INFO'。在这种情况下，此函数将返回级别所对应的数字值。

如果未传入可匹配的数字或字符串值，则返回字符串'Level %s' % level。

> **备注**
>
> 级别在内部以整数表示（因为它们在日志记录逻辑中需要进行比较）。此函数被用于在整数级别与通过 `%(levelname)s` 格式描述符方式在格式化日志输出中显示的级别名称之间进行相互的转换 (参见*LogRecord 属性*)。

在 3.4 版本发生变更: 在早于 3.4 的 Python 版本中，此函数也可传入一个字符串形式的级别名称，并将返回对应的级别数字值。此未记入文档的行为被视为是一个错误，并在 Python 3.4 中被移除，但又在 3.4.2 中被恢复以保持向下兼容性。

`logging.`**`getHandlerByName`**`(`*name*`)`

返回具有指定 *name* 的处理器，或者如果指定名称的处理器不存在则返回 `None`。

Added in version 3.12.

`logging.`**`getHandlerNames`**`()`

返回一个由所有已知处理器名称组成的不可变集合。

Added in version 3.12.

`logging.`**`makeLogRecord`**`(`*attrdict*`)`

创建并返回一个新的*LogRecord* 实例，实例属性由 *attrdict* 定义。此函数适用于接受一个通过套接字传输的封存好的*LogRecord* 属性字典，并在接收端将其重建为一个*LogRecord* 实例。

`logging.`**`basicConfig`**`(`***kwargs*`)`

通过使用默认的*Formatter* 创建一个*StreamHandler* 并将其加入根日志记录器来为日志记录系统执行基本配置。如果没有为根日志记录器定义处理器则*debug()*，*info()*，*warning()*，*error()* 和*critical()* 等函数将自动调用*basicConfig()*。

如果根日志记录器已配置了处理器则此函数将不执行任何操作，除非关键字参数 *force* 被设为 `True`。

> **备注**
>
> 此函数应当在其他线程启动之前从主线程被调用。在 2.7.1 和 3.2 之前的 Python 版本中，如果此函数从多个线程被调用，一个处理器（在极少的情况下）有可能被多次加入根日志记录器，导致非预期的结果例如日志中的消息出现重复。

支持以下关键字参数。

格式	描述
filename	使用指定的文件名创建一个*FileHandler*，而不是*StreamHandler*。
filemode	如果指定了 *filename*，则用此模式 打开该文件。默认模式为 `'a'`。
format	使用指定的格式字符串作为处理器。默认为属性以冒号分隔的 `levelname`, `name` 和 `message`。
datefmt	使用指定的日期/时间格式，与*time.strftime()* 所接受的格式相同。
style	如果指定了 *format*，将为格式字符串使用此风格。`'%'`, `'{'` 或 `'$'` 分别对应于*printf 风格*, *str.format()* 或*string.Template*。默认为 `'%'`。
level	设置根记录器级别为指定的*level*.
stream	使用指定的流初始化*StreamHandler*。请注意此参数与 *filename* 不兼容——如果两者同时存在，则会引发 `ValueError`。
handlers	如果指定，这应为一个包含要加入根日志记录器的已创建处理器的可迭代对象。任何尚未设置格式描述符的处理器将被设置为在此函数中创建的默认格式描述符。请注意此参数与 *filename* 或 *stream* 不兼容——如果两者同时存在，则会引发 `ValueError`。
force	如果将此关键字参数指定为 true，则在执行其他参数指定的配置之前，将移除并关闭附加到根记录器的所有现有处理器。
encoding	如果此关键字参数与 *filename* 一同被指定，则其值会在创建*FileHandler* 时被使用，因而也会在打开输出文件时被使用。
errors	如果此关键字参数与 *filename* 一同被指定，则其值会在创建*FileHandler* 时被使用，因而也会在打开输出文件时被使用。如果未指定，则会使用值'backslashreplace'。请注意如果指定为 `None`，它将被原样传给*open()*，这意味着它将会当作传入'errors' 一样处理。

在 3.2 版本发生变更: 增加了 *style* 参数。

在 3.3 版本发生变更: 增加了 *handlers* 参数。增加了额外的检查来捕获指定不兼容参数的情况 (例如同时指定 *handlers* 与 *stream* 或 *filename*，或者同时指定 *stream* 与 *filename*)。

在 3.8 版本发生变更: 增加了 *force* 参数。

在 3.9 版本发生变更: 增加了 *encoding* 和 *errors* 参数。

`logging.`**`shutdown`**`()`

通过刷新和关闭所有处理程序来通知日志记录系统执行有序停止。此函数应当在应用退出时被调用并且在此调用之后不应再使用日志记录系统。

当 logging 模块被导入时，它会将此函数注册为退出处理程序 (参见*atexit*)，因此通常不需要手动执行该操作。

`logging.`**`setLoggerClass`**`(`*klass*`)`

通知日志记录系统在实例化日志记录器时使用 *klass* 类。该类应当定义 `__init__()` 使其只需要一个 name 参数，并且 `__init__()` 应当调用 `Logger.__init__()`。此函数通常会在需要使用自定义日志记录器行为的应用程序实例化任何日志记录器之前被调用。在此调用之后，在其他任何时候都不要直接使用该子类来实例化日志记录器：请继续使用*logging.getLogger()* API 来获取你的日志记录器。

`logging.`**`setLogRecordFactory`**`(`*factory*`)`

设置一个用来创建*LogRecord* 的可调用对象。

参数
`factory` -- 用来实例化日志记录的工厂可调用对象。

Added in version 3.2: 此函数与*getLogRecordFactory()* 一起提供，以便允许开发者对如何构造表示日志记录事件的*LogRecord* 有更好的控制。

可调用对象 factory 具有如下签名:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

name
日志记录器名称

level
日志记录级别（数字）。

fn
进行日志记录调用的文件的完整路径名。

lno
记录调用所在文件中的行号。

msg
日志消息。

args
日志记录消息的参数。

exc_info
异常元组，或 None 。

func
调用日志记录调用的函数或方法的名称。

sinfo
与*traceback.print_stack()* 所提供的类似的栈回溯信息，显示调用的层级结构。

kwargs
其他关键字参数。

16.4.11 模块级属性

logging.**lastResort**

通过此属性提供的“最后处理者”。这是一个以 WARNING 级别写入到 sys.stderr 的*StreamHandler*，用于在没有任何日志记录配置的情况下处理日志记录事件。最终结果就是将消息打印到 sys.stderr，这会替代先前形式为”no handlers could be found for logger XYZ” 的错误消息。如果出于某种原因你需要先前的行为，可将 lastResort 设为 None。

Added in version 3.2.

logging.**raiseExceptions**

用于查看在处理过程中异常是否应当被传播。

默认值: True。

如果*raiseExceptions* 为 False，则异常会被静默地忽略。这大多数情况下是日志系统所需要的——大多数用户不会关心日志系统中的错误，他们对应用程序错误更感兴趣。

16.4.12 与警告模块集成

captureWarnings() 函数可用来将*logging* 和*warnings* 模块集成。

logging.**captureWarnings**(*capture*)

此函数用于打开和关闭日志系统对警告的捕获。

如果 *capture* 是 True，则*warnings* 模块发出的警告将重定向到日志记录系统。具体来说，将使用*warnings.formatwarning()* 格式化警告信息，并将结果字符串使用*WARNING* 等级记录到名为 'py.warnings' 的记录器中。

如果 *capture* 是 False，则将停止将警告重定向到日志记录系统，并且将警告重定向到其原始目标（即在 captureWarnings(True) 调用之前的有效目标）。

参见

logging.config 模块
日志记录模块的配置 API 。

logging.handlers 模块
日志记录模块附带的有用处理器。

PEP 282 - Logging 系统
该提案描述了 Python 标准库中包含的这个特性。

Original Python logging package
这是该*logging* 包的原始来源。该站点提供的软件包版本适用于 Python 1.5.2、2.1.x 和 2.2.x，它们不被*logging* 包含在标准库中。

16.5 logging.config --- 日志记录配置

源代码： Lib/logging/config.py

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程
- 日志记录操作手册

这一节描述了用于配置 logging 模块的 API。

16.5.1 配置函数

下列函数可配置 logging 模块。它们位于*logging.config* 模块中。它们的使用是可选的 --- 要配置 logging 模块你可以使用这些函数，也可以通过调用主 API (在*logging* 本身定义) 并定义在*logging* 或*logging.handlers* 中声明的处理器。

logging.config.**dictConfig**(*config*)

从一个字典获取日志记录配置。字典的内容描述见下文的配置字典架构。

如果在配置期间遇到错误，此函数将引发*ValueError*，*TypeError*，*AttributeError* 或*ImportError* 并附带适当的描述性消息。下面是将会引发错误的（可能不完整的）条件列表:

- `level` 不是字符串或者不是对应于实际日志记录级别的字符串。
- `propagate` 值不是布尔类型。
- id 没有对应的目标。
- 在增量调用期间发现不存在的处理器 id。
- 无效的日志记录器名称。
- 无法解析为内部或外部对象。

解析由 `DictConfigurator` 类执行，该类的构造器可传入用于配置的字典，并且具有 `configure()` 方法。*logging.config* 模块具有可调用属性 `dictConfigClass`，其初始值设为 `DictConfigurator`。你可以使用你自己的适当实现来替换 `dictConfigClass` 的值。

dictConfig() 会调用 `dictConfigClass` 并传入指定的字典，然后在所返回的对象上调用 `configure()` 方法以使配置生效:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

例如，DictConfigurator 的子类可以在它自己的 __init__() 中调用 DictConfigurator.__init__()，然后设置可以在后续 configure() 调用中使用的自定义前缀。dictConfigClass 将被绑定到这个新的子类，然后就能以与在默认的未定制状态下完全相同的方式调用*dictConfig()*。

Added in version 3.2.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*, *encoding=None*)

从一个*configparser* 格式文件中读取日志记录配置。文件格式应当与配置文件格式 中的描述一致。此函数可在应用程序中被多次调用，以允许最终用户在多个预设配置中进行选择（如果开发者提供了展示选项并加载选定配置的机制）。

如果文件不存在将引发*FileNotFoundError* 而如果文件无效或为空则将引发*RuntimeError*。

参数

- **fname** -- 一个文件名，或一个文件型对象，或是一个派生自*RawConfigParser* 的实例。如果传入了一个派生自 RawConfigParser 的实例，它会被原样使用。否则，将会实例化一个*ConfigParser*，并且它会从作为 fname 传入的对象中读取配置。如果存在*readline()* 方法，则它会被当作一个文件型对象并使用*read_file()* 来读取；在其他情况下，它会被当作一个文件名并传递给*read()*。
- **defaults** -- 要传给 ConfigParser 的默认值可在此参数中指定。
- **disable_existing_loggers** -- 如果指定为 False，则当执行此调用时已存在的日志记录器会保持启用。默认值为 True 因为这将以向下兼容方式启用旧行为。此行为是禁用任何现有的非根日志记录器除非它们或它们的上级在日志记录配置中被显式地命名。
- **encoding** -- 当 *fname* 为文件名时被用于打开文件的编码格式。

在 3.4 版本发生变更: 现在接受*RawConfigParser* 子类的实例作为 fname 的值。这有助于:

- 使用一个配置文件，其中日志记录配置只是全部应用程序配置的一部分。
- 使用从一个文件读取的配置，它随后会在被传给 fileConfig 之前由使用配置的应用程序来修改（例如基于命令行参数或运行时环境的其他部分）。

在 3.10 版本发生变更: 增加了 *encoding* 形参。

在 3.12 版本发生变更: 如果所提供的文件不存在或无效或为空则将抛出一个异常。

logging.config.**listen**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

在指定的端口上启动套接字服务器，并监听新的配置。如果未指定端口，则会使用模块默认的 DEFAULT_LOGGING_CONFIG_PORT。日志记录配置将作为适合由*dictConfig()* 或*fileConfig()* 进行处理的文件来发送。返回一个*Thread* 实例，你可以在该实例上调用*start()* 来启动服务器，对该服务器你可以在适当的时候执行*join()*。要停止该服务器，请调用*stopListening()*。

如果指定 verify 参数，则它应当是一个可调用对象，该对象应当验证通过套接字接收的字节数据是否有效且应被处理。这可以通过对通过套接字发送的内容进行加密和/或签名来完成，这样 verify 可调用对象就能执行签名验证和/或解密。verify 可调用对象的调用会附带一个参数——通过套接字接收的字节数据——并应当返回要处理的字节数据，或者返回 None 来指明这些字节数据应当被丢弃。返回的字节数据可以与传入的字节数据相同（例如在只执行验证的时候），或者也可以完全不同（例如在可能执行了解密的时候）。

要将配置发送到套接字，请读取配置文件并将其作为字节序列发送到套接字，字节序列要以使用 struct.pack('>L', n) 打包为二进制格式的四字节长度的字符串打头。

备注

因为配置的各部分是通过*eval()* 传递的，使用此函数可能让用户面临安全风险。虽然此函数仅绑定到 localhost 上的套接字，因此并不接受来自远端机器的连接，但在某些场景中不受

信任的代码可以在调用*listen()* 的进程的账户下运行。具体来说，如果如果调用*listen()* 的进程在用户无法彼此信任的多用户机器上运行，则恶意用户就能简单地通过连接到受害者的*listen()* 套接字并发送运行攻击者想在受害者的进程上执行的任何代码的配置的方式，安排运行几乎任意的代码。如果是使用默认端口这会特别容易做到，即便使用了不同端口也不难做到。要避免发生这种情况的风险，请在*listen()* 中使用 `verify` 参数来防止未经认可的配置被应用。

在 3.4 版本发生变更: 添加了 `verify` 参数。

备注

如果你希望将配置发送给未禁用现有日志记录器的监听器，你将需要使用 JSON 格式的配置，该格式将使用*dictConfig()* 进行配置。此方法允许你在你发送的配置中将 `disable_existing_loggers` 指定为 `False`。

`logging.config.`**`stopListening`**`()`

停止通过对*listen()* 的调用所创建的监听服务器。此函数的调用通常会先于在*listen()* 的返回值上调用 `join()`。

16.5.2 安全考量

日志配置功能试图提供便利，从某种角度来说，这是通过将配置文件中的文本转换为日志配置中使用的 Python 对象来完成的——如用户定义对象 中所述。但是，这些相同的机制（从用户定义的模块中导入可调用对象并使用配置中的参数调用它们）可用于调用您指定的任何代码，因此您应该 * 非常谨慎 * 地处理来自非信任源的配置文件。并且在实际加载前，您应该确信加载不会导致坏事情。

16.5.3 配置字典架构

描述日志记录配置需要列出要创建的不同对象及它们之间的连接；例如，你可以创建一个名为'console' 的处理器，然后名为'startup' 的日志记录器将可以把它的消息发送给'console' 处理器。这些对象并不仅限于*logging* 模块所提供的对象，因为你还可以编写你自己的格式化或处理器类。这些类的形参可能还需要包括 `sys.stderr` 这样的外部对象。描述这些对象和连接的语法会在下面的对象连接 中定义。

字典架构细节

传给*dictConfig()* 的字典必须包含以下的键:

- *version* - 应设为代表架构版本的整数值。目前唯一有效的值是 1，使用此键可允许架构在继续演化的同时保持向下兼容性。

所有其他键都是可选项，但如存在它们将根据下面的描述来解读。在下面提到'configuring dict' 的所有情况下，都将检查它的特殊键 `'()'` 以确定是否需要自定义实例化。如果需要，则会使用下面用户定义对象 所描述的机制来创建一个实例；否则，会使用上下文来确定要实例化的对象。

- *formatters* - 对应的值将是一个字典，其中每个键是一个格式器 ID 而每个值则是一个描述如何配置相应*Formatter* 实例的字典。

  在配置字典中搜索以下可选键，这些键对应于创建*Formatter* 对象时传入的参数。:

  - `format`
  - `datefmt`
  - `style`
  - `validate` (从版本 >=3.8 起)
  - `defaults` (版本 >=3.12)

可选的 `class` 键指定格式化器类的名称（形式为带点号的模块名和类名）。实例化的参数与*Formatter* 的相同，因此这个键对于实例化自定义的*Formatter* 子类最为有用。如果，替代类可能会以扩展和精简格式呈现异常回溯信息。如果你的格式化器需要不同的或额外的配置键，你应当使用用户定义对象。

- *filters* - 对应的值将是一个字典，其中每个键是一个过滤器 ID 而每个值则是一个描述如何配置相应 Filter 实例的字典。

  将在配置字典中搜索键 `name` (默认值为空字符串) 并且该键会被用于构造*logging.Filter* 实例。

- *handlers* - 对应的值将是一个字典，其中每个键是一个处理器 ID 而每个值则是一个描述如何配置相应 Handler 实例的字典。

  将在配置字典中搜索下列键:

  - `class` (强制)。这是处理器类的完整限定名称。
  - `level` (可选)。处理器的级别。
  - `formatter` (可选)。处理器所对应格式化器的 ID。
  - `filters` (可选)。由处理器所对应过滤器的 ID 组成的列表。

    在 3.11 版本发生变更: `filters` 除了 id 以外还能接受 filter 实例。

  所有 *其他键*会被作为关键字参数传递给处理器类的构造器。例如，给定如下配置:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  ID 为 `console` 的处理器会被实例化为*logging.StreamHandler*，并使用 `sys.stdout` 作为下层流。ID 为 `file` 的处理器会被实例化为*logging.handlers.RotatingFileHandler*，并附带关键字参数 `filename='logconfig.log', maxBytes=1024, backupCount=3`。

- *loggers* - 对应的值将是一个字典，其中每个键是一个日志记录器名称而每个值则是一个描述如何配置相应 Logger 实例的字典。

  将在配置字典中搜索下列键:

  - `level` (可选)。日志记录器的级别。
  - `propagate` (可选)。日志记录器的传播设置。
  - `filters` (可选)。由日志记录器对应过滤器的 ID 组成的列表。

    在 3.11 版本发生变更: `filters` 除了 id 以外还能接受 filter 实例。

  - `handlers` (可选)。由日志记录器对应处理器的 ID 组成的列表。

  指定的记录器将根据指定的级别、传播、过滤器和处理器来配置。

- *root* - 这将成为根日志记录器对应的配置。配置的处理方式将与所有日志记录器一致，除了 `propagate` 设置将不可用之外。

- *incremental* - 配置是否要被解读为在现有配置上新增。该值默认为 `False`，这意味着指定的配置将以与当前*fileConfig()* API 所使用的相同语义来替代现有的配置。

  如果指定的值为 `True`，配置会按照增量配置 部分所描述的方式来处理。

- *disable_existing_loggers* - 是否要禁用任何现有的非根日志记录器。该设置对应于*fileConfig()* 中的同名形参。如果省略，则此形参默认为 `True`。如果 *incremental* 为 `True` 则该省会被忽略。

增量配置

为增量配置提供完全的灵活性是很困难的。例如，由于过滤器和格式化器这样的对象是匿名的，一旦完成配置，在增加配置时就不可能引用这些匿名对象。

此外，一旦完成了配置，在运行时任意改变日志记录器、处理器、过滤器、格式化器的对象图就不是很有必要；日志记录器和处理器的详细程度只需通过设置级别即可实现控制（对于日志记录器则可设置传播旗标）。在多线程环境中以安全的方式任意改变对象图也许会导致问题；虽然并非不可能，但这样做的好处不足以抵销其所增加的实现复杂度。

这样，当配置字典的 `incremental` 键存在且为 `True` 时，系统将完全忽略任何 `formatters` 和 `filters` 条目，并仅会处理 `handlers` 条目中的 `level` 设置，以及 `loggers` 和 `root` 条目中的 `level` 和 `propagate` 设置。

使用配置字典中的值可让配置以封存字典对象的形式通过线路传送给套接字监听器。这样，长时间运行的应用程序的日志记录的详细程度可随时间改变而无须停止并重新启动应用程序。

对象连接

该架构描述了一组日志记录对象——日志记录器、处理器、格式化器、过滤器——它们在对象图中彼此连接。因此，该架构需要能表示对象之间的连接。例如，在配置完成后，一个特定的日志记录器关联到了一个特定的处理器。出于讨论的目的，我们可以说该日志记录器代表两者间连接的源头，而处理器则代表对应的目标。当然在已配置对象中这是由包含对处理器的引用的日志记录器来代表的。在配置字典中，这是通过给每个目标对象一个 ID 来无歧义地标识它，然后在源头对象中使用该 ID 来实现的。

因此，举例来说，考虑以下 YAML 代码段:

```
formatters:
  brief:
    # 以下为针对格式化器 id 'brief' 的配置
  precise:
    # 以下为针对格式化器 'precise' 的配置
handlers:
  h1: # 这是一个 id
   # 以下是针对处理器 id 'h1' 的配置
   formatter: brief
  h2: # 这是另一个 id
   # 以下是针对处理器 id 'h2' 的配置
   formatter: precise
loggers:
  foo.bar.baz:
    # 针对日志记录器 'foo.bar.baz' 的其它配置
    handlers: [h1, h2]
```

（注：这里使用 YAML 是因为它的可读性比表示字典的等价 Python 源码形式更好。）

日志记录器 ID 就是日志记录器的名称，它会在程序中被用来获取对日志记录器的引用，例如 `foo.bar.baz`。格式化器和过滤器的 ID 可以是任意字符串值 (例如上面的 `brief`, `precise`) 并且它们是瞬态的，因为它们仅对处理配置字典有意义并会被用来确定对象之间的连接，而当配置调用完成时不会在任何地方保留。

上面的代码片段指明名为 `foo.bar.baz` 的日志记录器应当关联到两个处理器，它们的 ID 是 `h1` 和 `h2`。`h1` 的格式化器的 ID 是 `brief`，而 `h2` 的格式化器的 ID 是 `precise`。

用户定义对象

此架构支持用户定义对象作为处理器、过滤器和格式化器。（日志记录器的不同实例不需要具有不同类型，因此这个配置架构并不支持用户定义日志记录器类。）

要配置的对象是由字典描述的，其中包含它们的配置详情。在某些地方，日志记录系统将能够从上下文中推断出如何实例化一个对象，但是当要实例化一个用户自定义对象时，系统将不知道要如何做。为了

提供用户自定义对象实例化的完全灵活性，用户需要提供一个‘工厂’函数——即在调用时传入配置字典并返回实例化对象的可调用对象。这是用一个通过特殊键 '()' 来访问的工厂函数的绝对导入路径来标示的。下面是一个实际的例子:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

上面的 YAML 代码片段定义了三个格式化器。第一个的 ID 为 brief，是带有特殊格式字符串的标准*logging.Formatter* 实例。第二个的 ID 为 default，具有更长的格式同时还显式地定义了时间格式，并将最终实例化一个带有这两个格式字符串的*logging.Formatter*。以 Python 源代码形式显示的 brief 和 default 格式化器分别具有下列配置子字典:

```
{
  'format' : '%(message)s'
}
```

和:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

并且由于这些字典不包含特殊键 '()'，实例化方式是从上下文中推断出来的：结果会创建标准的*logging.Formatter* 实例。第三个格式器的 ID 为 custom，对应配置子字典为:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

并且它包含特殊键 '()'，这意味着需要用户自定义实例化方式。在此情况下，将使用指定的工厂可调用对象。如果它本身就是一个可调用对象则将被直接使用——否则如果你指定了一个字符串（如这个例子所示）则将使用正常的导入机制来定位实例的可调用对象。调用该可调用对象将传入配置子字典中 **剩余的**条目作为关键字参数。在上面的例子中，调用将预期返回 ID 为 custom 的格式化器:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

⚠ 警告

上面示例中 bar, spam 和 answer 等键的值不应是配置目录或引用如 cfg://foo 或 ext://bar，因为它们将不会被配置机制所处理，而是被原样传给可调用对象。

将 '()' 用作特殊键是因为它不是一个有效的关键字形参名称，这样就不会与调用中使用的关键字参数发生冲突。'()' 还被用作表明对应值为可调用对象的助记符。

在 3.11 版本发生变更: handlers 和 loggers 的 filters 成员除了 id 以外还能接受 filter 实例。

你还可以指定一个特殊键 '.' 其值应为一个将属性名称映射到对应值的字典。如果找到，则指定的属性在被返回之前将在用户定义的对象上设置。因此，使用以下配置:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

被返回的格式化器的 foo 属性将设为 'bar' 而 baz 属性将设为 'bozz'。

> **警告**
>
> 上面示例中 foo 和 baz 等属性的值不应是配置目录或引用如 cfg://foo 或 ext://bar，因为它们将不会被配置机制所处理，而是被原样设置为属性。

处理器配置顺序

处理器按其键的字母顺序进行配置，而已配置的处理器将替换配置方案内部 handlers 字典（的一个工作副本）中的配置字典。如果你使用 cfg://handlers.foo 这样的构造，那么在初始状态下 handlers['foo'] 会指向名为 foo 的处理器的配置字典，随后（一旦配置了该处理器）它将指向已配置的处理器实例。因此，cfg://handlers.foo 可以解析为一个字典或处理器实例。通常来说，对于带依赖的处理器采用在它们所依赖的任何处理器完成配置 _ 之后 _ 再进行配置的方式来命名处理器是一种明智的做法；这将允许使用 cfg://handlers.foo 这样的构造来配置依赖于处理器 foo 的处理器。如果这个带依赖的处理器被命名为 bar，则会导致问题，因为 bar 的配置将在 foo 的配置之前被尝试使用，而 foo 将尚未配置完成。但是，如果带依赖的处理器被命名为 foobar，则它将在 foo 之后被配置，结果就是 cfg://handlers.foo 将被解析为已配置的处理器 foo，而不是其配置字典。

访问外部对象

有时一个配置需要引用配置以外的对象，例如 sys.stderr。如果配置字典是使用 Python 代码构造的，这会很直观，但是当配置是通过文本文件（例如 JSON, YAML）提供的时候就会引发问题。在一个文本文件中，没有将 sys.stderr 与字符串字面值 'sys.stderr' 区分开来的标准方式。为了实现这种区分，配置系统会在字符串值中查找规定的特殊前缀并对其做特殊处理。例如，如果在配置中将字符串字面值 'ext://sys.stderr' 作为一个值来提供，则 ext:// 将被去除而该值的剩余部分将使用正常导入机制来处理。

此类前缀的处理方式类似于协议处理：存在一种通用机制来查找与正则表达式 ^(?P<prefix>[a-z]+)://(?P<suffix>.*)$ 相匹配的前缀，如果识别出了 prefix，则 suffix 会以与前缀相对应的方式来处理并且处理的结果将替代原字符串值。如果未识别出前缀，则原字符串将保持不变。

访问内部对象

除了外部对象，有时还需要引用配置中的对象。这将由配置系统针对它所了解的内容隐式地完成。例如，在日志记录器或处理器中表示 level 的字符串值 'DEBUG' 将被自动转换为值 logging.DEBUG，而 handlers, filters 和 formatter 条目将接受一个对象 ID 并解析为适当的目标对象。

但是，对于 *logging* 模块所不了解的用户自定义对象则需要一种更通用的机制。例如，考虑 *logging.handlers.MemoryHandler*，它接受一个 target 参数即其所委托的另一个处理器。由于系统已经知道存在该类，因而在配置中，给定的 target 只需为相应目标处理器的的对象 ID 即可，而系统将根据该 ID 解析出处理器。但是，如果用户定义了一个具有 alternate 处理器的 my.package.MyHandler，则配置程序将不知道 alternate 指向的是一个处理器。为了应对这种情况，通用解析系统允许用户指定：

```
handlers:
  file:
    # 以下是文件处理器的配置
```

(续下页)

(接上页)

```
  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

字符串字面值 'cfg://handlers.file' 将按照与 ext:// 前缀类似的方式被解析为结果字符串，但查找操作是在配置自身而不是在导入命名空间中进行。该机制允许按点号或按索引来访问，与 str.format 所提供的方式类似。这样，给定以下代码段:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

在该配置中，字符串 'cfg://handlers' 将解析为包含 handlers 键的字典，字符串 'cfg://handlers.email 将解析为 handlers 字典中包含 email 键的字典，依此类推。字符串 'cfg://handlers.email.toaddrs[1] 将解析为 'dev_team@domain.tld' 而字符串 'cfg://handlers.email.toaddrs[0]' 将解析为值 'support_team@domain.tld'。subject 值可以使用 'cfg://handlers.email.subject' 或者等价的 'cfg://handlers.email[subject]' 来访问。后一种形式仅在键包含空格或非字母数字类字符的情况下才需要使用。请注意字符 [和] 不允许在键中使用。如果一个索引仅由十进制数码构成，则将尝试使用相应的整数值来访问，如有必要则将回退为字符串值。

给定字符串 cfg://handlers.myhandler.mykey.123，这将解析为 config_dict['handlers']['myhandler']['mykey']['123']。如果字符串被指定为 cfg://handlers.myhandler.mykey[123]，系统将尝试从 config_dict['handlers']['myhandler']['mykey'][123] 中提取值，并在尝试失败时回退为 config_dict['handlers']['myhandler']['mykey']['123']。

导入解析与定制导入器

导入解析默认使用内置的 *__import__()* 函数来执行导入。你可能想要将其替换为你自己的导入机制：如果是这样的话，你可以替换 DictConfigurator 或其超类 BaseConfigurator 类的 importer 属性。但是你必须小心谨慎，因为函数是从类中通过描述器方式来访问的。如果你使用 Python 可调用对象来执行导入，并且你希望在类层级而不是在实例层级上定义它，则你需要用 *staticmethod()* 来装饰它。例如:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

如果你是在一个配置器的 *实例* 上设置导入可调用对象则你不需要用 *staticmethod()* 来装饰。

配置 QueueHandler 和 QueueListener

如果你想要配置一个 *QueueHandler*，请注意它通常是与 *QueueListener* 一起使用的，你可以同时配置这两者。配置完成之后，QueueListener 实例将可作为所创建的处理器的 *listener* 属性来访问，而你也将可以使用 *getHandlerByName()* 来访问它并将你所使用的名称作为配置中的 QueueHandler 传入。用于配置这两者的字典规格显示在下面的 YAML 实例代码段中。

```
handlers:
  qhand:
    class: logging.handlers.QueueHandler
    queue: my.module.queue_factory
    listener: my.package.CustomListener
    handlers:
```

(续下页)

(接上页)

```
    - hand_name_1
    - hand_name_2
    ...
```

queue 和 listener 键是可选的。

如果存在 queue 键，相应的值可以是下列几项之一:

- 一个实现*Queue.put_nowait* 和*Queue.get* 公有 API 的对象。例如，这可以是一个*queue.Queue* 或其子类的具体实例，或者是一个由*multiprocessing.managers.SyncManager.Queue()* 获取的代理对象。

  这当然仅在你通过代码中构造或修改配置字典时才是可能的。

- 一个将被求值为可调用对象的字符串，当不带任何参数被调用时，它将返回要使用的队列实例。该可调用对象可以是一个*queue.Queue* 子类或是一个返回适当的队列实例的函数，如 my.module.queue_factory()。

- 一个带有 '()' 键的字典，它使用用户定义对象 中所介绍的通常方式构造。这样构造的结果应当是一个*queue.Queue* 实例。

如果没有 queue 键，则会创建并使用一个标准的未绑定*queue.Queue* 实例。

如果存在 listener 键，则相应的值可以是下列几项中的一个:

- 一个*logging.handlers.QueueListener* 的子类。这当然仅在你通过代码构造或修改配置字典时才是可能的。

- 一个将被求值为属于 QueueListener 的子类的类的字符串，例如 'my.package.CustomListener'。

- 一个带有 '()' 键的字典，它使用用户定义对象 中所介绍的通常方式构造。这样构造的结果应当是一个与 QueueListener 初始化器具有相同签名的可调用对象。

如果不存在 listener 键，则会使用*logging.handlers.QueueListener*。

在 handlers 键之下的值是配置中其他处理器的名称（未显示在上面的代码片段中），它们将被传给队列监听器。

任何队列处理器和监听器类都需要定义为具有与*QueueHandler* 和*QueueListener* 相同的初始化签名。

Added in version 3.12.

16.5.4 配置文件格式

fileConfig() 所能理解的配置文件格式是基于*configparser* 功能的。该文件必须包含 [loggers], [handlers] 和 [formatters] 等小节，它们通过名称来标识文件中定义的每种类型的实体。对于每个这样的实体，都有单独的小节来标识实体的配置方式。因此，对于 [loggers] 小节中名为 log01 的日志记录器，相应的配置详情保存在 [logger_log01] 小节中。类似地，对于 [handlers] 小节中名为 hand01 的处理器，其配置将保存在名为 [handler_hand01] 的小节中，而对于 [formatters] 小节中名为 form01 的格式化器，其配置将在名为 [formatter_form01] 的小节中指定。根日志记录器的配置必须在名为 [logger_root] 的小节中指定。

> **备注**
>
> *fileConfig()* API 比*dictConfig()* API 更旧因而没有提供涵盖日志记录特定方面的功能。例如，你无法配置*Filter* 对象，该对象使用*fileConfig()* 提供超出简单整数级别的消息过滤功能。如果你想要在你的日志记录配置中包含*Filter* 的实例，你将必须使用*dictConfig()*。请注意未来还将向*dictConfig()* 添加对配置功能的强化，因此值得考虑在方便的时候转换到这个新 API。

在文件中这些小节的例子如下所示。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

根日志记录器必须指定一个级别和一个处理器列表。根日志小节的例子如下所示。

```
[logger_root]
level=NOTSET
handlers=hand01
```

level 条目可以为 DEBUG, INFO, WARNING, ERROR, CRITICAL 或 NOTSET 之一。作为仅适用于根日志记录器的设置，NOTSET 表示将会记录所有消息。级别值会在 logging 包命名空间的上下文中进行求值。

handlers 条目是以逗号分隔的处理器名称列表，它必须出现于 [handlers] 小节并且在配置文件中有相应的小节。

对于根日志记录器以外的日志记录器，还需要某些附加信息。下面的例子演示了这些信息。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

level 和 handlers 条目的解释方式与根日志记录器的一致，不同之处在于如果一个非根日志记录器的级别被指定为 NOTSET，则系统会咨询更高层级的日志记录器来确定该日志记录器的有效级别。propagate 条目设为 1 表示消息必须从此日志记录器传播到更高层级的处理器，设为 0 表示消息 **不会**传播到更高层级的处理器。qualname 条目是日志记录器的层级通道名称，也就是应用程序获取日志记录器所用的名称。

指定处理器配置的小节说明如下。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

class 条目指明处理器的类（由 logging 包命名空间中的*eval()* 来确定）。level 会以与日志记录器相同的方式来解读，NOTSET 会被视为表示‘记录一切消息’。

formatter 条目指明此处理器的格式化器的键名称。如为空白，则会使用默认的格式化器 (logging._defaultFormatter)。如果指定了名称，则它必须出现于 [formatters] 小节并且在配置文件中有相应的小节。

args 条目，当在 logging 包命名空间的上下文中被求值 时，将是传给处理器类构造器的参数列表。请参阅相应处理器的构造器说明，或是下面的示例，以了解典型的条目是如何构造的。如果未提供，则其默认值为 ()。

可选的 kwargs 条目，当在 logging 包命名空间的上下文中被求值 时，将是传给处理器的构造器的关键字参数字典。如果未提供，则其默认值为 {}。

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')
```

(续下页)

(接上页)

```
[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

指定格式化器配置的小节说明如下。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s %(customfield)s
datefmt=
style=%
validate=True
defaults={'customfield': 'defaultvalue'}
class=logging.Formatter
```

用于格式化器配置的参数与字典规范格式化器部分 中的键相同。

`defaults` 条目，当在 `logging` 包的命名空间的上下文中求值 时，将是一个由自定义格式化字段的默认值组成的字典。如果未提供，则默认为 `None`。

备注

由于如上所述使用了*eval()*，因此使用*listen()* 通过套接字来发送和接收配置会导致潜在的安全风险。此风险仅限于相互间没有信任的多个用户在同一台机器上运行代码的情况；请参阅*listen()* 了解更多信息。

参见

模块*logging*
日志记录模块的 API 参考。

***logging.handlers* 模块**
日志记录模块附带的有用处理器。

16.6 logging.handlers --- 日志处理器

源代码: Lib/logging/handlers.py

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程
- 日志记录操作手册

这个包提供了以下有用的处理程序。请注意有三个处理程序类 (*StreamHandler*, *FileHandler* 和*NullHandler*) 实际上是在*logging* 模块本身定义的，但其文档与其他处理程序一同记录在此。

16.6.1 StreamHandler

StreamHandler 类位于核心*logging* 包，它可将日志记录输出发送到数据流例如 *sys.stdout*, *sys.stderr* 或任何文件型对象（或者更精确地说，任何支持 `write()` 和 `flush()` 方法的对象）。

class logging.**StreamHandler**(*stream=None*)

返回一个新的*StreamHandler* 类。如果指定了 *stream*，则实例将用它作为日志记录输出；在其他情况下将使用 *sys.stderr*。

emit(*record*)

如果指定了一个格式化器，它会被用来格式化记录。随后记录会被写入到*terminator* 之后的流中。如果存在异常信息，则会使用*traceback.print_exception()* 来格式化并添加到流中。

flush()

通过调用流的*flush()* 方法来刷新它。请注意 `close()` 方法是继承自*Handler* 的所以没有输出，因此有时可能需要显式地调用*flush()*。

setStream(*stream*)

将实例的流设为指定值，如果两者不一致的话。旧的流会在设置新的流之前被刷新。

参数
stream -- 处理程序应当使用的流。

返回
旧的流，如果流已被改变的话，或者如果未被改变则为 `None`。

Added in version 3.7.

terminator

当将已格式化的记录写入到流时被用作终止符的字符串。默认值为 `'\n'`。

如果你不希望以换行符终止，你可以将处理程序类实例的 `terminator` 属性设为空字符串。

在较早的版本中，终止符被硬编码为 `'\n'`。

Added in version 3.2.

16.6.2 FileHandler

FileHandler 类位于核心*logging* 包，它可将日志记录输出到磁盘文件中。它从*StreamHandler* 继承了输出功能。

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

返回一个*FileHandler* 类的新实例。将打开指定的文件并将其用作日志记录流。如果未指定 *mode*，则会使用 `'a'`。如果 *encoding* 不为 `None`，则会将其用作打开文件的编码格式。如果 *delay* 为真值，则文件打开会被推迟至第一次调用*emit()* 时。默认情况下，文件会无限增长。如果指定了 *errors*，它会被用于确定编码格式错误的处理方式。

在 3.6 版本发生变更: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

在 3.9 版本发生变更: 增加了 *errors* 形参。

close()

关闭文件。

emit(*record*)

将记录输出到文件。

请注意如果文件因日志记录在退出后终止而被关闭并且文件打开模式为'w'，则记录将不会被发送 (参见 bpo-42378)。

16.6.3 NullHandler

Added in version 3.1.

NullHandler 类位于核心*logging* 包，它不执行任何格式化或输出。它实际上是一个供库开发者使用的‘无操作’处理程序。

class logging.**NullHandler**

返回一个*NullHandler* 类的新实例。

emit(*record*)

此方法不执行任何操作。

handle(*record*)

此方法不执行任何操作。

createLock()

此方法会对锁返回 `None`，因为没有下层 I/O 的访问需要被序列化。

请参阅 library-config 了解有关如何使用*NullHandler* 的更多信息。

16.6.4 WatchedFileHandler

WatchedFileHandler 类位于*logging.handlers* 模块，这个 `FileHandler` 用于监视它所写入日志记录的文件。如果文件发生变化，它会被关闭并使用文件名重新打开。

发生文件更改可能是由于使用了执行文件轮换的程序例如 *newsyslog* 和 *logrotate*。这个处理程序在 Unix/Linux 下使用，它会监视文件来查看自上次发出数据后是否有更改。(如果文件的设备或 inode 发生变化就认为已被更改。) 如果文件被更改，则会关闭旧文件流，并再打开文件以获得新文件流。

这个处理程序不适合在 Windows 下使用，因为在 Windows 下打开的日志文件无法被移动或改名——日志记录会使用排他的锁来打开文件——因此这样的处理程序是没有必要的。此外，*ST_INO* 在 Windows 下不受支持；`stat()` 将总是为该值返回零。

class logging.handlers.**WatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

返回一个`WatchedFileHandler`类的新实例。将打开指定的文件并将其用作日志记录流。如果未指定 *mode*，则会使用 `'a'`。如果 *encoding* 不为 `None`，则会将其用作打开文件的编码格式。如果 *delay* 为真值，则文件打开会被推迟至第一次调用`emit()`时。默认情况下，文件会无限增长。如果提供了 *errors*，它会被用于确定编码格式错误的处理方式。

在 3.6 版本发生变更: 除了字符串值，也接受`Path`对象作为 *filename* 参数。

在 3.9 版本发生变更: 增加了 *errors* 形参。

reopenIfNeeded()

检查文件是否已更改。如果已更改，则会刷新并关闭现有流然后重新打开文件，这通常是将记录输出到文件的先导操作。

Added in version 3.6.

emit(*record*)

将记录输出到文件，但如果文件已更改则会先调用`reopenIfNeeded()`来重新打开它。

16.6.5 BaseRotatingHandler

`BaseRotatingHandler`类位于`logging.handlers`模块中，它是轮换文件处理程序类`RotatingFileHandler`和`TimedRotatingFileHandler`的基类。你不需要实例化此类，但它具有你可能需要重写的属性和方法。

class logging.handlers.**BaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*, *errors=None*)

类的形参与 `FileHandler` 的相同。其属性有:

namer

如果此属性被设为一个可调用对象，则`rotation_filename()`方法会委托给该可调用对象。传给该可调用对象的形参与传给`rotation_filename()`的相同。

> **备注**
>
> namer 函数会在轮换期间被多次调用，因此它应当尽可能的简单快速。它还应当对给定的输入每次都返回相同的输出，否则轮换行为可能无法按预期工作。
>
> 还有一点值得注意的是当使用命名器来保存文件名中要在轮换中使用的特定属性时必须小心处理。例如，`RotatingFileHandler`会要求有一组名称中包含连续整数的日志文件，以便轮换的效果能满足预期，而`TimedRotatingFileHandler`会通过确定要删除的最旧文件（根据传递纵使中处理器的初始化器的 `backupCount` 形参）来删除旧的日志文件。为了达成这样的效果，文件名应当是可以使用文件名的日期/时间部分来排序的，而且命名器需要遵循此排序。（如果想使用不遵循此规则的命名器，则需要在一个重写了`getFilesToDelete()`方法的`TimedRotatingFileHandler`的子类中使用它以便与自定义的命名规则进行配合。）

Added in version 3.3.

rotator

如果此属性被设为一个可调用对象，则`rotate()`方法会委托给该可调用对象。传给该可调用对象的形参与传给`rotate()`的相同。

Added in version 3.3.

rotation_filename(*default_name*)

当轮换时修改日志文件的文件名。

提供该属性以便可以提供自定义文件名。

默认实现会调用处理程序的'namer' 属性，如果它是可调用对象的话，并传给它默认的名称。如果该属性不是可调用对象 (默认值为 None)，则将名称原样返回。

参数

default_name -- 日志文件的默认名称。

Added in version 3.3.

rotate(*source*, *dest*)

当执行轮换时，轮换当前日志。

默认实现会调用处理程序的'rotator' 属性，如果它是可调用对象的话，并传给它 source 和 dest 参数。如果该属性不是可调用对象 (默认值为 None)，则将源简单地重命名为目标。

参数

- **source** -- 源文件名。这通常为基本文件名，例如'test.log'。
- **dest** -- 目标文件名。这通常是源被轮换后的名称，例如'test.log.1'。

Added in version 3.3.

该属性存在的理由是让你不必进行子类化——你可以使用与*RotatingFileHandler*和*TimedRotatingFileHandler*的实例相同的可调用对象。如果 namer 或 rotator 可调用对象引发了异常，将会按照与 emit() 调用期间的任何其他异常相同的方式来处理，例如通过处理程序的 handleError() 方法。

如果你需要对轮换进程执行更多的修改，你可以重写这些方法。

请参阅 cookbook-rotator-namer 获取具体示例。

16.6.6 RotatingFileHandler

*RotatingFileHandler*类位于*logging.handlers*模块，它支持磁盘日志文件的轮换。

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*)

返回一个*RotatingFileHandler*类的新实例。将打开指定的文件并将其用作日志记录流。如果未指定 *mode*，则会使用 'a'。如果 *encoding* 不为 None，则会将其用作打开文件的编码格式。如果 *delay* 为真值，则文件打开会被推迟至第一次调用*emit()*。默认情况下，文件会无限增长。如果提供了 *errors*，它会被用于确定编码格式错误的处理方式。

你可以使用 *maxBytes* 和 *backupCount* 值来允许文件以预定的大小执行 *rollover*。当即将超出预定大小时，将关闭旧文件并打开一个新文件用于输出。只要当前日志文件长度接近 *maxBytes* 就会发生轮换；但是如果 *maxBytes* 或 *backupCount* 两者之一的值为零，就不会发生轮换，因此你通常要设置 *backupCount* 至少为 1，而 *maxBytes* 不能为零。当 *backupCount* 为非零值时，系统将通过为原文件名添加扩展名'.1', '.2' 等来保存旧日志文件。例如，当 *backupCount* 为 5 而基本文件名为 app.log 时，你将得到 app.log, app.log.1, app.log.2 直至 app.log.5。当前被写入的文件总是 app.log。当此文件写满时，它会被关闭并重户名为 app.log.1，而如果文件 app.log.1, app.log.2 等存在，则它们会被分别重命名为 app.log.2, app.log.3 等等。

在 3.6 版本发生变更: 除了字符串值，也接受*Path*对象作为 *filename* 参数。

在 3.9 版本发生变更: 增加了 *errors* 形参。

doRollover()

执行上文所描述的轮换。

emit(*record*)

将记录输出到文件，以适应上文所描述的轮换。

16.6.7 TimedRotatingFileHandler

TimedRotatingFileHandler 类位于*logging.handlers* 模块，它支持基于特定时间间隔的磁盘日志文件轮换。

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*, *errors=None*)

返回一个新的*TimedRotatingFileHandler* 类实例。指定的文件会被打开并用作日志记录的流。对于轮换操作它还会设置文件名前缀。轮换的发生是基于 *when* 和 *interval* 的积。

你可以使用 *when* 来指定 *interval* 的类型。可能的值列表如下。请注意它们不是大小写敏感的。

值	间隔类型	如果/如何使用 *atTime*
`'S'`	秒	忽略
`'M'`	分钟	忽略
`'H'`	小时	忽略
`'D'`	天	忽略
`'W0'-'W6'`	工作日 (0= 星期一)	用于计算初始轮换时间
`'midnight'`	如果未指定 *atTime* 则在午夜执行轮换，否则将使用 *atTime*。	用于计算初始轮换时间

当使用基于星期的轮换时，星期一为'W0'，星期二为'W1'，以此类推直至星期日为'W6'。在这种情况下，传入的 *interval* 值不会被使用。

系统将通过为文件名添加扩展名来保存旧日志文件。扩展名是基于日期和时间的，根据轮换间隔的长短使用 strftime 格式 `%Y-%m-%d_%H-%M-%S` 或是其中有变动的部分。

当首次计算下次轮换的时间时（即当处理程序被创建时），现有日志文件的上次被修改时间或者当前时间会被用来计算下次轮换的发生时间。

如果 *utc* 参数为真值，将使用 UTC 时间；否则会使用本地时间。

如果 *backupCount* 不为零，则最多将保留 *backupCount* 个文件，而如果当轮换发生时创建了更多的文件，则最旧的文件会被删除。删除逻辑使用间隔时间来确定要删除的文件，因此改变间隔时间可能导致旧文件被继续保留。

如果 *delay* 为真值，则会将文件打开延迟到首次调用*emit()* 的时候。

如果 *atTime* 不为 `None`，则它必须是一个 `datetime.time` 的实例，该实例指定轮换在一天内的发生时间，用于轮换被设为“在午夜”或“在每星期的某一天”之类的情况。请注意在这些情况下，*atTime* 值实际上会被用于计算*初始*轮换，而后续轮换将会通过正常的间隔时间计算来得出。

如果指定了 *errors*，它会被用来确定编码错误的处理方式。

> **备注**
>
> 初始轮换时间的计算是在处理程序被初始化时执行的。后续轮换时间的计算则仅在轮换发生时执行，而只有当提交输出时轮换才会发生。如果不记住这一点，你就可能会感到困惑。例如，如果设置时间间隔为“每分钟”，那并不意味着你总会看到（文件名中）带有间隔一分钟时间的日志文件；如果在应用程序执行期间，日志记录输出的生成频率高于每分钟一次，*那么*你可以预期看到间隔一分钟时间的日志文件。另一方面，如果（假设）日志记录消息每五分钟才输出一次，那么文件时间将会存在对应于没有输出（因而没有轮换）的缺失。

在 3.4 版本发生变更: 添加了 *atTime* 形参。

在 3.6 版本发生变更: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

在 3.9 版本发生变更: 增加了 *errors* 形参。

doRollover()

执行上文所描述的轮换。

emit(*record*)

将记录输出到文件，以适应上文所描述的轮换。

getFilesToDelete()

返回由应当作为轮转的一部分被删除的文件名组成的列表。它们是由处理程序写入的最旧的备份日志文件的绝对路径。

16.6.8 SocketHandler

SocketHandler 类位于*logging.handlers* 模块，它会将日志记录输出发送到网络套接字。基类所使用的是 TCP 套接字。

class logging.handlers.**SocketHandler**(*host*, *port*)

返回一个*SocketHandler* 类的新实例，该实例旨在与使用 *host* 与 *port* 给定地址的远程主机进行通信。

在 3.4 版本发生变更: 如果 `port` 指定为 `None`，会使用 `host` 中的值来创建一个 Unix 域套接字——在其他情况下，则会创建一个 TCP 套接字。

close()

关闭套接字。

emit()

对记录的属性字典执行封存并以二进制格式将其写入套接字。如果套接字存在错误，则静默地丢弃数据包。如果连接在此之前丢失，则重新建立连接。要在接收端将记录解封并输出到*LogRecord*，请使用*makeLogRecord()* 函数。

handleError()

处理在*emit()* 期间发生的错误。最可能的原因是连接丢失。关闭套接字以便我们能在下次事件时重新尝试。

makeSocket()

这是一个工厂方法，它允许子类定义它们想要的套接字的准确类型。默认实现会创建一个 TCP 套接字 (*socket.SOCK_STREAM*)。

makePickle(*record*)

将记录的属性字典封存为带有长度前缀的二进制格式，并将其返回以准备通过套接字进行传输。此操作在细节上相当于:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

请注意封存操作不是绝对安全的。如果你关心安全问题，你可能会想要重写此方法以实现更安全的机制。例如，你可以使用 HMAC 对封存对象进行签名然后在接收端验证它们，或者你也可以在接收端禁用全局对象的解封操作。

send(*packet*)

将封存后的字节串 *packet* 发送到套接字。所发送字节串的格式与*makePickle()* 文档中的描述一致。

此函数允许部分发送，这可能会在网络繁忙时发生。

createSocket()

尝试创建一个套接字；失败时将使用指数化回退算法处理。在失败初次发生时，处理程序将丢弃它正尝试发送的消息。当后续消息交由同一实例处理时，它将不会尝试连接直到经过一段时间以后。默认形参设置为初始延迟一秒，如果在延迟之后连接仍然无法建立，处理程序将每次把延迟翻倍直至达到 30 秒的最大值。

此行为由下列处理程序属性控制:

- `retryStart` (初始延迟，默认为 1.0 秒)。
- `retryFactor` (倍数，默认为 2.0)。
- `retryMax` (最大延迟，默认为 30.0 秒)。

这意味着如果远程监听器在处理程序被使用 *之后* 启动，你可能会丢失消息（因为处理程序在延迟结束之前甚至不会尝试连接，而在延迟期间静默地丢弃消息）。

16.6.9 DatagramHandler

DatagramHandler 类位于 *logging.handlers* 模块，它继承自 *SocketHandler*，支持通过 UDP 套接字发送日志记录消息。

class logging.handlers.**DatagramHandler**(*host*, *port*)

返回一个 *DatagramHandler* 类的新实例，该实例旨在与使用 *host* 与 *port* 给定地址的远程主机进行通信。

> **备注**
>
> 由于 UDP 不是流式协议，在该处理器与 *host* 之前不存在持久连接。因为这个原因，当使用网络套接字时，每当有事件被写入日志时都可能要进行 DNS 查询，这会给系统带来一些延迟。如果这对你有影响，你可以自己执行查询并使用已查询到的 IP 地址而不是主机名来初始化这个处理器。

在 3.4 版本发生变更: 如果 `port` 指定为 `None`，会使用 `host` 中的值来创建一个 Unix 域套接字——在其他情况下，则会创建一个 UDP 套接字。

emit()

对记录的属性字典执行封存并以二进制格式将其写入套接字。如果套接字存在错误，则静默地丢弃数据包。要在接收端将记录解封并输出到 *LogRecord*，请使用 *makeLogRecord()* 函数。

makeSocket()

SocketHandler 的工厂方法会在此被重写以创建一个 UDP 套接字 (*socket.SOCK_DGRAM*)。

send(*s*)

将封存后的字节串发送到套接字。所发送字节串的格式与 *SocketHandler.makePickle()* 文档中的描述一致。

16.6.10 SysLogHandler

SysLogHandler 类位于 *logging.handlers* 模块，它支持将日志记录消息发送到远程或本地 Unix syslog。

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT)*, *facility=LOG_USER*, *socktype=socket.SOCK_DGRAM*)

返回一个 *SysLogHandler* 类的新实例用来与通过 *address* 以 `(host, port)` 元组形式给出地址的远程 Unix 机器进行通讯。如果未指定 *address*，则使用 `('localhost', 514)`。该地址会被用于打开套接字。提供 `(host, port)` 元组的一种替代方式是提供字符串形式的地址，例如'/dev/log'。在这种情况下，会使用 Unix 域套接字将消息发送到 syslog。如果未指定 *facility*，则使用 `LOG_USER`。打开的套接字类型取决于 *socktype* 参数，该参数的默认值为 *socket.SOCK_DGRAM* 即打开一个 UDP 套接字。要打开一个 TCP 套接字（用来配合较新的 syslog 守护程序例如 rsyslog 使用），请指定值为 *socket.SOCK_STREAM*。

请注意如果你的服务器不是在 UDP 端口 514 上进行侦听，则 *SysLogHandler* 可能无法正常工作。在这种情况下，请检查你应当为域套接字所使用的地址——它依赖于具体的系统。例如，在 Linux 上通常为'/dev/log' 而在 OS/X 上则为'/var/run/syslog'。你需要检查你的系统平台并使用适当的地址（如果你的应用程序需要在多个平台上运行则可能需要在运行时进行这样的检查）。在 Windows 上，你大概必须要使用 UDP 选项。

> **备注**
>
> 在 macOS 12.x (Monterey) 上，Apple 修改了其 syslog 守护进程的行为——它不再监听某个域套接字。因此，你不能再预期*SysLogHandler* 在此系统上有效。
>
> 请参阅 gh-91070 了解更多信息。

在 3.2 版本发生变更: 添加了 *socktype*。

close()

关闭连接远程主机的套接字。

createSocket()

尝试创建一个套接字，如果它不是一个数据报套接字，则将其连接到另一端。此方法会在处理器初始化期间被调用，但是如果此时另一端还没有监听则它不会被视为出错——如果此时套接字还不存在，此方法将在发出事伯时再次被调用。

Added in version 3.11.

emit(*record*)

记录会被格式化，然后发送到 syslog 服务器。如果存在异常信息，则它 不会被发送到服务器。

在 3.2.1 版本发生变更: (参见: bpo-12168。) 在较早的版本中，发送至 syslog 守护程序的消息总是以一个 NUL 字节结束，因为守护程序的早期版本期望接收一个以 NUL 结束的消息——即使它不包含于对应的规范说明 (**RFC 5424**)。这些守护程序的较新版本不再期望接收 NUL 字节，如果其存在则会将其去除，而最新的守护程序（更紧密地遵循 RFC 5424）会将 NUL 字节作为消息的一部分传递出去。

为了在面对所有这些不同守护程序行为时能够更方便地处理 syslog 消息，通过使用类层级属性 `append_nul`，添加 NUL 字节的操作已被作为可配置项。该属性默认为 `True` (保留现有行为) 但可在 `SysLogHandler` 实例上设为 `False` 以便让实例 不会添加 NUL 结束符。

在 3.3 版本发生变更: (参见: bpo-12419。) 在较早的版本中，没有"ident" 或"tag" 前缀工具可以用来标识消息的来源。现在则可以使用一个类层级属性来设置它，该属性默认为 `""` 表示保留现有行为，但可在 `SysLogHandler` 实例上重写以便让实例不会为所处理的每条消息添加标识。请注意所提供的标识必须为文本而非字节串，并且它会被原封不动地添加到消息中。

encodePriority(*facility*, *priority*)

将功能和优先级编码为一个整数。你可以传入字符串或者整数——如果传入的是字符串，则会使用内部的映射字典将其转换为整数。

符号 `LOG_` 的值在*SysLogHandler* 中定义并且是 `sys/syslog.h` 头文件中所定义值的镜像。

优先级

名称（字符串）	符号值
`alert`	LOG_ALERT
`crit` 或 `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` 或 `panic`	LOG_EMERG
`err` 或 `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` 或 `warning`	LOG_WARNING

设备

名称（字符串）	符号值
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)

将日志记录级别名称映射到 syslog 优先级名称。如果你使用自定义级别，或者如果默认算法不适合你的需要，你可能需要重写此方法。默认算法将 DEBUG, INFO, WARNING, ERROR 和 CRITICAL 映射到等价的 syslog 名称，并将所有其他级别名称映射到'warning'。

16.6.11 NTEventLogHandler

NTEventLogHandler 类位于 *logging.handlers* 模块，它支持将日志记录消息发送到本地 Windows NT, Windows 2000 或 Windows XP 事件日志。在你使用它之前，你需要安装 Mark Hammond 的 Python Win32 扩展。

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

返回一个 *NTEventLogHandler* 类的新实例。*appname* 用来定义出现在事件日志中的应用名称。将使用此名称创建适当的注册表条目。*dllname* 应当给出要包含在日志中的消息定义的.dll 或.exe 的完整限定路径名称（如未指定则会使用 'win32service.pyd' ——此文件随 Win32 扩展安装且包含一些基本的消息定义占位符。请注意使用这些占位符将使你的事件日志变得很大，因为整个消息源都会被放入日志。如果你希望有较小的日志，你必须自行传入包含你想要在事件日志中使用的消息定义的.dll 或.exe 名称）。*logtype* 为 'Application', 'System' 或 'Security' 之一，且默认值为 'Application'。

close()

这时，你就可以从注册表中移除作为事件日志条目来源的应用名称。但是，如果你这样做，你将无法如你所预期的那样在事件日志查看器中看到这些事件——它必须能访问注册表来获取.dll 名称。当前版本并不会这样做。

emit(*record*)

确定消息 ID，事件类别和事件类型，然后将消息记录到 NT 事件日志中。

getEventCategory(*record*)

返回记录的事件类别。如果你希望指定你自己的类别就要重写此方法。此版本将返回 0。

getEventType(*record*)

返回记录的事件类型。如果你希望指定你自己的类型就要重写此方法。此版本将使用处理程序的 typemap 属性来执行映射，该属性在 __init__() 被设置为一个字典，其中包含 DEBUG, INFO, WARNING, ERROR 和 CRITICAL 的映射。如果你使用你自己的级别，你将需要重写此方法或者在处理程序的 *typemap* 属性中放置一个合适的字典。

getMessageID(*record*)

返回记录的消息 ID。如果你使用你自己的消息，你可以通过将 *msg* 传给日志记录器作为 ID 而非格式字符串实现此功能。然后，你可以在这里使用字典查找来获取消息 ID。此版本将返回 1，是 `win32service.pyd` 中的基本消息 ID。

16.6.12 SMTPHandler

SMTPHandler 类位于 *logging.handlers* 模块，它支持将日志记录消息通过 SMTP 发送到一个电子邮件地址。

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)

返回一个 *SMTPHandler* 类的新实例。该实例使用电子邮件的发件人、收件人地址和主题行进行初始化。*toaddrs* 应当为字符串列表。要指定一个非标准 SMTP 端口，请使用 (host, port) 元组格式作为 *mailhost* 参数。如果你使用一个字符串，则会使用标准 SMTP 端口。如果你的 SMTP 服务器要求验证，你可以指定一个 (username, password) 元组作为 *credentials* 参数。

要指定使用安全协议 (TLS)，请传入一个元组作为 *secure* 参数。这将仅在提供了验证凭据时才能被使用。元组应当或是一个空元组，或是一个包含密钥文件名的单值元组，或是一个包含密钥文件和证书文件的 2 值元组。(此元组会被传给 *smtplib.SMTP.starttls()* 方法。)

可以使用 *timeout* 参数为与 SMTP 服务器的通信指定超时限制。

在 3.3 版本发生变更: 增加了 *timeout* 形参。

emit(*record*)

对记录执行格式化并将其发送到指定的地址。

getSubject(*record*)

如果你想要指定一个基于记录的主题行，请重写此方法。

16.6.13 MemoryHandler

MemoryHandler 类位于 *logging.handlers* 模块，它支持在内存中缓冲日志记录，并定期将其刷新到 *target* 处理程序中。刷新会在缓冲区满的时候，或是在遇到特定或更高严重程度事件的时候发生。

MemoryHandler 是更通用的 *BufferingHandler* 的子类，后者属于抽象类。它会在内存中缓冲日志记录。当每条记录被添加到缓冲区时，会通过调用 `shouldFlush()` 来检查缓冲区是否应当刷新。如果应当刷新，则要使用 `flush()` 来执行刷新。

class logging.handlers.**BufferingHandler**(*capacity*)

使用指定容量的缓冲区初始化处理程序。这里，*capacity* 是指缓冲的日志记录数量。

emit(*record*)

将记录添加到缓冲区。如果 *shouldFlush()* 返回真值，则会调用 *flush()* 来处理缓冲区。

flush()

对于 *BufferingHandler* 的实例，刷新意味着将缓冲区设为一个空列表。此方法可被覆盖以实现更有用的刷新行为。

shouldFlush(*record*)

如果缓冲区容量已满则返回 `True`。可以重写此方法以实现自定义的刷新策略。

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)

返回一个 *MemoryHandler* 类的新实例。该实例使用 *capacity* 指定的缓冲区大小（要缓冲的记录数量）来初始化。如果 *flushLevel* 未指定，则使用 `ERROR`。如果未指定 *target*，则需要在此处理程序执行任何实际操作之前使用 *setTarget()* 来设置目标。如果 *flushOnClose* 指定为 `False`，则当处理程序被关闭时 不会刷新缓冲区。如果未指定或指定为 `True`，则当处理程序被关闭时将会发生之前的缓冲区刷新行为。

在 3.6 版本发生变更: 增加了 *flushOnClose* 形参。

close()

调用*flush()*，设置目标为 None 并清空缓冲区。

flush()

对于*MemoryHandler* 的实例，刷新意味着将缓冲的记录发送到目标，如果目标存在的话。当缓冲的记录被发送到目标时缓冲区也将被清空。如果你想要不同的行为请重写此方法。

setTarget(*target*)

设置此处理程序的目标处理程序。

shouldFlush(*record*)

检测缓冲区是否已满或是有记录为 *flushLevel* 或更高级别。

16.6.14 HTTPHandler

HTTPHandler 类位于*logging.handlers* 模块，它支持使用 GET 或 POST 语义将日志记录消息发送到 Web 服务器。

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

返回一个*HTTPHandler* 类的新实例。*host* 可以为 host:port 的形式，如果你需要使用指定端口号的话。如果没有指定 *method*，则会使用 GET。如果 *secure* 为真值，则将使用 HTTPS 连接。*context* 形参可以设为一个*ssl.SSLContext* 实例以配置用于 HTTPS 连接的 SSL 设置。如果指定了 *credentials*，它应当为包含 userid 和 password 的二元组，该元组将被放入使用 Basic 验证的 HTTP 'Authorization' 标头中。如果你指定了 credentials，你还应当指定 secure=True 这样你的 userid 和 password 就不会以明文在线路上传输。

在 3.5 版本发生变更: 增加了 *context* 形参。

mapLogRecord(*record*)

基于 record 提供一个字典，它将被执行 URL 编码并发送至 Web 服务器。默认实现仅返回 record.__dict__。在只需将*LogRecord* 的某个子集发送至 Web 服务器，或者需要对发送至服务器的内容进行更多定制时可以重写此方法。

emit(*record*)

将记录以 URL 编码字典的形式发送至 Web 服务器。*mapLogRecord()* 方法会被用来将要发送的记录转换为字典。

> **备注**
>
> 由于记录发送至 Web 服务器所需的预处理与通用的格式化操作不同，使用*setFormatter()* 来指定一个*Formatter* 用于*HTTPHandler* 是没有效果的。此处理程序不会调用*format()*，而是调用*mapLogRecord()* 然后再调用*urllib.parse.urlencode()* 来以适合发送至 Web 服务器的形式对字典进行编码。

16.6.15 QueueHandler

Added in version 3.2.

QueueHandler 类位于*logging.handlers* 模块，它支持将日志记录消息发送到一个队列，例如在*queue* 或*multiprocessing* 模块中实现的队列。

配合*QueueListener* 类使用，*QueueHandler* 可被用来使处理程序在与执行日志记录的线程不同的线程上完成工作。这对 Web 应用程序以及其他服务于客户端的线程需要尽可能快地响应的服务应用程序来说很重要，任何潜在的慢速操作（例如通过*SMTPHandler* 发送邮件）都要在单独的线程上完成。

class logging.handlers.**QueueHandler**(*queue*)

返回一个*QueueHandler* 类的新实例。该实例使用队列来初始化以向其发送消息。*queue* 可以为任何队列类对象；它由*enqueue()* 方法来使用，该方法需要知道如何向其发送消息。队列 不要求具有任务跟踪 API，这意味着你可以为 *queue* 使用*SimpleQueue* 实例。

> **备注**
>
> 如果你在使用*multiprocessing*，则你应当避免使用*SimpleQueue* 而要改用*multiprocessing.Queue*。

emit(*record*)

将准备 LogRecord 的结果排入队列。如果发生了异常（例如由于有界队列已满），则会调用*handleError()* 方法来处理错误。这可能导致记录被静默地丢弃（当*logging.raiseExceptions* 为 `False` 时）或者消息被打印到 `sys.stderr`（当*logging.raiseExceptions* 为 `True` 时）。

prepare(*record*)

准备用于队列的记录。此方法返回的对象会被排入队列。

该基本实现会对记录进行格式化以合并消息、参数、异常和栈信息，如果它们存在的话。它还会从记录中原地移除不可 pickle 的条目。具体来说，它会用合并后的消息（通过调用处理器的*format()* 方法获得）覆盖记录的 `msg` 和 `message` 属性，并将 `args`, `exc_info` 和 `exc_text` 属性设置为 `None`。

如果你想要将记录转换为 dict 或 JSON 字符串，或者发送记录被修改后的副本而让初始记录保持原样，则你可能会想要重写此方法。

> **备注**
>
> 该基本实现会使用这些参数对消息进行格式化，将 `message` 和 `msg` 属性设置为已格式化的消息并将 `args` 和 `exc_text` 属性设置为 `None` 以允许 pickle 操作并防止更多的格式化尝试。这意味着*QueueListener* 一方的处理器将没有自定义格式化所需的信息，例如异常信息等。你可能会想要子类化 `QueueHandler` 并重写此方法以便避免将 `exc_text` 设置为 `None`。请注意对 `message` / `msg` / `args` 的改变与确保记录可以 pickle 是相关联的，根据你的 `args` 是否可以 pickle 你将可能或不可能避免这样做。（请注意你可能必须不仅要考虑你自己的代码还要考虑你所使用的任何库中的代码。）

enqueue(*record*)

使用 `put_nowait()` 将记录排入队列；如果你想要使用阻塞行为，或超时设置，或自定义的队列实现，则你可能会想要重写此方法。

listener

当通过使用*dictConfig()* 的配置创建时，该属性将包含一个*QueueListener* 实例供此处理器使用。在其他情况下，它将为 `None`。

Added in version 3.12.

16.6.16 QueueListener

Added in version 3.2.

QueueListener 类位于*logging.handlers* 模块，它支持从一个队列接收日志记录消息，例如在*queue* 或*multiprocessing* 模块中实现的队列。消息是在内部线程中从队列接收并在同一线程上传递到一个或多个处理程序进行处理的。尽管*QueueListener* 本身并不是一个处理程序，但由于它要与*QueueHandler* 配合工作，因此也在此处介绍。

配合*QueueHandler* 类使用，*QueueListener* 可被用来使处理程序在与执行日志记录的线程不同的线程上完成工作。这对 Web 应用程序以及其他服务于客户端的线程需要尽可能快地响应的服务应用程序来说很重要，任何潜在的慢速动作（例如通过*SMTPHandler* 发送邮件）都要在单独的线程上完成。

class logging.handlers.**QueueListener**(*queue*, **handlers*, *respect_handler_level=False*)

返回一个*QueueListener* 类的新实例。该实例初始化时要传入一个队列以向其发送消息，还要传入一个处理程序列表用来处理放置在队列中的条目。队列可以是任何队列类对象；它会被原样传

给*dequeue()*方法，该方法需要知道如何从其获取消息。队列 不要求具有任务跟踪 API（但如提供则会使用它），这意味着你可以为 *queue* 使用*SimpleQueue*实例。

备注

如果你在使用*multiprocessing*，则你应当避免使用*SimpleQueue*而要改用*multiprocessing.Queue*。

如果 `respect_handler_level` 为 `True`，则在决定是否将消息传递给处理程序之前会遵循处理程序的级别（与消息的级别进行比较）；在其他情况下，其行为与之前的 Python 版本一致——总是将每条消息传递给每个处理程序。

在 3.5 版本发生变更: 增加了 `respect_handler_level` 参数。

dequeue(*block*)

从队列移出一条记录并将其返回，可以选择阻塞。

基本实现使用 `get()`。如果你想要使用超时设置或自定义的队列实现，则你可能会想要重写此方法。

prepare(*record*)

准备一条要处理的记录。

该实现只是返回传入的记录。如果你想要对记录执行任何自定义的 marshal 操作或在将其传给处理程序之前进行调整，则你可能会想要重写此方法。

handle(*record*)

处理一条记录。

此方法简单地循环遍历处理程序，向它们提供要处理的记录。实际传给处理程序的对象就是从*prepare()*返回的对象。

start()

启动监听器。

此方法启动一个后台线程来监视 LogRecords 队列以进行处理。

在 3.13.3 (unreleased) 版本发生变更: Raises *RuntimeError* if called and the listener is already running.

stop()

停止监听器。

此方法要求线程终止，然后等待它完成终止操作。请注意在你的应用程序退出之前如果你没有调用此方法，则可能会有一些记录在留在队列中，它们将不会被处理。

enqueue_sentinel()

将一个标记写入队列以通知监听器退出。此实现会使用 `put_nowait()`。如果你想要使得超时设置或自定义的队列实现，则你可能会想要重写此方法。

Added in version 3.3.

参见

模块*logging*
日志记录模块的 API 参考。

***logging.config* 模块**
日志记录模块的配置 API 。

16.7 platform --- 访问底层平台的标识数据

源代码： Lib/platform.py

备注

特定平台按字母顺序排列，Linux 包括在 Unix 小节之中。

16.7.1 跨平台

platform.**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)

查询给定的可执行文件（默认为 Python 解释器二进制码文件）来获取各种架构信息。

返回一个元素 `(bits, linkage)`，其中包含可执行文件所使用的位架构和链接格式信息。这两个值均以字符串形式返回。

无法确定的值将返回为形参预设所给出的值。如果给出的位数为 `''`，则会使用 `sizeof(pointer)` (或者当 Python 版本 < 1.5.2 时为 `sizeof(long)`) 作为所支持的指针大小的提示。

此函数依赖于系统的 `file` 命令来执行实际的操作。这在几乎所有 Unix 平台和某些非 Unix 平台上只有当可执行文件指向 Python 解释器时才可用。当以上要求不满足时将会使用合理的默认值。

备注

在 macOS (也许还有其他平台) 上，可执行文件可能是包含多种架构的通用文件。

要获取当前解释器的“64 位性”，更可靠的做法是查询*sys.maxsize* 属性:

```
is_64bits = sys.maxsize > 2**32
```

platform.**machine**()

返回机器类型，例如 `'AMD64'`。如果该值无法确定则会返回一个空字符串。

platform.**node**()

返回计算机的网络名称（可能不是完整限定名称!）。如果该值无法确定则会返回一个空字符串。

platform.**platform**(*aliased=False*, *terse=False*)

返回一个标识底层平台的字符串，其中带有尽可能多的有用信息。

输出信息的目标是“人类易读”而非机器易解析。它在不同平台上可能看起来不一致，这是有意为之的。

如果 *aliased* 为真值，此函数将使用各种平台不同与其通常名称的别名来报告系统名称，例如 SunOS 将被报告为 Solaris。*system_alias()* 函数将被用于实现此功能。

将 *terse* 设为真值将导致此函数只返回标识平台所必须的最小量信息。

在 3.8 版本发生变更: 在 macOS 上，此函数现在会在*mac_ver()* 返回的发布版字符串非空时使用它，以便获取 macOS 版本而非 darwin 版本。

platform.**processor**()

返回（真实的）处理器名称，例如 `'amdk6'`。

如果该值无法确定则将返回空字符串。请注意许多平台都不提供此信息或是简单地返回与*machine()* 相同的值。NetBSD 则会提供此信息。

platform.**python_build**()

返回一个元组 `(buildno, builddate)`，以字符串表示的 Python 编译代码和日期。

platform.**python_compiler**()

返回一个标识用于编译 Python 的编译器的的字符串。

platform.**python_branch**()

返回一个标识 Python 实现的 SCM 分支的字符串。

platform.**python_implementation**()

返回一个标识 Python 实现的字符串。可能的返回值有: 'CPython', 'IronPython', 'Jython', 'PyPy'。

platform.**python_revision**()

返回一个标识 Python 实现的 SCM 修订版的字符串。

platform.**python_version**()

将 Python 版本以字符串 `'major.minor.patchlevel'` 形式返回。

请注意此返回值不同于 Python `sys.version`，它将总是包括 patchlevel (默认为 0)。

platform.**python_version_tuple**()

将 Python 版本以字符串元组 `(major, minor, patchlevel)` 形式返回。

请注意此返回值不同于 Python `sys.version`，它将总是包括 patchlevel (默认为 `'0'`)。

platform.**release**()

返回系统的发布版本，例如 `'2.2.0'` 或 `'NT'`，如果该值无法确定则将返回一个空字符串。

platform.**system**()

返回系统平台/OS 的名称，例如 `'Linux'`, `'Darwin'`, `'Java'`, `'Windows'`。如果该值无法确定则将返回一个空字符串。

在 iOS 和 Android 上，这将返回面向用户的 OS 名称 (即 `'iOS'`, `'iPadOS'` 或 `'Android'`)。要获取内核名称 (`'Darwin'` 或 `'Linux'`)，请使用*os.uname()*。

platform.**system_alias**(*system*, *release*, *version*)

返回别名为某些系统所使用的常见营销名称的 `(system, release, version)`。它还会在可能导致混淆的情况下对信息进行一些重排序操作。

platform.**version**()

返回系统的发布版本信息，例如 `'#3 on degas'`。如果该值无法确定则将返回一个空字符串。

在 iOS 和 Android 上，这将是面向用户的 OS 版本号。要获取 Darwin 或 Linux 内核版本号，请使用*os.uname()*。

platform.**uname**()

具有高可移植性的 uname 接口。返回包含六个属性的*namedtuple()*: *system*, *node*, *release*, *version*, *machine* 和*processor*。

processor 将根据需要延后获取。

注意: 前两个属性名称与*os.uname()* 所提供的名称不同，后者是被命名为 `sysname` 和 `nodename`。

无法确定的条目会被设为 `''`。

在 3.3 版本发生变更: 结果由元组改为*namedtuple()* 。

在 3.9 版本发生变更: *processor* 将延后而不是立即被获取。

16.7.2 Java 平台

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)

Jython 的版本接口

返回一个元组 `(release, vendor, vminfo, osinfo)`，其中 *vminfo* 为元组 `(vm_name, vm_release, vm_vendor)` 而 *osinfo* 为元组 `(os_name, os_version, os_arch)`。无法确定的值将设为由形参所给出的默认值 (默认均为 `''`)。

Deprecated since version 3.13, will be removed in version 3.15: 它基本上未经测试，具有令人迷惑的 API，并且仅适用于 Jython 支持。

16.7.3 Windows 平台

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)

从 Windows 注册表获取额外的版本信息并返回一个元组 `(release, version, csd, ptype)` 表示 OS 发行版, 版本号, CSD 级别 (Service Pack) 和 OS 类型 (多个/单个处理器)。无法确定的值被设置为作为参数给出的默认值（这些参数都默认为一个空字符串）。

一点提示: *ptype* 在单处理器的 NT 机器上为 `'Uniprocessor Free'` 而在多处理器的机器上则为 `'Multiprocessor Free'`。`'Free'` 是指该 OS 版本不包含调试代码。它还可能包含 `'Checked'` 表示该 OS 版本使用了调试代码，即检测参数、范围等的代码。

platform.**win32_edition**()

返回一个代表当前 Windows 版本的字符串，或者在该值无法确定时返回 `None` 。可能的值包括但不限于 `'Enterprise'`, `'IoTUAP'`, `'ServerStandard'` 和 `'nanoserver'`。

Added in version 3.8.

platform.**win32_is_iot**()

如果*win32_edition()* 返回的 Windows 版本被识别为 IoT 版则返回 `True`。

Added in version 3.8.

16.7.4 macOS 平台

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)

获取 macOS 版本信息并将其返回为元组 `(release, versioninfo, machine)`，其中 *versioninfo* 是一个元组 `(version, dev_stage, non_release_version)`。

无法确定的条目会被设为 `''`。所有元组条目均为字符串。

16.7.5 iOS 平台

platform.**ios_ver**(*system=''*, *release=''*, *model=''*, *is_simulator=False*)

获取 iOS 版本信息并将其以具有下列属性的*namedtuple()* 形式返回：

- `system` 为 OS 名称；为 `'iOS'` 或 `'iPadOS'`。
- `release` 为字符串形式的 iOS 版本号 (例如 `'17.2'`)。
- `model` 为设备型号标识符；这对物理设备来说将是如 `'iPhone13,2'` 形式的字符串，或者对于模拟器来说则为 `'iPhone'`。
- `is_simulator` 是一个描述 app 是运行在模拟器上还是物理设备上的布尔值。

无法确定的条目会被设为由形参所给出的默认值。

16.7.6 Unix 平台

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)

尝试确定可执行文件（默认为 Python 解释器）所链接到的 libc 版本。返回一个字符串元组 `(lib, version)`，当查找失败时其默认值将设为给定的形参值。

请注意此函数对于不同 libc 版本向可执行文件添加符号的方式有深层的关联，可能仅适用于使用 **gcc** 编译出来的可执行文件。

文件将按 *chunksize* 个字节的分块来读取和扫描。

16.7.7 Linux 平台

platform.**freedesktop_os_release**()

从 os-release 文件获取操作系统标识并将其作为一个字典返回。os-release 文件是 freedesktop.org 标准 并在大多数 Linux 发行版上可用。一个重要的例外是 Android 和基于 Android 的发行版。

当 /etc/os-release 或 /usr/lib/os-release 均无法读取时将引发*OSError* 或其子类。

成功时，该函数将返回一个字典，其中键和值均为字符串。值当中的特殊字符例如 " 和 {TX-PL-LABEL}#x60; 会被复原。字段 NAME, ID 和 PRETTY_NAME 总是会按照标准来定义。所有其他字段都是可选的。厂商可能会包括额外的字段。

请注意 NAME, VERSION 和 VARIANT 等字段是适用于向用户展示的字符串。程序应当使用 ID, ID_LIKE, VERSION_ID 或 VARIANT_ID 等字段来标识 Linux 发行版。

示例:

```
def get_like_distro():
    info = platform.freedesktop_os_release()
    ids = [info["ID"]]
    if "ID_LIKE" in info:
        # ids 以空格分隔并按优先级排序
        ids.extend(info["ID_LIKE"].split())
    return ids
```

Added in version 3.10.

16.7.8 Android 平台

platform.**android_ver**(*release=''*, *api_level=0*, *manufacturer=''*, *model=''*, *device=''*, *is_emulator=False*)

获取 Android 设备信息。返回一个具有下列属性的*namedtuple()*。无法确定的值会被设为由形参给出的默认值。

- release - 为字符串形式的 Android 版本 (例如 "14")。
- api_level - 正在运行的设备的 API 级别，为整数形式 (例如 34 对应 Android 14)。要获取构建 Python 所使用的 API 级别，参见*sys.getandroidapilevel()*。
- manufacturer - 厂商名称。
- model - 型号名称 –通常为商业名称或型号数字。
- device - 设备名称 –通常为型号数字或代号名。
- is_emulator - 如果设备为模拟器则为 True，如果为物理设备则为 False。

Google 维护了一个 已知型号和设备名称列表。

Added in version 3.13.

16.8 errno --- 标准 errno 系统符号

该模块提供了标准的 errno 系统符号。每个符号的值都是相应的整数值。名称和描述借用自 linux/include/errno.h，它应该是全包含的。

errno.**errorcode**

提供从 errno 值到底层系统中字符串名称的映射的字典。例如，errno.errorcode[errno.EPERM] 映射为 'EPERM'。

如果要将数字的错误代码转换为错误信息，请使用*os.strerror()*。

在下面的列表中，当前平台上没有使用的符号没有被本模块定义。已定义的符号的具体列表可参见 errno.errorcode.keys()。可用的符号包括:

errno.**EPERM**
操作不允许。这个错误被映射到异常*PermissionError* 。

errno.**ENOENT**
没有这样的文件或目录。这个错误被映射到异常*FileNotFoundError* 。

errno.**ESRCH**
没有这样的进程。这个错误被映射到异常*ProcessLookupError* 。

errno.**EINTR**
系统调用中断。这个错误被映射到异常*InterruptedError* 。

errno.**EIO**
I/O 错误

errno.**ENXIO**
无此设备或地址

errno.**E2BIG**
参数列表过长

errno.**ENOEXEC**
执行格式错误

errno.**EBADF**
错误的文件号

errno.**ECHILD**
没有子进程。这个错误被映射到异常*ChildProcessError* 。

errno.**EAGAIN**
再试一次。这个错误被映射到异常*BlockingIOError* 。

errno.**ENOMEM**
内存不足

errno.**EACCES**
权限被拒绝。这个错误被映射到异常*PermissionError* 。

errno.**EFAULT**
错误的地址

errno.**ENOTBLK**
需要块设备

errno.**EBUSY**
设备或资源忙

errno.**EEXIST**
文件存在。这个错误被映射到异常*FileExistsError* 。

errno.**EXDEV**
跨设备链接

errno.**ENODEV**
无此设备

errno.**ENOTDIR**
不是一个目录。这个错误被映射到异常*NotADirectoryError* 。

errno.**EISDIR**
是一个目录。这个错误被映射到异常*IsADirectoryError* 。

errno.**EINVAL**
无效的参数

errno.**ENFILE**
文件表溢出

errno.**EMFILE**
打开的文件过多

errno.**ENOTTY**
不是打字机

errno.**ETXTBSY**
文本文件忙

errno.**EFBIG**
文件过大

errno.**ENOSPC**
设备已无可用空间

errno.**ESPIPE**
非法查找

errno.**EROFS**
只读文件系统

errno.**EMLINK**
链接过多

errno.**EPIPE**
管道中断。这个错误被映射到异常*BrokenPipeError* 。

errno.**EDOM**
数学参数超出函数范围

errno.**ERANGE**
数学运算结果无法表示

errno.**EDEADLK**
将发生资源死锁

errno.**ENAMETOOLONG**
文件名过长

errno.**ENOLCK**
没有可用的记录锁

errno.**ENOSYS**
功能未实现

errno.**ENOTEMPTY**
目录非空

errno.**ELOOP**
遇到过多的符号链接

errno.**EWOULDBLOCK**
操作会阻塞。这个错误被映射到异常*BlockingIOError* 。

errno.**ENOMSG**
没有所需类型的消息

errno.**EIDRM**
标识符被移除

errno.**ECHRNG**
信道编号超出范围

errno.**EL2NSYNC**
级别 2 未同步

errno.**EL3HLT**
级别 3 已停止

errno.**EL3RST**
级别 3 重置

errno.**ELNRNG**
链接编号超出范围

errno.**EUNATCH**
未附加协议驱动

errno.**ENOCSI**
没有可用的 CSI 结构

errno.**EL2HLT**
级别 2 已停止

errno.**EBADE**
无效的交换

errno.**EBADR**
无效的请求描述符

errno.**EXFULL**
交换已满

errno.**ENOANO**
没有阳极

errno.**EBADRQC**
无效的请求码 ·

errno.**EBADSLT**
无效的槽位

errno.**EDEADLOCK**
文件锁定死锁错误

errno.**EBFONT**
错误的字体文件格式

errno.**ENOSTR**
设备不是流

errno.**ENODATA**
没有可用的数据

errno.**ETIME**
计时器已到期

errno.**ENOSR**
流资源不足

errno.**ENONET**
机器不在网络上

errno.**ENOPKG**
包未安装

errno.**EREMOTE**
对象是远程的

errno.**ENOLINK**
链接已被切断

errno.**EADV**
广告错误

errno.**ESRMNT**
挂载错误

errno.**ECOMM**
发送时通讯错误

errno.**EPROTO**
协议错误

errno.**EMULTIHOP**
已尝试多跳

errno.**EDOTDOT**
RFS 专属错误

errno.**EBADMSG**
非数据消息

errno.**EOVERFLOW**
值相对于已定义数据类型过大

errno.**ENOTUNIQ**
名称在网络上不唯一

errno.**EBADFD**
文件描述符处于错误状态

errno.**EREMCHG**
远端地址已改变

errno.**ELIBACC**
无法访问所需的共享库

errno.**ELIBBAD**
访问已损坏的共享库

errno.**ELIBSCN**
a.out 中的.lib 部分已损坏

errno.**ELIBMAX**
尝试链接过多的共享库

errno.**ELIBEXEC**
无法直接执行共享库

errno.**EILSEQ**
非法字节序列

errno.**ERESTART**
已中断系统调用需要重启

errno.**ESTRPIPE**
流管道错误

errno.**EUSERS**
用户过多

errno.**ENOTSOCK**
在非套接字上执行套接字操作

errno.**EDESTADDRREQ**
需要目标地址

errno.**EMSGSIZE**
消息过长

errno.**EPROTOTYPE**
套接字的协议类型错误

errno.**ENOPROTOOPT**
协议不可用

errno.**EPROTONOSUPPORT**
协议不受支持

errno.**ESOCKTNOSUPPORT**
套接字类型不受支持

errno.**EOPNOTSUPP**
操作在传输端点上不受支持

errno.**ENOTSUP**
操作不受支持

Added in version 3.2.

errno.**EPFNOSUPPORT**
协议族不受支持

errno.**EAFNOSUPPORT**
地址族不受协议支持

errno.**EADDRINUSE**
地址已被使用

errno.**EADDRNOTAVAIL**
无法分配要求的地址

errno.**ENETDOWN**
网络已断开

errno.**ENETUNREACH**
网络不可达

errno.**ENETRESET**
网络因重置而断开连接

errno.**ECONNABORTED**
软件导致连接中止。这个错误被映射到异常*ConnectionAbortedError* 。

errno.**ECONNRESET**
连接被对方重置。这个错误被映射到异常*ConnectionResetError* 。

errno.**ENOBUFS**
没有可用的缓冲区空间

errno.**EISCONN**
传输端点已连接

errno.**ENOTCONN**
传输端点未连接

errno.**ESHUTDOWN**
在传输端点关闭后无法发送。这个错误被映射到异常*BrokenPipeError* 。

errno.**ETOOMANYREFS**
引用过多：无法拼接

errno.**ETIMEDOUT**
连接超时。这个错误被映射到异常*TimeoutError* 。

errno.**ECONNREFUSED**
连接被拒绝。这个错误被映射到异常*ConnectionRefusedError* 。

errno.**EHOSTDOWN**
主机已关闭

errno.**EHOSTUNREACH**
没有到主机的路由

errno.**EALREADY**
操作已经在进行中。这个错误被映射到异常*BlockingIOError* 。

errno.**EINPROGRESS**
操作现在正在进行中。这个错误被映射到异常*BlockingIOError* 。

errno.**ESTALE**
过期的 NFS 文件句柄

errno.**EUCLEAN**
结构需要清理

errno.**ENOTNAM**
不是 XENIX 命名类型文件

errno.**ENAVAIL**
没有可用的 XENIX 信标

errno.**EISNAM**
是命名类型文件

errno.**EREMOTEIO**
远程 I/O 错误

errno.**EDQUOT**
超出配额

errno.**EQFULL**
接口输出队列已满

Added in version 3.11.

errno.**ENOMEDIUM**
未找到媒介

errno.**EMEDIUMTYPE**
错误的媒介类型

errno.**ENOKEY**
需要的密钥不可用

errno.**EKEYEXPIRED**
密钥已到期

errno.**EKEYREVOKED**
密钥已被撤销

errno.**EKEYREJECTED**
密钥被服务拒绝

errno.**ERFKILL**
操作因 RF-kill 而无法执行

errno.**ELOCKUNMAPPED**
锁定的锁未被映射

errno.**ENOTACTIVE**
功能未被激活

errno.**EAUTH**
认证错误

Added in version 3.2.

errno.**EBADARCH**
可执行文件中有错误的 CPU 类型

Added in version 3.2.

errno.**EBADEXEC**
错误的可执行文件（或共享库）

Added in version 3.2.

errno.**EBADMACHO**
畸形的 Mach-o 文件

Added in version 3.2.

errno.**EDEVERR**
设备错误

Added in version 3.2.

errno.**EFTYPE**
不正确的文件类型或格式

Added in version 3.2.

errno.**ENEEDAUTH**
需要认证

Added in version 3.2.

errno.**ENOATTR**
属性未找到

Added in version 3.2.

errno.**ENOPOLICY**
策略未找到

Added in version 3.2.

errno.**EPROCLIM**
进程过多

Added in version 3.2.

errno.**EPROCUNAVAIL**
错误的程序步骤

Added in version 3.2.

errno.**EPROGMISMATCH**
程序版本错误

Added in version 3.2.

errno.**EPROGUNAVAIL**
RPC 程序不可用

Added in version 3.2.

errno.**EPWROFF**
设备电源关闭

Added in version 3.2.

errno.**EBADRPC**
RPC 结构错误

Added in version 3.2.

errno.**ERPCMISMATCH**
RPC 版本错误

Added in version 3.2.

errno.**ESHLIBVERS**
共享库版本不匹配

Added in version 3.2.

errno.**ENOTCAPABLE**
功能不足。此错误被映射到异常*PermissionError*。

Availability: WASI, FreeBSD

Added in version 3.11.1.

errno.**ECANCELED**
操作已被取消

Added in version 3.2.

errno.**EOWNERDEAD**
所有者已不存在

Added in version 3.2.

errno.**ENOTRECOVERABLE**
状态无法恢复

Added in version 3.2.

16.9 ctypes --- Python 的外部函数库

源代码: Lib/ctypes

---

ctypes 是 Python 的外部函数库。它提供了与 C 兼容的数据类型，并允许调用 DLL 或共享库中的函数。可使用该模块以纯 Python 形式对这些库进行封装。

16.9.1 ctypes 教程

注：本教程中的示例代码使用*doctest* 来保证它们能正确运行。由于有些代码示例在 Linux, Windows 或 macOS 上的行为有所不同，它们在注释中包含了一些 doctest 指令。

注意：部分示例代码引用了 ctypes *c_int* 类型。在 `sizeof(long) == sizeof(int)` 的平台上此类型是*c_long* 的一个别名。所以，在程序输出*c_long* 而不是你期望的*c_int* 时不必感到迷惑 --- 它们实际上是同一种类型。

载入动态连接库

ctypes 导出了 *cdll* 对象，在 Windows 系统中还导出了 *windll* 和 *oledll* 对象用于载入动态连接库。

您可以通过访问这些对象的属性来加载库。*cdll* 加载使用标准 `cdecl` 调用约定导出函数的库，而 *windll* 库则使用 `stdcall` 调用约定调用函数。*oledll* 也使用 `stdcall` 调用约定，并假定函数返回 Windows `HRESULT` 错误代码。当函数调用失败时会使用错误代码自动引发*OSError* 异常。

在 3.3 版本发生变更: 原来在 Windows 下抛出的异常类型*WindowsError* 现在是*OSError* 的一个别名。

这是一些 Windows 下的例子。请注意 `msvcrt` 是包含大部分 C 函数的 MS 标准 C 库，并会使用 `cdecl` 调用惯例:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows 会自动添加通常的 `.dll` 文件扩展名。

> **备注**
>
> 通过 `cdll.msvcrt` 调用的标准 C 函数，可能会导致调用一个过时的，与当前 Python 所不兼容的函数。因此，请尽量使用标准的 Python 函数，而不要使用 `msvcrt` 模块。

在 Linux 中，要求指定文件名 包括扩展名来加载库，因此不能使用属性访问的方式来加载库。你应当使用 dll 加载器的*LoadLibrary()* 方法，或是应当通过调用构造器创建 CDLL 的实例来加载库:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

操作导入的动态链接库中的函数

通过操作 dll 对象的属性来操作这些函数。

```
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

请注意 win32 系统的动态库如 `kernel32` 和 `user32` 通常会同时导出一个函数的 ANSI 版本和 UNICODE 版本。UNICODE 版本导出时会在名称后加上 `W`，而 ANSI 版本导出时会在名称后加上 `A`。win32 `GetModuleHandle` 函数会为给定的模块名称返回一个 *模块句柄*，它具有以下的 C 原型，以及一个被用来根据是否定义了 UNICODE 将其中之一暴露为 `GetModuleHandle` 的宏:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll 不会通过这样的魔法手段来帮你决定选择哪一种函数，你必须显式的调用 `GetModuleHandleA` 或 `GetModuleHandleW`，并分别使用字节对象或字符串对象作参数。

有时候，dlls 的导出的函数名不符合 Python 的标识符规范，比如 `"??2@YAPAXI@Z"`。此时，你必须使用*getattr()* 方法来获得该函数。

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Windows 下，有些 dll 导出的函数没有函数名，而是通过其顺序号调用。对此类函数，你也可以通过 dll 对象的数值索引来操作这些函数。

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

调用函数

你可以像任何其它 Python 可调用对象一样调用这些函数。这个例子使用了 `rand()` 函数，它不接收任何参数并返回一个伪随机整数:

```
>>> print(libc.rand())
1804289383
```

在 Windows 上，你可以调用 `GetModuleHandleA()` 函数，它返回一个 win32 模块句柄 (将 `None` 作为唯一参数传入以使用 `NULL` 指针来调用它):

```
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

如果你用 `cdecl` 调用方式调用 `stdcall` 约定的函数，则会甩出一个异常*ValueError*。反之亦然。

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

你必须阅读这些库的头文件或说明文档来确定它们的正确的调用协议。

在 Windows 中，*ctypes* 使用 win32 结构化异常处理来防止由于在调用函数时使用非法参数导致的程序崩溃。

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

然而，总有许多办法，通过调用*ctypes* 使得 Python 程序崩溃。因此，你必须小心使用。*faulthandler* 模块可以用于帮助诊断程序崩溃的原因。（比如由于错误的 C 库函数调用导致的段错误）。

None、整数、字节串对象和（Unicode）字符串是仅有的可以直接作为这些函数调用的形参的原生 Python 对象。None 将作为 C NULL 指针传入，字节串对象和字符串将作为指向包含其数据 (char* 或 wchar_t*) 的内存块的指针传入。Python 整数将作为平台默认的 C int 类型传入，它们的值会被截断以适应 C 类型的长度。

在我们开始调用函数前，我们必须先了解作为函数参数的*ctypes* 数据类型。

基础数据类型

ctypes 定义了一些和 C 兼容的基本数据类型：

ctypes 类型	C 类型	Python 类型
c_bool	_Bool	bool (1)
c_char	char	单字符字节串对象
c_wchar	wchar_t	单字符字符串
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 或 long long	int
c_ulonglong	unsigned __int64 或 unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t 或 Py_ssize_t	int
c_time_t	time_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char* (以 NUL 结尾)	字节串对象或 None
c_wchar_p	wchar_t* (以 NUL 结尾)	字符串或 None
c_void_p	void*	int 或 None

(1) 构造函数接受任何具有真值的对象。

所有这些类型都可以通过使用正确类型和值的可选初始值调用它们来创建:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

由于这些类型是可变的，它们的值也可以在以后更改:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

给指针类型的实例 Assigning a new value to instances of the pointer types *c_char_p*, *c_wchar_p* 和*c_void_p* 赋新值会改变它们所指向的 内存位置，而不是内存块的 内容 (当然不是，因为 Python 字符串对象是不可变的):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # 内存分配已改变
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # 第一个对象未改变
Hello, World
>>>
```

但你要注意不能将它们传递给会改变指针所指内存的函数。如果你需要可改变的内存块，ctypes 提供了*create_string_buffer()* 函数，它提供多种方式创建这种内存块。当前的内存块内容可以通过 `raw` 属性存取，如果你希望将它作为 NUL 结束的字符串，请使用 `value` 属性:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # 创建一个 3 字节的缓冲区，初始化为 NUL 字节
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # 创建一个包含以 NUL 结束的字符串的缓冲区
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # 创建一个 10 字节的缓冲区
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

create_string_buffer() 函数取代了旧了 c_buffer() 函数（后者仍可作为别名使用）。要创建一个包含 C 类型 wchar_t 的 unicode 字符的可变内存块，请使用*create_unicode_buffer()* 函数。

调用函数，继续

注意 printf 将打印到真正标准输出设备，而 * 不是 * *sys.stdout*，因此这些实例只能在控制台提示符下工作，而不能在 *IDLE* 或 *PythonWin* 中运行。

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: Don't know how to convert parameter 2
>>>
```

正如前面所提到过的，除了整数、字符串以及字节串之外，所有的 Python 类型都必须使用它们对应的*ctypes* 类型包装，才能够被正确地转换为所需的 C 语言类型。

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

调用可变函数

在许多平台上通过 ctypes 调用可变函数与调用带有固定数量形参的函数是完全一样的。在某些平台，特别是针对 Apple 平台的 ARM64 上，可变函数的调用约定与常规函数则是不同的。

在这些平台上要求为常规、非可变函数参数指定*argtypes* 属性：

```
libc.printf.argtypes = [ctypes.c_char_p]
```

因为指定该属性不会影响可移植性所以建议总是为所有可变函数指定*argtypes*。

使用自定义的数据类型调用函数

您也可以通过自定义*ctypes* 参数转换方式来允许将你自己的类实例作为函数参数。*ctypes* 会寻找 _as_parameter_ 属性并使用它作为函数参数。属性必须是整数、字符串、字节串、*ctypes* 实例或者带有 _as_parameter_ 属性的对象：

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

如果你不想将实例数据存储在 _as_parameter_ 实例变量中，可以定义一个根据请求提供属性的*property*。

指定必选参数的类型 (函数原型)

可以通过设置*argtypes* 属性来指定从 DLL 导出函数的必选参数类型。

argtypes 必须是一个 C 数据类型的序列（这里 `printf()` 函数可能不是一个好例子，因为它会根据格式字符串的不同接受可变数量和不同类型的形参，但另一方面这对尝试此功能来说也很方便）：

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

指定数据类型可以防止不合理的参数传递（就像 C 函数的原型），并且会自动尝试将参数转换为需要的类型:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: 'int' object cannot be interpreted as ctypes.c_
↪char_p
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

如果你定义了自己的类并将其传递给函数调用，则你必须为它们实现*from_param()* 类方法才能够在*argtypes* 序列中使用它们。*from_param()* 类方法接受传递给函数调用的 Python 对象，它应该进行类型检查或者其他必要的操作以确保这个对象是可接受的，然后返回对象本身、它的 `_as_parameter_` 属性，或在此情况下作为 C 函数参数传入的任何东西。同样，结果应该是整数、字符串、字节串、*ctypes* 实例或是具有 `_as_parameter_` 属性的对象。

返回类型

在默认情况下都会假定函数返回 C `int` 类型。其他返回类型可通过设置函数对象的 setting the *restype* 属性来指定。

`time()` 的 C 原型是 `time_t time(time_t *)`。由于 `time_t` 的类型可能不同于默认返回类型 `int`，你应当指定 `restype` 属性:

```
>>> libc.time.restype = c_time_t
```

参数类型可以使用*argtypes* 来指定:

```
>>> libc.time.argtypes = (POINTER(c_time_t),)
```

调用该函数时如果要将 `NULL` 指针作为第一个参数，请使用 `None`:

```
>>> print(libc.time(None))
1150640792
```

下面是一个更高级的示例，它使用了 `strchr()` 函数，该函数接收一个字符串指针和一个字符，并返回一个字符串指针:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

如果你想要避免上面的*ord("x")* 调用，你可以设置*argtypes* 属性，第二个参数将从单字符 Python 字节串对象转换为 C char：

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
b'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
ctypes.ArgumentError: argument 2: TypeError: one character bytes, bytearray or integer␣
↪expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
b'def'
>>>
```

如果外部函数返回一个整数，你也可以使用一个 Python 可调用对象（例如函数或类）作为*restype* 属性。该可调用对象被调用时将附带 C 函数返回的 整数，其调用结果将被用作函数调用的结果值。这对于检查错误返回值并自动引发异常来说很有用处：

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` 函数可以调用 Windows 的 `FormatMessage()` API 获取错误码的字符串说明，然后 返回一个异常。`WinError` 接收一个可选的错误码作为参数，如果没有的话，它将调用*GetLastError()* 获取错误码。

请注意通过*errcheck* 属性可提供更强大的错误检查机制；详情见参考手册。

传递指针（或以引用方式传递形参）

有时候 C 函数接口可能由于要往某个地址写入值，或者数据太大不适合作为值传递，从而希望接收一个指针作为数据参数类型。这和 传递参数引用类似。

ctypes 暴露了*byref()* 函数用于通过引用传递参数，使用*pointer()* 函数也能达到同样的效果，只不过*pointer()* 需要更多步骤，因为它要先构造一个真实指针对象。所以在 Python 代码本身不需要使用这个指针对象的情况下，使用*byref()* 效率更高。

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

结构体和联合

结构体和联合必须派生自*Structure*和*Union*基类，这两个基类是在*ctypes*模块中定义的。每个子类都必须定义*_fields_*属性。_fields_必须是一个 *2* 元组的列表，其中包含一个 *字段名称*和一个 *字段类型*。

type 字段必须是一个*ctypes*类型，比如*c_int*，或者其他*ctypes*类型: 结构体、联合、数组、指针。

这是一个简单的 POINT 结构体，它包含名称为 *x* 和 *y* 的两个变量，还展示了如何通过构造函数初始化结构体。

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

当然，你可以构造更复杂的结构体。一个结构体可以通过设置 type 字段包含其他结构体或者自身。

这是以一个 RECT 结构体，他包含了两个 POINT ，分别叫 *upperleft* 和 *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

嵌套结构体可以通过几种方式构造初始化:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

可以通过 类获取字段*descriptor* ，它能提供很多有用的调试信息。

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

警告

ctypes 不支持带位域的结构体、联合以值的方式传给函数。这可能在 32 位 x86 平台上可以正常工作，但是对于一般情况，这种行为是未定义的。带位域的结构体、联合应该总是通过指针传递给函数。

结构体/联合字段对齐及字节顺序

在默认情况下，Structure 和 Union 字段使用与 C 编译器一样的方式进行对齐。可以通过在子类定义中指定 *_pack_* 类属性来覆盖此行为。该属性必须被设为一个正整数来指明字段的最大对齐值。这也是 `#pragma pack(n)` 在 MSVC 中所做的事情。还可以使用与. It is also possible to set a minimum alignment for how the subclass itself is packed in the same way `#pragma align(n)` 在 MSVC 中一样的方式来设置子类本身数据打包的最小对齐值。这可以通过在子类定义中指定 *_align_* 类属性来实现。

ctypes 中的结构体和联合使用的是本地字节序。要使用非本地字节序，可以使用 *BigEndianStructure*, *LittleEndianStructure*, *BigEndianUnion*, and *LittleEndianUnion* 作为基类。这些类不能包含指针字段。

结构体和联合中的位域

可以创建包含位字段的结构体和联合。位字段只适用于整数字段，位宽度是由 *_fields_* 元组中的第三项来指定的:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

数组

数组是一个序列，包含指定个数元素，且必须类型相同。

创建数组类型的推荐方式是使用一个类型乘以一个正数:

```
TenPointsArrayType = POINT * 10
```

下面是一个构造的数据案例，结构体中包含了 4 个 POINT 和一些其他东西。

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

和平常一样，通过调用它创建实例:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

以上代码会打印几行 `0 0`，因为数组内容被初始化为 0.

也能通过指定正确类型的数据来初始化:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
```

(续下页)

(接上页)

```
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

指针

可以将*ctypes*类型数据传入*pointer()*函数创建指针:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

指针实例拥有*contents*属性，它返回指针指向的真实对象，如上面的`i`对象:

```
>>> pi.contents
c_long(42)
>>>
```

注意*ctypes*并没有 OOR （返回原始对象），每次访问这个属性时都会构造返回一个新的相同对象:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

将这个指针的 contents 属性赋值为另一个*c_int*实例将会导致该指针指向该实例的内存地址:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

指针对象也可以通过整数下标进行访问:

```
>>> pi[0]
99
>>>
```

通过整数下标赋值可以改变指针所指向的真实内容:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

使用 0 以外的索引也是合法的，但是你必须确保知道自己为什么这么做，就像 C 语言中: 你可以访问或者修改任意内存内容。通常只会在函数接收指针是才会使用这种特性，而且你*知道*这个指针指向的是一个数组而不是单个值。

内部细节，*pointer()*函数不只是创建了一个指针实例，它首先创建了一个指针*类型*。这是通过调用*POINTER()*函数实现的，它接收*ctypes*类型为参数，返回一个新的类型:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

无参调用指针类型可以创建一个 NULL 指针。NULL 指针的布尔值是 False

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

解引用指针的时候，*ctypes* 会帮你检测是否指针为 NULL (但是解引用无效的非 NULL 指针仍会导致 Python 崩溃):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

类型转换

通常，ctypes 会进行严格的类型检查。这意味着，如果在某个函数的*argtypes* 列表中有 POINTER(c_int) 或在结构体定义中将其用作成员字段的类型，则只接受完全相同类型的实例。此规则也有一些例外情况，在这些情况下 ctypes 可以接受其他对象。例如，你可以传入兼容的数组实例而不是指针类型。因此，对于 POINTER(c_int)，ctypes 接受一个 c_int 数组:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

此外，如果一个函数参数在*argtypes* 中显式地声明为指针类型 (如 POINTER(c_int))，则可以向该函数传递所指向的类型的对象 (在本例中为 c_int)。在这种情况下，ctypes 将自动应用所需的*byref()* 转换。conversion in this case automatically.

可以给指针内容赋值为 None 将其设置为 Null

```
>>> bar.values = None
>>>
```

有时候你拥有一个不兼容的类型。在 C 中，你可以将一个类型强制转换为另一个。*ctypes* 中的 a *cast()* 函数提供了相同的功能。上面的结构体 `Bar` 的 `value` 字段接收 `POINTER(c_int)` 指针或者*c_int* 数组，但是不能接受其他类型的实例:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

这种情况下，需要手动使用*cast()* 函数。

cast() 函数可以将一个指针实例强制转换为另一种 ctypes 类型。*cast()* 接收两个参数，一个 ctypes 指针对象或者可以被转换为指针的其他类型对象，和一个 ctypes 指针类型。返回第二个类型的一个实例，该返回实例和第一个参数指向同一片内存空间:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

所以*cast()* 可以用来给结构体 `Bar` 的 `values` 字段赋值:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

不完整类型

不完整类型即还没有定义成员的结构体、联合或者数组。在 C 中，它们通常用于前置声明，然后在后面定义:

```
struct cell; /* 前向声明 */

struct cell {
    char *name;
    struct cell *next;
};
```

直接翻译成 ctypes 的代码如下，但是这行不通:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

因为新的 `class cell` 在 class 语句本身中是不可用的。在*ctypes* 中，我们可以定义 `cell` 类再在 class 语句之后设置*_fields_* 属性:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
```

(续下页)

(接上页)

```
...                 ("next", POINTER(cell))]
>>>
```

让我们试试。我们定义两个 cell 实例，让它们互相指向对方，然后通过指针链式访问几次:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

回调函数

ctypes 允许创建一个指向 Python 可调用对象的 C 函数。它们有时候被称为 *回调函数*。

首先，你必须为回调函数创建一个类，这个类知道调用约定，包括返回值类型以及函数接收的参数类型及个数。

CFUNCTYPE() 工厂函数使用 cdecl 调用约定创建回调函数类型。在 Windows 上，*WINFUNCTYPE()* 工厂函数使用 stdcall 调用约定为回调函数创建类型。

这些工厂函数的第一个参数是返回值类型，回调函数的参数类型作为剩余参数。

这里展示一个使用标准 C 库的 qsort() 函数例子，使用它在一个回调函数的协助下对条目进行排序。qsort() 将被用来给一个整数的数组排序:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

qsort() 被调用时必须传入一个指向要排序的数据的指针、数据数组中的条目数、每条目的大小以及一个指向比较函数即回调函数的指针。回调函数将附带两个指向条目的指针进行调用，如果第一个条目小于第二个条目则它必须返回一个负整数，如果两者相等则返回零，在其他情况下则返回一个正整数。

所以，我们的回调函数要接收两个整数指针，返回一个整数。首先我们创建回调函数的 类型

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

首先，这是一个简单的回调，它会显示传入的值:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

结果:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
```

(续下页)

(接上页)

```
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

现在我们可以比较两个元素并返回有用的结果了:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

我们可以轻易地验证，现在数组是有序的了:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

这些工厂函数可以当作装饰器工厂，所以可以这样写:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

备注

请确保你维持的*CFUNCTYPE()* 对象的引用周期与它们在 C 代码中的使用期一样长。*ctypes* 不会确保这一点，如果不这样做，它们可能会被垃圾回收，导致程序在执行回调函数时发生崩溃。

注意，如果回调函数在 Python 之外的另外一个线程使用 (比如，外部代码调用这个回调函数)，ctypes 会在每一次调用上创建一个虚拟 Python 线程。这个行为在大多数情况下是合理的，但也意味着如果有数据使用*threading.local* 方式存储，将无法访问，就算它们是在同一个 C 线程中调用的。

访问 dll 的导出变量

某些共享库不仅会导出函数，还会导出变量。一个例子就是 Python 库本身的 `Py_Version`，Python 运行时版本号被编码为单个整数常量。

ctypes 可以通过类型的*in_dll()* 类方法访问这样的值。*pythonapi* 是一个用于访问 Python C api 预定义符号:

```
>>> version = ctypes.c_int.in_dll(ctypes.pythonapi, "Py_Version")
>>> print(hex(version.value))
0x30c00a0
```

一个扩展例子, 同时也展示了使用指针访问 Python 导出的 `PyImport_FrozenModules` 指针对象。

对文档中这个值的解释说明

> 该指针被初始化为指向一个 `_frozen` 记录的数组，以一个所有成员均为 `NULL` 或零的记录表示结束。当一个冻结模块被导入时，它将在此表中被搜索。第三方代码可以利用此方式来提供动态创建的冻结模块集。

这足以证明修改这个指针是很有用的。为了让实例大小不至于太长，这里只展示如何使用*ctypes* 读取这个表:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int),
...                 ("get_code", POINTER(c_ubyte)),  # 函数指针
...                ]
...
>>>
```

我们定义了 `_frozen` 数据类型，所以我们可以获取表的指针:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "_PyImport_FrozenBootstrap")
>>>
```

由于 `table` 是指向 `struct_frozen` 数组的 指针，我们可以遍历它，只不过需要自己判断循环是否结束，因为指针本身并不包含长度。它早晚会因为访问到野指针或者什么的把自己搞崩溃，所以我们最好在遇到 `NULL` 后就让它退出循环:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
zipimport 12345
>>>
```

Python 的冻结模块和冻结包 (由负 `size` 成员表示) 并不是广为人知的事情，它们仅仅用于实验。例如，可以使用 `import __hello__` 尝试一下这个功能。

意外

ctypes 也有自己的边界，有时候会发生一些意想不到的事情。

比如下面的例子:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
```

(续下页)

(接上页)

```
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # 现在交换这两个
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

嗯。我们预想应该打印 `3 4 1 2` 。但是为什么呢? 这是 `rc.a, rc.b = rc.b, rc.a` 这行代码展开后的步骤:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

注意 `temp0` 和 `temp1` 对象始终引用了对象 `rc` 的内容。然后执行 `rc.a = temp0` 会把 `temp0` 的内容拷贝到 `rc` 的空间。这也改变了 `temp1` 的内容。最终导致赋值语句 `rc.b = temp1` 没有产生预想的效果。

记住，访问被包含在结构体、联合、数组中的对象并不会将其 复制出来，而是得到了一个代理对象，它是对根对象的内部内容的一层包装。

下面是另一个可能和预期有偏差的例子:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

备注

使用*c_char_p* 实例化的对象只能将其值设置为 bytes 或者整数。

为什么这里打印了 `False` ？ ctypes 实例是一些内存块加上一些用于访问这些内存块的*descriptor* 组成。将 Python 对象存储在内存块并不会存储对象本身，而是存储了对象的 内容。每次访问对象的内容都会构造一个新的 Python 对象。

变长数据类型

ctypes 对变长数组和结构体提供了一些支持。

The *resize()* function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a *ValueError* is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
```

(续下页)

(接上页)

```
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

这非常好，但是要怎么访问数组中额外的元素呢？因为数组类型已经定义包含 4 个元素，导致我们访问新增元素时会产生以下错误:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

使用 *ctypes* 访问变长数据类型的一个可行方法是利用 Python 的动态特性，根据具体情况，在知道这个数据的大小后，(重新) 指定这个数据的类型。

16.9.2 ctypes 参考手册

寻找动态链接库

在编译型语言中，动态链接库会在编译、链接或者程序运行时访问。

find_library() 函数的目的是以类似于编译器或运行时加载器的方式来定位库（在有多个共享库版本的平台上应当加载最新的版本），而 ctypes 库加载器的行为类似于程序已经运行时直接调用运行时加载器。

`ctypes.util` 模块提供了一个函数，可以帮助确定要加载的库。

`ctypes.util.`**`find_library`**(*name*)

> 尝试寻找一个库然后返回其路径名，*name* 是库名称, 且去除了 *lib* 等前缀和 `.so` 、`.dylib` 、版本号等后缀 (这是 posix 连接器 `-l` 选项使用的格式)。如果没有找到对应的库，则返回 `None` 。

确切的功能取决于系统。

在 Linux 中，*find_library()* 会尝试运行外部程序 (`/sbin/ldconfig`, `gcc`, `objdump` 和 `ld`) 来查找库文件。它会返回库文件的文件名。

在 3.6 版本发生变更: 在 Linux 上，如果其他方式找不到的话，会使用环境变量 `LD_LIBRARY_PATH` 搜索动态链接库。

这是一些例子:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

在 macOS 和 Android 上，*find_library()* 使用系统的标准命名方案和路径来定位库，并在成功时返回完整的路径名:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
```

(续下页)

(接上页)

```
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

在 Windows 中，*find_library()* 会沿着系统搜索路径进行搜索，并返回完整的路径名称，但由于没有预定义的命名方案因此像 `find_library("c")` 这样的调用会失败并返回 `None`。

如果使用*ctypes* 包装一个共享库，则更好的做法 可能是开发时就确定好共享库的名称，并将其硬编码到包装模块中而不是在运行时使用*find_library()* 来定位库。

加载动态链接库

有很多方式可以将动态链接库加载到 Python 进程。其中之一是实例化以下类的其中一个:

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

该类的实例代表已加载的共享库。这些库中的函数使用标准的 C 调用约定，并被预期会返回 `int`。

在 Windows 上创建*CDLL* 实例可能会失败，即使 DLL 名称确实存在。当某个被加载 DLL 所依赖的 DLL 未找到时，将引发*OSError* 错误并附带消息 *"[WinError 126] The specified module could not be found"*. 此错误消息不包含缺失 DLL 的名称，因为 Windows API 并不会返回此类信息，这使得此错误难以诊断。要解决此错误并确定是哪一个 DLL 未找到，你需要找出所依赖的 DLL 列表并使用 Windows 调试与跟踪工具确定是哪一个未找到。

在 3.12 版本发生变更: 现在 *name* 形参可以是一个*path-like object*。

参见

Microsoft DUMPBIN 工具 -- 一个用于查找 DLL 依赖的工具。

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

这个类的实例代表已加载的共享库，这些库中的函数使用 `stdcall` 调用约定，并且预期返回 Windows 专属的*HRESULT* 代码。*HRESULT* 值包含指明函数调用是失败还是成功的信息，并带有额外的错误码。如果返回值是提示失败，则会自动引发*OSError*。

Availability: Windows

在 3.3 版本发生变更: 过去会引发*WindowsError*，现在它是*OSError* 的别名。

在 3.12 版本发生变更: 现在 *name* 形参可以是一个*path-like object*。

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

这个类的实例代表已加载的共享库，这些库中的函数使用 `stdcall` 调用约定，并且默认预期返回 `int`。

Availability: Windows

在 3.12 版本发生变更: 现在 *name* 形参可以是一个*path-like object*。

调用动态库导出的函数之前，Python 会释放*global interpreter lock* ，并在调用后重新获取。

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

这个类实例的行为与*CDLL* 类似，只不过 不会在调用函数的时候释放 GIL 锁，且调用结束后会检查 Python 错误码。如果错误码被设置，会抛出一个 Python 异常。

所以，它只在直接调用 Python C 接口函数的时候有用。

在 3.12 版本发生变更: 现在 *name* 形参可以是一个*path-like object*。

所有这些类均可通过附带至少一个参数即共享库的路径名来调用它们进行实例化。如果你有一个对应已加载共享库的现有句柄，可以将其作为 handle 具名形参传入，否则会使用下层平台的 dlopen() 或 LoadLibrary() 函数将库加载到进程中，并获取对应的句柄。

mode 可以指定库加载方式。详情请参见 *dlopen(3)* 手册页。在 Windows 上，会忽略 *mode*，在 posix 系统上，总是会加上 RTLD_NOW，且无法配置。

当 *use_errno* 形参被设为真值时，将启用以安全方式访问系统 *errno* 错误号的 ctypes 机制。*ctypes* 将维护一份系统 *errno* 变量的线程局部副本；如果你调用设置了 use_errno=True 的外部函数那么 *errno* 将在函数调用之前与 ctypes 私有副本互换，同样的情况也会在函数调用之后立即发生。

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

当 *use_last_error* 形参设为真值时，为 Windows 错误代码也启用与由 *GetLastError()* 和 SetLastError() Windows API 函数管理相同的机制；*ctypes.get_last_error()* 和 *ctypes.set_last_error()* 会被用于请求和更改 Windows 错误代码的 ctypes 私有副本。

winmode 形参用于在 Windows 上指定库的加载方式（因为 *mode* 会被忽略）。它接受任何对 Win32 API LoadLibraryEx 旗标形参来说合法的值。当被省略时，默认使用表示最安全的 DLL 加载的旗标，这将避免 DLL 劫持等问题。传入 DLL 的完整路径是确保正确加载库及其依赖的最安全的方式。

在 3.8 版本发生变更: 增加了 *winmode* 参数。

ctypes.**RTLD_GLOBAL**

用于 *mode* 参数的标识值。在此标识不可用的系统上，它被定义为整数 0。

ctypes.**RTLD_LOCAL**

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

ctypes.**DEFAULT_MODE**

加载动态链接库的默认模式。在 OSX 10.3 上，它是 *RTLD_GLOBAL*，其余系统上是 *RTLD_LOCAL*。

这些类的实例没有共用方法。动态链接库的导出函数可以通过属性或者索引的方式访问。注意，通过属性的方式访问会缓存这个函数，因而每次访问它时返回的都是同一个对象。另一方面，通过索引访问，每次都会返回一个新的对象:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # 在 Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

还有下面这些属性可用，他们的名称以下划线开头，以避免和导出函数重名:

PyDLL.**_handle**

用于访问库的系统句柄。

PyDLL.**_name**

传入构造函数的库名称。

共享库也可以通过使用一个预制对象来加载，这种对象是 *LibraryLoader* 类的实例，具体做法是调用 *LoadLibrary()* 方法，或是将库作为加载器实例的属性来提取。

class ctypes.**LibraryLoader**(*dlltype*)

加载共享库的类。*dlltype* 应当为 *CDLL*, *PyDLL*, *WinDLL* 或 *OleDLL* 类型之一。

__getattr__() 具有特殊的行为：它允许通过一个作为库加载器实例的属性访问共享库来加载它。访问结果会被缓存，因此每次重复的属性访问都会返回相同的库。

LoadLibrary(*name*)

加载一个共享库到进程中并将其返回。此方法总是返回一个新的库实例。

可用的预制库加载器有如下这些:

ctypes.**cdll**

创建*CDLL* 实例。

ctypes.**windll**

创建*WinDLL* 实例。

Availability: Windows

ctypes.**oledll**

创建*OleDLL* 实例。

Availability: Windows

ctypes.**pydll**

创建*PyDLL* 实例。

要直接访问 C Python api，可以使用一个现成的 Python 共享库对象:

ctypes.**pythonapi**

一个将 Python C API 函数作为属性公开出来的*PyDLL* 实例。请注意所有这些函数都应返回 C `int`，当然也并非总是如此，因此您必须分配正确的 `restype` 属性才能使用这些函数。

通过这些对象中的任何一个加载库都将引发一个审计事件 `ctypes.dlopen` 并附带字符串参数 `name`，即用于加载库的名称。

在加载的库上访问一个函数将引发一个审计事件 `ctypes.dlsym` 并附带参数 `library` (库对象) 和 `name` (以字符串或整数表示的符号名称).

在只有库句柄而非对象可用的情况下，访问函数会引发一个审计事件 `ctypes.dlsym/handle` 并附带参数 `handle` (原始库句柄) 和 `name`。

外部函数

正如前一节的说明，外部函数可作为已加载共享库的属性来访问。用此方式创建的函数对象默认接受任意数量的参数，接受任意 ctypes 数据实例作为参数，并且返回库加载器所指定的默认结果类型。

它们是私有局部类 `_FuncPtr` 的实例（未在 `ctypes` 中暴露），该类继承自私有的*_CFuncPtr* 类:

```
>>> import ctypes
>>> lib = ctypes.CDLL(None)
>>> issubclass(lib._FuncPtr, ctypes._CFuncPtr)
True
>>> lib._FuncPtr is ctypes._CFuncPtr
False
```

class ctypes.**_CFuncPtr**

C 可调用外部函数的基类。

外部函数的实例也是兼容 C 的数据类型；它们代表 C 函数指针。

此行为可通过对外部函数对象的特殊属性赋值来自定义。

restype

分配一个 ctypes 类型来指定外部函数的结果类型。使用 `None` 来表示 `void`，即不返回任何结果的函数。

赋值为一个非 ctypes 类型的可调用 Python 对象也是可以的，在这种情况下函数应返回 C `int`，并且该可调用对象将附带此整数被调用，以允许进一步的处理或错误检查。这种用法已被弃用，为了更灵活地进行后续处理或错误检查请使用 ctypes 数据类型作为 `restype` 并将*errcheck* 属性赋值为一个可调用对象。

argtypes

赋值为一个 ctypes 类型的元组来指定函数所接受的参数类型。使用 `stdcall` 调用规范的函数只能附带与此元组长度相同数量的参数进行调用；使用 C 调用规范的函数还可接受额外的未指明参数。

当调用外部函数时，每个实际参数都会被传给*argtypes* 元组中条目的*from_param()* 类方法，该方法允许将实际参数适配为此外部函数所接受的对象。例如，*argtypes* 元组中的*c_char_p* 条目将使用 ctypes 转换规则把作为参数传入的字符串转换为字节串对象。

新特性：现在可以在 argtypes 中放入非 ctypes 类型的条目，但每个条目必须具有*from_param()* 方法用于返回一个可作为参数的值（整数、字符串、ctypes 实例）。这样就允许定义可将将自定义对象适配为函数参数的适配器。

errcheck

将一个 Python 函数或其他可调用对象赋值给此属性。该可调用对象将附带三个及以上的参数被调用。

callable(*result*, *func*, *arguments*)

result 是外部函数返回的结果，由 `restype` 属性指明。

func 是外部函数对象本身，这样就允许重新使用相同的可调用对象来对多个函数进行检查或后续处理。

arguments 是一个包含最初传递给函数调用的形参的元组，这样就允许对所用参数的行为进行特别处理。

此函数所返回的对象将会由外部函数调用返回，但它还可以在外部函数调用失败时检查结果并引发异常。

exception ctypes.**ArgumentError**

此异常会在外部函数无法对某个传入参数执行转换时被引发。

在 Windows 上，当外部函数调用引发一个系统异常时（例如由于访问冲突），它将被捕获并被替换为适当的 Python 异常。此外，还将引发一个审计事件 `ctypes.set_exception` 并附带参数 `code`，以允许审计钩子将原异常替换为它自己的异常。

某些发起外部函数调用的方式可能会引发一个审计事件 `ctypes.call_function` 并附带参数 `function pointer` 和 `arguments`。

函数原型

外部函数也可通过实例化函数原型来创建。函数原型类似于 C 中的函数原型；它们在不定义具体实现的情况下描述了一个函数（返回类型、参数类型、调用约定）。工厂函数必须使用函数所需要的结果类型和参数类型来调用，并可被用作装饰器工厂函数，在此情况下可以通过 `@wrapper` 语法应用于函数。请参阅回调函数 了解有关示例。

ctypes.**CFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

返回的函数原型会创建使用标准 C 调用约定的函数。该函数在调用过程中将释放 GIL。如果 *use_errno* 设为真值，则在调用之前和之后系统*errno* 变量的 ctypes 私有副本会与真正的*errno* 值进行交换；*use_last_error* 会为 Windows 错误码执行同样的操作。

ctypes.**WINFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

返回的函数原型会创建使用 `stdcall` 调用约定的函数。该函数在调用过程中将会释放 GIL。*use_errno* 和 *use_last_error* 具有与上文相同的含义。

Availability: Windows

ctypes.**PYFUNCTYPE**(*restype*, **argtypes*)

返回的函数原型会创建使用 Python 调用约定的函数。该函数在调用过程中将 不会释放 GIL。

这些工厂函数所创建的函数原型可通过不同的方式来实例化，具体取决于调用中的类型与数量:

prototype(*address*)

在指定地址上返回一个外部函数，地址值必须为整数。

prototype(*callable*)

基于 Python *callable* 创建一个 C 可调用函数（回调函数）。

prototype(*func_spec*[, *paramflags*])

返回由一个共享库导出的外部函数。*func_spec* 必须为一个 2 元组 `(name_or_ordinal, library)`。第一项是字符串形式的所导出函数名称，或小整数形式的所导出函数序号。第二项是该共享库实例。

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

返回将调用一个 COM 方法的外部函数。*vtbl_index* 虚拟函数表中的索引。*name* 是 COM 方法的名称。*iid* 是可选的指向接口标识符的指针，它被用于扩展的错误报告。

COM 方法使用特殊的调用约定：除了在 `argtypes` 元组中指定的形参，它们还要求一个指向 COM 接口的指针作为第一个参数。

可选的 *paramflags* 形参会创建相比上述特性具有更多功能的外部函数包装器。

paramflags 必须为一个与*argtypes* 长度相同的元组。.

此元组中的每一项都包含有关形参的更多信息，它必须为包含一个、两个或更多条目的元组。

第一项是包含形参指令旗标组合的整数。

1

指定函数的一个输入形参。

2

输出形参。外部函数会填入一个值。

4

默认为整数零值的输入形参。

可选的第二项是字符串形式的形参名称。如果指定此项，则可以使用该形参名称来调用外部函数。

可选的第三项是该形参的默认值。

下面的例子演示了如何包装 Windows 的 `MessageBoxW` 函数以使其支持默认形参和命名参数。相应的 Windows 头文件的 C 声明是这样的:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

这是使用*ctypes* 的包装:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes"), (1,
↪"flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

现在 `MessageBox` 外部函数可以通过以下方式来调用:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

第二个例子演示了输出形参。这个 win32 `GetWindowRect` 函数通过将指定窗口的维度拷贝至调用者必须提供的 `RECT` 结构体来提取这些值。这是相应的 C 声明:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

这是使用*ctypes*的包装:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

带有输出形参的函数如果输出形参存在单一值则会自动返回该值，或是当输出形参存在多个值时返回包含这些值的元组，因此当 GetWindowRect 被调用时现在将返回一个 RECT 实例。

输出形参可以与*errcheck*协议相结合以执行进一步的输出处理和错误检查。Win32 `GetWindowRect` API 函数返回一个 `BOOL` 来表示成功或失败，因此该函数可以执行错误检查，并在 API 调用失败时引发异常:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

如果*errcheck*函数原封不动地返回它所接收的参数元组，则*ctypes*会继续对输出形参执行正常处理。如果你希望返回一个窗口坐标的元组而非 `RECT` 实例，你可以在函数中检索字段并返回它们，正常处理将不会再执行:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

工具函数

ctypes.**addressof**(*obj*)

以整数形式返回内存缓冲区地址。*obj* 必须为一个 ctypes 类型的实例。

引发一个审计事件 `ctypes.addressof` 并附带参数 `obj`。

ctypes.**alignment**(*obj_or_type*)

返回一个 ctypes 类型的对齐要求。*obj_or_type* 必须为一个 ctypes 类型或实例。

ctypes.**byref**(*obj*[, *offset*])

返回指向 *obj* 的轻量指针，该对象必须为一个 ctypes 类型的实例。*offset* 默认值为零，且必须为一个将被添加到内部指针值的整数。

`byref(obj, offset)` 对应于这段 C 代码:

```
(((char *)&obj) + offset)
```

返回的对象只能被用作外部函数调用形参。它的行为类似于 `pointer(obj)`，但构造起来要快很多。

ctypes.**cast**(*obj*, *type*)

此函数类似于 C 的强制转换运算符。它返回一个 *type* 的新实例，该实例指向与 *obj* 相同的内存块。*type* 必须为指针类型，而 *obj* 必须为可以被作为指针来解读的对象。

ctypes.**create_string_buffer**(*init_or_size*, *size=None*)

此函数会创建一个可变的字符缓冲区。返回的对象是一个*c_char*的 ctypes 数组。

init_or_size 必须是一个指明数组大小的整数，或者是一个将被用来初始化数组条目的字节串对象。

如果将一个字节串对象指定为第一个参数，则将使缓冲区大小比其长度多一项以便数组的最后一项为一个 NUL 终结符。可以传入一个整数作为第二个参数以允许在不使用字节串长度的情况下指定数组大小。

引发一个审计事件 `ctypes.create_string_buffer` 并附带参数 `init`, `size`。

`ctypes.`**`create_unicode_buffer`**(*init_or_size*, *size=None*)

此函数会创建一个可变的 unicode 字符缓冲区。返回的对象是一个 *c_wchar* 的 ctypes 数组。

init_or_size 必须是一个指明数组大小的整数，或者是一个将被用来初始化数组条目的字符串。

如果将一个字符串指定为第一个参数，则将使缓冲区大小比其长度多一项以便数组的最后一项为一个 NUL 终结符。可以传入一个整数作为第二个参数以允许在不使用字符串长度的情况下指定数组大小。

引发一个审计事件 `ctypes.create_unicode_buffer` 并附带参数 `init`, `size`。

`ctypes.`**`DllCanUnloadNow`**()

此函数是一个允许使用 ctypes 实现进程内 COM 服务器的钩子。它将由 _ctypes 扩展所导出的 DllCanUnloadNow 函数来调用。

Availability: Windows

`ctypes.`**`DllGetClassObject`**()

此函数是一个允许使用 ctypes 实现进程内 COM 服务器的钩子。它将由 `_ctypes` 扩展 DLL 所导出的 DllGetClassObject 函数来调用。

Availability: Windows

`ctypes.util.`**`find_library`**(*name*)

尝试寻找一个库并返回路径名称。*name* 是库名称并且不带任何前缀如 `lib` 以及后缀如 `.so`，`.dylib` 或版本号（形式与 posix 链接器选项 `-l` 所用的一致）。如果找不到库，则返回 `None`。

确切的功能取决于系统。

`ctypes.util.`**`find_msvcrt`**()

返回 Python 以及扩展模块所使用的 VC 运行时库的文件名。如果库名称无法确定，则返回 `None`。

如果你需要通过调用 `free(void *)` 来释放内存，例如某个扩展模块所分配的内存，重要的一点是你应当使用分配内存的库中的函数。

Availability: Windows

`ctypes.`**`FormatError`**([*code*])

返回错误码 *code* 的文本描述。如果未指定错误码，则会通过调用 Windows API 函数 GetLastError 来获取最近的错误码。

Availability: Windows

`ctypes.`**`GetLastError`**()

返回 Windows 在调用线程中设置的最近的错误码。此函数会直接调用 Windows `GetLastError()` 函数，它并不返回错误码的 ctypes 私有副本。

Availability: Windows

`ctypes.`**`get_errno`**()

返回调用线程中系统 *errno* 变量的 ctypes 私有副本的当前值。

引发一个不带参数的审计事件 `ctypes.get_errno`。

`ctypes.`**`get_last_error`**()

返回调用线程中系统 `LastError` 变量的 ctypes 私有副本的当前值。

Availability: Windows

引发一个不带参数的审计事件 `ctypes.get_last_error`。

ctypes.**memmove**(*dst*, *src*, *count*)

与标准 C memmove 库函数相同：将 *count* 个字节从 *src* 拷贝到 *dst*。*dst* 和 *src* 必须为整数或可被转换为指针的 ctypes 实例。

ctypes.**memset**(*dst*, *c*, *count*)

与标准 C memset 库函数相同：将位于地址 *dst* 的内存块用 *count* 个字节的 *c* 值填充。*dst* 必须为指定地址的整数或 ctypes 实例。

ctypes.**POINTER**(*type*, /)

创建并返回一个新的 ctypes 指针类型。指针类型会被缓存并在内部重复使用，因此重复调用此函数耗费不大。*type* 必须为 ctypes 类型。

ctypes.**pointer**(*obj*, /)

创建一个新的指针实例，指向 *obj*。返回的对象类型为 `POINTER(type(obj))`。

注意：如果你只是想向外部函数调用传递一个对象指针，你应当使用更为快速的 `byref(obj)`。

ctypes.**resize**(*obj*, *size*)

此函数可改变 *obj* 的内部内存缓冲区大小，其参数必须为 ctypes 类型的实例。没有可能将缓冲区设为小于对象类型的本机大小值，该值由 `sizeof(type(obj))` 给出，但将缓冲区加大则是可能的。

ctypes.**set_errno**(*value*)

设置调用线程中系统*errno* 变量的 ctypes 私有副本的当前值为 *value* 并返回原来的值。

引发一个审计事件 `ctypes.set_errno` 并附带参数 `errno`。

ctypes.**set_last_error**(*value*)

在调用线程中将系统 `LastError` 变量的 ctypes 私有副本的当前值设为 *value* 并返回之前的值。

Availability: Windows

引发一个审计事件 `ctypes.set_last_error` 并附带参数 `error`。

ctypes.**sizeof**(*obj_or_type*)

返回 ctypes 类型或实例的内存缓冲区以字节表示的大小。其功能与 C `sizeof` 运算符相同。

ctypes.**string_at**(*ptr*, *size=-1*)

返回位于 *void *ptr* 的字节串。如果指定了 *size*，它将被用作字节串的大小，否则将假定字节串以零值结尾。

引发一个审计事件 `ctypes.string_at` 并附带参数 `ptr`, `size`。

ctypes.**WinError**(*code=None*, *descr=None*)

此函数可能是 ctypes 中命名得最糟糕的。它会创建一个*OSError* 的实例。如果未指定 *code*，则会调用 `GetLastError` 来确定错误码。如果未指定 *descr*，则会调用*FormatError()* 来获取错误的文本描述。is called to get a textual description of the error.

Availability: Windows

在 3.3 版本发生变更: 过去会创建*WindowsError* 的实例，现在它是*OSError* 的别名。

ctypes.**wstring_at**(*ptr*, *size=-1*)

返回位于 *void *ptr* 的宽字符串。如果指定了 *size*，它将被用作字符串的字符数量，否则将假定字符串以零值结尾。

引发一个审计事件 `ctypes.wstring_at` 并附带参数 `ptr`, `size`。

数据类型

class ctypes.**_CData**

这个非公有类是所有 ctypes 数据类型的共同基类。另外，所有 ctypes 类型的实例都包含一个存放 C 兼容数据的内存块；该内存块的地址可由*addressof()* 辅助函数返回。还有一个实例变量被公开为*_objects*；此变量包含其他在内存块包含指针的情况下需要保持存活的 Python 对象。

ctypes 数据类型的通用方法，它们都是类方法（严谨地说，它们是*metaclass* 的方法）：

from_buffer(*source*[, *offset*])

此方法返回一个共享 *source* 对象缓冲区的 ctypes 实例。*source* 对象必须支持可写缓冲区接口。可选的 *offset* 形参指定以字节表示的源缓冲区内偏移量；默认值为零。如果源缓冲区不够大则会引发 *ValueError*。

引发一个审计事件 `ctypes.cdata/buffer` 并附带参数 `pointer`, `size`, `offset`。

from_buffer_copy(*source*[, *offset*])

此方法创建一个 ctypes 实例，从 *source* 对象缓冲区拷贝缓冲区，该对象必须是可读的。可选的 *offset* 形参指定以字节表示的源缓冲区内偏移量；默认值为零。如果源缓冲区不够大则会引发 *ValueError*。

引发一个审计事件 `ctypes.cdata/buffer` 并附带参数 `pointer`, `size`, `offset`。

from_address(*address*)

此方法会使用 *address* 所指定的内存返回一个 ctypes 类型的实例，该参数必须为一个整数。

这个方法以及其他间接调用了它的方法会引发一个审计事件 `ctypes.cdata`，附带参数 `address`。

from_param(*obj*)

此方法会将 *obj* 适配为一个 ctypes 类型。当该类型出现在外部函数的 *argtypes* 元组中时它将会被调用并传入在外部函数中使用的实际对象；它必须返回一个可被用作函数调用形参的对象。

所有 ctypes 数据类型都带有这个类方法的默认实现，它通常会返回 *obj*，如果该对象是此类型的实例的话。某些类型也能接受其他对象。

in_dll(*library*, *name*)

此方法返回一个由共享库导出的 ctypes 类型。*name* 为导出数据的符号名称，*library* 为所加载的共享库。

ctypes 数据类型的通用实例变量:

_b_base_

有时 ctypes 数据实例并不拥有它们所包含的内存块，它们只是共享了某个基对象的部分内存块。*_b_base_* 只读成员是拥有内存块的根 ctypes 对象。

_b_needsfree_

这个只读变量在 ctypes 数据实例自身已分配了内存块时为真值，否则为假值。

_objects

这个成员或者为 `None`，或者为一个包含需要保持存活以使内存块的内存保持有效的 Python 对象的字典。这个对象只是出于调试目的而对外公开；绝对不要修改此字典的内容。

基础数据类型

class ctypes.**_SimpleCData**

这个非公有类是所有基本 ctypes 数据类型的基类。它在这里被提及是因为它包含基本 ctypes 数据类型共有的属性。*_SimpleCData* 是 *_CData* 的子类，因此继承了其方法和属性。非指针及不包含指针的 ctypes 数据类型现在将可以被封存。

实例拥有一个属性:

value

这个属性包含实例的实际值。对于整数和指针类型，它是一个整数，对于字符类型，它是一个单字符字符串对象或字符串，对于字符指针类型，它是一个 Python 字节串对象或字符串。

当从 ctypes 实例提取 `value` 属性时，通常每次会返回一个新的对象。*ctypes* 并没有实现原始对象返回，它总是会构造一个新的对象。所有其他 ctypes 对象实例也同样如此。

当作为外部函数调用结果，或者举例来说，作为结构字段成员或数组条目被提取时，基本数据类型会被透明与转换为原生 Python 类型。换句话说，如果某个外部函数的 *restype* 是 *c_char_p*，那么你将总是得到一个 Python 字节串对象，而 不是一个 *c_char_p* 实例。

基本数据类型的子类 不会继承这种行为。因此，如果一个外部函数的 `restype` 是*c_void_p* 的子类，则你将从函数调用得到一个该子类的实例。当然，你可以通过访问 `value` 属性来获取指针的值。

这些是基本 ctypes 数据类型:

class ctypes.**c_byte**

代表 C `signed char` 数据类型，并将值解读为一个小整数。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_char**

代表 C `char` 数据类型，并将值解读为单个字符。该构造器接受一个可选的字符串初始值，字符串的长度必须恰好为一个字符。

class ctypes.**c_char_p**

当指向一个以零为结束符的字符串时代表 C `char*` 数据类型。对于通用字符指针来说也可能指向二进制数据，必须要使用 `POINTER(c_char)`。该构造器接受一个整数地址，或者一个字节串对象。

class ctypes.**c_double**

代表 C `double` 数据类型。该构造器接受一个可选的浮点数初始值。

class ctypes.**c_longdouble**

代表 C `long double` 数据类型。该构造器接受一个可选的浮点数初始值。在 `sizeof(long double) == sizeof(double)` 的平台上它是*c_double* 的一个别名。

class ctypes.**c_float**

代表 C `float` 数据类型。该构造器接受一个可选的浮点数初始值。datatype. The constructor accepts an optional float initializer.

class ctypes.**c_int**

代表 C `signed int` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。在 `sizeof(int) == sizeof(long)` 的平台上它是*c_long* 的一个别名。

class ctypes.**c_int8**

代表 C 8 位 `signed int` 数据类型。通常是*c_byte* 的一个别名。

class ctypes.**c_int16**

代表 C 16 位 `signed int` 数据类型。通常是*c_short* 的一个别名。

class ctypes.**c_int32**

代表 C 32 位 `signed int` 数据类型。通常是*c_int* 的一个别名。

class ctypes.**c_int64**

代表 C 64 位 `signed int` 数据类型。通常是*c_longlong* 的一个别名。

class ctypes.**c_long**

代表 C `signed long` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_longlong**

代表 C `signed long long` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_short**

代表 C `signed short` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_size_t**

代表 C `size_t` 数据类型。

class ctypes.**c_ssize_t**

代表 C `ssize_t` 数据类型。

Added in version 3.2.

class ctypes.**c_time_t**

代表 C `time_t` 数据类型。

Added in version 3.12.

class ctypes.**c_ubyte**

代表 C `unsigned char` 数据类型，它将值解读为一个小整数。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_uint**

代表 C `unsigned int` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。在 `sizeof(int) == sizeof(long)` 的平台上它是*c_ulong* 的一个别名。

class ctypes.**c_uint8**

代表 C 8 位 `unsigned int` 类型。通常是*c_ubyte* 的一个别名。.

class ctypes.**c_uint16**

代表 C 16 位 `unsigned int` 数据类型。通常是*c_ushort* 的一个别名。.

class ctypes.**c_uint32**

代表 C 32 位 `unsigned int` 数据类型。通常是*c_uint* 的一个别名。

class ctypes.**c_uint64**

代表 C 64 位 `unsigned int` 数据类型。通常是*c_ulonglong* 的一个别名。

class ctypes.**c_ulong**

代表 C `unsigned long` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_ulonglong**

代表 C `unsigned long long` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_ushort**

代表 C `unsigned short` 数据类型。该构造器接受一个可选的整数初始值；不会执行溢出检查。

class ctypes.**c_void_p**

代表 C `void*` 类型。该值被表示为整数形式。该构造器接受一个可选的整数初始值。

class ctypes.**c_wchar**

代表 C `wchar_t` 数据类型，并将值解读为一单个字符的 unicode 字符串。该构造器接受一个可选的字符串初始化器，字符串的长度必须恰好为一个字符。

class ctypes.**c_wchar_p**

代表 C `wchar_t*` 数据类型，它必须为指向以零为续签符的宽字符串的指针。该构造器接受一个整数地址，或一个字符串。

class ctypes.**c_bool**

代表 C `bool` 数据类型 (更准确地说，是 C99 `_Bool`)。它的值可以为 `True` 或 `False`，并且该构造器接受任何具有逻辑值的对象。

class ctypes.**HRESULT**

代表一个 `HRESULT` 值，它包含某个函数或方法调用的成功或错误信息。

Availability: Windows

class ctypes.**py_object**

代表 C `PyObject*` 数据类型。不带参数地调用此构造器将创建一个 `NULL` `PyObject*` 指针。

`ctypes.wintypes` 模块提供了其他许多 Windows 专属的数据类型，例如 `HWND`, `WPARAM` 或 `DWORD`。还定义了一些有用的结构体如 `MSG` 或 `RECT`。

结构化数据类型

class ctypes.**Union**(**args*, ***kw*)

本机字节序的联合所对应的抽象基类。

class ctypes.**BigEndianUnion**(**args*, ***kw*)

大端字节序的联合所对应的抽象基类。

Added in version 3.11.

class ctypes.**LittleEndianUnion**(**args*, ***kw*)

小端字节序的联合所对应的抽象基类。

Added in version 3.11.

class ctypes.**BigEndianStructure**(**args*, ***kw*)

大端字节序的结构体所对应的抽象基类。

class ctypes.**LittleEndianStructure**(**args*, ***kw*)

小端字节序的结构体所对应的抽象基类。

非本机字节序的结构体和联合不能包含指针类型字段，或任何其他包含指针类型字段的数据类型。

class ctypes.**Structure**(**args*, ***kw*)

本机字节序的结构体所对应的抽象基类。

实际的结构体和联合类型必须通过子类化这些类型之一来创建，并且至少要定义一个*_fields_* 类变量。*ctypes* 将创建*descriptor*，它允许通过直接属性访问来读取和写入字段。这些是

fields

一个定义结构体字段的序列。其中的条目必须为 2 元组或 3 元组。元组的第一项是字段名称，第二项指明字段类型；它可以是任何 ctypes 数据类型。

对于整数类型字段例如*c_int*，可以给定第三个可选项。它必须是一个定义字段比特位宽度的小正整数。

字段名称在一个结构体或联合中必须唯一。不会检查这个唯一性，但当名称出现重复时将只有一个字段可被访问。

可以在定义 Structure 子类的类语句之后再定义*_fields_* 类变量，这将允许创建直接或间接引用其自身的数据类型:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

但是，*_fields_* 类变量必须在类型第一次被使用（创建实例，调用*sizeof()* 等等）之前进行定义。在此之后对*_fields_* 类变量赋值将会引发 AttributeError。

可以定义结构体类型的子类，它们会继承基类的字段再加上在子类中定义的任何*_fields_*。

pack

一个可选的小整数，它允许覆盖实例中结构体字段的对齐方式。当*_fields_* 被赋值时必须已经定义了*_pack_*，否则它将没有效果。将该属性设为 0 的效果与不设置它一样。

align

一个可选的小整数，它允许覆盖结构体在针对内存执行打包或解包时的对齐方式。将该属性设为 0 与完全不设置它效果相同。

Added in version 3.13.

anonymous

一个可选的序列，它会列出未命名（匿名）字段的名称。当*_fields_* 被赋值时必须已经定义了*_anonymous_*，否则它将没有效果。

在此变量中列出的字段必须为结构体或联合类型字段。*ctypes* 将在结构体类型中创建描述器以允许直接访问嵌套字段，而无需创建对应的结构体或联合字段。

以下是一个示例类型（Windows）：

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

TYPEDESC 结构体描述了一个 COM 数据类型，vt 字段指明哪个联合字段是有效的。由于 u 字段被定义为匿名字段，现在可以直接从 TYPEDESC 实例访问成员。td.lptdesc 和 td.u.lptdesc 是等价的，但前者速度更快，因为它不需要创建临时的联合实例：

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

可以定义结构体的子类，它们会继承基类的字段。如果子类定义具有单独的*_fields_* 变量，在其中指定的字段会被添加到基类的字段中。

结构体和联合的构造器均可接受位置和关键字参数。位置参数用于按照*_fields_* 中的出现顺序来初始化成员字段。构造器中的关键字参数会被解读为属性赋值，因此它们将以相应的名称来初始化*_fields_*，或为不存在于*_fields_* 中的名称创建新的属性。

数组与指针

class ctypes.**Array**(**args*)

数组的抽象基类。

创建实体数组类型的推荐方式是通过将任意*ctypes* 数据类型与一个非负整数相乘。作为替代方式，你也可以子类化这个类型并定义*_length_* 和*_type_* 类变量。数组元素可使用标准的抽取和切片操作来进行读写；对于切片读取，结果对象本身 不是一个*Array*。

length

一个指明数组中元素数量的正整数。超出范围的抽取会导致*IndexError*。该值将由*len()* 返回。

type

指明数组中每个元素的类型。

Array 子类构造器可接受位置参数，用来按顺序初始化元素。

ctypes.**ARRAY**(*type*, *length*)

创建一个数组。等价于 type * length，其中 *type* 是一个*ctypes* 数据类型而 *length* 是一个整数。

该函数已被*soft deprecated* 而应改用乘法。尚无移除它的计划。

class ctypes.**_Pointer**

私有对象，指针的抽象基类。

实际的指针类型是通过调用*POINTER()* 并附带其将指向的类型来创建的；这会由*pointer()* 自动完成。

如果一个指针指向的是数组，则其元素可使用标准的抽取和切片方式来读写。指针对象没有长度，因此*len()* 将引发*TypeError*。抽取负值将会从指针 之前的内存中读取（与 C 一样），并且超出范围的抽取将可能因非法访问而导致崩溃（视你的运气而定）。

type

指明所指向的类型。

contents

返回指针所指向的对象。对此属性赋值会使指针改为指向所赋值的对象。

CHAPTER 17

Command Line Interface Libraries

The modules described in this chapter assist with implementing command line and terminal interfaces for applications.

Here's an overview:

17.1 argparse --- 用于命令行选项、参数和子命令的解析器

Added in version 3.2.

源代码： Lib/argparse.py

备注

虽然*argparse*是推荐的用于实现基本命令行应用程序的标准库模块，但对于命令行应用程序的行为有更明确的要求的开发者可能会发现它并未提供所需层次的控制能力。请参阅*选择参数解析库*了解当 argparse 不支持特定应用程序需要的行为（例如完全禁用散开选项和位置参数，或接受以 - 开头的选项形参值即使它们对应了其它已定义的选项）时可以考虑的替代。

教程

此页面包含该 API 的参考信息。有关 Python 命令行解析更细致的介绍，请参阅*argparse 教程*。

argparse 模块让编写用户友好的命令行接口变得容易。程序定义它需要哪些参数，argparse 将会知道如何从*sys.argv*解析它们。argparse 模块还能自动生成帮助和用法消息文本。该模块还会在用户向程序传入无效参数时发出错误提示。

argparse 模块对命令行界面的支持是围绕*argparse.ArgumentParser*实例建立的。它是一个用于参数规格说明的容器并包含应用于解析器的一组选项:

```
parser = argparse.ArgumentParser(
                    prog='ProgramName',
                    description='What the program does',
                    epilog='Text at the bottom of help')
```

ArgumentParser.add_argument() 方法将单个参数规格说明关联到解析器。它支持位置参数，接受各种值的选项，以及各种启用/禁用旗标:

```
parser.add_argument('filename')           # 位置参数
parser.add_argument('-c', '--count')      # 接受一个值的选项
parser.add_argument('-v', '--verbose',
                    action='store_true')  # 启用/禁用旗标
```

ArgumentParser.parse_args() 方法运行解析器并将提取的数据放入*argparse.Namespace* 对象:

```
args = parser.parse_args()
print(args.filename, args.count, args.verbose)
```

备注

如果您正在寻找如何将*optparse* 代码升级到 `argparse` 的指南，请参阅*Upgrading Optparse Code*。

17.1.1 ArgumentParser 对象

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*)

创建一个新的*ArgumentParser* 对象。所有的参数都应当作为关键字参数传入。每个参数在下面都有它更详细的描述，但简而言之，它们是:

- *prog* - 程序的名称 (默认值: `os.path.basename(sys.argv[0])`)
- *usage* - 描述程序用途的字符串（默认值：从添加到解析器的参数生成）
- *description* - 要在参数帮助信息之前显示的文本（默认：无文本）
- *epilog* - 要在参数帮助信息之后显示的文本（默认：无文本）
- *parents* - 一个*ArgumentParser* 对象的列表，它们的参数也应包含在内
- *formatter_class* - 用于自定义帮助文档输出格式的类
- *prefix_chars* - 可选参数的前缀字符集合（默认值：'-'）
- *fromfile_prefix_chars* - 当需要从文件中读取其他参数时，用于标识文件名的前缀字符集合（默认值：`None`）
- *argument_default* - 参数的全局默认值（默认值：`None`）
- *conflict_handler* - 解决冲突选项的策略（通常是不必要的）
- *add_help* - 为解析器添加一个 `-h/--help` 选项（默认值：`True`）
- *allow_abbrev* - 如果缩写是无歧义的，则允许缩写长选项（默认值：`True`）
- *exit_on_error* - 确定当出现错误时，`ArgumentParser` 是否退出并显示错误信息。(默认值:`True`)

在 3.5 版本发生变更: 增加了 *allow_abbrev* 参数。

在 3.8 版本发生变更: 在之前的版本中，*allow_abbrev* 还会禁用短旗标分组，例如 `-vv` 表示为 `-v -v`。

在 3.9 版本发生变更: 添加了 *exit_on_error* 形参。

以下部分描述这些参数如何使用。

prog

默认情况下，*ArgumentParser* 根据 Python 解释器的运行方式，计算出要在帮助信息中显示的程序名称：

- 如果作为参数传递的是文件，则是 “sys.argv[0]“ 的*base name* 。
- 如果作为参数传递的是目录或 ZIP 文件，则是 Python 解释器名称，后接 “sys.argv[0]“。
- 如果使用 `-m` 选项则是在 Python 解释器名称后接 `-m` 再加模块或包的名称。

默认行为几乎总是符合需求因为它将使帮助消息与在命令行上唤起程序所用的字符串相匹配。不过，要更改此默认行为，可以将 `prog=` 参数传入*ArgumentParser* 以提供另外的值:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

options:
 -h, --help  show this help message and exit
```

需要注意的是，无论是从 `sys.argv[0]` 或是从 `prog=` 参数确定的程序名称，都可以在帮助消息里通过 `%(prog)s` 格式说明符来引用。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage（用法）

默认情况下，*ArgumentParser* 会从它所包含的参数中计算出用法消息。默认消息可以通过 `usage=` 关键字参数进行覆盖。

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

options:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

在用法消息中可以使用 `%(prog)s` 格式说明符来填入程序名称。

description（描述）

大多数对*ArgumentParser* 构造器的调用都会使用 `description=` 关键字参数。这个参数简要描述这个程序做什么以及怎么做。在帮助消息中，这个描述会显示在命令行用法字符串和各种参数的帮助消息之间。

在默认情况下，description 将被换行以便适应给定的空间。如果想改变这种行为，见*formatter_class* 参数。

epilog

一些程序喜欢在 description 参数后显示额外的对程序的描述。这种文字能够通过给*ArgumentParser*:: 提供 `epilog=` 参数而被指定。

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

options:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

和*description* 参数一样，`epilog=` text 在默认情况下会换行，但是这种行为能够被调整通过提供*formatter_class* 参数给 `ArgumentParse`.

parents

有些时候，少数解析器会使用同一系列参数。单个解析器能够通过提供 `parents=` 参数给*ArgumentParser* 而使用相同的参数而不是重复这些参数的定义。`parents=` 参数使用*ArgumentParser* 对象的列表，从它们那里收集所有的位置和可选的行为，然后将这写行为加到正在构建的*ArgumentParser* 对象。

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

请注意大多数父解析器会指定 `add_help=False` . 否则，`ArgumentParse` 将会看到两个 `-h/--help` 选项（一个在父参数中一个在子参数中）并且产生一个错误。

> **备注**
>
> 你在通过 `parents=` 传递解析器之前必须完全初始化它们。如果你在子解析器之后改变父解析器，这些改变将不会反映在子解析器上。

formatter_class

ArgumentParser 对象允许通过指定备用格式化类来自定义帮助格式。目前，有四种这样的类。

class argparse.**RawDescriptionHelpFormatter**

class argparse.**RawTextHelpFormatter**

class argparse.**ArgumentDefaultsHelpFormatter**

class argparse.**MetavarTypeHelpFormatter**

RawDescriptionHelpFormatter 和*RawTextHelpFormatter* 在正文的描述和展示上给与了更多的控制。*ArgumentParser* 对象会将*description* 和*epilog* 的文字在命令行中自动换行。

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

options:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

传*RawDescriptionHelpFormatter* 给 formatter_class= 表示*description* 和*epilog* 已经被正确的格式化了，不能在命令行中被自动换行:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

options:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter 保留各种帮助文本的空白符，包括参数描述。但是，多个换行符会被替换为一个。如果你希望保留多个空行，请在换行符之间添加空格。

ArgumentDefaultsHelpFormatter 自动添加默认的值的信息到每一个帮助信息的参数中:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

options:
```

(续下页)

(接上页)

```
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter 为它的值在每一个参数中使用*type* 的参数名当作它的显示名（而不是使用通常的格式*dest* ）:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

options:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

大多数命令行选项会使用 - 作为前缀，例如 `-f/--foo`。如果解析器需要支持不同的或者额外的前缀字符，例如像 `+f` 或 `/foo` 之类的选项，可以在*ArgumentParser* 构造器中使用 `prefix_chars=` 参数来指定它们:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

`prefix_chars=` 参数默认使用 `'-'`。提供一组不包括 - 的字符将导致 `-f/--foo` 选项不被允许。

fromfile_prefix_chars

在某些时候，如在处理一个特别长的参数列表时，把参数列表保存在一个文件中而不是在命令行中打印出来会更有意义。如果提供 `fromfile_prefix_chars=` 参数给*ArgumentParser* 构造器，则任何以指定字符打头的参数都将被当作文件来处理，并将被它们包含的参数所替代。举例来说:

```
>>> with open('args.txt', 'w', encoding=sys.getfilesystemencoding()) as fp:
...     fp.write('-f\nbar')
...
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

从文件读取的参数在默认情况下必须一个一行（但是可参见*convert_arg_line_to_args()*）并且它们被视为与命令行上的原始文件引用参数位于同一位置。所以在以上例子中，`['-f', 'foo', '@args.txt']` 的表示和 `['-f', 'foo', '-f', 'bar']` 的表示相同。

ArgumentParser 使用*filesystem encoding and error handler* 来读取包含参数的文件。

`fromfile_prefix_chars=` 参数默认为 `None`，意味着参数不会被当作文件对待。

在 3.12 版本发生变更: *ArgumentParser* 将读取参数的编码格式和错误处理方式从默认值 (即*locale.getpreferredencoding(False)* 和 `"strict"`) 改为*filesystem encoding and error handler*。在 Windows 上参数文件应当以 UTF-8 而不是 ANSI 代码页来编码。

argument_default

一般情况下，参数默认会通过设置一个默认到*add_argument()* 或者调用带一组指定键值对的*ArgumentParser.set_defaults()* 方法。但是有些时候，为参数指定一个普遍适用的解析器会更有用。这能够通过传输 argument_default= 关键词参数给*ArgumentParser* 来完成。举个栗子，要全局禁止在*parse_args()* 中创建属性，我们提供 argument_default=SUPPRESS:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

正常情况下，当你向*ArgumentParser* 的*parse_args()* 方法传入一个参数列表时，它会*recognizes abbreviations*。

这个特性可以设置 allow_abbrev 为 False 来关闭:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Added in version 3.5.

conflict_handler

ArgumentParser 对象不允许在相同选项字符串下有两种行为。默认情况下，*ArgumentParser* 对象会产生一个异常如果去创建一个正在使用的选项字符串参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

有些时候（例如：使用*parents*），重写旧的有相同选项字符串的参数会更有用。为了产生这种行为，'resolve' 值可以提供给*ArgumentParser* 的 conflict_handler= 参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

options:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

注意*ArgumentParser* 对象只能移除一个行为如果它所有的选项字符串都被重写。所以，在上面的例子中，旧的 -f/--foo 行为回合 -f 行为保持一样，因为只有 --foo 选项字符串被重写。

add_help

在默认情况下，*ArgumentParser* 对象会添加一个简单地显示解析器的帮助消息的选项。如果在命令行中提供了 `-h` 或 `--help`，则会打印 `ArgumentParser` 帮助消息。

有时候可能会需要关闭额外的帮助信息。这可以通过在*ArgumentParser* 中设置 `add_help=` 参数为 `False` 来实现。

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

options:
 --foo FOO  foo help
```

帮助选项一般为 `-h/--help`。如果 `prefix_chars=` 被指定并且没有包含 `-` 字符，在这种情况下，`-h --help` 不是有效的选项。此时，`prefix_chars` 的第一个字符将用作帮助选项的前缀。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

options:
  +h, ++help  show this help message and exit
```

exit_on_error

通常，当你向*ArgumentParser parse_args()* 方法传入一个无效的参数列表时，它会将 *message* 打印到*sys.stderr* 并附带状态码 2 退出程序。

如果用户想要手动捕获错误，可通过将 `exit_on_error` 设为 `False` 来启用该特性:

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None, const=None,␣
↪default=None, type=<class 'int'>, choices=None, help=None, metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

Added in version 3.9.

17.1.2 add_argument() 方法

`ArgumentParser.`**`add_argument`**(*name or flags...*, *[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*][, *deprecated*])

定义单个的命令行参数应当如何解析。每个形参都在下面有它自己更多的描述，长话短说有:

- *name or flags* - 一个名称或是由选项字符串组成的列表，例如 `'foo'` 或 `'-f', '--foo'`。
- *action* - 当参数在命令行中出现时使用的动作基本类型。
- *nargs* - 命令行参数应当消耗的数目。
- *const* - 被一些*action* 和*nargs* 选择所需求的常数。
- *default* - 当参数未在命令行中出现并且也不存在于命名空间对象时所产生的值。
- *type* - 命令行参数应当被转换成的类型。
- *choices* - 由允许作为参数的值组成的序列。

- *required* - 此命令行选项是否可省略（仅选项可用）。
- *help* - 一个此选项作用的简单描述。
- *metavar* - 在使用方法消息中使用的参数值示例。
- *dest* - 被添加到*parse_args()* 所返回对象上的属性名。
- *deprecated* - 参数的使用是否已被弃用。

以下部分描述这些参数如何使用。

name or flags

add_argument() 方法必须知道是要接收一个可选参数，如 `-f` 或 `--foo`，还是一个位置参数，如由文件名组成的列表。因此首先传递给*add_argument()* 的参数必须是一组旗标，或一个简单的参数名称。

例如，可以这样创建可选参数:

```
>>> parser.add_argument('-f', '--foo')
```

而位置参数可以这么创建:

```
>>> parser.add_argument('bar')
```

当*parse_args()* 被调用，选项会以 - 前缀识别，剩下的参数则会被假定为位置参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

action

ArgumentParser 对象将命令行参数与动作相关联。这些动作可以做与它们相关联的命令行参数的任何事，尽管大多数动作只是简单的向*parse_args()* 返回的对象上添加属性。`action` 命名参数指定了这个命令行参数应当如何处理。供应的动作有:

- `'store'` - 这用于存储参数的值。这是默认的动作。
- `'store_const'` - 存储由*const* 关键字参数指定的值；请注意*const* 关键字参数默认为 `None`。`'store_const'` 动作最常被用于指定某类旗标的可选参数。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- `'store_true'` 和 `'store_false'` - 这些是 `'store_const'` 的特例，分别用于存储 `True` 和 `False` 值。此外，它们还会分别创建默认的 `False` 和 `True` 值:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_true')
  >>> parser.add_argument('--bar', action='store_false')
  >>> parser.add_argument('--baz', action='store_false')
  >>> parser.parse_args('--foo --bar'.split())
  Namespace(foo=True, bar=False, baz=True)
  ```

- `'append'` - 存储一个列表，并将每个参数值添加到该列表。它适用于允许可多次指定的选项。如果默认值非空，则默认的元素将出现在该选项的已解析值中，并将所有来自命令行的值添加在默认值之后。示例用法:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- `'append_const'` - 存储一个列表，并将由*const* 关键字参数指定的值添加到列表中；请注意*const* 关键字参数默认为 `None`。`'append_const'` 动作通常适用于多个参数需要将常量存储到同一列表的场合。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'extend'` - 这将存储一个列表并添加来自于多值参数列表的每个条目。`'extend'` 动作通常配合*nargs* 关键字参数值 `'+'` 或 `'*'` 使用。请注意当*nargs* 为 `None` (默认值) 或 `'?'` 时，将把参数字符串中的每个字符添加到列表。示例用法:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

  Added in version 3.8.

- `'count'` - 计算一个关键字参数出现的数目或次数。例如，对于一个增长的详情等级来说有用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

  请注意，*default* 将为 `None`，除非显式地设为 *0*。

- `'help'` - 打印所有当前解析器中的选项和参数的完整帮助信息，然后退出。默认情况下，一个 help 动作会被自动加入解析器。关于输出是如何创建的，参与*ArgumentParser*。

- `'version'` - 期望有一个 `version=` 命名参数在*add_argument()* 调用中，并打印版本信息并在调用后退出:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

只有消耗命令行参数的动作 (例如 `'store'`, `'append'` 或 `'extend'`) 可以配合位置参数使用。

class argparse.**BooleanOptionalAction**

你还可以通过传递一个*Action* 子类或实现了相同接口的其他对象来指定任意动作。`BooleanOptionalAction` 在 `argparse` 中可用并会添加对布尔类型动作如 `--foo` 和 `--no-foo` 的支持:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

Added in version 3.9.

创建自定义动作的推荐方式是扩展*Action*，重写 `__call__()` 方法以及可选的 `__init__()` 和 `format_usage()` 方法。你还可以使用*register()* 方法注册自定义动作并通过其注册名称来引用它们。

一个自定义动作的例子:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

更多描述，见*Action*。

nargs

ArgumentParser 对象通常会将一个单独的命令行参数关联到一个单独的要执行的动作。`nargs` 关键字参数会将不同数量的命令行参数关联到一个单独的动作。另请参阅指定有歧义的参数。受支持的值有:

- `N` （一个整数）。命令行中的 `N` 个参数会被聚集到一个列表中。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  注意 `nargs=1` 会产生一个单元素列表。这和默认的元素本身是不同的。

- `'?'`。如果可能的话，会从命令行中消耗一个参数，并产生一个单独项。如果当前没有命令行参数，将会产生*default* 值。注意对于可选参数来说，还有一个额外情况——出现了选项字符串但没有跟随命令行参数，在此情况下将会产生*const* 值。一些说明这种情况的例子如下:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  Namespace(bar='d', foo='d')
  ```

  `nargs='?'` 的一个更普遍用法是允许可选的输入或输出文件:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
  ...                     default=sys.stdin)
  >>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
  ```

(续下页)

(接上页)

```
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- `'*'`。所有当前命令行参数被聚集到一个列表中。注意通过 `nargs='*'` 来实现多个位置参数通常没有意义，但是多个选项是可能的。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`。和 `'*'` 类似，所有当前命令行参数被聚集到一个列表中。另外，当前没有至少一个命令行参数时会产生一个错误信息。例如:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

如果未提供 `nargs` 关键字参数，则消耗参数的数量将由*action* 来确定。通常这意味着消耗一个命令行参数并产生一个条目（而非列表）。不消耗命令行参数的动作 (例如 `'store_const'`) 则要设置 `nargs=0`。

const

add_argument() 的 `const` 参数用于保存不从命令行中读取但被各种*ArgumentParser* 动作需求的常数值。最常用的两例为:

- 当*add_argument()* 附带 `action='store_const'` 或 `action='append_const'` 被调用时。这些动作会把 `const` 值添加到*parse_args()* 所返回的对象的属性中。请查看*action* 的示例描述。如果未将 `const` 提供给*add_argument()*，它将接收一个 `None` 的默认值。
- 当*add_argument()* 附带选项字符串 (如 `-f` 或 `--foo`) 和 `nargs='?'` 被调用的时候。这会创建一个可以跟随零到一个命令行参数的选项参数。当解析该命令行时，如果选项字符串没有跟随任何命令行参数，`const` 的值将被假定为以 `None` 代替。请参阅*nargs* 描述中的示例。

在 3.11 版本发生变更: 在默认情况下 `const=None`，包括 `action='append_const'` 或 `action='store_const'` 的时候。

默认值

所有选项和一些位置参数可能在命令行中被忽略。*add_argument()* 的命名参数 `default`，默认值为 `None`，指定了在命令行参数未出现时应当使用的值。对于选项，`default` 值在选项未在命令行中出现时使用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

如果目标命名空间已经有一个属性集，则 *default* 动作不会覆盖它:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

如果 default 值是一个字符串，解析器解析此值就像一个命令行参数。特别是，在将属性设置在*Namespace* 的返回值之前，解析器应用任何提供的*type* 转换参数。否则解析器使用原值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

对于*nargs* 等于 ? 或 * 的位置参数，default 值在没有命令行参数出现时使用。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

对于*required* 参数，default 值将被忽略。例如，这将适用于带有 ? 和 * 以外的*nargs* 值的位置参数，或者被标记为 required=True 的可选参数。

提供 default=argparse.SUPPRESS 导致命令行参数未出现时没有属性被添加:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type -- 类型

默认情况下，解析器会将命令行参数当作简单字符串读入。然而，命令行字符串经常应当被解读为其他类型，例如*float* 或*int*。*add_argument()* 的 type 关键字允许执行任何必要的类型检查和类型转换。

如果*type* 关键字使用了*default* 关键字，则类型转换器仅会在默认值为字符串时被应用。

传给 type 的参数可以是接受一个字符串或一个已注册类型名称 (参见*register()*) 的可调用对象。如果该函数引发了*ArgumentTypeError*，*TypeError* 或*ValueError*，异常将被捕获并显示经良好格式化的错误消息。其他异常类型则不会被处理。

普通内置类型和函数可被用作类型转换器:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('dest_file', type=argparse.FileType('w', encoding='latin-1'))
parser.add_argument('datapath', type=pathlib.Path)
```

用户自定义的函数也可以被使用:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

不建议将*bool()* 函数用作类型转换器。它所做的只是将空字符串转为 `False` 而将非空字符串转为 `True`。这通常不是用户所想要的。

通常，`type` 关键字是仅应被用于只会引发上述三种被支持的异常的简单转换的便捷选项。任何具有更复杂错误处理或资源管理的转换都应当在参数被解析后由下游代码来完成。

例如，JSON 或 YAML 转换具有复杂的错误情况，需要能给出比 `type` 关键字所能给出的更好的报告。*JSONDecodeError* 将不会被良好地格式化而*FileNotFoundError* 异常则完全不会被处理。

甚至*FileType* 在用于 `type` 关键字时也是有限制的。如果一个参数使用了*FileType* 并且有一个后续参数出错，则将报告错误但文件并不会被自动关闭。在此情况下，更好的做法是等待直到解析器运行完毕再使用 `with` 语句来管理文件。

对于简单地检查一组固定值的类型检查器，请考虑改用*choices* 关键字。

choices

某些命令行参数应当从一组受限的值中选择。这可以通过将一个序列对象作为 *choices* 关键字参数传给*add_argument()* 来处理。当执行命令行解析时，参数值将被检查，如果参数不是可接受的值之一就将显示错误消息:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

注意包括在 *choices* 序列中的内容会在执行任意*type* 转换之后被检查，因此 *choices* 序列中对象的类型应当与指定的*type* 相匹配。

任何序列都可作为 *choices* 值传入，因此*list* 对象、*tuple* 对象以及自定义序列都是受支持的。

不建议使用*enum.Enum*，因为要控制其在用法、帮助和错误消息中的外观是很困难的。

已格式化的选项会覆盖默认的 *metavar*，该值一般是派生自 *dest*。这通常就是你所需要的，因为用户永远不会看到 *dest* 形参。如果不想要这样的显示（或许因为有很多选择），只需指定一个显式的*metavar*。

required

通常，`argparse` 模块会假定像 `-f` 和 `--bar` 这样的旗标是表示 可选参数，它们总是可以在命令行中被省略。要让一个选项成为 必需，则可以将 `True` 作为 `required=` 关键字参数传给*add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

如这个例子所示，如果一个选项被标记为 `required`，则当该选项未在命令行中出现时，*parse_args()* 将会报告一个错误。

备注

必需的选项通常被认为是不适宜的，因为用户会预期 *options* 都是 可选的，因此在可能的情况下应当避免使用它们。

help

help 值是一个包含参数的简短描述的字符串。当用户请求帮助时（一般是通过在命令行中使用 -h 或 --help 的方式），这些 help 描述信息将随每个参数一同显示。

help 字符串可包括各种格式描述符以避免重复使用程序名称或参数*default* 等文本。有效的描述符包括程序名称 %(prog)s 和传给*add_argument()* 的大部分关键字参数，例如 %(default)s, %(type)s 等等:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

options:
 -h, --help  show this help message and exit
```

由于帮助字符串支持%-formatting，如果你希望在帮助字符串中显示 % 字面值，你必须将其转义为 %%。

argparse 支持静默特定选面的帮助条目，具体方式是将 help 值设为 argparse.SUPPRESS:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

options:
  -h, --help  show this help message and exit
```

metavar

当*ArgumentParser* 生成帮助消息时，它需要用某种方式来引用每个预期的参数。默认情况下，ArgumentParser 对象使用*dest* 值作为每个对象的“名字”。默认情况下，对于位置参数动作，*dest* 值将被直接使用，而对于可选参数动作，*dest* 值将被转为大写形式。因此，对于 dest='bar' 位置参数的引用形式将为 bar。带有单个命令行参数的可选参数 --foo 的引用形式将为 FOO。一个样例:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

options:
 -h, --help  show this help message and exit
 --foo FOO
```

可以使用 metavar 来指定一个替代名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

options:
 -h, --help  show this help message and exit
 --foo YYY
```

请注意 metavar 仅改变 显示的名称 - *parse_args()* 对象的属性名称仍然会由*dest* 值确定。

不同的 nargs 值可能导致 metavar 被多次使用。提供一个元组给 metavar 即为每个参数指定不同的显示信息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

options:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

大多数*ArgumentParser* 动作会添加一些值作为*parse_args()* 所返回对象的一个属性。该属性的名称由*add_argument()* 的 dest 关键字参数确定。对于位置参数动作，dest 通常会作为*add_argument()* 的第一个参数提供:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

对于可选参数动作，dest 的值通常取自选项字符串。*ArgumentParser* 会通过接受第一个长选项字符串并去掉开头的 -- 字符串来生成 dest 的值。如果没有提供长选项字符串，则 dest 将通过接受第一个短选项字符串并去掉开头的 - 字符来获得。任何内部的 - 字符都将被转换为 _ 字符以确保字符串是有效的属性名称。下面的例子显示了这种行为:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest 允许提供自定义属性名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

deprecated

在一个项目的生命期内，某些命令行参数可能需要被移除。在移除它们之前，你应当通知你的用户这些参数已被弃用并将被移除。*add_argument()* 的 deprecated 关键字参数指明参数已被弃用并将在未来被移除，其默认值为 False。对于每个参数，如果 deprecated 为 True，那么当该参数被使用时会打印一条警告到*sys.stderr*:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='snake.py')
>>> parser.add_argument('--legs', default=0, type=int, deprecated=True)
>>> parser.parse_args([])
Namespace(legs=0)
>>> parser.parse_args(['--legs', '4'])
snake.py: warning: option '--legs' is deprecated
Namespace(legs=4)
```

Added in version 3.13.

Action 类

Action 类实现了 Action API，是一个返回可调用对象的可调用对象，它返回的可调用对象将处理来自命令行的参数。任何遵循此 API 的对象均可作为 action 形参传给*add_argument()*。

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action 对象会被*ArgumentParser* 用来代表从命令行中的一个或多个字符串中解析出单个参数所必须的信息。Action 类必须接受两个位置参数加上传给*ArgumentParser.add_argument()* 的除了 action 本身的任何关键字参数。

Action 的实例(或作为 action 形参的任何可调用对象的返回值)应当定义 dest, option_strings, default, type, required, help 等属性。确定这些属性的最容易的方式是调用 Action.__init__()。

__call__(*parser*, *namespace*, *values*, *option_string=None*)

Action 的实例应当为可调用对象，因此所有子类都必须重写 __call__() 方法，它应当接受四个形参:

- *parser* - 包含此动作的*ArgumentParser* 对象。
- *namespace* - 将由*parse_args()* 返回的*Namespace* 对象。大多数动作会使用*setattr()* 为此对象添加一个属性。
- *values* - 关联的命令行参数，并提供要应用的类型转换。类型转换是由传给*add_argument()* 的*type* 关键字参数指定的。
- *option_string* - 被用于唤起此动作的选项字符串。option_string 参数是可选的，并且在动作被关联到位置参数时将被略去。

__call__() 方法可以执行任意动作，但通常将基于 dest 和 values 来设置 namespace 的属性。

format_usage()

Action 子类可以定义 format_usage() 方法，该方法不接受任何参数并返回一个将被用于打印程序的用法说明的字符串。如果未提供此方法，则将使用适当的默认值。

17.1.3 parse_args() 方法

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)

将参数字符串转换为对象并将其设为命名空间的属性。返回带有成员的命名空间。

之前对*add_argument()* 的调用决定了哪些对象会被创建以及它们如何被赋值。请参阅 add_argument() 的文档了解详情。

- *args* - 要解析的字符串列表。默认值是从*sys.argv* 获取。

- *namespace* - 用于获取属性的对象。默认值是一个新的空*Namespace* 对象。

选项值语法

parse_args() 方法支持多种指定选项值的方式（如果它接受选项的话）。在最简单的情况下，选项和它的值是作为两个单独参数传入的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

对于长选项（名称长度超过一个字符的选项），选项和值也可以作为单个命令行参数传入，使用 = 分隔它们即可:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

对于短选项（长度只有一个字符的选项），选项和它的值可以拼接在一起:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

有些短选项可以使用单个 - 前缀来进行合并，如果仅有最后一个选项（或没有任何选项）需要值的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

无效的参数

在解析命令行时，*parse_args()* 会检测多种错误，包括有歧义的选项、无效的类型、无效的选项、错误的位置参数个数等等。当遇到这种错误时，它将退出并打印出错误文本同时附带用法消息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

包含 - 的参数

parse_args() 方法会在用户明显出错时尝试给出错误信息，但某些情况本身就存在歧义。例如，命令行参数 `-1` 可能是尝试指定一个选项也可能是尝试提供一个位置参数。*parse_args()* 方法在此会谨慎行事：位置参数只有在它们看起来像负数并且解析器中没有任何选项看起来像负数时才能以 - 打头。：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # 没有负数选项，因此 -1 是位置参数
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # 没有负数选项，因此 -1 和 -5 是位置参数
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # 有负数选项，因此 -1 是选项
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # 有负数选项，因此 -2 是选项
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # 有负数选项，因此两个 -1 都是选项
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

如果你有必须以 - 打头的位置参数并且看起来不像负数，你可以插入伪参数 `'--'` 以告诉*parse_args()* 在那之后的内容是一个位置参数:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

另请参阅*针对有歧义参数的 argparse 指引* 来了解更多细节。

参数缩写（前缀匹配）

parse_args() 方法*在默认情况下* 允许将长选项缩写为前缀，如果缩写无歧义（即前缀与一个特定选项相匹配）的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

可产生一个以上选项的参数会引发错误。此特定可通过将*allow_abbrev* 设为 `False` 来禁用。

在 sys.argv 以外

有时在*sys.argv* 以外增加*ArgumentParser* 解析参数也很有用处。这可以通过将一个字符串列表传给*parse_args()* 来实现。它适用于在交互提示符下进行测试:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

命名空间对象

class argparse.**Namespace**

由*parse_args()* 默认使用的简单类，可创建一个存放属性的对象并将其返回。

这个类被有意做得很简单，只是一个具有可读字符串表示形式的*object*。如果你更喜欢类似字典的属性视图，你可以使用标准 Python 中惯常的*vars()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

另一个用处是让*ArgumentParser* 为一个已存在对象而不是为一个新的*Namespace* 对象的属性赋值。这可以通过指定 namespace= 关键字参数来实现:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

17.1.4 其它实用工具

子命令

ArgumentParser.**add_subparsers**(*[, *title*][, *description*][, *prog*][, *parser_class*][, *action*][, *dest*][, *required*][, *help*][, *metavar*])

许多程序都将其功能拆分为一系列子命令，例如，svn 程序可唤起像 svn checkout, svn update 和 svn commit 等子命令。当一个程序执行需要多组不同各类命令行参数时这种拆分功能的方式是一个非常好的主意。*ArgumentParser* 通过 add_subparsers() 方法支持创建这样的子命令。add_subparsers() 方法通常不带参数地调用并返回一个特殊的动作对象。该对象只有一个方法 add_parser()，它接受一个命令名称和任意多个 ArgumentParser 构造器参数，并返回一个可以通常方式进行修改的 ArgumentParser 对象。

形参的描述

- *title* - 帮助输出中子解析器分组的标题；如果提供了描述则默认为"subcommands"，否则使用位置参数的标题

- *description* - 帮助输出中对子解析器组的描述，默认为 None
- *prog* - 将与子命令帮助一同显示的用法信息，默认为程序名称和子解析器参数之前的任何位置参数
- *parser_class* - 将被用于创建子解析器实例的类，默认为当前解析器类 (例如*ArgumentParser*)
- *action* - 当此参数在命令行中出现时要执行动作的基本类型
- *dest* - 将被用于保存子命令名称的属性名；默认为 None 即不保存任何值
- *required* - 是否必须要提供子命令，默认为 False (在 3.7 中新增)
- *help* - 在输出帮助中的子解析器分组帮助信息，默认为 None
- *metavar* - 帮助信息中表示可用子命令的字符串；默认为 None 并以 {cmd1, cmd2, ..} 的形式表示子命令

一些使用示例:

```
>>> # 创建一个最高层级解析器
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='subcommand help')
>>>
>>> # 创建一个针对 "a" 命令的解析器
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # 创建一个针对 "b" 命令的解析器
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices=('X', 'Y', 'Z'), help='baz help')
>>>
>>> # 解析一些参数列表
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

请注意*parse_args()* 返回的对象将只包含主解析器和由命令行所选择的子解析器的属性（而没有任何其他子解析器）。因此在上面的例子中，当指定了 a 命令时，将只存在 foo 和 bar 属性，而当指定了 b 命令时，则只存在 foo 和 baz 属性。

类似地，当一个子解析器请求帮助消息时，只有该特定解析器的帮助消息会被打印出来。帮助消息将不包括父解析器或同级解析器的消息。（每个子解析器命令一条帮助消息，但是，也可以像上面那样通过将 help= 参数传入 add_parser() 来给出。）

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   subcommand help
    a     a help
    b     b help

options:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help
```

(续下页)

(接上页)

```
options:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

options:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

add_subparsers() 方法也支持 `title` 和 `description` 关键字参数。当两者都存在时，子解析器的命令将出现在输出帮助消息中它们自己的分组内。例如:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

options:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

此外，`add_parser()` 还支持附加的 *aliases* 参数，它允许多个字符串指向同一个子解析器。下面的例子，类似于 `svn`，将别名 `co` 设为 `checkout` 的缩写形式:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

`add_parser()` 也支持附加的 *deprecated* 参数，它允许弃用子解析器。

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='chicken.py')
>>> subparsers = parser.add_subparsers()
>>> run = subparsers.add_parser('run')
>>> fly = subparsers.add_parser('fly', deprecated=True)
>>> parser.parse_args(['fly'])
chicken.py: warning: command 'fly' is deprecated
Namespace()
```

Added in version 3.13.

一个特别有效的处理子命令的方式是将*add_subparsers()* 方法与对*set_defaults()* 的调用结合起来使用以便每个子解析器能知道应当执行哪个 Python 函数。例如:

```
>>> # 子命令函数
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
```

(续下页)

(接上页)

```
...     print('((%s))' % args.z)
...
>>> # 创建最高层级解析器
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(required=True)
>>>
>>> # 创建针对 "foo" 命令的解析器
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # 创建针对 "bar" 命令的解析器
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # 解析参数并调用被选定的任何函数
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # 解析参数并调用被选定的任何函数
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

通过这种方式，你可以在参数解析结束后让*parse_args()* 执行调用适当函数的任务。像这样将函数关联到动作通常是你处理每个子解析器的不同动作的最简便方式。但是，如果有必要检查被唤起的子解析器的名称，则*add_subparsers()* 调用的 `dest` 关键字参数将可实现:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

在 3.7 版本发生变更: 新增 *required* 仅限关键字形参。

FileType 对象

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

FileType 工厂类用于创建可作为*ArgumentParser.add_argument()* 的 type 参数传入的对象。以*FileType* 对象作为其类型的参数将使用命令行参数以所请求模式、缓冲区大小、编码格式和错误处理方式打开文件（请参阅*open()* 函数了解详情）:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=<_io.
↪FileIO name='raw.dat' mode='wb'>)
```

FileType 对象能理解伪参数 `'-'` 并会自动将其转换为*sys.stdin* 用于可读的*FileType* 对象以及*sys.stdout* 用于可写的*FileType* 对象:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

在 3.4 版本发生变更: 增加了 *encodings* 和 *errors* 形参。

参数组

ArgumentParser.**add_argument_group**(*title=None*, *description=None*, ***[, *argument_default*][, *conflict_handler*])

在默认情况下，*ArgumentParser* 会在显示帮助消息时将命令行参数分为“位置参数”和“选项”两组。当存在比默认更好的分组概念时，可以使用 add_argument_group() 方法来创建适当的分组:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

add_argument_group() 方法返回一个具有*add_argument()* 方法的参数分组对象，这与常规的*ArgumentParser* 一样。当一个参数被加入分组时，解析器儒将地它视为一个正常的参数，但是会在单独的帮助消息分组中显示该参数。add_argument_group() 方法接受 *title* 和 *description* 参数以使用它们来定制显示内容:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

可选的仅限关键字形参*argument_default* 和*conflict_handler* 允许对参数分组行为进行更细粒度的控制。这些形参具有与*ArgumentParser* 构造器中相同的含义，但是专门应用于参数分组而不是整个解析器。

请注意任意不在你的自定义分组中的参数最终都将回到通常的“位置参数”和“可选参数”分组中。

在 3.11 版本发生变更: 在参数分组上调用*add_argument_group()* 已被弃用。此特性从未受到支持并且不保证总能正确工作。此函数出现在 API 中是继承导致的意外并将在未来被移除。

互斥

ArgumentParser.**add_mutually_exclusive_group**(*required=False*)

创建一个互斥分组。argparse 将会确保互斥分组中只有一个参数在命令行中可用:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

add_mutually_exclusive_group() 方法也接受一个 *required* 参数，表示在互斥组中至少有一个参数是需要的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

请注意当前互斥的参数组不支持*add_argument_group()* 的 *title* 和 *description* 参数。但是，互斥的参数数可以被添加到具有 title 和 description 的参数组中。例如:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_argument_group('Group title', 'Group description')
>>> exclusive_group = group.add_mutually_exclusive_group(required=True)
>>> exclusive_group.add_argument('--foo', help='foo help')
>>> exclusive_group.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [-h] (--foo FOO | --bar BAR)

options:
  -h, --help  show this help message and exit

Group title:
  Group description

  --foo FOO   foo help
  --bar BAR   bar help
```

在 3.11 版本发生变更: 在互斥的分组上调用*add_argument_group()* 或*add_mutually_exclusive_group()* 已被弃用。这些特性从未受到支持并且不保证总能正确工作。这些函数出现在 API 中是继承导致的意外并将在未来被移除。

解析器默认值

ArgumentParser.**set_defaults**(***kwargs*)

在大多数时候，*parse_args()* 所返回对象的属性将完全通过检查命令行参数和参数动作来确定。*set_defaults()* 则允许加入一些无须任何命令行检查的额外属性:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
```

(续下页)

(接上页)

```
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

请注意解析器层级的默认值总是会覆盖参数层级的默认值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

解析器层级默认值在需要多解析器时会特别有用。请参阅*add_subparsers()* 方法了解此类型的一个示例。

ArgumentParser.**get_default**(*dest*)

获取一个命名空间属性的默认值，该值是由*add_argument()* 或*set_defaults()* 设置的:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

打印帮助

在大多数典型应用中，*parse_args()* 将负责任何用法和错误消息的格式化和打印。但是，也可使用某些其他格式化方法:

ArgumentParser.**print_usage**(*file=None*)

打印一段简短描述，说明应当如何在命令行中唤起*ArgumentParser*。如果 *file* 为 None，则默认使用*sys.stdout*。

ArgumentParser.**print_help**(*file=None*)

打印一条帮助消息，包括程序用法和通过*ArgumentParser* 注册的相关参数信息。如果 *file* 为 None，则默认使用*sys.stdout*。

还存在这些方法的几个变化形式，它们只返回字符串而不打印消息:

ArgumentParser.**format_usage**()

返回一个包含简短描述的字符串，说明应当如何在命令行中唤起*ArgumentParser*。

ArgumentParser.**format_help**()

反回一个包含帮助消息的字符串，包括程序用法和通过*ArgumentParser* 注册的相关参数信息。

部分解析

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

有时一个脚本可能只解析部分命令行参数，而将其余的参数继续传递给另一个脚本或程序。在这种情况下，*parse_known_args()* 方法会很有用处。它的作用方式很类似*parse_args()* 但区别在于当存在额外参数时它不会产生错误。而是会返回一个由两个条目构成的元组，其中包含带成员的命名空间和剩余参数字符串的列表。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

> **警告**
>
> 前缀匹配 规则应用于*parse_known_args()*。一个解析器即使在某个选项只是已知选项的前缀时也能读取该选项，而不是将其放入剩余参数列表。

自定义文件解析

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)

从文件读取的参数（见*ArgumentParser* 的 *fromfile_prefix_chars* 关键字参数）将是一行读取一个参数。*convert_arg_line_to_args()* 可被重写以使用更复杂的读取方式。

此方法接受从参数文件读取的字符串形式的单个参数 *arg_line*。它返回从该字符串解析出的参数列表。此方法将在每次按顺序从参数文件读取一行时被调用一次。

此方法的一个有用的重写是将每个以空格分隔的单词视为一个参数。下面的例子演示了如何实现这一点:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

退出方法

ArgumentParser.**exit**(*status=0*, *message=None*)

此方法将终结程序，退出时附带指定的 *status*，并且如果给出了 *message* 则会在退出前将其打印到*sys.stderr*。用户可重写此方法以不同方式来处理这些步骤:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

ArgumentParser.**error**(*message*)

此方法将把用法消息包括 *message* 打印到*sys.stderr* 并附带状态码 2 终结程序。

混合解析

ArgumentParser.**parse_intermixed_args**(*args=None*, *namespace=None*)

ArgumentParser.**parse_known_intermixed_args**(*args=None*, *namespace=None*)

许多 Unix 命令允许用户混用可选参数与位置参数。*parse_intermixed_args()* 和*parse_known_intermixed_args()* 方法均支持这种解析风格。

这些解析器并不支持所有的 argparse 特性，并且当不受支持的特性被使用时将会引发异常。具体来说，子解析器以及同时包括可选参数和位置参数的互斥分组是不受支持的。

下面的例子显示了*parse_known_args()* 与*parse_intermixed_args()* 之间的差异：前者会将 `['2', '3']` 返回为未解析的参数，而后者会将所有位置参数收集至 rest 中。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

parse_known_intermixed_args() 返回由两个条目组成的元组，其中包含带成员的命名空间以及剩余参数字符串列表。当存在任何剩余的未解析参数字符串时*parse_intermixed_args()* 将引发一个错误。

Added in version 3.7.

注册自定义的类型或动作

ArgumentParser.**register**(*registry_name*, *value*, *object*)

在某些时候有必要在错误消息中使用自定义的字符串以提供对用户更友好的输出。在这种情况下，可以使用 `register()` 来注册带有解析器的自定义动作或类型并允许你按其注册名称而非其可调用对象名称来引用该类型。

`register()` 方法接受三个参数——*registry_name*，指定将要存储对象的内部注册表 (例如 `action`, `type`)，*value*，将要注册对象对应的键，以及 object，将要注册的可调用对象。

下面的例子演示了如何注册一个带解析器的自定义类型:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.register('type', 'hexadecimal integer', lambda s: int(s, 16))
>>> parser.add_argument('--foo', type='hexadecimal integer')
_StoreAction(option_strings=['--foo'], dest='foo', nargs=None, const=None, default=None,
↪ type='hexadecimal integer', choices=None, required=False, help=None, metavar=None,␣
↪deprecated=False)
>>> parser.parse_args(['--foo', '0xFA'])
Namespace(foo=250)
>>> parser.parse_args(['--foo', '1.2'])
usage: PROG [-h] [--foo FOO]
PROG: error: argument --foo: invalid 'hexadecimal integer' value: '1.2'
```

17.1.5 异常

exception argparse.**ArgumentError**

来自创建或使用某个参数的消息（可选或位置参数）。

此异常的字符串值即异常消息，并附带有导致该异常的参数的相关信息。

exception argparse.**ArgumentTypeError**

当从一个命令行字符串到特定类型的转换出现问题时引发。

指南与教程

argparse 教程

作者
Tshepang Mbambo

这篇教程旨在作为*argparse* 的入门介绍，此模块是 Python 标准库中推荐的命令行解析模块。

备注

标准库还包括另两个与命令行形参处理直接相关的库：低层级的*optparse* 模块 (对于特定应用程序它可能需要更多的代码来配置，但也允许应用程序请求 `argparse` 所不支持的行为)，以及更低层级的*getopt* (它被作为供 C 程序员使用的 `getopt()` 函数族的等价物)。这些模块并未在本指南中直接介绍，`argparse` 中的许多核心概念最初都是来自 `optparse`，因此本教程的某些部分对 `optparse` 用户来说也是有用的。

概念

让我们利用 **ls** 命令来展示我们将要在这篇入门教程中探索的功能：

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

我们可以从这四个命令中学到几个概念：

- **ls** 是一个即使在运行的时候没有提供任何选项，也非常有用的命令。在默认情况下他会输出当前文件夹包含的文件和文件夹。
- 如果我们想要使用比它默认提供的更多功能，我们需要告诉该命令更多信息。在这个例子里，我们想要查看一个不同的目录，pypy。我们所做的是指定所谓的位置参数。之所以这样命名，是因为程序应该如何处理该参数值，完全取决于它在命令行出现的位置。更能体现这个概念的命令如 **cp**，它最基本的用法是 `cp SRC DEST`。第一个位置参数指的是 * 你想要复制的 *，第二个位置参数指的是 * 你想要复制到的位置 *。
- 现在假设我们想要改变这个程序的行为。在我们的例子中，我们不仅仅只是输出每个文件的文件名，还输出了更多信息。在这个例子中，`-l` 被称为可选参数。
- 这是一段帮助文档的文字。它是非常有用的，因为当你遇到一个你从未使用过的程序时，你可以通过阅读它的帮助文档来弄清楚它是如何运行的。

基础

让我们从一个简单到（几乎）什么也做不了的例子开始：

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

以下是该代码的运行结果：

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

程序运行情况如下：

- 在没有任何选项的情况下运行脚本不会在标准输出显示任何内容。这没有什么用处。

- 第二行代码开始展现出*argparse*模块的作用。我们几乎什么也没有做，但已经得到一条很好的帮助信息。
- `--help`选项，也可缩写为`-h`，是唯一一个可以直接使用的选项（即不需要指定该选项的内容）。指定任何内容都会导致错误。即便如此，我们也能直接得到一条有用的用法信息。

位置参数介绍

举个例子：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

运行此程序：

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

options:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

程序运行情况如下：

- 我们增加了*add_argument()*方法，该方法用于指定程序将能接受哪些命令行选项。在这个例子中，我将它命名为`echo`以与其对应的函数保持一致。
- 现在调用我们的程序必须要指定一个选项。
- *parse_args()*方法实际将返回来自指定选项的某些数据，在这个例子中是`echo`。
- 这一变量是*argparse*免费施放的某种“魔法”（即是说，不需要指定哪个变量是存储哪个值的）。你也可以注意到，这一名称与传递给方法的字符串参数一致，都是`echo`。

然而请注意，尽管显示的帮助看起来清楚完整，但它可以比现在更有帮助。比如我们可以知道`echo`是一个位置参数，但我们除了靠猜或者看源代码，没法知道它是用来干什么的。所以，我们可以把它改造得更有用：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

然后我们得到：

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

options:
  -h, --help  show this help message and exit
```

现在，来做一些更有用的事情：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print(args.square**2)
```

以下是该代码的运行结果：

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

进展不太顺利。那是因为*argparse* 会把我们传递给它的选项视作为字符串，除非我们告诉它别这样。所以，让我们来告诉*argparse* 来把这一输入视为整数：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

以下是该代码的运行结果：

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

做得不错。当这个程序在收到错误的无效的输入时，它甚至能在执行计算之前先退出，还能显示很有帮助的错误信息。

可选参数介绍

到目前为止，我们一直在研究位置参数。让我们看看如何添加可选的：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

和输出：

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

options:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
```

(续下页)

(接上页)

```
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

程序运行情况如下：

- 这一程序被设计为当指定 `--verbosity` 选项时显示某些东西，否则不显示。
- 为表明此选项确实是可选的，当不附带该选项运行程序时将不会提示任何错误。请注意在默认情况下，如果一个可选参数未被使用，则关联的变量，在这个例子中是 `args.verbosity`，将被赋值为 `None`，这也就是它在 `if` 语句中无法通过真值检测的原因。
- 帮助信息有点不同。
- 使用 `--verbosity` 选项时，必须指定一个值，但可以是任何值。

上述例子接受任何整数值作为 `--verbosity` 的参数，但对于我们的简单程序而言，只有两个值有实际意义：`True` 或者 `False`。让我们据此修改代码：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

和输出：

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

程序运行情况如下：

- 现在此选项更像是一个旗标而不需要接受特定的值。我们甚至改变了此选项的名字来匹配这一点。请注意我们现在指定了一个新的关键词 `action`，并将其赋值为 `"store_true"`。这意味着，如果指定了该选项，则将值 `True` 赋给 `args.verbose`。如未指定则表示其值为 `False`。
- 当你为其指定一个值时，它会报错，符合作为标志的真正的精神。
- 留意不同的帮助文字。

短选项

如果你熟悉命令行的用法，你会发现我还没讲到这一选项的短版本。这也很简单：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

效果就像这样：

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]

options:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

可以注意到，这一新的能力也反映在帮助文本里。

结合位置参数和可选参数

我们的程序变得越来越复杂了：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print(f"the square of {args.square} equals {answer}")
else:
    print(answer)
```

接着是输出：

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- 我们带回了一个位置参数，结果发生了报错。
- 注意顺序无关紧要。

给我们的程序加上接受多个冗长度的值，然后实际来用用：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

和输出：

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

除了最后一个，看上去都不错。最后一个暴露了我们的程序中有一个 bug。我们可以通过限制 `--verbosity` 选项可以接受的值来修复它：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

和输出：

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity {0,1,2}
                        increase output verbosity
```

注意这一改变同时反应在错误信息和帮助信息里。

现在，让我们使用另一种的方式来改变冗长度。这种方式更常见，也和 CPython 的可执行文件处理它自己的冗长度参数的方式一致（参考 `python --help` 的输出）：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
```

(续下页)

(接上页)

```
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

我们引入了另一种动作"count"，来统计特定选项出现的次数。

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

options:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- 是的，它现在比前一版本更像是一个标志（和 `action="store_true"` 相似）。这能解释它为什么报错。
- 它也表现得与“store_true”的行为相似。
- 这给出了一个关于 `count` 动作的效果的演示。你之前很可能应该已经看过这种用法。
- 如果你不添加 `-v` 标志，这一标志的值会是 `None`。
- 如期望的那样，添加该标志的长形态能够获得相同的输出。
- 可惜的是，对于我们的脚本获得的新能力，我们的帮助输出并没有提供很多信息，但我们总是可以通过改善文档来修复这一问题（比如通过 `help` 关键字参数）。
- 最后一个输出暴露了我们程序中的一个 bug。

让我们修复一下：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

这是它给我们的输出：

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '>=' not supported between instances of 'NoneType' and 'int'
```

- 第一组输出很好，修复了之前的 bug。也就是说，我们希望任何 >= 2 的值尽可能详尽。
- 第三组输出并不理想。

让我们修复那个 bug：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

我们刚刚引入了又一个新的关键字 `default`。我们把它设置为 `0` 来让它可以与其他整数值相互比较。记住，默认情况下如果一个可选参数没有被指定，它的值会是 `None`，并且它不能和整数值相比较（所以产生了 *TypeError* 异常）。

然后：

```
$ python prog.py 4
16
```

凭借我们目前已学的东西你就可以做到许多事情，而我们还仅仅学了一些皮毛而已。*argparse* 模块是非常强大的，在结束篇教程之前我们将再探索更多一些内容。

进行一些小小的改进

如果我们想扩展我们的简短程序来执行其他幂次的运算，而不仅是乘方：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"{args.x} to the power {args.y} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.x}^{args.y} == {answer}")
else:
    print(answer)
```

输出：

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent

options:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

请注意到目前为止我们一直在使用详细级别来 *更改*所显示的文本。以下示例则使用详细级别来显示 *更多*的文本:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"Running '{__file__}'")
if args.verbosity >= 1:
    print(f"{args.x}^{args.y} == ", end="")
print(answer)
```

输出:

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

指定有歧义的参数

当在确定一个参数是位置参数还是从属于另一个参数存在歧义时，可以使用 `--` 来告诉*parse_args()* 在它之后的参数是位置参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-n', nargs='+')
>>> parser.add_argument('args', nargs='*')

>>> # ambiguous, so parse_args assumes it's an option
>>> parser.parse_args(['-f'])
usage: PROG [-h] [-n N [N ...]] [args ...]
PROG: error: unrecognized arguments: -f

>>> parser.parse_args(['--', '-f'])
Namespace(args=['-f'], n=None)

>>> # ambiguous, so the -n option greedily accepts arguments
>>> parser.parse_args(['-n', '1', '2', '3'])
```

(续下页)

(接上页)

```
Namespace(args=[], n=['1', '2', '3'])

>>> parser.parse_args(['-n', '1', '--', '2', '3'])
Namespace(args=['2', '3'], n=['1'])
```

矛盾的选项

到目前为止，我们一直在使用*argparse.ArgumentParser* 实例的两个方法。让我们再介绍第三个方法*add_mutually_exclusive_group()*。它允许我们指定彼此相冲突的选项。让我们再修改程序的其余部分以便使新功能更有意义：我们将引入 `--quiet` 选项，它将与 `--verbose` 的作用相反：

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

我们的程序现在变得更简洁了，我们出于演示需要略去了一些功能。无论如何，输出是这样的：

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

这应该很容易理解。我添加了末尾的输出这样你就可以看到其所达到的灵活性，即混合使用长和短两种形式的选项。

在我们收尾之前，你也许希望告诉你的用户这个程序的主要目标，以免他们还不清楚：

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y
```

(续下页)

(接上页)

```
if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

请注意用法文本中有细微的差异。注意 `[-v | -q]`，它的意思是说我们可以使用 `-v` 或 `-q`，但不能同时使用两者：

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

如何翻译 argparse 的输出

argparse 模块的输出例如它的帮助文本和错误消息都可以通过*gettext* 模块实现翻译。这允许应用程序轻松本地化*argparse* 所产生的消息。另请参见国际化 *(I18N)* 你的程序和模块。

例如，在这个*argparse* 输出中：

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

字符串 `usage:`, `positional arguments:`, `options:` 和 `show this help message and exit` 都是可翻译的。

要翻译这些字符串，必须先将它们提取到一个 `.po` 文件中。例如，使用 Babel，运行这条命令：

```
$ pybabel extract -o messages.po /usr/lib/python3.12/argparse.py
```

此命令将从*argparse* 模块提取所有可翻译的字符串，并将其输出到名为 `messages.po` 的文件中。此命令假定你的 Python 安装位置为 `/usr/lib`。

你可以使用以下脚本查找*argparse* 模块在系统中的位置：

```
import argparse
print(argparse.__file__)
```

一旦 `.po` 文件中的文本信息翻译完毕并使用*gettext* 安装了译文，*argparse* 将能显示翻译后的信息。

要翻译在*argparse* 输出中的字符串，请使用*gettext*。

自定义类型转换器

argparse 模块允许您为命令行参数指定自定义类型转换器。这使您能够在用户输入存储在*argparse.Namespace* 中之前对其进行修改。当您需要在程序中使用输入之前对其进行预处理时，这会很有用。

使用自定义类型转换器时，您可以使用任何可调用对象，该对象接受单个字符串参数（参数值）并返回转换后的值。但是，如果需要处理更复杂的情况，可以使用带有 **action** 形参的自定义动作类。

例如，假设您希望处理带有不同前缀的参数并相应地进行处理:

```
import argparse

parser = argparse.ArgumentParser(prefix_chars='-+')

parser.add_argument('-a', metavar='<value>', action='append',
                    type=lambda x: ('-', x))
parser.add_argument('+a', metavar='<value>', action='append',
                    type=lambda x: ('+', x))

args = parser.parse_args()
print(args)
```

输出:

```
$ python prog.py -a value1 +a value2
Namespace(a=[('-', 'value1'), ('+', 'value2')])
```

在这个例子中，我们:

- 使用 “prefix_chars“ 形参创建了带有自定义前缀字符的解析器。
- 定义了两个参数,“-a“ 和 “+a“, 它们使用 “type“ 形参创建自定义类型转换器，以便将值存储在带有前缀的元组中。

如果没有自定义类型转换器，参数会将 “-a“ 和 “+a“ 视为同一个参数，这是不可取的。通过使用自定义类型转换器，我们能够区分这两个参数。

后记

除了这里显示的内容，*argparse* 模块还提供了更多功能。它的文档相当详细和完整，包含大量示例。完成这个教程之后，你应该能毫不困难地阅读该文档。

Migrating `optparse` code to `argparse`

The *argparse* module offers several higher level features not natively provided by the *optparse* module, including:

- Handling positional arguments.
- Supporting subcommands.
- Allowing alternative option prefixes like `+` and `/`.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

Originally, the *argparse* module attempted to maintain compatibility with *optparse*. However, the fundamental design differences between supporting declarative command line option processing (while leaving positional argument processing to application code), and supporting both named options and positional arguments in the declarative interface mean that the API has diverged from that of `optparse` over time.

As described in 选择参数解析库, applications that are currently using *optparse* and are happy with the way it works can just continue to use `optparse`.

Application developers that are considering migrating should also review the list of intrinsic behavioural differences described in that section before deciding whether or not migration is desirable.

For applications that do choose to migrate from *optparse* to *argparse*, the following suggestions should be helpful:

- Replace all *optparse.OptionParser.add_option()* calls with *ArgumentParser.add_argument()* calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional *ArgumentParser.add_argument()* calls for the positional arguments. Keep in mind that what was previously called `options`, now in the *argparse* context is called `args`.
- Replace *optparse.OptionParser.disable_interspersed_args()* by using *parse_intermixed_args()* instead of *parse_args()*.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace *optparse.Values* with *Namespace* and *optparse.OptionError* and *optparse.OptionValueError* with *ArgumentError*.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.

17.2 optparse --- 命令行选项的解析器

源代码： Lib/optparse.py

17.2.1 选择参数解析库

标准库包括三个参数解析库：

- *getopt*: 一个忠实地复刻过程式 C `getopt` API 的模块。在最初的 Python 1.0 发布版之前就已包括在标准库中。
- *optparse*: 一个 `getopt` 的声明式替代物，提供了等价的功能而不要求每个应用程序实现它自己的过程式选项解析逻辑。自 Python 2.3 发布版起即包括在标准库中。
- *argparse*: 一个更有针对性的 `optparse` 替代物，默认提供了更多功能，代价是降低了应用程序在具体如何控制参数处理时的灵活性。自 Python 2.7 和 Python 3.2 发布版起被包括在标准库中。

在缺少更具体的参数解析设计约束的情况下，*argparse* 是实现命令行应用程序的推荐选择，因为它能以最少的程用程序层级代码提供最高层级的基本功能。

getopt 被保留几乎完全是出于向下兼容性理由。不过，它也适合在基于 `getopt` 的 C 应用程序中作为命令行参数处理原型搭建和测试场景中的工具。

optparse 应当在下列场景中被作为*argparse* 的替代物：

- 一个应用程序已在使用*optparse* 并且不想再冒当迁移至*argparse* 时可能引发的微妙行为改变的风险。
- 这个应用程序需要对命令行中选项和位置形参的交错方式进行额外的控制（包括完全禁用交错特性的能力）
- 这个应用程序需要对命令行元素进行渐进式解析的过程施加更多控制（虽然 `argparse` 确实支持这个功能，但实际运作的方式在某些情况下并非理想）

- 这个应用程序需要对某些可能接受以 - 开头的参数值的命令行选项施行特殊的控制（例如程序本身调用另一个程序，而这些选项是要原样传给被调用的子程序的）
- 这个应用程序需要其他一些 argparse 不支持，但是可利用 optparse 提供之底层接口实现的特殊的命令行参数处理行为

这些考量也意味着*optparse* 或可为代码库编写者撰写第三方命令行参数解析库时提供更好的基础。

以下提供一个具体的例子。请见下方两段命令行参数解析代码，其中第一段使用 optparse 实现；第二段使用 argparse 实现：

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

最明显的差异就是在 optparse 版本中，无选项参数是在选项处理完成后由应用程序单独处理的。在 argparse 版本中，位置参数则是以与带名称的选项相同的方式声明和处理的。

不过，argparse 版本还会以与 optparse 版本不同的方式处理某些形参组合。例如（除其他差异之外）：

- 当使用 optparse 时 `-o -v` 是指定 `output="-v"` 和 `verbose=False`，但对于 argparse 则为错误用法（将报告没有为 `-o/--output` 提供值，因为 `-v` 会被解读为代表 verbosity 旗标）
- 类似地，当使用 optparse 时提供 `-o --` 是指定 `output="--"` 和 `args=()`，但对于 argparse 则为错误用法（同样将报告没有为 `-o/--output` 提供值，因为 `--` 会被解读为结束选项处理并将所有剩余的值视为位置参数）
- 当使用 optparse 时提供 `-o=foo` 是指定 `output="=foo"`，但对于 argparse 则为 `output="foo"` (因为 = 是被作为选项形参值的替代分隔符的特例)。

这些在 argparse 版本中存在差异的行为是有必要的还是有问题的将取决于特定命令行应用程序的使用场景。

参见

click 是一个第三方参数处理库（最初是以 optparse 为基础），它允许命令行应用程序以一组被装饰的命令实现函数的方式来开发。

其他的第三方库，例如 typer 或 msgspec-click，允许命令行界面以更有效率地集成 Python 类型标注静态检查的方式来指定。

17.2.2 概述

optparse 是一个相比极简风格的*getopt* 模块更为方便、灵活和强大的命令行选项解析库。*optparse* 使用更趋声明式的命令行解析风格：创建一个*OptionParser* 的实例，使用选项来填充它，然后解析命令行。*optparse* 允许用户以传统的 GNU/POSIX 语法来指定选项，并会为你生成额外的用法和帮助消息。

下面是在一个简单脚本中使用*optparse* 的示例：

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

通过这几行代码，你的脚本的用户可以在命令行上完成“常见任务”，例如:

```
<yourscript> --file=outfile -q
```

在它解析命令行时，*optparse* 会根据用户提供的命令行值设置*parse_args()* 所返回的 options 对象的属性。当*parse_args()* 从解析此命令行返回时，options.filename 将为 "outfile" 而 options.verbose 将为 False。*optparse* 支持长短两种形式的选项，允许多个短选项合并到一起，并允许选项以多种方式与其参数相关联。因此，以下命令行均等价于以上示例:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

此外，用户还可以运行以下命令之一

```
<yourscript> -h
<yourscript> --help
```

这样*optparse* 将打印出你的脚本的选项概要:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

其中 *yourscript* 的值是在运行时确定的 (通常来自 sys.argv[0])。

17.2.3 背景

optparse 被显式地设计为鼓励创建带有遵循 C 开发者所用的 getopt() 函数族的惯例的简明直观的命令行接口的程序。为了这个目标，它仅支持最常见的命令行语法和在 Unix 下使用的规范语义。如果你不熟悉这些惯例，阅读本章节将使你开始了解它们。

术语

argument -- 参数
: 在命令行中输入的字符串，并会被 shell 传给 execl() 或 execv()。在 Python 中，参数将是 sys.argv[1:] 的元素 (sys.argv[0] 是被执行的程序的名称)。Unix shell 也使用术语”word” 来指代参数。

  有时替换 sys.argv[1:] 以外的参数列表也是必要的，所以你应当将” 参数” 当作是”sys.argv[1:] 的一个元素，或者是作为 sys.argv[1:] 的替代的其他列表”。

选项
: 一个用来提供额外信息以指导或定制程序的执行的参数。对于选项有许多不同的语法；传统的 Unix 语法是一个连字符 (”-”) 后面跟单个字母，例如 -x 或 -F。此外，传统的 Unix 语法允许将多个选项合并为一个参数，例如 -x -F 就等价于 -xF。GNU 项目引入了 -- 后面跟一串以连字符分隔的单词，例如 --file 或 --dry-run。它们是*optparse* 所提供的仅有的两种选项语法。

存在于世上的其他一些选项语法包括:

- 一个连字符后面跟几个字母，例如 `-pf` (这与多个选项合并成单个参数 并不一样)
- 一个连字符后面跟一个完整单词，例如 `-file` (这在技术上等同于前面的语法，但它们通常不在同一个程序中出现)
- 一个加号后面跟一个字母，或几个字母，或一个单词，例如 `+f`, `+rgb`
- 一个斜杠后面跟一个字母，或几个字母，或一个单词，例如 `/f`, `/file`

这些选项语法都不被*optparse*所支持，也永远不会支持。这是有意为之的：前三种在任何环境下都是非标准的，而最后一种只在你专门针对 Windows 或某些旧平台（例如 VMS, MS-DOS）时才有意义。

可选参数:
一个跟在某个选项之后的参数，与该选项紧密相关，并会在该选项被消耗时从参数列表中被消耗。使用*optparse*，选项参数可以是其对应选项以外的一个单独参数:

```
-f foo
--file foo
```

或是包括在同一个参数中:

```
-ffoo
--file=foo
```

通常，一个给定的选项将接受一个参数或是不接受。许多人想要“可选的可选参数”特性，即某些选项将在看到特定参数时接受它，而如果没看到特定参数则不接受。在某种程度上说这一特性是存在争议的，因它它将使解析发生歧义：如果 `-a` 接受一个可选参数而 `-b` 是完全不同的另一个选项，那我们该如何解读 `-ab` 呢？由于这会存在歧义，因此*optparse*不支持这一特性。

positional argument -- 位置参数
在解析选项之后，即在选项及其参数解析完成并从参数列表中移除后参数列表中余下的内容。

必选选项
必须在命令行中提供的选项；请注意在英文中”required option” 这个短语是自相矛盾的。*optparse*不会阻止你实现必须选项，但也不会在这方面给你什么帮助。

例如，考虑这个假设的命令行:

```
prog -v --report report.txt foo bar
```

`-v` 和 `--report` 都是选项。假定 `--report` 接受一个参数，`report.txt` 是一个选项参数。`foo` 和 `bar` 是位置参数。

选项的作用是什么？

选项被用来提供额外信息以便微调或定制程序的执行。需要明确的一点是，选项通常都是 可选的。一个程序应当能在没有设置任何选项的的情况下正常运行。(从 Unix 或 GNU 工具集中随机挑选一个程序。它是否能在未设置任何选项的情况下运行并且仍然得到有意义的结果？主要的例外有 `find`, `tar` 和 `dd` ---它们都是些因为语法不标准和界面混乱而受到公正抨击的变异奇行种。)

有很多人希望他们的程序具有“必需选项”。请再思考一下。如果某个项是必需的，那么它就 不是可选的！如果你的程序必需要有某项信息才能成功运行，则它更适合作为位置参数。

作为良好的命令行界面设计的一个例子，请看基本的用于拷贝文件的 `cp` 工具。试图拷贝文件而不提供一个目标和至少一个源是没有什么意义的。因此，如果你不带参数地运行 `cp` 它将会报错。不过，它具有一个完全不需要任何选项的灵活、易用的语法:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

你只使用这个语法就能畅行无阻。大多数 cp 实现还提供了许多精确调整文件拷贝方式的选项：你可以保留模式和修改时间，避免跟随符号链接，覆盖现有文件之前先询问，诸如此类。但这些都不会破坏 cp 的核心任务，即将一个文件拷贝为另一个文件，或将多个文件拷贝到另一个目录。

位置参数有什么用？

位置参数是对于你的程序运行来说绝对、肯定需要的信息片段。

一个好的用户界面应当尽可能少地设置绝对必需提供的信息。如果你的程序必需提供 17 项不同的信息片段才能成功运行，那么你要 *如何* 从用户获取这些信息将不是问题的关键 --- 大多数人会在他们成功运行此程序之前放弃并离开。无论用户界面是命令行、配置文件还是 GUI 都一样适用：如果你对你的用户提出如此多的要求，它们大多将会直接放弃。

简而言之，请尽量最小化绝对要求用户提供的信息量 --- 只要有可能就使用合理的默认值。当然，你希望程序足够灵活也是合理的。这就是选项的作用。同样，选项是配置文件中的条目，GUI 中的“首选项”对话框中的控件，还是命令行选项不是问题的关键 --- 你实现的选项越多，你的程序就越灵活，它的具体实现也会变得更为复杂。当然，太大的灵活性也存在缺点；过多的选项会让用户更难掌握并使你的代码更难维护。

17.2.4 教程

虽然*optparse* 非常灵活和强大，但在大多数情况下它也很简明易用。本小节介绍了任何基于*optparse* 的程序中常见的代码模式。

首先，你需要导入 OptionParser 类；然后在主程序的开头部分，创建一个 OptionParser 实例：

```
from optparse import OptionParser
...
parser = OptionParser()
```

然后你可以开始定义选项。基本语法如下：

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

每个选项有一个或多个选项字符串，如 -f 或 --file，以及一些选项属性用来告诉*optparse* 当它在命令行中遇到该选项时将得到什么和需要做什么。

通常，每个选项都会有一个短选项字符串和一个长选项字符串，例如

```
parser.add_option("-f", "--file", ...)
```

你可以随你的喜好自由定义任意数量的短选项字符串和任意数量的长选项字符串（包括零个），只要总计至少有一个选项字符串。

传给*OptionParser.add_option()* 的选项字符串实际上是特定调用所定义的选项的标签。为了表述简单，我们将经常会说在命令行中 *遇到一个选项*；而实际上，*optparse* 是遇到了 *选项字符串* 并根据它们来查找选项。

一旦你定义好所有的选项，即可指令*optparse* 来解析你的程序的命令行：

```
(options, args) = parser.parse_args()
```

（如果你愿意，可以将自定义的参数列表传给*parse_args()*，但很少有必要这样做：默认它将使用 sys.argv[1:]。）

parse_args() 返回两个值：

- options，一个包含你所有的选项的值的对象 --- 举例来说，如果 --file 接受一个字符串参数，则 options.file 将为用户所提供的文件名，或者如果用户未提供该选项则为 None
- args，由解析选项之后余下的位置参数组成的列表

本教学章节只介绍了四个最重要的选项属性：*action*, *type*, *dest* (destination) 和*help*。其中，*action* 是最基本的一个。

理解选项动作

动作是告诉*optparse* 当它在命令行中遇到某个选项时要做什么。有一个固定的动作集被硬编码到*optparse* 内部；添加新的动作是将在*扩展 optparse* 章节中介绍的进阶内容。大多数动作都是告诉*optparse* 将特定的值存储到某个变量中 --- 例如，从命令行接收一个字符串并将其存储到 options 的某个选项中。

如果你没有指定一个选项动作，*optparse* 将默认选择 store。

store 动作

最常用的选项动作是 store，它告诉*optparse* 接收下一个参数（或当前参数的剩余部分），确认其为正确的类型，并将其保存至你选择的目标。

例如：

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

现在让我们编一个虚假的命令行并让*optparse* 来解析它：

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

当*optparse* 看到选项字符串 -f 时，它将获取下一个参数 foo.txt，并将其保存到 options.filename 中。因此，在这个对*parse_args()* 的调用之后，options.filename 将为 "foo.txt"。

受到*optparse* 支持的其他一些选项类型有 int 和 float。下面是一个接受整数参数的选项：

```
parser.add_option("-n", type="int", dest="num")
```

请注意这个选项没有长选项字符串，这是完全可接受的。而且，它也没有显式的动作，因为使用默认的 store。

让我们解析另一个虚假的命令行。这一次，我们将让选项参数与选项紧贴在一起：因为 -n42 (一个参数) 与 -n 42 (两个参数) 是等价的，以下代码

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

将会打印 42。

如果你没有指明类型，*optparse* 会假定类型为 string。加上默认动作为 store 这一事实，意味着我们的第一个示例可以变得更加简短：

```
parser.add_option("-f", "--file", dest="filename")
```

如果你没有提供目标，*optparse* 会从选项字符串推断出一个合理的默认目标：如果第一个长选项字符串为 --foo-bar，则默认目标为 foo_bar。如果没有长选项字符串，则*optparse* 会查找第一个短选项字符串：针对 -f 的默认目标将为 f。

optparse 还包括了内置的 complex 类型。添加类型的方式将在*扩展 optparse* 一节中介绍。

处理布尔值（旗标）选项

旗标选项 --- 当看到特定选项时将某个变量设为真值或假值 --- 是相当常见的。*optparse* 通过两个单独的动作支持它们，store_true 和 store_false。例如，你可能会有个 verbose 旗标将通过 -v 来启用并通过 -q 来禁用：

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

这里我们有两个相同目标的不同选项，这是完全可行的。(只是这意味着在设置默认值时你必须更加小心 --- 见下文所述。)

当*optparse* 在命令行中遇到 `-v` 时，它会将 `options.verbose` 设为 `True`；当它遇到 `-q` 时，则会将 `options.verbose` 设为 `False`。

其他动作

受到*optparse* 支持的其他动作还有:

"store_const"
存储一个常量值，通过*Option.const* 预设

"append"
将此选项的参数添加到一个列表

"count"
让指定的计数器加一

"callback"
调用指定函数

这些在*参考指南*，以及*选项回调* 等章节中有说明。

默认值

上述示例全都涉及当看到特定命令行选项时设置某些变量（即“目标”）的操作。如果从未看到这些选项那么会发生什么叱？由于我们没有提供任何默认值，它们全都会被设为 `None`。这通常是可以的，但有时你会想要更多的控制。*optparse* 允许你为每个目标提供默认值，它们将在解析命令行之前被赋值。

首先，考虑这个 verbose/quiet 示例。如果我们希望*optparse* 将 `verbose` 设为 `True` 除非看到了 `-q`，那么我们可以这样做:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

由于默认值将应用到 *destination* 而不是任何特定选项，并且这两个选项正好具有相同的目标，因此这是完全等价的:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

考虑一下:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

同样地，`verbose` 的默认值将为 `True`: 最终生效的将是最后提供给任何特定目标的默认值。

一种更清晰的默认值指定方式是使用 OptionParser 的 `set_defaults()` 方法，你可以在调用*parse_args()* 之前的任何时候调用它:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

如前面一样，最终生效的将是最后为特定选项目标指定的值。为清楚起见，请使用一种或另外一种设置默认值的方法，而不要同时使用。

生成帮助

optparse 自动生成帮助和用法文本的功能适用于创建用户友好的命令行界面。你所要做的只是为每个选项提供*help* 值，并可选项为你的整个程序提供一条简短的用法消息。下面是一个填充了用户友好的（文档）选项的 OptionParser:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

如果*optparse*在命令行中遇到了 `-h` 或 `--help`，或者如果你调用了 `parser.print_help()`，它会把以下内容打印到标准输出:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

（如果帮助输出是由 help 选项触发的，*optparse*将在打印帮助文本之后退出。）

在这里为帮助*optparse*生成尽可能好的帮助消息做了很多工作:

- 该脚本定义了自己的用法消息:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  *optparse*会将用法字符串中的 `%prog` 扩展为当前程序的名称，即 `os.path.basename(sys.argv[0])`。随后将在详细选项帮助之前打印这个经过扩展的字符串。

  如果你未提供用法字符串，*optparse*将使用一个直白而合理的默认值: `"Usage: %prog [options]"`，这在你的脚本不接受任何位置参数时是可以的。

- 每个选项都定义了帮助字符串，并且不用担心换行问题 --- *optparse*将负责执行换行并使帮助输出有良好的外观格式。

- 需要接受值的选项会在它们自动生成的帮助消息中提示这一点，例如对于”mode” 选项:

  ```
  -m MODE, --mode=MODE
  ```

  在这里，”MODE” 被称为元变量：它代表预期用户会提供给 `-m`/`--mode` 的参数。在默认情况下，*optparse*会将目标变量名转换为大写形式并将其用作元变量。有时，这并不是你所希望的 --- 例如，`--filename` 选项显式地设置了 `metavar="FILE"`，结果将自动生成这样的选项描述:

  ```
  -f FILE, --filename=FILE
  ```

  不过，这具有比节省一点空间更重要的作用：手动编写的帮助文本使用元变量 `FILE` 来提示用户在半正式的语法 `-f FILE` 和非正式的描述”write output to FILE” 之间存在联系。这是一种使你的帮助文本更清晰并对最终用户来说更易用的简单而有效的方式。

- 具有默认值的选项可以在帮助字符串中包括 `%default` --- *optparse*将用该选项的默认值的*str()*来替代它。如果一个选项没有默认值 (或默认值为 `None`)，则 `%default` 将被扩展为 `none`。

选项分组

在处理大量选项时，可以方便地将选项进行分组以提供更好的帮助输出。*OptionParser* 可以包含多个选项分组，每个分组可以包含多个选项。

选项分组是使用*OptionGroup* 类来生成的:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

其中

- parser 是分组将被插入的*OptionParser* 实例
- title 是分组的标题
- description，可选项，是分组的长描述文本

OptionGroup 继承自 OptionContainer (类似*OptionParser*) 因此 add_option() 方法可被用来向分组添加选项。

一旦声明了所有选项，使用*OptionParser* 方法 add_option_group() 即可将分组添加到之前定义的解析器。

继续使用前一节定义的解析器，很容易将*OptionGroup* 添加到解析器中:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

这将产生以下帮助输出:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

更完整一些的示例可能涉及使用多个分组：继续扩展之前的例子:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

这会产生以下输出:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

另一个有趣的方法，特别适合在编程处理选项分组时使用:

OptionParser.**get_option_group**(*opt_str*)

返回短或长选项字符串 *opt_str* (例如 `'-o'` 或 `'--option'`) 所属的*OptionGroup*。如果没有对应的*OptionGroup*，则返回 None。

打印版本字符串

与简短用法字符串类似，*optparse* 还可以打印你的程序的版本字符串。你必须将该字符串作为 version 参数提供给 OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

%prog 会像在 usage 中那样被扩展。除了这一点，version 还可包含你想存放的任何东西。当你提供它时，*optparse* 将自动向你的解析器添加一个 --version 选项。如果它在命令行中遇到了该选项，它将扩展你的 version 字符串 (通过替换 %prog)，将其打印到标准输出，然后退出。

举例来说，如果你的脚本是 /usr/bin/foo:

```
$ /usr/bin/foo --version
foo 1.0
```

下列两个方法可被用来打印和获取 version 字符串:

OptionParser.**print_version**(*file=None*)

将当前程序的版本消息 (self.version) 打印到 *file* (默认为 stdout)。就像*print_usage()* 一样，任何在 self.version 中出现的 %prog 将被替换为当前程序的名称。如果 self.version 为空或未定义则不做任何操作。

OptionParser.**get_version**()

与*print_version()* 相似但是会返回版本字符串而不是打印它。

optparse 如何处理错误 handles errors

optparse 必须考虑两种宽泛的错误类：程序员错误和用户错误。程序员错误通常是对*OptionParser.add_option()* 的错误调用，例如无效的选项字符串，未知的选项属性，不存在的选项属性等等。这些错误将以通常的方式来处理：引发一个异常 (或者是*optparse.OptionError* 或者是*TypeError*) 并让程序崩溃。

处理用户错误更为重要，因为无论你的代码有多稳定他们都肯定会发生。*optparse* 可以自动检测部分用户错误，例如不正确的选项参数（如传入 `-n 4x` 而 `-n` 接受整数参数），缺少参数（如 `-n` 位于命令行的末尾，而 `-n` 接受任意类型的参数）。并且，你可以调用 `OptionParser.error()` 来指明应用程序自定义的错误条件:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

在两种情况下，*optparse* 都是以相同方式处理错误的：它会将程序的用法消息和错误消息打印到标准错误并附带错误状态 2 退出。

考虑上面的第一个示例，当用户向一个接受整数的选项传入了 `4x`:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

或者，当用户未传入任何值:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse 生成的错误消息总是会确保提示在错误中涉及的选项；请确保在从你的应用程序代码调用 `OptionParser.error()` 时也做同样的事。

如果*optparse* 的默认错误处理行为不适合你的需求，你需要子类化 OptionParser 并重写它的 `exit()` 和/或 `error()` 方法。

合并所有代码

下面是基于*optparse* 的脚本通常的结构:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

17.2.5 参考指南

创建解析器

使用*optparse*的第一步是创建 OptionParser 实例。

class optparse.**OptionParser**(...)

OptionParser 构造器没有必需的参数，只有一些可选的关键字参数。你应当始终以关键字参数形式传入它们，即不要依赖于声明参数所在位置的顺序。

usage (默认: "%prog [options]")
当你的程序不正确地运行或附带 help 选项运行时将打印的用法说明。当*optparse*打印用法字符串时，它会将 `%prog` 扩展为 `os.path.basename(sys.argv[0])` (或者如果你传入 `prog` 则为该关键字参数值)。要屏蔽用法说明，请传入特殊值 `optparse.SUPPRESS_USAGE`。

option_list (默认: [])
一个用于填充解析器的由 Option 对象组成的列表。`option_list` 中的选项将添加在 `standard_option_list` (一个可由 OptionParser 的子类设置的类属性) 中的任何选项之后，以及任何版本或帮助选项之前。已被弃用；请改为在创建解析器之后使用*add_option()*。

option_class (默认: optparse.Option)
当在*add_option()* 中向解析器添加选项时要使用的类。

version (默认: None)
当用户提供了 version 选项时将会打印的版本字符串。如果你为 `version` 提供真值，*optparse*将自动添加单个选项字符串 `--version` 形式的 version 选项。子字符串 `%prog` 会以与 `usage` 相同的方式扩展。

conflict_handler (默认: "error")
指定当有相互冲突的选项字符串的选项被添加到解析器时要如何做；参见*选项之间的冲突*一节。

description (默认: None)
一段提供你的程序的简短介绍的文本。*optparse*会重格式化段落以适合当前终端宽度并在用户请求帮助时打印其内容（在 `usage` 之后，选项列表之前）。

formatter (默认: 一个新的 IndentedHelpFormatter)
一个将被用于打印帮助文本的 optparse.HelpFormatter 实例。*optparse*为此目的提供了两个实体类: IndentedHelpFormatter 和 TitledHelpFormatter。

add_help_option (默认: True)
如为真值，*optparse*将向解析器添加一个 help 选项 (使用选项字符串 `-h` 和 `--help`)。

prog
当在 `usage` 和 `version` 中用于代替 `os.path.basename(sys.argv[0])` 来扩展 `%prog` 的字符串。

epilog (默认: None)
一段将在选项帮助之后打印的帮助文本。

填充解析器

有几种方式可以为解析器填充选项。最推荐的方式是使用*OptionParser.add_option()*，如*教程*一节所演示的。`add_option()` 可以通过两种方式来调用:

- 传入一个 Option 实例（即 `make_option()` 所返回的对象）
- 传入 `make_option()` 可接受的（即与 Option 构造器相同的）任意位置和关键字参数组合，它将为你创建 Option 实例

另一种方式是将由预先构造的 Option 实例组成的列表传给 OptionParser 构造器，如下所示:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
```

(续下页)

(接上页)

```
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(make_option() 是一个用于创建 Option 实例的工厂函数；目前它是 Option 构造器的一个别名。未来的*optparse* 版本可能会将 Option 拆分为多个类，而 make_option() 将选择适当的类来实例化。请不要直接实例化 Option。)

定义选项

每个 Option 实例代表一组同义的命令行选项字符串，例如 -f 和 --file。你可以指定任意数量的短和长选项字符串，但你必须指定总计至少一个选项字符串。

创建*Option* 的正规方式是使用*OptionParser* 的 add_option() 方法。

OptionParser.**add_option**(*option*)

OptionParser.**add_option**(**opt_str*, *attr=value*, ...)

定义只有一个短选项字符串的选项:

```
parser.add_option("-f", attr=value, ...)
```

以及定义只有一个长选项字符串的选项:

```
parser.add_option("--foo", attr=value, ...)
```

该关键字参数定义新 Option 对象的属性。最重要的选项属性是*action*，它主要负责确定其他的属性是相关的还是必须的。如果你传入了不相关的选项属性，或是未能传入必须的属性，*optparse* 将引发一个*OptionError* 异常来说明你的错误。

选项的 *action* 决定当*optparse* 在命令行中遇到该选项时要做什么。硬编码在*optparse* 中的标准选项动作有:

"store"
存储此选项的参数（默认）

"store_const"
存储一个常量值，通过*Option.const* 预设

"store_true"
存储 True

"store_false"
存储 False

"append"
将此选项的参数添加到一个列表

"append_const"
将指定常量值添加到一个列表，可通过*Option.const* 预设

"count"
让指定的计数器加一

"callback"
调用指定函数

"help"
打印用法消息，包括所有选项和它们的文档

（如果你没有提供动作，则默认为 "store"。对于此动作，你还可以提供*type* 和*dest* 选项属性；参见*标准选项动作*。）

如你所见，大多数动作都在某处保存或更新一个值。*optparse* 总是会为此创建一个特殊对象，它被恰当地称为 options，是*optparse.Values* 的实例。

class optparse.**Values**

一个将被解析的参数名和值作为属性保存的对象。一般是通过调用*OptionParser.parse_args()* 来创建，并可被传给*OptionParser.parse_args()* 的 *values* 参数的自定义子类所覆盖（如在*解析参数* 中描述的那样）。

Option 参数（以及各种其他的值）将根据*dest* (目标) 选项属性被保存为此对象的属性。

例如，当你调用

```
parser.parse_args()
```

optparse 首先会做的一件事情是创建 options 对象:

```
options = Values()
```

如果该解析器中的某个选项定义带有

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

并且被解析的命令行包括以下任意一项:

```
-ffoo
-f foo
--file=foo
--file foo
```

那么*optparse* 在看到此选项时，将执行这样的操作

```
options.filename = "foo"
```

type 和*dest* 选项属性几乎与*action* 一样重要，但*action* 是唯一对 所有选项都有意义的。

选项属性

class optparse.**Option**

一个单独的命令行参数，带有以关键字参数形式传给构造器的各种属性。通常使用*OptionParser.add_option()* 创建而不是直接创建，并可被作为*OptionParser* 的 *option_class* 参数传入的自定义类来重写。

下列选项属性可以作为关键字参数传给*OptionParser.add_option()*。如果你传入一个与特定选项无关的选项属性，或是未能传入必要的选项属性，*optparse* 将会引发*OptionError*。

Option.**action**

(默认: "store")

用于当在命令行中遇到此选项时确定*optparse* 的行为；可用的选项记录在*这里*。

Option.**type**

(默认: "string")

此选项所接受的参数类型 (例如 "string" 或 "int")；可用的选项类型记录在*这里*。

Option.**dest**

(默认: 获取自选项字符串)

如果此选项的动作涉及在某处写入或修改一个值，该属性将告诉*optparse* 将它写入到哪里: *dest* 指定*optparse* 在解析命令行时构建的 options 对象的某个属性。

Option.**default**

当未在命令行中遇到此选项时将被用作此选项的目标的值。另请参阅*OptionParser.set_defaults()*。

Option.**nargs**

(默认: 1)

当遇到此选项时应当读取多少个*type* 类型的参数。如果 > 1，*optparse* 会将由多个值组成的元组保存到*dest*。

Option.**const**

对于保存常量值的动作，指定要保存的常量值。

Option.**choices**

对于 "choice" 类型的选项，由用户可选择的字符串组成的列表。

Option.**callback**

对于使用 "callback" 动作的选项，当遇到此选项时要调用的可调用对象。请参阅选项回调 一节了解关于传给可调用对象的参数的详情。

Option.**callback_args**

Option.**callback_kwargs**

将在四个标准回调参数之后传给 callback 的额外的位置和关键字参数。

Option.**help**

当用户提供*help* 选项 (如 --help) 之后将在列出所有可有远项时针对此选项打印的文本。如果没有提供帮助文本，则列出选项时将不附带帮助文本。要隐藏此选项，请使用特殊值 optparse.SUPPRESS_HELP。

Option.**metavar**

(默认: 获取自选项字符串)

当打印帮助文本时要使用的代表选项参数的名称。请参阅教程 一节查看相应示例。

标准选项动作

各种选项动作具有略微不同的要求和效果。大多数动作都具有几个可被你指定的独步选项属性用来控制*optparse* 的行为；少数还具有一些必需属性，你必须为任何使用该动作的选项指定这些属性。

- "store" [关联: *type*, *dest*, *nargs*, *choices*]

  该选项后必须跟一个参数，它将根据*type* 被转换为相应的值并保存至*dest*。如果*nargs* > 1，则将从命令行读取多个参数；它们将全部根据*type* 被转换并以元组形式保存至*dest*。参见标准选项类型 一节。

  如果提供了*choices* (由字符串组成的列表和元组)，则类型默认为 "choice"。

  如果未提供*type*，则默认为 "string"。

  如果未提供*dest*，则*optparse* 会从第一个长选项字符串派生出目标 (例如 --foo-bar 将对应 foo_bar)。如果不存在长选项字符串，则*optparse* 会从第一个短选项字符串派生出目标 (例如 -f 将对应 f)。

  示例:

  ```
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```

  当它解析命令行

  ```
  -f foo.txt -p 1 -3.5 4 -fbar.txt
  ```

  optparse 将设置

  ```
  options.f = "foo.txt"
  options.point = (1.0, -3.5, 4.0)
  options.f = "bar.txt"
  ```

- "store_const" [要求: *const*; 关联: *dest*]

  值*const* 将存放到*dest* 中。

  示例:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  如果看到了 --noisy，*optparse* 将设置

```
options.verbose = 2
```

- "store_true" [关联: *dest*]

  将 True 存放到*dest* 中的 "store_const" 的特例。

- "store_false" [关联: *dest*]

  类似于 "store_true"，但是存放 False。

  示例:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [关联: *type*, *dest*, *nargs*, *choices*]

  该选项必须跟一个参数，该参数将被添加到*dest* 的列表中。如果未提供*dest* 的默认值，那么当*optparse* 首次在命令行中遇到该选项时将自动创建一个空列表。如果*nargs* > 1，则会读取多个参数，并将一个长度为*nargs* 的元组添加到*dest*。

  type 和*dest* 的默认值与 "store" 动作的相同。

  示例:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  如果在命令行中遇到 -t3，*optparse* 将执行这样的操作:

```
options.tracks = []
options.tracks.append(int("3"))
```

  如果，在稍后的时候，再遇到 --tracks=4，它将执行:

```
options.tracks.append(int("4"))
```

  append 动作会在选项的当前值上调用 append 方法。这意味着任何被指定的默认值必须具有 append 方法。这还意味着如果默认值非空，则其中的默认元素将存在于选项的已解析值中，而任何来自命令行的值将被添加到这些默认值之后:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [需要: *const*; 关联: *dest*]

  与 "store_const" 类似，但*const* 值将被添加到*dest*；与 "append" 一样，*dest* 默认为 None，并且当首次遇到该选项时将自动创建一个空列表。

- "count" [关联: *dest*]

  对保存在*dest* 的整数执行递增。如果未提供默认值，则*dest* 会在第一次执行递增之前被设为零。

  示例:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  第一次在命令行中看到 -v 时，*optparse* 将执行这样的操作:

```
options.verbosity = 0
options.verbosity += 1
```

  后续每次出现 -v 都将导致

```
options.verbosity += 1
```

- "callback" [需要: *callback*; 关联: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  调用*callback* 所指定的函数，它将以如下形式被调用

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  请参阅选项回调 一节了解详情。

- "help"

  为当前选项解析器中所有的选项打印完整帮助消息。该帮助消息是由传给 OptionParser 的构造器的 usage 字符串和传给每个选项的*help* 字符串构造而成的。

  如果没有为某个选项提供*help* 字符串，它仍将在帮助消息中列出。要完全略去某个选项，请使用特殊值 optparse.SUPPRESS_HELP。

  optparse 将自动为所有 OptionParser 添加*help* 选项，因此你通常不需要创建选项。

  示例:

```
from optparse import OptionParser, SUPPRESS_HELP

# 通常，help 选项会自动添加，但可以
# 使用 add_help_option 参数来屏蔽
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

  如果*optparse* 在命令行中看到 -h 或 --help，它将将类似下面这样的帮助消息打印到 stdout (假设 sys.argv[0] 为 "foo.py"):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

  在打印帮助消息之后，*optparse* 将使用 sys.exit(0) 来终结你的进程。

- "version"

  将提供给 OptionParser 的版本号打印到 stdout 并退出。该版本号实际上是由 OptionParser 的 print_version() 方法进行格式化并打印的。这通常只在向 OptionParser 构造器提供了 version

参数时才有意义。与*help* 选项类似，你很少会创建 version 选项，因为*optparse* 会在需要时自动添加它们。

标准选项类型

optparse 有五种内置选项类型: "string", "int", "choice", "float" 和 "complex"。如果你需要添加新的选项类型，请参阅*扩展 optparse* 一节。

传给 string 类型选项的参数不会以任何方式进行检查或转换：命令行中的文本将被原样保存至目标（或传给回调）。

整数参数 ("int" 类型) 将以如下方式解析:

- 如果数字开头为 0x，它将被解析为十六进制数
- 如果数字开头为 0，它将被解析为八进制数
- 如果数字开头为 0b，它将被解析为二进制数
- 在其他情况下，数字将被解析为十进制数

转换操作是通过调用*int()* 并传入适当的 base (2, 8, 10 或 16) 来完成的。如果转换失败，*optparse* 也将失败，但它会附带更有用的错误消息。

"float" 和 "complex" 选项参数会直接通过*float()* 和*complex()* 来转换，使用类似的错误处理。

"choice" 选项是 "string" 选项的子类型。*choices* 选项属性（由字符串组成的序列）定义了允许的选项参数的集合。optparse.check_choice() 将用户提供的选项参数与这个主列表进行比较并会在给出无效的字符串时引发*OptionValueError*。

解析参数

创建和填充 OptionParser 的基本目的是调用其*parse_args()* 方法。

OptionParser.**parse_args**(*args=None*, *values=None*)

解析 *args* 中的命令行选项。

输入形参为

args
要处理的参数列表 (默认: sys.argv[1:])

values
要用于存储选项参数的*Values* 对象（默认值：一个新的*Values* 实例）-- 如果你给出一个现有对象，则不会基于它来初始化选项的默认值。

并且返回值是一个 (options, args) 对，其中

options
就是作为 *values* 传入的同一个对象，或是由*optparse* 创建的 optparse.Values 实例

args
在所有选项被处理完毕后余下的位置参数

最常见的用法是不提供任何关键字参数。如果你提供了 values，它将通过重复的*setattr()* 调用来修改（大致为每个存储到指定选项目标的选项参数调用一次）并由*parse_args()* 返回。

如果*parse_args()* 在参数列表中遇到任何错误，它将调用 OptionParser 的 error() 方法并附带适当的最终用户错误消息。这会完全终结你的进程并将退出状态设为 2 (传统的针对命令行错误的 Unix 退出状态)。

查询和操纵你的选项解析器

选项解析器的默认行为可被轻度地定制，并且你还可以调整你的选项解析器查看实际效果如何。OptionParser 提供了一些方法来帮助你进行定制:

OptionParser.**disable_interspersed_args**()

设置解析在第一个非选项处停止。举例来说，如果 -a 和 -b 都是不接受参数的简单选项，则*optparse*通常会接受这样的语法:

```
prog -a arg1 -b arg2
```

并会这样处理它

```
prog -a -b arg1 arg2
```

要禁用此特性，则调用*disable_interspersed_args()*。这将恢复传统的 Unix 语法，其中选项解析会在第一个非选项参数处停止。

如果你用一个命令处理程序来运行另一个拥有它自己的选项的命令而你希望参确保这些选项不会被混淆就可以使用此方法。例如，每个命令可能具有不同的选项集合。

OptionParser.**enable_interspersed_args**()

设置解析不在第一个非选项处停止，允许多个命令行参数的插入相互切换。这是默认的行为。

OptionParser.**get_option**(*opt_str*)

返回具有选项字符串 *opt_str* 的的 Option 实例，或者如果不存在具有该选项字符串的选项则返回 None。

OptionParser.**has_option**(*opt_str*)

如果 OptionParser 包含一个具有选项字符串 *opt_str* 的选项 (例如 -q 或 --verbose) 则返回 True。

OptionParser.**remove_option**(*opt_str*)

如果*OptionParser*包含一个对应于 *opt_str* 的选项，则移除该选项。如果该选项提供了任何其他选项字符串，这些选项字符串将全部不可用。如果 *opt_str* 不存在于任何属于此*OptionParser*的选项之中，则会引发*ValueError*。

选项之间的冲突

如果你不够小心，很容易会定义具有相互冲突的选项字符串的多个选项:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

（当你自定义具有某些标准选项的 OptionParser 时特别容易发生这种情况。）

每当你添加一个选项时，*optparse*会检查它是否与现有选项冲突。如果发现存在冲突，它将唤起当前的冲突处理机制。你可以选择在构造器中设置冲突处理机制:

```
parser = OptionParser(..., conflict_handler=handler)
```

或是在单独调用中设置:

```
parser.set_conflict_handler(handler)
```

可用的冲突处理器有:

"error" (默认)
将选项冲突视为编程错误并引发*OptionConflictError*

"resolve"
智能地解决选项冲突（见下文）

举例来说，让我们定义一个智能地解决冲突的*OptionParser*并向其添加相互冲突的选项:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

这时，*optparse* 会检测到之前添加的选项已经在使用 -n 选项字符串。由于 conflict_handler 为 "resolve"，它将通过在之前选项的选项字符串列表中移除 -n 来解决冲突。现在 --dry-run 将是用户激活该选项的唯一方式。如果用户要获取帮助，帮助消息将反映这一变化:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

之前添加的选项有可能不断更取代直到一个都不剩，这样用户将无法再从命令行唤起相应的选项。在这种情况下，*optparse* 将完全移除这样的选项，使它不会在帮助文本或任何其他地方显示。如果我们继续修改现有的 OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

这时，原有的 -n/--dry-run 选项将不再可用，因此*optparse* 会将其移除，帮助文本将变成这样:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

清理

OptionParser 实例存在一些循环引用。这对 Python 的垃圾回收器来说应该不是问题，但你可能希望在你完成对 OptionParser 的使用后通过调用其 destroy() 来显式地中断循环引用。这在可以从你的 OptionParser 访问大型对象图的长期运行应用程序中特别有用。

其他方法

OptionParser 还支持其他一些公有方法:

OptionParser.**set_usage**(*usage*)

根据上述的规则为 usage 构造器关键字参数设置用法字符串。传入 None 将设置默认的用法字符串；使用 optparse.SUPPRESS_USAGE 将屏蔽用法说明。

OptionParser.**print_usage**(*file=None*)

将当前程序的用法消息 (self.usage) 打印到 *file* (默认为 stdout)。出现在 self.usage 中的字符串 %prog 将全部被替换为当前程序的名称。如果 self.usage 为空或未定义则不做任何事情。

OptionParser.**get_usage**()

与*print_usage()* 类似但是将返回用法字符串而不是打印它。

OptionParser.**set_defaults**(*dest=value, ...*)

一次性地为多个选项目标设置默认值。使用*set_defaults()* 是为选项设置默认值的推荐方式同，因为多个选项可以共享同一个目标。举例来说，如果几个"mode" 选项全部设置了相同的目标，则它们中的任何一个都可以设置默认值，而最终生效的将是最后一次设置的默认值:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # 覆盖下面的设置
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # 覆盖上面的设置
```

为避免混淆，请使用*set_defaults()*:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

17.2.6 选项回调

当*optparse*的内置动作和类型不能满足你的需要时，你有两个选择：扩展*optparse*或定义一个回调选项。扩展*optparse*的方式更为通用，但对许多简单场景来说是大材小用了。你所需要的往往只是一个简单的回调。

定义一个回调选项分为两步:

- 使用 "callback" 动作定义选项本身
- 编写回调；它是一个接受至少四个参数的函数（或方法），如下所述

定义回调选项

一般来说，定义回调选项的最简单方式是使用*OptionParser.add_option()*方法。在*action*之外，你必须指定的唯一选项属性是 callback，即要调用的函数:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

callback 是一个函数（或其他可调用对象），因此当你创建这个回调选项时你必须已经定义了 my_callback()。在这个简单的例子中，*optparse*甚至不知道 -c 是否接受任何参数，这通常意味着该选项不接受任何参数 --- 它需要知道的就是存在命令行参数 -c。但是，在某些情况下，你可能希望你的回调接受任意数量的命令行参数。这是编写回调的一个麻烦之处；本小节将稍后讲解这个问题。

*optparse*总是会向你的回调传递四个特定参数，它只会在你通过*callback_args*和*callback_kwargs*进行指定时才传入额外的参数。因此，最小化的回调函数签名如下:

```
def my_callback(option, opt, value, parser):
```

对传给回调的四个参数的说明见下文。

当你定义回调选项时还可以提供一些其他的选项属性:

type
: 具有其通常的含义：与在 "store" 或 "append" 动作中一样，它指示*optparse*读取一个参数并将其转换为*type*。但是，*optparse*并不会将所转换的值保存到某个地方，而是将其传给你的回调函数。

nargs
: 同样具有其通常的含义：如果提供了该属性并且其值 > 1，*optparse*将读取*nargs*个参数，每个参数都必须可被转换为*type*。随后它会将被转换值的元组传给你的回调。

callback_args
: 一个要传给回调的由额外位置参数组成的元组

callback_kwargs
: 一个要传给回调的由额外关键字参数组成的字典

回调应当如何调用

所有回调都将使用以下方式调用:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

其中

option
: 是调用该回调的 Option 实例

opt_str
: 是在触发回调的命令行参数中出现的选项字符串。(如果使用了长选项的缩写形式，则 opt_str 将为完整规范的选项字符串 --- 举例来说，如果用户在命令行中将 --foo 作为 --foobar 的缩写形式，则 opt_str 将为 "--foobar"。)

value
是在命令行中提供给该选项的参数。*optparse* 将只在设置了*type* 的时候才接受参数；value 将为该选项的类型所指定的类型。如果该选项的*type* 为 None (不接受参数)，则 value 将为 None。如果*nargs* > 1，则 value 将为由指定类型的值组成的元组。

parser
是驱动选项解析过程的 OptionParser 实例，主要作用在于你可以通过其实例属性访问其他一些相关数据:

parser.largs
当前的剩余参数列表，即已被读取但不属于选项或选项参数的参数。可以任意修改 parser.largs，例如通过向其添加更多的参数。(该列表将成为 args，即*parse_args()* 的第二个返回值。)

parser.rargs
当前的保留参数列表，即移除了 opt_str 和 value (如果可用)，并且只有在它们之后的参数才会被保留。可以任意修改 parser.rargs，例如通过读取更多的参数。

parser.values
作为选项值默认保存位置的对象（一个 optparse.OptionValues 实例）。这使得回调能使用与*optparse* 的其他部分相同的机制来保存选项值；你不需要手动处理全局变量或闭包。你还可以访问或修改在命令行中遇到的任何选项的值。

args
是一个由通过*callback_args* 选项属性提供的任意位置参数组成的元组。

kwargs
是一个由通过*callback_kwargs* 提供的任意关键字参数组成的字典。

在回调中引发错误

如果选项或其参数存在任何问题则回调函数应当引发*OptionValueError*。*optparse* 将捕获该异常并终止程序，将你提供的错误消息打印到 stderr。你的消息应当清晰、简洁、准确并指明出错的选项。否则，用户将很难弄清楚自己做错了什么。

回调示例 1：最简回调

下面是一个不接受任何参数，只是简单地记录所遇见的选项的回调选项示例:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

当然，你也可以使用 "store_true" 动作做到这一点。

回调示例 2：检查选项顺序

下面是一个更有趣些的示例：当看到 -a 出现时将会记录，而如果它在命令行中出现于 -b 之后则将报告错误。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

回调示例 3：检查选项顺序（通用）

如果你希望为多个类似的选项重用此回调（设置旗标，并在 -b 已经出现时触发），则需要一些额外工作：它设置的错误消息和旗标必须进行通用化。

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

回调示例 4：检查任意条件

当然，你可以设置任何条件 --- 并不限于检查已定义选项的值。举例来说，如果你有一个不应当在满月时被调用的选项，你就可以这样做：

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

（定义 `is_moon_full()` 的任务将作为留给读者的练习。）

回调示例 5：固定的参数

当你定义接受固定数量参数的 callback 选项时情况会变得更有趣一点。指定一个 callback 选项接受参数的操作类似于定义一个 `"store"` 或 `"append"` 选项：如果你定义*type*，那么该选项将接受一个必须可被转换为相应类型的参数；如果你进一步定义*nargs*，那么该选项将接受*nargs* 个参数。

下面是一个模拟了标准 `"store"` 动作的示例：

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

请注意*optparse* 将为你读取 3 个参数并将它们转换为整数；你所要做的只是保存它们。（或任何其他操作；对于这个示例显然你不需要使用回调。）

回调示例 6：可变的参数

当你想要一个选项接受可变数数量的参数时情况会变得更麻烦一点。对于这种场景，你必须编写一个回调，因为*optparse* 没有为它提供任何内置的相应功能。而你必须处理*optparse* 通常会为你处理的传统 Unix 命令行的某些细节问题。特别地，回调应当实现单个 `--` 和 `-` 参数的惯例规则：

- `--` 或 `-` 都可以作为选项参数
- 单个 `--` (如果不是某个选项的参数): 停止命令行处理并丢弃该 `--`
- 单个 `-` (如果不是某个选项的参数): 停止命令行处理但保留 `-` (将其添加到 `parser.largs`)

如果你想要一个选项接受可变数量的参数，那么有几个微妙、棘手的问题需要考虑到。你选择的具体实现将基于你的应用程序对于各方面利弊的权衡（这就是为什么*optparse* 没有直接支持这一功能）。

无论如何，下面是一个对于接受可变参数的选项的回调的尝试：

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

17.2.7 扩展 optparse

由于控制*optparse* 如何读取命令行选项的两个主要因子是每个选项的动作和类型，所以扩展最可能的方向就是添加新的动作和新的类型。

添加新的类型

要添加新的类型，你必须自定义*optparse* 的*Option* 类的子类。这个类包含几个用来定义*optparse* 的类型的属性: *TYPES* 和*TYPE_CHECKER*。

Option.**TYPES**

一个由类型名称组成的元组；在你的子类中，简单地定义一个在标准元组基础上构建的新元组*TYPES*。

Option.**TYPE_CHECKER**

一个将类型名称映射到类型检查函数的字典。类型检查函数具有如下签名:

```
def check_mytype(option, opt, value)
```

其中 option 是一个*Option* 实例，opt 是一个选项字符串 (例如 -f)，而 value 是来自命令行的必须被检查并转换为你想要的类型的字符串。check_mytype() 应当返回假设的类型 mytype 的对象。类型检查函数所返回的值将最终出现在*OptionParser.parse_args()* 所返回的 OptionValues 实例中，或是作为 value 形参传给一个回调。

如果你的类型检查函数遇到任何问题则应当引发*OptionValueError*。*OptionValueError* 接受一个字符串参数，该参数将被原样传递给*OptionParser* 的 error() 方法，该方法将随后附加程序名称和字符串 "error:" 并在终结进程之前将所有信息打印到 stderr。

下面这个很傻的例子演示了如何添加一个 "complex" 选项类型以便在命令行中解析 Python 风格的复数。(现在这个例子比以前更傻了，因为*optparse* 1.3 增加了对复数的内置支持，但是不必管它了。)

首先，必要的导入操作:

```
from copy import copy
from optparse import Option, OptionValueError
```

你必须先定义自己的类型检查器，因为以后它会被引用（在你的 Option 子类的*TYPE_CHECKER* 类属性中）：

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

最后，是 Option 子类:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

（如果我们不对*Option.TYPE_CHECKER* 执行*copy()*，我们就将修改*optparse* 的 Option 类的*TYPE_CHECKER* 属性。Python 就是这样，除了礼貌和常识以外没有任何东西能阻止你这样做。）

就是这样！现在你可以编写一个脚本以与其他基于*optparse* 的脚本相同的方式使用新的选项类型，除了你必须指示你的 OptionParser 使用 MyOption 而不是 Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

作为替代选择，你可以构建你自己的选项列表并将它传给 OptionParser；如果你不是以上述方式使用 `add_option()`，则你不需要告诉 OptionParser 使用哪个选项类:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

添加新的动作

添加新的动作有一点复杂，因为你必须理解*optparse* 对于动作有几种分类:

”store” 类动作
: 会使得*optparse* 将某个值保存到当前 OptionValues 实例的特定属性中的动作；这些选项要求向 Option 构造器提供一个*dest* 属性。attribute to be supplied to the constructor.

”typed” 类动作
: 从命令行接受某个值并预期它是一个特定类型；或者更准确地说，是可被转换为一个特定类型的字符串的动作。这些选项要求向 Option 构造器提供一个*type* 属性。attribute to the constructor.

这些是相互重叠的集合：默认的”store” 类动作有 `"store"`, `"store_const"`, `"append"` 和 `"count"`，而默认的”typed” 类动作有 `"store"`, `"append"` 和 `"callback"`。

当你添加一个动作时，你需要将它列在 Option 的以下类属性的至少一个当中以对它进行分类（全部为字符串列表）：

`Option.`**`ACTIONS`**
: 所有动作必须在 ACTIONS 中列出。

`Option.`**`STORE_ACTIONS`**
: ”store” 类动作要额外地在此列出。

`Option.`**`TYPED_ACTIONS`**
: ”typed” 类动作要额外地在此列出。

`Option.`**`ALWAYS_TYPED_ACTIONS`**
: 总是会接受一个类型的动作（即其选项总是会接受一个值）要额外地在此列出。它带来的唯一影响是*optparse* 会将默认类型 `"string"` 赋值给动作在*ALWAYS_TYPED_ACTIONS* 中列出而未显式指定类型的选项。

为了真正实现你的新动作，你必须重写 Option 的 take_action() 方法并添加一个识别你的动作的分支。

例如，让我们添加一个 "extend" 动作。它类似于标准的 "append" 动作，但 "extend" 不是从命令行接受单个值并将其添加到现有列表，而是接受形式为以单个逗号分隔的多个值的字符串，并用这些值来扩展现有列表。也就是说，如果 --names 是一个类型为 "string" 的 "extend" 选项，则命令行

```
--names=foo,bar --names blah --names ding,dong
```

将得到一个列表

```
["foo", "bar", "blah", "ding", "dong"]
```

我们再定义一个 Option 的子类:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

应注意的特性:

- "extend" 既预期在命令行接受一个值又会将该值保存到某处，因此它同时被归类于*STORE_ACTIONS* 和*TYPED_ACTIONS*。
- 为确保*optparse* 将 "string" 的默认类型赋值给 "extend" 动作，我们同时将 "extend" 动作归类于*ALWAYS_TYPED_ACTIONS*。
- MyOption.take_action() 只实现了这一个新动作，并将控制权回传给 Option.take_action() 以执行标准的*optparse* 动作。
- values 是 optparse_parser.Values 类的一个实例，该类提供了非常有用的 ensure_value() 方法。ensure_value() 实际就是一个带有安全阀的*getattr()*；它的调用形式为

  ```
  values.ensure_value(attr, value)
  ```

  如果 values 的 attr 属性不存在或为 None，则 ensure_value() 会先将其设为 value，然后返回 value。这非常适用于 "extend", "append" 和 "count" 等动作，它们会将数据累积在一个变量中并预期该变量属于特定的类型（前两项是一个列表，后一项是一个整数）。使用 ensure_value() 意味着使用你的动作的脚本无需关心为相应的选项目标设置默认值；可以简单地保持默认的 None 而 ensure_value() 将在必要时负责为其设置适当的值。

17.2.8 异常

exception optparse.**OptionError**

当使用无效或不一致的参数创建*Option* 实例时将被引发。

exception optparse.**OptionConflictError**

当向*OptionParser* 添加相互冲突的选项时将被引发。

exception optparse.**OptionValueError**

当在命令行中遇到无效的选项值时将被引发。

exception optparse.**BadOptionError**

当在命令行中传入无效的选项时将被引发。

exception optparse.**AmbiguousOptionError**

当在命令行中传入有歧义的选项时将被引发。

17.3 getpass --- 可移植的密码输入

源代码: Lib/getpass.py

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅 *WebAssembly 平台* 了解详情。

getpass 模块提供了两个函数:

getpass.**getpass**(*prompt='Password: '*, *stream=None*)

提示用户输入一个密码且不会回显。用户会看到字符串 *prompt* 作为提示，其默认值为 `'Password: '`。在 Unix 上，如有必要提示会使用替换错误句柄写入到文件型对象 *stream*。*stream* 默认指向控制终端 (`/dev/tty`)，如果不可用则指向 `sys.stderr` (此参数在 Windows 上会被忽略)。

如果回显自由输入不可用则 getpass() 将回退为打印一条警告消息到 *stream* 并且从 `sys.stdin` 读取同时发出 *GetPassWarning*。

> **备注**
>
> 如果你从 IDLE 内部调用 getpass，输入可能是在你启动 IDLE 的终端中而非在 IDLE 窗口本身中完成。

exception getpass.**GetPassWarning**

一个当密码输入可能被回显时发出的 *UserWarning* 子类。

getpass.**getuser**()

返回用户的“登录名称”。

此函数会按顺序检查环境变量 `LOGNAME`, `USER`, `LNAME` 和 `USERNAME`，并返回其中第一个被设为非空字符串的值。如果全都未设置，则在支持 *pwd* 模块的系统上将返回来自密码数据库的登录名，在其他情况下，将会引发 *OSError*。

通常情况下，此函数应优先于 *os.getlogin()*。

在 3.13 版本发生变更: 在之前版本中，会引发包括 *OSError* 在内的多种异常。

17.4 fileinput --- 迭代来自多个输入流的行

源代码: Lib/fileinput.py

此模块实现了一个辅助类和一些函数用来快速编写访问标准输入或文件列表的循环。如果你只想要读写一个文件请参阅 *open()*。

典型用法为:

```
import fileinput
for line in fileinput.input(encoding="utf-8"):
    process(line)
```

此程序会迭代 sys.argv[1:] 中列出的所有文件内的行，如果列表为空则会使用 sys.stdin。如果有一个文件名为 '-'，它也会被替换为 sys.stdin 并且可选参数 *mode* 和 *openhook* 会被忽略。要指定替代文件列表，请将其作为第一个参数传给*input()*。也允许使用单个文件。

所有文件都默认以文本模式打开，但你可以通过在调用*input()* 或*FileInput* 时指定 *mode* 形参来覆盖此行为。如果在打开或读取文件时发生了 I/O 错误，将会引发*OSError*。

在 3.3 版本发生变更: 原来会引发*IOError*；现在它是*OSError* 的别名。

如果 sys.stdin 被使用超过一次，则第二次之后的使用将不返回任何行，除非是被交互式的使用，或都是被显式地重置 (例如使用 sys.stdin.seek(0))。

空文件打开后将立即被关闭；它们在文件列表中会被注意到的唯一情况只有当最后打开的文件为空的时候。

反回的行不会对换行符做任何处理，这意味着文件中的最后一行可能不带换行符。

你可以通过将 *openhook* 形参传给*fileinput.input()* 或*FileInput()* 来提供一个打开钩子以控制文件的打开方式。此钩子必须为一个函数，它接受两个参数，*filename* 和 *mode*，并返回一个以相应模式打开的文件型对象。如果指定了 *encoding* 和/或 *errors*，它们将作为额外的关键字参数被传给这个钩子。此模块提供了一个*hook_compressed()* 来支持压缩文件。

以下函数是此模块的初始接口：

fileinput.**input**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

创建一个*FileInput* 类的实例。该实例将被用作此模块中函数的全局状态，并且还将在迭代期间被返回使用。此函数的形参将被继续传递给*FileInput* 类的构造器。

FileInput 实例可以在 with 语句中被用作上下文管理器。在这个例子中，*input* 在 with 语句结束后将会被关闭，即使发生了异常也是如此:

```
with fileinput.input(files=('spam.txt', 'eggs.txt'), encoding="utf-8") as f:
    for line in f:
        process(line)
```

在 3.2 版本发生变更: 可以被用作上下文管理器。

在 3.8 版本发生变更: 关键字形参 *mode* 和 *openhook* 现在是仅限关键字形参。

在 3.10 版本发生变更: 增加了仅限关键字形参 *encoding* 和 *errors*。

下列函数会使用*fileinput.input()* 所创建的全局状态；如果没有活动的状态，则会引发*RuntimeError*。

fileinput.**filename**()

返回当前被读取的文件名。在第一行被读取之前，返回 None。

fileinput.**fileno**()

返回以整数表示的当前文件 “文件描述符”。当未打开文件时（处在第一行和文件之间），返回 -1。

fileinput.**lineno**()

返回已被读取的累计行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回该行的行号。

fileinput.**filelineno**()

返回当前文件中的行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回此文件中该行的行号。

fileinput.**isfirstline**()

如果刚读取的行是其所在文件的第一行则返回 True，否则返回 False。

fileinput.**isstdin**()

如果最后读取的行来自 sys.stdin 则返回 True，否则返回 False。

fileinput.**nextfile**()

关闭当前文件以使下次迭代将从下一个文件（如果存在）读取第一行；不是从该文件读取的行将不会被计入累计行数。直到下一个文件的第一行被读取之后文件名才会改变。在第一行被读取之前，此函数将不会生效；它不能被用来跳过第一个文件。在最后一个文件的最后一行被读取之后，此函数将不再生效。

fileinput.**close**()

关闭序列。

此模块所提供的实现了序列行为的类同样也可用于子类化：

class fileinput.**FileInput**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

类*FileInput* 是具体的实现；它的方法*filename()*，*fileno()*，*lineno()*，*filelineno()*，*isfirstline()*，*isstdin()*，*nextfile()* 和*close()* 对应于此模块具有相同名称的函数。此外它还是一个*iterable* 并且具有可返回下一个输入行的*readline()* 方法。此序列必须以严格的序列顺序来访问；随机访问和*readline()* 不可被混用。

通过 *mode* 你可以指定要传给*open()* 的文件模式。它必须为 `'r'` 和 `'rb'` 中的一个。

openhook 如果给出则必须为一个函数，它接受两个参数 *filename* 和 *mode*，并相应地返回一个打开的文件型对象。你不能同时使用 *inplace* 和 *openhook*。

你可以指定 *encoding* 和 *errors* 来将其传给*open()* 或 *openhook*。

FileInput 实例可以在 `with` 语句中被用作上下文管理器。在这个例子中，*input* 在 `with` 语句结束后将会被关闭，即使发生了异常也是如此：

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

在 3.2 版本发生变更: 可以被用作上下文管理器。

在 3.8 版本发生变更: 关键字形参 *mode* 和 *openhook* 现在是仅限关键字形参。

在 3.10 版本发生变更: 增加了仅限关键字形参 *encoding* 和 *errors*。

在 3.11 版本发生变更: `'rU'` 和 `'U'` 模式以及 `__getitem__()` 方法已被移除。

可选的原地过滤: 如果传递了关键字参数 `inplace=True` 给*fileinput.input()* 或*FileInput* 构造器，则文件会被移至备份文件并将标准输出定向到输入文件（如果已存在与备份文件同名的文件，它将被静默地替换）。这使得编写一个能够原地重写其输入文件的过滤器成为可能。如果给出了 *backup* 形参 (通常形式为 `backup='.<some extension>'`)，它将指定备份文件的扩展名，并且备份文件会被保留；默认情况下扩展名为 `'.bak'` 并且它会在输出文件关闭时被删除。在读取标准输入时原地过滤会被禁用。

此模块提供了以下两种打开文件钩子：

fileinput.**hook_compressed**(*filename*, *mode*, ***, *encoding=None*, *errors=None*)

使用*gzip* 和*bz2* 模块透明地打开 gzip 和 bzip2 压缩的文件（通过扩展名 `'.gz'` 和 `'.bz2'` 来识别）。如果文件扩展名不是 `'.gz'` 或 `'.bz2'`，文件会以正常方式打开（即使用*open()* 并且不带任何解压操作）。

encoding 和 *errors* 值会被传给*io.TextIOWrapper* 用于压缩文件以及打开普通文件。

用法示例: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed, encoding="utf-8")`

在 3.10 版本发生变更: 增加了仅限关键字形参 *encoding* 和 *errors*。

fileinput.**hook_encoded**(*encoding*, *errors=None*)

返回一个通过*open()* 打开每个文件的钩子，使用给定的 *encoding* 和 *errors* 来读取文件。

使用示例: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

在 3.6 版本发生变更: 添加了可选的 *errors* 形参。

自 3.10 版本弃用: 此函数已被弃用，因为*fileinput.input()* 和*FileInput* 现在有了 *encoding* 和 *errors* 形参。

17.5 curses --- 字符单元显示的终端处理

源代码: Lib/curses

---

curses 模块提供了 curses 库的接口，这是可移植高级终端处理的事实标准。

虽然 curses 在 Unix 环境中使用最为广泛，但也有适用于 Windows，DOS 以及其他可能的系统的版本。此扩展模块旨在匹配 ncurses 的 API，这是一个部署在 Linux 和 Unix 的 BSD 变体上的开源 curses 库。

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或*WebAssembly 平台* 上不受支持。

备注

每当文档提到 **字符**时，它可以被指定为一个整数，一个单字符 Unicode 字符串或者一个单字节的字节字符串。

每当此文档提到 **字符串**时，它可以被指定为一个 Unicode 字符串或者一个字节字符串。

参见

模块*curses.ascii*
在 ASCII 字符上工作的工具，无论你的区域设置是什么。

模块*curses.panel*
为 curses 窗口添加深度的面板栈扩展。

模块*curses.textpad*
用于使 curses 支持 **Emacs** 式绑定的可编辑文本部件。

curses-howto
关于配合 Python 使用 curses 的教学材料，由 Andrew Kuchling 和 Eric Raymond 撰写。

17.5.1 函数

curses 模块定义了以下异常:

exception curses.**error**
当 curses 库中函数返回一个错误时引发的异常。

备注

只要一个函数或方法的 *x* 或 *y* 参数是可选项，它们会默认为当前光标位置。而当 *attr* 是可选项时，它会默认为*A_NORMAL*。

curses 模块定义了以下函数:

curses.**baudrate**()
以每秒比特数为单位返回终端输出速度。在软件终端模拟器上它将具有一个固定的最高值。此函数出于历史原因被包括；在以前，它被用于写输出循环以提供时间延迟，并偶尔根据线路速度来改变接口。

curses.**beep**()

发出短促的提醒声音。

curses.**can_change_color**()

根据程序员能否改变终端显示的颜色返回 True 或 False。

curses.**cbreak**()

进入 cbreak 模式。在 cbreak 模式（有时也称为“稀有”模式）通常的 tty 行缓冲会被关闭并且字符可以被一个一个地读取。但是，与原始模式不同，特殊字符（中断、退出、挂起和流程控制）会在 tty 驱动和调用程序上保留其效果。首先调用*raw()* 然后调用*cbreak()* 会将终端置于 cbreak 模式。

curses.**color_content**(*color_number*)

返回颜色值 *color_number* 中红、绿和蓝（RGB）分量的强度，此强度值必须介于 0 和 COLORS - 1 之间。返回一个 3 元组，其中包含给定颜色的 R,G,B 值，它们必须介于 0 (无分量) 和 1000 (最大分量) 之间。

curses.**color_pair**(*pair_number*)

返回用于以指定颜色对显示文本的属性值。仅支持前 256 个颜色对。该属性值可与*A_STANDOUT*, *A_REVERSE* 以及其他 A_* 属性组合使用。*pair_number()* 是此函数的对应操作。

curses.**curs_set**(*visibility*)

设置光标状态。*visibility* 可设为 0, 1 或 2 表示不可见、正常与高度可见。如果终端支持所请求的可见性，则返回之前的光标状态；否则会引发异常。在许多终端上，“正常可见”模式为下划线光标而“高度可见”模式为方块形光标。

curses.**def_prog_mode**()

将当前终端模式保存为”program” 模式，即正在运行的程序使用 curses 的模式。(与其相对的是”shell” 模式，即程序不使用 curses。）对*reset_prog_mode()* 的后续调用将恢复此模式。

curses.**def_shell_mode**()

将当前终端模式保存为”shell” 模式，即正在运行的程序不使用 curses 的模式。（与其相对的是”program” 模式，即程序使用功能。）对*reset_shell_mode()* 的后续调用将恢复此模式。

curses.**delay_output**(*ms*)

在输出中插入 *ms* 毫秒的暂停。

curses.**doupdate**()

更新物理屏幕。curses 库会保留两个数据结构，一个代表当前物理屏幕的内容以及一个虚拟屏幕代表需要的后续状态。*doupdate()* 整体更新物理屏幕以匹配虚拟屏幕。

虚拟屏幕可以通过在写入操作例如在一个窗口上执行*addstr()* 之后调用*noutrefresh()* 来刷新。普通的*refresh()* 调用只是简单的 noutrefresh() 加 doupdate()；如果你需要更新多个窗口，你可以通过在所有窗口上发出 noutrefresh() 调用再加单次 doupdate() 来提升性能并可减少屏幕闪烁。

curses.**echo**()

进入 echo 模式。在 echo 模式下，输入的每个字符都会在输入后回显到屏幕上。

curses.**endwin**()

撤销库的初始化，使终端返回正常状态。

curses.**erasechar**()

将用户的当前擦除字符以单字节字节串对象的形式返回。在 Unix 操作系统下这是 curses 程序用来控制 tty 的属性，而不是由 curses 库本身来设置的。

curses.**filter**()

如果要使用*filter()* 例程，它必须在调用*initscr()* 之前被调用。其效果是在这些调用期间，LINES 会被设为 1；clear, cup, cud, cud1, cuu1, cuu, vpa 等功能会被禁用；而 home 字符串会被设为 cr 的值。其影响是光标会被限制在当前行内，屏幕刷新也是如此。这可被用于启用单字符模式的行编辑而不触及屏幕的其余部分。

curses.**flash**()

闪烁屏幕。也就是将其改为反显并在很短的时间内将其改回原状。有些人更喜欢这样的‘视觉响铃’而非*beep()* 所产生的听觉提醒信号。

curses.**flushinp**()

刷新所有输入缓冲区。这会丢弃任何已被用户输入但尚未被程序处理的预输入内容。

curses.**getmouse**()

在*getch()* 返回*KEY_MOUSE* 以发出鼠标事件信号之后，应当调用此方法来获取加入队列的鼠标事件，事件以一个 5 元组 (id, x, y, z, bstate) 来表示。其中 *id* 为用于区分多个设备的 ID 值，而 *x*, *y*, *z* 为事件的坐标。(*z* 目前未被使用。) *bstate* 为一个整数值，其各个比特位将被设置用来表示事件的类型，并将为下列常量中的一个或多个按位 OR 的结果，其中 *n* 是以 1 到 5 表示的键号: *BUTTONn_PRESSED*, *BUTTONn_RELEASED*, *BUTTONn_CLICKED*, *BUTTONn_DOUBLE_CLICKED*, *BUTTONn_TRIPLE_CLICKED*, *BUTTON_SHIFT*, *BUTTON_CTRL*, *BUTTON_ALT*。

在 3.10 版本发生变更: 现在 BUTTON5_* 常量如果是由下层 curses 库提供的则会对外公开。

curses.**getsyx**()

将当前虚拟屏幕光标的坐标作为元组 (y, x) 返回。如果*leaveok* 当前为 True，则返回 (-1, -1)。

curses.**getwin**(*file*)

读取由之前的*window.putwin()* 调用存储在文件中的窗口相关数据。该例程随后将使用该数据创建并初始化一个新窗口，并返回这个新窗口对象。

curses.**has_colors**()

如果终端能显示彩色则返回 True；否则返回 False。

curses.**has_extended_color_support**()

如果此模块支持扩展颜色则返回 True；否则返回 False。扩展颜色支持允许支持超过 16 种颜色的终端（例如 xterm-256color）支持超过 256 种颜色对。

扩展颜色支持要求 ncurses 版本为 6.1 或更新。

Added in version 3.10.

curses.**has_ic**()

如果终端具有插入和删除字符的功能则返回 True。此函数仅是出于历史原因而被包括的，因为所有现代软件终端模拟器都具有这些功能。

curses.**has_il**()

如果终端具有插入和删除字符功能，或者能够使用滚动区域来模拟这些功能则返回 True。此函数仅是出于历史原因而被包括的，因为所有现代软件终端模拟器都具有这些功能。

curses.**has_key**(*ch*)

接受一个键值 *ch*，并在当前终端类型能识别出具有该值的键时返回 True。

curses.**halfdelay**(*tenths*)

用于半延迟模式，与 cbreak 模式的类似之处是用户所键入的字符会立即对程序可用。但是，在阻塞 *tenths* 个十分之一秒之后，如果还未输入任何内容则将引发异常。*tenths* 值必须为 1 和 255 之间的数字。使用*nocbreak()* 可退出半延迟模式。

curses.**init_color**(*color_number*, *r*, *g*, *b*)

更改某个颜色的定义，接受要更改的颜色编号以及三个 RGB 值（表示红绿蓝三个分量的强度）。*color_number* 的值必须为 0 和 COLORS - 1 之间的数字。每个 *r*, *g*, *b* 值必须为 0 和 1000 之间的数字。当使用*init_color()* 时，出现在屏幕上的对应颜色会立即按照定定义来更改。此函数在大多数终端上都是无操作的；它仅会在*can_change_color()* 返回 True 时生效。

curses.**init_pair**(*pair_number*, *fg*, *bg*)

更改某个颜色对的定义。它接受三个参数：要更改的颜色对编号，前景色编号和背景色编号。*pair_number* 值必须为 1 和 COLOR_PAIRS - 1 之间的数字（并且 0 号颜色对固定为黑底白字而无法更改）。*fg* 和 *bg* 参数必须为 0 和 COLORS - 1 之间的数字，或者在调用*use_default_colors()* 之后则为 -1。如果颜色对之前已被初始化，则屏幕会被刷新使得出现在屏幕上的该颜色会立即按照新定义来更改。

curses.**initscr**()

初始化库。返回代表整个屏幕的窗口 对象。

> **备注**
>
> 如果打开终端时发生错误，则下层的 curses 库可能会导致解释器退出。

curses.**is_term_resized**(*nlines*, *ncols*)

如果*resize_term()* 会修改窗口结构则返回 `True`，否则返回 `False`。

curses.**isendwin**()

如果*endwin()* 已经被调用（即 curses 库已经被撤销初始化则返回 `True`。

curses.**keyname**(*k*)

将编号为 *k* 的键名称作为字节串对象返回。生成可打印 ASCII 字符的键名称就是键所对应的字符。Ctrl-键组合的键名称则是一个两字节的字节串对象，它由插入符 (`b'^'`) 加对应的可打印 ASCII 字符组成。Alt-键组合 (128--255) 的键名称则是由前缀 `b'M-'` 加对应的可打印 ASCII 字符组成的字节串对象。

curses.**killchar**()

将用户的当前行删除字符以单字节字节串对象的形式返回。在 Unix 操作系统下这是 curses 程序用来控制 tty 的属性，而不是由 curses 库本身来设置的。

curses.**longname**()

返回一个字节串对象，其中包含描述当前终端的 terminfo 长名称字段。详细描述的最大长度为 128 个字符。它仅在调用*initscr()* 之后才会被定义。

curses.**meta**(*flag*)

如果 *flag* 为 `True`，则允许输入 8 比特位的字符。如果 *flag* 为 `False`，则只允许 7 比特位的字符。

curses.**mouseinterval**(*interval*)

以毫秒为单位设置能够被识别为点击的按下和事件事件之间可间隔的最长时间，并返回之前的间隔值。默认值为 200 毫秒，即五分之一秒。

curses.**mousemask**(*mousemask*)

设置要报告的鼠标事件，并返回一个元组 `(availmask, oldmask)`。*availmask* 表明指定的鼠标事件中哪些可以被报告；当完全失败时将返回 `0`。*oldmask* 是给定窗口的鼠标事件之前的掩码值。如果从未调用此函数，则不会报告任何鼠标事件。

curses.**napms**(*ms*)

休眠 *ms* 毫秒。

curses.**newpad**(*nlines*, *ncols*)

创建并返回一个指向具有给定行数和列数新的面板数据结构的指针。将面板作为窗口对象返回。

面板类似于窗口，区别在于它不受屏幕大小的限制，并且不必与屏幕的特定部分相关联。面板可以在需要使用大窗口时使用，并且每次只需将窗口的一部分放在屏幕上。面板不会发生自动刷新（例如由于滚动或输入回显）。面板的*refresh()* 和*noutrefresh()* 方法需要 6 个参数来指定面板要显示的部分以及要用于显示的屏幕位置。这些参数是 *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*；*p* 参数表示要显示的面板区域的左上角而 *s* 参数定义了要显示的面板区域在屏幕上的剪切框。

curses.**newwin**(*nlines*, *ncols*)

curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

返回一个新的窗口，其左上角位于 `(begin_y, begin_x)`，并且其高度/宽度为 *nlines*/*ncols*。

默认情况下，窗口将从指定位置扩展到屏幕的右下角。

curses.**nl**()

进入 newline 模式。此模式会在输入时将回车转换为换行符，并在输出时将换行符转换为回车加换行。newline 模式会在初始时启用。

curses.**nocbreak**()

退出 cbreak 模式。返回具有行缓冲的正常”cooked” 模式。

curses.**noecho**()

退出 echo 模式。关闭输入字符的回显。

curses.**nonl**()

退出 newline 模式。停止在输入时将回车转换为换行，并停止在输出时从换行到换行/回车的底层转换（但这不会改变 `addch('\n')` 的行为，此行为总是在虚拟屏幕上执行相当于回车加换行的操作）。当停止转换时，curses 有时能使纵向移动加快一些；并且，它将能够在输入时检测回车键。

curses.**noqiflush**()

当使用 `noqiflush()` 例程时，与 `INTR`, `QUIT` 和 `SUSP` 字符相关联的输入和输出队列的正常刷新将不会被执行。如果你希望在处理程序退出后还能继续输出，就像没有发生过中断一样，你可能会想要在信号处理程序中调用 `noqiflush()`。

curses.**noraw**()

退出 raw 模式。返回具有行缓冲的正常”cooked” 模式。

curses.**pair_content**(*pair_number*)

返回包含对应于所请求颜色对的元组 `(fg, bg)`。*pair_number* 的值必须在 `0` 和 `COLOR_PAIRS - 1` 之间。

curses.**pair_number**(*attr*)

返回通过属性值 *attr* 所设置的颜色对的编号。*color_pair()* 是此函数的对应操作。

curses.**putp**(*str*)

等价于 `tputs(str, 1, putchar)`；为当前终端发出指定 terminfo 功能的值。请注意*putp()* 的输出总是前往标准输出。

curses.**qiflush**([*flag*])

如果 *flag* 为 `False`，则效果与调用*noqiflush()* 相同。如果 *flag* 为 `True` 或未提供参数，则在读取这些控制字符时队列将被刷新。

curses.**raw**()

进入 raw 模式。在 raw 模式下，正常的行缓冲和对中断、退出、挂起和流程控制键的处理会被关闭；字符会被逐个地提交给 curses 输入函数。

curses.**reset_prog_mode**()

将终端恢复到”program” 模式，如之前由*def_prog_mode()* 所保存的一样。

curses.**reset_shell_mode**()

将终端恢复到”shell” 模式，如之前由*def_shell_mode()* 所保存的一样。

curses.**resetty**()

将终端模式恢复到最后一次调用*savetty()* 时的状态。

curses.**resize_term**(*nlines*, *ncols*)

由*resizeterm()* 用来执行大部分工作的后端函数；当调整窗口大小时，*resize_term()* 会以空白填充扩展区域。调用方应用程序应当以适当的数据填充这些区域。`resize_term()` 函数会尝试调整所有窗口的大小。但是，由于面板的调用约定，在不与应用程序进行额外交互的情况下是无法调整其大小的。

curses.**resizeterm**(*nlines*, *ncols*)

将标准窗口和当前窗口的大小调整为指定的尺寸，并调整由 curses 库所使用的记录窗口尺寸的其他记录数据（特别是 SIGWINCH 处理程序）。

curses.**savetty**()

将终端模式的当前状态保存在缓冲区中，可供*resetty()* 使用。

curses.**get_escdelay**()

提取通过*set_escdelay()* 设置的值。

Added in version 3.9.

curses.**set_escdelay**(*ms*)

设置读取一个转义字符后要等待的毫秒数，以区分在键盘上输入的单个转义字符与通过光标和功能键发送的转义序列。

Added in version 3.9.

curses.**get_tabsize**()

提取通过*set_tabsize()* 设置的值。

Added in version 3.9.

curses.**set_tabsize**(*size*)

设置 curses 库在将制表符添加到窗口时将制表符转换为空格所使用的列数。

Added in version 3.9.

curses.**setsyx**(*y*, *x*)

将虚拟屏幕光标设置到 *y*, *x*。如果 *y* 和 *x* 均为 `-1`，则*leaveok* 将设为 `True`。

curses.**setupterm**(*term=None*, *fd=-1*)

初始化终端。*term* 为给出终端名称的字符串或为 `None`；如果省略或为 `None`，则将使用 `TERM` 环境变量的值。*fd* 是任何初始化序列将被发送到的文件描述符；如未指定或为 `-1`，则将使用 `sys.stdout` 的文件描述符。

curses.**start_color**()

如果程序员想要使用颜色，则必须在任何其他颜色操作例程被调用之前调用它。在*initscr()* 之后立即调用此例程是一个很好的做法。

start_color() 会初始化八种基本颜色（黑、红、绿、黄、蓝、品、青和白）以及*curses* 模块中的两个全局变量*COLORS* 和*COLOR_PAIRS*，其中包含终端可支持的颜色和颜色对的最大数量。它还会将终端中的颜色恢复为终端刚启动时的值。

curses.**termattrs**()

返回终端所支持的所有视频属性逻辑 OR 的值。此信息适用于当 curses 程序需要对屏幕外观进行完全控制的情况。

curses.**termname**()

将环境变量 `TERM` 的值截短至 14 个字节，作为字节串对象返回。

curses.**tigetflag**(*capname*)

将与 terminfo 功能名称 *capname* 相对应的布尔功能值以整数形式返回。如果 *capname* 不是一个布尔功能则返回 `-1`，如果其被取消或不存在于终端描述中则返回 `0`。

curses.**tigetnum**(*capname*)

将与 terminfo 功能名称 *capname* 相对应的数字功能值以整数形式返回。如果 *capname* 不是一个数字功能则返回 `-2`，如果其被取消或不存在于终端描述中则返回 `-1`。

curses.**tigetstr**(*capname*)

将与 terminfo 功能名称 *capname* 相对应的字符串功能值以字节串对象形式返回。如果 *capname* 不是一个 terminfo ” 字符串功能” 或者如果其被取消或不存在于终端描述中则返回 `None`。

curses.**tparm**(*str*[, ...])

使用提供的形参初始化字节串对象 *str*，其中 *str* 应当是从 terminfo 数据库获取的参数化字符串。例如 `tparm(tigetstr("cup"), 5, 3)` 的结果可能为 `b'\033[6;4H'`，实际结果将取决于终端类型。

curses.**typeahead**(*fd*)

指定将被用于预输入检查的文件描述符 *fd*。如果 *fd* 为 -1，则不执行预输入检查。

curses 库会在更新屏幕时通过定期查找预输入来执行" 断行优化"。如果找到了输入，并且输入是来自于 tty，则会将当前更新推迟至 refresh 或 doupdate 再次被调用的时候，以便允许更快地响应预先输入的命令。此函数允许为预输入检查指定其他的文件描述符。

curses.**unctrl**(*ch*)

返回一个字节串对象作为字符 *ch* 的可打印表示形式。控制字符会表示为一个变换符加相应的字符，例如 `b'^C'`。可打印字符则会保持原样。

curses.**ungetch**(*ch*)

推送 *ch* 以便让下一个*getch()* 返回该字符。

> **备注**
>
> 在 getch() 被调用之前只能推送一个 *ch*。

curses.**update_lines_cols**()

更新*LINES* 和*COLS* 模块变量。适用于检测手动调整屏幕大小。

Added in version 3.5.

curses.**unget_wch**(*ch*)

推送 *ch* 以便让下一个*get_wch()* 返回该字符。

> **备注**
>
> 在 get_wch() 被调用之前只能推送一个 *ch*。

Added in version 3.3.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)

将*KEY_MOUSE* 事件推送到输入队列，将其与给定的状态数据进行关联。

curses.**use_env**(*flag*)

如果使用此函数，则应当在调用*initscr()* 或 newterm 之前调用它。当 *flag* 为 False 时，将会使用在 terminfo 数据库中指定的行和列的值，即使设置了环境变量 LINES 和 COLUMNS (默认使用)，或者如果 curses 是在窗口中运行（在此情况下如果未设置 LINES 和 COLUMNS 则默认行为将是使用窗口大小）。

curses.**use_default_colors**()

允许在支持此特性的终端上使用默认的颜色值。使用此函数可在你的应用程序中支持透明效果。默认颜色会被赋给颜色编号 -1。举例来说，在调用此函数后，`init_pair(x, curses.COLOR_RED, -1)` 会将颜色对 *x* 初始化为红色前景和默认颜色背景。

curses.**wrapper**(*func*, */*, **args*, ***kwargs*)

初始化 curses 并调用另一个可调用对象 *func*，该对象应当为你的使用 curses 的应用程序的其余部分。如果应用程序引发了异常，此函数将在重新引发异常并生成回溯信息之前将终端恢复到正常状态。随后可调用对象 *func* 会被传入主窗口'stdscr' 作为其第一个参数，再带上其他所有传给 wrapper() 的参数。在调用 *func* 之前，wrapper() 会启用 cbreak 模式，关闭回显，启用终端键盘，并在终端具有颜色支持的情况下初始化颜色。在退出时（无论是正常退出还是异常退出）它会恢复 cooked 模式，打开回显，并禁用终端键盘。

17.5.2 Window 对象

Window 对象会由上面的*initscr()* 和*newwin()* 返回，它具有以下方法和属性:

window.**addch**(*ch*[, *attr*])

window.**addch**(*y*, *x*, *ch*[, *attr*])

将带有属性 *attr* 的字符 *ch* 绘制到 (y, x)，覆盖之前在该位置上绘制的任何字符。默认情况下，字符的位置和属性均为窗口对象的当前设置。

> **备注**
>
> 在窗口、子窗口或面板之外写入会引发*curses.error*。尝试在窗口、子窗口或面板的右下角写入将在字符被打印之后导致异常被引发。

window.**addnstr**(*str*, *n*[, *attr*])

window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])

将带有属性 *attr* 的字符串 *str* 中的至多 *n* 个字符绘制到 (y, x)，覆盖之前在屏幕上的任何内容。

window.**addstr**(*str*[, *attr*])

window.**addstr**(*y*, *x*, *str*[, *attr*])

将带有属性 *attr* 的字符串 *str* 绘制到 (y, x)，覆盖之前在屏幕上的任何内容。

> **备注**
>
> - 在窗口、子窗口或面板之外写入会引发*curses.error*。尝试在窗口、子窗口或面板的右下角写入将在字符串被打印之后导致异常被引发。
> - 此 Python 模块的后端 ncurses 中的一个缺陷 会在调整窗口大小时导致段错误。此缺陷已在 ncurses-6.1-20190511 中被修复。如果你必须使用较早版本的 ncurses，则你只要在调用*addstr()* 时不传入嵌入了换行符的 *str* 即可避免触发此错误。请为每一行分别调用*addstr()*。

window.**attroff**(*attr*)

从应用于写入到当前窗口的”background” 集中移除属性 *attr*。

window.**attron**(*attr*)

向应用于写入到当前窗口的”background” 集中添加属性 *attr*。

window.**attrset**(*attr*)

将”background” 属性集设为 *attr*。该集合初始时为 0 (无属性)。

window.**bkgd**(*ch*[, *attr*])

将窗口 background 特征属性设为带有属性 *attr* 的字符 *ch*。随后此修改将应用于放置到该窗口中的每个字符。

- 窗口中每个字符的属性会被修改为新的 background 属性。
- 不论之前的 background 字符出现在哪里，它都会被修改为新的 background 字符。

window.**bkgdset**(*ch*[, *attr*])

设置窗口的背景。窗口的背景由字符和属性的任意组合构成。背景的属性部分会与写入窗口的所有非空白字符合并（即 OR 运算）。背景和字符和属性部分均会与空白字符合并。背景将成为字符的特征属性并在任何滚动与插入/删除行/字符操作中与字符一起移动。

window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])

在窗口边缘绘制边框。每个参数指定用于边界特定部分的字符; 请参阅下表了解更多详情。

> **备注**
>
> 任何形参的值为 0 都将导致该形参使用默认字符。关键字形参 不可被使用。默认字符在下表中列出:

参数	描述	默认值
ls	左侧	*ACS_VLINE*
rs	右侧	*ACS_VLINE*
ts	顶部	*ACS_HLINE*
bs	底部	*ACS_HLINE*
tl	左上角	*ACS_ULCORNER*
tr	右上角	*ACS_URCORNER*
bl	左下角	*ACS_LLCORNER*
br	右下角	*ACS_LRCORNER*

window.**box**(*[vertch, horch]*)

类似于*border()*，但 *ls* 和 *rs* 均为 *vertch* 而 *ts* 和 *bs* 均为 *horch*。此函数总是会使用默认的转角字符。

window.**chgat**(*attr*)

window.**chgat**(*num*, *attr*)

window.**chgat**(*y*, *x*, *attr*)

window.**chgat**(*y*, *x*, *num*, *attr*)

在当前光标位置或是在所提供的位置 (y, x) 设置 *num* 个字符的属性。如果 *num* 未给出或为 -1，则将属性设置到所有字符上直至行尾。如果提供了位置 (y, x) 则此函数会将光标移至该位置。修改过的行将使用*touchline()* 方法处理以便下次窗口刷新时内容会重新显示。

window.**clear**()

类似于*erase()*，但还会导致在下次调用*refresh()* 时整个窗口被重新绘制。

window.**clearok**(*flag*)

如果 *flag* 为 True，则在下次调用*refresh()* 时将完全清除窗口。

window.**clrtobot**()

从光标位置开始擦除直至窗口末端：光标以下的所有行都会被删除，然后会执行*clrtoeol()* 的等效操作。

window.**clrtoeol**()

从光标位置开始擦除直至行尾。

window.**cursyncup**()

更新窗口所有上级窗口的当前光标位置以反映窗口的当前光标位置。

window.**delch**(*[y, x]*)

删除位于 (y, x) 的任何字符。

window.**deleteln**()

删除在光标之下的行。所有后续的行都会上移一行。

window.**derwin**(*begin_y*, *begin_x*)

window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

”derive window” 的缩写，*derwin()* 与调用*subwin()* 等效，不同之处在于 *begin_y* 和 *begin_x* 是想对于窗口的初始位置，而不是相对于整个屏幕。返回代表所派生窗口的窗口对象。

window.**echochar**(*ch*[, *attr*])

使用属性 *attr* 添加字符 *ch*，并立即在窗口上调用*refresh()*。

window.**enclose**(*y*, *x*)

检测给定的相对屏幕的字符-单元格坐标是否被给定的窗口所包围，返回 `True` 或 `False`。它适用于确定是哪个屏幕窗口子集包围着某个鼠标事件的位置。

在 3.10 版本发生变更: 在之前版本中它会返回 `1` 或 `0` 而不是 `True` 或 `False`。

window.**encoding**

用于编码方法参数（Unicode 字符串和字符）的编码格式。encoding 属性是在创建子窗口时从父窗口继承的，例如通过*window.subwin()*。在默认情况下，会使用当前语言区域的编码格式（参见*locale.getencoding()* 文档）。

Added in version 3.3.

window.**erase**()

清空窗口。

window.**getbegyx**()

返回以左上角为原点的坐标元组 `(y, x)`。

window.**getbkgd**()

返回给定窗口的当前背景字符/属性对。

window.**getch**([*y*, *x*])

获取一个字符。请注意所返回的整数 *不一定要在* ASCII 范围以内：功能键、小键盘键等等是由大于 255 的数字表示的。在无延迟模式下，如果没有输入则返回 `-1`，在其他情况下都会等待直至有键被按下。

window.**get_wch**([*y*, *x*])

获取一个宽字符。对于大多数键都是返回一个字符，对于功能键、小键盘键和其他特殊键则是返回一个整数。在无延迟模式下，如果没有输入则引发一个异常。

Added in version 3.3.

window.**getkey**([*y*, *x*])

获取一个字符，返回一个字符串而不是像*getch()* 那样返回一个整数。功能键、小键盘键和其他特殊键则是返回一个包含键名的多字节字符串。在无延迟模式下，如果没有输入则引发一个异常。

window.**getmaxyx**()

返回窗口高度和宽度的元组 `(y, x)`。

window.**getparyx**()

将此窗口相对于父窗口的起始坐标作为元组 `(y, x)` 返回。如果此窗口没有父窗口则返回 `(-1, -1)`。

window.**getstr**()

window.**getstr**(*n*)

window.**getstr**(*y*, *x*)

window.**getstr**(*y*, *x*, *n*)

从用户读取一个字节串对象，附带基本的行编辑功能。

window.**getyx**()

返回当前光标相对于窗口左上角的位置的元组 `(y, x)`。

window.**hline**(*ch*, *n*)

window.**hline**(*y*, *x*, *ch*, *n*)

显示一条起始于 `(y, x)` 长度为 *n* 个字符 *ch* 的水平线。

window.**idcok**(*flag*)

如果 *flag* 为 `False`，curses 将不再考虑使用终端的硬件插入/删除字符功能；如果 *flag* 为 `True`，则会启用字符插入和删除。当 curses 首次初始化时，默认会启用字符插入/删除。

window.**idlok**(*flag*)

如果 *flag* 为 True，*curses* 将尝试使用硬件行编辑功能。否则，行插入/删除会被禁用。

window.**immedok**(*flag*)

如果 *flag* 为 True，窗口图像中的任何改变都会自动导致窗口被刷新；你不必再自己调用*refresh()*。但是，这可能会由于重复调用 wrefresh 而显著降低性能。此选项默认被禁用。

window.**inch**([*y*, *x*])

返回窗口中给定位置上的字符。下面的 8 个比特位是字符本身，上面的比特位则为属性。

window.**insch**(*ch*[, *attr*])

window.**insch**(*y*, *x*, *ch*[, *attr*])

将带有属性 *attr* 的字符 *ch* 绘制到 (y, x)，将该行从位置 *x* 开始右移一个字符。

window.**insdelln**(*nlines*)

在指定窗口的当前行上方插入 *nlines* 行。下面的 *nlines* 行将丢失。对于 *nlines* 为负值的情况，则从光标下方的行开始删除 *nlines* 行，并将其余的行向上移动。下面的 *nlines* 行会被清空。当前光标位置将保持不变。

window.**insertln**()

在光标下方插入一个空行。所有后续的行都会下移一行。

window.**insnstr**(*str*, *n*[, *attr*])

window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])

在光标下方的字符之前插入一个至多为 *n* 个字符的字符串（字符数量将与该行相匹配）。如果 *n* 为零或负数，则插入整个字符串。光标右边的所有字符将被右移，该行右端的字符将丢失。光标位置将保持不变（在移到可能指定的 *y*, *x* 之后）。

window.**insstr**(*str*[, *attr*])

window.**insstr**(*y*, *x*, *str*[, *attr*])

在光标下方的字符之前插入一个字符串（字符数量将与该行相匹配）。光标右边的所有字符将被右移，该行右端的字符将丢失。光标位置将保持不变（在移到可能指定的 *y*, *x* 之后）。

window.**instr**([*n*])

window.**instr**(*y*, *x*[, *n*])

返回从窗口的当前光标位置，或者指定的 *y*, *x* 开始提取的字符所对应的字节串对象。属性会从字符中去除。如果指定了 *n*，*instr()* 将返回长度至多为 *n* 个字符的字符串（不包括末尾的 NUL）。

window.**is_linetouched**(*line*)

如果指定的行自上次调用*refresh()* 后发生了改变则返回 True；否则返回 False。如果 *line* 对于给定的窗口不可用则会引发*curses.error* 异常。

window.**is_wintouched**()

如果指定的窗口自上次调用*refresh()* 后发生了改变则返回 True；否则返回 False。

window.**keypad**(*flag*)

如果 *flag* 为 True，则某些键（小键盘键、功能键等）生成的转义序列将由*curses* 来解析。如果 *flag* 为 False，转义序列将保持在输入流中的原样。

window.**leaveok**(*flag*)

如果 *flag* 为 True，则在更新时光标将停留在原地，而不是在“光标位置”。这将可以减少光标的移动。在可能的情况下光标将变为不可见。

如果 *flag* 为 False，光标在更新后将总是位于“光标位置”。

window.**move**(*new_y*, *new_x*)

将光标移至 (new_y, new_x)。

window.**mvderwin**(*y*, *x*)

让窗口在其父窗口内移动。窗口相对于屏幕的参数不会被更改。此例程用于在屏幕的相同物理位置显示父窗口的不同部分。

window.**mvwin**(*new_y*, *new_x*)

移动窗口以使其左上角位于 (new_y, new_x)。

window.**nodelay**(*flag*)

如果 *flag* 为 True，则*getch()* 将为非阻塞的。

window.**notimeout**(*flag*)

如果 *flag* 为 True，则转义序列将不会发生超时。

如果 *flag* 为 False，则在几毫秒之后，转义序列将不会被解析，并将保持在输入流中的原样。

window.**noutrefresh**()

标记为刷新但保持等待。此函数会更新代表预期窗口状态的数据结构，但并不强制更新物理屏幕。要完成后者，请调用*doupdate()*。

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

将窗口覆盖在 *destwin* 上方。窗口的大小不必相同，只有重叠的区域会被复制。此复制是非破坏性的，这意味着当前背景字符不会覆盖掉 *destwin* 的旧内容。

为了获得对被复制区域的细粒度控制，可以使用*overlay()* 的第二种形式。*sminrow* 和 *smincol* 是源窗口的左上角坐标，而其他变量则在目标窗口中标记出一个矩形。

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

将窗口覆盖在 *destwin* 上方。窗口的大小不必相同，此时只有重叠的区域会被复制。此复制是破坏性的，这意味着当前背景字符会覆盖掉 *destwin* 的旧内容。

为了获得对被复制区域的细粒度控制，可以使用*overwrite()* 的第二种形式。*sminrow* 和 *smincol* 是源窗口的左上角坐标，而其他变量则在目标窗口中标记出一个矩形。

window.**putwin**(*file*)

将关联到窗口的所有数据写入到所提供的文件对象。此信息可在以后使用*getwin()* 函数来提取。

window.**redrawln**(*beg*, *num*)

指明从 *beg* 行开始的 *num* 个屏幕行已被破坏并且应当在下次*refresh()* 调用时完全重绘。

window.**redrawwin**()

触碰整个窗口，以使其在下次*refresh()* 调用时完全重绘。

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])

立即更新显示（将实际屏幕与之前的绘制/删除方法进行同步）。

6 个可选参数仅在窗口为使用*newpad()* 创建的面板时可被指定。需要额外的形参来指定所涉及到的是面板和屏幕的哪一部分。*pminrow* 和 *pmincol* 指定要在面板中显示的矩形的左上角。*sminrow*, *smincol*, *smaxrow* 和 *smaxcol* 指定要在屏幕中显示的矩形的边。要在面板中显示的矩形的右下角是根据屏幕坐标计算出来的，由于矩形的大小必须相同。两个矩形都必须完全包含在其各自的结构之内。负的 *pminrow*, *pmincol*, *sminrow* 或 *smincol* 值会被视为将它们设为零值。

window.**resize**(*nlines*, *ncols*)

为 curses 窗口重新分配存储空间以将其尺寸调整为指定的值。如果任一维度的尺寸大于当前值，则窗口的数据将以具有合并了当前背景渲染（由*bkgdset()* 设置）的空白来填充。

window.**scroll**([*lines=1*])

将屏幕或滚动区域向上滚动 *lines* 行。

window.**scrollok**(*flag*)

控制当一个窗口的光标移出窗口或滚动区域边缘时会发生什么，这可能是在底端行执行换行操作，或者在最后一行输入最后一个字符导致的结果。如果 *flag* 为 False，光标会留在底端行。如果 *flag* 为 True，窗口会向上滚动一行。请注意为了在终端上获得实际的滚动效果，还需要调用*idlok()*。

window.**setscrreg**(*top*, *bottom*)

设置从 *top* 行至 *bottom* 行的滚动区域。所有滚动操作将在此区域中进行。

window.**standend**()

关闭 standout 属性。在某些终端上此操作会有关闭所有属性的副作用。

window.**standout**()

启用属性 *A_STANDOUT*。

window.**subpad**(*begin_y*, *begin_x*)

window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)

返回一个子窗口，其左上角位于 `(begin_y, begin_x)`，并且其宽度/高度为 *ncols*/*nlines*。

window.**subwin**(*begin_y*, *begin_x*)

window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

返回一个子窗口，其左上角位于 `(begin_y, begin_x)`，并且其宽度/高度为 *ncols*/*nlines*。

默认情况下，子窗口将从指定位置扩展到窗口的右下角。

window.**syncdown**()

触碰已在上级窗口上被触碰的每个位置。此例程由*refresh()* 调用，因此几乎从不需要手动调用。

window.**syncok**(*flag*)

如果 *flag* 为 `True`，则*syncup()* 会在窗口发生改变的任何时候自动被调用。

window.**syncup**()

触碰已在窗口中被改变的此窗口的各个上级窗口中的所有位置。

window.**timeout**(*delay*)

为窗口设置阻塞或非阻塞读取行为。如果 *delay* 为负值，则会使用阻塞读取（这将无限期地等待输入）。如果 *delay* 为零，则会使用非阻塞读取，并且当没有输入在等待时*getch()* 将返回 `-1`。如果 *delay* 为正值，则*getch()* 将阻塞 *delay* 毫秒，并且当此延时结束时仍无输入将返回 `-1`。

window.**touchline**(*start*, *count*[, *changed*])

假定从行 *start* 开始的 *count* 行已被更改。如果提供了 *changed*，它将指明是将受影响的行标记为已更改 (*changed*=`True`) 还是未更改 (*changed*=`False`)。

window.**touchwin**()

假定整个窗口已被更改，其目的是用于绘制优化。

window.**untouchwin**()

将自上次调用*refresh()* 以来窗口中的所有行标记为未改变。

window.**vline**(*ch*, *n*[, *attr*])

window.**vline**(*y*, *x*, *ch*, *n*[, *attr*])

显示一行起始于 `(y, x)` 长度为 *n* 的由具有属性 *attr* 的字符 *ch* 组成的垂直线。

17.5.3 常量

curses 模块定义了以下数据成员:

curses.**ERR**

一些返回整数的 curses 例程，例如*getch()*，在失败时将返回*ERR*。

curses.**OK**

一些返回整数的 curses 例程，例如*napms()*，在成功时将返回*OK*。

curses.**version**

curses.**__version__**

一个代表模块当前版本的字节串对象。

curses.**ncurses_version**

一个具名元组，它包含构成 ncurses 库版本号的三个数字: *major*, *minor* 和 *patch*。三个值均为整数。三个值也可通过名称来访问，因此 `curses.ncurses_version[0]` 等价于 `curses.ncurses_version.major`，依此类推。

可用性：如果使用了 ncurses 库。

Added in version 3.8.

curses.**COLORS**

终端可支持的最大颜色数。它只有在调用 *start_color()* 之后才会被定义。

curses.**COLOR_PAIRS**

终端可支持的最大颜色对数。它只有在调用 *start_color()* 之后才会被定义。

curses.**COLS**

屏幕的宽度，即列数。它只有在调用 *initscr()* 之后才会被定义。可被 *update_lines_cols()*、*resizeterm()* 和 *resize_term()* 更新。

curses.**LINES**

屏幕的高度，即行数。它只有在调用 *initscr()* 之后才会被定义。可被 *update_lines_cols()*、*resizeterm()* 和 *resize_term()* 更新。

有些常量可用于指定字符单元属性。实际可用的常量取决于具体的系统。

属性	含意
curses.**A_ALTCHARSET**	备用字符集模式
curses.**A_BLINK**	闪烁模式
curses.**A_BOLD**	粗体模式
curses.**A_DIM**	暗淡模式
curses.**A_INVIS**	不可见或空白模式
curses.**A_ITALIC**	斜体模式
curses.**A_NORMAL**	正常属性
curses.**A_PROTECT**	保护模式
curses.**A_REVERSE**	反转背景色和前景色
curses.**A_STANDOUT**	突出模式
curses.**A_UNDERLINE**	下划线模式
curses.**A_HORIZONTAL**	水平突出显示
curses.**A_LEFT**	左高亮
curses.**A_LOW**	底部高亮
curses.**A_RIGHT**	右高亮
curses.**A_TOP**	顶部高亮
curses.**A_VERTICAL**	垂直突出显示

Added in version 3.7: A_ITALIC was added.

有几个常量可用于提取某些方法返回的相应属性。

位掩码	含意
curses.**A_ATTRIBUTES**	用于提取属性的位掩码
curses.**A_CHARTEXT**	用于提取字符的位掩码
curses.**A_COLOR**	用于提取颜色对字段信息的位掩码

键由名称以 KEY_ 开头的整数常量引用。确切的可用键取决于系统。

关键常数	键
curses.**KEY_MIN**	最小键值
curses.**KEY_BREAK**	中断键（不可靠）
curses.**KEY_DOWN**	向下箭头
curses.**KEY_UP**	向上箭头
curses.**KEY_LEFT**	向左箭头
curses.**KEY_RIGHT**	向右箭头
curses.**KEY_HOME**	Home 键 (上 + 左箭头)
curses.**KEY_BACKSPACE**	退格（不可靠）
curses.**KEY_F0**	功能键。支持至多 64 个功能键。
curses.**KEY_Fn**	功能键 *n* 的值
curses.**KEY_DL**	删除行
curses.**KEY_IL**	插入行
curses.**KEY_DC**	删除字符

续下页

表 1 – 接上页

关键常数	键
curses.**KEY_IC**	插入字符或进入插入模式
curses.**KEY_EIC**	退出插入字符模式
curses.**KEY_CLEAR**	清空屏幕
curses.**KEY_EOS**	清空至屏幕底部
curses.**KEY_EOL**	清空至行尾
curses.**KEY_SF**	向前滚动 1 行
curses.**KEY_SR**	向后滚动 1 行 (反转)
curses.**KEY_NPAGE**	下一页
curses.**KEY_PPAGE**	上一页
curses.**KEY_STAB**	设置制表符
curses.**KEY_CTAB**	清除制表符
curses.**KEY_CATAB**	清除所有制表符
curses.**KEY_ENTER**	回车或发送 (不可靠)
curses.**KEY_SRESET**	软 (部分) 重置 (不可靠)
curses.**KEY_RESET**	重置或硬重置 (不可靠)
curses.**KEY_PRINT**	打印
curses.**KEY_LL**	Home 向下或到底 (左下)
curses.**KEY_A1**	键盘的左上角

续下页

表 1 – 接上页

关键常数	键
curses.**KEY_A3**	键盘的右上角
curses.**KEY_B2**	键盘的中心
curses.**KEY_C1**	键盘左下方
curses.**KEY_C3**	键盘右下方
curses.**KEY_BTAB**	回退制表符
curses.**KEY_BEG**	Beg (开始)
curses.**KEY_CANCEL**	取消
curses.**KEY_CLOSE**	关闭
curses.**KEY_COMMAND**	Cmd (命令行)
curses.**KEY_COPY**	复制
curses.**KEY_CREATE**	创建
curses.**KEY_END**	End
curses.**KEY_EXIT**	退出
curses.**KEY_FIND**	查找
curses.**KEY_HELP**	帮助
curses.**KEY_MARK**	标记
curses.**KEY_MESSAGE**	消息
curses.**KEY_MOVE**	移动

续下页

表 1 – 接上页

关键常数	键
curses.**KEY_NEXT**	下一个
curses.**KEY_OPEN**	打开
curses.**KEY_OPTIONS**	选项
curses.**KEY_PREVIOUS**	Prev (上一个)
curses.**KEY_REDO**	重做
curses.**KEY_REFERENCE**	Ref (引用)
curses.**KEY_REFRESH**	刷新
curses.**KEY_REPLACE**	替换
curses.**KEY_RESTART**	重启
curses.**KEY_RESUME**	恢复
curses.**KEY_SAVE**	保存
curses.**KEY_SBEG**	Shift + Beg (开始)
curses.**KEY_SCANCEL**	Shift + Cancel
curses.**KEY_SCOMMAND**	Shift + Command
curses.**KEY_SCOPY**	Shift + Copy
curses.**KEY_SCREATE**	Shift + Create
curses.**KEY_SDC**	Shift + 删除字符
curses.**KEY_SDL**	Shift + 删除行

续下页

表 1 – 接上页

关键常数	键
curses.**KEY_SELECT**	选择
curses.**KEY_SEND**	Shift + End
curses.**KEY_SEOL**	Shift + 清空行
curses.**KEY_SEXIT**	Shift + 退出
curses.**KEY_SFIND**	Shift + 查找
curses.**KEY_SHELP**	Shift + 帮助
curses.**KEY_SHOME**	Shift + Home
curses.**KEY_SIC**	Shift + 输入
curses.**KEY_SLEFT**	Shift + 向左箭头
curses.**KEY_SMESSAGE**	Shift + 消息
curses.**KEY_SMOVE**	Shift + 移动
curses.**KEY_SNEXT**	Shift + 下一个
curses.**KEY_SOPTIONS**	Shift + 选项
curses.**KEY_SPREVIOUS**	Shift + 上一个
curses.**KEY_SPRINT**	Shift + 打印
curses.**KEY_SREDO**	Shift + 重做
curses.**KEY_SREPLACE**	Shift + 替换
curses.**KEY_SRIGHT**	Shift + 向右箭头

续下页

表 1 – 接上页

关键常数	键
curses.**KEY_SRSUME**	Shift + 恢复
curses.**KEY_SSAVE**	Shift + 保存
curses.**KEY_SSUSPEND**	Shift + 挂起
curses.**KEY_SUNDO**	Shift + 撤销
curses.**KEY_SUSPEND**	挂起
curses.**KEY_UNDO**	撤销操作
curses.**KEY_MOUSE**	鼠标事件已发生
curses.**KEY_RESIZE**	终端大小改变事件
curses.**KEY_MAX**	最大键值

在 VT100s 及其软件模拟器，如 X 终端模拟器上，通常至少有四个功能键 (*KEY_F1*, *KEY_F2*, *KEY_F3*, *KEY_F4*) 可用，并且方向键将明确地映射到*KEY_UP*, *KEY_DOWN*, *KEY_LEFT* 和*KEY_RIGHT*。如果你的机器有一个 PC 键盘，则保证能使用方向键和十二个功能键 (老式的 PC 键盘可能只有十个功能键)；此外，还有以下的标准小键盘映射:

键帽	常量
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

下表列出了替代字符集中的字符。这些字符继承自 VT100 终端，在 X 终端等软件模拟器上通常均为可用。当没有可用的图形时，curses 会回退为粗糙的可打印 ASCII 近似符号。

备注

只有在调用*initscr()* 之后才能使用它们

ACS 代码	含意
curses.**ACS_BBSS**	右上角的别名
curses.**ACS_BLOCK**	实心方块
curses.**ACS_BOARD**	正方形
curses.**ACS_BSBS**	水平线的别名
curses.**ACS_BSSB**	左上角的别名
curses.**ACS_BSSS**	顶部 T 型的别名
curses.**ACS_BTEE**	底部 T 型
curses.**ACS_BULLET**	正方形
curses.**ACS_CKBOARD**	棋盘（点刻）
curses.**ACS_DARROW**	向下箭头
curses.**ACS_DEGREE**	等级符
curses.**ACS_DIAMOND**	菱形
curses.**ACS_GEQUAL**	大于或等于
curses.**ACS_HLINE**	水平线
curses.**ACS_LANTERN**	灯形符号
curses.**ACS_LARROW**	向左箭头
curses.**ACS_LEQUAL**	小于或等于
curses.**ACS_LLCORNER**	左下角

续下页

表 2 – 接上页

ACS 代码	含意
curses.**ACS_LRCORNER**	右下角
curses.**ACS_LTEE**	左侧 T 型
curses.**ACS_NEQUAL**	不等号
curses.**ACS_PI**	字母 π
curses.**ACS_PLMINUS**	正负号
curses.**ACS_PLUS**	加号
curses.**ACS_RARROW**	向右箭头
curses.**ACS_RTEE**	右侧 T 型
curses.**ACS_S1**	扫描线 1
curses.**ACS_S3**	扫描线 3
curses.**ACS_S7**	扫描线 7
curses.**ACS_S9**	扫描线 9
curses.**ACS_SBBS**	右下角的别名
curses.**ACS_SBSB**	垂直线的别名
curses.**ACS_SBSS**	右侧 T 型的别名
curses.**ACS_SSBB**	左下角的别名
curses.**ACS_SSBS**	底部 T 型的别名
curses.**ACS_SSSB**	左侧 T 型的别名

续下页

表 2 – 接上页

ACS 代码	含意
curses.**ACS_SSSS**	交叉或大加号的替代名称
curses.**ACS_STERLING**	英镑
curses.**ACS_TTEE**	顶部 T 型
curses.**ACS_UARROW**	向上箭头
curses.**ACS_ULCORNER**	左上角
curses.**ACS_URCORNER**	右上角
curses.**ACS_VLINE**	垂线

下面列出了*getmouse()* 所使用的鼠标按键常量:

鼠标按键常量	含意
curses.**BUTTONn_PRESSED**	鼠标按键 *n* 被按下
curses.**BUTTONn_RELEASED**	鼠标按键 *n* 被释放
curses.**BUTTONn_CLICKED**	鼠标按键 *n* 被点击
curses.**BUTTONn_DOUBLE_CLICKED**	鼠标按键 *n* 被双击
curses.**BUTTONn_TRIPLE_CLICKED**	鼠标按键 *n* 被三击
curses.**BUTTON_SHIFT**	当按键状态改变时 Shift 被按下
curses.**BUTTON_CTRL**	当按键状态改变时 Control 被按下
curses.**BUTTON_ALT**	当按键状态改变时 Control 被按下

在 3.10 版本发生变更: 现在 BUTTON5_* 常量如果是由下层 curses 库提供的则会对外公开。

下表列出了预定义的颜色:

常量	颜色
curses.**COLOR_BLACK**	黑色
curses.**COLOR_BLUE**	蓝色
curses.**COLOR_CYAN**	青色（浅绿蓝色）
curses.**COLOR_GREEN**	绿色
curses.**COLOR_MAGENTA**	洋红色（紫红色）
curses.**COLOR_RED**	红色
curses.**COLOR_WHITE**	白色
curses.**COLOR_YELLOW**	黄色

17.6 curses.textpad --- 用于 curses 程序的文本输入控件

curses.textpad 模块提供了一个 *Textbox* 类，该类在 curses 窗口中处理基本的文本编辑，支持一组与 Emacs 类似的键绑定（因此这也适用于 Netscape Navigator, BBedit 6.x, FrameMaker 和许多其他程序）。该模块还提供了一个绘制矩形的函数，适用于容纳文本框或其他目的。

curses.textpad 模块定义了以下函数:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)

绘制一个矩形。第一个参数必须为窗口对象；其余参数均为相对于该窗口的坐标值。第二和第三个参数为要绘制的矩形的左上角的 y 和 x 坐标值；第四和第五个参数为其右下角的 y 和 x 坐标值。将会使用 VT100/IBM PC 形式的字符在可用的终端上（包括 xterm 和大多数其他软件终端模拟器）绘制矩形。在其他情况下则将使用 ASCII 横杠、竖线和加号绘制。

17.6.1 文本框对象

你可以通过如下方式实例化一个 *Textbox*:

class curses.textpad.**Textbox**(*win*)

返回一个文本框控件对象。*win* 参数必须是一个 curses 窗口 对象，文本框将被包含在其中。文本框的编辑光标在初始时位于包含窗口的左上角，坐标值为 `(0, 0)`。实例的 *stripspaces* 旗标初始时为启用。

Textbox 对象具有以下方法:

edit([*validator*])

这是你通常将使用的入口点。它接受编辑按键直到键入了一个终止按键。如果提供了 *validator*，它必须是一个函数。它将在每次按键时被调用并传入相应的按键作作为形参；命令发送将在结果上执行。此方法会以字符串形式返回窗口内容；是否包括窗口中的空白将受到 *stripspaces* 属性的影响。

do_command(*ch*)

处理单个按键命令。以下是支持的特殊按键：

按键	动作
Control-A	转到窗口的左边缘。
Control-B	光标向左，如果可能，包含前一行。
Control-D	删除光标下的字符。
Control-E	前往右边缘（stripspaces 关闭时）或者行尾（stripspaces 启用时）。
Control-F	向右移动光标，适当时换行到下一行。
Control-G	终止，返回窗口内容。
Control-H	向后删除字符。
Control-J	如果窗口是 1 行则终止，否则插入换行符。
Control-K	如果行为空，则删除它，否则清除到行尾。
Control-L	刷新屏幕。
Control-N	光标向下; 向下移动一行。
Control-O	在光标位置插入一个空行。
Control-P	光标向上; 向上移动一行。

如果光标位于无法移动的边缘，则移动操作不执行任何操作。在可能的情况下，支持以下同义词：

常量	按键
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

所有其他按键将被视为插入给定字符并右移的命令（带有自动折行）。

gather()

以字符串形式返回窗口内容；是否包括窗口中的空白将受到*stripspaces*成员的影响。

stripspaces

此属性是控制窗口中空白解读方式的旗标。当启用时，每一行的末尾空白会被忽略；任何将光标定位至末尾空白的光标动作都将改为前往该行末尾，并且在收集窗口内容时将去除末尾空白。

17.7 curses.ascii --- 用于 ASCII 字符的工具

源代码: Lib/curses/ascii.py

---

curses.ascii 模块提供了一些 ASCII 字符的名称常量以及在各种 ASCII 字符类中执行成员检测的函数。所提供的控制字符常量如下：

名称	含意
curses.ascii.**NUL**	
curses.ascii.**SOH**	标题开始，控制台中断

续下页

表 3 – 接上页

名称	含意
curses.ascii.**STX**	文本开始
curses.ascii.**ETX**	文本结束
curses.ascii.**EOT**	传输结束
curses.ascii.**ENQ**	查询，附带*ACK* 流量控制
curses.ascii.**ACK**	确认
curses.ascii.**BEL**	蜂鸣器
curses.ascii.**BS**	退格
curses.ascii.**TAB**	制表符
curses.ascii.**HT**	*TAB* 的别名：”水平制表符”
curses.ascii.**LF**	换行
curses.ascii.**NL**	*LF* 的别名：”新行”
curses.ascii.**VT**	垂直制表符
curses.ascii.**FF**	换页
curses.ascii.**CR**	回车
curses.ascii.**SO**	Shift-out，开始替换字符集
curses.ascii.**SI**	Shift-in，恢复默认字符集
curses.ascii.**DLE**	Data-link escape，数据链接转义
curses.ascii.**DC1**	XON，用于流程控制

续下页

表 3 – 接上页

名称	含意
curses.ascii.**DC2**	Device control 2，块模式流程控制
curses.ascii.**DC3**	XOFF，用于流程控制
curses.ascii.**DC4**	设备控制 4
curses.ascii.**NAK**	否定确认
curses.ascii.**SYN**	同步空闲
curses.ascii.**ETB**	末端传输块
curses.ascii.**CAN**	取消
curses.ascii.**EM**	媒体结束
curses.ascii.**SUB**	替换
curses.ascii.**ESC**	退出
curses.ascii.**FS**	文件分隔符
curses.ascii.**GS**	组分隔符
curses.ascii.**RS**	Record separator，块模式终止符
curses.ascii.**US**	单位分隔符
curses.ascii.**SP**	空格
curses.ascii.**DEL**	删除

请注意其中有许多在现今已经没有实际作用。这些助记符是来源于数字计算机之前的电传打印机规范。

此模块提供了下列函数，对应于标准 C 库中的函数:

curses.ascii.**isalnum**(*c*)

检测 ASCII 字母数字类字符；它等价于 `isalpha(c)` 或 `isdigit(c)`。

curses.ascii.**isalpha**(*c*)

检测 ASCII 字母类字符；它等价于 `isupper(c) or islower(c)`。

curses.ascii.**isascii**(*c*)
检测字符值是否在 7 位 ASCII 集范围内。

curses.ascii.**isblank**(*c*)
检测 ASCII 空白字符；包括空格或水平制表符。

curses.ascii.**iscntrl**(*c*)
检测 ASCII 控制字符（在 0x00 到 0x1f 或 0x7f 范围内）。

curses.ascii.**isdigit**(*c*)
检测 ASCII 十进制数码，即 `'0'` 至 `'9'`。它等价于 `c in string.digits`。

curses.ascii.**isgraph**(*c*)
检测任意 ASCII 可打印字符，不包括空白符。

curses.ascii.**islower**(*c*)
检测 ASCII 小写字母字符。

curses.ascii.**isprint**(*c*)
检测任意 ASCII 可打印字符，包括空白符。

curses.ascii.**ispunct**(*c*)
检测任意 ASCII 可打印字符，不包括空白符或字母数字类字符。

curses.ascii.**isspace**(*c*)
检测 ASCII 空白字符；包括空格，换行，回车，进纸，水平制表和垂直制表。

curses.ascii.**isupper**(*c*)
检测 ASCII 大写字母字符。

curses.ascii.**isxdigit**(*c*)
检测 ASCII 十六进制数码。这等价于 `c in string.hexdigits`。

curses.ascii.**isctrl**(*c*)
检测 ASCII 控制字符（码位值 0 至 31）。

curses.ascii.**ismeta**(*c*)
检测非 ASCII 字符（码位值 0x80 及以上）。

这些函数接受整数或单字符字符串；当参数为字符串时，会先使用内置函数*ord()* 进行转换。

请注意所有这些函数都是检测根据你传入的字符串的字符所生成的码位值；它们实际上完全不会知晓本机的字符编码格式。

以下两个函数接受单字符字符串或整数形式的字节值；它们会返回相同类型的值。

curses.ascii.**ascii**(*c*)
返回对应于 *c* 的下个 7 比特位的 ASCII 值。

curses.ascii.**ctrl**(*c*)
返回对应于给定字符的控制字符（字符比特值会与 0x1f 进行按位与运算）。

curses.ascii.**alt**(*c*)
返回对应于给定 ASCII 字符的 8 比特位字符（字符比特值会与 0x80 进行按位或运算）。

以下函数接受单字符字符串或整数值；它会返回一个字符串。

curses.ascii.**unctrl**(*c*)
返回 ASCII 字符 *c* 的字符串表示形式。如果 *c* 是可打印字符，则字符串为字符本身。如果该字符是控制字符 (0x00--0x1f) 则字符串由一个插入符 (`'^'`) 加相应的大写字母组成。如果该字符是 ASCII 删除符 (0x7f) 则字符串为 `'^?'`。如果该字符设置了元比特位 (0x80)，元比特位会被去除，应用以上规则后将在结果之前添加 `'!'`。

curses.ascii.**controlnames**

一个 33 元素的字符串数据，其中按从 0 (NUL) 到 0x1f (US) 的顺序包含了三十二个 ASCII 控制字符的 ASCII 助记符，另加空格符的助记符 SP。

17.8 curses.panel --- 针对 curses 的面板栈扩展

面板是具有添加深度功能的窗口，因此它们可以从上至下堆叠为栈，只有显示每个窗口的可见部分会显示出来。面板可以在栈中被添加、上移或下移，也可以被移除。

17.8.1 函数

curses.panel 模块定义了以下函数:

curses.panel.**bottom_panel**()

返回面板栈中的底部面板。

curses.panel.**new_panel**(*win*)

返回一个面板对象，将其与给定的窗口 *win* 相关联。请注意你必须显式地保持所返回的面板对象。如果你不这样做，面板对象会被垃圾回收并从面板栈中被移除。

curses.panel.**top_panel**()

返回面板栈中的顶部面板。

curses.panel.**update_panels**()

在面板栈发生改变后更新虚拟屏幕。这不会调用*curses.doupdate()*，因此你不必自己执行此操作。

17.8.2 Panel 对象

Panel 对象，如上面*new_panel()* 所返回的对象，是带有栈顺序的多个窗口。总是会有一个窗口与确定内容的面板相关联，面板方法会负责窗口在面板栈中的深度。

Panel 对象具有以下方法:

Panel.**above**()

返回当前面板之上的面板。

Panel.**below**()

返回当前面板之下的面板。

Panel.**bottom**()

将面板推至栈底部。

Panel.**hidden**()

如果面板被隐藏（不可见）则返回 True，否则返回 False。

Panel.**hide**()

隐藏面板。这不会删除对象，它只是让窗口在屏幕上不可见。

Panel.**move**(*y*, *x*)

将面板移至屏幕坐标 (y, x)。

Panel.**replace**(*win*)

将与面板相关联的窗口改为窗口 *win*。

Panel.**set_userptr**(*obj*)

将面板的用户指向设为 *obj*。这被用来将任意数据与面板相关联，数据可以是任何 Python 对象。

Panel.**show**()

显示面板（面板可能已被隐藏）。

Panel.**top**()

将面板推至栈顶部。

Panel.**userptr**()

返回面板的用户指针。这可以是任何 Python 对象。

Panel.**window**()

返回与面板相关联的窗口对象。

CHAPTER 18

并发执行

本章中描述的模块支持并发执行代码。适当的工具选择取决于要执行的任务（CPU 密集型或 IO 密集型）和偏好的开发风格（事件驱动的协作式多任务或抢占式多任务处理）。这是一个概述：

18.1 `threading` --- 基于线程的并行

源代码: Lib/threading.py

这个模块在低层级的*_thread*模块之上构造了高层级的线程接口。

在 3.7 版本发生变更: 这个模块曾经为可选项，但现在总是可用。

参见

concurrent.futures.ThreadPoolExecutor 提供了一个高层级接口用来向后台线程推送任务而不会阻塞调用方线程的执行，同时仍然能够在需要时获取任务的结果。

queue 提供了一个线程安全的接口用来在运行中的线程之间交换数据。

asyncio 提供了一个替代方式用来实现任务层级的并发而不要求使用多个操作系统线程。

备注

在 Python 2.x 系列中，此模块包含有某些方法和函数 `camelCase` 形式的名称。它们在 Python 3.10 中已弃用，但为了与 Python 2.5 及更旧版本的兼容性而仍受到支持。

在 CPython 中，由于存在*全局解释器锁*，同一时刻只有一个线程可以执行 Python 代码（虽然某些性能导向的库可能会去除此限制）。如果你想让你的应用更好地利用多核心计算机的计算资源，推荐你使用*multiprocessing* 或*concurrent.futures.ProcessPoolExecutor*。但是，如果你想要同时运行多个 I/O 密集型任务，则多线程仍然是一个合适的模型。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

这个模块定义了以下函数：

threading.**active_count**()

返回当前存活的*Thread*对象的数量。返回值与*enumerate()*所返回的列表长度一致。

函数 activeCount 是此函数的已弃用别名。

threading.**current_thread**()

返回当前对应调用者的控制线程的*Thread*对象。如果调用者的控制线程不是利用*threading*创建，会返回一个功能受限的虚拟线程对象。

函数 currentThread 是此函数的已弃用别名。

threading.**excepthook**(*args*, /)

处理由*Thread.run()*引发的未捕获异常。

args 参数具有以下属性:

- *exc_type*: 异常类型
- *exc_value*: 异常值，可以是 None.
- *exc_traceback*: 异常回溯，可以是 None.
- *thread*: 引发异常的线程，可以为 None。

如果 *exc_type* 为*SystemExit*，则异常会被静默地忽略。在其他情况下，异常将被打印到*sys.stderr*。

如果此函数引发了异常，则会调用*sys.excepthook()*来处理它。

*threading.excepthook()*可以被重载以控制由*Thread.run()*引发的未捕获异常的处理方式。

使用定制钩子存放 *exc_value* 可能会创建引用循环。它应当在不再需要异常时被显式地清空以打破引用循环。

如果一个对象正在被销毁，那么使用自定义的钩子储存 *thread* 可能会将其复活。请在自定义钩子生效后避免储存 *thread*，以避免对象的复活。

> **参见**
>
> *sys.excepthook()* 处理未捕获的异常。

Added in version 3.8.

threading.**__excepthook__**

保存*threading.excepthook()*的原始值。它被保存以便在原始值碰巧被已损坏或替代对象所替换的情况下可被恢复。

Added in version 3.10.

threading.**get_ident**()

返回当前线程的“线程标识符”。它是一个非零的整数。它的值没有直接含义，主要是用作 magic cookie，比如作为含有线程相关数据的字典的索引。线程标识符可能会在线程退出，新线程创建时被复用。

Added in version 3.3.

threading.**get_native_id**()

返回内核分配给当前线程的原生集成线程 ID。这是一个非负整数。它的值可被用来在整个系统中唯一地标识这个特定线程（直到线程终结，在那之后该值可能会被 OS 回收再利用）。

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

在 3.13 版本发生变更: 增加了对 GNU/kFreeBSD 的支持。

threading.**enumerate**()

返回当前所有存活的*Thread*对象的列表。该列表包括守护线程以及*current_thread()*创建的空线程。它不包括已终结的和尚未开始的线程。但是，主线程将总是结果的一部分，即使是在已终结的时候。

threading.**main_thread**()

返回主*Thread*对象。一般情况下，主线程是 Python 解释器开始时创建的线程。

Added in version 3.4.

threading.**settrace**(*func*)

为所有*threading*模块开始的线程设置追踪函数。在每个线程的*run()*方法被调用前，*func* 会被传递给*sys.settrace()*。

threading.**settrace_all_threads**(*func*)

为从*threading*模块启动的所有线程和当前正在执行的所有 Python 线程设置追踪函数。

func 将为每个线程传递给*sys.settrace()*，在其*run()*方法被调用之前。

Added in version 3.12.

threading.**gettrace**()

返回由*settrace()*设置的跟踪函数。

Added in version 3.10.

threading.**setprofile**(*func*)

为所有*threading*模块开始的线程设置性能测试函数。在每个线程的*run()*方法被调用前，*func* 会被传递给*sys.setprofile()*。

threading.**setprofile_all_threads**(*func*)

为从*threading*模块启动的所有线程和当前正在执行的所有 Python 线程设置性能分析函数。

func 将为每个线程传递给*sys.setprofile()*，在其*run()*方法被调用之前。

Added in version 3.12.

threading.**getprofile**()

返回由*setprofile()*设置的性能分析函数。

Added in version 3.10.

threading.**stack_size**([*size*])

返回创建线程时使用的堆栈大小。可选参数 *size* 指定之后新建的线程的堆栈大小，而且一定要是 0（根据平台或者默认配置）或者最小是 32,768(32KiB) 的一个正整数。如果 *size* 没有指定，默认是 0。如果不支持改变线程堆栈大小，会抛出*RuntimeError*错误。如果指定的堆栈大小不合法，会抛出*ValueError*错误并且不会修改堆栈大小。32KiB 是当前最小的能保证解释器有足够堆栈空间的堆栈大小。需要注意的是部分平台对于堆栈大小会有特定的限制，例如要求大于 32KiB 的堆栈大小或者需要根据系统内存页面的整数倍进行分配 - 应当查阅平台文档有关详细信息（4KiB 页面比较普遍，在没有更具体信息的情况下，建议的方法是使用 4096 的倍数作为堆栈大小）。

Availability: Windows, pthreads.

带有 POSIX 线程支持的 Unix 平台。

这个模块同时定义了以下常量：

threading.**TIMEOUT_MAX**

阻塞函数（*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()*, ...）中形参 *timeout* 允许的最大值。传入超过这个值的 timeout 会抛出*OverflowError*异常。

Added in version 3.2.

这个模块定义了许多类，详见以下部分。

该模块的设计基于 Java 的线程模型。但是，在 Java 里面，锁和条件变量是每个对象的基础特性，而在 Python 里面，这些被独立成了单独的对象。Python 的*Thread*类只是 Java 的 Thread 类的一个子集；目前

还没有优先级，没有线程组，线程还不能被销毁、停止、暂停、恢复或中断。Java 的 Thread 类的静态方法在实现时会映射为模块级函数。

下述方法的执行都是原子性的。

18.1.1 线程本地数据

线程本地数据是特定线程的数据。管理线程本地数据，只需要创建一个*local*（或者一个子类型）的实例并在实例中储存属性：

```
mydata = threading.local()
mydata.x = 1
```

在不同的线程中，实例的值会不同。

class threading.**local**

一个代表线程本地数据的类。

更多相关细节和大量示例，请参阅 _threading_local 模块的文档字符串: Lib/_threading_local.py。

18.1.2 线程对象

Thread 类代表一个在独立控制线程中运行的活动。指定活动有两种方式：向构造器传递一个可调用对象，或在子类中重载*run()* 方法。其他方法不应在子类中重载（除了构造器）。换句话说，只能重载这个类的 __init__() 和*run()* 方法。

当线程对象一旦被创建，其活动必须通过调用线程的*start()* 方法开始。这会在独立的控制线程中唤起*run()* 方法。

一旦线程活动开始，该线程会被认为是'存活的'。当它的*run()* 方法终结了（不管是正常的还是抛出未被处理的异常），就不是'存活的'。*is_alive()* 方法用于检查线程是否存活。

其他线程可以调用一个线程的*join()* 方法。这会阻塞调用该方法的线程，直到被调用*join()* 方法的线程终结。

线程有名字。名字可以传递给构造函数，也可以通过*name* 属性读取或者修改。

如果*run()* 方法引发了异常，则会调用*threading.excepthook()* 来处理它。在默认情况下，*threading.excepthook()* 会静默地忽略*SystemExit*。

一个线程可以被标记成一个“守护线程”。这个标识的意义是，当剩下的线程都是守护线程时，整个 Python 程序将会退出。初始值继承于创建线程。这个标识可以通过*daemon* 特征属性或者 *daemon* 构造器参数来设置。

备注

守护线程在程序关闭时会突然关闭。他们的资源（例如已经打开的文档，数据库事务等等）可能没有被正确释放。如果你想你的线程正常停止，设置他们成为非守护模式并且使用合适的信号机制，例如：*Event*。

有个”主线程”对象；这对应 Python 程序里面初始的控制线程。它不是一个守护线程。

创建“虚拟线程对象”是有可能的。它们是与“外部线程”相对应的线程对象，是在 threading 模块之外启动的控制线程，例如直接来自 C 代码。虚拟线程对象的功能是受限的；它们总是会被视为处于激活和守护状态，且无法被合并。它们绝不会被删除，因为检测外部线程的终结是不可能做到的。

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*)

应当始终使用关键字参数调用此构造函数。参数如下：

group 应为 None；保留给将来实现 ThreadGroup 类的扩展使用。

target 是用于*run()* 方法调用的可调用对象。默认是 None，表示不需要调用任何方法。

name 是线程名称。在默认情况下，会以”Thread-*N*” 的形式构造唯一名称，其中 *N* 为一个较小的十进制数值，或是”Thread-*N* (target)” 的形式，其中”target” 为 `target.__name__`，如果指定了 *target* 参数的话。

args 是用于唤起目标函数的参数列表或元组。默认为 `()`。

kwargs 是用于调用目标函数的关键字参数字典。默认是 `{}`。

如果不是 `None`，*daemon* 参数将显式地设置该线程是否为守护模式。如果是 `None` (默认值)，线程将继承当前线程的守护模式属性。

如果子类型重载了构造函数，它一定要确保在做任何事前，先唤起基类构造器 (`Thread.__init__()`)。

在 3.3 版本发生变更: 增加了 *daemon* 形参。

在 3.10 版本发生变更: 使用 *target* 名称，如果 *name* 参数被省略的话。

start()

开始线程活动。

它在一个线程里最多只能被调用一次。它安排对象的*run()* 方法在一个独立的控制线程中被调用。

如果同一个线程对象中调用这个方法的次数大于一次，会抛出*RuntimeError* 。

run()

代表线程活动的方法。

你可以在子类型里重载这个方法。标准的*run()* 方法会对作为 *target* 参数传递给该对象构造器的可调用对象（如果存在）被唤起，并附带从 *args* 和 *kwargs* 参数分别获取的位置和关键字参数。

使用列表或元组作为传给*Thread* 的 *args* 参数可以达成同样的效果。

示例:

```
>>> from threading import Thread
>>> t = Thread(target=print, args=[1])
>>> t.run()
1
>>> t = Thread(target=print, args=(1,))
>>> t.run()
1
```

join(*timeout=None*)

等待，直到线程终结。这会阻塞调用这个方法的线程，直到被调用*join()* 的线程终结 -- 不管是正常终结还是抛出未处理异常 -- 或者直到发生超时，超时选项是可选的。

当 *timeout* 参数存在而且不是 `None` 时，它应该是一个用于指定操作超时的以秒为单位的浮点数（或者分数）。因为*join()* 总是返回 `None` ，所以你一定要在*join()* 后调用*is_alive()* 才能判断是否发生超时 -- 如果线程仍然存活，则*join()* 超时。

当 *timeout* 参数不存在或者是 `None` ，这个操作会阻塞直到线程终结。

一个线程可以被合并多次。

如果尝试加入当前线程会导致死锁，*join()* 会引起*RuntimeError* 异常。如果尝试*join()* 一个尚未开始的线程，也会抛出相同的异常。

name

只用于识别的字符串。它没有语义。多个线程可以赋予相同的名称。初始名称由构造函数设置。

getName()

setName()

已被弃用的*name* 的取值/设值 API；请改为直接以特征属性方式使用它。

自 3.10 版本弃用.

ident

这个线程的’线程标识符’，如果线程尚未开始则为 None 。这是个非零整数。参见*get_ident()* 函数。当一个线程退出而另外一个线程被创建，线程标识符会被复用。即使线程退出后，仍可得到标识符。

native_id

此线程的线程 ID (TID)，由 OS (内核) 分配。这是一个非负整数，或者如果线程还未启动则为 None。请参阅*get_native_id()* 函数。这个值可被用来在全系统范围内唯一地标识这个特定线程 (直到线程终结，在那之后该值可能会被 OS 回收再利用)。

> **备注**
>
> 类似于进程 ID，线程 ID 的有效期（全系统范围内保证唯一）将从线程被创建开始直到线程被终结。

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD.

Added in version 3.8.

is_alive()

返回线程是否存活。

当*run()* 方法刚开始直到*run()* 方法刚结束，这个方法返回 True 。模块函数*enumerate()* 返回包含所有存活线程的列表。

daemon

一个布尔值，表示这个线程是否是一个守护线程（True）或不是（False）。这个值必须在调用*start()* 之前设置，否则会引发*RuntimeError* 。它的初始值继承自创建线程；主线程不是一个守护线程，因此所有在主线程中创建的线程默认为*daemon* = False。

当没有存活的非守护线程时，整个 Python 程序才会退出。

isDaemon()

setDaemon()

已被弃用的*daemon* 的取值/设值 API；请改为直接以特征属性方式使用它。

自 3.10 版本弃用.

18.1.3 锁对象

原始锁是一个在锁定时不属于特定线程的同步基元组件。在 Python 中，它是能用的最低级的同步基元组件，由*_thread* 扩展模块直接实现。

原始锁处于” 锁定” 或者” 非锁定” 两种状态之一。它被创建时为非锁定状态。它有两个基本方法，*acquire()* 和*release()* 。当状态为非锁定时，*acquire()* 将状态改为锁定并立即返回。当状态是锁定时，*acquire()* 将阻塞至其他线程调用*release()* 将其改为非锁定状态，然后*acquire()* 调用重置其为锁定状态并返回。*release()* 只在锁定状态下调用；它将状态改为非锁定并立即返回。如果尝试释放一个非锁定的锁，则会引发*RuntimeError* 异常。

锁同样支持*上下文管理协议*。

当多个线程在*acquire()* 等待状态转变为未锁定被阻塞，然后*release()* 重置状态为未锁定时，只有一个线程能继续执行；至于哪个等待线程继续执行没有定义，并且会根据实现而不同。

所有方法的执行都是原子性的。

class threading.**Lock**

实现原始锁对象的类。一旦一个线程获得一个锁，会阻塞随后尝试获得锁的线程，直到它被释放；任何线程都可以释放它。

在 3.13 版本发生变更: 现在 Lock 是一个类。在更早的 Python 版本中，Lock 是一个返回下层私有锁类型的实例的工厂函数。

acquire(*blocking=True*, *timeout=-1*)

可以阻塞或非阻塞地获得锁。

当调用时参数 *blocking* 设置为 True（缺省值），阻塞直到锁被释放，然后将锁锁定并返回 True。

在参数 *blocking* 被设置为 False 的情况下调用，将不会发生阻塞。如果调用时 *blocking* 设为 True 会阻塞，并立即返回 False；否则，将锁锁定并返回 True。

当参数 *timeout* 使用设置为正值的浮点数调用时，最多阻塞 *timeout* 指定的秒数，在此期间锁不能被获取。设置 *timeout* 参数为 -1 specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is False。

如果成功获得锁，则返回 True，否则返回 False (例如发生 超时的时候)。

在 3.2 版本发生变更: 新的 *timeout* 形参。

在 3.2 版本发生变更: 现在如果底层线程实现支持，则可以通过 POSIX 上的信号中断锁的获取。

release()

释放一个锁。这个方法可以在任何线程中调用，不单指获得锁的线程。

当锁被锁定，将它重置为未锁定，并返回。如果其他线程正在等待这个锁解锁而被阻塞，只允许其中一个允许。

当在未锁定的锁上唤起时，会引发*RuntimeError*。

没有返回值。

locked()

当锁被获取时，返回 True。

18.1.4 递归锁对象

重入锁是一个可以被同一个线程多次获取的同步基元组件。在内部，它在基元锁的锁定/非锁定状态上附加了" 所属线程" 和" 递归等级" 的概念。在锁定状态下，某些线程拥有锁；在非锁定状态下，没有线程拥有它。

线程调用锁的*acquire()* 方法来锁定它，并调用*release()* 方法来解锁。

> **备注**
>
> 重入型锁支持上下文管理协议，因此推荐使用 with 而不是手动调用*acquire()* 和*release()* 来针对一个代码块处理锁的获取和释放。

RLock 的*acquire()*/*release()* 调用对可以嵌套，这不同于 Lock 的*acquire()*/*release()*。只有最终的*release()* (最外面一对的*release()*) 会将锁重置为已解锁状态并允许在*acquire()* 中被阻塞的其他线程继续执行。

acquire()/*release()* 必须成对使用：每个 acquire 必须在获取锁的线程中有对应的 release。如果锁调用 release 的次数未能与 acquire 的次数一致则会导致死锁。

class threading.**RLock**

此类实现了重入锁对象。重入锁必须由获取它的线程释放。一旦线程获得了重入锁，同一个线程再次获取它将不阻塞；线程必须在每次获取它时释放一次。

需要注意的是 RLock 其实是一个工厂函数，返回平台支持的具体递归锁类中最有效的版本的实例。

acquire(*blocking=True*, *timeout=-1*)

可以阻塞或非阻塞地获得锁。

> **参见**
>
> **将 *RLock* 用作上下文管理器**
> 在大多数场合下相比手动的 `acquire()` 和*`release()`* 调用更为推荐。

当被唤起时将 *blocking* 参数设为 `True` (默认值):

- 如无任何线程持有锁，则获取锁并立即返回。
- 如有其他线程持有锁，则阻塞执行直至能够获取锁，或直至 *timeout*，如果将其设为一个正浮点数值的话。
- 如同一线程持有锁，则再次获取该锁，并立即返回。这是*`Lock`* 和 `RLock` 之间的区别；*`Lock`* 将以与之前相同的方式处理此情况，即阻塞执行直至能够获取锁。

当被唤起时将 *blocking* 参数设为 `False`:

- 如无任何线程持有锁，则获取锁并立即返回。
- 如有其他线程持有锁，则立即返回。
- 如同一线程持有锁，则再次获取该锁并立即返回。

在所有情况下，如果线程能够获取锁，则返回 `True`。如果线程不能获取锁（即未阻塞执行或达到超时限制）则返回 `False`。

如果被多次调用，则未能调用相同次数的*`release()`* 可能导致死锁。请考虑将 `RLock` 用作上下文管理器而不是直接调用 acquire/release。

在 3.2 版本发生变更: 新的 *timeout* 形参。

release()

释放锁，自减递归等级。如果减到零，则将锁重置为非锁定状态 (不被任何线程拥有)，并且，如果其他线程正被阻塞着等待锁被解锁，则仅允许其中一个线程继续。如果自减后，递归等级仍然不是零，则锁保持锁定，仍由调用线程拥有。

只有在调用方线程持有锁时才能调用此方法。如果在未获取锁的情况下调用此方法则会引发*`RuntimeError`*。

没有返回值。

18.1.5 条件对象

条件变量总是与某种类型的锁对象相关联，锁对象可以通过传入获得，或者在缺省的情况下自动创建。当多个条件变量需要共享同一个锁时，传入一个锁很有用。锁是条件对象的一部分，你不必单独地跟踪它。

条件变量遵循*上下文管理协议*：使用 `with` 语句会在它包围的代码块内获取关联的锁。*`acquire()`* 和*`release()`* 方法也能调用关联锁的相关方法。

其它方法必须在持有关联的锁的情况下调用。*`wait()`* 方法释放锁，然后阻塞直到其它线程调用*`notify()`* 方法或*`notify_all()`* 方法唤醒它。一旦被唤醒，*`wait()`* 方法重新获取锁并返回。它也可以指定超时时间。

The *`notify()`* method wakes up one of the threads waiting for the condition variable, if any are waiting. The *`notify_all()`* method wakes up all threads waiting for the condition variable.

注意：*`notify()`* 方法和*`notify_all()`* 方法并不会释放锁，这意味着被唤醒的线程不会立即从它们的*`wait()`* 方法调用中返回，而是会在调用了*`notify()`* 方法或*`notify_all()`* 方法的线程最终放弃了锁的所有权后返回。

使用条件变量的典型编程风格是将锁用于同步某些共享状态的权限，那些对状态的某些特定改变感兴趣的线程，它们重复调用*wait()* 方法，直到看到所期望的改变发生；而对于修改状态的线程，它们将当前状态改变为可能是等待者所期待的新状态后，调用*notify()* 方法或者*notify_all()* 方法。例如，下面的代码是一个通用的无限缓冲区容量的生产者-消费者情形：

```
# 消费一个条目
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# 生产一个条目
with cv:
    make_an_item_available()
    cv.notify()
```

使用 `while` 循环检查所要求的条件成立与否是有必要的，因为*wait()* 方法可能要经过不确定长度的时间后才会返回，而此时导致*notify()* 方法调用的那个条件可能已经不再成立。这是多线程编程所固有的问题。*wait_for()* 方法可自动化条件检查，并简化超时计算。

```
# 消费一个条目
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

选择*notify()* 还是*notify_all()* ，取决于一次状态改变是只能被一个还是能被多个等待线程所用。例如在一个典型的生产者-消费者情形中，添加一个项目到缓冲区只需唤醒一个消费者线程。

class threading.**Condition**(*lock=None*)

实现条件变量对象的类。一个条件变量对象允许一个或多个线程在被其它线程所通知之前进行等待。

如果给出了非 `None` 的 *lock* 参数，则它必须为*Lock* 或者*RLock* 对象，并且它将被用作底层锁。否则，将会创建新的*RLock* 对象，并将其用作底层锁。

在 3.3 版本发生变更: 从工厂函数变为类。

acquire(**args*)

请求底层锁。此方法调用底层锁的相应方法，返回值是底层锁相应方法的返回值。

release()

释放底层锁。此方法调用底层锁的相应方法。没有返回值。

wait(*timeout=None*)

等待直到被通知或发生超时。如果线程在调用此方法时没有获得锁，将会引发*RuntimeError* 异常。

这个方法释放底层锁，然后阻塞，直到在另外一个线程中调用同一个条件变量的*notify()* 或*notify_all()* 唤醒它，或者直到可选的超时发生。一旦被唤醒或者超时，它重新获得锁并返回。

当提供了 *timeout* 参数且不是 `None` 时，它应该是一个浮点数，代表操作的超时时间，以秒为单位（可以为小数）。

当底层锁是个*RLock* ，不会使用它的*release()* 方法释放锁，因为当它被递归多次获取时，实际上可能无法解锁。相反，使用了*RLock* 类的内部接口，即使多次递归获取它也能解锁它。然后，在重新获取锁时，使用另一个内部接口来恢复递归级别。

返回 `True` ，除非提供的 *timeout* 过期，这种情况下返回 `False`。

在 3.2 版本发生变更: 在此之前，方法总是返回 `None`。

wait_for(*predicate*, *timeout=None*)

等待，直到条件计算为真。*predicate* 应该是一个可调用对象而且它的返回值可被解释为一个布尔值。可以提供 *timeout* 参数给出最大等待时间。

这个实用方法会重复地调用*wait()* 直到满足判断式或者发生超时。返回值是判断式最后一个返回值，而且如果方法发生超时会返回 False 。

忽略超时功能，调用此方法大致相当于编写:

```
while not predicate():
    cv.wait()
```

因此，规则同样适用于*wait()* ：锁必须在被调用时保持获取，并在返回时重新获取。随着锁定执行判断式。

Added in version 3.2.

notify(*n=1*)

默认唤醒一个等待这个条件的线程。如果调用线程在没有获得锁的情况下调用这个方法，会引发*RuntimeError* 异常。

这个方法唤醒最多 *n* 个正在等待这个条件变量的线程；如果没有线程在等待，这是一个空操作。

当前实现中，如果至少有 *n* 个线程正在等待，准确唤醒 *n* 个线程。但是依赖这个行为并不安全。未来，优化的实现有时会唤醒超过 *n* 个线程。

注意：被唤醒的线程并没有真正恢复到它调用的*wait()* ，直到它可以重新获得锁。因为*notify()* 不释放锁，其调用者才应该这样做。

notify_all()

唤醒所有正在等待这个条件的线程。这个方法行为与*notify()* 相似，但并不只唤醒单一线程，而是唤醒所有等待线程。如果调用线程在调用这个方法时没有获得锁，会引发*RuntimeError* 异常。

notifyAll 方法是此方法的已弃用别名。

18.1.6 信号量对象

这是计算机科学史上最古老的同步原语之一，早期的荷兰科学家 Edsger W. Dijkstra 发明了它。(他使用名称 P() 和 V() 而不是*acquire()* 和*release()*)。

一个信号量管理一个内部计数器，该计数器因*acquire()* 方法的调用而递减，因*release()* 方法的调用而递增。计数器的值永远不会小于零；当*acquire()* 方法发现计数器为零时，将会阻塞，直到其它线程调用*release()* 方法。

信号量对象也支持上下文管理协议 。

class threading.**Semaphore**(*value=1*)

该类实现信号量对象。信号量对象管理一个原子性的计数器，代表*release()* 方法的调用次数减去*acquire()* 的调用次数再加上一个初始值。如果需要，*acquire()* 方法将会阻塞直到可以返回而不会使得计数器变成负数。在没有显式给出 *value* 的值时，默认为 1。

可选参数 *value* 赋予内部计数器初始值，默认值为 1 。如果 *value* 被赋予小于 0 的值，将会引发*ValueError* 异常。

在 3.3 版本发生变更: 从工厂函数变为类。

acquire(*blocking=True*, *timeout=None*)

获取一个信号量。

在不带参数的情况下调用时:

- 如果在进入时内部计数器的值大于零，则将其减一并立即返回 True。
- 如果在进入时内部计数器的值为零，则将会阻塞直到被对*release()* 的调用唤醒。一旦被唤醒（并且计数器的值大于 0），则将计数器减 1 并返回 True。每次对*release()* 的调用将只唤醒一个线程。线程被唤醒的次序是不可确定的。

当 *blocking* 设置为 `False` 时调用，不会阻塞。如果没有参数的调用会阻塞，立即返回 `False`；否则，做与无参数调用相同的事情时返回 `True`。

当被唤起时如果 *timeout* 不为 `None`，则它将阻塞最多 *timeout* 秒。请求在此时段时未能成功完成获取则将返回 `False`。在其他情况下返回 `True`。

在 3.2 版本发生变更: 新的 *timeout* 形参。

release(*n=1*)

释放一个信号量，将内部计数器的值增加 *n*。当进入时值为零且有其他线程正在等待它再次变为大于零时，则唤醒那 *n* 个线程。

在 3.9 版本发生变更: 增加了 *n* 形参以一次性释放多个等待线程。

class threading.**BoundedSemaphore**(*value=1*)

该类实现有界信号量。有界信号量通过检查以确保它当前的值不会超过初始值。如果超过了初始值，将会引发 *ValueError* 异常。在大多情况下，信号量用于保护数量有限的资源。如果信号量被释放的次数过多，则表明出现了错误。没有指定时，*value* 的值默认为 1。

在 3.3 版本发生变更: 从工厂函数变为类。

Semaphore 例子

信号量通常用于保护数量有限的资源，例如数据库服务器。在资源数量固定的任何情况下，都应该使用有界信号量。在生成任何工作线程前，应该在主线程中初始化信号量。

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

工作线程生成后，当需要连接服务器时，这些线程将调用信号量的 acquire 和 release 方法：

```
with pool_sema:
    conn = connectdb()
    try:
        # ... 使用连接 ...
    finally:
        conn.close()
```

使用有界信号量能减少这种编程错误：信号量的释放次数多于其请求次数。

18.1.7 事件对象

这是线程之间通信的最简单机制之一：一个线程发出事件信号，而其他线程等待该信号。

一个事件对象管理一个内部标识，调用 *set()* 方法可将其设置为 true ，调用 *clear()* 方法可将其设置为 false ，调用 *wait()* 方法将进入阻塞直到标识为 true 。

class threading.**Event**

实现事件对象的类。事件对象管理一个内部标识，调用 *set()* 方法可将其设置为 true。调用 *clear()* 方法可将其设置为 false 。调用 *wait()* 方法将进入阻塞直到标识为 true。这个标识初始时为 false 。

在 3.3 版本发生变更: 从工厂函数变为类。

is_set()

当且仅当内部标识为 true 时返回 `True` 。

`isSet` 方法是此方法的已弃用别名。

set()

将内部标识设置为 true 。所有正在等待这个事件的线程将被唤醒。当标识为 true 时，调用 *wait()* 方法的线程不会被被阻塞。

clear()

将内部标识设置为 false 。之后调用*wait()* 方法的线程将会被阻塞，直到调用*set()* 方法将内部标识再次设置为 true 。

wait(*timeout=None*)

只要内部旗标为假值且未超出所给出的 timeout 值就保持阻塞。返回值表示阻塞方法返回的原因；如果返回是因为内部旗标被设为真值则为 `True`，或者如果给出了 timeout 值而内部旗标在给定的等待时间内没有变成真值则为 `False`。

当提供了 timeout 参数且不为 `None` 时，它应当为一个指定操作的超时限制秒数的浮点值，也可以为分数。

在 3.1 版本发生变更: 在此之前，方法总是返回 `None`。

18.1.8 定时器对象

此类表示一个操作应该在等待一定的时间之后运行 --- 相当于一个定时器。*Timer* 类是*Thread* 类的子类，因此可以像一个自定义线程一样工作。

与线程一样，定时器也是通过调用其*Timer.start* 方法来启动的。定时器可以通过调用*cancel()* 方法来停止（在其动作开始之前）。定时器在执行其行动之前要等待的时间间隔可能与用户指定的时间间隔不完全相同。

例如:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # 30 秒之后，将打印 "hello, world"
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)

创建一个定时器，在经过 *interval* 秒的间隔事件后，将会用参数 *args* 和关键字参数 *kwargs* 调用 *function*。如果 *args* 为 `None` （默认值），则会使用一个空列表。如果 *kwargs* 为 `None` （默认值），则会使用一个空字典。

在 3.3 版本发生变更: 从工厂函数变为类。

cancel()

停止定时器并取消执行计时器将要执行的操作。仅当计时器仍处于等待状态时有效。

18.1.9 栅栏对象

Added in version 3.2.

栅栏类提供一个简单的同步原语，用于应对固定数量的线程需要彼此相互等待的情况。线程调用*wait()* 方法后将阻塞，直到所有线程都调用了*wait()* 方法。此时所有线程将被同时释放。

栅栏对象可以被多次使用，但进程的数量不能改变。

这是一个使用简便的方法实现客户端进程与服务端进程同步的例子:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
```

(续下页)

(接上页)

```
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

创建一个需要 *parties* 个线程的栅栏对象。如果提供了可调用的 *action* 参数，它会在所有线程被释放时在其中一个线程中自动调用。*timeout* 是默认的超时时间，如果没有在*wait()* 方法中指定超时时间的话。

wait(*timeout=None*)

冲出栅栏。当栅栏中所有线程都已经调用了这个函数，它们将同时被释放。如果提供了 *timeout* 参数，这里的 *timeout* 参数优先于创建栅栏对象时提供的 *timeout* 参数。

函数返回值是一个整数，取值范围在 0 到 *parties* -- 1，在每个线程中的返回值不相同。可用于从所有线程中选择唯一的一个线程执行一些特别的工作。例如：

```
i = barrier.wait()
if i == 0:
    # 只有一个线程需要打印此文本
    print("passed the barrier")
```

如果创建栅栏对象时在构造函数中提供了 *action* 参数，它将在其中一个线程释放前被调用。如果此调用引发了异常，栅栏对象将进入损坏态。

如果发生了超时，栅栏对象将进入破损态。

如果栅栏对象进入破损态，或重置栅栏时仍有线程等待释放，将会引发*BrokenBarrierError* 异常。

reset()

重置栅栏为默认的初始态。如果栅栏中仍有线程等待释放，这些线程将会收到*BrokenBarrierError* 异常。

请注意使用此函数时，如果存在状态未知的其他线程，则可能需要执行外部同步。如果栅栏已损坏则最好将其废弃并新建一个。

abort()

使栅栏处于损坏状态。这将导致任何现有和未来对*wait()* 的调用失败并引发*BrokenBarrierError*。例如可以在需要中止某个线程时使用此方法，以避免应用程序的死锁。

更好的方式是：创建栅栏时提供一个合理的超时时间，来自动避免某个线程出错。

parties

冲出栅栏所需要的线程数量。

n_waiting

当前时刻正在栅栏中阻塞的线程数量。

broken

一个布尔值，值为 `True` 表明栅栏为破损态。

exception threading.**BrokenBarrierError**

异常类，是*RuntimeError* 异常的子类，在*Barrier* 对象重置时仍有线程阻塞时和对象进入破损态时被引发。

18.1.10 在 `with` 语句中使用锁、条件和信号量

本模块提供的所有具有 `acquire` 和 `release` 方法的对象都可用作 `with` 语句的上下文管理器。进入语句块时将调用 `acquire` 方法，退出语句块时将调用 `release` 方法。因此，下面的代码段：

```
with some_lock:
    # 执行某种操作...
```

相当于:

```
some_lock.acquire()
try:
    # 执行某种操作...
finally:
    some_lock.release()
```

现在*Lock* 、 *RLock* 、 *Condition* 、 *Semaphore* 和*BoundedSemaphore* 对象可以用作 `with` 语句的上下文管理器。

18.2 `multiprocessing` --- 基于进程的并行

源代码 Lib/multiprocessing/

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或*WebAssembly 平台* 上不受支持。

18.2.1 概述

multiprocessing 是一个支持使用与*threading* 模块类似的 API 来产生进程的包。*multiprocessing* 包同时提供了本地和远程并发操作，通过使用子进程而非线程有效地绕过了*全局解释器锁*。因此，*multiprocessing* 模块允许程序员充分利用给定机器上的多个处理器。它在 POSIX 和 Windows 上均可运行。

multiprocessing 模块还引入了在*threading* 模块中没有的 API。一个主要的例子就是*Pool* 对象，它提供了一种快捷的方法，赋予函数并行化处理一系列输入值的能力，可以将输入数据分配给不同进程处理（数据并行）。下面的例子演示了在模块中定义此类函数的常见做法，以便子进程可以成功导入该模块。这个数据并行的基本例子使用了*Pool* ，

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

将在标准输出中打印

```
[1, 4, 9]
```

参见

concurrent.futures.ProcessPoolExecutor 提供了一个更高层级的接口用来将任务推送到后台进程而不会阻塞调用方进程的执行。与直接使用*Pool* 接口相比，*concurrent.futures* API 能更好地允许将工作单元发往无需等待结果的下层进程池。

Process 类

在*multiprocessing* 中，通过创建一个*Process* 对象然后调用它的 start() 方法来生成进程。*Process* 和*threading.Thread* API 相同。一个简单的多进程程序示例是:

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

要显示所涉及的各个进程 ID，这是一个扩展示例:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

关于为什么 if __name__ == '__main__' 部分是必需的解释，请参见编程指导。

上下文和启动方法

根据不同的平台，*multiprocessing* 支持三种启动进程的方法。这些 *启动方法*有

spawn
: 父进程会启动一个新的 Python 解释器进程。子进程将只继承那些运行进程对象的*run()* 方法所必须的资源。特别地，来自父进程的非必需文件描述符和句柄将不会被继承。使用此方法启动进程相比使用 *fork* 或 *forkserver* 要慢上许多。

  在 POSIX 和 Windows 平台上可用。默认在 Windows 和 macOS 上。

fork
: 父进程使用*os.fork()* 来产生 Python 解释器分叉。子进程在开始时实际上与父进程相同。父进程的所有资源都由子进程继承。请注意，安全分叉多线程进程是棘手的。

  在 POSIX 系统上可用。目前在除 macOS 之外的 POSIX 上为默认值。

  > **备注**
  >
  > 在 Python 3.14 上默认的启动方法将不再为 *fork*。需要 *fork* 的代码应当显式地通过*get_context()* 或*set_start_method()* 来指定。

  在 3.12 版本发生变更: 如果 Python 能够检测到你的进程有多个线程，那么在该启动方法内部调用*os.fork()* 函数将引发*DeprecationWarning*。请使用其他启动方法。进一步的解释参见*os.fork()* 文档。

forkserver

当程序启动并选择 *forkserver* 启动方法时，将产生一个服务器进程。从那时起，每当需要一个新进程时，父进程就会连接到该服务器并请求它分叉一个新进程。分叉服务器进程是单线程的，除非因系统库或预加载导入的附带影响改变了这一点，因此使用`os.fork()` 通常是安全的。没有不必要的资源被继承。

在支持通过 Unix 管道传递文件描述符的 POSIX 平台上可用，例如 Linux。

在 3.4 版本发生变更: 在所有 POSIX 平台上添加了 *spawn*，并为某些 POSIX 平台添加了 *forkserver*。在 Windows 上子进程将不再继承父进程的所有可继承句柄。

在 3.8 版本发生变更: 在 macOS 上，现在 *spawn* 启动方法是默认值。由于 *fork* 启动方法可能因 macOS 系统库也许会启动线程导致子进程崩溃因而应当被视为是不安全的。参见 bpo-33725。

在 POSIX 上使用 *spawn* 或 *forkserver* 启动方法将同时启动一个 资源追踪器进程，负责追踪当前程序的进程产生的已取消连接的命名系统资源（如命名信号量或`SharedMemory` 对象）。当所有进程退出后资源追踪器会负责取消链接任何仍然被追踪的对象。在通常情况下应该是没有此种对象的，但是如果一个子进程是被某个信号杀掉的则可能存在一些“泄露”的资源。（泄露的信号量或共享的内存段不会被自动取消链接直到下一次重启。对于这两种对象来说会有问题因为系统只允许有限数量的命名信号量，而共享的内存段在主内存中占用一些空间。）

要选择一个启动方法，你应该在主模块的 `if __name__ == '__main__'` 子句中调用`set_start_method()` 。例如：

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

在程序中`set_start_method()` 不应该被多次调用。

或者，你可以使用`get_context()` 来获取上下文对象。上下文对象与 multiprocessing 模块具有相同的 API，并允许在同一程序中使用多种启动方法。：

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

请注意，对象在不同上下文创建的进程间可能并不兼容。特别是，使用 *fork* 上下文创建的锁不能传递给使用 *spawn* 或 *forkserver* 启动方法启动的进程。

想要使用特定启动方法的库应该使用`get_context()` 以避免干扰库用户的选择。

警告

`'spawn'` 和 `'forkserver'` 启动方法在 POSIX 系统上通常不能与“已冻结”可执行程序一同使用

（例如由 **PyInstaller** 和 **cx_Freeze** 等软件包产生的二进制文件）。如果代码没有使用线程则可以使用 `'fork'` 启动方法。

在进程之间交换对象

multiprocessing 支持进程之间的两种通信通道：

队列

Queue 类是一个近似*queue.Queue* 的克隆。例如：

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # 打印 "[42, None, 'hello']"
    p.join()
```

队列是线程和进程安全的。任何放入*multiprocessing* 队列的对象都将被序列化。

管道

Pipe() 函数返回一个由管道连接的连接对象，默认情况下是双工（双向）。例如：

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # 打印 "[42, None, 'hello']"
    p.join()
```

返回的两个连接对象*Pipe()* 表示管道的两端。每个连接对象都有 `send()` 和 `recv()` 方法（相互之间的）。请注意，如果两个进程（或线程）同时尝试读取或写入管道的 同一端，则管道中的数据可能会损坏。当然，在不同进程中同时使用管道的不同端的情况下不存在损坏的风险。

`send()` 方法将序列化对象而 `recv()` 将重新创建对象。

进程间同步

multiprocessing 包含来自*threading* 的所有同步原语的等价物。例如，可以使用锁来确保一次只有一个进程打印到标准输出：

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()
```

（续下页）

(接上页)

```
if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

不使用锁的情况下，来自于多进程的输出很容易产生混淆。

进程间共享状态

如上所述，在进行并发编程时，通常最好尽量避免使用共享状态。使用多个进程时尤其如此。

但是，如果你真的需要使用一些共享数据，那么*multiprocessing* 提供了两种方法。

共享内存

可以使用*Value* 或*Array* 将数据存储在共享内存映射中。例如，以下代码:

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

将打印

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

创建 num 和 arr 时使用的 'd' 和 'i' 参数是*array* 模块使用的类型的 typecode： 'd' 表示双精度浮点数，'i' 表示有符号整数。这些共享对象将是进程和线程安全的。

为了更灵活地使用共享内存，可以使用*multiprocessing.sharedctypes* 模块，该模块支持创建从共享内存分配的任意 ctypes 对象。

服务进程

由*Manager()* 返回的管理器对象控制一个服务进程，该进程保存 Python 对象并允许其他进程使用代理操作它们。

Manager() 返回的管理器支持类型：*list* 、*dict* 、*Namespace* 、*Lock* 、*RLock* 、*Semaphore* 、*BoundedSemaphore* 、*Condition* 、*Event* 、*Barrier* 、*Queue* 、*Value* 和*Array* 。例如

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()
```

(续下页)

(接上页)

```
if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

将打印

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

使用服务进程的管理器比使用共享内存对象更灵活，因为它们可以支持任意对象类型。此外，单个管理器可以通过网络由不同计算机上的进程共享。但是，它们比使用共享内存慢。

使用工作进程

Pool 类表示一个工作进程池。它具有允许以几种不同方式将任务分配到工作进程的方法。

例如：

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # 启动 4 个工作进程
    with Pool(processes=4) as pool:

        # 打印 "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # 以任意顺序打印同样的数字
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # 异步地对 "f(20)" 求值
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # 异步地对 "os.getpid()" 求值
        res = pool.apply_async(os.getpid, ()) # *仅在* 一个进程中运行
        print(res.get(timeout=1))             # 打印进程的 PID

        # 异步地进行多次求值 *可能* 会使用更多进程
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # 让一个工作进程休眠 10 秒
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
```

(续下页)

(接上页)

```
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # 退出 'with' 代码块将停止进程池
    print("Now the pool is closed and no longer available")
```

请注意，进程池的方法只能由创建它的进程使用。

备注

这个包中的功能要求子进程可以导入 __main__ 模块。虽然这在编程指导 中有描述，但还是需要提前说明一下。这意味着一些示例在交互式解释器中不起作用，比如*multiprocessing.pool.Pool* 示例。例如:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...     p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
Traceback (most recent call last):
Traceback (most recent call last):
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_importlib.
↪BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_importlib.
↪BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_importlib.
↪BuiltinImporter'>)>
```

（如果尝试执行上面的代码，它会以一种半随机的方式将三个完整的堆栈内容交替输出，然后你只能以某种方式停止父进程。）

18.2.2 参考

multiprocessing 包主要复制了*threading* 模块的 API。

Process 和异常

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

进程对象表示在单独进程中运行的活动。*Process* 类拥有和*threading.Thread* 等价的大部分方法。

构造器被调用时应当总是传入关键字参数。*group* 应当始终为 None；它的存在仅是为了与*threading.Thread* 兼容。*target* 是由*run()* 方法来唤起的可调用对象。它的默认值为 None，表示不调用任何东西。*name* 是进程名称（请参阅*name* 了解详情）。*args* 是针对目标调用的参数元组。*kwargs* 是针对目标调用的关键字参数字典。如果提供，则仅限关键字参数 *daemon* 会将进程的*daemon* 旗标设为 True 或 False。如果为 None (默认值)，则该旗标将从创建方进程继承。

在默认情况下，不会将任何参数传递给 *target*。*args* 参数默认值为 ()，可被用来指定要传递给 *target* 的参数列表或元组。

如果子类重写构造函数，它必须确保它在对进程执行任何其他操作之前调用基类构造函数 (Process.__init__())。

在 3.3 版本发生变更: 增加了 *daemon* 形参。

run()

表示进程活动的方法。

你可以在子类中重写此方法。标准*run()* 方法调用传递给对象构造函数的可调用对象作为目标参数（如果有），分别从 *args* 和 *kwargs* 参数中获取顺序和关键字参数。

使用列表或元组作为传给*Process* 的 *args* 参数可以达成同样的效果。

示例：

```
>>> from multiprocessing import Process
>>> p = Process(target=print, args=[1])
>>> p.run()
1
>>> p = Process(target=print, args=(1,))
>>> p.run()
1
```

start()

启动进程活动。

这个方法每个进程对象最多只能调用一次。它会将对象的*run()* 方法安排在一个单独的进程中调用。

join([*timeout*])

如果可选参数 *timeout* 是 None （默认值），则该方法将阻塞，直到调用*join()* 方法的进程终止。如果 *timeout* 是一个正数，它最多会阻塞 *timeout* 秒。请注意，如果进程终止或方法超时，则该方法返回 None 。检查进程的*exitcode* 以确定它是否终止。

一个进程可以被 join 多次。

进程无法 join 自身，因为这会导致死锁。尝试在启动进程之前 join 进程是错误的。

name

进程的名称。该名称是一个字符串，仅用于识别目的。它没有语义。可以为多个进程指定相同的名称。

初始名称由构造器设定。如果没有为构造器提供显式名称，则会构造一个形式为'Process-N_1:N_2:...:N_k' 的名称，其中每个 N_k 是其父亲的第 N 个孩子。

is_alive()

返回进程是否还活着。

粗略地说，从*start()* 方法返回到子进程终止之前，进程对象仍处于活动状态。

daemon

进程的守护标志，一个布尔值。这必须在*start()* 被调用之前设置。

初始值继承自创建进程。

当进程退出时，它会尝试终止其所有守护进程子进程。

请注意，不允许在守护进程中创建子进程。这是因为当守护进程由于父进程退出而中断时，其子进程会变成孤儿进程。另外，这些 **不是** Unix 守护进程或服务，它们是正常进程，如果非守护进程已经退出，它们将被终止（并且不被合并）。

除了*threading.Thread* API ， *Process* 对象还支持以下属性和方法：

pid

返回进程 ID。在生成该进程之前，这将是 None 。

exitcode

子进程的退出代码。如果该进程尚未终止则为 None 。

如果子进程的*run()* 方法正常返回，退出代码将是 0 。如果它通过*sys.exit()* 终止，并有一个整数参数 *N* ，退出代码将是 *N* 。

如果子进程由于在*run()* 内的未捕获异常而终止，退出代码将是 1 。如果它是由信号 *N* 终止的，退出代码将是负值 *-N* 。

authkey

进程的身份验证密钥（字节字符串）。

当*multiprocessing* 初始化时，主进程使用*os.urandom()* 分配一个随机字符串。

当创建*Process* 对象时，它将继承其父进程的身份验证密钥，尽管可以通过将*authkey* 设置为另一个字节字符串来更改。

参见*认证密码* 。

sentinel

系统对象的数字句柄，当进程结束时将变为"ready" 。

如果你想使用*multiprocessing.connection.wait()* 来一次等待多个事件则可以使用此值。在其他情况下调用*join()* 更为简单。

在 Windows 上，这是一个可以与 WaitForSingleObject 和 WaitForMultipleObjects API 调用族一起使用的 OS 句柄。在 POSIX 上，这是一个可以与来自*select* 模块的原语一起使用的文件描述符。

Added in version 3.3.

terminate()

终结进程。在 POSIX 上这是使用*SIGTERM* 信号来完成的；在 Windows 上则会使用 TerminateProcess()。请注意 exit 处理器和 finally 子句等将不会被执行。

请注意，进程的后代进程将不会被终止——它们将简单地变成孤立的。

> **警告**
>
> 如果在关联进程使用管道或队列时使用此方法，则管道或队列可能会损坏，并可能无法被其他进程使用。类似地，如果进程已获得锁或信号量等，则终止它可能导致其他进程死锁。

kill()

与*terminate()* 相同但在 POSIX 上将使用 SIGKILL 信号。

Added in version 3.7.

close()

关闭*Process* 对象，释放与之关联的所有资源。如果底层进程仍在运行，则会引发*ValueError* 。一旦*close()* 成功返回，*Process* 对象的大多数其他方法和属性将引发*ValueError* 。

Added in version 3.7.

注意*start()* 、*join()* 、*is_alive()* 、*terminate()* 和*exitcode* 方法只能由创建进程对象的进程调用。

Process 一些方法的示例用法：

```
>>> import multiprocessing, time, signal
>>> mp_context = multiprocessing.get_context('spawn')
>>> p = mp_context.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<...Process ... initial> False
>>> p.start()
```

(续下页)

(接上页)

```
>>> print(p, p.is_alive())
<...Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<...Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**

所有*multiprocessing* 异常的基类。

exception multiprocessing.**BufferTooShort**

当所提供的缓冲区对象太小而无法读取消息时由 `Connection.recv_bytes_into()` 引发的异常。

如果 `e` 是一个*BufferTooShort* 实例，那么 `e.args[0]` 将把消息作为字节字符串给出。

exception multiprocessing.**AuthenticationError**

出现身份验证错误时引发。

exception multiprocessing.**TimeoutError**

有超时的方法超时时引发。

管道和队列

使用多进程时，一般使用消息机制实现进程间通信，尽可能避免使用同步原语，例如锁。

消息机制包含：*Pipe()* (可以用于在两个进程间传递消息)，以及队列 (能够在多个生产者和消费者之间通信)。

Queue, *SimpleQueue* 以及*JoinableQueue* 都是多生产者，多消费者，并且实现了 FIFO 的队列类型，其表现与标准库中的*queue.Queue* 类相似。不同之处在于*Queue* 缺少标准库的*queue.Queue* 从 Python 2.5 开始引入的*task_done()* 和*join()* 方法。

如果你使用了*JoinableQueue* ，那么你 **必须** 对每个已经移出队列的任务调用*JoinableQueue.task_done()*。不然的话用于统计未完成任务的信号量最终会溢出并抛出异常。

与其他 Python 队列实现的区别之一，在于*multiprocessing* 队列会使用*pickle* 来序列化所有被放入的对象。由获取方法所返回的对象是重新创建的对象，它不会与原始对象共享内存。

另外还可以通过使用一个管理器对象创建一个共享队列，详见管理器 。

备注

multiprocessing 使用了普通的*queue.Empty* 和*queue.Full* 异常去表示超时。你需要从*queue* 中导入它们，因为它们并不在*multiprocessing* 的命名空间中。

备注

当一个对象被放入一个队列中时，这个对象首先会被一个后台线程用 pickle 序列化，并将序列化后的数据通过一个底层管道的管道传递到队列中。这种做法会有点让人惊讶，但一般不会出现什么问题。如果它们确实妨碍了你，你可以使用一个由管理器*manager* 创建的队列替换它。

(1) 将一个对象放入一个空队列后，可能需要极小的延迟，队列的方法*empty()* 才会返回*False* 。而*get_nowait()* 可以不抛出*queue.Empty* 直接返回。

(2) 如果有多个进程同时将对象放入队列，那么在队列的另一端接受到的对象可能是无序的。但是由同一个进程放入的多个对象的顺序在另一端输出时总是一样的。

警告

如果一个进程在尝试使用*Queue*期间被*Process.terminate()*或*os.kill()*调用终止了，那么队列中的数据很可能被破坏。这可能导致其他进程在尝试使用该队列时发生异常。

警告

正如刚才提到的，如果一个子进程将一些对象放进队列中 (并且它没有用*JoinableQueue.cancel_join_thread*方法)，那么这个进程在所有缓冲区的对象被刷新进管道之前，是不会终止的。

这意味着，除非你确定所有放入队列中的对象都已经被消费了，否则如果你试图等待这个进程，你可能会陷入死锁中。相似地，如果该子进程不是后台进程，那么父进程可能在试图等待所有非后台进程退出时挂起。

注意用管理器创建的队列不存在这个问题，详见*编程指导*。

该*例子*展示了如何使用队列实现进程间通信。

`multiprocessing.`**`Pipe`**`(`[*duplex*]`)`

返回一对*Connection*对象 `(conn1, conn2)`，分别表示管道的两端。

如果 *duplex* 被置为 `True` (默认值)，那么该管道是双向的。如果 *duplex* 被置为 `False`，那么该管道是单向的，即 `conn1` 只能用于接收消息，而 `conn2` 仅能用于发送消息。

The `send()` 方法将使用*pickle*来序列化对象而 `recv()` 将重新创建对象。

`class` `multiprocessing.`**`Queue`**`(`[*maxsize*]`)`

返回一个使用一个管道和少量锁和信号量实现的共享队列实例。当一个进程将一个对象放进队列中时，一个写入线程会启动并将对象从缓冲区写入管道中。

一旦超时，将抛出标准库*queue*模块中常见的异常*queue.Empty*和*queue.Full*。

除了*task_done()*和*join()*之外，*Queue*实现了标准库类*queue.Queue*中所有的方法。

`qsize``()`

返回队列的大致长度。由于多线程或者多进程的上下文，这个数字是不可靠的。

请注意这可能会在未实现 `sem_getvalue()` 的平台如 macOS 上引发*NotImplementedError*。

`empty``()`

如果队列是空的，返回 `True`，反之返回 `False`。由于多线程或多进程的环境，该状态是不可靠的。

在已关闭的队列上可能会引发*OSError*。(但不保证如此)

`full``()`

如果队列是满的，返回 `True`，反之返回 `False`。由于多线程或多进程的环境，该状态是不可靠的。

`put``(`*obj*[, *block*[, *timeout*]]`)`

将 obj 放入队列。如果可选参数 *block* 是 `True` (默认值) 而且 *timeout* 是 `None` (默认值)，将会阻塞当前进程，直到有空的缓冲槽。如果 *timeout* 是正数，将会在阻塞了最多 *timeout* 秒之后还是没有可用的缓冲槽时抛出*queue.Full*异常。反之 (*block* 是 `False` 时)，仅当有可用缓冲槽时才放入对象，否则抛出*queue.Full*异常 (在这种情形下 *timeout* 参数会被忽略)。

在 3.8 版本发生变更: 如果队列已经关闭，会抛出*ValueError*而不是*AssertionError*。

`put_nowait``(`*obj*`)`

相当于 `put(obj, False)`。

get(*[block[, timeout]]*)

从队列中取出并返回对象。如果可选参数 *block* 是 `True` (默认值) 而且 *timeout* 是 `None` (默认值), 将会阻塞当前进程，直到队列中出现可用的对象。如果 *timeout* 是正数，将会在阻塞了最多 *timeout* 秒之后还是没有可用的对象时抛出*queue.Empty* 异常。反之 (*block* 是 `False` 时)，仅当有可用对象能够取出时返回，否则抛出*queue.Empty* 异常 (在这种情形下 *timeout* 参数会被忽略)。

在 3.8 版本发生变更: 如果队列已经关闭，会抛出*ValueError* 而不是*OSError* 。

get_nowait()

相当于 `get(False)` 。

multiprocessing.Queue 类有一些在*queue.Queue* 类中没有出现的方法。这些方法在大多数情形下并不是必须的。

close()

指示当前进程将不会再往队列中放入对象。一旦所有缓冲区中的数据被写入管道之后，后台的线程会退出。这个方法在队列被 gc 回收时会自动调用。

join_thread()

等待后台线程。这个方法仅在调用了*close()* 方法之后可用。这会阻塞当前进程，直到后台线程退出，确保所有缓冲区中的数据都被写入管道中。

默认情况下，如果一个不是队列创建者的进程试图退出，它会尝试等待这个队列的后台线程。这个进程可以使用*cancel_join_thread()* 让*join_thread()* 方法什么都不做直接跳过。

cancel_join_thread()

防止*join_thread()* 方法阻塞当前进程。具体而言，这防止进程退出时自动等待后台线程退出。详见*join_thread()*。

这个方法更好的名字可能是 `allow_exit_without_flush()`。这可能会导致已排入队列的数据丢失，几乎可以肯定你将不需要用到这个方法。实际上它仅适用于当你需要当前进程立即退出而不必等待将已排入的队列更新到下层管道，并且你不担心丢失数据的时候。

备注

该类的功能依赖于宿主操作系统具有可用的共享信号量实现。否则该类将被禁用，任何试图实例化一个*Queue* 对象的操作都会抛出*ImportError* 异常，更多信息详见 bpo-3770 。后续说明的任何专用队列对象亦如此。

class multiprocessing.**SimpleQueue**

这是一个简化的*Queue* 类的实现，很像带锁的*Pipe* 。

close()

关闭队列：释放内部资源。

队列在被关闭后就不可再被使用。例如不可再调用*get()*, *put()* 和*empty()* 等方法。

Added in version 3.9.

empty()

如果队列为空返回 `True` ， 否则返回 `False` 。

如果 SimpleQueue 已关闭则总是会引发*OSError*。

get()

从队列中移出并返回一个对象。

put(*item*)

将 *item* 放入队列。

class multiprocessing.**JoinableQueue**([*maxsize*])

JoinableQueue 类是*Queue* 的子类，额外添加了*task_done()* 和*join()* 方法。

task_done()

指出之前进入队列的任务已经完成。由队列的消费者进程使用。对于每次调用*get()* 获取的任务，执行完成后调用*task_done()* 告诉队列该任务已经处理完成。

如果*join()* 方法正在阻塞之中，该方法会在所有对象都被处理完的时候返回 (即对之前使用*put()* 放进队列中的所有对象都已经返回了对应的*task_done()*)。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 异常。

join()

阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费者进程调用*task_done()* 表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，join() 阻塞被解除。

杂项

multiprocessing.**active_children**()

返回当前进程存活的子进程的列表。

调用该方法有“等待”已经结束的进程的副作用。

multiprocessing.**cpu_count**()

返回系统的 CPU 数量。

该数值不等于当前进程可使用的 CPU 数量。可用的 CPU 数量可以通过*os.process_cpu_count()* (或 len(os.sched_getaffinity(0))) 获得。

当 CPU 的数量无法确定时，会引发*NotImplementedError* 。

> **参见**
>
> *os.cpu_count()* *os.process_cpu_count()*

在 3.13 版本发生变更: 返回值也可使用 -X cpu_count 旗标或 PYTHON_CPU_COUNT 来覆盖因为这只是一个针对*os* cpu count API 的包装器。

multiprocessing.**current_process**()

返回与当前进程相对应的*Process* 对象。

和*threading.current_thread()* 相同。

multiprocessing.**parent_process**()

返回父进程*Process* 对象，和父进程调用*current_process()* 返回的对象一样。如果一个进程已经是主进程，parent_process 会返回 None.

Added in version 3.8.

multiprocessing.**freeze_support**()

为使用了*multiprocessing* 的程序，提供冻结以产生 Windows 可执行文件的支持。(在 **py2exe**, **PyInstaller** 和 **cx_Freeze** 上测试通过)

需要在 main 模块的 if __name__ == '__main__' 该行之后马上调用该函数。例如:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')
```

(续下页)

(接上页)

```
if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

如果没有调用 `freeze_support()` 在尝试运行被冻结的可执行文件时会抛出*RuntimeError* 异常。

对 `freeze_support()` 的调用在非 Windows 平台上是无效的。如果该模块在 Windows 平台的 Python 解释器中正常运行 (该程序没有被冻结)，调用 `freeze_support()` 也是无效的。

`multiprocessing.`**`get_all_start_methods`**`()`

返回由受支持的启动方法组成的列表，其中第一项将为默认值。可用的启动方法有 `'fork'`，`'spawn'` 和 `'forkserver'`。并非所有的平台都支持所有的方法。参见上下文和启动方法。

Added in version 3.4.

`multiprocessing.`**`get_context`**`(`*method=None*`)`

返回一个 Context 对象。该对象具有和*multiprocessing* 模块相同的 API。

如果 *method* 为 `None` 则将返回默认的上下文。否则 *method* 应为 `'fork'`，`'spawn'`，`'forkserver'`。如果指定的启动方法不可用则将引发*ValueError*。参见上下文和启动方法。

Added in version 3.4.

`multiprocessing.`**`get_start_method`**`(`*allow_none=False*`)`

返回启动进程时使用的启动方法名。

如果启动方法已经固定，并且 *allow_none* 被设置成 False ，那么启动方法将被固定为默认的启动方法，并且返回其方法名。如果启动方法没有设定，并且 *allow_none* 被设置成 True ，那么将返回 `None` 。

返回值可以为 `'fork'`，`'spawn'`，`'forkserver'` 或 `None`。参见上下文和启动方法。

Added in version 3.4.

在 3.8 版本发生变更: 对于 macOS，*spawn* 启动方式是默认方式。因为 *fork* 可能导致 subprocess 崩溃，被认为是不安全的，查看 bpo-33725 。

`multiprocessing.`**`set_executable`**`(`*executable*`)`

设置在启动子进程时使用的 Python 解释器路径。(默认使用*sys.executable*) 嵌入式编程人员可能需要这样做:

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

以使他们可以创建子进程。

在 3.4 版本发生变更: 当使用 `'spawn'` 启动方法时在 POSIX 上受到支持。

在 3.11 版本发生变更: 接受一个*path-like object*。

`multiprocessing.`**`set_forkserver_preload`**`(`*module_names*`)`

为 forkserver 主进程设置一个可尝试导入的模块名称列表以使得它们已导入的状态被分叉进程所继承。当执行操作时引发的任何*ImportError* 会被静默地忽略。这可被用作一种性能增强措施以避免在每个进程中的重复操作。

要让此方法发挥作用，它必须在 forkserver 进程执行之前被调用（在创建 `Pool` 或启动*Process* 之前）。

仅在使用 `'forkserver'` 启动方法时是有意义的。参见上下文和启动方法。

Added in version 3.4.

`multiprocessing.`**`set_start_method`**`(`*method*, *force=False*`)`

设置应当被用于启动子进程的方法。*method* 方法可以为 `'fork'`，`'spawn'` 或 `'forkserver'`。如果启动方法已经设置且 *force* 不为 `True` 则会引发*RuntimeError*。如果 *method* 为 `None` 而 *force* 为 `True` 则启动方法会被设为 `None`。如果 *method* 为 `None` 而 *force* 为 `False` 则上下文会被设为默认的上下文。

注意这最多只能调用一次，并且需要藏在 main 模块中，由 `if __name__ == '__main__'` 保护着。

参见上下文和启动方法。

Added in version 3.4.

备注

multiprocessing 并没有包含类似*threading.active_count()*，*threading.enumerate()*，*threading.settrace()*，*threading.setprofile()*，*threading.Timer*，或者*threading.local*的方法和类。

连接对象（Connection）

Connection 对象允许收发可以序列化的对象或字符串。它们可以看作面向消息的连接套接字。

通常使用*Pipe*创建 Connection 对象。详见：监听器及客户端.

class multiprocessing.connection.**Connection**

send(*obj*)

将一个对象发送到连接的另一端，可以用*recv()*读取。

发送的对象必须是可以序列化的，过大的对象（接近 32MiB+，这个值取决于操作系统）有可能引发*ValueError*异常。

recv()

返回一个由另一端使用*send()*发送的对象。该方法会一直阻塞直到接收到对象。如果对端关闭了连接或者没有东西可接收，将抛出*EOFError*异常。

fileno()

返回由连接对象使用的描述符或者句柄。

close()

关闭连接对象。

当连接对象被垃圾回收时会自动调用。

poll([*timeout*])

返回连接对象中是否有可以读取的数据。

如果未指定 *timeout*，此方法会马上返回。如果 *timeout* 是一个数字，则指定了最大阻塞的秒数。如果 *timeout* 是 None，那么将一直等待，不会超时。

注意通过使用*multiprocessing.connection.wait()*可以一次轮询多个连接对象。

send_bytes(*buffer*[, *offset*[, *size*]])

从一个*bytes-like object*对象中取出字节数组并作为一条完整消息发送。

如果由 *offset* 给定了在 *buffer* 中读取数据的位置。如果给定了 *size*，那么将会从缓冲区中读取多个字节。过大的缓冲区（接近 32MiB+，此值依赖于操作系统）有可能引发*ValueError*异常。

recv_bytes([*maxlength*])

以字符串形式返回一条从连接对象另一端发送过来的字节数据。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError*异常。

如果给定了 *maxlength* 并且消息长于 *maxlength* 那么将抛出*OSError*并且该连接对象将不再可读。

在 3.3 版本发生变更: 曾经该函数抛出*IOError*，现在这是*OSError*的别名。

recv_bytes_into(*buffer*[, *offset*])

将一条完整的字节数据消息读入 *buffer* 中并返回消息的字节数。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError* 异常。

buffer must be a writable *bytes-like object*. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

如果缓冲区太小，则将引发 `BufferTooShort` 异常，并且完整的消息将会存放在异常实例 `e` 的 `e.args[0]` 中。

在 3.3 版本发生变更: 现在连接对象自身可以通过*Connection.send()* 和*Connection.recv()* 在进程之间传递。

连接对象现在支持上下文管理协议 -- 参见*上下文管理器类型*。*__enter__()* 返回连接对象，而*__exit__()* 将调用*close()*。

例如:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告

Connection.recv() 方法会自动解封它收到的数据，除非你能够信任发送消息的进程，否则此处可能有安全风险。

因此，除非连接对象是由 `Pipe()` 产生的，否则你应该仅在使用了某种认证手段之后才使用*recv()* 和*send()* 方法。参考*认证密码*。

警告

如果一个进程在试图读写管道时被终止了，那么管道中的数据很可能是不完整的，因为此时可能无法确定消息的边界。

同步原语

通常来说同步原语在多进程环境中并不像它们在多线程环境中那么必要。参考*threading* 模块的文档。

注意可以使用管理器对象创建同步原语，参考*管理器* 。

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])

类似*threading.Barrier* 的栅栏对象。

Added in version 3.3.

class multiprocessing.**BoundedSemaphore**([*value*])

非常类似*threading.BoundedSemaphore* 的有界信号量对象。

一个小小的不同在于，它的 `acquire` 方法的第一个参数名是和*Lock.acquire()* 一样的 *block* 。

> **备注**
>
> 在 macOS 平台上，该对象于*Semaphore* 不同在于 `sem_getvalue()` 方法并没有在该平台上实现。

class multiprocessing.**Condition**([*lock*])

条件变量： *threading.Condition* 的别名。

指定的 *lock* 参数应该是*multiprocessing* 模块中的*Lock* 或者*RLock* 对象。

在 3.3 版本发生变更: 新增了*wait_for()* 方法。

class multiprocessing.**Event**

A clone of *threading.Event*.

class multiprocessing.**Lock**

原始锁（非递归锁）对象，类似于*threading.Lock* 。一旦一个进程或者线程拿到了锁，后续的任何其他进程或线程的其他请求都会被阻塞直到锁被释放。任何进程或线程都可以释放锁。除非另有说明，否则*multiprocessing.Lock* 用于进程或者线程的概念和行为都和*threading.Lock* 一致。

注意*Lock* 实际上是一个工厂函数。它返回由默认上下文初始化的 `multiprocessing.synchronize.Lock` 对象。

Lock supports the *context manager* protocol and thus may be used in `with` statements.

acquire(*block=True*, *timeout=None*)

可以阻塞或非阻塞地获得锁。

如果 *block* 参数被设为 `True` (默认值) ，对该方法的调用在锁处于释放状态之前都会阻塞，然后将锁设置为锁住状态并返回 `True` 。需要注意的是第一个参数名与*threading.Lock.acquire()* 的不同。

如果 *block* 参数被设置成 `False` ，方法的调用将不会阻塞。如果锁当前处于锁住状态，将返回 `False` ；否则将锁设置成锁住状态，并返回 `True` 。

当 *timeout* 是一个正浮点数时，会在等待锁的过程中最多阻塞等待 *timeout* 秒，当 *timeout* 是负数时，效果和 *timeout* 为 0 时一样，当 *timeout* 是 `None` （默认值）时，等待时间是无限长。需要注意的是，对于 *timeout* 参数是负数和 `None` 的情况，其行为与*threading.Lock.acquire()* 是不一样的。当 *block* 参数为 `False` 时，*timeout* 并没有实际用处，会直接忽略。否则，函数会在拿到锁后返回 `True` 或者超时没拿到锁后返回 `False` 。

release()

释放锁，可以在任何进程、线程使用，并不限于锁的拥有者。

当尝试释放一个没有被持有的锁时，会抛出*ValueError* 异常，除此之外其行为与*threading.Lock.release()* 一样。

class multiprocessing.**RLock**

递归锁对象: 类似于*threading.RLock* 。递归锁必须由持有线程、进程亲自释放。如果某个进程或者线程拿到了递归锁，这个进程或者线程可以再次拿到这个锁而不需要等待。但是这个进程或者线程的拿锁操作和释放锁操作的次数必须相同。

注意*RLock* 是一个工厂函数，调用后返回一个使用默认 context 初始化的 `multiprocessing.synchronize.RLock` 实例。

RLock 支持*context manager* 协议，因此可在 `with` 语句内使用。

acquire(*block=True*, *timeout=None*)

可以阻塞或非阻塞地获得锁。

当 *block* 参数设置为 `True` 时，会一直阻塞直到锁处于空闲状态（没有被任何进程、线程拥有），除非当前进程或线程已经拥有了这把锁。然后当前进程/线程会持有这把锁（在锁没有其他持有者的情况下），锁内的递归等级加一，并返回 `True`。注意，这个函数第一个参数的行为和*threading.RLock.acquire()* 的实现有几个不同点，包括参数名本身。

当 *block* 参数是 `False`，将不会阻塞，如果此时锁被其他进程或者线程持有，当前进程、线程获取锁操作失败，锁的递归等级也不会改变，函数返回 `False`，如果当前锁已经处于释放状态，则当前进程、线程则会拿到锁，并且锁内的递归等级加一，函数返回 `True`。

timeout 参数的使用方法及行为与*Lock.acquire()* 一样。但是要注意 *timeout* 的其中一些行为和*threading.RLock.acquire()* 中实现的行为是不同的。

release()

释放锁，使锁内的递归等级减一。如果释放后锁内的递归等级降低为 0，则会重置锁的状态为释放状态（即没有被任何进程、线程持有），重置后如果有有其他进程和线程在等待这把锁，他们中的一个会获得这个锁而继续运行。如果释放后锁内的递归等级还没到达 0，则这个锁仍将保持未释放状态且当前进程和线程仍然是持有者。

只有当前进程或线程是锁的持有者时，才允许调用这个方法。如果当前进程或线程不是这个锁的拥有者，或者这个锁处于已释放的状态 (即没有任何拥有者)，调用这个方法会抛出*AssertionError* 异常。注意这里抛出的异常类型和*threading.RLock.release()* 中实现的行为不一样。

class multiprocessing.**Semaphore**([*value*])

一种信号量对象: 类似于*threading.Semaphore*.

一个小小的不同在于，它的 `acquire` 方法的第一个参数名是和*Lock.acquire()* 一样的 *block*。

备注

在 macOS 上，不支持 `sem_timedwait`，所以，调用 `acquire()` 时如果使用 timeout 参数，会通过循环 sleep 来模拟这个函数的行为。

备注

这个包的某些功能依赖于宿主机系统的共享信号量的实现，如果系统没有这个特性，`multiprocessing.synchronize` 会被禁用，尝试导入这个模块会引发*ImportError* 异常，详细信息请查看 bpo-3770。

共享 ctypes 对象

在共享内存上创建可被子进程继承的共享对象时是可行的。

multiprocessing.**Value**(*typecode_or_type*, **args*, *lock=True*)

返回一个从共享内存上创建的*ctypes* 对象。默认情况下返回的对象实际上是经过了同步器包装过的。可以通过*Value* 的 *value* 属性访问这个对象本身。

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

如果 *lock* 参数是 `True`（默认值），将会新建一个递归锁用于同步对于此值的访问操作。如果 *lock* 是*Lock* 或者*RLock* 对象，那么这个传入的锁将会用于同步对这个值的访问操作，如果 *lock* 是 `False`，那么对这个对象的访问将没有锁保护，也就是说这个变量不是进程安全的。

诸如 `+=` 这类的操作会引发独立的读操作和写操作，也就是说这类操作符并不具有原子性。所以，如果你想让递增共享变量的操作具有原子性，仅仅以这样的方式并不能达到要求:

```
counter.value += 1
```

共享对象内部关联的锁是递归锁 (默认情况下就是) 的情况下，你可以采用这种方式

```
with counter.get_lock():
    counter.value += 1
```

注意 *lock* 只能是命名参数。

multiprocessing.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

从共享内存中申请并返回一个具有 ctypes 类型的数组对象。默认情况下返回值实际上是被同步器包装过的数组对象。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的每一个元素，并且会根据元素个数自动判断数组的长度。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对返回对象的访问将不会自动得到锁的保护，也就是说它不是“进程安全的”。

请注意 *lock* 是一个仅限关键字参数。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串。

multiprocessing.sharedctypes 模块

multiprocessing.sharedctypes 模块提供了一些函数，用于分配来自共享内存的、可被子进程继承的*ctypes* 对象。

备注

虽然可以将指针存储在共享内存中，但请记住它所引用的是特定进程地址空间中的位置。而且，指针很可能在第二个进程的上下文中无效，尝试从第二个进程对指针进行解引用可能会导致崩溃。

multiprocessing.sharedctypes.**RawArray**(*typecode_or_type*, *size_or_initializer*)

从共享内存中申请并返回一个 ctypes 数组。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中使用的类型字符。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的每一个元素，并且会根据元素个数自动判断数组的长度。

注意对元素的访问、赋值操作可能是非原子操作 - 使用*Array()* ，从而借助其中的锁保证操作的原子性。

multiprocessing.sharedctypes.**RawValue**(*typecode_or_type*, **args*)

从共享内存中申请并返回一个 ctypes 对象。

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

注意对 value 的访问、赋值操作可能是非原子操作 - 使用*Value()* ，从而借助其中的锁保证操作的原子性。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串 - 请查看*ctypes* 文档。

multiprocessing.sharedctypes.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的进程安全的对象，这取决于 *lock* 参数的值，除此之外，和*RawArray()* 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对所返回对象的访问将不会自动得到锁的保护，也就是说它将不是“进程安全的”。

注意 *lock* 只能是命名参数。

`multiprocessing.sharedctypes.`**`Value`**(*typecode_or_type*, **args*, *lock=True*)

返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的进程安全的对象，这取决于 *lock* 参数的值，除此之外，和 `RawArray()` 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对所返回对象的访问将不会自动得到锁的保护，也就是说它将不是“进程安全的”。

注意 *lock* 只能是命名参数。

`multiprocessing.sharedctypes.`**`copy`**(*obj*)

从共享内存中申请一片空间将 ctypes 对象 *obj* 过来，然后返回一个新的 ctypes 对象。

`multiprocessing.sharedctypes.`**`synchronized`**(*obj*[, *lock*])

将一个 ctypes 对象包装为进程安全的对象并返回，使用 *lock* 同步对于它的操作。如果 *lock* 是 `None` (默认值)，则会自动创建一个*multiprocessing.RLock* 对象。

同步器包装后的对象会在原有对象基础上额外增加两个方法: `get_obj()` 返回被包装的对象，`get_lock()` 返回内部用于同步的锁。

需要注意的是，访问包装后的 ctypes 对象会比直接访问原来的纯 ctypes 对象慢得多。

在 3.5 版本发生变更: 同步器包装后的对象支持*context manager* 协议。

下面的表格对比了创建普通 ctypes 对象和基于共享内存上创建共享 ctypes 对象的语法。（表格中的 `MyStruct` 是*ctypes.Structure* 的子类）

ctypes	使用类型的共享 ctypes	使用 typecode 的共享 ctypes
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

下面是一个在子进程中修改多个 ctypes 对象的例子。

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
```

(续下页)

(接上页)

```
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

输出如下

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

管理器

管理器提供了一种创建共享数据的方法，从而可以在不同进程中共享，甚至可以通过网络跨机器共享数据。管理器维护一个用于管理 共享对象的服务。其他进程可以通过代理访问这些共享对象。

multiprocessing.**Manager**()

返回一个已启动的*SyncManager* 管理器对象，这个对象可以用于在不同进程中共享数据。返回的管理器对象对应了一个已经启动的子进程，并且拥有一系列方法可以用于创建共享对象、返回对应的代理。

当管理器被垃圾回收或者父进程退出时，管理器进程会立即退出。管理器类定义在*multiprocessing.managers* 模块:

class multiprocessing.managers.**BaseManager**(*address=None*, *authkey=None*, *serializer='pickle'*, *ctx=None*, ***, *shutdown_timeout=1.0*)

创建一个 BaseManager 对象。

一旦创建，应该及时调用*start()* 或者 `get_server().serve_forever()` 以确保管理器对象对应的管理进程已经启动。

address 是管理器服务进程监听的地址。如果 *address* 是 `None`，则允许和任意主机的请求建立连接。

authkey 是认证标识，用于检查连接服务进程的请求合法性。如果 *authkey* 是 `None`，则会使用 `current_process().authkey`，否则，就使用 *authkey*，需要保证它必须是 byte 类型的字符串。

serializer 必须为 `'pickle'` (使用*pickle* 序列化) 或 `'xmlrpclib'` (使用*xmlrpc.client* 序列化)。

ctx 是一个上下文对象，或者为 `None` (使用当前上下文)。参见 `get_context()` 函数。

shutdown_timeout 是用于等待直到*shutdown()* 方法中的管理器所使用的进程结束的超时秒数。如果关闭超时，进程将被终结。如果终结进程的操作也超时，进程将被杀掉。

在 3.11 版本发生变更: 添加了 *shutdown_timeout* 形参。

start([*initializer*[, *initargs*]])

为管理器开启一个子进程，如果 *initializer* 不是 `None`，子进程在启动时将会调用 `initializer(*initargs)`。

get_server()

返回一个 `Server` 对象，它是管理器在后台控制的真实的服务。`Server` 对象拥有 `serve_forever()` 方法。

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
```

(续下页)

(接上页)

```
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server 额外拥有一个*address* 属性。

connect()

将本地管理器对象连接到一个远程管理器进程:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown()

停止管理器的进程。这个方法只能用于已经使用*start()* 启动的服务进程。

它可以被多次调用。

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])

一个 classmethod，可以将一个类型或者可调用对象注册到管理器类。

typeid 是一种" 类型标识符"，用于唯一表示某种共享对象类型，必须是一个字符串。

callable 是一个用来为此类型标识符创建对象的可调用对象。如果一个管理器实例将使用*connect()* 方法连接到服务器，或者 *create_method* 参数为 False，那么这里可留下 None。

proxytype 是*BaseProxy* 的子类，可以根据 *typeid* 为共享对象创建一个代理，如果是 None , 则会自动创建一个代理类。

exposed 是一个函数名组成的序列，用来指明只有这些方法可以使用*BaseProxy._callmethod()* 代理。(如果 *exposed* 是 None, 则会在 proxytype._exposed_ 存在的情况下转而使用它) 当暴露的方法列表没有指定的时候，共享对象的所有“公共方法”都会被代理。(这里的“公共方法”是指所有拥有 __call__() 方法并且不是以 '_' 开头的属性)

method_to_typeid 是一个映射，用来指定那些应该返回代理对象的暴露方法所返回的类型。(如果 *method_to_typeid* 是 None, 则 proxytype._method_to_typeid_ 会在存在的情况下被使用) 如果方法名称不在这个映射中或者映射是 None , 则方法返回的对象会是一个值拷贝。

create_method 指明，是否要创建一个以 *typeid* 命名并返回一个代理对象的方法，这个函数会被服务进程用于创建共享对象，默认为 True 。

BaseManager 实例也有一个只读属性。

address

管理器所用的地址。

在 3.3 版本发生变更: 管理器对象支持上下文管理协议 - 查看上下文管理器类型 。*__enter__()* 启动服务进程（如果它还没有启动）并且返回管理器对象，*__exit__()* 会调用*shutdown()* 。

在之前的版本中，如果管理器服务进程没有启动，*__enter__()* 不会负责启动它。

class multiprocessing.managers.**SyncManager**

BaseManager 的子类，可用于进程的同步。这个类型的对象使用*multiprocessing.Manager()* 创建。

它拥有一系列方法，可以为大部分常用数据类型创建并返回代理对象 代理，用于进程间同步。甚至包括共享列表和字典。

Barrier(*parties*[, *action*[, *timeout*]])

创建一个共享的*threading.Barrier* 对象并返回它的代理。

Added in version 3.3.

BoundedSemaphore([*value*])

创建一个共享的*threading.BoundedSemaphore* 对象并返回它的代理。

Condition([*lock*])

创建一个共享的*threading.Condition*对象并返回它的代理。

如果提供了 *lock* 参数，那它必须是*threading.Lock*或*threading.RLock*的代理对象。

在 3.3 版本发生变更: 新增了*wait_for()* 方法。

Event()

创建一个共享的*threading.Event*对象并返回它的代理。

Lock()

创建一个共享的*threading.Lock*对象并返回它的代理。

Namespace()

创建一个共享的*Namespace*对象并返回它的代理。

Queue([*maxsize*])

创建一个共享的*queue.Queue*对象并返回它的代理。

RLock()

创建一个共享的*threading.RLock*对象并返回它的代理。

Semaphore([*value*])

创建一个共享的*threading.Semaphore*对象并返回它的代理。

Array(*typecode*, *sequence*)

创建一个数组并返回它的代理。

Value(*typecode*, *value*)

创建一个具有可写 `value` 属性的对象并返回它的代理。

dict()

dict(*mapping*)

dict(*sequence*)

创建一个共享的*dict*对象并返回它的代理。

list()

list(*sequence*)

创建一个共享的*list*对象并返回它的代理。

在 3.6 版本发生变更: 共享对象能够嵌套。例如, 共享的容器对象如共享列表，可以包含另一个共享对象，他们全都会在*SyncManager*中进行管理和同步。

class multiprocessing.managers.**Namespace**

一个可以注册到*SyncManager*的类型。

命名空间对象没有公共方法，但是拥有可写的属性。直接 print 会显示所有属性的值。

值得一提的是，当对命名空间对象使用代理的时候，访问所有名称以 `'_'` 开头的属性都只是代理器上的属性，而不是命名空间对象的属性。

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # 这是该代理的一个属性
>>> print(Global)
Namespace(x=10, y='hello')
```

自定义管理器

要创建一个自定义的管理器，需要新建一个*BaseManager* 的子类，然后使用这个管理器类上的*register()* 类方法将新类型或者可调用方法注册上去。例如:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # 打印 7
        print(maths.mul(7, 8))         # 打印 56
```

使用远程管理器

可以将管理器服务运行在一台机器上，然后使用客户端从其他机器上访问。(假设它们的防火墙允许)

运行下面的代码可以启动一个服务，此付包含了一个共享队列，允许远程客户端访问:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

远程客户端可以通过下面的方式访问服务:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

也可以通过下面的方式:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

本地进程也可以访问这个队列，利用上面的客户端代码通过远程方式访问:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

代理对象

代理是一个 指向其他共享对象的对象，这个对象 (很可能) 在另外一个进程中。共享对象也可以说是代理指涉的对象。多个代理对象可能指向同一个指涉对象。

代理对象代理了指涉对象的一系列方法调用 (虽然并不是指涉对象的每个方法都有必要被代理)。通过这种方式，代理的使用方法可以和它的指涉对象一样:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

注意，对代理使用*str()* 函数会返回指涉对象的字符串表示，但是*repr()* 却会返回代理本身的内部字符串表示。

被代理的对象很重要的一点是必须可以被序列化，这样才能允许他们在进程间传递。因此，指涉对象可以包含*代理对象* 。这允许管理器中列表、字典或者其他*代理对象* 对象之间的嵌套。

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # a 的引用对象现在包含 b 的引用对象
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

类似地，字典和列表代理也可以相互嵌套:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
```

(续下页)

(接上页)

```
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

如果指涉对象包含了普通*list* 或*dict* 对象，对这些内部可变对象的修改不会通过管理器传播，因为代理无法得知被包含的值什么时候被修改了。但是把存放在容器代理中的值本身是会通过管理器传播的（会触发代理对象中的 __setitem__ ）从而有效修改这些对象，所以可以把修改过的值重新赋值给容器代理:

```
# 创建一个代理列表并添加一个可变对象（字典）
lproxy = manager.list()
lproxy.append({})
# 现在改变该字典
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# 这时，对 d 的改变尚未同步，但通过更新该字典，
# 代理将得到改变的通知
lproxy[0] = d
```

在大多是使用情形下，这种实现方式并不比嵌套*代理对象* 方便，但是依然演示了对于同步的一种控制级别。

备注

multiprocessing 中的代理类并没有提供任何对于代理值比较的支持。所以，我们会得到如下结果:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

当需要比较值的时候，应该替换为使用指涉对象的拷贝。

class multiprocessing.managers.**BaseProxy**

代理对象是*BaseProxy* 派生类的实例。

_callmethod(*methodname*[, *args*[, *kwds*]])

调用指涉对象的方法并返回结果。

如果 `proxy` 是一个代理且其指涉的是 `obj`，那么下面的表达式:

```
proxy._callmethod(methodname, args, kwds)
```

相当于求取以下表达式的值:

```
getattr(obj, methodname)(*args, **kwds)
```

于管理器进程。

返回结果会是一个值拷贝或者一个新的共享对象的代理 - 见函数*BaseManager.register()* 中关于参数 *method_to_typeid* 的文档。

如果这个调用熬出了异常，则这个异常会被*_callmethod()* 透传出来。如果是管理器进程本身抛出的一些其他异常，则会被*_callmethod()* 转换为 `RemoteError` 异常重新抛出。

特别注意，如果 *methodname* 没有 暴露出来，将会引发一个异常。

_callmethod() 的一个使用示例:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # 等价于 l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # 等价于 l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

返回指涉对象的一份拷贝。

如果指涉对象无法序列化，则会抛出一个异常。

__repr__()

返回代理对象的内部字符串表示。

__str__()

返回指涉对象的内部字符串表示。

清理

代理对象使用了一个弱引用回调函数，当它被垃圾回收时，会将自己从拥有此指涉对象的管理器上反注册，

当共享对象没有被任何代理器引用时，会被管理器进程删除。

进程池

可以创建一个进程池，它将使用*Pool*类执行提交给它的任务。

class multiprocessing.pool.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

一个进程池对象，它控制可以提交作业的工作进程池。它支持带有超时和回调的异步结果，以及一个并行的 map 实现。

processes 是要使用的工作进程数量。如果 *processes* 为 None 则使用*os.process_cpu_count()* 所返回的数值。

如果 *initializer* 不为 None，则每个工作进程将会在启动时调用 initializer(*initargs)。

maxtasksperchild 是一个工作进程在它退出或被一个新的工作进程代替之前能完成的任务数量，为了释放未使用的资源。默认的 *maxtasksperchild* 是 None，意味着工作进程寿与池齐。

context 可被用于指定启动的工作进程的上下文。通常一个进程池是使用函数 multiprocessing.Pool() 或者一个上下文对象的*Pool()* 方法创建的。在这两种情况下，*context* 都是适当设置的。

注意，进程池对象的方法只有创建它的进程能够调用。

> **警告**
>
> *multiprocessing.pool* 对象具有需要正确管理的内部资源（像任何其他资源一样），具体方式是将进程池用作上下文管理器，或者手动调用*close()* 和*terminate()*。未做此类操作将导致进程在终结阶段挂起。
>
> 请注意依赖垃圾回收器来销毁进程池是 **不正确的**做法，因为 CPython 并不保证进程池终结器会被调用（请参阅 object.__del__() 来了解详情）。

在 3.2 版本发生变更: 增加了 *maxtasksperchild* 形参。

在 3.4 版本发生变更: 增加了 *context* 形参。

在 3.13 版本发生变更: 在默认情况下 *processes* 将使用`os.process_cpu_count()`，而不是`os.cpu_count()`。

备注

通常来说，`Pool` 中的 Worker 进程的生命周期和进程池的工作队列一样长。一些其他系统中（如 Apache, mod_wsgi 等）也可以发现另一种模式，他们会让工作进程在完成一些任务后退出，清理、释放资源，然后启动一个新的进程代替旧的工作进程。`Pool` 的 *maxtasksperchild* 参数给用户提供了这种能力。

apply(*func*[, *args*[, *kwds*]])

使用 *args* 参数以及 *kwds* 命名参数调用 *func*，它会返回结果前阻塞。这种情况下，`apply_async()` 更适合并行化工作。另外 *func* 只会在一个进程池中的一个工作进程中执行。

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])

`apply()` 方法的一个变种，返回一个`AsyncResult` 对象。

如果指定了 *callback*，它必须是一个接受单个参数的可调用对象。当执行成功时，*callback* 会被用于处理执行后的返回结果，否则，调用 *error_callback*。

如果指定了 *error_callback*，它必须是一个接受单个参数的可调用对象。当目标函数执行失败时，会将抛出的异常对象作为参数传递给 *error_callback* 执行。

回调函数应该立即执行完成，否则会阻塞负责处理结果的线程。

map(*func*, *iterable*[, *chunksize*])

内置`map()` 函数的并行版本（但它只支持一个 *iterable* 参数，对于多个可迭代对象请参阅`starmap()`）。它会保持阻塞直到获得结果。

这个方法会将可迭代对象分割为许多块，然后提交给进程池。可以将 *chunksize* 设置为一个正整数来指定每个块的（近似）大小。

注意对于很长的迭代对象，可能消耗很多内存。可以考虑使用`imap()` 或`imap_unordered()` 并且显式指定 *chunksize* 以提升效率。

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

`map()` 方法的一个变种，返回一个`AsyncResult` 对象。

如果指定了 *callback*，它必须是一个接受单个参数的可调用对象。当执行成功时，*callback* 会被用于处理执行后的返回结果，否则，调用 *error_callback*。

如果指定了 *error_callback*，它必须是一个接受单个参数的可调用对象。当目标函数执行失败时，会将抛出的异常对象作为参数传递给 *error_callback* 执行。

回调函数应该立即执行完成，否则会阻塞负责处理结果的线程。

imap(*func*, *iterable*[, *chunksize*])

`map()` 的延迟执行版本。

chunksize 参数的作用和`map()` 方法的一样。对于很长的迭代器，给 *chunksize* 设置一个很大的值会比默认值 `1` **极大**地加快执行速度。

同样，如果 *chunksize* 是 `1`，那么`imap()` 方法所返回的迭代器的 `next()` 方法拥有一个可选的 *timeout* 参数：如果无法在 *timeout* 秒内执行得到结果，则 `next(timeout)` 会抛出`multiprocessing.TimeoutError` 异常。

imap_unordered(*func*, *iterable*[, *chunksize*])

和`imap()` 相同，只不过通过迭代器返回的结果是任意的。（当进程池中只有一个工作进程的时候，返回结果的顺序才能认为是” 有序” 的）

starmap(*func*, *iterable*[, *chunksize*])

和*map()* 类似，不过 *iterable* 中的每一项会被解包再作为函数参数。

比如可迭代对象 `[(1,2), (3, 4)]` 会转化为等价于 `[func(1,2), func(3,4)]` 的调用。

Added in version 3.3.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

相当于*starmap()* 与*map_async()* 的结合，迭代 *iterable* 的每一项，解包作为 *func* 的参数并执行，返回用于获取结果的对象。

Added in version 3.3.

close()

阻止后续任务提交到进程池，当所有任务执行完成后，工作进程会退出。

terminate()

不必等待未完成的任务，立即停止工作进程。当进程池对象被垃圾回收时，会立即调用*terminate()*。

join()

等待工作进程结束。调用*join()* 前必须先调用*close()* 或者*terminate()* 。

在 3.3 版本发生变更: 进程池对象现在支持上下文管理器协议 - 参见*上下文管理器类型* 。*__enter__()* 返回进程池对象，*__exit__()* 会调用*terminate()* 。

class multiprocessing.pool.**AsyncResult**

Pool.apply_async() 和*Pool.map_async()* 返回对象所属的类。

get([*timeout*])

用于获取执行结果。如果 *timeout* 不是 None 并且在 *timeout* 秒内仍然没有执行完得到结果，则抛出*multiprocessing.TimeoutError* 异常。如果远程调用发生异常，这个异常会通过*get()* 重新抛出。

wait([*timeout*])

阻塞，直到返回结果，或者 *timeout* 秒后超时。

ready()

返回执行状态，是否已经完成。

successful()

判断调用是否已经完成并且未引发异常。如果还未获得结果则将引发*ValueError*。

在 3.7 版本发生变更: 如果没有执行完，会抛出*ValueError* 异常而不是*AssertionError* 。

下面的例子演示了进程池的用法:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # 启动 4 个工作进程
        result = pool.apply_async(f, (10,)) # 在单个进程中异步地对 "f(10)" 求值
        print(result.get(timeout=1))         # 打印 "100" 除非你的计算机 *非常* 慢

        print(pool.map(f, range(10)))        # 打印 "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # 打印 "0"
        print(next(it))                      # 打印 "1"
        print(it.next(timeout=1))            # 打印 "4" 除非你的计算机 *非常* 慢
```

(续下页)

(接上页)

```
        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))        # 引发 multiprocessing.TimeoutError
```

监听器及客户端

通常情况下，进程间通过队列或者*Pipe()* 返回的*Connection* 传递消息。

不过，*multiprocessing.connection* 模块其实提供了一些更灵活的特性。最基础的用法是通过它抽象出来的高级 API 来操作 socket 或者 Windows 命名管道。也提供一些高级用法，如通过*hmac* 模块来支持摘要认证，以及同时监听多个管道连接。

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)

发送一个随机生成的消息到另一端，并等待回复。

如果收到的回复与使用 *authkey* 作为键生成的信息摘要匹配成功，就会发送一个欢迎信息给管道另一端。否则抛出*AuthenticationError* 异常。

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)

接收一条信息，使用 *authkey* 作为键计算信息摘要，然后将摘要发送回去。

如果没有收到欢迎消息，就抛出*AuthenticationError* 异常。

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])

尝试使用 *address* 地址上的监听器建立一个连接，返回*Connection* 。

连接的类型取决于 *family* 参数，但是通常可以省略，因为可以通过 *address* 的格式推导出来。(查看地址格式)

如果给出了 *authkey* 并且不为 None，则它应为一个字节串并且会被用作基于 HMAC 认证的密钥。如果 *authkey* 为 None 则不会执行认证。如果认证失败则会引发*AuthenticationError*。参见认证密码。

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])

可以监听连接请求，是对于绑定套接字或者 Windows 命名管道的封装。

address 是监听器对象中的绑定套接字或命名管道使用的地址。

> **备注**
>
> 如果使用'0.0.0.0' 作为监听地址，那么在 Windows 上这个地址无法建立连接。想要建立一个可连接的端点，应该使用'127.0.0.1' 。

family 是套接字 (或者命名管道) 使用的类型。它可以是以下一种: 'AF_INET' (TCP 套接字类型), 'AF_UNIX' (Unix 域套接字) 或者 'AF_PIPE' (Windows 命名管道)。其中只有第一个保证各平台可用。如果 *family* 是 None , 那么 family 会根据 *address* 的格式自动推导出来。如果 *address* 也是 None , 则取默认值。默认值为可用类型中速度最快的。见地址格式 。注意，如果 *family* 是 'AF_UNIX' 而 address 是 None , 套接字会在一个*tempfile.mkstemp()* 创建的私有临时目录中创建。

如果监听器对象使用了套接字，*backlog* (默认值为 1) 会在套接字绑定后传递给它的*listen()* 方法。

如果给出了 *authkey* 并且不为 None，则它应为一个字节串并且会被用作基于 HMAC 认证的密钥。如果 *authkey* 为 None 则不会执行认证。如果认证失败则会引发*AuthenticationError*。参见认证密码。

accept()

接受一个连接并返回一个*Connection* 对象，其连接到的监听器对象已绑定套接字或者命名管道。如果已经尝试过认证并且失败了，则会抛出*AuthenticationError* 异常。

close()

关闭监听器对象上的绑定套接字或者命名管道。此函数会在监听器被垃圾回收后自动调用。不过仍然建议显式调用函数关闭。

监听器对象拥有下列只读属性:

address

监听器对象使用的地址。

last_accepted

最后一个连接所使用的地址。如果没有的话就是 None 。

在 3.3 版本发生变更: 监听器对象现在支持了上下文管理协议 - 见上下文管理器类型 。*__enter__()* 返回一个监听器对象, *__exit__()* 会调用 *close()* 。

multiprocessing.connection.**wait**(*object_list*, *timeout=None*)

一直等待直到 *object_list* 中某个对象处于就绪状态。返回 *object_list* 中处于就绪状态的对象。如果 *timeout* 是一个浮点型，该方法会最多阻塞这么多秒。如果 *timeout* 是 None ，则会允许阻塞的事件没有限制。timeout 为负数的情况下和为 0 的情况相同。

对于 POSIX 和 Windows，满足下列条件的对象可以出现在 *object_list* 中

- 可读的 *Connection* 对象；
- 一个已连接并且可读的 *socket.socket* 对象；或者
- *Process* 对象中的 *sentinel* 属性。

当一个连接或者套接字对象拥有有效的数据可被读取的时候，或者另一端关闭后，这个对象就处于就绪状态。

POSIX: wait(object_list, timeout) 和 select.select(object_list, [], [], timeout) 几乎相同。差别在于，如果 *select.select()* 被信号中断，它会引发 *OSError* 并附带错误号 EINTR，而 *wait()* 则不会。

Windows: *object_list* 中的条目必须是一个可等待的整数句柄 (根据 Win32 函数 WaitForMultipleObjects() 文档所使用的定义) 或者一个具有 *fileno()* 方法的对象，该方法返回一个套接字句柄或管道句柄。(注意管道句柄和套接字句柄 **不是**可等待的句柄。)

Added in version 3.3.

示例

下面的服务代码创建了一个使用 'secret password' 作为认证密码的监听器。它会等待连接然后发送一些数据给客户端:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # 协议簇缩减为 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

下面的代码连接到服务然后从服务器上 j 接收一些数据:

```
from multiprocessing.connection import Client
from array import array
```

(续下页)

(接上页)

```
address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

下面的代码使用了*wait()*，以便在同时等待多个进程发来消息。

```
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # 现在我们关闭管道的可写端以确定
        # p 是拥有其所对应句柄的唯一进程。
        # 这将确保当 p 关闭可写端的句柄时，
        # wait() 将立即报告可读端已经就绪。
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

地址格式

- `'AF_INET'` 地址是 `(hostname, port)` 形式的元组类型，其中 *hostname* 是一个字符串，*port* 是整数。
- `'AF_UNIX'` 地址是文件系统上文件名的字符串。
- `'AF_PIPE'` 地址是一个 `r'\\.\pipe\`*PipeName*`'` 形式的字符串。要使用*Client()*来连接到远程计算机上一个名为 *ServerName* 的命名管道则应当改用 `r'\\`*ServerName*`\pipe\`*PipeName*`'` 形式的地址。

注意，使用两个反斜线开头的字符串默认被当做 `'AF_PIPE'` 地址而不是 `'AF_UNIX'` 地址。

认证密码

当使用*Connection.recv* 接收数据时，数据会自动被反序列化。不幸的是，对于一个不可信的数据源发来的数据，反序列化是存在安全风险的。所以*Listener* 和*Client()* 之间使用*hmac* 模块进行摘要认证。

认证密钥是一个 byte 类型的字符串，可以认为是和密码一样的东西，连接建立好后，双方都会要求另一方证明知道认证密钥。(这个证明过程不会通过连接发送密钥)

如果要求认证但是没有指定认证密钥，则会使用 `current_process().authkey` 的返回值 (参见*Process*)。这个值将被当前进程所创建的任何*Process* 对象自动继承。这意味着 (在默认情况下) 一个包含多进程的程序中的所有进程会在相互间建立连接的时候共享单个认证密钥。

os.urandom() 也可以用来生成合适的认证密钥。

日志记录

当前模块也提供了一些对 logging 的支持。注意，*logging* 模块本身并没有使用进程间共享的锁，所以来自于多个进程的日志可能（具体取决于使用的日志 handler 类型）相互覆盖或者混杂。

`multiprocessing.`**`get_logger`**`()`

返回*multiprocessing* 使用的 logger，必要的话会创建一个新的。

当首次创建时日志记录器级别为*logging.NOTSET* 并且没有默认处理器。发送到这个日志记录器的消息默认将不会传播到根日志记录器。

注意在 Windows 上，子进程只会继承父进程 logger 的日志级别 - 对于 logger 的其他自定义项不会继承。

`multiprocessing.`**`log_to_stderr`**`(`*level=None*`)`

此函数会调用*get_logger()* 但是会在返回的 logger 上增加一个 handler，将所有输出都使用 `'[%(levelname)s/%(processName)s] %(message)s'` 的格式发送到*sys.stderr* 。你可以通过传递一个 `level` 参数来修改记录器的 `levelname` 。

下面是一个在交互式解释器中打开日志功能的例子:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

要查看日志等级的完整列表，见*logging* 模块。

multiprocessing.dummy 模块

multiprocessing.dummy 复制了*multiprocessing* 的 API，不过是在*threading* 模块之上包装了一层。

特别地，*multiprocessing.dummy* 所提供的 `Pool` 函数会返回一个*ThreadPool* 的实例，该类是*Pool* 的子类，它支持所有相同的方法调用但会使用一个工作线程池而非工作进程池。

`class` `multiprocessing.pool.`**`ThreadPool`**`(`[*processes*[, *initializer*[, *initargs*]]]`)`

一个线程池对象，用来控制可向其提交任务的工作线程池。*ThreadPool* 实例与*Pool* 实例是完全接口兼容的，并且它们的资源也必须被正确地管理，或者是将线程池作为上下文管理器来使用，或者是通过手动调用*close()* 和*terminate()*。

processes 是要使用的工作线程数量。如果 *processes* 为 `None` 则使用*os.process_cpu_count()* 所返回的数值。

如果 *initializer* 不为 `None`，则每个工作进程将会在启动时调用 `initializer(*initargs)`。

不同于*Pool*，*maxtasksperchild* 和 *context* 不可被提供。

> **备注**
>
> *ThreadPool* 具有与*Pool* 相同的接口，它围绕一个进程池进行设计并且先于*concurrent.futures* 模块的引入。因此，它继承了一些对于基于线程的池来说没有意义的操作，并且它具有自己的用于表示异步任务状态的类型*AsyncResult*，该类型不为任何其他库所知。
>
> 用户通常应该倾向于使用*concurrent.futures.ThreadPoolExecutor*，它拥有从一开始就围绕线程进行设计的更简单接口，并且返回与许多其他库相兼容的*concurrent.futures.Future* 实例，包括*asyncio* 库。

18.2.3 编程指导

使用*multiprocessing* 时，应遵循一些指导原则和习惯用法。

所有 start 方法

下面这些适用于所有 start 方法。

避免共享状态

应该尽可能避免在进程间传递大量数据，越少越好。

最好坚持使用队列或者管道进行进程间通信，而不是底层的同步原语。

可序列化

保证所代理的方法的参数是可以序列化的。

代理的线程安全性

不要在多线程中同时使用一个代理对象，除非你用锁保护它。

（而在不同进程中使用 相同的代理对象却没有问题。）

使用 Join 避免僵尸进程

在 POSIX 上当一个进程结束但没有被合并则它将变成僵尸进程。这样的进程应该不会很多因为每次启动新进程（或*active_children()* 被调用）时所有尚未被合并的已完成进程都将被合并。而且调用一个已结束进程的*Process.is_alive* 也会合并这个进程。虽然如此但显式地合并你所启动的所有进程仍然是个好习惯。

继承优于序列化、反序列化

当使用 *spawn* 或者 *forkserver* 的启动方式时，*multiprocessing* 中的许多类型都必须是可序列化的，这样子进程才能使用它们。但是通常我们都应该避免使用管道和队列发送共享对象到另外一个进程，而是重新组织代码，对于其他进程创建出来的共享对象，让那些需要访问这些对象的子进程可以直接将这些对象从父进程继承过来。

避免杀死进程

通过*Process.terminate* 停止一个进程很容易导致这个进程正在使用的共享资源（如锁、信号量、管道和队列）损坏或者变得不可用，无法在其他进程中继续使用。

所以，最好只对那些从来不使用共享资源的进程调用*Process.terminate* 。

Join 使用队列的进程

记住，往队列放入数据的进程会一直等待直到队列中所有项被”feeder” 线程传给底层管道。（子进程可以调用队列的*Queue.cancel_join_thread* 方法禁止这种行为）

这意味着，任何使用队列的时候，你都要确保在进程 join 之前，所有存放到队列中的项将会被其他进程、线程完全消费。否则不能保证这个写过队列的进程可以正常终止。记住非精灵进程会自动 join 。

下面是一个会导致死锁的例子:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # 这将死锁
    obj = queue.get()
```

交换最后两行可以修复这个问题（或者直接删掉 `p.join()`）。

显式传递资源给子进程

在 POSIX 上使用 *fork* 启动方法，子进程将能够访问使用全局资源在父进程中创建的共享资源。但是，更好的做法是将对象作为子进程构造器的参数来传入。

除了（部分原因）让代码兼容 Windows 以及其他的进程启动方式外，这种形式还保证了在子进程生命期这个对象是不会被父进程垃圾回收的。如果父进程中的某些对象被垃圾回收会导致资源释放，这就变得很重要。

所以对于实例：

```
from multiprocessing import Process, Lock

def f():
    ... 使用锁进行一些操作 ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

应当重写成这样：

```
from multiprocessing import Process, Lock

def f(l):
    ... 使用 "l" 进行一些操作 ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

谨防将*sys.stdin* 数据替换为“类似文件的对象”

multiprocessing 原本会无条件地这样调用：

```
os.close(sys.stdin.fileno())
```

在 `multiprocessing.Process._bootstrap()` 方法中——这会导致与“进程中的进程”相关的一些问题。这已经被修改成了：

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

它解决于进程相互冲突导致文件描述符损坏错误的基础性问题，但是又对使用带输出缓冲的“文件型对象”替代*sys.stdin()* 的应用程序引入了潜在的危险。这种危险在于如果有多个进程在此文件型对象上调用*close()*，可能导致相同的数据被多次刷写到对象，造成数据损坏。

如果你写入文件型对象并实现了自己的缓存，可以在每次追加缓存数据时记录当前进程 id，从而将其变成 fork 安全的，当发现进程 id 变化后舍弃之前的缓存，例如:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

需要更多信息，请查看 bpo-5155, bpo-5313 以及 bpo-5331

spawn 和 *forkserver* 启动方式

还有一些没有被应用到 *fork* 启动方法的额外限制。

更依赖序列化

`Process.__init__()` 的所有参数都必须可序列化。同样的，当你继承*`Process`* 时，需要保证当调用*`Process.start`* 方法时，实例可以被序列化。

全局变量

记住，如果子进程中的代码尝试访问一个全局变量，它所看到的值（如果有）可能和父进程中执行*`Process.start`* 那一刻的值不一样。

当全局变量只是模块级别的常量时，是不会有问题的。

安全导入主模块

确保新的 Python 解释器可以安全地导入主模块，而不会导致意想不到的副作用（如启动新进程）。

例如，使用 *spawn* 或 *forkserver* 启动方式执行下面的模块，会引发*`RuntimeError`* 异常而失败。

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

应该通过下面的方法使用 `if __name__ == '__main__':` ，从而保护程序” 入口点”:

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

（如果程序将正常运行而不是冻结，则可以省略 `freeze_support()` 行）

这允许新启动的 Python 解释器安全导入模块然后运行模块中的 `foo()` 函数。

如果主模块中创建了进程池或者管理器，这个规则也适用。

18.2.4 例子

创建和使用自定义管理器、代理的示例:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# 一个简单的生成器函数
def baz():
    for i in range(10):
        yield i*i

# 针对生成器对象的代理类型
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# 返回 operator 模块的函数
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# 注册 Foo 类; 使 `f()` 和 `g()` 可通过代理访问
MyManager.register('Foo1', Foo)

# 注册 Foo 类; 使 `g()` 和 `_h()` 可通过代理访问
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# 注册生成器函数 baz; 使用 `GeneratorProxy` 来作为代理
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# 注册 get_operator_module(); 使公有函数可通过代理访问
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
```

(续下页)

(接上页)

```
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

使用*Pool*:

```
import multiprocessing
import time
import random
import sys

#
# 供测试代码使用的函数
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
```

(续下页)

(接上页)

```
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# 测试代码
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # 测试
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')
```

(续下页)

(接上页)

```
        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # 测试超时
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()
```

(续下页)

(接上页)

```
if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

一个演示如何使用队列来向一组工作进程提供任务并收集结果的例子：

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# 由工作进程运行的函数
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# 用于计算结果的函数
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# 被任务引用的函数
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # 创建队列
    task_queue = Queue()
    done_queue = Queue()

    # 提交任务
    for task in TASKS1:
        task_queue.put(task)

    # 启动工作进程
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()
```

(续下页)

(接上页)

```
    # 获取并打印结果
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # 使用 `put()` 添加更多任务
    for task in TASKS2:
        task_queue.put(task)

    # 获取并打印更多结果
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # 通知子进程停止运行
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

18.3 multiprocessing.shared_memory --- 可跨进程直接访问的共享内存

源代码: Lib/multiprocessing/shared_memory.py

Added in version 3.8.

该模块提供了一个*SharedMemory* 类，用于分配和管理多核或对称多处理器（SMP）机器上进程间的共享内存。为了协助进行不同进程间共享内存的生命周期管理，在*multiprocessing.managers* 模块中还提供了一个*BaseManager* 的子类*SharedMemoryManager*。

在本模块中，共享内存是指“POSIX 风格”的共享内存块（虽然它并不一定被显式地以这种风格实现）而不是指“分布式共享内存”。这种风格的共享内存允许不同进程读写一块共同的（或共享的）易失性内存区域。进程在传统上被限制为只能访问它们自己的进程内存空间而共享内存则允许跨进程共享数据，从而避免通过进程间发送消息的形式传递数据。相比通过磁盘或套接字或者其他需要序列化/反序列化以及数据拷贝的共享形式，直接通过内存共享数据可提供显著的性能提升。

class multiprocessing.shared_memory.**SharedMemory**(*name=None*, *create=False*, *size=0*, ***, *track=True*)

创建一个 SharedMemory 类的实例用来新建一个共享内存块或关联到一个已存在的共享内存块。每个共享内存块都被赋予一个独有的名称。通过这种方式，进程可以创建一个具有特定名称的共享内存块然后别的进程可以使用相同的名称关联到相同的共享内存块。

作为一种跨进程共享数据的方式，共享内存块的寿命可以超过创建它的原始进程。当一个进程不再需要访问一个可能仍被其他进程所需要的的共享内存块时，应当调用*close()* 方法。当一个共享内存块不再被任何进程所需要时，则应当调用*unlink()* 方法以确保执行适当的清理操作。

参数

- **name** (str | *None*) -- 被请求的共享内存的独有名称，以字符串形式指定。当创建新的共享内存块时，如果提供 None 作为名称（默认值），将随机生成一个新名称。
- **create** (bool) -- 控制是要创建新的共享内存块 (True) 还是关联到已有的共享内存块 (False)。

- **size** (int) -- 当创建新的共享内存块时所请求的字节数。由于某些平台会选择根据平台的内存页大小来分配内存块，因此共享内存块的实际大小可能会大于等于所请求的大小。当关联到已有的共享内存块时，*size* 形参将被忽略。
- **track** (bool) -- 当为 `True` 时，将在 OS 不会自动为共享内存块注册资源跟踪器进程的平台上执行注册操作。资源跟踪器会确保共享内存的正确清理，即使全部其他具有该内存访问权限的进程均未执行清理即退出。使用*multiprocessing* 的工具创建的具有共同上级的 Python 进程将共享一个资源跟踪器进程，并且共享内存段的生命周期将在这些进程中自动进行管理。以任何其他方式创建的 Python 进程在启用 *track* 的情况下访问共享内存时将获得它们自己的资源跟踪器。这将导致共享内存会被第一个终结的进程的资源跟踪器删除。为避免此问题，*subprocess* 或独立 Python 进程的使用者在已经有另一个进程执行跟踪记录时应当将 *track* 设为 `False`。在 Windows 上 *track* 将被忽略，因为该系统有自己的跟踪机制并会在所有指向特定共享内存的句柄被关闭时自动删除它。

在 3.13 版本发生变更: 增加了 *track* 形参。

close()

关闭该实例指向共享内存的文件描述符/句柄。一旦不再需要从该实例指向共享内存块的访问权限*close()* 就应当被调用。根据具体的操作系统，即使所有指向下层内存的句柄都已被关闭，内存都有可能被释放也有可能不被释放。要确保正确的清理，请使用*unlink()* 方法。

unlink()

删除下层的共享内存块。此方法在每个共享内存块上应当只被调用一次，无论指向它的句柄数量有多少。*unlink()* 和*close()* 可以按任意顺序调用，但在 can be called in any order, but trying to access data inside a shared memory block after *unlink()* 之后再试图访问共享内存块中的数据将导致内存访问错误，其种类取决于具体的系统平台。

此方法在 Windows 上无效，在该系统上删除共享内存块的唯一方式是关闭所有的句柄。

buf

共享内存块内容的 memoryview 。

name

共享内存块的唯一标识，只读属性。

size

共享内存块的字节大小，只读属性。

以下示例展示了*SharedMemory* 底层的用法:

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # 一次修改多个字节
>>> buffer[4] = 100                           # 一次修改单个字节
>>> # 关联到现有的共享内存块
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # 将数据拷贝到一个新的 array.array
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # 通过 shm_b 使用字节串来修改
>>> bytes(shm_a.buf[:5])      # 通过 shm_a 来访问
b'howdy'
>>> shm_b.close()   # 关闭每个 SharedMemory 实例
>>> shm_a.close()
>>> shm_a.unlink()  # 仅调用一次 unlink 来释放共享内存
```

下面的例子展示了*SharedMemory* 类配合 NumPy 数组 的实际应用，从两个独立的 Python shell 访问相同的 `numpy.ndarray`:

```
>>> # 在第一个 Python 交互式 shell 中
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8])  # 从一个现有的 NumPy 数组开始
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # 我们没有指定名称因此会自动为我们选择一个
'psm_21467_46075'

>>> # 在同一个 shell 中或同一台机器上新的 Python shell 中
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # 关联到现有的共享内存块
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # 请注意在这个例子中 a.shape 为 (6,) 而 a.dtype 为 np.int64
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # 回到第一个 Python 交互式 shell, b 将反映出此变化
>>> b
array([  1,   1,   2,   3,   5, 888])

>>> # 在第二个 Python shell 中执行清理
>>> del c  # 不是必须的; 只是强调该数组已不再使用
>>> existing_shm.close()

>>> # 在第一个 Python shell 中执行清理
>>> del b  # 不是必须的; 只是强制该数组已不再使用
>>> shm.close()
>>> shm.unlink()  # 最后清理并释放共享内存块
```

class multiprocessing.managers.**SharedMemoryManager**([*address*[, *authkey*]])

multiprocessing.managers.BaseManager 的子类，可被用于跨进程的共享内存块管理。

在 SharedMemoryManager 实例上调用*start()* 方法会导致启动一个新进程。这个新进程的唯一目的就是管理所有通过它创建的共享内存块的生命周期。想要释放该进程所管理的全部共享内存块，可以在实例上调用*shutdown()*。这会触发执行该进程所管理的所有*SharedMemory* 对象上的*unlink()* 调用，然后停止该进程本身。通过 SharedMemoryManager 创建 SharedMemory 实例，我们可以避免手动跟踪并触发共享内存资源的释放。

这个类提供了创建和返回*SharedMemory* 实例的方法，以及以共享内存为基础创建一个列表类对象 (*ShareableList*) 的方法。

请参阅*BaseManager* 查看有关被继承的可选输入参数 *address* 和 *authkey* 以及如何使用它们来从其他进程连接已有的 optional input arguments and how they may be used to connect to an existing SharedMemoryManager 服务的说明。

SharedMemory(*size*)

新建并返回一个具有指定的 *size* 个字节的*SharedMemory* 对象。

ShareableList(*sequence*)

新建并返回一个*ShareableList* 对象，使用从 *sequence* 输入的值来初始化。

下面的例子展示了*SharedMemoryManager* 的基本机制：

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # 启动管理共享内存块的进程
>>> sl = smm.ShareableList(range(4))
>>> sl
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # 在 sl, raw_shm 和 another_sl 上调用 unlink()
```

下面的例子展示了使用*SharedMemoryManager* 对象的一种更方便的方式，通过 `with` 语句来确保所有共享内存块在它们不再被需要时得到释放：

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # 将工作分给两个进程，将部分结果存储在 sl 中
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # 用 multiprocessing.Pool 可能会更高效
...     p1.join()
...     p2.join()   # 等等两个进程中的所有工作完成
...     total_result = sum(sl)  # 现在合并 sl 中的部分结果
```

当在 `with` 语句中使用*SharedMemoryManager* 对象时，使用这个管理器创建的共享内存块会在 `with` 语句代码块结束执行时全部被释放。

class multiprocessing.shared_memory.**ShareableList**(*sequence=None*, *, *name=None*)

提供一个可变的列表型对象，其中存储的所有值都是存储在一个共享内存块中。这会将可存储的值限制为下列内置数据类型：

- *int* (有符号 64 位)
- *float*
- *bool*
- *str* (当使用 UTF-8 编码时每个小于 10M 字节)
- *bytes* (每个小于 10M 字节)
- None

它与内置*list* 类型的显著区别还在于这些列表无法改变其总长度（即没有 `append()`, `insert()` 等）并且不支持通过切片动态地创建新的 `ShareableList`。

sequence 会被用来填充已有值的新 `ShareableList`。设为 `None` 则会基于唯一的共享内存名称联系到现有的 `ShareableList`。

name 是所请求的共享内存的唯一名称，与*SharedMemory* 的定义中描述的一致。当关联到现有的 `ShareableList` 时，将指明其共享内存块的唯一名称并将 *sequence* 设为 `None`。

备注

bytes 和*str* 值存在一个已知问题。如果它们以 `\x00` 空字节或字符结尾，那么当按索引号从 `ShareableList` 提取这些值时它们可能会被 静默地去除。这种 `.rstrip(b'\x00')` 行为并认为是一个程序错误并可能在未来被修复。参见 gh-106939。

对于某些应用来说在右侧截去尾部空值会造成问题，要绕过此问题可以在存储这样的值时总是无条件地在其末尾附加一个额外的非 0 字节并在获取时无条件地移除它:

```
>>> from multiprocessing import shared_memory
>>> nul_bug_demo = shared_memory.ShareableList(['?\x00', b'\x03\x02\x01\x00\x00\x00'])
>>> nul_bug_demo[0]
'?'
>>> nul_bug_demo[1]
b'\x03\x02\x01'
>>> nul_bug_demo.shm.unlink()
>>> padded = shared_memory.ShareableList(['?\x00\x07', b'\x03\x02\x01\x00\x00\x00\x07'])
>>> padded[0][:-1]
'?\x00'
>>> padded[1][:-1]
b'\x03\x02\x01\x00\x00\x00'
>>> padded.shm.unlink()
```

count(*value*)

返回 *value* 出现的次数。

index(*value*)

返回 *value* 首次出现的索引位置。如果 *value* 不存在则会引发*ValueError*。

format

包含由所有当前存储值所使用的*struct* 打包格式的只读属性。

shm

存储了值的*SharedMemory* 实例。

下面的例子演示了*ShareableList* 实例的基本用法:

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True, 42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>, <class
↪'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is unsupported
```

下面的例子演示了一个、两个或多个进程如何通过提供下层的共享内存块名称来访问同一个*ShareableList*:

```
>>> b = shared_memory.ShareableList(range(5))          # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name)   # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

下面的例子显示*ShareableList* (以及下层的*SharedMemory*) 对象可以在必要时被封存和解封。请注意，它将仍然为同一个共享对象。出现这种情况是因为被反序列化的对象具有相同的唯一名称并会使用这个相同的名称附加到现有的对象上（如果对象仍然存活）:

```
>>> import pickle
>>> from multiprocessing import shared_memory
>>> sl = shared_memory.ShareableList(range(10))
>>> list(sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> deserialized_sl = pickle.loads(pickle.dumps(sl))
>>> list(deserialized_sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl[0] = -1
>>> deserialized_sl[1] = -2
>>> list(sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(deserialized_sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl.shm.close()
>>> sl.shm.unlink()
```

18.4 concurrent 包

目前，此包中只有一个模块:

- *concurrent.futures* ——启动并行任务

18.5 concurrent.futures --- 启动并行任务

Added in version 3.2.

源码: Lib/concurrent/futures/thread.py 和 Lib/concurrent/futures/process.py

---

concurrent.futures 模块提供异步执行可调用对象高层接口。

异步执行可以由*ThreadPoolExecutor* 使用线程或由*ProcessPoolExecutor* 使用单独的进程来实现。两者都是实现抽象类*Executor* 定义的接口。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

18.5.1 Executor 对象

class concurrent.futures.**Executor**

抽象类提供异步执行调用方法。要通过它的子类调用，而不是直接调用。

submit(*fn*, */*, **args*, ***kwargs*)

调度可调用对象 *fn*，以 `fn(*args, **kwargs)` 方式执行并返回一个代表该可调用对象的执行的*Future* 对象。

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*fn*, **iterables*, *timeout=None*, *chunksize=1*)

类似于*map(fn, *iterables)* 但有以下差异：

- *iterables* 是立即执行而不是延迟执行的；
- *fn* 是异步执行的并且可以并发对 *fn* 的多个调用。

如果*__next__()* 被调用且从对*Executor.map()* 原始调用 *timeout* 秒之后其结果还不可用则已返回的迭代器将引发*TimeoutError*。*timeout* 可以是整数或浮点数。如果 *timeout* 未指定或为 `None`，则不限制等待时间。

如果 *fn* 调用引发了异常，那么当从迭代器获取其值时该异常将被引发。

使用*ProcessPoolExecutor* 时，这个方法会将 *iterables* 分割任务块并作为独立的任务并提交到执行池中。这些块的大概数量可以由 *chunksize* 指定正整数设置。对很长的迭代器来说，使用大的 *chunksize* 值比默认值 1 能显著地提高性能。*chunksize* 对*ThreadPoolExecutor* 没有效果。

在 3.5 版本发生变更: 加入 *chunksize* 参数。

shutdown(*wait=True*, ***, *cancel_futures=False*)

当待执行的 future 对象完成执行后向执行者发送信号，它就会释放正在使用的任何资源。在关闭后调用*Executor.submit()* 和*Executor.map()* 将会引发*RuntimeError*。

如果 *wait* 为 `True` 则此方法只有在所有待执行的 future 对象完成执行且释放已分配的资源后才会返回。如果 *wait* 为 `False`，方法立即返回，所有待执行的 future 对象完成执行后会释放已分配的资源。不管 *wait* 的值是什么，整个 Python 程序将等到所有待执行的 future 对象完成执行后才退出。

如果 *cancel_futures* 为 `True`，此方法将取消所有执行器还未开始运行的挂起的 Future。无论 *cancel_futures* 的值是什么，任何已完成或正在运行的 Future 都不会被取消。

如果 *cancel_futures* 和 *wait* 均为 `True`，则执行器已开始运行的所有 Future 将在此方法返回之前完成。其余的 Future 会被取消。

如果使用 `with` 语句，你就可以避免显式调用这个方法，它将会停止*Executor* (就好像*Executor.shutdown()* 调用时 *wait* 设为 `True` 一样等待):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

在 3.9 版本发生变更: 增加了 *cancel_futures*。

18.5.2 ThreadPoolExecutor

ThreadPoolExecutor 是*Executor* 的子类，它使用线程池来异步执行调用。

当可调用对象已关联了一个*Future* 然后在等待另一个*Future* 的结果时就会导致死锁情况。例如:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b 永远不会结束因为它在等待 a。
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a 永远不会结束因为它在等待 b。
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

与:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # 这将永远不会完成因为只有一个工作线程
    # 并且它正在执行此函数。
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

Executor 子类使用最多 *max_workers* 个线程的线程池来异步执行调用。

所有排入 ThreadPoolExecutor 的队列的线程将在解释器退出之前被合并。请注意执行此操作的 exit 处理器会在任何使用 atexit 添加的 exit 处理器 之前被执行。这意味着主线程中的异常必须被捕获和处理以便向线程发出信号使其能够优雅地退出。由于这个原理，建议不要将 ThreadPoolExecutor 用于长期运行的任务。

initializer 是在每个工作者线程开始处调用的一个可选可调用对象。*initargs* 是传递给初始化器的元组参数。任何向池提交更多工作的尝试，*initializer* 都将引发一个异常，当前所有等待的工作都会引发一个*BrokenThreadPool*。

在 3.5 版本发生变更: 如果 *max_workers* 为 None 或没有指定，将默认为机器处理器的个数，假如*ThreadPoolExecutor* 侧重于 I/O 操作而不是 CPU 运算，那么可以乘以 5 ，同时工作线程的数量可以比*ProcessPoolExecutor* 的数量高。

在 3.6 版本发生变更: 增加了 *thread_name_prefix* 形参来允许用户控制由线程池创建的*threading.Thread* 工作线程名称以方便调试。

在 3.7 版本发生变更: 加入 *initializer* 和 *initargs* 参数。

在 3.8 版本发生变更: *max_workers* 的默认值已改为 min(32, os.cpu_count() + 4)。这个默认值会保留至少 5 个工作线程用于 I/O 密集型任务。对于那些释放了 GIL 的 CPU 密集型任务，它最多会使用 32 个 CPU 核心。这样能够避免在多核机器上不知不觉地使用大量资源。

现在 ThreadPoolExecutor 在启动 *max_workers* 个工作线程之前也会重用空闲的工作线程。

在 3.13 版本发生变更: *max_workers* 的默认值已改为 min(32, (os.process_cpu_count() or 1) + 4)。

ThreadPoolExecutor 例子

```
import concurrent.futures
import urllib.request
```

(续下页)

(接上页)

```
URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistent-subdomain.python.org/']

# 获取一个页面并报告其 URL 和内容
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# 我们可以使用一个 with 语句来确保线程被迅速清理
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # 开始加载操作并以每个 Future 对象的 URL 对其进行标记
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

18.5.3 ProcessPoolExecutor

ProcessPoolExecutor 类是*Executor* 的子类，它使用进程池来异步地执行调用。*ProcessPoolExecutor* 会使用*multiprocessing* 模块，这允许它绕过*全局解释器锁* 但也意味着只可以处理和返回可封存的对象。

__main__ 模块必须可以被工作者子进程导入。这意味着*ProcessPoolExecutor* 不可以工作在交互式解释器中。

从可调用对象中调用*Executor* 或*Future* 的方法提交给*ProcessPoolExecutor* 会导致死锁。

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*, *max_tasks_per_child=None*)

异步地执行调用的*Executor* 子类使用最多 *max_workers* 个进程的进程池。如果 *max_workers* 为 None 或未给出，它将默认为*os.process_cpu_count()*。如果 *max_workers* 小于等于 0，则将引发*ValueError*。在 Windows 上，*max_workers* 必须小于等于 61。如果不是这样则将引发*ValueError*。如果 *max_workers* 为 None，则选择的默认值最多为 61，即使存在更多的处理器。*mp_context* 可以是一个*multiprocessing* 上下文或是 None。它将被用来启动工作进程。如果 *mp_context* 为 None 或未给出，则将使用默认的*multiprocessing* 上下文。参见*上下文和启动方法*。

initializer 是一个可选的可调用对象，它会在每个工作进程启动时被调用；*initargs* 是传给 initializer 的参数元组。如果 *initializer* 引发了异常，则所有当前在等待的任务以及任何向进程池提交更多任务的尝试都将引发*BrokenProcessPool*。

max_tasks_per_child 是指定单个进程在其退出并替换为新工作进程之前可以执行的最大任务数量的可选参数。在默认情况下 *max_tasks_per_child* 为 None 表示工作进程将存活与进程池一样长的时间。当指定了最大数量时，则如果不存在 *mp_context* 形参则将默认使用”spawn” 多进程启动方法。此特性不能兼容”fork” 启动方法。

在 3.3 版本发生变更: 当某个工作进程突然终止时，现在将引发*BrokenProcessPool*。在之前版本中，它的行为是未定义的但在执行器上的操作或它的 future 对象往往会被冻结或死锁。

在 3.7 版本发生变更: 添加 *mp_context* 参数允许用户控制由进程池创建给工作者进程的开始方法。

加入 *initializer* 和 *initargs* 参数。

> **备注**
>
> 在 Python 3.14 中默认的*multiprocessing* 启动方法 (参见*上下文和启动方法*) 将改为不再使用 *fork*。需要为其*ProcessPoolExecutor* 使用 *fork* 的代码应当通过传入 `mp_context=multiprocessing.get_context("fork")` 形参来显式地指明这一点。

在 3.11 版本发生变更: 增加了 *max_tasks_per_child* 参数以允许用户控制进程池中工作进程的生命期。

在 3.12 版本发生变更: 在 POSIX 系统上，如果你的应用程序有多个线程而*multiprocessing* 上下文使用了 `"fork"` 启动方法: 内部调用的*os.fork()* 函数来生成工作进程可能会引发*DeprecationWarning*。请传递配置为使用不同启动方法的 *mp_context*。进一步的解释请参阅*os.fork()* 文档。

在 3.13 版本发生变更: 在默认情况下 *max_workers* 将使用*os.process_cpu_count()*，而不是*os.cpu_count()*。

ProcessPoolExecutor 例子

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

18.5.4 Future 对象

Future 类将可调用对象封装为异步执行。*Future* 实例由*Executor.submit()* 创建。

class concurrent.futures.**Future**

将可调用对象封装为异步执行。*Future* 实例由*Executor.submit()* 创建，除非测试，不应直接创建。

cancel()

尝试取消调用。如果调用正在执行或已结束运行不能被取消则该方法将返回 `False`，否则调用会被取消并且该方法将返回 `True`。

cancelled()

如果调用成功取消返回 `True`。

running()

如果调用正在执行而且不能被取消那么返回 `True` 。

done()

如果调用已被取消或正常结束那么返回 `True`。

result(*timeout=None*)

返回调用所返回的值。如果调用尚未完成则此方法将等待至多 *timeout* 秒。如果调用在 *timeout* 秒内仍未完成，则将引发*TimeoutError*。*timeout* 可以为整数或浮点数。如果 *timeout* 未指定或为 `None`，则不限制等待时间。

如果 future 在完成前被取消则*CancelledError* 将被触发。

如果调用引发了一个异常，这个方法也会引发同样的异常。

exception(*timeout=None*)

返回调用所引发的异常。如果调用尚未完成则此方法将等待至多 *timeout* 秒。如果调用在 *timeout* 秒内仍未完成，则将引发*TimeoutError*。*timeout* 可以为整数或浮点数。如果 *timeout* 未指定或为 `None`，则不限制等待时间。

如果 future 在完成前被取消则*CancelledError* 将被触发。

如果调用正常完成那么返回 `None`。

add_done_callback(*fn*)

附加可调用 *fn* 到 future 对象。当 future 对象被取消或完成运行时，将会调用 *fn*，而这个 future 对象将作为它唯一的参数。

加入的可调用对象总被属于添加它们的进程中的线程按加入的顺序调用。如果可调用对象引发一个*Exception* 子类，它会被记录下来并被忽略掉。如果可调用对象引发一个*BaseException* 子类，这个行为没有定义。

如果 future 对象已经完成或已取消，*fn* 会被立即调用。

下面这些*Future* 方法用于单元测试和*Executor* 实现。

set_running_or_notify_cancel()

这个方法只可以在执行关联*Future* 工作之前由*Executor* 实现调用或由单测试调用。

如果此方法返回 `False` 则*Future* 已被取消，即*Future.cancel()* 已被调用并返回 `True`。任何等待*Future* 完成 (即通过*as_completed()* 或*wait()*) 的线程将被唤醒。

如果此方法返回 `True` 则*Future* 没有被取消并已被置为正在运行的状态，即对*Future.running()* 的调用将返回 `True`。

这个方法只可以被调用一次并且不能在调用*Future.set_result()* 或*Future.set_exception()* 之后再调用。

set_result(*result*)

设置将*Future* 关联工作的结果给 *result* 。

这个方法只可以由*Executor* 实现和单元测试使用。

在 3.8 版本发生变更: 如果*Future* 已经完成则此方法会引发*concurrent.futures.InvalidStateError*。

set_exception(*exception*)

设置*Future* 关联工作的结果给*Exception* *exception* 。

这个方法只可以由*Executor* 实现和单元测试使用。

在 3.8 版本发生变更: 如果*Future* 已经完成则此方法会引发*concurrent.futures.InvalidStateError*。

18.5.5 模块函数

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

等待由 *fs* 指定的*Future* 实例（可能由不同的*Executor* 实例创建）完成。重复传给 *fs* 的 future 会被移除并将只返回一次。返回一个由集合组成的具名 2 元组。第一个集合的名称为 `done`，包含在等待完成之前已完成的 future（包括正常结束或被取消的 future）。第二个集合的名称为 `not_done`，包含未完成的 future（包括挂起的或正在运行的 future）。

timeout 可以用来控制返回前最大的等待秒数。*timeout* 可以为 int 或 float 类型。如果 *timeout* 未指定或为 `None` ，则不限制等待时间。

return_when 指定此函数应在何时返回。它必须为以下常数之一:

常量	描述
concurrent.futures.**FIRST_COMPLETED**	函数将在任意可等待对象结束或取消时返回。
concurrent.futures.**FIRST_EXCEPTION**	该函数将在任何 future 对象通过引发异常而结束时返回。如果没有任何 future 对象引发引发那么它将等价于*ALL_COMPLETED*。
concurrent.futures.**ALL_COMPLETED**	函数将在所有可等待对象结束或取消时返回。

concurrent.futures.**as_completed**(*fs*, *timeout=None*)

返回一个迭代器，每当 *fs* 所给出的*Future* 实例（可能由不同的*Executor* 实例创建）完成时这个迭代器会产生新的 future（包括正常结束或被取消的 future 对象）。任何由 *fs* 给出的重复的 future 对象将只被返回一次。任何在*as_completed()* 被调用之前完成的 future 对象将优先被产生。如果*__next__()* 被调用并且在最初调用*as_completed()* 之后的 *timeout* 秒内其结果仍不可用，这个迭代器将引发*TimeoutError*。*timeout* 可以为整数或浮点数。如果 *timeout* 未指定或为 `None`，则不限制等待时间。

参见

PEP 3148 -- future 对象 - 异步执行指令。
该提案描述了 Python 标准库中包含的这个特性。

18.5.6 Exception 类

exception concurrent.futures.**CancelledError**

future 对象被取消时会触发。

exception concurrent.futures.**TimeoutError**

TimeoutError 的一个已被弃用的别名，会在 future 操作超出了给定的时限时被引发。

在 3.11 版本发生变更: 这个类是*TimeoutError* 的别名。

exception concurrent.futures.**BrokenExecutor**

当执行器被某些原因中断而且不能用来提交或执行新任务时就会被引发派生于*RuntimeError* 的异常类。

Added in version 3.7.

exception concurrent.futures.**InvalidStateError**

当某个操作在一个当前状态所不允许的 future 上执行时将被引发。

Added in version 3.8.

exception concurrent.futures.thread.**BrokenThreadPool**

派生自*BrokenExecutor*，这个异常类会在*ThreadPoolExecutor* 的某个工作线程初始化失败时被引发。

Added in version 3.7.

exception concurrent.futures.process.**BrokenProcessPool**

派生自*BrokenExecutor* (原为*RuntimeError*)，这个异常类会在*ProcessPoolExecutor* 的某个工作进程以不完整的方式终结（例如，从外部杀掉）时被引发。

Added in version 3.3.

18.6 subprocess --- 子进程管理

源代码: Lib/subprocess.py

---

subprocess 模块允许你生成新的进程，连接它们的输入、输出、错误管道，并且获取它们的返回码。此模块打算代替一些老旧的模块与功能：

```
os.system
os.spawn*
```

在下面的段落中，你可以找到关于*subprocess* 模块如何代替这些模块和功能的相关信息。

参见

PEP 324 -- 提出 subprocess 模块的 PEP

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或*WebAssembly 平台* 上不受支持。

18.6.1 使用 subprocess 模块

推荐的调用子进程的方式是在任何它支持的用例中使用*run()* 函数。对于更进阶的用例，也可以使用底层的*Popen* 接口。

subprocess.**run**(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, **other_popen_kwargs*)

运行被 *arg* 描述的指令. 等待指令完成, 然后返回一个*CompletedProcess* 实例.

以上显示的参数仅仅是最简单的一些，下面*常用参数* 描述（因此在缩写签名中使用仅关键字标示）。完整的函数头和*Popen* 的构造函数一样，此函数接受的大多数参数都被传递给该接口。（*timeout*, *input*, *check* 和 *capture_output* 除外）。

如果 *capture_output* 为真值，则 stdout 和 stderr 将被捕获。当被使用时，内部*Popen* 对象将自动创建并把 *stdout* 和 *stderr* 均设为*PIPE*。*stdout* 和 *stderr* 参数不可与 *capture_output* 同时提供。如果你希望捕获并将两个流合并在一起，请将 *stdout* 设为*PIPE* 并将 *stderr* 设为*STDOUT*，而不是使用 *capture_output*。

可以指定以秒为单位的 *timeout*，它会在内部传递给*Popen.communicate()*。如果达到超时限制，子进程将被杀掉并等待。*TimeoutExpired* 异常将在子进程终结后重新被引发。在许多平台 API 上初始进程创建本身不可以被打断因此不保证你能看到超时异常直到至少进程创建花费的时间结束后。

input 参数将被传递给*Popen.communicate()* 以及子进程的 stdin。如果使用此参数则它必须是一个字节序列，或者如果指定了 *encoding* 或 *errors* 或 *text* 为真值则可以是一个字符串。当使用此参数时，将自动创建内部的*Popen* 对象并将其 *stdin* 设为*PIPE*，并且不可同时使用 *stdin* 参数。

如果 *check* 设为 True, 并且进程以非零状态码退出, 一个*CalledProcessError* 异常将被抛出. 这个异常的属性将设置为参数, 退出码, 以及标准输出和标准错误, 如果被捕获到.

如果指定了 *encoding* 或 *error*，或者 *text* 被设为真值，标准输入、标准输出和标准错误的文件对象将使用指定的 *encoding* 和 *errors* 或者*io.TextIOWrapper* 默认值以文本模式打开。*universal_newlines* 参数等同于 *text* 并且提供了向后兼容性。默认情况下, 文件对象是以二进制模式打开的。

如果 *env* 不为 None，则它必须是一个为新进程定义环境变量的映射；它们将顶替继承当前进程环境的默认行为被使用。它会被直接传递给*Popen*。这个映射在任何平台上均可以是字符串到字符串的映射或者在 POSIX 平台上也可以是字节串到字节串的映射，就像是*os.environ* 或者*os.environb*。

示例:

```
>>> subprocess.run(["ls", "-l"])  # 不捕获输出
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

Added in version 3.5.

在 3.6 版本发生变更: 添加了 *encoding* 和 *errors* 形参.

在 3.7 版本发生变更: 添加了 *text* 形参，作为 *universal_newlines* 的一个更好理解的别名。添加了 *capture_output* 形参。

在 3.12 版本发生变更: 针对 shell=True 改变的 Windows shell 搜索顺序。当前目录和 %PATH% 会被替换为 %COMSPEC% 和 %SystemRoot%\System32\cmd.exe。因此，在当前目录中投放一个命名为 cmd.exe 的恶意程序不会再起作用。

class subprocess.**CompletedProcess**

run() 的返回值, 代表一个进程已经结束.

args

被用作启动进程的参数. 可能是一个列表或字符串.

returncode

子进程的退出状态码. 通常来说, 一个为 0 的退出码表示进程运行正常.

一个负值 -N 表示子进程被信号 N 中断 (仅 POSIX).

stdout

从子进程捕获到的标准输出. 一个字节序列, 或一个字符串, 如果*run()* 是设置了 *encoding*, *errors* 或者 text=True 来运行的. 如果未有捕获, 则为 None.

如果你通过 stderr=subprocess.STDOUT 运行进程，标准输入和标准错误将被组合在这个属性中，并且*stderr* 将为 None。

stderr

捕获到的子进程的标准错误. 一个字节序列, 或者一个字符串, 如果*run()* 是设置了参数 *encoding*, *errors* 或者 text=True 运行的. 如果未有捕获, 则为 None.

check_returncode()

如果*returncode* 非零, 抛出*CalledProcessError*.

Added in version 3.5.

subprocess.**DEVNULL**

可被*Popen* 的 *stdin*, *stdout* 或者 *stderr* 参数使用的特殊值, 表示使用特殊文件*os.devnull*.

Added in version 3.3.

subprocess.**PIPE**

可被*Popen* 的 *stdin*, *stdout* 或者 *stderr* 参数使用的特殊值, 表示打开标准流的管道. 常用于*Popen.communicate()*.

subprocess.**STDOUT**

可被*Popen* 的 *stderr* 参数使用的特殊值，表示标准错误与标准输出使用同一句柄。

exception subprocess.**SubprocessError**

此模块的其他异常的基类。

Added in version 3.3.

exception subprocess.**TimeoutExpired**

SubprocessError 的子类，等待子进程的过程中发生超时时被抛出。

cmd

用于创建子进程的指令。

timeout

超时秒数。

output

当被*run()* 或*check_output()* 捕获时的子进程的输出。在其它情况下将为 None。当有任何输出被捕获时这将始终为*bytes* 而不考虑是否设置了 text=True。当未检测到输出时它可能会保持为 None 而不是 b''。

stdout

对 output 的别名，对应的有*stderr*。

stderr

当被*run()* 捕获时的标准错误输出。在其它情况下将为 None。当有标准错误输出被捕获时这将始终为*bytes* 而不考虑是否设置了 text=True。当未检测到标准错误输出时它可能会保持为 None 而不是 b''。

Added in version 3.3.

在 3.5 版本发生变更: 添加了 *stdout* 和 *stderr* 属性。

exception subprocess.**CalledProcessError**

SubprocessError 的子类, 当一个由*check_call()*, *check_output()* 或*run()* (附带 check=True) 运行的进程返回了非零退出状态码时将被引发。

returncode

子进程的退出状态。如果程序由一个信号终止，这将会被设为一个负的信号码。

cmd

用于创建子进程的指令。

output

子进程的输出，如果被*run()* 或*check_output()* 捕获。否则为 None。

stdout

对 output 的别名，对应的有*stderr*。

stderr

子进程的标准错误输出，如果被*run()* 捕获。否则为 None。

在 3.5 版本发生变更: 添加了 *stdout* 和 *stderr* 属性。

常用参数

为了支持丰富的使用案例，*Popen* 的构造函数（以及方便的函数）接受大量可选的参数。对于大多数典型的用例，许多参数可以被安全地留以它们的默认值。通常需要的参数有:

args 被所有调用需要，应当为一个字符串，或者一个程序参数序列。提供一个参数序列通常更好，它可以更小心地使用参数中的转义字符以及引用（例如允许文件名中的空格）。如果传递一个简单的字符串，则 *shell* 参数必须为*True*（见下文）或者该字符串中将被运行的程序名必须用简单的命名而不指定任何参数。

stdin, *stdout* 和 *stderr* 分别指定被执行程序的标准输入、标准输出和标准错误文件句柄。合法的值包括 None, *PIPE*, *DEVNULL*, 现存的文件描述符（一个正整数），现存的具有合法文件描述符的*file object*。当使用默认设置 None 时，将不会进行任何重定向。*PIPE* 表示应当新建一个连接子进程的管道。*DEVNULL* 表示将使用特殊文件*os.devnull*。此外，*stderr* 还可以为*STDOUT*，这表示来自子进程的 stderr 数据应当被捕获到与 *stdout* 相同的文件句柄中。

如果指定了 *encoding* 或 *errors*，或者 *text* (也称 *universal_newlines*) 为真，则文件对象 *stdin*、*stdout* 与 *stderr* 将会使用在此次调用中指定的 *encoding* 和 *errors* 或者*io.TextIOWrapper* 的默认值以文本模式打开。

当构造函数的 *newline* 参数为 None 时。对于 *stdin*，输入的换行符 '\n' 将被转换为默认的换行符*os.linesep*。对于 *stdout* 和 *stderr*，所有输出的换行符都被转换为 '\n'。更多信息，查看*io.TextIOWrapper* 类的文档。

如果文本模式未被使用，*stdin*，*stdout* 和 *stderr* 将会以二进制流模式打开。没有编码与换行符转换发生。

在 3.6 版本发生变更: 增加了 *encoding* 和 *errors* 形参。

在 3.7 版本发生变更: 添加了 *text* 形参作为 *universal_newlines* 的别名。

> **备注**
>
> 文件对象*Popen.stdin* 、*Popen.stdout* 和*Popen.stderr* 的换行符属性不会被*Popen.communicate()* 方法更新。

如果 *shell* 设为 True,，则使用 shell 执行指定的指令。如果您主要使用 Python 增强的控制流（它比大多数系统 shell 提供的强大），并且仍然希望方便地使用其他 shell 功能，如 shell 管道、文件通配符、环境变量展开以及 ~ 展开到用户家目录，这将非常有用。但是，注意 Python 自己也实现了许多类似 shell 的特性（例如*glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()* 和*shutil*）。

在 3.3 版本发生变更: 当 *universal_newlines* 被设为 True，则类将使用*locale.getpreferredencoding(False)* 编码格式来代替 locale.getpreferredencoding()。关于它们的区别的更多信息，见*io.TextIOWrapper*。

> **备注**
>
> 在使用 shell=True 之前，请阅读*Security Considerations* 段落。

这些选项以及所有其他选项在*Popen* 构造函数文档中有更详细的描述。

Popen 构造函数

此模块的底层的进程创建与管理由*Popen* 类处理。它提供了很大的灵活性，因此开发者能够处理未被便利函数覆盖的不常见用例。

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, ***, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*, *pipesize=-1*, *process_group=None*)

在一个新的进程中执行子程序。在 POSIX 上，该类会使用类似于*os.execvpe()* 的行为来执行子程序。在 Windows 上，该类会使用 Windows `CreateProcess()` 函数。*Popen* 的参数如下。

args 应当是一个程序参数的序列或者是一个单独的字符串或*path-like object*。默认情况下，如果 *args* 是序列则要运行的程序为 *args* 中的第一项。如果 *args* 是字符串，则其解读依赖于具体平台，如下所述。请查看 *shell* 和 *executable* 参数了解其与默认行为的其他差异。除非另有说明，否则推荐以序列形式传入 *args*。

警告

为了最大化可靠性，请使用可执行文件的完整限定路径。要在 `PATH` 中搜索一个非限定名称，请使用*shutil.which()*。在所有平台上，传入*sys.executable* 是再次启动当前 Python 解释器的推荐方式，并请使用 `-m` 命令行格式来启动已安装的模块。

对 *executable* (或 *args* 的第一项) 路径的解析方式依赖于具体平台。对于 POSIX，请参阅*os.execvpe()*，并要注意当解析或搜索可执行文件路径时，*cwd* 会覆盖当前工作目录而 *env* 可以覆盖 `PATH` 环境变量。对于 Windows，请参阅 `lpApplicationName` 的文档以及 `lpCommandLine` 形参 (传给 WinAPI `CreateProcess`)，并要注意当解析或搜索可执行文件路径时如果传入 `shell=False`，则 *cwd* 不会覆盖当前工作目录而 *env* 无法覆盖 `PATH` 环境变量。使用完整路径可避免所有这些变化情况。

向外部函数传入序列形式参数的一个例子如下:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

在 POSIX，如果 *args* 是一个字符串，此字符串被作为将被执行的程序的命名或路径解释。但是，只有在不传递任何参数给程序的情况下才能这么做。

备注

将 shell 命令拆分为参数序列的方式可能并不很直观，特别是在复杂的情况下。*shlex.split()* 可以演示如何确定 *args* 适当的拆分形式:

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '
↪$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

特别注意，由 shell 中的空格分隔的选项（例如 *-input*）和参数（例如 *eggs.txt* ）位于分开的列表元素中，而在需要时使用引号或反斜杠转义的参数在 shell （例如包含空格的文件名或上面显示的 *echo* 命令）是单独的列表元素。

在 Windows，如果 *args* 是一个序列，他将通过一个在*在 Windows 上将参数列表转换为一个字符串*描述的方式被转换为一个字符串。这是因为底层的 `CreateProcess()` 只处理字符串。

在 3.6 版本发生变更: 在 POSIX 上如果 *shell* 为 `False` 并且序列包含路径类对象则 *args* 形参可以接受一个*path-like object*。

在 3.8 版本发生变更: 如果在 Windows 上 *shell* 为 `False` 并且序列包含字节串和路径类对象则 *args* 形参可以接受一个*path-like object*。

参数 *shell* （默认为 `False`）指定是否使用 shell 执行程序。如果 *shell* 为 `True`，更推荐将 *args* 作为字符串传递而非序列。

在 POSIX，当 `shell=True`，shell 默认为 `/bin/sh`。如果 *args* 是一个字符串，此字符串指定将通过 shell 执行的命令。这意味着字符串的格式必须和在命令提示符中所输入的完全相同。这包括，例如，引号和反斜杠转义包含空格的文件名。如果 *args* 是一个序列，第一项指定了命令，另外的项目将作为传递给 shell （而非命令）的参数对待。也就是说，*Popen* 等同于:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

在 Windows，使用 `shell=True`，环境变量 `COMSPEC` 指定了默认 shell。在 Windows 你唯一需要指定 `shell=True` 的情况是你想要执行内置在 shell 中的命令（例如 **`dir`** 或者 **`copy`**）。在运行一个批处理文件或者基于控制台的可执行文件时，不需要 `shell=True`。

备注

在使用 `shell=True` 之前，请阅读*Security Considerations* 段落。

bufsize 将在*open()* 函数创建了 stdin/stdout/stderr 管道文件对象时作为对应的参数供应:

- `0` 表示不使用缓冲区（读取与写入是一个系统调用并且可以返回短内容）
- `1` 表示带有行缓冲（仅在 `text=True` 或 `universal_newlines=True` 时有用）
- 任何其他正值表示使用一个约为对应大小的缓冲区
- 负的 *bufsize* （默认）表示使用系统默认的 io.DEFAULT_BUFFER_SIZE。

在 3.3.1 版本发生变更: *bufsize* 现在默认为 -1 表示启用缓冲以符合大多数代码所期望的行为。在 Python 3.2.4 和 3.3.1 之前的版本中它错误地将默认值设为 `0` 即无缓冲并且允许短读取。这是无意的失误并且与大多数代码所期望的 Python 2 的行为不一致。

executable 参数指定一个要执行的替换程序。这很少需要。当 `shell=True`，*executable* 替换 *args* 指定运行的程序。但是，原始的 *args* 仍然被传递给程序。大多数程序将被 *args* 指定的程序作为命令名对待，这可以与实际运行的程序不同。在 POSIX，*args* 名作为实际调用程序中可执行文件的显示名称，例如 **`ps`**。如果 `shell=True`，在 POSIX，*executable* 参数指定用于替换默认 shell `/bin/sh` 的 shell。

在 3.6 版本发生变更: 在 POSIX 上 *executable* 形参可以接受一个*path-like object*。

在 3.8 版本发生变更: 在 Windows 上 *executable* 形参可以接受一个字节串和*path-like object*。

在 3.12 版本发生变更: 针对 `shell=True` 改变的 Windows shell 搜索顺序。当前目录和 `%PATH%` 会被替换为 `%COMSPEC%` 和 `%SystemRoot%\System32\cmd.exe`。因此，在当前目录中投放一个命名为 `cmd.exe` 的恶意程序不会再起作用。

stdin, *stdout* 和 *stderr* 分别指定被执行程序的标准输入、标准输出和标准错误文件句柄。合法的值包括 `None`, *PIPE*, *DEVNULL*, 现在的文件描述符（一个正整数），现存的具有合法文件描述符的*file object*。当使用默认设置 `None` 时，将不会进行任何重定向。*PIPE* 表示应当新建一个连接子进程的管道。*DEVNULL* 表示将使用特殊文件*os.devnull*。此外，*stderr* 还可以为*STDOUT*，这表示来自子进程的 stderr 数据应当被捕获到与 *stdout* 相同的文件句柄中。

如果 *preexec_fn* 被设为一个可调用对象，此对象将在子进程刚创建时被调用。（仅 POSIX）

警告

preexec_fn 形参在应用程序中存在多线程时是不安全的。子进程在 exec 被调用之前可能会死锁。

备注

如果你需要为子进程修改环境请使用 *env* 形参而不要在 *preexec_fn* 中操作。*start_new_session* 和 *process_group* 形参应当代替使用 *preexec_fn* 的代码来在子进程中调用`os.setsid()` 或`os.setpgid()`。

在 3.8 版本发生变更: *preexec_fn* 形参在子解释器中已不再受支持。在子解释器中使用此形参将引发`RuntimeError`。这个新限制可能会影响部署在 mod_wsgi, uWSGI 和其他嵌入式环境中的应用。

如果 *close_fds* 为真值，则除 `0`, `1` 和 `2` 之外的所有文件描述符都将在子进程执行前被关闭。而当 *close_fds* 为假值时，文件描述符将遵循它们的可继承旗标，如文件描述符的继承 所描述的。

在 Windows，如果 *close_fds* 为真，则子进程不会继承任何句柄，除非在 `STARTUPINFO.lpAttributeList` 的 `handle_list` 的键中显式传递，或者通过标准句柄重定向传递。

在 3.2 版本发生变更: *close_fds* 的默认值已经从`False` 修改为上述值。

在 3.7 版本发生变更: 在 Windows，当重定向标准句柄时 *close_fds* 的默认值从`False` 变为`True`。现在重定向标准句柄时有可能设置 *close_fds* 为`True`。（标准句柄指三个 stdio 的句柄）

pass_fds 是一个可选的在父子进程间保持打开的文件描述符序列。提供任何 *pass_fds* 将强制 *close_fds* 为`True`。（仅 POSIX）

在 3.2 版本发生变更: 加入了 *pass_fds* 形参。

如果 *cwd* 不为 `None`，此函数在执行子进程前会将当前工作目录改为 *cwd*。*cwd* 可以是一个字符串、字节串或路径类对象。在 POSIX 上，如果可执行文件路径为相对路径则此函数会相对于 *cwd* 来查找 *executable* (或 *args* 的第一项)。

在 3.6 版本发生变更: 在 POSIX 上 *cwd* 形参接受一个*path-like object*。

在 3.7 版本发生变更: 在 Windows 上 *cwd* 形参接受一个*path-like object*。

在 3.8 版本发生变更: 在 Windows 上 *cwd* 形参接受一个字节串对象。

如果 *restore_signals* 为 true（默认值），则 Python 设置为 SIG_IGN 的所有信号将在 exec 之前的子进程中恢复为 SIG_DFL。目前，这包括 SIGPIPE，SIGXFZ 和 SIGXFSZ 信号。（仅 POSIX）

在 3.2 版本发生变更: *restore_signals* 被加入。

如果 *start_new_session* 为真值则 `setsid()` 系统调用将在执行子进程之前在子进程中执行。

Availability: POSIX

在 3.2 版本发生变更: *start_new_session* 被添加。

如果 *process_group* 为非负整数，则 `setpgid(0, value)` 系统调用将在执行子进程之前在子进程中执行。

Availability: POSIX

在 3.11 版本发生变更: 添加了 *process_group*。

如果 *group* 不为 `None`，则 setregid() 系统调用将在子进程执行之前在下级进程中进行。如果所提供的值是一个字符串，将通过`grp.getgrnam()` 来查找它并将使用 `gr_gid` 中的值。如果该值是一个整数，它将被原样传递。（POSIX 专属）

Availability: POSIX

Added in version 3.9.

如果 *extra_groups* 不为 `None`，则 setgroups() 系统调用将在子进程执行之前在下级进程中进行。在 *extra_groups* 中提供的字符串将通过`grp.getgrnam()` 来查找并将使用 `gr_gid` 中的值。整数值将被原样传递。（POSIX 专属）

Availability: POSIX

Added in version 3.9.

如果 *user* 不为 None，则 setreuid() 系统调用将在子进程执行之前在下级进程中进行。如果所提供的值是一个字符串，将通过*pwd.getpwnam()* 来查找它并将使用 pw_uid 中的值。如果该值是一个整数，它将被原样传递。(POSIX 专属)

Availability: POSIX

Added in version 3.9.

如果 *umask* 不为负值，则 umask() 系统调用将在子进程执行之前在下级进程中进行。

Availability: POSIX

Added in version 3.9.

如果 *env* 不为 None，则它必须是一个为新进程定义环境变量的映射；它们将顶替继承当前环境的默认行为被使用。这个映射在任何平台上均可以是字符串到字符串的映射或者在 POSIX 平台上也可以是字节串到字节串的映射，就像是*os.environ* 或者*os.environb*。

备注

如果指定，*env* 必须提供所有被子进程需求的变量。在 Windows，为了运行一个 side-by-side assembly，指定的 *env* **必须**包含一个有效的 SystemRoot。

如果指定了 *encoding* 或 *errors*，或者如果 *text* 为真值，则文件对象 *stdin*, *stdout* 和 *stderr* 将使用指定的 *encoding* 和 *errors* 以文本模式打开，就如上文常用参数 中所描述的。*universal_newlines* 参数等同于 *text* 且是出于下向兼容性考虑而提供的。在默认情况下，文件对象将以二进制模式打开。

Added in version 3.6: *encoding* 和 *errors* 被添加。

Added in version 3.7: *text* 作为 *universal_newlines* 的一个更具可读性的别名被添加。

如果给出，*startupinfo* 将是一个*STARTUPINFO* 对象，它会被传递给下层的 CreateProcess 函数。

如果给出，*creationflags* 可以是下列旗标中的一个或多个:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*
- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

当*PIPE* 被用作 *stdin*, *stdout* 或 *stderr* 时 *pipesize* 可被用于改变管道的大小。管道的大小仅会在受支持的平台上被改变（当撰写本文档时只有 Linux 支持）。其他平台将忽略此形参。

在 3.10 版本发生变更: 增加了 *pipesize* 形参。

Popen 对象支持通过 with 语句作为上下文管理器，在退出时关闭文件描述符并等待进程:

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Popen 和此模块中用到它的其他函数会引发一个审计事件 subprocess.Popen，附带参数 executable, args, cwd 和 env。args 的值可以是单个字符串或字符串列表，取决于具体的平台。

在 3.2 版本发生变更: 添加了上下文管理器支持。

在 3.6 版本发生变更: 现在，如果 Popen 析构时子进程仍然在运行，则析构器会发送一个 *ResourceWarning* 警告。

在 3.8 版本发生变更: 在某些情况下 Popen 可以使用 *os.posix_spawn()* 以获得更好的性能。在适用于 Linux 的 Windows 子系统和 QEMU 用户模拟器上，使用 *os.posix_spawn()* 的 Popen 构造器不再会因找不到程序等错误而引发异常，而是上下级进程失败并返回一个非零的 *returncode*。

异常

在子进程中抛出的异常，在新的进程开始执行前，将会被再次在父进程中抛出。

被引发的最一般异常是 *OSError*。例如这会在尝试执行一个不存在的文件时发生。应用程序应当为 *OSError* 异常做好准备。请注意，如果 shell=True，则 *OSError* 仅会在未找到选定的 shell 本身时被引发。要确定 shell 是否未找到所请求的应用程序，必须检查来自子进程的返回码或输出。

如果 *Popen* 调用时有无效的参数，则一个 *ValueError* 将被抛出。

check_call() 与 *check_output()* 在调用的进程返回非零退出码时将抛出 *CalledProcessError*。

所有接受 *timeout* 形参的函数与方法，例如 *run()* 和 *Popen.communicate()* 将会在进程退出前超时到期时引发 *TimeoutExpired*。

此模块中定义的异常都继承自 *SubprocessError*。

Added in version 3.3: 基类 *SubprocessError* 被添加。

18.6.2 安全考量

不同于某些其他的 popen 函数，这个库将不会隐式地选择调用系统 shell。这意味着所有字符，包括 shell 元字符都可以被安全地传递给子进程。如果 shell 是通过 shell=True 被显式地唤起的，则应用程序要负责确保所有空白符和元字符被适当地转义以避免 shell 注入 安全漏洞。在某些平台 上，可以使用 *shlex.quote()* 来执行这种转义。

在 Windows 上，批处理文件 (*.bat 或 *.cmd) 可以在系统 shell 中通过操作系统调用来启动而忽略传给该库的参数。这可能导致根据 shell 规则来解析参数，而没有任何 Python 添加的转义。如果你想要附带来自不受信任源的参数启动批处理文件，请考虑传入 shell=True 以允许 Python 转义特殊字符。请参阅 gh-114539 了解相关讨论。

18.6.3 Popen 对象

Popen 类的实例拥有以下方法：

Popen.**poll**()

检查子进程是否已被终止。设置并返回 *returncode* 属性。否则返回 None。

Popen.**wait**(*timeout=None*)

等待子进程被终止。设置并返回 *returncode* 属性。

如果进程在 *timeout* 秒后未中断，抛出一个 *TimeoutExpired* 异常，可以安全地捕获此异常并重新等待。

> **备注**
>
> 当 stdout=PIPE 或者 stderr=PIPE 并且子进程产生了足以阻塞 OS 管道缓冲区接收更多数据的输出到管道时，将会发生死锁。当使用管道时用 *Popen.communicate()* 来规避它。

> **备注**
>
> 当 timeout 形参不为 None 时，该函数（在 POSIX 上）将使用一个忙循环（非阻塞调用及短睡眠）来实现。使用 *asyncio* 模块进行异步等待：参见 *asyncio.create_subprocess_exec*。

在 3.3 版本发生变更: *timeout* 被添加

Popen.**communicate**(*input=None*, *timeout=None*)

与进程交互：将数据发送到 stdin。从 stdout 和 stderr 读取数据，直到抵达文件结尾。等待进程终止并设置 *returncode* 属性。可选的 *input* 参数应为要发送到下级进程的数据，或者如果没有要发送到下级进程的数据则为 None。如果流是以文本模式打开的，则 *input* 必须为字符串。在其他情况下，它必须为字节串。

communicate() 返回一个 (stdout_data, stderr_data) 元组。如果文件以文本模式打开则为字符串；否则字节。

注意如果你想要向进程的 stdin 传输数据，你需要通过 stdin=PIPE 创建此 Popen 对象。类似的，要从结果元组获取任何非 None 值，你同样需要设置 stdout=PIPE 或者 stderr=PIPE。

如果进程在 *timeout* 秒后未终止，一个 *TimeoutExpired* 异常将被抛出。捕获此异常并重新等待将不会丢失任何输出。

如果超时到期，子进程不会被杀死，所以为了正确清理一个行为良好的应用程序应该杀死子进程并完成通讯。

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

> **备注**
>
> 内存里数据读取是缓冲的，所以如果数据尺寸过大或无限，不要使用此方法。

在 3.3 版本发生变更: *timeout* 被添加

Popen.**send_signal**(*signal*)

将信号 *signal* 发送给子进程。

如果进程已完成则不做任何操作。

> **备注**
>
> 在 Windows 上，SIGTERM 是 *terminate()* 的别名。CTRL_C_EVENT 和 CTRL_BREAK_EVENT 可被发送给以包括 CREATE_NEW_PROCESS_GROUP 的 *creationflags* 形参来启动的进程。

Popen.**terminate**()

停止子进程。在 POSIX 操作系统上此方法会发送 *SIGTERM* 给子进程。在 Windows 上则会调用 Win32 API 函数 TerminateProcess() 来停止子进程。

Popen.**kill**()

杀死子进程。在 POSIX 操作系统上，此函数会发送 SIGKILL 给子进程。在 Windows 上 *kill()* 则是 *terminate()* 的别名。

下列属性也会通过类来设置以供你访问。将它们重赋新值是不受支持的:

Popen.**args**

args 参数传递给*Popen* -- 一个程序参数的序列或者一个简单字符串。

Added in version 3.3.

Popen.**stdin**

如果 *stdin* 参数为*PIPE*，此属性是一个类似*open()* 所返回对象的可写流对象。如果指定了 *encoding* 或 *errors* 参数或者 *text* 或 *universal_newlines* 参数为 `True`，则这个流将是一个文本流，否则将是一个字节流。如果 *stdin* 参数不为*PIPE*，则此属性将为 `None`。

Popen.**stdout**

如果 *stdout* 参数为*PIPE*，此属性是一个类似*open()* 所返回对象的可读流对象。从流中读取将会提供来自子进程的输出。如果 *encoding* 或 *errors* 参数被指定或者 *text* 或 *universal_newlines* 参数为 `True`，则这个流将是一个文本流，否则将是一个字节流。如果 *stdout* 参数不为*PIPE*，则此属性将为 `None`。

Popen.**stderr**

如果 *stderr* 参数为*PIPE*，此属性是一个类似*open()* 所返回对象的可读流对象。从流中读取将会提供来自子进程的错误输出。如果 *encoding* 或 *errors* 参数被指定或者 *text* 或 *universal_newlines* 参数为 `True`，则这个流将是一个文本流，否则将是一个字节流。如果 *stderr* 参数不为*PIPE*，则此属性将为 `None`。

警告

使用*communicate()* 而非*.stdin.write*，*.stdout.read* 或者*.stderr.read* 来避免由于任意其他 OS 管道缓冲区被子进程填满阻塞而导致的死锁。

Popen.**pid**

子进程的进程号。

注意如果你设置了 *shell* 参数为 `True`，则这是生成的子 shell 的进程号。

Popen.**returncode**

子进程的返回码。初始为 `None`，*returncode* 是通过在检测到进程终结时调用*poll()*，*wait()* 或*communicate()* 等方法来设置的。

`None` 值表示在最近一次方法调用时进程尚未终结

一个负值 `-N` 表示子进程被信号 `N` 中断 (仅 POSIX).

18.6.4 Windows Popen 助手

STARTUPINFO 类和以下常数仅在 Windows 有效。

class subprocess.**STARTUPINFO**(*, *dwFlags=0*, *hStdInput=None*, *hStdOutput=None*, *hStdError=None*, *wShowWindow=0*, *lpAttributeList=None*)

在*Popen* 创建时部分支持 Windows 的 STARTUPINFO 结构。接下来的属性仅能通过关键词参数设置。

在 3.7 版本发生变更: 仅关键词参数支持被加入。

dwFlags

一个位字段，用于确定进程在创建窗口时是否使用某些*STARTUPINFO* 属性。

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput

如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准输入句柄，如果*STARTF_USESTDHANDLES* 未指定，则默认的标准输入是键盘缓冲区。

hStdOutput

如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准输出句柄。除此之外，此此属性将被忽略并且默认标准输出是控制台窗口缓冲区。

hStdError

如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准错误句柄。除此之外，此属性将被忽略并且默认标准错误为控制台窗口的缓冲区。

wShowWindow

如果*dwFlags* 指定了*STARTF_USESHOWWINDOW*，此属性可为能被指定为函数 ShowWindow 的 nCmdShow 的形参的任意值，除了 `SW_SHOWDEFAULT`。如此之外，此属性被忽略。

SW_HIDE 被提供给此属性。它在*Popen* 由 `shell=True` 调用时使用。

lpAttributeList

`STARTUPINFOEX` 给出的用于进程创建的额外属性字典，参阅 UpdateProcThreadAttribute。

支持的属性：

handle_list
将被继承的句柄的序列。如果非空，*close_fds* 必须为 true。

当传递给*Popen* 构造函数时，这些句柄必须暂时地能被*os.set_handle_inheritable()* 继承，否则*OSError* 将以 Windows error `ERROR_INVALID_PARAMETER` (87) 抛出。

> **警告**
>
> 在多线程进程中，请谨慎使用，以便在将此功能与对继承所有句柄的其他进程创建函数——例如*os.system()* 的并发调用——相结合时，避免泄漏标记为可继承的句柄。这也应用于临时性创建可继承句柄的标准句柄重定向。

Added in version 3.7.

Windows 常数

subprocess 模块曝出以下常数。

subprocess.**STD_INPUT_HANDLE**
标准输入设备，这是控制台输入缓冲区 `CONIN$`。

subprocess.**STD_OUTPUT_HANDLE**
标准输出设备。最初，这是活动控制台屏幕缓冲区 `CONOUT$`。

subprocess.**STD_ERROR_HANDLE**
标准错误设备。最初，这是活动控制台屏幕缓冲区 `CONOUT$`。

subprocess.**SW_HIDE**
隐藏窗口。另一个窗口将被激活。

subprocess.**STARTF_USESTDHANDLES**
指明*STARTUPINFO.hStdInput*，*STARTUPINFO.hStdOutput* 和*STARTUPINFO.hStdError* 属性包含额外的信息。

subprocess.**STARTF_USESHOWWINDOW**
指明*STARTUPINFO.wShowWindow* 属性包含额外的信息。

subprocess.**STARTF_FORCEONFEEDBACK**
STARTUPINFO.dwFlags 形参指明在进程启动时将显示一个 正在后台操作鼠标提示。这是 GUI 进程的默认行为。

Added in version 3.13.

subprocess.**STARTF_FORCEOFFFEEDBACK**

A *STARTUPINFO.dwFlags* 形参指明在启动进程时鼠标提示将不会改变。

Added in version 3.13.

subprocess.**CREATE_NEW_CONSOLE**

新的进程将有新的控制台，而不是继承父进程的（默认）控制台。

subprocess.**CREATE_NEW_PROCESS_GROUP**

用于指明将创建一个新的进程组的*Popen* creationflags 形参。这个旗标对于在子进程上使用*os.kill()* 来说是必须的。

如果指定了*CREATE_NEW_CONSOLE* 则这个旗标会被忽略。

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**

用于指明一个新进程将具有高于平均的优先级的*Popen* creationflags 形参。

Added in version 3.7.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**

用于指明一个新进程将具有低于平均的优先级的*Popen* creationflags 形参。

Added in version 3.7.

subprocess.**HIGH_PRIORITY_CLASS**

用于指明一个新进程将具有高优先级的*Popen* creationflags 形参。

Added in version 3.7.

subprocess.**IDLE_PRIORITY_CLASS**

用于指明一个新进程将具有空闲（最低）优先级的*Popen* creationflags 形参。

Added in version 3.7.

subprocess.**NORMAL_PRIORITY_CLASS**

用于指明一个新进程将具有正常（默认）优先级的*Popen* creationflags 形参。

Added in version 3.7.

subprocess.**REALTIME_PRIORITY_CLASS**

用于指明一个新进程将具有实时优先级的*Popen* creationflags 形参。你应当几乎永远不使用 REALTIME_PRIORITY_CLASS，因为这会中断管理鼠标输入、键盘输入以及后台磁盘刷新的系统线程。这个类只适用于直接与硬件“对话”，或者执行短暂任务具有受限中断的应用。

Added in version 3.7.

subprocess.**CREATE_NO_WINDOW**

指明一个新进程将不会创建窗口的*Popen* creationflags 形参。

Added in version 3.7.

subprocess.**DETACHED_PROCESS**

指明一个新进程将不会继承其父控制台的*Popen* creationflags 形参。这个值不能与 CREATE_NEW_CONSOLE 一同使用。

Added in version 3.7.

subprocess.**CREATE_DEFAULT_ERROR_MODE**

指明一个新进程不会继承调用方进程的错误模式的*Popen* creationflags 形参。新进程会转为采用默认的错误模式。这个特性特别适用于运行时禁用硬错误的多线程 shell 应用。

Added in version 3.7.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**

指明一个新进程不会关联到任务的*Popen* creationflags 形参。

Added in version 3.7.

18.6.5 较旧的高阶 API

在 Python 3.5 之前，这三个函数组成了 subprocess 的高阶 API。现在你可以在许多情况下使用*run()*，但有大量现在代码仍会调用这些函数。

subprocess.**call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

运行由 *args* 所描述的命令。等待命令完成，然后返回*returncode* 属性。

需要捕获 stdout 或 stderr 的代码应当改用*run()*：

```
run(...).returncode
```

要屏蔽 stdout 或 stderr，可提供*DEVNULL* 这个值。

上面显示的参数只是常见的一些。完整的函数签名与*Popen* 构造器的相同——此函数会将所提供的 *timeout* 之外的全部参数直接传递给目标接口。

> **备注**
>
> 请不要在此函数中使用 `stdout=PIPE` 或 `stderr=PIPE`。如果子进程向管道生成了足以填满 OS 管理缓冲区的输出而管道还未被读取时它将会阻塞。

在 3.3 版本发生变更: *timeout* 被添加

在 3.12 版本发生变更: 针对 `shell=True` 改变的 Windows shell 搜索顺序。当前目录和 `%PATH%` 会被替换为 `%COMSPEC%` 和 `%SystemRoot%\System32\cmd.exe`。因此，在当前目录中投放一个命名为 `cmd.exe` 的恶意程序不会再起作用。

subprocess.**check_call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

附带参数运行命令。等待命令完成。如果返回码为零则正常返回，否则引发*CalledProcessError*。*CalledProcessError* 对象将在*returncode* 属性中保存返回码。如果*check_call()* 无法开始进程则它将传播已被引发的异常。

需要捕获 stdout 或 stderr 的代码应当改用*run()*：

```
run(..., check=True)
```

要屏蔽 stdout 或 stderr，可提供*DEVNULL* 这个值。

上面显示的参数只是常见的一些。完整的函数签名与*Popen* 构造器的相同——此函数会将所提供的 *timeout* 之外的全部参数直接传递给目标接口。

> **备注**
>
> 请不要在此函数中使用 `stdout=PIPE` 或 `stderr=PIPE`。如果子进程向管道生成了足以填满 OS 管理缓冲区的输出而管道还未被读取时它将会阻塞。

在 3.3 版本发生变更: *timeout* 被添加

在 3.12 版本发生变更: 针对 `shell=True` 改变的 Windows shell 搜索顺序。当前目录和 `%PATH%` 会被替换为 `%COMSPEC%` 和 `%SystemRoot%\System32\cmd.exe`。因此，在当前目录中投放一个命名为 `cmd.exe` 的恶意程序不会再起作用。

subprocess.**check_output**(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=None*, *timeout=None*, *text=None*, ***other_popen_kwargs*)

附带参数运行命令并返回其输出。

如果返回码非零则会引发*CalledProcessError*。*CalledProcessError* 对象将在*returncode* 属性中保存返回码并在*output* 属性中保存所有输出。

这相当于：

```
run(..., check=True, stdout=PIPE).stdout
```

上面显示的参数只是常见的一些。完整的函数签名与*run()* 的大致相同——大部分参数会通过该接口直接传递。存在一个与*run()* 行为不同的 API 差异：传递 `input=None` 的行为将与 `input=b''` (或 `input=''`，具体取决于其他参数) 一样而不是使用父对象的标准输入文件处理。

默认情况下，此函数将把数据返回为已编码的字节串。输出数据的实际编码格式将取决于唤起的命令，因此解码为文本的操作往往需要在应用程序层级上进行处理。

此行为可以通过设置 *text*, *encoding*, *errors* 或将 *universal_newlines* 设为 `True` 来重载，具体描述见*常用参数* 和*run()*。

要在结果中同时捕获标准错误，请使用 `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

Added in version 3.1.

在 3.3 版本发生变更: *timeout* 被添加

在 3.4 版本发生变更: 增加了对 *input* 关键字参数的支持。

在 3.6 版本发生变更: 增加了 *encoding* 和 *errors*。详情参见*run()*。

Added in version 3.7: *text* 作为 *universal_newlines* 的一个更具可读性的别名被添加。

在 3.12 版本发生变更: 针对 `shell=True` 改变的 Windows shell 搜索顺序。当前目录和 `%PATH%` 会被替换为 `%COMSPEC%` 和 `%SystemRoot%\System32\cmd.exe`。因此，在当前目录中投放一个命名为 `cmd.exe` 的恶意程序不会再起作用。

18.6.6 使用 subprocess 模块替换旧函数

在这一节中，”a 改为 b” 意味着 b 可以被用作 a 的替代。

备注

在这一节中的所有”a” 函数会在找不到被执行的程序时（差不多）静默地失败；”b” 替代函数则会改为引发*OSError*。

此外，在使用*check_output()* 时如果替代函数所请求的操作产生了非零返回值则将失败并引发*CalledProcessError*。操作的输出仍能以所引发异常的*output* 属性的方式被访问。

在下列例子中，我们假定相关的函数都已从*subprocess* 模块中导入了。

替代 /bin/sh shell 命令替换

```
output=$(mycmd myarg)
```

改为:

```
output = check_output(["mycmd", "myarg"])
```

替代 shell 管道

```
output=$(dmesg | grep hda)
```

改为:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # 允许 p1 在 p2 退出时接收 SIGPIPE。
output = p2.communicate()[0]
```

启动 p2 之后再执行 `p1.stdout.close()` 调用很重要，这是为了让 p1 能在 p2 先于 p1 退出时接收到 SIGPIPE。

另外，对于受信任的输入，shell 本身的管道支持仍然可被直接使用:

```
output=$(dmesg | grep hda)
```

改为:

```
output = check_output("dmesg | grep hda", shell=True)
```

替代 `os.system()`

```
sts = os.system("mycmd" + " myarg")
# 变为
retcode = call("mycmd" + " myarg", shell=True)
```

注释:

- 通过 shell 来调用程序通常是不必要的。
- *`call()`* 返回值的编码方式与*`os.system()`* 的不同。
- *`os.system()`* 函数在命令运行期间会忽略 SIGINT 和 SIGQUIT 信号，但调用方必须在使用*`subprocess`* 模块时分别执行此操作。

一个更现实的例子如下所示:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

替代 `os.spawn` 函数族

P_NOWAIT 示例:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT 示例:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector 示例:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment 示例:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

替代 os.popen(), os.popen2(), os.popen3()

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

返回码以如下方式处理转写:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

替换来自 popen2 模块的函数

> **备注**
>
> 如果 popen2 函数的 cmd 参数是一个字符串，命令会通过 /bin/sh 来执行。如果是一个列表，命令会被直接执行。

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
```

(续下页)

(接上页)

```
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` 和 `popen2.Popen4` 基本上类似于*`subprocess.Popen`*，不同之处在于:

- *`Popen`* 如果执行失败会引发一个异常。
- *capturestderr* 参数被替换为 *stderr* 参数。
- 必须指定 `stdin=PIPE` 和 `stdout=PIPE`。
- popen2 默认会关闭所有文件描述符，但对于*`Popen`* 你必须指明 `close_fds=True` 以才能在所有平台或较旧的 Python 版本中确保此行为。

18.6.7 旧式的 Shell 发起函数

此模块还提供了以下来自 2.x `commands` 模块的旧版函数。这些操作会隐式地唤起系统 shell 并且上文所描述的有关安全与异常处理一致性保证都不适用于这些函数。

`subprocess.`**`getstatusoutput`**(*cmd*, *, *encoding=None*, *errors=None*)

返回在 shell 中执行 *cmd* 产生的 `(exitcode, output)`。

在 shell 中使用 `Popen.check_output()` 来执行字符串 *cmd* 并返回一个 2 元组 `(exitcode, output)`。将使用 *encoding* 和 *errors* 来对输出进行解码；请参阅*常用参数* 中的说明来了解更多细节。

末尾的一个换行符会从输出中被去除。命令的退出码可被解读为子进程的返回码。例如:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Availability: Unix, Windows.

在 3.3.4 版本发生变更: 添加了 Windows 支持。

此函数现在返回 (exitcode, output) 而不是像 Python 3.3.3 及更早的版本那样返回 (status, output)。exitcode 的值与*`returncode`* 相同。

在 3.11 版本发生变更: 增加了 *encoding* 和 *errors* 形参。

`subprocess.`**`getoutput`**(*cmd*, *, *encoding=None*, *errors=None*)

返回在 shell 中执行 *cmd* 产生的输出（stdout 和 stderr）。

类似于*`getstatusoutput()`*，但退出码会被忽略并且返回值为包含命令输出的字符串。例如:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: Unix, Windows.

在 3.3.4 版本发生变更: 添加了 Windows 支持

在 3.11 版本发生变更: 增加了 *encoding* 和 *errors* 形参。

18.6.8 备注

在 Windows 上将参数列表转换为一个字符串

在 Windows 上，*args* 序列会被转换为可使用以下规则来解析的字符串（对应于 MS C 运行时所使用的规则）：

1. 参数以空白符分隔，即空格符或制表符。
2. 用双引号标示的字符串会被解读为单个参数，而不再考虑其中的空白符。一个参数可以嵌套用引号标示的字符串。
3. 带有一个反斜杠前缀的双引号会被解读为双引号字面值。
4. 反斜杠会按字面值解读，除非它是作为双引号的前缀。
5. 如果反斜杠被作为双引号的前缀，则每个反斜杠对会被解读为一个反斜杠字面值。如果反斜杠数量为奇数，则最后一个反斜杠会如规则 3 所描述的那样转义下一个双引号。

参见

shlex
此模块提供了用于解析和转义命令行的函数。

禁用 `vfork()` 或 `posix_spawn()`

在 Linux 上，*subprocess* 默认会在内部使用 `vfork()` 系统调用而不是 `fork()`，只要这样做是安全的。这极大地提升了性能。

如果你遇到了在预想中非常不正常的需要防止 `vfork()` 被 Python 所使用的情况，你可以将 `subprocess._USE_VFORK` 属性设为假值。

```
subprocess._USE_VFORK = False  # 参见 CPython 程序问题 gh-NNNNNN。
```

设置该值对于 `posix_spawn()` 的使用没有影响，它可以在内部使用在其 libc 实现中的 `vfork()`。如果你需要防止其被使用则可利用类似的 `subprocess._USE_POSIX_SPAWN` 属性。

```
subprocess._USE_POSIX_SPAWN = False  # 参见 CPython 程序问题 gh-NNNNNN。
```

在任何 Python 版本上将这些属性设为假值都是安全的。它们在不受支持的旧版本上将没有任何效果。请不要假定这些属性都是可读的。尽管它们被如此命名，但将其设为真值并不表示对应的函数将被使用，只表示有这样的可能性。

当你不得不使用这些私有属性并遇到问题时请随时提交问题并附带你所看到的问题的重现方式。请从你代码中的某条注释链接到该问题。

Added in version 3.8: `_USE_POSIX_SPAWN`

Added in version 3.11: `_USE_VFORK`

18.7 `sched` --- 事件调度器

源码： Lib/sched.py

---

sched 模块定义了一个实现通用事件调度程序的类：

class sched.**scheduler**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

scheduler 类定义了一个调度事件的通用接口。它需要两个函数来实际处理“外部世界”——*timefunc* 应当不带参数地调用，并返回一个数字（“时间”，可以为任意单位）。*delayfunc* 函数应当带一个参数调用，与 *timefunc* 的输出相兼容，并且应当延迟其所指定的时间单位。每个事件运行后还将调用 *delayfunc* 并传入参数 `0` 以允许其他线程有机会在多线程应用中运行。

在 3.3 版本发生变更: *timefunc* 和 *delayfunc* 参数是可选的。

在 3.3 版本发生变更: *scheduler* 类可以安全的在多线程环境中使用。

示例:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     # 虽然具有更高的优先级，'keyword' 将在 'positional' 之后运行因为 enter() 是相对的
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.enterabs(1_650_000_000, 10, print_time, argument=("first enterabs",))
...     s.enterabs(1_650_000_000, 5, print_time, argument=("second enterabs",))
...     s.run()
...     print(time.time())
...
>>> print_some_times()
1652342830.3640375
From print_time 1652342830.3642538 second enterabs
From print_time 1652342830.3643398 first enterabs
From print_time 1652342835.3694863 positional
From print_time 1652342835.3696074 keyword
From print_time 1652342840.369612 default
1652342840.3697174
```

18.7.1 调度器对象

scheduler 实例拥有以下方法和属性:

scheduler.**enterabs**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)

安排一个新事件。*time* 参数应该有一个数字类型兼容的返回值，与传递给构造函数的 *timefunc* 函数的返回值兼容。计划在相同 *time* 的事件将按其 *priority* 的顺序执行。数字越小表示优先级越高。

执行事件意为执行 `action(*argument, **kwargs)`。*argument* 是包含有 *action* 的位置参数的序列。*kwargs* 是包含 *action* 的关键字参数的字典。

返回值是一个事件，可用于以后取消事件（参见*cancel()*）。

在 3.3 版本发生变更: *argument* 参数是可选的。

在 3.3 版本发生变更: 添加了 *kwargs* 形参。

scheduler.**enter**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)

安排延后 *delay* 时间单位的事件。除了时间是相对的，其他参数、效果和返回值与*enterabs()*相同。

在 3.3 版本发生变更: *argument* 参数是可选的。

在 3.3 版本发生变更: 添加了 *kwargs* 形参。

scheduler.**cancel**(*event*)

从队列中删除事件。如果 *event* 不是当前队列中的事件，则此方法将引发*ValueError*。

scheduler.**empty**()

如果事件队列为空则返回 True。

scheduler.**run**(*blocking=True*)

运行所有计划事件。此方法将等待（使用传递给构造器的 *delayfunc* 函数）进行下一个事件，然后执行它，依此类推直到没有更多的计划事件。

如果 *blocking* 为 false，则执行最快到期（如果有）的预定事件，然后在调度程序中返回下一个预定调用的截止时间（如果有）。

action 或 *delayfunc* 都可以引发异常。在任何一种情况下，调度程序都将保持一致状态并传播异常。如果 *action* 引发异常，则在将来调用 *run()* 时不会尝试该事件。

如果一系列事件的运行时间大于下一个事件发生前的可用时间，那么调度程序只会保持落后。没有事件会被丢弃；调用代码负责取消不再相关的事件。

在 3.3 版本发生变更: 添加了 *blocking* 形参。

scheduler.**queue**

只读属性，按照计划运行的顺序返回即将发生的事件列表。每个事件都显示为*named tuple* ，包含以下字段：time、priority、action、argument、kwargs。

18.8 queue --- 同步队列类

源代码: Lib/queue.py

---

queue 模块实现了多生产者、多消费者队列。这特别适用于消息必须安全地在多线程间交换的线程编程。模块中的*Queue* 类实现了所有所需的锁定语义。

本模块实现了三种类型的队列，它们的区别仅仅是条目的提取顺序。在 FIFO 队列中，先添加的任务会先被提取。在 LIFO 队列中，最近添加的条目会先被提取 (类似于一个栈)。在优先级队列中，条目将保持已排序状态 (使用*heapq* 模块) 并且值最小的条目会先被提取。

在内部，这三个类型的队列使用锁来临时阻塞竞争线程；然而，它们并未被设计用于线程的重入性处理。

此外，模块实现了一个" 简单的" FIFO 队列类型，*SimpleQueue* ，这个特殊实现为小功能在交换中提供额外的保障。

queue 模块定义了下列类和异常：

class queue.**Queue**(*maxsize=0*)

Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.**LifoQueue**(*maxsize=0*)

LIFO 队列构造函数。*maxsize* 是个整数，用于设置可以放入队列中的项目数的上限。当达到这个大小的时候，插入操作将阻塞至队列中的项目被消费掉。如果 *maxsize* 小于等于零，队列尺寸为无限大。

class queue.**PriorityQueue**(*maxsize=0*)

优先级队列构造函数。*maxsize* 是个整数，用于设置可以放入队列中的项目数的上限。当达到这个大小的时候，插入操作将阻塞至队列中的项目被消费掉。如果 *maxsize* 小于等于零，队列尺寸为无限大。

值最小的条目会先被提取 (值最小的条目是由 the lowest valued entry is the one that would be returned by `min(entries)` 返回的)。条目的典型模式是如下形式的元组: `(priority_number, data)`。

如果 *data* 元素没有可比性，数据将被包装在一个类中，忽略数据值，仅仅比较优先级数字：

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.**SimpleQueue**

无界的 FIFO 队列构造函数。简单的队列，缺少任务跟踪等高级功能。

Added in version 3.7.

exception queue.**Empty**

对空的*Queue* 对象，调用非阻塞的*get()* (or *get_nowait()*) 时，引发的异常。

exception queue.**Full**

对满的*Queue* 对象，调用非阻塞的*put()* (or *put_nowait()*) 时，引发的异常。

exception queue.**ShutDown**

当在已被关闭的*Queue* 对象上调用*put()* 或*get()* 时引发的异常。

Added in version 3.13.

18.8.1 Queue 对象

队列对象 (*Queue*, *LifoQueue*, 或者*PriorityQueue*) 提供下列描述的公共方法。

Queue.**qsize**()

返回队列的大致大小。注意，qsize() > 0 不保证后续的 get() 不被阻塞，qsize() < maxsize 也不保证 put() 不被阻塞。

Queue.**empty**()

如果队列为空，返回 `True` ，否则返回 `False` 。如果 empty() 返回 `True` ，不保证后续调用的 put() 不被阻塞。类似的，如果 empty() 返回 `False` ，也不保证后续调用的 get() 不被阻塞。

Queue.**full**()

如果队列是满的返回 `True` ，否则返回 `False` 。如果 full() 返回 `True` 不保证后续调用的 get() 不被阻塞。类似的，如果 full() 返回 `False` 也不保证后续调用的 put() 不被阻塞。

Queue.**put**(*item*, *block=True*, *timeout=None*)

将 *item* 加入队列。如果可选参数 *block* 为真值并且 *timeout* 为 `None` (默认值)，则会在必要时阻塞直到有空闲槽位可用。如为 *timeout* 为正数，则将阻塞最多 *timeout* 秒并会在没有可用的空闲槽位时引发*Full* 异常。在其他情况下 (*block* 为假值)，则如果空闲槽位立即可用则将条目加入队列，否则将引发*Full* 异常 (*timeout* 在此情况下将被忽略)。

如果队列已被关闭则会引发*ShutDown*。

Queue.**put_nowait**(*item*)

相当于 `put(item, block=False)` 。

Queue.**get**(*block=True*, *timeout=None*)

从队列中移除并返回一个项目。如果可选参数 *block* 是 true 并且 *timeout* 是 `None` (默认值)，则在必要时阻塞至项目可得到。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间内项目不能得到，将引发*Empty* 异常。反之 (*block* 是 false)，如果一个项目立即可得到，则返回一个项目，否则引发*Empty* 异常 (这种情况下，*timeout* 将被忽略)。

POSIX 系统上在 3.0 之前，以及在 Windows 上的所有版本中，如果 *block* 为真值并且 *timeout* 为 `None`，此操作将进入在底层锁上的不可中断的等待。这意味着不会发生任何异常，特别是 SIGINT 将不会触发*KeyboardInterrupt*。

如果队列已被关闭并且为空，或者如果队列已被立即关闭则会引发*ShutDown*。

Queue.**get_nowait**()

相当于 `get(False)`。

提供了两个方法，用于支持跟踪排队的任务是否被守护的消费者线程完整的处理。

Queue.**task_done**()

表示前面排队的任务已经被完成。被队列的消费者线程使用。每个*get()* 被用于获取一个任务，后续调用*task_done()* 告诉队列，该任务的处理已经完成。

如果*join()* 当前正在阻塞，在所有条目都被处理后，将解除阻塞(意味着每个*put()* 进队列的条目的*task_done()* 都被收到)。

`shutdown(immediate=True)` 将为队列中每个剩余的项调用*task_done()*。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 异常。

Queue.**join**()

阻塞至队列中所有的元素都被接收和处理完毕。

当一个条目被添加到队列的时候未完成任务的计数将会增加。每当一个消费者线程调用*task_done()* 来表明该条目已被提取且其上的所有工作已完成时未完成计数将会减少。当未完成计数降为零时，*join()* 将解除阻塞。

如何等待排队的任务被完成的示例：

```
import threading
import queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# 启动工作线程。
threading.Thread(target=worker, daemon=True).start()

# 向工作线程发送三十个任务请求。
for item in range(30):
    q.put(item)

# 阻塞直到所有任务完成。
q.join()
print('All work completed')
```

终结队列

Queue 可以通过将其关闭以防止继续交互。

Queue.**shutdown**(*immediate=False*)

关闭队列，让*get()* 和*put()* 引发*ShutDown*。

在默认情况下，在已关闭的队列上执行*get()* 只会在队列为空时引发异常。将 *immediate* 设为真值以改为让*get()* 立即引发异常。

所有*put()* 和*get()* 被阻塞的调用方将被撤销阻塞。如果 *immediate* 为真值，一个任务将对队列中每个剩余的项标记为已完成，它可能撤销对*join()* 的调用方的阻塞。

Added in version 3.13.

18.8.2 SimpleQueue 对象

SimpleQueue 对象提供下列描述的公共方法。

SimpleQueue.**qsize**()

返回队列的大致大小。注意，qsize() > 0 不保证后续的 get() 不被阻塞。

SimpleQueue.**empty**()

如果队列为空则返回 True，否则返回 False。如果 empty() 返回 False 则不保证后续对 get() 的调用将不会阻塞。

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)

将 *item* 放入队列。此方法永不阻塞，始终成功（除了潜在的低级错误，例如内存分配失败）。可选参数 *block* 和 *timeout* 仅仅是为了保持*Queue.put()* 的兼容性而提供，其值被忽略。

此方法具有一个可重入的 C 实现。也就是说，一个 put() 或 get() 调用可以被同一线程中的另一个 put() 调用打断而不会发生死锁或破坏队列内部的状态。这使得它适用于析构器如 __del__ 方法或*weakref* 回调。

SimpleQueue.**put_nowait**(*item*)

相当于 put(item, block=False)，为保持与*Queue.put_nowait()* 的兼容性而提供。

SimpleQueue.**get**(*block=True*, *timeout=None*)

从队列中移除并返回一个项目。如果可选参数 *block* 是 true 并且 *timeout* 是 None (默认值)，则在必要时阻塞至项目可得到。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间内项目不能得到，将引发*Empty* 异常。反之 (*block* 是 false)，如果一个项目立即可得到，则返回一个项目，否则引发*Empty* 异常 (这种情况下，*timeout* 将被忽略)。

SimpleQueue.**get_nowait**()

相当于 get(False) 。

参见

类*multiprocessing.Queue*

一个用于多进程上下文的队列类（而不是多线程）。

collections.deque 是无界队列的一个替代实现，具有快速的不需要锁并且支持索引的原子化*append()* 和*popleft()* 操作。

18.9 contextvars --- 上下文变量

本模块提供了相关 API 用于管理、存储和访问上下文相关的状态。*ContextVar* 类用于声明 上下文变量并与其一起使用。函数*copy_context()* 和类*Context* 用于管理当前上下文和异步框架中。

当在并发代码中使用时，有状态的上下文管理器应当使用上下文变量而不是*threading.local()* 以防止它们的状态意外地泄露到其他代码中。

更多信息参见 **PEP 567** 。

Added in version 3.7.

18.9.1 上下文变量

class contextvars.**ContextVar**(*name*[, *, *default*])

此类用于声明一个新的上下文变量，如:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

name 参数用于内省和调试，必需。

调用*ContextVar.get()* 时，如果上下文中没有找到此变量的值，则返回可选的仅命名参数 *default* 。

重要：上下文变量应该在顶级模块中创建，且永远不要在闭包中创建。*Context* 对象拥有对上下文变量的强引用，这可以让上下文变量被垃圾收集器正确回收。

name

上下文变量的名称，只读属性。

Added in version 3.7.1.

get(*[default]*)

返回当前上下文中此上下文变量的值。

如果当前上下文中此变量没有值，则此方法会:

- 如果提供了 *default*，返回其值；或者
- 返回上下文变量本身的默认值，如果创建此上下文变量时提供了默认值；或者
- 抛出*LookupError* 异常。

set(*value*)

调用此方法设置上下文变量在当前上下文中的值。

必选参数 *value* 是上下文变量的新值。

返回一个*Token* 对象，可通过*ContextVar.reset()* 方法将上下文变量还原为之前某个状态。

reset(*token*)

将上下文变量重置为调用*ContextVar.set()* 之前、创建 *token* 时候的状态。

例如:

```
var = ContextVar('var')

token = var.set('new value')
# 使用 'var' 的代码; var.get() 将返回 'new value'。
var.reset(token)

# 在重置调用之后 var 将不再有值，
# 因此 var.get() 将引发一个 LookupError。
```

class contextvars.**Token**

ContextVar.set() 方法返回 *Token* 对象。此对象可以传递给*ContextVar.reset()* 方法用于将上下文变量还原为调用 *set* 前的状态。

var

只读属性。指向创建此 token 的*ContextVar* 对象。

old_value

一个只读属性。会被设为在创建此令牌的*ContextVar.set()* 方法调用之前该变量所具有的值。如果调用之前变量没有设置值则它会指令*Token.MISSING*。

MISSING

Token.old_value 会用到的一个标记对象。

18.9.2 手动上下文管理

contextvars.**copy_context**()

返回当前上下文中*Context* 对象的拷贝。

以下代码片段会获取当前上下文的拷贝并打印设置到其中的所有变量及其值:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

此函数具有 *O*(1) 复杂度，也就是说对于只包含几个上下文变量和很多上下文变量的情况运行速度是相同的。

class contextvars.**Context**

ContextVars 与其值的映射。

`Context()` 创建一个不包含任何值的空上下文。如果要获取当前上下文的拷贝，使用*copy_context()* 函数。

每个线程有它自己在用的 `Context` 对象栈。*current context* 是位于当前线程的栈的顶部的 `Context` 对象。栈中所有的 `Context` 对象都被视为是 进入过的。

进入一个上下文，这可以通过调用其*run()* 方法来完成，将把该上下文推入当前线程的栈的顶部使它成为当前上下文。

退出当前上下文，这可以通过从传给*run()* 方法的回调退出，通过将该上下文弹出上下文栈的顶部将当前上下文恢复至该上下文被进入之前的上下文来完成。

由于每个线程都有它自己的上下文栈，*ContextVar* 对象的行为类似于*threading.local()* 在不同线程中赋值时的行为。

尝试进入一个已被进入的上下文，包括在其他线程中被进入的上下文，将会引发*RuntimeError*。

在退出一个上下文之后，它可以在稍后被重新进入（从任何线程）。

任何通过*ContextVar.set()* 方法对*ContextVar* 值的修改都将在当前上下文中被记录。*ContextVar.get()* 方法将返回关联到当前上下文的值。退出一个上下文肯定会撤销在进入该上下文期间对上下文变量的任何修改（如有必要，这些值可通过重新进入相应的上下文来恢复）。

Context 实现了*collections.abc.Mapping* 接口。

run(*callable*, **args*, ***kwargs*)

进入 Context，执行 `callable(*args, **kwargs)`，然后退出 Context。返回 *callable* 的返回值，或者如果发生了异常则传播该异常。

示例:

```
import contextvars

var = contextvars.ContextVar('var')
var.set('spam')
print(var.get())  # 'spam'

ctx = contextvars.copy_context()

def main():
    # 在调用 'copy_context()' 和 'ctx.run(main)' 之前
    # 'var' 被设为 'spam', 因此:
    print(var.get())  # 'spam'
    print(ctx[var])  # 'spam'

    var.set('ham')

    # 在将 'var' 设为 'ham' 之后:
    print(var.get())  # 'ham'
    print(ctx[var])  # 'ham'

# 'main' 函数对 'var' 的任何修改
# 都将包含在 'ctx' 中。
ctx.run(main)

# 'main()' 函数是在 'ctx' 上下文中运行的,
```

(续下页)

(接上页)

```
# 因此对 'var' 的修改将包含在其中:
print(ctx[var])  # 'ham'

# 不过，在 'ctx' 之外，'var' 仍为 'spam':
print(var.get())  # 'spam'
```

copy()

返回此上下文对象的浅拷贝。

var in context

如果 *context* 中含有名称为 *var* 的变量，返回 `True`，否则返回 `False`。

context[var]

返回名称为 *var* 的*ContextVar* 变量。如果上下文对象中不包含这个变量，则抛出*KeyError* 异常。

get(*var*[, *default*])

如果 *var* 在上下文对象中具有值则返回 *var* 的值。在其他情况下返回 *default*。如果未给出 *default* 则返回 `None`。

iter(context)

返回一个存储在上下文对象中的变量的迭代器。

len(proxy)

返回上下文对象中所设的变量的数量。

keys()

返回上下文对象中的所有变量的列表。

values()

返回上下文对象中所有变量值的列表。

items()

返回包含上下文对象中所有变量及其值的 2 元组的列表。

18.9.3 asyncio 支持

上下文变量在*asyncio* 中有原生的支持并且无需任何额外配置即可被使用。例如，以下是一个简单的回显服务器，它使用上下文变量来让远程客户端的地址在处理该客户端的 Task 中可用:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # 可直接访问当前处理的客户端地址
    # 而无需显式地将其传给此函数。

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\r\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # 现在将可以在我们所调用的任何代码中通过
    # 'client_addr_var.get()' 调用来获取客户端地址。

    while True:
```

(续下页)

(接上页)

```
        line = await reader.readline()
        print(line)
        if not line.strip():
            break

    writer.write(b'HTTP/1.1 200 OK\r\n')  # 状态行
    writer.write(b'\r\n')  # 标头
    writer.write(render_goodbye())  # 消息体
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# 要进行测试你可以使用 telnet 或 curl:
#     telnet 127.0.0.1 8081
#     curl 127.0.0.1:8081
```

以下是上述某些服务的支持模块:

18.10 _thread --- 低层级多线程 API

该模块提供了操作多个线程（也被称为 *轻量级进程* 或 *任务*）的底层原语——多个控制线程共享全局数据空间。为了处理同步问题，也提供了简单的锁机制（也称为 *互斥锁* 或 *二进制信号*）。*threading* 模块基于该模块提供了更易用的高级多线程 API。

在 3.7 版本发生变更: 这个模块曾经为可选项，但现在总是可用。

这个模块定义了以下常量和函数:

exception _thread.**error**

发生线程相关错误时抛出。

在 3.3 版本发生变更: 现在是内建异常*RuntimeError* 的别名。

_thread.**LockType**

锁对象的类型。

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])

开启一个新线程并返回其标识。线程执行函数 *function* 并附带参数列表 *args* (必须是元组)。可选的 *kwargs* 参数指定一个关键字参数字典。

当函数返回时，线程会静默地退出。

当函数因某个未处理异常而终结时，*sys.unraisablehook()* 会被调用以处理异常。钩子参数的 *object* 属性为 *function*。在默认情况下，会打印堆栈回溯然后该线程将退出（但其他线程会继续运行）。

当函数引发*SystemExit* 异常时，它会被静默地忽略。

引发一个审计事件 _thread.start_new_thread 并附带参数 function, args, kwargs。

在 3.8 版本发生变更: 现在会使用*sys.unraisablehook()* 来处理未处理的异常。

_thread.**interrupt_main**(*signum=signal.SIGINT*, /)

模拟一个信号到达主线程的效果。线程可使用此函数来打断主线程，虽然并不保证打断将立即发生。

如果给出 *signum*，则表示要模拟的信号的编号。如果未给出 *signum*，则将模拟*signal.SIGINT*。

如果给出的信号未被 Python 处理 (它被设为*signal.SIG_DFL* 或*signal.SIG_IGN*)，则此函数将不做任何操作。

在 3.10 版本发生变更: 添加了 *signum* 参数来定制信号的编号。

备注

这并不会发出对应的信号而是将一个调用排入关联处理器的计划任务（如果句柄存在的话）。如果你想要真的发出信号，请使用*signal.raise_signal()*。

_thread.**exit**()

抛出*SystemExit* 异常。如果没有捕获的话，这个异常会使线程退出。

_thread.**allocate_lock**()

返回一个新的锁对象。锁中的方法在后面描述。初始情况下锁处于解锁状态。

_thread.**get_ident**()

返回当前线程的“线程标识符”。它是一个非零的整数。它的值没有直接含义，主要是用作 magic cookie，比如作为含有线程相关数据的字典的索引。线程标识符可能会在线程退出，新线程创建时被复用。

_thread.**get_native_id**()

返回内核分配给当前线程的原生集成线程 ID。这是一个非负整数。它的值可被用来在整个系统中唯一地标识这个特定线程（直到线程终结，在那之后该值可能会被 OS 回收再利用）。

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

在 3.13 版本发生变更: 增加了对 GNU/kFreeBSD 的支持。

_thread.**stack_size**([*size*])

返回创建线程时使用的堆栈大小。可选参数 *size* 指定之后新建的线程的堆栈大小，而且一定要是 0（根据平台或者默认配置）或者最小是 32,768(32KiB) 的一个正整数。如果 *size* 没有指定，默认是 0。如果不支持改变线程堆栈大小，会抛出*RuntimeError* 错误。如果指定的堆栈大小不合法，会抛出*ValueError* 错误并且不会修改堆栈大小。32KiB 是当前最小的能保证解释器有足够堆栈空间的堆栈大小。需要注意的是部分平台对于堆栈大小会有特定的限制，例如要求大于 32KiB 的堆栈大小或者需要根据系统内存页面的整数倍进行分配 - 应当查阅平台文档有关详细信息（4KiB 页面比较普遍，在没有更具体信息的情况下，建议的方法是使用 4096 的倍数作为堆栈大小）。

Availability: Windows, pthreads.

带有 POSIX 线程支持的 Unix 平台。

_thread.**TIMEOUT_MAX**

Lock.acquire 的 *timeout* 形参所允许的最大值。指定大于该值的 timeout 将引发*OverflowError*。

Added in version 3.2.

锁对象有以下方法：

lock.**acquire**(*blocking=True*, *timeout=-1*)

没有任何可选参数时，该方法无条件申请获得锁，有必要的话会等待其他线程释放锁（同时只有一个线程能获得锁——这正是锁存在的原因）。

如果提供了 *blocking* 参数，具体的行为将取决于它的值：如果它为假值，则只在能够立即获取到锁而无需等待时才会获取，而如果它为真值，则会与上面一样无条件地获取锁。

如果提供了浮点数形式的 *timeout* 参数且为正值，它将指明在返回之前的最大等待秒数。负的 *timeout* 参数表示无限期的等待。如果 *blocking* 为假值则你不能指定 *timeout*。

如果成功获取到所会返回 `True`，否则返回 `False`。

在 3.2 版本发生变更: 新的 *timeout* 形参。

在 3.2 版本发生变更: 现在获取锁的操作可以被 POSIX 信号中断。

`lock.`**`release`**`()`

释放锁。锁必须已经被获取过，但不一定是同一个线程获取的。

`lock.`**`locked`**`()`

返回锁的状态：如果已被某个线程获取，返回 `True`，否则返回 `False`。

除了这些方法之外，锁对象也可以通过 `with` 语句使用，例如：

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("在执行这段代码时，a_lock 已被锁定")
```

注意事项：

- 中断总是会到主线程 (*KeyboardInterrupt* 异常将由该线程接收。)
- 调用 *sys.exit()* 或是抛出 *SystemExit* 异常等效于调用 *_thread.exit()*。
- 一个锁的 *acquire()* 方法是否可被中断（这样 *KeyboardInterrupt* 异常将立即发生，而不是要等到获取锁或者操作超时）取决于具体的平台。它在 POSIX 上可被中断，但在 Windows 上则不可以。
- 当主线程退出时，由系统决定其他线程是否存活。在大多数系统中，这些线程会直接被杀掉，不会执行 `try` ... `finally` 语句，也不会执行对象析构函数。

CHAPTER 19

网络和进程间通信

本章介绍的模块提供了网络和进程间通信的机制。

某些模块仅适用于同一台机器上的两个进程，例如*signal* 和*mmap* 。其他模块支持两个或多个进程可用于跨机器通信的网络协议。

本章中描述的模块列表是：

19.1 asyncio --- 异步 I/O

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio 是用来编写 **并发**代码的库，使用 **async/await** 语法。

asyncio 被用作多个提供高性能 Python 异步框架的基础，包括网络和网站服务，数据库连接库，分布式任务队列等等。

asyncio 往往是构建 IO 密集型和高层级 **结构化**网络代码的最佳选择。

asyncio 提供一组 **高层级** API 用于：

- 并发地运行 *Python* 协程 并对其执行过程实现完全控制；
- 执行网络 *IO* 和 *IPC*；
- 控制子进程；
- 通过队列 实现分布式任务；
- 同步 并发代码；

此外，还有一些 **低层级** API 以支持 库和框架的开发者实现:

- 创建和管理事件循环，它提供用于连接网络, 运行子进程, 处理*OS 信号* 等功能的异步 API；
- 使用*transports* 实现高效率协议；
- 通过 async/await 语法桥接 基于回调的库和代码。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

asyncio REPL

你可以在*REPL* 中尝试使用 `asyncio` 并发上下文:

```
$ python -m asyncio
asyncio REPL ...
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
'hello'
```

引发一个不带参数的审计事件 `cpython.run_stdin`。

在 3.12.5 版本发生变更: （还有 3.11.10, 3.10.15, 3.9.20 和 3.8.20）将发出审计事件。

在 3.13 版本发生变更: 如果无法做到则使用 PyREPL，在此情况下 `PYTHONSTARTUP` 也会被执行。将发出审计事件。

参考

19.1.1 运行器

源代码: Lib/asyncio/runners.py

本节将简述用于运行异步代码的高层级异步原语。

它们构建于事件循环 之上，其目标是简化针对常见通用场景的异步代码的用法。

- 运行 *asyncio* 程序
- 运行器上下文管理器
- 处理键盘中断

运行 asyncio 程序

`asyncio.run`(*coro*, *, *debug=None*, *loop_factory=None*)

执行*coroutine* *coro* 并返回结果。

此函数会运行传入的协程，负责管理 asyncio 事件循环，终结异步生成器，并关闭执行器。

当有其他 asyncio 事件循环在同一线程中运行时，此函数不能被调用。

如果 *debug* 为 `True`，事件循环将运行于调试模式。`False` 将显式地禁用调试模式。使用 `None` 将沿用全局*Debug 模式* 设置。

如果 *loop_factory* 不为 `None`，它将被用来创建一个新的事件循环；否则将会使用*asyncio.new_event_loop()*。最终该循环将被关闭。此函数应当被用作 asyncio 程序的主入口点，在理想情况下应当只被调用一次。建议使用 *loop_factory* 来配置事件循环而不是使用策略。传入*asyncio.EventLoop* 将允许不带策略系统地运行 asyncio。

执行器的关闭有 5 分钟的超时限制。如果执行器未在时限之内结束，将发出警告消息并关闭执行器。

示例：

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

Added in version 3.7.

在 3.9 版本发生变更: 更新为使用 *loop.shutdown_default_executor()*。

在 3.10 版本发生变更: 默认情况下 *debug* 为 None 即沿用全局调试模式设置。

在 3.12 版本发生变更: 增加了 *loop_factory* 形参。

运行器上下文管理器

class asyncio.**Runner**(*, *debug=None*, *loop_factory=None*)

对在相同上下文中 多个异步函数调用进行简化的上下文管理器。

有时多个最高层级异步函数应当在同一个事件循环 和 *contextvars.Context* 中被调用。

如果 *debug* 为 True，事件循环将运行于调试模式。False 将显式地禁用调试模式。使用 None 将沿用全局 *Debug 模式* 设置。

loop_factory 可被用来重载循环的创建。*loop_factory* 要负责将所创建的循环设置为当前事件循环。在默认情况下如果 *loop_factory* 为 None 则会使用 *asyncio.new_event_loop()* 并通过 *asyncio.set_event_loop()* 将其设置为当前事件循环。

基本上，*asyncio.run()* 示例可以通过运行器的用法来重写:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

with asyncio.Runner() as runner:
    runner.run(main())
```

Added in version 3.11.

run(*coro*, *, *context=None*)

在嵌入的循环中运行一个协程 *coro*。

返回协程的结果或者引发其异常。

可选的仅限关键字参数 *context* 允许指定一个自定义 *contextvars.Context* 用作 *coro* 运行所在的上下文。如果为 None 则会使用运行器的默认上下文。

当有其他 asyncio 事件循环在同一线程中运行时，此函数不能被调用。

close()

关闭运行器。

最终化异步生成器，停止默认执行器，关闭事件循环并释放嵌入的 *contextvars.Context*。

get_loop()

返回关联到运行器实例的事件循环。

备注

Runner 会使用惰性初始化策略，它的构造器不会初始化下层的低层级结构体。

嵌入的 *loop* 和 *context* 是在进入 `with` 语句体或者对*run()* 或*get_loop()* 的首次调用时被创建的。

处理键盘中断

Added in version 3.11.

当*signal.SIGINT* 被 `Ctrl-C` 引发时，默认将在主线程中引发*KeyboardInterrupt*。但是这并不适用于*asyncio* 因为它可以中断异步的内部操作并能挂起要退出的程序。

为解决此问题，*asyncio* 将按以下步骤处理*signal.SIGINT*:

1. *asyncio.Runner.run()* 在任何用户代码被执行之前安装一个自定义的*signal.SIGINT* 处理器并在从该函数退出时将其移除。
2. *Runner* 为所传入的协程创建主任务供其执行。creates the main task for the passed coroutine for its execution.
3. 当*signal.SIGINT* 被 `Ctrl-C` 引发时，自定义的信号处理器将通过调用*asyncio.Task.cancel()* 在主任务内部引发*asyncio.CancelledError* 来取消主任务。这将导致 Python 栈回退，`try/except` 和 `try/finally` 代码块可被用于资源清理。在主任务被取消之后，*asyncio.Runner.run()* 将引发*KeyboardInterrupt*。
4. 用户可以编写无法通过*asyncio.Task.cancel()* 来中断的紧密循环，在这种情况下后续的第二次 `Ctrl-C` 将立即引发*KeyboardInterrupt* 而不会取消主任务。

19.1.2 协程与任务

本节将简述用于协程与任务的高层级 API。

- 协程
- 可等待对象
- 创建任务
- 任务取消
- 任务组
- 休眠
- 并发运行任务
- 主动任务工厂
- 屏蔽取消操作
- 超时
- 简单等待
- 在线程中运行
- 跨线程调度
- 内省
- *Task* 对象

协程

源码： Lib/asyncio/coroutines.py

通过 async/await 语法来声明协程 是编写 asyncio 应用的推荐方式。例如，以下代码段会打印”hello”，等待 1 秒，再打印”world”:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

注意：简单地调用一个协程并不会使其被调度执行

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

要实际运行一个协程，asyncio 提供了以下几种机制:

- *asyncio.run()* 函数用来运行最高层级的入口点”main()” 函数 (参见上面的示例。)
- 对协程执行 await。以下代码段会在等待 1 秒后打印”hello”，然后 再次等待 2 秒后打印”world”:

  ```
  import asyncio
  import time

  async def say_after(delay, what):
      await asyncio.sleep(delay)
      print(what)

  async def main():
      print(f"started at {time.strftime('%X')}")

      await say_after(1, 'hello')
      await say_after(2, 'world')

      print(f"finished at {time.strftime('%X')}")

  asyncio.run(main())
  ```

  预期的输出:

  ```
  started at 17:13:52
  hello
  world
  finished at 17:13:55
  ```

- *asyncio.create_task()* 函数用来并发运行作为 asyncio 任务 的多个协程。

  让我们修改以上示例，并发运行两个 `say_after` 协程:

  ```
  async def main():
      task1 = asyncio.create_task(
          say_after(1, 'hello'))

      task2 = asyncio.create_task(
  ```

(续下页)

(接上页)

```
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # 等待直到两个任务都完成
    # (会花费约 2 秒钟。)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

注意，预期的输出显示代码段的运行时间比之前快了 1 秒:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

- *asyncio.TaskGroup* 类提供了*create_task()* 的更现代化的替代。使用此 API，之前的例子将变为:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(
            say_after(1, 'hello'))

        task2 = tg.create_task(
            say_after(2, 'world'))

        print(f"started at {time.strftime('%X')}")

    # 当存在上下文管理器时 await 是隐式执行的。

    print(f"finished at {time.strftime('%X')}")
```

用时和输出结果应当与之前的版本相同。

Added in version 3.11: *asyncio.TaskGroup*。

可等待对象

如果一个对象可以在 `await` 语句中使用，那么它就是 **可等待**对象。许多 asyncio API 都被设计为接受可等待对象。

可等待对象有三种主要类型: **协程**, **任务**和 **Future**.

协程

Python 协程属于 可等待对象，因此可以在其他协程中被等待:

```
import asyncio

async def nested():
    return 42

async def main():
    # 如果我们只调用 "nested()" 则无事发生。
    # 一个协程对象会被创建但是不会被等待，
    # 因此它 *根本不会运行*。
    nested()  # 将引发 "RuntimeWarning"。
```

(续下页)

(接上页)

```
    # 现在让我们改为等待它:
    print(await nested())  # 将打印 "42"。

asyncio.run(main())
```

重要

在本文档中" 协程" 可用来表示两个紧密关联的概念:

- 协程函数: 定义形式为 async def 的函数;
- 协程对象: 调用 协程函数所返回的对象。

任务

任务被用来“并行的”调度协程

当一个协程通过*asyncio.create_task()* 等函数被封装为一个 任务，该协程会被自动调度执行:

```
import asyncio

async def nested():
    return 42

async def main():
    # 将 nested() 加入计划任务
    # 立即与 "main()" 并发运行。
    task = asyncio.create_task(nested())

    # 现在可以使用 "task" 来取消 "nested()", or
    # 或简单地等待它直到它被完成:
    await task

asyncio.run(main())
```

Futures

Future 是一种特殊的 **低层级**可等待对象，表示一个异步操作的 **最终结果**。

当一个 Future 对象 被等待，这意味着协程将保持等待直到该 Future 对象在其他地方操作完毕。

在 asyncio 中需要 Future 对象以便允许通过 async/await 使用基于回调的代码。

通常情况下 **没有必要**在应用层级的代码中创建 Future 对象。

Future 对象有时会由库和某些 asyncio API 暴露给用户，用作可等待对象:

```
async def main():
    await function_that_returns_a_future_object()

    # 这样也可以:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

一个很好的返回对象的低层级函数的示例是*loop.run_in_executor()*。

创建任务

源码： Lib/asyncio/tasks.py

---

asyncio.**create_task**(*coro*, *, *name=None*, *context=None*)

将 *coro* 协程 封装为一个*Task* 并调度其执行。返回 Task 对象。

name 不为 None，它将使用*Task.set_name()* 来设为任务的名称。

可选的 *context* 参数允许指定自定义的*contextvars.Context* 供 *coro* 运行。当未提供 *context* 时将创建当前上下文的副本。

该任务会在*get_running_loop()* 返回的循环中执行，如果当前线程没有在运行的循环则会引发*RuntimeError*。

备注

asyncio.TaskGroup.create_task() 是一个平衡了结构化并发的新选择；它允许等待一组相关任务并具有极强的安全保证。

重要

保存一个指向此函数的结果的引用，以避免任务在执行过程中消失。事件循环将只保留对任务的弱引用。未在其他地方被引用的任务可能在任何时候被作为垃圾回收，即使是在它被完成之前。如果需要可靠的“发射后不用管”后台任务，请将它们放到一个多项集中:

```
background_tasks = set()

for i in range(10):
    task = asyncio.create_task(some_coro(param=i))

    # 将任务加入集合。 这将创建一个强引用。
    background_tasks.add(task)

    # 为避免永远保留对已结束任务的引用，
    # 让每个任务在完成后将对自己的引用
    # 移出集合:
    task.add_done_callback(background_tasks.discard)
```

Added in version 3.7.

在 3.8 版本发生变更: 增加了 *name* 形参。

在 3.11 版本发生变更: 增加了 *context* 形参。

任务取消

任务可以便捷和安全地取消。当任务被取消时，*asyncio.CancelledError* 将在遇到机会时在任务中被引发。

推荐协程使用 try/finally 代码块来可靠地执行清理逻辑。对于*asyncio.CancelledError* 被显式捕获的情况，它通常应当在清理完成时被传播。*asyncio.CancelledError* 会直接子类化*BaseException* 因此大多数代码都不需要关心这一点。

启用结构化并发的 asyncio 组件，如*asyncio.TaskGroup* 和*asyncio.timeout()*，在内部是使用撤销操作来实现的因而在协程屏蔽了*asyncio.CancelledError* 时可能无法正常工作。类似地，用户代码通常也不应调用*uncancel*。但是，在确实想要屏蔽*asyncio.CancelledError* 的情况下，则还有必要调用 uncancel() 来完全移除撤销状态。

任务组

任务组合并了一套用于等待分组中所有任务完成的方便可靠方式的任务创建 API。

class asyncio.**TaskGroup**

持有一个任务分组的 异步上下文管理器。可以使用*create_task()* 将任务添加到分组中。当该上下文管理器退出时所有任务都将被等待。

Added in version 3.11.

create_task(*coro*, ***, *name=None*, *context=None*)

在该任务组中创建一个任务。签名与*asyncio.create_task()* 的签名相匹配。如果该任务组未激活（例如尚未进入、已经结束或在关闭过程中），我们将关闭所给出的 coro。

在 3.13 版本发生变更: 如果任务组未激活则关闭所给出的协程。

示例:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(some_coro(...))
        task2 = tg.create_task(another_coro(...))
    print(f"Both tasks have completed now: {task1.result()}, {task2.result()}")
```

async with 语句将等待分组中的所有任务结束。在等待期间，仍可将新任务添加到分组中 (例如，通过将 tg 传入某个协程并在该协程中调用 tg.create_task())。一旦最后的任务完成并退出 async with 代码块，将无法再向分组添加新任务。

当首次有任何属于分组的任务因*asyncio.CancelledError* 以外的异常而失败时，分组中的剩余任务将被取消。在此之后将无法添加更多任务到该分组中。在这种情况下，如果 async with 语句体仍然为激活状态（即 __aexit__() 尚未被调用），则直接包含 async with 语句的任务也会被取消。结果*asyncio.CancelledError* 将中断一个 await，但它将不会跳出包含的 async with 语句。

一旦所有任务被完成，如果有任何任务因*asyncio.CancelledError* 以外的异常而失败，这些异常会被组合在*ExceptionGroup* 或*BaseExceptionGroup* 中（选择其中较适合的一个；参见其文档）并将随后引发。

两个基础异常会被特别对待：如果有任何任务因*KeyboardInterrupt* 或*SystemExit* 而失败，任务分组仍然会取消剩余的任务并等待它们，但随后初始*KeyboardInterrupt* 或*SystemExit* 而不是*ExceptionGroup* 或*BaseExceptionGroup* 会被重新引发。

如果 async with 语句体因异常而退出（这样将调用 __aexit__() 并附带一个异常），此种情况会与有任务失败时一样对待：剩余任务将被取消然后被等待，而非取消类异常会被加入到一个异常分组并被引发。传入到 __aexit__() 的异常，除了*asyncio.CancelledError* 以外，也都会被包括在该异常分组中。同样的特殊对待也适用于上一段所说的*KeyboardInterrupt* 和*SystemExit*。

对于任务组应当注意不要将用于“唤醒”其 __aexit__() 的内部取消请求与其他地方对其运行的任务提出的取消请求相混淆。具体来说，当一个任务组在语法上嵌套于另一个任务组中，而两个任务组的某个子任务同时发生异常时，内层的任务组将处理其异常，然后外层的任务组将收到另一个取消请求并处理它自己的异常。

对于任务组在外部被取消同时必须引发*ExceptionGroup* 的情况，它将调用父任务的*cancel()* 方法。这样可以确保*asyncio.CancelledError* 会在下一次 await 时被引发，因此取消操作不会丢失。

任务组将保留*asyncio.Task.cancelling()* 所报告的取消次数。

在 3.13 版本发生变更: 改进了同时处理内部和外部取消操作以及正确保留取消计数的功能。

终结一个任务组

虽然标准库没有对终结任务组的原生支持，但可通过向任务组添加一个引发异常的任务并无视被引发的异常来达成终结效果：

```
import asyncio
from asyncio import TaskGroup

class TerminateTaskGroup(Exception):
    """Exception raised to terminate a task group."""

async def force_terminate_task_group():
    """Used to force termination of a task group."""
    raise TerminateTaskGroup()

async def job(task_id, sleep_time):
    print(f'Task {task_id}: start')
    await asyncio.sleep(sleep_time)
    print(f'Task {task_id}: done')

async def main():
    try:
        async with TaskGroup() as group:
            # 创建一些任务
            group.create_task(job(1, 0.5))
            group.create_task(job(2, 1.5))
            # 休眠 1 秒
            await asyncio.sleep(1)
            # 添加一个引发异常的任务以强制终结分组
            group.create_task(force_terminate_task_group())
    except* TerminateTaskGroup:
        pass

asyncio.run(main())
```

期待的输出:

```
Task 1: start
Task 2: start
Task 1: done
```

休眠

async asyncio.**sleep**(*delay*, *result=None*)

阻塞 *delay* 指定的秒数。

如果指定了 *result*，则当协程完成时将其返回给调用者。

`sleep()` 总是会挂起当前任务，以允许其他任务运行。

将 delay 设为 0 将提供一个经优化的路径以允许其他任务运行。这可供长期间运行的函数使用以避免在函数调用的全过程中阻塞事件循环。

以下协程示例运行 5 秒，每秒显示一次当前日期:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

在 3.10 版本发生变更: 移除了 *loop* 形参。

在 3.13 版本发生变更: 如果 *delay* 为 *nan* 则会引发 *ValueError*。

并发运行任务

awaitable asyncio.**gather**(**aws*, *return_exceptions=False*)

并发运行 *aws* 序列中的可等待对象。

如果 *aws* 中的某个可等待对象为协程，它将自动被作为一个任务调度。

如果所有可等待对象都成功完成，结果将是一个由所有返回值聚合而成的列表。结果值的顺序与 *aws* 中可等待对象的顺序一致。

如果 *return_exceptions* 为 False (默认)，所引发的首个异常会立即传播给等待 gather() 的任务。*aws* 序列中的其他可等待对象 **不会被取消**并将继续运行。

如果 *return_exceptions* 为 True，异常会和成功的结果一样处理，并聚合至结果列表。

如果 gather() 被取消，所有被提交 (尚未完成) 的可等待对象也会 被取消。

如果 *aws* 序列中的任一 Task 或 Future 对象 被取消，它将被当作引发了 *CancelledError* 一样处理 -- 在此情况下 gather() 调用 **不会**被取消。这是为了防止一个已提交的 Task/Future 被取消导致其他 Tasks/Future 也被取消。

> **备注**
>
> 一个创建然后并发地运行任务等待它们完成的新选择是 *asyncio.TaskGroup*。*TaskGroup* 提供了针对调度嵌套子任务的比 *gather* 更强的安全保证：如果一个任务（或子任务，即由一个任务调度的任务）引发了异常，*TaskGroup* 将取消剩余的已排期任务）。

示例:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}), currently i={i}...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # 将三个调用 *并发地* 加入计划任务:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# 预期的输出:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
```

(续下页)

(接上页)

```
#     Task C: Compute factorial(4), currently i=3...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4), currently i=4...
#     Task C: factorial(4) = 24
#     [2, 6, 24]
```

备注

如果 *return_exceptions* 为假值，则在 gather() 被标记为完成后取消它将不会取消任何已提交的可等待对象。例如，在将一个异常传播给调用者之后，gather 可被标记为已完成，因此，在从 gather 捕获一个（由可等待对象所引发的）异常之后调用 `gather.cancel()` 将不会取消任何其他可等待对象。

在 3.7 版本发生变更: 如果 *gather* 本身被取消，则无论 *return_exceptions* 取值为何，消息都会被传播。

在 3.10 版本发生变更: 移除了 *loop* 形参。

自 3.10 版本弃用: 如果未提供位置参数或者并非所有位置参数均为 Future 类对象并且没有正在运行的事件循环则会发出弃用警告。

主动任务工厂

asyncio.**eager_task_factory**(*loop*, *coro*, *, *name=None*, *context=None*)

用于主动任务执行的任务工厂

当使用这个工厂函数时 (通过*loop.set_task_factory(asyncio.eager_task_factory)*)，协程将在*Task* 构造期间同步地开始执行。任务仅会在它们阻塞时被加入事件循环上的计划任务。这可以达成性能提升因为对同步完成的协程来说可以避免循环调度的开销。

此特性会带来好处的一个常见例子是应用了缓存或记忆功能以便在可能的情况避免实际 I/O 的协程。

备注

协程是立即执行是一项语言改变。如果协程返回或引发异常，其任务将不会被加入事件循环上的计划任务。如果协程执行发生阻塞，其任务将被加入事件循环上的计划任务。这项改变可能会向现有应用程序引入行为变化。例如，应用程序的任务执行顺序可能会发生改变。

Added in version 3.12.

asyncio.**create_eager_task_factory**(*custom_task_constructor*)

创建一个主动型任务工厂，类似于*eager_task_factory()*，在创建新任务时使用所提供的 *custom_task_constructor* 而不是默认的*Task*。

custom_task_constructor 必须是一个 可调用对象，其签名与*Task.__init__* 的签名相匹配。该可调用对象必须返回一个兼容*asyncio.Task* 的对象。

此函数返回一个 可调用对象，将通过*loop.set_task_factory(factory)*) 被用作一个事件循环的任务工厂。

Added in version 3.12.

屏蔽取消操作

awaitable asyncio.**shield**(*aw*)

保护一个可等待对象 防止其被取消。

如果 *aw* 是一个协程，它将自动被作为任务调度。

以下语句:

```
task = asyncio.create_task(something())
res = await shield(task)
```

相当于:

```
res = await something()
```

不同之处在于如果包含它的协程被取消，在 `something()` 中运行的任务不会被取消。从 `something()` 的角度看来，取消操作并没有发生。然而其调用者已被取消，因此"await" 表达式仍然会引发*CancelledError*。

如果通过其他方式取消 `something()` (例如在其内部操作) 则 `shield()` 也会取消。

如果希望完全忽略取消操作 (不推荐) 则 `shield()` 函数需要配合一个 try/except 代码段，如下所示:

```
task = asyncio.create_task(something())
try:
    res = await shield(task)
except CancelledError:
    res = None
```

重要

保存一个传给此函数的任务的引用，以避免任务在执行过程中消失。事件循环将只保留对任务的弱引用。未在其他地方被引用的任务可能在任何时候被作为垃圾回收，即使是在它被完成之前。

在 3.10 版本发生变更: 移除了 *loop* 形参。

自 3.10 版本弃用: 如果 *aw* 不是 Future 类对象并且没有正在运行的事件循环则会发出弃用警告。

超时

asyncio.**timeout**(*delay*)

返回一个可被用于限制等待某个操作所耗费时间的 异步上下文管理器。

delay 可以为 `None`，或是一个表示等待秒数的浮点数/整数。如果 *delay* 为 `None`，将不会应用时间限制；如果当创建上下文管理器时无法确定延时则此设置将很适用。

在两种情况下，该上下文管理器都可以在创建之后使用*Timeout.reschedule()* 来重新安排计划。

示例:

```
async def main():
    async with asyncio.timeout(10):
        await long_running_task()
```

如果 `long_running_task` 耗费 10 秒以上完成，该上下文管理器将取消当前任务并在内部处理所引发的*asyncio.CancelledError*，将其转化为可被捕获和处理的*TimeoutError*。

备注

asyncio.timeout() 上下文管理器负责将*asyncio.CancelledError* 转化为*TimeoutError*，这意味着*TimeoutError* 只能在该上下文管理器 之外被捕获。

捕获*TimeoutError* 的示例:

```
async def main():
    try:
        async with asyncio.timeout(10):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

asyncio.timeout() 所产生的上下文管理器可以被重新调整到不同的终止点并执行检查。

class asyncio.**Timeout**(*when*)

一个用于撤销已过期协程的 异步上下文管理器。

when 应当是一个指明上下文将要过期的绝对时间，由事件循环的时钟来计时。

- 如果 when 为 None，则超时将永远不会被触发。
- 如果 when < loop.time()，则超时将在事件循环的下一次迭代中被触发。

when() → *float* | *None*

返回当前终止点，或者如果未设置当前终止点则返回 None。

reschedule(*when:* float | None)

重新安排超时。

expired() → *bool*

返回上下文管理器是否已超出时限（过期）。

示例:

```
async def main():
    try:
        # 当开始时我们并不知道超时值，所以我们传入 ``None``。
        async with asyncio.timeout(None) as cm:
            # 现在我们知道超时值了，所以我们将它重新加入计划任务。
            new_deadline = get_running_loop().time() + 10
            cm.reschedule(new_deadline)

            await long_running_task()
    except TimeoutError:
        pass

    if cm.expired():
        print("Looks like we haven't finished on time.")
```

超时上下文管理器可以被安全地嵌套。

Added in version 3.11.

asyncio.**timeout_at**(*when*)

类似于*asyncio.timeout()*，不同之处在于 *when* 是停止等待的绝对时间，或者为 None。

示例:

```
async def main():
    loop = get_running_loop()
    deadline = loop.time() + 20
    try:
        async with asyncio.timeout_at(deadline):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")
```

(续下页)

(接上页)

```
    print("This statement will run regardless.")
```

Added in version 3.11.

async asyncio.**wait_for**(*aw*, *timeout*)

等待 *aw* 可等待对象 完成，指定 timeout 秒数后超时。

如果 *aw* 是一个协程，它将自动被作为任务调度。

timeout 可以为 `None`，也可以为 float 或 int 型数值表示的等待秒数。如果 *timeout* 为 `None`，则等待直到完成。

如果发生超时，将取消任务并引发*TimeoutError*。

要避免任务取消，可以加上*shield()*。

此函数将等待直到 Future 确实被取消，所以总等待时间可能超过 *timeout*。如果在取消期间发生了异常，异常将会被传播。

如果等待被取消，则 *aw* 指定的对象也会被取消。

示例:

```
async def eternity():
    # 休眠一小时
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # 等待至多 1 秒
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except TimeoutError:
        print('timeout!')

asyncio.run(main())

# 预期的输出:
#
#     timeout!
```

在 3.7 版本发生变更: 当 *aw* 由于超时被取消时，`wait_for` 会等待 *aw* 被取消。在之前版本中，它会立即引发*TimeoutError*。

在 3.10 版本发生变更: 移除了 *loop* 形参。

在 3.11 版本发生变更: 引发*TimeoutError* 而不是*asyncio.TimeoutError*。

简单等待

async asyncio.**wait**(*aws*, ***, *timeout=None*, *return_when=ALL_COMPLETED*)

并发地运行 *aws* 可迭代对象中的*Future* 和*Task* 实例并进入阻塞状态直到满足 *return_when* 所指定的条件。

aws 可迭代对象必须不为空。

返回两个 Task/Future 集合: `(done, pending)`。

用法:

```
done, pending = await asyncio.wait(aws)
```

如指定 *timeout* (float 或 int 类型) 则它将被用于控制返回之前等待的最长秒数。

请注意此函数不会引发*TimeoutError*。当超时发生时尚未完成的 Future 或 Task 会在设定的秒数后被直接返回。

return_when 指定此函数应在何时返回。它必须为以下常数之一:

常量	描述
asyncio.**FIRST_COMPLETED**	函数将在任意可等待对象结束或取消时返回。
asyncio.**FIRST_EXCEPTION**	该函数将在任何 future 对象通过引发异常而结束时返回。如果没有任何 future 对象引发引发那么它将等价于*ALL_COMPLETED*。
asyncio.**ALL_COMPLETED**	函数将在所有可等待对象结束或取消时返回。

与*wait_for()* 不同，wait() 在超时发生时不会取消可等待对象。

在 3.10 版本发生变更: 移除了 *loop* 形参。

在 3.11 版本发生变更: 直接向 wait() 传入协程对象的方式已被弃用。

在 3.12 版本发生变更: 增加了对产生任务的生成器的支持。

asyncio.**as_completed**(*aws*, *, *timeout=None*)

并发地运行 *aws* 可迭代对象中的可等待对象。返回的对象可以被迭代以获取可等待对象结束时的结果。

由 as_completed() 返回的对象可作为*asynchronous iterator* 或普通的*iterator* 被迭代。当使用异步迭代时，原来提供的可等待对象如果为 Task 或 Future 对象则会被产出。这样可以更容易地将之前加入计划的任务与其结果进行对应。例如:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

async for earliest_connect in as_completed(tasks):
    # earliest_connect 已完成。 要获取结果可通过等待它
    # 或是调用 calling earliest_connect.result()
    reader, writer = await earliest_connect

    if earliest_connect is ipv6_connect:
        print("IPv6 connection established.")
    else:
        print("IPv4 connection established.")
```

在异步迭代期间，将为不属于 Task 或 Future 对象的可等待对象产出隐式创建的任务。

当被用作普通的迭代器时，每次迭代将产出一个返回结果的新协程或是引发下一个完成的等待对象对应的异常。此模式将与 Python 3.13 之前的版本保持兼容:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

for next_connect in as_completed(tasks):
    # next_connect 不是原始任务对象之一。
    # 它必须被等待以获取结果值或是引发
    # 接下来要结束的可等待对象的异常。
    reader, writer = await next_connect
```

如果在所有可等待对象完成之前达到超时限制则会引发*TimeoutError*。这将在异步迭代期间由 `async for` 循环引发或是在普通迭代期间由所产出的协程引发。

在 3.10 版本发生变更: 移除了 *loop* 形参。

自 3.10 版本弃用: 如果 *aws* 可迭代对象中的可等待对象不全为 Future 类对象并且没有正在运行的事件循环则会发出弃用警告。

在 3.12 版本发生变更: 增加了对产生任务的生成器的支持。

在 3.13 版本发生变更: 该结果现在可被用作*asynchronous iterator* 或是普通的*iterator* (在之前它只是普通的迭代器)。

在线程中运行

async asyncio.**to_thread**(*func*, */*, **args*, ***kwargs*)

在不同的线程中异步地运行函数 *func*。

向此函数提供的任何 *args 和 **kwargs 会被直接传给 *func*。并且，当前*contextvars.Context* 会被传播，允许在不同的线程中访问来自事件循环的上下文变量。

返回一个可被等待以获取 *func* 的最终结果的协程。

这个协程函数主要是用于执行在其他情况下会阻塞事件循环的 IO 密集型函数/方法。例如:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # 请注意 time.sleep() 可被替换为任意一种
    # 阻塞式 IO 密集型操作，例如文件操作。
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# 预期的输出:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

在任何协程中直接调用 `blocking_io()` 将会在调用期间阻塞事件循环，导致额外的 1 秒运行时间。但是，通过改用 `asyncio.to_thread()`，我们可以在单独的线程中运行它从而不会阻塞事件循环。

备注

由于*GIL* 的存在，`asyncio.to_thread()` 通常只能被用来将 IO 密集型函数变为非阻塞的。但是，对于会释放 GIL 的扩展模块或无此限制的替代性 Python 实现来说，`asyncio.to_thread()` 也可被用于 CPU 密集型函数。

Added in version 3.9.

跨线程调度

asyncio.**run_coroutine_threadsafe**(*coro*, *loop*)

向指定事件循环提交一个协程。(线程安全)

返回一个*concurrent.futures.Future* 以等待来自其他 OS 线程的结果。

此函数应该从另一个 OS 线程中调用，而非事件循环运行所在线程。示例:

```
# 创建一个协程
coro = asyncio.sleep(1, result=3)

# 将协程提交到给定的循环
future = asyncio.run_coroutine_threadsafe(coro, loop)

# 等待结果并可选择设置超时参数
assert future.result(timeout) == 3
```

如果在协程内产生了异常，将会通知返回的 Future 对象。它也可被用来取消事件循环中的任务:

```
try:
    result = future.result(timeout)
except TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

参见*concurrency and multithreading* 部分的文档。

不同于其他 asyncio 函数，此函数要求显式地传入 *loop* 参数。

Added in version 3.5.1.

内省

asyncio.**current_task**(*loop=None*)

返回当前运行的*Task* 实例，如果没有正在运行的任务则返回 None。

如果 *loop* 为 None 则会使用*get_running_loop()* 获取当前事件循环。

Added in version 3.7.

asyncio.**all_tasks**(*loop=None*)

返回事件循环所运行的未完成的*Task* 对象的集合。

如果 *loop* 为 None，则会使用*get_running_loop()* 获取当前事件循环。

Added in version 3.7.

asyncio.**iscoroutine**(*obj*)

如果 *obj* 是一个协程对象则返回 True。

Added in version 3.4.

Task 对象

class asyncio.**Task**(*coro*, ***, *loop=None*, *name=None*, *context=None*, *eager_start=False*)

一个与*Future 类似* 的对象，可运行 Python 协程。非线程安全。

Task 对象被用来在事件循环中运行协程。如果一个协程在等待一个 Future 对象，Task 对象会挂起该协程的执行并等待该 Future 对象完成。当该 Future 对象 完成，被打包的协程将恢复执行。

事件循环使用协同日程调度: 一个事件循环每次运行一个 Task 对象。而一个 Task 对象会等待一个 Future 对象完成，该事件循环会运行其他 Task、回调或执行 IO 操作。

使用高层级的*asyncio.create_task()* 函数来创建 Task 对象，也可用低层级的*loop.create_task()* 或*ensure_future()* 函数。不建议手动实例化 Task 对象。

要取消一个正在运行的 Task 对象可使用*cancel()* 方法。调用此方法将使该 Task 对象抛出一个*CancelledError* 异常给打包的协程。如果取消期间一个协程正在对 Future 对象执行 await，该 Future 对象也将被取消。

cancelled() 可被用来检测 Task 对象是否被取消。如果打包的协程没有抑制*CancelledError* 异常并且确实被取消，该方法将返回 `True`。

asyncio.Task 从*Future* 继承了其除*Future.set_result()* 和*Future.set_exception()* 以外的所有 API。

可选的仅限关键字参数 *context* 允许指定自定义的*contextvars.Context* 供 *coro* 运行。如果未提供 *context*，Task 将拷贝当前上下文并随后在拷贝的上下文中运行其协程。

可选的仅限关键字参数 *eager_start* 允许在任务创建时主动开始*asyncio.Task* 的执行。如果设为 `True` 并且事件循环正在运行，任务将立即开始执行协程，直到该协程第一次阻塞。如果协程未发生阻塞即返回或引发异常，任务将主动结束并将跳过向事件循环添加计划任务。

在 3.7 版本发生变更: 加入对*contextvars* 模块的支持。

在 3.8 版本发生变更: 增加了 *name* 形参。

自 3.10 版本弃用: 如果未指定 *loop* 并且没有正在运行的事件循环则会发出弃用警告。

在 3.11 版本发生变更: 增加了 *context* 形参。

在 3.12 版本发生变更: 增加了 *eager_start* 形参。

done()

如果 Task 对象 *已完成*则返回 `True`。

当 Task 所封包的协程返回一个值、引发一个异常或 Task 本身被取消时，则会被认为 *已完成*。

result()

返回 Task 的结果。

如果 Task 对象 *已完成*，其封包的协程的结果会被返回 (或者当协程引发异常时，该异常会被重新引发。)

如果 Task 对象 *被取消*，此方法会引发一个*CancelledError* 异常。

如果 Task 对象的结果还不可用，此方法会引发一个*InvalidStateError* 异常。

exception()

返回 Task 对象的异常。

如果所封包的协程引发了一个异常，该异常将被返回。如果所封包的协程正常返回则该方法将返回 `None`。

如果 Task 对象 *被取消*，此方法会引发一个*CancelledError* 异常。

如果 Task 对象尚未 *完成*，此方法将引发一个*InvalidStateError* 异常。

add_done_callback(*callback*, *, *context=None*)

添加一个回调，将在 Task 对象 *完成*时被运行。

此方法应该仅在低层级的基于回调的代码中使用。

要了解更多细节请查看*Future.add_done_callback()* 的文档。

remove_done_callback(*callback*)

从回调列表中移除 *callback* 。

此方法应该仅在低层级的基于回调的代码中使用。

要了解更多细节请查看*Future.remove_done_callback()* 的文档。

get_stack(*, *limit=None*)

返回此 Task 对象的栈框架列表。

如果所封包的协程未完成，这将返回其挂起所在的栈。如果协程已成功完成或被取消，这将返回一个空列表。如果协程被一个异常终止，这将返回回溯框架列表。

框架总是从按从旧到新排序。

每个被挂起的协程只返回一个栈框架。

可选的 *limit* 参数指定返回框架的数量上限；默认返回所有框架。返回列表的顺序要看是返回一个栈还是一个回溯：栈返回最新的框架，回溯返回最旧的框架。(这与 traceback 模块的行为保持一致。)

print_stack(*, *limit=None*, *file=None*)

打印此 Task 对象的栈或回溯。

此方法产生的输出类似于 traceback 模块通过*get_stack()* 所获取的框架。

limit 参数会直接传递给*get_stack()*。

file 参数是输出所写入的 I/O 流；在默认情况下输出会写入到*sys.stdout*。

get_coro()

返回由*Task* 包装的协程对象。

> **备注**
>
> 这对于已经主动完成的任务将返回 None。参见主动任务工厂。

Added in version 3.8.

在 3.12 版本发生变更: 新增加的主动任务执行意味着结果可能为 None。

get_context()

返回关联到该任务的*contextvars.Context* 对象。

Added in version 3.12.

get_name()

返回 Task 的名称。

如果没有一个 Task 名称被显式地赋值，默认的 asyncio Task 实现会在实例化期间生成一个默认名称。

Added in version 3.8.

set_name(*value*)

设置 Task 的名称。

value 参数可以为任意对象，它随后会被转换为字符串。

在默认的 Task 实现中，名称将在任务对象的*repr()* 输出中可见。

Added in version 3.8.

cancel(*msg=None*)

请求取消 Task 对象。

如果该 Task 已经 完成或 取消，则返回 False，否则，将返回 True。

该方法将安排在下一轮事件循环中将*CancelledError* 异常抛出给被封包的协程。

协程随后将有机会进行清理甚至通过 try ... ... except CancelledError ... finally 代码块抑制异常来拒绝请求。因此，不同于*Future.cancel()*，*Task.cancel()* 不保证 Task 会被取消，虽然完全抑制撤销并不常见也很不建议这样做。但是如果协程决定要抑制撤销，那么它需要额外调用*Task.uncancel()* 来捕获异常。

在 3.9 版本发生变更: 增加了 *msg* 形参。

在 3.11 版本发生变更: `msg` 形参将从被取消的任务传播到其等待方。 以下示例演示了协程是如何侦听取消请求的:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # 等待 1 小时
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # 创建一个 "cancel_me" 任务
    task = asyncio.create_task(cancel_me())

    # 等待 1 秒
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# 预期的输出:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()

如果 Task 对象 被取消则返回 `True`。

当使用*cancel()* 发出取消请求时 Task 会被取消，其封包的协程将传播被抛入的*CancelledError* 异常。

uncancel()

递减对此任务的取消请求计数。

返回剩余的取消请求数量。

请注意一旦被取消的任务执行完成，继续调用*uncancel()* 将是低效的。

Added in version 3.11.

此方法是供 asyncio 内部使用而不应被最终用户代码所使用。特别地，在一个 Task 成功地保持未取消状态的时候使用，这可以允许结构化的并发元素如任务组 和*asyncio.timeout()* 继续运行，将取消操作隔离在相应的结构化代码块中。例如:

```
async def make_request_with_timeout():
    try:
        async with asyncio.timeout(1):
            # 受超时影响的结构块:
            await make_request()
            await make_another_request()
```

(续下页)

(接上页)

```
    except TimeoutError:
        log("There was a timeout")
    # 不受超时影响的外层代码:
    await unrelated_code()
```

带有 `make_request()` 和 `make_another_request()` 的代码块可能因超时而被取消，而 `unrelated_code()` 应当在超时的情况下继续运行。这是通过*uncancel()* 实现的。*TaskGroup* 上下文管理器也会以类似的方式来使用*uncancel()*。

如果最终用户代码出于某种原因通过捕获*CancelledError* 抑制撤销操作，那么它需要调用此方法来移除撤销状态。

当该方法将取消计数递减至零，该方法会检查之前的*cancel()* 调用是否已安排将*CancelledError* 抛出到任务中。如果尚未抛出，则该安排将被撤销（通过重置内部的 `_must_cancel` 旗标）。

在 3.13 版本发生变更: 更改为在到达零值时撤回待处理的取消请求。

cancelling()

返回对此 Task 的挂起请求次数，即对*cancel()* 的调用次数减去*uncancel()* 的调用次数。

请注意如果该数字大于零但相应 Task 仍在执行，*cancelled()* 仍将返回 `False`。这是因此该数字可通过调用*uncancel()* 来减少，这会导致任务在取消请求降到零时尚未被取消。

此方法是供 asyncio 内部使用而不应被最终用户代码所使用。请参阅*uncancel()* 了解详情。

Added in version 3.11.

19.1.3 流

源码: Lib/asyncio/streams.py

流是用于处理网络连接的支持 async/await 的高层级原语。流允许发送和接收数据，而不需要使用回调或低级协议和传输。

下面是一个使用 asyncio streams 编写的 TCP echo 客户端示例:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

参见下面的*Examples* 部分。

Stream 函数

下面的高级 asyncio 函数可以用来创建和处理流:

async asyncio.**open_connection**(*host=None*, *port=None*, ***, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)

建立网络连接并返回一对 `(reader, writer)` 对象。

返回的 *reader* 和 *writer* 对象是`StreamReader` 和`StreamWriter` 类的实例。

limit 确定返回的`StreamReader` 实例使用的缓冲区大小限制。默认情况下，*limit* 设置为 64 KiB 。

其余的参数直接传递到`loop.create_connection()` 。

> **备注**
>
> *sock* 参数可将套接字的所有权转给所创建的`StreamWriter`。要关闭该套接字，请调用其`close()` 方法。

在 3.7 版本发生变更: 添加了 *ssl_handshake_timeout* 形参。

在 3.8 版本发生变更: 增加了 *happy_eyeballs_delay* 和 *interleave* 形参。

在 3.10 版本发生变更: 移除了 *loop* 形参。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

async asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, ***, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

启动套接字服务。

当一个新的客户端连接被建立时，回调函数 *client_connected_cb* 会被调用。该函数会接收到一对参数 `(reader, writer)` ，reader 是类`StreamReader` 的实例，而 writer 是类`StreamWriter` 的实例。

client_connected_cb 即可以是普通的可调用对象也可以是一个协程函数; 如果它是一个协程函数，它将自动作为`Task` 被调度。

limit 确定返回的`StreamReader` 实例使用的缓冲区大小限制。默认情况下，*limit* 设置为 64 KiB 。

余下的参数将会直接传递给`loop.create_server()`.

> **备注**
>
> *sock* 参数可将套接字的所有权转给所创建的服务器。要关闭该套接字，请调用服务器的`close()` 方法。

在 3.7 版本发生变更: 增加了 *ssl_handshake_timeout* 和 *start_serving* 形参。

在 3.10 版本发生变更: 移除了 *loop* 形参。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

在 3.13 版本发生变更: 增加了 *keep_alive* 形参。

Unix 套接字

async asyncio.**open_unix_connection**(*path=None*, ***, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

建立一个 Unix 套接字连接并返回 `(reader, writer)` 这对返回值。

与*open_connection()* 相似，但是是在 Unix 套接字上的操作。

请看文档*loop.create_unix_connection()*.

备注

sock 参数可将套接字的所有权转给所创建的*StreamWriter*。要关闭该套接字，请调用其*close()* 方法。

Availability: Unix.

在 3.7 版本发生变更: 增加了 *ssl_handshake_timeout* 形参。现在 *path* 形参可以是一个*path-like object*

在 3.10 版本发生变更: 移除了 *loop* 形参。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

async asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, ***, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

启动一个 Unix 套接字服务。

与*start_server()* 相似，但是是在 Unix 套接字上的操作。

请看文档*loop.create_unix_server()*.

备注

sock 参数可将套接字的所有权转给所创建的服务器。要关闭该套接字，请调用服务器的*close()* 方法。

Availability: Unix.

在 3.7 版本发生变更: 增加了 *ssl_handshake_timeout* 和 *start_serving* 形参。现在 *path* 形参可以是一个*path-like object*。

在 3.10 版本发生变更: 移除了 *loop* 形参。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

StreamReader

class asyncio.**StreamReader**

代表一个提供从 IO 流读取数据的 API 的读取器。作为一个*asynchronous iterable*，该对象支持 `async for` 语句。

不推荐直接实例化 *StreamReader* 对象，建议使用*open_connection()* 和*start_server()* 来获取 *StreamReader* 实例。

feed_eof()

识别 EOF。

async read(*n=-1*)

从流读取至多 *n* 个字节。

如果未提供 *n* 或是设为 `-1`，则一直读取到 EOF，然后返回所读取的全部*bytes*。如果收到 EOF 并且内部缓冲区为空，则返回一个空 `bytes` 对象。object.

如果 *n* 为 `0`，则立即返回一个空 `bytes` 对象。

如果 *n* 为正值，则一旦内部缓冲区有至少 1 个字节可用就返回至多 *n* 个可用的 `bytes`。如果在读取任何字节之前收到 EOF，则返回一个空 `bytes` 对象。

async readline()

读取一行，其中“行”指的是以 `\n` 结尾的字节序列。

如果读到 EOF 而没有找到 `\n`，该方法返回部分读取的数据。

如果读到 EOF，且内部缓冲区为空，则返回一个空的 `bytes` 对象。

async readexactly(*n*)

精确读取 *n* 个 bytes，不会超过也不能少于。

如果在读取完 *n* 个 byte 之前读取到 EOF，则会引发*IncompleteReadError* 异常。使用*IncompleteReadError.partial* 属性来获取到达流结束之前读取的 bytes 字符串。

async readuntil(*separator=b'\n'*)

从流中读取数据直至遇到 *separator*

成功后，数据和指定的 separator 将从内部缓冲区中删除 (或者说被消费掉)。返回的数据将包括在末尾的指定 separator。

如果读取的数据量超过了配置的流限制，将引发*LimitOverrunError* 异常，数据将留在内部缓冲区中并可以再次读取。

如果在找到完整的 separator 之前到达 EOF，则会引发*IncompleteReadError* 异常，并重置内部缓冲区。*IncompleteReadError.partial* 属性可能包含指定 separator 的一部分。

separator 也可以是由分隔符组成的元组。在这种情况下返回值将为以任意分隔符为前缀的最短可能值。对于*LimitOverrunError* 来说，分隔符的最短可能值将被认为是匹配的值。

Added in version 3.5.2.

在 3.13 版本发生变更: 现在 *separator* 形参可以是由分隔符组成的*tuple*。

at_eof()

如果缓冲区为空并且*feed_eof()* 被调用，则返回 `True` 。

StreamWriter

class asyncio.**StreamWriter**

这个类表示一个写入器对象，该对象提供 api 以便于写数据至 IO 流中。

不建议直接实例化 *StreamWriter*；而应改用*open_connection()* 和*start_server()*。

write(*data*)

此方法会尝试立即将 *data* 写入到下层的套接字。如果写入失败，数据会被排入内部写缓冲队列直到可以被发送。

此方法应当与 `drain()` 方法一起使用:

```
stream.write(data)
await stream.drain()
```

writelines(*data*)

此方法会立即尝试将一个字节串列表（或任何可迭代对象）写入到下层的套接字。如果写入失败，数据会被排入内部写缓冲队列直到可以被发送。

此方法应当与 `drain()` 方法一起使用:

```
stream.writelines(lines)
await stream.drain()
```

close()

此方法会关闭流以及下层的套接字。

此方法应当于 `wait_closed()` 方法一起使用，但这并非强制要求:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

如果下层的传输支持*write_eof()* 方法则返回 `True`，否则返回 `False`。

write_eof()

在已缓冲的写入数据被刷新后关闭流的写入端。

transport

返回下层的 asyncio 传输。

get_extra_info(*name*, *default=None*)

访问可选的传输信息；详情参见*BaseTransport.get_extra_info()*。

async drain()

等待直到可以适当地恢复写入到流。示例:

```
writer.write(data)
await writer.drain()
```

这是一个与下层的 IO 写缓冲区进行交互的流程控制方法。当缓冲区大小达到最高水位（最大上限）时，*drain()* 会阻塞直到缓冲区大小减少至最低水位以便恢复写入。当没有要等待的数据时，*drain()* 会立即返回。

async start_tls(*sslcontext*, ***, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

将现有基于流的连接升级到 TLS。

参数:

- *sslcontext*： 一个已经配置好的*SSLContext* 实例。
- *server_hostname*： 设置或者覆盖目标服务器证书中相对应的主机名。
- *ssl_handshake_timeout* 是在放弃连接之前要等待 TLS 握手完成的秒数。如为 `None` (默认值) 则使用 `60.0`。
- *ssl_shutdown_timeout* 是在放弃连接之前要等待 SSL 关闭完成的秒数。如为 `None` (默认值) 则使用 `30.0`。

Added in version 3.11.

在 3.12 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

is_closing()

如果流已被关闭或正在被关闭则返回 `True`。

Added in version 3.7.

async wait_closed()

等待直到流被关闭。

应当在*close()* 之后调用以等待直到下层连接被关闭，确保所有数据在退出程序之前已刷新。

Added in version 3.7.

例子

使用流的 TCP 回显客户端

使用*asyncio.open_connection()* 函数的 TCP 回显客户端:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

参见

使用低层级*loop.create_connection()* 方法的*TCP 回显客户端协议* 示例。

使用流的 TCP 回显服务器

TCP 回显服务器使用*asyncio.start_server()* 函数:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

参见

使用*loop.create_server()* 方法的*TCP 回显服务器协议* 示例。

获取 HTTP 标头

查询命令行传入 URL 的 HTTP 标头的简单示例:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()
    await writer.wait_closed()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

用法:

```
python example.py http://example.com/path/page.html
```

或使用 HTTPS:

```
python example.py https://example.com/path/page.html
```

注册一个打开的套接字以等待使用流的数据

使用*open_connection()* 函数实现等待直到套接字接收到数据的协程:

```
import asyncio
import socket

async def wait_for_data():
```

(续下页)

(接上页)

```
    # 获取一个指向当前事件循环的引用
    # 因为我们想要访问低层级的 API。
    loop = asyncio.get_running_loop()

    # 创建一对已接连的套接字。
    rsock, wsock = socket.socketpair()

    # 注册打开的套接字以等待数据。
    reader, writer = await asyncio.open_connection(sock=rsock)

    # 模拟从网络接收数据
    loop.call_soon(wsock.send, 'abc'.encode())

    # 等待数据
    data = await reader.read(100)

    # 获得数据，我们已完成：关闭套接字
    print("Received:", data.decode())
    writer.close()
    await writer.wait_closed()

    # 关闭第二个套接字
    wsock.close()

asyncio.run(wait_for_data())
```

参见

使用低层级协议以及*loop.create_connection()* 方法的*注册一个打开的套接字以等待使用协议的数据* 示例。

使用低层级的*loop.add_reader()* 方法来监视文件描述符的*监视文件描述符以读取事件* 示例。

19.1.4 同步原语

源代码: Lib/asyncio/locks.py

---

asyncio 同步原语被设计为与*threading* 模块的类似，但有两个关键注意事项:

- asyncio 原语不是线程安全的，因此它们不应被用于 OS 线程同步 (而应当使用*threading*)；
- 这些同步原语的方法不接受 *timeout* 参数；请使用*asyncio.wait_for()* 函数来执行带有超时的操作。

asyncio 具有下列基本同步原语:

- *Lock*
- *Event*
- *Condition*
- *Semaphore*
- *BoundedSemaphore*
- *Barrier*

Lock

class asyncio.**Lock**

实现一个用于 asyncio 任务的互斥锁。非线程安全。

asyncio 锁可被用来保证对共享资源的独占访问。

使用 Lock 的推荐方式是通过 `async with` 语句:

```
lock = asyncio.Lock()

# ... 稍后
async with lock:
    # 访问共享状态
```

这等价于:

```
lock = asyncio.Lock()

# ... 稍后
await lock.acquire()
try:
    # 访问共享状态
finally:
    lock.release()
```

在 3.10 版本发生变更: 移除了 *loop* 形参。

async acquire()

获取锁。

此方法会等待直至锁为 *unlocked*，将其设为 *locked* 并返回 `True`。

当有一个以上的协程在*acquire()* 中被阻塞则会等待解锁，最终只有一个协程会被执行。

锁的获取是 公平的: 被执行的协程将是第一个开始等待锁的协程。

release()

释放锁。

当锁为 *locked* 时，将其设为 *unlocked* 并返回。

如果锁为 *unlocked*，则会引发*RuntimeError*。

locked()

如果锁为 *locked* 则返回 `True`。

Event

class asyncio.**Event**

事件对象。该对象不是线程安全的。

asyncio 事件可被用来通知多个 asyncio 任务已经有事件发生。

Event 对象会管理一个内部旗标，可通过*set()* 方法将其设为 *true* 并通过*clear()* 方法将其重设为 *false*。*wait()* 方法会阻塞直至该旗标被设为 *true*。该旗标初始时会被设为 *false*。

在 3.10 版本发生变更: 移除了 *loop* 形参。 示例:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
```

(续下页)

(接上页)

```
    # 创建一个 Event 对象。
    event = asyncio.Event()

    # 产生一个任务等待直到 'event' 被设置。
    waiter_task = asyncio.create_task(waiter(event))

    # 休眠 1 秒钟并设置事件。
    await asyncio.sleep(1)
    event.set()

    # 等待直到 waiter 任务完成。
    await waiter_task

asyncio.run(main())
```

async wait()

等待直至事件被设置。

如果事件已被设置，则立即返回 `True`。否则将阻塞直至另一个任务调用*set()*。

set()

设置事件。

所有等待事件被设置的任务将被立即唤醒。

clear()

清空（取消设置）事件。

通过*wait()* 进行等待的任务现在将会阻塞直至*set()* 方法被再次调用。

is_set()

如果事件已被设置则返回 `True`。

Condition

class asyncio.**Condition**(*lock=None*)

条件对象。该对象不是线程安全的。

asyncio 条件原语可被任务用于等待某个事件发生，然后获取对共享资源的独占访问。

在本质上，Condition 对象合并了*Event* 和*Lock* 的功能。多个 Condition 对象有可能共享一个 Lock，这允许关注于共享资源的特定状态的不同任务实现对共享资源的协同独占访问。

可选的 *lock* 参数必须为*Lock* 对象或 `None`。在后一种情况下会自动创建一个新的 Lock 对象。

在 3.10 版本发生变更: 移除了 *loop* 形参。

使用 Condition 的推荐方式是通过 `async with` 语句:

```
cond = asyncio.Condition()

# ... 稍后
async with cond:
    await cond.wait()
```

这等价于:

```
cond = asyncio.Condition()

# ... 稍后
await cond.acquire()
try:
    await cond.wait()
```

(续下页)

(接上页)

```
finally:
    cond.release()
```

async acquire()

获取下层的锁。

此方法会等待直至下层的锁为 *unlocked*，将其设为 *locked* 并返回 returns `True`。

notify(*n=1*)

唤醒正在等待此条件的 *n* 个任务（默认 1 个）。如果正在等待的任务少于 *n* 个则它们都将被唤醒。tasks are waiting they are all awakened.

锁必须在此方法被调用前被获取并在随后被快速释放。如果通过一个 *unlocked* 锁调用则会引发*RuntimeError*。

locked()

如果下层的锁已被获取则返回 `True`。

notify_all()

唤醒所有正在等待此条件的任务。

此方法的行为类似于*notify()*，但会唤醒所有正在等待的任务。

锁必须在此方法被调用前被获取并在随后被快速释放。如果通过一个 *unlocked* 锁调用则会引发*RuntimeError*。

release()

释放下层的锁。

当在未锁定的锁上唤起时，会引发*RuntimeError*。

async wait()

等待直至收到通知。

当此方法被调用时如果调用方任务未获得锁，则会引发*RuntimeError*。

这个方法会释放下层的锁，然后保持阻塞直到被*notify()* 或*notify_all()* 调用所唤醒。一旦被唤醒，Condition 会重新获取它的锁并且此方法将返回 `True`。

请注意一个任务 *有可能*虚假地从此调用返回，这就是为什么调用者总是应当重新检查状态并准备好再次执行*wait()*。出于此理由，你可能会倾向于改用*wait_for()*。

async wait_for(*predicate*)

等待直到目标值变为 *true*。

目标必须为一个可调用对象，其结果将被解读为一个布尔值。此方法将重复地执行*wait()* 直到目标被求值为 *真*。最终的值将被作为返回值。

Semaphore

class asyncio.**Semaphore**(*value=1*)

信号量对象。该对象不是线程安全的。

信号量会管理一个内部计数器，该计数器会随每次*acquire()* 调用递减并随每次*release()* 调用递增。计数器的值永远不会降到零以下；当*acquire()* 发现其值为零时，它将保持阻塞直到有某个任务调用了*release()*。

可选的 *value* 参数用来为内部计数器赋初始值 (默认值为 `1`)。如果给定的值小于 `0` 则会引发*ValueError*。

在 3.10 版本发生变更: 移除了 *loop* 形参。

使用 Semaphore 的推荐方式是通过 `async with` 语句。:

```
sem = asyncio.Semaphore(10)

# ... 稍后
async with sem:
    # 操作共享的资源
```

这等价于:

```
sem = asyncio.Semaphore(10)

# ... 稍后
await sem.acquire()
try:
    # 操作共享的资源
finally:
    sem.release()
```

async acquire()

获取一个信号量。

如果内部计数器的值大于零，则将其减一并立即返回 `True`。如果其值为零，则会等待直到*release()* 并调用并返回 `True`。

locked()

如果信号量对象无法被立即获取则返回 `True`。

release()

释放一个信号量对象，将内部计数器的值加一。可以唤醒一个正在等待获取信号量对象的任务。

不同于*BoundedSemaphore*，*Semaphore* 允许执行的 `release()` 调用多于 `acquire()` 调用。

BoundedSemaphore

class asyncio.**BoundedSemaphore**(*value=1*)

绑定的信号量对象。该对象不是线程安全的。

BoundedSemaphore 是特殊版本的*Semaphore*，如果在*release()* 中内部计数器值增加到初始 *value* 以上它将引发一个*ValueError*。

在 3.10 版本发生变更: 移除了 *loop* 形参。

Barrier

class asyncio.**Barrier**(*parties*)

屏障对象。该对象不是线程安全的。

屏障是一个允许阻塞直到 *parties* 个任务在其上等待的简单同步原语。任务可以在*wait()* 方法上等待并将被阻塞直到有指定数量的任务在*wait()* 上等待。在那时所有正在等待的任务将同时撤销阻塞。

`async with` 可以被用作在*wait()* 上等待的替代。

屏障可被重复使用任意次数。

示例:

```
async def example_barrier():
    # 包含三部分的屏障
    b = asyncio.Barrier(3)

    # 新建 2 个等待任务
    asyncio.create_task(b.wait())
```

(续下页)

(接上页)

```
    asyncio.create_task(b.wait())

    await asyncio.sleep(0)
    print(b)

    # 第三个 .wait() 调用通过屏障
    await b.wait()
    print(b)
    print("barrier passed")

    await asyncio.sleep(0)
    print(b)

asyncio.run(example_barrier())
```

该示例的结果为:

```
<asyncio.locks.Barrier object at 0x... [filling, waiters:2/3]>
<asyncio.locks.Barrier object at 0x... [draining, waiters:0/3]>
barrier passed
<asyncio.locks.Barrier object at 0x... [filling, waiters:0/3]>
```

Added in version 3.11.

async wait()

穿过屏障。当屏障汇集的所有任务都调用了此函数时，它们将被同时撤销阻塞。

当该屏障中等待或阻塞的任务被取消时，此任务将退出屏障而屏障将保持相同状态。如果屏障的状态为" 正在填充"，则等待的任务数量将减 1。

返回值是一个 0 到 `parties-1` 之间的整数，对于每个任务来说各不相同。这可被用来选择一个任务以执行某些特别的操作。例如:

```
...
async with barrier as position:
    if position == 0:
        # 只有一个任务会打印此内容
        print('End of *draining phase*')
```

如果屏障在有任务在等待时已被破坏或重置则此方法可能会引发*BrokenBarrierError*。如果有任务被取消则它可能会引发*CancelledError*。

async reset()

将屏障返回为默认的空白状态。任何正在其上等待的任务将会接收到*BrokenBarrierError*异常。

如果屏障已被破坏则最好的是让其保持原状并创建一个新的屏障。

async abort()

将屏障置为已破坏状态。这会导致任何已激活或未来对*wait()* 的调用失败并引发*BrokenBarrierError*。例如可以在需要中止某个任务时使用此方法，以避免无限等待任务。

parties

请求穿过该屏障的任务的数量。

n_waiting

当执行填充时正在屏障中等待的任务的数量。

broken

一个布尔值，值为 `True` 表明栅栏为破损态。

exception asyncio.**BrokenBarrierError**

异常类，是*RuntimeError* 异常的子类，在*Barrier* 对象重置时仍有线程阻塞时和对象进入破损态时被引发。

在 3.9 版本发生变更: 使用 `await lock` 或 `yield from lock` 以及/或者 `with` 语句 (`with await lock`, `with (yield from lock)`) 来获取锁的操作已被移除。请改用 `async with lock`。

19.1.5 子进程集

源代码: Lib/asyncio/subprocess.py, Lib/asyncio/base_subprocess.py

---

本节介绍了用于创建和管理子进程的高层级 async/await asyncio API。

下面的例子演示了如何用 asyncio 运行一个 shell 命令并获取其结果:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

将打印:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

由于所有 asyncio 子进程函数都是异步的并且 asyncio 提供了许多工具用来配合这些函数使用，因此并行地执行和监视多个子进程十分容易。要修改上面的例子来同时运行多个命令确实是非常简单的:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

另请参阅*Examples* 小节。

创建子进程

async asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

创建一个子进程。

limit 参数为*stdout* 和*stderr* 设置*StreamReader* 包装器的缓冲区上限（如果将*subprocess.PIPE* 传给 *stdout* 和 *stderr* 参数）。

返回一个*Process* 实例。

有关其他形参的说明请查阅*loop.subprocess_exec()* 的文档。

在 3.10 版本发生变更: 移除了 *loop* 形参。

async asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

运行 *cmd* shell 命令。

limit 参数为*stdout* 和*stderr* 设置*StreamReader* 包装器的缓冲区上限（如果将*subprocess.PIPE* 传给 *stdout* 和 *stderr* 参数）。

返回一个*Process* 实例。

有关其他形参的说明请查阅*loop.subprocess_shell()* 的文档。

重要

应用程序要负责确保正确地转义所有空白字符和特殊字符以防止 shell 注入 漏洞。*shlex.quote()* 函数可以被用来正确地转义字符串中可以被用来构造 shell 命令的空白字符和特殊 shell 字符。

在 3.10 版本发生变更: 移除了 *loop* 形参。

备注

如果使用了*ProactorEventLoop* 则子进程将在 Windows 中可用。详情参见*Windows 上的子进程支持*。

参见

asyncio 还有下列低层级 API 可配合子进程使用: *loop.subprocess_exec()*, *loop.subprocess_shell()*, *loop.connect_read_pipe()*, *loop.connect_write_pipe()* 以及*子进程传输* 和*子进程协议*。

常量

asyncio.subprocess.**PIPE**

可以被传递给 *stdin*, *stdout* 或 *stderr* 形参。

如果将 *PIPE* 传给 *stdin* 参数，则*Process.stdin* 属性将指向一个*StreamWriter* 实例。

如果将 *PIPE* 传给 *stdout* 或 *stderr* 参数，则*Process.stdout* 和*Process.stderr* 属性将指向*StreamReader* 实例。

asyncio.subprocess.**STDOUT**

可以用作 *stderr* 参数的特殊值，表示标准错误应当被重定向到标准输出。

asyncio.subprocess.**DEVNULL**

可以用作 *stdin*, *stdout* 或 *stderr* 参数来处理创建函数的特殊值。它表示将为相应的子进程流使用特殊文件*os.devnull*。

与子进程交互

create_subprocess_exec() 和*create_subprocess_shell()* 函数都返回 *Process* 类的实例。*Process* 是一个高层级包装器，它允许与子进程通信并监视其完成情况。

class asyncio.subprocess.**Process**

一个用于包装由*create_subprocess_exec()* 和*create_subprocess_shell()* 函数创建的 OS 进程的对象。

这个类被设计为具有与*subprocess.Popen* 类相似的 API，但两者有一些重要的差异:

- 不同于 Popen，Process 实例没有与*poll()* 方法等价的方法；
- *communicate()* 和*wait()* 方法没有 *timeout* 形参：请使用*wait_for()* 函数；
- *Process.wait()* 方法是异步的，而*subprocess.Popen.wait()* 方法则被实现为阻塞型忙循环；
- *universal_newlines* 形参不被支持。

这个类不是线程安全的（*not thread safe*）。

请参阅子进程和线程 部分。

async wait()

等待子进程终结。

设置并返回*returncode* 属性。

> **备注**
>
> 当使用 `stdout=PIPE` 或 `stderr=PIPE` 并且子进程产生了足以阻塞 OS 管道缓冲区等待接收更多的数据的输出时，此方法会发生死锁。当使用管道时请使用*communicate()* 方法来避免这种情况。

async communicate(*input=None*)

与进程交互:

1. 发送数据到 *stdin* (如果 *input* 不为 `None`)；
2. 关闭 *stdin*;
3. 从 *stdout* 和 *stderr* 读取数据，直至到达 EOF；
4. 等待进程终结。

可选的 *input* 参数为将被发送到子进程的数据 (*bytes* 对象)。

返回一个元组 `(stdout_data, stderr_data)`。

如果在将 *input* 写入到 *stdin* 时引发了*BrokenPipeError* 或*ConnectionResetError* 异常，异常会被忽略。此条件会在进程先于所有数据被写入到 *stdin* 之前退出时发生。

如果想要将数据发送到进程的 *stdin*，则创建进程时必须使用 `stdin=PIPE`。类似地，要在结果元组中获得任何不为 `None` 的值，则创建进程时必须使用 `stdout=PIPE` 和/或 `stderr=PIPE` 参数。

注意，数据读取在内存中是带缓冲的，因此如果数据量过大或不受则不要使用此方法。

在 3.12 版本发生变更: *stdin* 在 *input=None* 时也会被关闭。

send_signal(*signal*)

将信号 *signal* 发送给子进程。

> **备注**
>
> 在 Windows 上，*SIGTERM* 是*terminate()* 的别名。`CTRL_C_EVENT` 和 `CTRL_BREAK_EVENT` 可被发送给启动时带有 *creationflags* 形参且其中包括 `CREATE_NEW_PROCESS_GROUP` 的进程。

terminate()

停止子进程。

在 POSIX 系统上此方法会发送*SIGTERM* 给子进程。

在 Windows 上会调用 Win32 API 函数 `TerminateProcess()` 来停止子进程。

kill()

杀掉子进程。

在 POSIX 系统中此方法会发送*SIGKILL* 给子进程。

在 Windows 上此方法是*terminate()* 的别名。

stdin

标准输入流 (*StreamWriter*) 或者如果进程创建时设置了 `stdin=None` 则为 `None`。

stdout

标准输出流 (*StreamReader*) 或者如果进程创建时设置了 `stdout=None` 则为 `None`。

stderr

标准错误流 (*StreamReader*) 或者如果进程创建时设置了 `stderr=None` 则为 `None`。

警告

使用*communicate()* 方法而非*process.stdin.write()*, *await process.stdout.read()* 或*await process.stderr.read()*。这可以避免由于流暂停读取或写入并阻塞子进程而导致的死锁。

pid

进程标识号（PID）。

请注意对于由*create_subprocess_shell()* 函数所创建的进程，这个属性将是所生成的 shell 的 PID。

returncode

当进程退出时返回其代号。

`None` 值表示进程尚未终止。

一个负值 `-N` 表示子进程被信号 `N` 中断 (仅 POSIX).

子进程和线程

标准 asyncio 事件循环默认支持从不同线程中运行子进程。

在 Windows 上子进程（默认）只由*ProactorEventLoop* 提供，*SelectorEventLoop* 没有子进程支持。

在 UNIX 上会使用 *child watchers* 来让子进程结束等待，详情请参阅进程监视器。

在 3.8 版本发生变更: UNIX 对于从不同线程中无限制地生成子进程会切换为使用*ThreadedChildWatcher*。

使用 *不活动的*当前子监视器生成子进程将引发*RuntimeError*。

请注意其他的事件循环实现可能有其本身的限制；请查看它们各自的文档。

参见

asyncio 中的并发和多线程 章节。

例子

一个使用*Process* 类来控制子进程并用*StreamReader* 类来从其标准输出读取信息的示例。

这个子进程是由*create_subprocess_exec()* 函数创建的:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # 创建子进程；重定向标准输出
    # 至一个管道中。
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # 读取一行输出。
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # 等待子进程退出。
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

另请参阅使用低层级 API 编写的相同示例。

19.1.6 队列集

源代码: Lib/asyncio/queues.py

---

asyncio 队列被设计成与*queue* 模块类似。尽管 asyncio 队列不是线程安全的，但是他们是被设计专用于 async/await 代码。

注意 asyncio 的队列没有 *timeout* 形参；请使用*asyncio.wait_for()* 函数为队列添加超时操作。

参见下面的*Examples* 部分。

Queue

class asyncio.**Queue**(*maxsize=0*)

先进，先出（FIFO）队列

如果 *maxsize* 小于等于零，则队列尺寸是无限的。如果是大于 `0` 的整数，则当队列达到 *maxsize* 时，`await put()` 将阻塞至某个元素被*get()* 取出。

不像标准库中的并发型*queue* ，队列的尺寸一直是已知的，可以通过调用*qsize()* 方法返回。

在 3.10 版本发生变更: 移除了 *loop* 形参。

这个类不是线程安全的（*not thread safe*）。

maxsize

队列中可存放的元素数量。

empty()

如果队列为空返回 `True` ，否则返回 `False` 。

full()

如果有*maxsize* 个条目在队列中，则返回 `True` 。

如果队列用 `maxsize=0` (默认值) 初始化，则*full()* 永远不会返回 `True`。

async get()

从队列中删除并返回一个元素。如果队列为空，则等待，直到队列中有元素。

如果队列已被关闭并且为空，或者如果队列已被立即关闭则会引发*QueueShutDown*。

get_nowait()

立即返回一个队列中的元素，如果队列内有值，否则引发异常*QueueEmpty* 。

async join()

阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费协程调用*task_done()* 表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，*join()* 阻塞被解除。

async put(*item*)

添加一个元素进队列。如果队列满了，在添加元素之前，会一直等待空闲插槽可用。

如果队列已被关闭则会引发*QueueShutDown*。

put_nowait(*item*)

不阻塞的放一个元素入队列。

如果没有立即可用的空闲槽，引发*QueueFull* 异常。

qsize()

返回队列用的元素数量。

shutdown(*immediate=False*)

关闭队列，让*get()* 和*put()* 引发*QueueShutDown*。

在默认情况下，在已关闭的队列上执行*get()* 只会在队列为空时引发异常。将 *immediate* 设为真值以改为让*get()* 立即引发异常。

所有*put()* 和*get()* 被阻塞的调用方将被撤销阻塞。如果 *immediate* 为真值，一个任务将对队列中每个剩余的项标记为已完成，它可能撤销对*join()* 的调用方的阻塞。

Added in version 3.13.

task_done()

表明之前加入队列的工作条目已经完成。

由队列的消费者使用。对于每个被用于获取工作条目的*get()*，将有一个对*task_done()* 的后续调用来告诉队列工作条目的操作已完成。

如果*join()* 当前正在阻塞，在所有条目都被处理后，将解除阻塞 (意味着每个*put()* 进队列的条目的*task_done()* 都被收到)。

`shutdown(immediate=True)` 将为队列中每个剩余的项调用*task_done()*。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 。

优先级队列

class asyncio.**PriorityQueue**

Queue 的变体；按优先级顺序取出条目 (最小的先取出)。

条目通常是 `(priority_number, data)` 形式的元组。

后进先出队列

class asyncio.**LifoQueue**

Queue 的变体，先取出最近添加的条目（后进，先出）。

异常

exception asyncio.**QueueEmpty**

当队列为空的时候，调用*get_nowait()* 方法而引发这个异常。

exception asyncio.**QueueFull**

当队列中条目数量已经达到它的 *maxsize* 的时候，调用*put_nowait()* 方法而引发的异常。

exception asyncio.**QueueShutDown**

当在已被关闭的列队上调用*put()* 或*get()* 时引发的异常。

Added in version 3.13.

例子

队列能被用于多个的并发任务的工作量分配：

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # 从队列获取一个“工作项”。
        sleep_for = await queue.get()

        # 休眠 "sleep_for" 秒。
        await asyncio.sleep(sleep_for)

        # 通知队列“工作项”已被处理。
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # 创建一个用于存储我们的“工作项”的队列。
    queue = asyncio.Queue()

    # 生成随机时段并将它们放入队列。
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # 创建三个工作任务来并发地处理队列。
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # 等待直到队列处理完毕。
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at
```

(续下页)

(接上页)

```
    # 取消我们的工作任务。
    for task in tasks:
        task.cancel()
    # 等待直到所有工作任务都被取消。
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

19.1.7 异常

源代码: Lib/asyncio/exceptions.py

---

exception asyncio.**TimeoutError**

TimeoutError 的一个已被弃用的别名，会在操作超出了给定的时限时引发。

在 3.11 版本发生变更: 这个类是*TimeoutError* 的别名。

exception asyncio.**CancelledError**

该操作已被取消。

取消 asyncio 任务时，可以捕获此异常以执行自定义操作。在几乎所有情况下，都必须重新引发异常。

在 3.8 版本发生变更: *CancelledError* 现在是*BaseException* 的子类而不是*Exception* 的子类。

exception asyncio.**InvalidStateError**

Task 或*Future* 的内部状态无效。

在为已设置结果值的未来对象设置结果值等情况下，可以引发此问题。

exception asyncio.**SendfileNotAvailableError**

”sendfile” 系统调用不适用于给定的套接字或文件类型。

子类*RuntimeError* 。

exception asyncio.**IncompleteReadError**

请求的读取操作未完全完成。

由*asyncio stream APIs* 提出

此异常是*EOFError* 的子类。

expected

预期字节的总数（*int* ）。

partial

到达流结束之前读取的*bytes* 字符串。

exception asyncio.**LimitOverrunError**

在查找分隔符时达到缓冲区大小限制。

由*asyncio stream APIs* 提出

consumed

要消耗的字节总数。

19.1.8 事件循环

源代码: Lib/asyncio/events.py, Lib/asyncio/base_events.py

前言

事件循环是每个 asyncio 应用的核心。事件循环会运行异步任务和回调，执行网络 IO 操作，以及运行子进程。

应用开发者通常应当使用高层级的 asyncio 函数，例如*asyncio.run()*，应当很少有必要引用循环对象或调用其方法。本节所针对的主要是低层级代码、库和框架的编写者，他们需要更细致地控制事件循环行为。

获取事件循环

以下低层级函数可被用于获取、设置或创建事件循环:

asyncio.**get_running_loop**()

返回当前 OS 线程中正在运行的事件循环。

如果没有正在运行的事件循环则会引发*RuntimeError*。

此函数只能由协程或回调来调用。

Added in version 3.7.

asyncio.**get_event_loop**()

获取当前事件循环。

当在协程或回调中被调用时（例如通过 call_soon 或类似 API 加入计划任务），此函数将始终返回正在运行的事件循环。

如果没有设置正在运行的事件循环，此函数将返回 `get_event_loop_policy().get_event_loop()` 调用的结果。

由于此函数具有相当复杂的行为（特别是在使用了自定义事件循环策略的时候），更推荐在协程和回调中使用*get_running_loop()* 函数而非*get_event_loop()*。

如上文所说，请考虑使用高层级的*asyncio.run()* 函数，而不是使用这些低层级的函数来手动创建和关闭事件循环。

自 3.12 版本弃用: 如果没有当前事件循环则会发出弃用警告。在未来的 Python 发布版中这将被改为错误。

asyncio.**set_event_loop**(*loop*)

将 *loop* 设为当前 OS 线程的当前事件循环。

asyncio.**new_event_loop**()

创建并返回一个新的事件循环对象。

请注意*get_event_loop()*, *set_event_loop()* 以及*new_event_loop()* 函数的行为可以通过*设置自定义事件循环策略* 来改变。

目录

本文档包含下列小节:

- *事件循环方法集* 章节是事件循环 APIs 的参考文档;
- *回调处理* 章节是从调度方法例如*loop.call_soon()* 和*loop.call_later()* 中返回*Handle* 和*TimerHandle* 实例的文档。
- *Server Objects* 章节记录了从事件循环方法返回的类型，比如*loop.create_server()*；
- *Event Loop Implementations* 章节记录了*SelectorEventLoop* 和*ProactorEventLoop* 类;
- *Examples* 章节展示了如何使用某些事件循环 API。

事件循环方法集

事件循环有下列 **低级** APIs：

- 运行和停止循环
- 安排回调
- 调度延迟回调
- 创建 *Future* 和 *Task*
- 打开网络连接
- 创建网络服务
- 传输文件
- *TLS* 升级
- 监控文件描述符
- 直接使用 *socket* 对象
- *DNS*
- 使用管道
- *Unix* 信号
- 在线程或者进程池中执行代码。
- 错误处理 *API*
- 开启调试模式
- 运行子进程

运行和停止循环

`loop.`**`run_until_complete`**`(`*`future`*`)`

运行直到 *future* (`Future` 的实例) 被完成。

如果参数是*coroutine object* ，将被隐式调度为`asyncio.Task` 来运行。

返回 Future 的结果或者引发相关异常。

`loop.`**`run_forever`**`()`

运行事件循环直到`stop()` 被调用。

如果`stop()` 在调用`run_forever()` 之前被调用，循环将轮询一次 I/O 选择器并设置超时为零，再运行所有已加入计划任务的回调来响应 I/O 事件（以及已加入计划任务的事件），然后退出。

如果`stop()` 在`run_forever()` 运行期间被调用，循环将运行当前批次的回调然后退出。请注意在此情况下由回调加入计划任务的新回调将不会运行；它们将会在下次`run_forever()` 或`run_until_complete()` 被调用时运行。

`loop.`**`stop`**`()`

停止事件循环。

`loop.`**`is_running`**`()`

返回 `True` 如果事件循环当前正在运行。

`loop.`**`is_closed`**`()`

如果事件循环已经被关闭，返回 `True` 。

loop.**close**()

关闭事件循环。

当这个函数被调用的时候，循环必须处于非运行状态。pending 状态的回调将被丢弃。

此方法清除所有的队列并立即关闭执行器，不会等待执行器完成。

这个方法是幂等的和不可逆的。事件循环关闭后，不应调用其他方法。

async loop.**shutdown_asyncgens**()

安装所有当前打开的*asynchronous generator* 对象通过 aclose() 调用来关闭。在调用此方法后，如果有新的异步生成器被迭代则事件循环将会发出警告。这应当被用来可靠地最终化所有已加入计划任务的异步生成器。

请注意当使用*asyncio.run()* 时不必调用此函数。

示例:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

Added in version 3.6.

async loop.**shutdown_default_executor**(*timeout=None*)

安排默认执行器的关闭并等待它合并*ThreadPoolExecutor* 中的所有线程。一旦此方法被调用，将默认执行器与*loop.run_in_executor()* 一起使用将引发*RuntimeError*。

timeout 形参指定提供给执行器结束合并的时间限制（为*float* 形式的秒数）。在默认情况下，该值为 None，即允许执行器无时间限制地执行。

如果达到了 *timeout*，将会发出*RuntimeWarning* 并且默认执行器将会终结而不等待其线程结束合并。

> **备注**
>
> 当使用*asyncio.run()* 不要调用此方法，因为它会自动处理默认执行器的关闭。

Added in version 3.9.

在 3.12 版本发生变更: 增加了 *timeout* 形参。

安排回调

loop.**call_soon**(*callback*, **args*, *context=None*)

安排 *callback* *callback* 在事件循环的下一次迭代时附带 *args* 参数被调用。

返回一个*asyncio.Handle* 的实例，可在此后被用来取消回调。

回调按其注册顺序被调用。每个回调仅被调用一次。

可选的仅限关键字参数 *context* 指定一个自定义的*contextvars.Context* 供 *callback* 在其中运行。当未提供 *context* 时回调将使用当前上下文。

与*call_soon_threadsafe()* 不同，此方法不是线程安全的。

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)

call_soon() 的线程安全版。当从另一个线程安排回调时，必须使用此函数，因为*call_soon()* 不是线程安全的。

此函数可以从一个重入上下文或信号处理器安全地调用，不过，在这样的上下文中使用所返回的句柄则不是安全或有益的。

如果在已被关闭的循环上调用则会引发*RuntimeError*。这可能会在主应用程序被关闭时在二级线程上发生。

参见*concurrency and multithreading* 部分的文档。

在 3.7 版本发生变更: 加入键值类形参 *context*。请参阅 **PEP 567** 查看更多细节。

备注

大多数*asyncio* 的调度函数不让传递关键字参数。为此，请使用*functools.partial()*：

```
# 将把 "print("Hello", flush=True)" 加入计划任务
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

使用 partial 对象通常比使用 lambda 更方便，asyncio 在调试和错误消息中能更好的呈现 partial 对象。

调度延迟回调

事件循环提供安排调度函数在将来某个时刻调用的机制。事件循环使用单调时钟来跟踪时间。

`loop.call_later`(*delay*, *callback*, **args*, *context=None*)

安排 *callback* 在给定的 *delay* 秒（可以是 int 或者 float）后被调用。

返回一个*asyncio.TimerHandle* 实例，该实例能用于取消回调。

callback 只被调用一次。如果两个回调被安排在同样的时间点，执行顺序未限定。

可选的位置参数 *args* 在被调用的时候传递给 *callback* 。如果你想把关键字参数传递给 *callback* ，请使用*functools.partial()* 。

可选键值类的参数 *context* 允许 *callback* 运行在一个指定的自定义*contextvars.Context* 对象中。如果没有提供 *context* ，则使用当前上下文。

在 3.7 版本发生变更: 加入键值类形参 *context*。请参阅 **PEP 567** 查看更多细节。

在 3.8 版本发生变更: 在 Python 3.7 和更早版本的默认事件循环实现中，*delay* 不能超过一天。这在 Python 3.8 中已被修复。

`loop.call_at`(*when*, *callback*, **args*, *context=None*)

安排 *callback* 在给定的绝对时间戳 *when* (int 或 float) 被调用，使用与*loop.time()* 同样的时间参考。

本方法的行为和*call_later()* 方法相同。

返回一个*asyncio.TimerHandle* 实例，该实例能用于取消回调。

在 3.7 版本发生变更: 加入键值类形参 *context*。请参阅 **PEP 567** 查看更多细节。

在 3.8 版本发生变更: 在 Python 3.7 和更早版本的默认事件循环实现中，*when* 和当前时间相差不能超过一天。在这 Python 3.8 中已被修复。

`loop.time`()

根据时间循环内部的单调时钟，返回当前时间为一个*float* 值。

备注

在 3.8 版本发生变更: 在 Python 3.7 和更早版本中超时 (相对的 *delay* 或绝对的 *when*) 不能超过一天。这在 Python 3.8 中已被修复。

参见

asyncio.sleep() 函数。

创建 Future 和 Task

loop.**create_future**()

创建一个附加到事件循环中的*asyncio.Future* 对象。

这是在 asyncio 中创建 Futures 的首选方式。这让第三方事件循环可以提供 Future 对象的替代实现(更好的性能或者功能)。

Added in version 3.5.2.

loop.**create_task**(*coro*, *, *name=None*, *context=None*)

将协程 *coro* 的执行加入计划任务。返回一个*Task* 对象。

第三方的事件循环可以使用它们自己的*Task* 子类来满足互操作性。这种情况下结果类型是一个*Task* 的子类。

如果提供了 *name* 参数且不为 None，它会使用*Task.set_name()* 来设为任务的名称。

可选的 *context* 参数允许指定自定义的*contextvars.Context* 供 *coro* 运行。当未提供 *context* 时将创建当前上下文的副本。

在 3.8 版本发生变更: 添加了 *name* 参数。

在 3.11 版本发生变更: 增加了 *context* 形参。

loop.**set_task_factory**(*factory*)

设置一个任务工厂，它将由*loop.create_task()* 来使用。

如果 *factory* 为 None 则将设置默认的任务工厂。在其他情况下，*factory* 必须是一个 可调用对象且其签名要匹配 (loop, coro, **kwargs)，其中 *loop* 是一个指向活动事件循环的引用，而 *coro* 是一个协程对象。该可调用对象必须传给所有 *kwargs*，并且返回一个*asyncio.Task* 兼容对象。

loop.**get_task_factory**()

返回一个任务工厂，或者如果是使用默认值则返回 None。

打开网络连接

async loop.**create_connection**(*protocol_factory*, *host=None*, *port=None*, *, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*, *all_errors=False*)

打开一个流式传输连接，连接到由 *host* 和 *port* 指定的地址。

套接字族可以是*AF_INET* 或*AF_INET6*，具体取决于 *host* (或 *family* 参数，如果有提供的话)。

套接字类型将为*SOCK_STREAM*。

protocol_factory 必须为一个返回*asyncio 协议* 实现的可调用对象。

这个方法会尝试在后台创建连接。当创建成功，返回 (transport, protocol) 组合。

底层操作的大致的执行顺序是这样的:

1. 创建连接并为其创建一个传输。
2. 不带参数地调用 *protocol_factory* 并预期返回一个协议 实例。
3. 协议实例通过调用其*connection_made()* 方法与传输进行配对。
4. 成功时返回一个 (transport, protocol) 元组。

创建的传输是一个具体实现相关的双向流。

其他参数:

- *ssl*: 如果给定该参数且不为假值，则会创建一个 SSL/TLS 传输（默认创建一个纯 TCP 传输）。如果 *ssl* 是一个*ssl.SSLContext* 对象，则会使用此上下文来创建传输对象；如果 *ssl* 为*True*，则会使用从*ssl.create_default_context()* 返回的默认上下文。

> **参见**
>
> *SSL/TLS security considerations*

- *server_hostname* 设置或覆盖目标服务器的证书将要匹配的主机名。应当只在 *ssl* 不为 `None` 时传入。默认情况下会使用 *host* 参数的值。如果 *host* 为空那就没有默认值，你必须为 *server_hostname* 传入一个值。如果 *server_hostname* 为空字符串，则主机名匹配会被禁用（这是一个严重的安全风险，使得潜在的中间人攻击成为可能）。
- *family*, *proto*, *flags* 是可选的地址族、协议和标志，它们会被传递给 getaddrinfo() 来对 *host* 进行解析。如果要指定的话，这些都应该是来自于 *socket* 模块的对应常量。
- 如果给出 *happy_eyeballs_delay*，它将为此链接启用 Happy Eyeballs。该函数应当为一个表示在开始下一个并行尝试之前要等待连接尝试完成的秒数的浮点数。这也就是在 **RFC 8305** 中定义的" 连接尝试延迟"。该 RFC 所推荐的合理默认值为 `0.25` (250 毫秒)。
- *interleave* 控制当主机名解析为多个 IP 地址时的地址重排序。如果该参数为 `0` 或未指定，则不会进行重排序，这些地址会按 *getaddrinfo()* 所返回的顺序进行尝试。如果指定了一个正整数，这些地址会按地址族交错排列，而指定的整数会被解读为 **RFC 8305** 所定义的" 首个地址族计数"。如果 *happy_eyeballs_delay* 未指定则默认值为 `0`，否则为 `1`。
- 如果给出 *sock*，它应当是一个已存在、已连接并将被传输所使用的 *socket.socket* 对象。如果给出了 *sock*，则 *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave* 和 *local_addr* 都不应当被指定。

> **备注**
>
> *sock* 参数可将套接字的所有权转给所创建的传输。要关闭该套接字，请调用传输的 *close()* 方法。

- 如果给出 *local_addr*，它应当是一个用来在本地绑定套接字的 `(local_host, local_port)` 元组。*local_host* 和 *local_port* 会使用 `getaddrinfo()` 来查找，这与 *host* 和 *port* 类似。
- *ssl_handshake_timeout* 是（用于 TLS 连接的）在放弃连接之前要等待 TLS 握手完成的秒数。如果参数为 `None` 则使用 (默认的) `60.0`。
- *ssl_shutdown_timeout* 是在放弃连接之前要等待 SSL 关闭完成的秒数。如为 `None` (默认值) 则使用 `30.0`。
- *all_errors* 确定当无法创建连接时要引发何种异常。在默认情况下，只有一个 `Exception` 会被引发：即只有一个异常或所有错误的消息相同，或者是合并了多个错误消息的单个 `OSError`。当 `all_errors` 为 `True` 时，将引发一个包含所有异常的 `ExceptionGroup` (即使只有一个异常)。

在 3.5 版本发生变更: *ProactorEventLoop* 类中添加 SSL/TLS 支持。

在 3.6 版本发生变更: 套接字选项 *socket.TCP_NODELAY* 默认将为所有 TCP 连接设置。

在 3.7 版本发生变更: 添加了 *ssl_handshake_timeout* 参数。

在 3.8 版本发生变更: 增加了 *happy_eyeballs_delay* 和 *interleave* 形参。

Happy Eyeballs 算法：成功使用双栈主机。当服务器的 IPv4 路径和协议工作正常，但服务器的 IPv6 路径和协议工作不正常时，双线客户端应用程序相比仅有 IPv4 的客户端会感受到明显的连接延迟。这是不可接受的因为它会导致双栈客户端糟糕的用户体验。这篇文档指明了减少这种用户可见延迟的算法要求并提供了具体的算法。

详情参见: https://datatracker.ietf.org/doc/html/rfc6555

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

在 3.12 版本发生变更: 添加了 *all_errors*。

> **参见**
>
> *open_connection()* 函数是一个高层级的替代 API。它返回一对 (*StreamReader*, *StreamWriter*)，可在 async/await 代码中直接使用。

async loop.**create_datagram_endpoint**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

创建一个数据报连接。

套接字族可以是*AF_INET*, *AF_INET6* 或*AF_UNIX*，具体取决于 *host* (或 *family* 参数，如果有提供的话)。

套接字类型将为*SOCK_DGRAM*。

protocol_factory 必须为一个返回协议 实现的可调用对象。

成功时返回一个 `(transport, protocol)` 元组。

其他参数：

- 如果给出 *local_addr*，它应当是一个用来在本地绑定套接字的 `(local_host, local_port)` 元组。*local_host* 和 *local_port* 是使用*getaddrinfo()* 来查找的。
- *remote_addr*，如果指定的话，就是一个 `(remote_host, remote_port)` 元组，用于同一个远程地址连接。*remote_host* 和 *remote_port* 是使用*getaddrinfo()* 来查找的。
- *family*, *proto*, *flags* 是可选的地址族，协议和标志，其会被传递给*getaddrinfo()* 来完成 *host* 的解析。如果要指定的话，这些都应该是来自于*socket* 模块的对应常量。
- *reuse_port* 告知内核允许此端点绑定到其他现有端点所绑定的相同端口上，只要它们在创建时都设置了这个旗标。这个选项在 Windows 和某些 Unix 上将不受支持。如果*socket.SO_REUSEPORT* 常量未被定义那么该功能就是不受支持的。
- *allow_broadcast* 告知内核允许此端点向广播地址发送消息。
- *sock* 可选择通过指定此值用于使用一个预先存在的，已经处于连接状态的*socket.socket* 对象，并将其提供给此传输对象使用。如果指定了这个值，*local_addr* 和 *remote_addr* 就应该被忽略 (必须为*None*)。

> **备注**
>
> *sock* 参数可将套接字的所有权转给所创建的传输。要关闭该套接字，请调用传输的*close()* 方法。

参见*UDP echo 客户端协议* 和*UDP echo 服务端协议* 的例子。

在 3.4.4 版本发生变更: 添加了 *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast* 和 *sock* 等形参。

在 3.8 版本发生变更: 添加 WIndows 的支持。

在 3.8.1 版本发生变更: *reuse_address* 形参已不再受支持，因为使用*socket.SO_REUSEADDR* 对于 UDP 会造成显著的安全问题。显式地传入 `reuse_address=True` 将引发异常。

当具有不同 UID 的多个进程将套接字赋给具有 `SO_REUSEADDR` 的相同 UDP 套接字地址时，传入的数据包可能会在套接字间随机分配。

对于受支持的平台，可以使用 *reuse_port* 作为类似功能的替代。通过 *reuse_port*，将会改用*socket.SO_REUSEPORT*，它能防止具有不同 UID 的进程将套接字赋给相同的套接字地址。

在 3.11 版本发生变更: 自 Python 3.8.1, 3.7.6 和 3.6.10 起被禁用的 *reuse_address* 形参现已被完全移除。

async loop.**create_unix_connection**(*protocol_factory*, *path=None*, *, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

创建 Unix 连接

套接字族将为*AF_UNIX*；套接字类型将为*SOCK_STREAM*。

成功时返回一个 `(transport, protocol)` 元组。

path 是所要求的 Unix 域套接字的名字，除非指定了 *sock* 形参。抽象的 Unix 套接字, *str*, *bytes* 和*Path* 路径都是受支持的。

请查看*loop.create_connection()* 方法的文档了解有关此方法的参数的信息。

Availability: Unix.

在 3.7 版本发生变更: 增加了 *ssl_handshake_timeout* 参数。现在 *path* 参数可以是一个*path-like object* 。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

创建网络服务

async loop.**create_server**(*protocol_factory*, *host=None*, *port=None*, *, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

创建一个 TCP 服务器 (套接字类型*SOCK_STREAM*) 在 *host* 地址的 *port* 上进行监听。

返回一个*Server* 对象。

参数:

- *protocol_factory* 必须为一个返回协议 实现的可调用对象。
- *host* 形参可被设为几种类型，它确定了服务器所应监听的位置:
  - 如果 *host* 是一个字符串，则 TCP 服务器会被绑定到 *host* 所指明的单一网络接口。
  - 如果 *host* 是一个字符串序列，则 TCP 服务器会被绑定到序列所指明的所有网络接口。
  - 如果 *host* 是一个空字符串或 `None`，则会应用所有接口并将返回包含多个套接字的列表（通常是一个 IPv4 的加一个 IPv6 的）。
- 可以设置 *port* 参数来指定服务器应该监听哪个端口。如果为 `0` 或者 `None` （默认），将选择一个随机的未使用的端口（注意，如果 *host* 解析到多个网络接口，将为每个接口选择一个不同的随机端口）。
- *family* 可被设为*socket.AF_INET* 或*AF_INET6* 以强制此套接字使用 IPv4 或 IPv6。如果未设定，则 *family* 将通过主机名为确定 (默认为*AF_UNSPEC*)。
- *flags* 是用于*getaddrinfo()* 的位掩码。
- 可以选择指定 *sock* 以便使用预先存在的套接字对象。如果指定了此参数，则不可再指定 *host* 和 *port*。

  > **备注**
  >
  > *sock* 参数可将套接字的所有权转给所创建的服务器。要关闭该套接字，请调用服务器的*close()* 方法。

- *backlog* 是传递给*listen()* 的最大排队连接的数量（默认为 100）。
- *ssl* 可被设置为一个*SSLContext* 实例以在所接受的连接上启用 TLS。
- *reuse_address* 告知内核要重用一个处于 `TIME_WAIT` 状态的本地套接字，而不是等待其自然超时失效。如果未指定此参数则在 Unix 上将自动设置为 `True`。

- *reuse_port* 告知内核，只要在创建的时候都设置了这个标志，就允许此端点绑定到其它端点列表所绑定的同样的端口上。这个选项在 Windows 上是不支持的。
- *keep_alive* 设为 `True` 将通过启用定期的消息传输来使连接保持活动状态。

在 3.13 版本发生变更: 增加了 *keep_alive* 形参。

- *ssl_handshake_timeout* 是（用于 TLS 服务器的）在放弃连接之前要等待 TLS 握手完成的秒数。如果参数为 (默认值) `None` 则为 `60.0` 秒。
- *ssl_shutdown_timeout* 是在放弃连接之前要等待 SSL 关闭完成的秒数。如为 `None` (默认值) 则使用 `30.0`。
- *start_serving* 设置成 `True` (默认值) 会导致创建 server 并立即开始接受连接。设置成 `False`，用户需要等待*`Server.start_serving()`* 或者*`Server.serve_forever()`* 以使 server 开始接受连接。

在 3.5 版本发生变更: *`ProactorEventLoop`* 类中添加 SSL/TLS 支持。

在 3.5.1 版本发生变更: *host* 形参可以是一个字符串的序列。

在 3.6 版本发生变更: 增加了 *ssl_handshake_timeout* 和 *start_serving* 形参。套接字选项*socket.TCP_NODELAY* 会默认为所有 TCP 连接设置。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

参见

`start_server()` 函数是一个高层级的替代 API，它返回一对*`StreamReader`* 和*`StreamWriter`*，可在 async/await 代码中使用。

async `loop.`**create_unix_server**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*, *cleanup_socket=True*)

与*`loop.create_server()`* 类似但是专用于*`AF_UNIX`* 套接字族。

path 是必要的 Unix 域套接字名称，除非提供了 *sock* 参数。抽象的 Unix 套接字, *`str`*, *`bytes`* 和*`Path`* 路径都是受支持的。

如果 *cleanup_socket* 为真值那么当服务器关闭时 Unix 套接字将自动从文件系统中被移除，除非套接字在服务器创建之后被替换。

请查看*`loop.create_server()`* 方法的文档了解有关此方法的参数的信息。

Availability: Unix.

在 3.7 版本发生变更: 增加了 *ssl_handshake_timeout* 和 *start_serving* 参数。现在 *path* 参数可以是一个*`Path`* 对象。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

在 3.13 版本发生变更: 增加了 *cleanup_socket* 形参。

async `loop.`**connect_accepted_socket**(*protocol_factory*, *sock*, ***, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

将已被接受的连接包装成一个传输/协议对。

此方法可被服务器用来接受 asyncio 以外的连接，但是使用 asyncio 来处理它们。

参数:

- *protocol_factory* 必须为一个返回协议 实现的可调用对象。
- *sock* 是一个预先存在的套接字对象，它是由*`socket.accept`* 返回的。

> **备注**
>
> *sock* 参数可将套接字的所有权转给所创建的传输。要关闭该套接字，请调用传输的`close()`方法。

- *ssl* 可被设置为一个`SSLContext`以在接受的连接上启用 SSL。
- *ssl_handshake_timeout* 是 (为一个 SSL 连接) 在中止连接前，等待 SSL 握手完成的时间【单位秒】。如果为 `None` (缺省) 则是 `60.0` 秒。
- *ssl_shutdown_timeout* 是在放弃连接之前要等待 SSL 关闭完成的秒数。如为 `None` (默认值) 则使用 `30.0`。

返回一个 `(transport, protocol)` 对。

Added in version 3.5.3.

在 3.7 版本发生变更: 添加了 *ssl_handshake_timeout* 参数。

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

传输文件

async loop.**sendfile**(*transport*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

将 *file* 通过 *transport* 发送。返回所发送的字节总数。

如果可用的话，该方法将使用高性能的`os.sendfile()`。

file 必须是个二进制模式打开的常规文件对象。

offset 指明从何处开始读取文件。如果指定了 *count*，它是要传输的字节总数而不再一直发送文件直至抵达 EOF。文件位置总是会被更新，即使此方法引发了错误，并可以使用`file.tell()`来获取实际发送的字节总数。

fallback 设为 `True` 会使得 asyncio 在平台不支持 sendfile 系统调用时手动读取并发送文件（例如 Windows 或 Unix 上的 SSL 套接字）。

如果系统不支持 *sendfile* 系统调用且 *fallback* 为 `False` 则会引发`SendfileNotAvailableError`。

Added in version 3.7.

TLS 升级

async loop.**start_tls**(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

将现有基于传输的连接升级到 TLS。

创建一个 TLS 编码器/解码器实例并将其插入到 *transport* 和 *protocol* 之间。该编码器/解码器同时实现了面向 *transport* 的协议和面向 *protocol* 的传输。

返回已创建的双接口实例。在 *await* 之后，*protocol* 必须使用原始 *transport* 来停止并仅与所返回的对象通信因为编码器会缓存 *protocol* 方的数据并会不定期地与 *transport* 交换额外的 TLS 会话数据包。

在某些情况下（例如当传入的 transport 已经关闭）这可能返回 `None`。

参数:

- *transport* 和 *protocol* 实例的方法与`create_server()`和`create_connection()`所返回的类似。
- *sslcontext*： 一个已经配置好的`SSLContext`实例。
- 当服务端连接已升级时 (如`create_server()`所创建的对象) *server_side* 会传入 `True`。
- *server_hostname*： 设置或者覆盖目标服务器证书中相对应的主机名。
- *ssl_handshake_timeout* 是（用于 TLS 连接的）在放弃连接之前要等待 TLS 握手完成的秒数。如果参数为 `None` 则使用 (默认的) `60.0`。

- *ssl_shutdown_timeout* 是在放弃连接之前要等待 SSL 关闭完成的秒数。如为 `None` (默认值) 则使用 `30.0`。

Added in version 3.7.

在 3.11 版本发生变更: 添加了 *ssl_shutdown_timeout* 形参。

监控文件描述符

`loop.`**`add_reader`**(*fd*, *callback*, **args*)

开始监视 *fd* 文件描述符以获取读取的可用性，一旦 *fd* 可用于读取，使用指定的参数调用 *callback* 。

任何为 *fd* 注册的预先存在的回调都会被取消并由 *callback* 代替。

`loop.`**`remove_reader`**(*fd*)

停止监视 *fd* 文件描述符的读取可用性。如果之前正在监视 *fd* 的读取则返回 `True`。

`loop.`**`add_writer`**(*fd*, *callback*, **args*)

开始监视 *fd* 文件描述符的写入可用性，一旦 *fd* 可用于写入，使用指定的参数调用 *callback* 。

任何为 *fd* 注册的预先存在的回调都会被取消并由 *callback* 代替。

使用*functools.partial() 传递关键字参数* 给 *callback*.

`loop.`**`remove_writer`**(*fd*)

停止监视 *fd* 文件描述符的写入可用性。如果之前正在监视 *fd* 的写入则返回 `True`。

另请查看*平台支持* 一节了解以上方法的某些限制。

直接使用 socket 对象

通常，使用基于传输的 API 的协议实现，例如*loop.create_connection()* 和*loop.create_server()* 比直接使用套接字的实现更快。但是，在某些应用场景下性能并不非常重要，直接使用*socket* 对象会更方便。

`async` `loop.`**`sock_recv`**(*sock*, *nbytes*)

从 *sock* 接收至多 *nbytes*。*socket.recv()* 的异步版本。

返回接收到的数据【bytes 对象类型】。

sock 必须是个非阻塞 socket。

在 3.7 版本发生变更: 虽然这个方法总是被记录为协程方法，但它在 Python 3.7 之前的发行版中会返回一个*Future*。从 Python 3.7 开始它则是一个 `async def` 方法。

`async` `loop.`**`sock_recv_into`**(*sock*, *buf*)

从 *sock* 接收数据放入 *buf* 缓冲区。模仿了阻塞型的*socket.recv_into()* 方法。

返回写入缓冲区的字节数。

sock 必须是个非阻塞 socket。

Added in version 3.7.

`async` `loop.`**`sock_recvfrom`**(*sock*, *bufsize*)

从 *sock* 接收最大为 *bufsize* 的数据报。*socket.recvfrom()* 的异步版本。

返回一个 (已接收数据, 远程地址) 元组。

sock 必须是个非阻塞 socket。

Added in version 3.11.

async loop.**sock_recvfrom_into**(*sock*, *buf*, *nbytes=0*)

从 *sock* 接收最大为 *nbytes* 的数据报并放入 *buf*。*socket.recvfrom_into()* 的异步版本。

返回一个 (已接收字节数, 远程地址) 元组。

sock 必须是个非阻塞 socket。

Added in version 3.11.

async loop.**sock_sendall**(*sock*, *data*)

将 *data* 发送到 *sock* 套接字。*socket.sendall()* 的异步版本。

此方法会持续发送数据到套接字直至 *data* 中的所有数据发送完毕或是有错误发生。当成功时会返回 None。当发生错误时，会引发一个异常。此外，没有办法能确定有多少数据或是否有数据被连接的接收方成功处理。

sock 必须是个非阻塞 socket。

在 3.7 版本发生变更: 虽然这个方法一直被标记为协程方法。但是，Python 3.7 之前，该方法返回*Future*，从 Python 3.7 开始，这个方法是 async def 方法。

async loop.**sock_sendto**(*sock*, *data*, *address*)

从 *sock* 向 *address* 发送数据报。*socket.sendto()* 的异步版本。

返回已发送的字节数。

sock 必须是个非阻塞 socket。

Added in version 3.11.

async loop.**sock_connect**(*sock*, *address*)

将 *sock* 连接到位于 *address* 的远程套接字。

socket.connect() 的异步版本。

sock 必须是个非阻塞 socket。

在 3.5.2 版本发生变更: address 不再需要被解析。sock_connect 将尝试检查 *address* 是否已通过调用*socket.inet_pton()* 被解析。如果没有，则将使用*loop.getaddrinfo()* 来解析 *address*。

参见

loop.create_connection() 和*asyncio.open_connection()*。

async loop.**sock_accept**(*sock*)

接受一个连接。模仿了阻塞型的*socket.accept()* 方法。

此 *scoket* 必须绑定到一个地址上并且监听连接。返回值是一个 (conn, address) 对，其中 *conn* 是一个 新 * 的套接字对象，用于在此连接上收发数据，**address* 是连接的另一端的套接字所绑定的地址。

sock 必须是个非阻塞 socket。

在 3.7 版本发生变更: 虽然这个方法一直被标记为协程方法。但是，Python 3.7 之前，该方法返回*Future*，从 Python 3.7 开始，这个方法是 async def 方法。

参见

loop.create_server() 和*start_server()*。

async loop.**sock_sendfile**(*sock*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

在可能的情况下使用高性能的*os.sendfile* 发送文件。返回所发送的字节总数。

socket.sendfile() 的异步版本。

sock 必须为非阻塞型的*socket.SOCK_STREAM socket*。

file 必须是个用二进制方式打开的常规文件对象。

offset 指明从何处开始读取文件。如果指定了 *count*，它是要传输的字节总数而不再一直发送文件直至抵达 EOF。文件位置总是会被更新，即使此方法引发了错误，并可以使用*file.tell()* 来获取实际发送的字节总数。

当 *fallback* 被设为 `True` 时，会使用 asyncio 在平台不支持 sendfile 系统调用时手动读取并发送文件（例如 Windows 或 Unix 上的 SSL 套接字）。

如果系统不支持 *sendfile* 并且 *fallback* 为 `False`，引发*SendfileNotAvailableError* 异常。

sock 必须是个非阻塞 socket。

Added in version 3.7.

DNS

async `loop.`**getaddrinfo**(*host*, *port*, *, *family=0*, *type=0*, *proto=0*, *flags=0*)

异步版的*socket.getaddrinfo()* 。

async `loop.`**getnameinfo**(*sockaddr*, *flags=0*)

异步版的*socket.getnameinfo()* 。

备注

getaddrinfo 和 *getnameinfo* 均在内部通过循环的默认线程池执行器使用其同步版本。当执行器饱和时，这些方法可能会遭遇延迟，对此高层级网络库可能报告为更多的超时。为缓解此问题，可以考虑使用针对其他用户任务的自定义执行器，或者设置具有更多工作线程的默认执行器。

在 3.7 版本发生变更: *getaddrinfo* 和 *getnameinfo* 方法一直被标记返回一个协程，但是 Python 3.7 之前，实际返回的是*asyncio.Future* 对象。从 Python 3.7 开始，这两个方法是协程。

使用管道

async `loop.`**connect_read_pipe**(*protocol_factory*, *pipe*)

在事件循环中注册 *pipe* 的读取端。

protocol_factory 必须为一个返回*asyncio 协议* 实现的可调用对象。

pipe 是个*类似文件型对象*.

返回一对 `(transport, protocol)`，其中 *transport* 支持*ReadTransport* 接口而 *protocol* 是由 *protocol_factory* 所实例化的对象。

使用*SelectorEventLoop* 事件循环，*pipe* 被设置为非阻塞模式。

async `loop.`**connect_write_pipe**(*protocol_factory*, *pipe*)

在事件循环中注册 *pipe* 的写入端。

protocol_factory 必须为一个返回*asyncio 协议* 实现的可调用对象。

pipe 是个*类似文件型对象*.

返回一对 `(transport, protocol)`，其中 *transport* 支持*WriteTransport* 接口而 *protocol* 是由 *protocol_factory* 所实例化的对象。

使用*SelectorEventLoop* 事件循环，*pipe* 被设置为非阻塞模式。

备注

在 Windows 中*SelectorEventLoop* 不支持上述方法。对于 Windows 请改用*ProactorEventLoop*。

参见

loop.subprocess_exec() 和*loop.subprocess_shell()* 方法。

Unix 信号

`loop.`**`add_signal_handler`**(*signum*, *callback*, **args*)

设置 *callback* 作为 *signum* 信号的处理程序。

此回调将与该事件循环中其他加入队列的回调和可运行协程一起由 *loop* 唤起。不同与使用*signal.signal()* 注册的信号处理程序，使用此函数注册的回调可以与事件循环进行交互。

如果信号数字非法或者不可捕获，就抛出一个*ValueError* 。如果建立处理器的过程中出现问题，会抛出一个*RuntimeError* 。

使用*functools.partial() 传递关键字参数* 给 *callback*.

和*signal.signal()* 一样，这个函数只能在主线程中调用。

`loop.`**`remove_signal_handler`**(*sig*)

移除 *sig* 信号的处理程序。

如果信号处理程序被移除则返回 `True`，否则如果给定信号未设置处理程序则返回 `False`。

Availability: Unix.

参见

signal 模块。

在线程或者进程池中执行代码。

`awaitable` `loop.`**`run_in_executor`**(*executor*, *func*, **args*)

安排在指定的执行器中调用 *func* 。

executor 参数应当是一个*concurrent.futures.Executor* 实例。如果 *executor* 为 `None` 则为使用默认执行器。默认执行器可通过*loop.set_default_executor()* 来设置，在其他情况下，将在有需要时惰性初始化*concurrent.futures.ThreadPoolExecutor* 并由*run_in_executor()* 来使用。

示例:

```
import asyncio
import concurrent.futures

def blocking_io():
    # 文件操作（如日志记录）会阻塞事件循环：
    # 应在线程池中运行它们。
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU 密集型操作会阻塞事件循环：
    # 通常建议在进程中运行它们。
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. 运行默认循环的执行器：
```

(续下页)

(接上页)

```
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. 在自定义的线程池中运行:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. 在自定义的进程池中运行:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

if __name__ == '__main__':
    asyncio.run(main())
```

请注意需要为选项 3 设置入口点保护 (`if __name__ == '__main__'`)，这是由于*multiprocessing* 的特殊性，它将由*ProcessPoolExecutor* 来使用。参见主模块的安全导入。

这个方法返回一个*asyncio.Future* 对象。

使用*functools.partial() 传递关键字参数* 给 *func* 。

在 3.5.3 版本发生变更: *loop.run_in_executor()* 不会再配置它所创建的线程池执行器的 `max_workers`，而是将其留给线程池执行器 (*ThreadPoolExecutor*) 来设置默认值。

`loop.`**`set_default_executor`**(*executor*)

将 *executor* 设为*run_in_executor()* 所使用的默认执行器。*executor* 必须是*ThreadPoolExecutor* 的实例。

在 3.11 版本发生变更: *executor* 必须是*ThreadPoolExecutor* 的实例。

错误处理 API

允许自定义事件循环中如何去处理异常。

`loop.`**`set_exception_handler`**(*handler*)

将 *handler* 设置为新的事件循环异常处理器。

如果 *handler* 为 `None`，将设置默认的异常处理程序。在其他情况下，*handler* 必须是一个可调用对象且签名匹配 (`loop, context`)，其中 `loop` 是对活动事件循环的引用，而 `context` 是一个包含异常详情的 `dict` (请查看*call_exception_handler()* 文档来获取关于上下文的更多信息)。

如果针对一个*Task* 或*Handle* 调用了该异常处理器，它将在相应任务或回调句柄的*contextvars.Context* 中运行。

在 3.12 版本发生变更: 该处理器可能会在发生异常的任务或句柄的*Context* 中被调用。

`loop.`**`get_exception_handler`**()

返回当前的异常处理器，如果没有设置异常处理器，则返回 `None` 。

Added in version 3.5.2.

`loop.`**`default_exception_handler`**(*context*)

默认的异常处理器。

此方法会在发生异常且未设置异常处理程序时被调用。此方法也可以由想要具有不同于默认处理程序的行为的自定义异常处理程序来调用。

context 参数和*call_exception_handler()* 中的同名参数完全相同。

loop.**call_exception_handler**(*context*)

调用当前事件循环的异常处理器。

context 是个包含下列键的 dict 对象 (未来版本的 Python 可能会引入新键):

- 'message': 错误消息;
- 'exception' （可选）: 异常对象;
- 'future' （可选）: *asyncio.Future* 实例;
- 'task' (可选): *asyncio.Task* 实例;
- 'handle' （可选）: *asyncio.Handle* 实例;
- 'protocol' （可选）: *Protocol* 实例;
- 'transport' （可选）: *Transport* 实例;
- 'socket' (可选): *socket.socket* 实例;
- 'source_traceback' (可选): 源的回溯;
- 'handle_traceback' (可选): 句柄的回溯;
- **'asyncgen' (可选): 异步生成器，它导致了**
  这个异常

> **备注**
>
> 此方法不应在子类化的事件循环中被重载。对于自定义的异常处理，请使用*set_exception_handler()* 方法。

开启调试模式

loop.**get_debug**()

获取事件循环调试模式设置（*bool*）。

如果环境变量 PYTHONASYNCIODEBUG 是一个非空字符串，就返回 True ，否则就返回 False 。

loop.**set_debug**(*enabled:* bool)

设置事件循环的调试模式。

在 3.7 版本发生变更: 现在也可以通过新的*Python 开发模式* 来启用调试模式。

loop.**slow_callback_duration**

该属性可用于设置会被视为“缓慢”的以秒数表示的最短执行时间。当启用调试模式时，“缓慢”的回调将被记录到日志。

默认值为 100 毫秒。

> **参见**
>
> *debug mode of asyncio*.

运行子进程

本小节所描述的方法都是低层级的。在常规 async/await 代码中请考虑改用高层级的*asyncio.create_subprocess_shell()* 和*asyncio.create_subprocess_exec()* 便捷函数。

备注

在 Windows 中，默认的事件循环*ProactorEventLoop* 支持子进程，而*SelectorEventLoop* 不支持。参见*Windows 中的子进程支持* 。

async loop.**subprocess_exec**(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

用 *args* 指定的一个或者多个字符串型参数创建一个子进程。

args 必须是个由下列形式的字符串组成的列表：

- *str*；
- 或者由*文件系统编码格式* 编码的*bytes*。

第一个字符串指定可执行程序，其余的字符串指定其参数。所有字符串参数共同组成了程序的 argv。

此方法类似于调用标准库*subprocess.Popen* 类，设置 shell=False 并将字符串列表作为第一个参数传入；但是，*Popen* 只接受一个单独的字符串列表参数，而 *subprocess_exec* 接受多个字符串参数。

protocol_factory 必须为一个返回*asyncio.SubprocessProtocol* 类的子类的可调用对象。

其他参数：

- *stdin* 可以是以下对象之一：
  - 一个文件型对象
  - 一个现有的文件描述符（一个正整数），例如用*os.pipe()* 创建的文件描述符
  - *subprocess.PIPE* 常量（默认），将创建并连接一个新的管道。
  - None 值，这将使得子进程继承来自此进程的文件描述符
  - *subprocess.DEVNULL* 常量，这表示将使用特殊的*os.devnull* 文件
- *stdout* 可以是以下对象之一：
  - 一个文件型对象
  - *subprocess.PIPE* 常量（默认），将创建并连接一个新的管道。
  - None 值，这将使得子进程继承来自此进程的文件描述符
  - *subprocess.DEVNULL* 常量，这表示将使用特殊的*os.devnull* 文件
- *stderr* 可以是以下对象之一：
  - 一个文件型对象
  - *subprocess.PIPE* 常量（默认），将创建并连接一个新的管道。
  - None 值，这将使得子进程继承来自此进程的文件描述符
  - *subprocess.DEVNULL* 常量，这表示将使用特殊的*os.devnull* 文件
  - *subprocess.STDOUT* 常量，将把标准错误流连接到进程的标准输出流
- 所有其他关键字参数会被不加解释地传给*subprocess.Popen*，除了 *bufsize*, *universal_newlines*, *shell*, *text*, *encoding* 和 *errors*，它们都不应当被指定。

  asyncio 子进程 API 不支持将流解码为文本。可以使用*bytes.decode()* 来将从流返回的字节串转换为文本。

如果作为 *stdin*, *stdout* 或 *stderr* 传入的文件型对象是代表一个管道，则该管道的另一端应当用*connect_write_pipe()* 或*connect_read_pipe()* 来注册以配合事件循环使用。

其他参数的文档，请参阅*subprocess.Popen* 类的构造函数。

返回一对 (transport, protocol)，其中 *transport* 来自*asyncio.SubprocessTransport* 基类而 *protocol* 是由 *protocol_factory* 所实例化的对象。

async loop.**subprocess_shell**(*protocol_factory*, *cmd*, *, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

基于 *cmd* 创建一个子进程，该参数可以是一个*str* 或者按*文件系统编码格式* 编码得到的*bytes*，使用平台的"shell" 语法。

这类似与用 shell=True 调用标准库的*subprocess.Popen* 类。

protocol_factory 必须为一个返回*SubprocessProtocol* 类的子类的可调用对象。

请参阅*subprocess_exec()* 了解有关其余参数的详情。

返回一对 (transport, protocol)，其中 *transport* 来自*SubprocessTransport* 基类而 *protocol* 是由 *protocol_factory* 所实例化的对象。

备注

应用程序要负责确保正确地转义所有空白字符和特殊字符以防止 shell 注入 漏洞。*shlex.quote()* 函数可以被用来正确地转义字符串中可能被用来构造 shell 命令的空白字符和特殊字符。

回调处理

class asyncio.**Handle**

由*loop.call_soon()*, *loop.call_soon_threadsafe()* 所返回的回调包装器对象。

get_context()

返回关联到该句柄的*contextvars.Context* 对象。

Added in version 3.12.

cancel()

取消回调。如果此回调已被取消或已被执行，此方法将没有任何效果。

cancelled()

如果此回调已被取消则返回 True。

Added in version 3.7.

class asyncio.**TimerHandle**

由*loop.call_later()* 和*loop.call_at()* 所返回的回调包装器对象。

这个类是*Handle* 的子类。

when()

返回加入计划任务的回调时间，以*float* 值表示的秒数。

时间值是一个绝对时间戳，使用与*loop.time()* 相同的时间引用。

Added in version 3.7.

Server 对象

Server 对象可使用*loop.create_server()*, *loop.create_unix_server()*, *start_server()* 和*start_unix_server()* 等函数来创建。

请不要直接实例化*Server* 类。

class asyncio.**Server**

Server 对象是异步上下文管理器。当用于 async with 语句时，异步上下文管理器可以确保 Server 对象被关闭，并且在 async with 语句完成后，不接受新的连接。

```
srv = await loop.create_server(...)

async with srv:
    # 一些代码

# 此时，srv 已关闭并不再接受新的连接。
```

在 3.7 版本发生变更: Python3.7 开始，Server 对象是一个异步上下文管理器。

在 3.11 版本发生变更: 这个类在 Python 3.9.11, 3.10.3 和 3.11 中作为 `asyncio.Server` 对外公开。

close()

停止服务：关闭监听的套接字并且设置*sockets* 属性为 `None` 。

用于表示已经连进来的客户端连接会保持打开的状态。

服务器是被异步关闭的；使用*wait_closed()* 协程来等待服务器关闭（将不再有激活的连接）。

close_clients()

关闭所有的外来客户端连接。

在所有关联的传输上调用*close()*。

当关闭服务器时*close()* 应当在*close_clients()* 之前被调用以避免与新的客户端连接发生竞争。

Added in version 3.13.

abort_clients()

立即关闭所有当前的外来客户端连接，无需等待挂起的操作完成。

在所有关联的传输上调用*abort()*。

当关闭服务器时*close()* 应当在*abort_clients()* 之前被调用以避免与新的服务器连接发生竞争。

Added in version 3.13.

get_loop()

返回与服务器对象相关联的事件循环。

Added in version 3.7.

async start_serving()

开始接受连接。

此方法是幂等的，所以它可在服务器已在运行时被调用。

传给*loop.create_server()* 和*asyncio.start_server()* 的 *start_serving* 仅限关键字形参允许创建不接受初始连接的 Server 对象。在此情况下可以使用 `Server.start_serving()` 或*Server.serve_forever()* 让 Server 对象开始接受连接。

Added in version 3.7.

async serve_forever()

开始接受连接，直到协程被取消。`serve_forever` 任务的取消将导致服务器被关闭。

如果服务器已经在接受连接了，这个方法可以被调用。每个 *Server* 对象，仅能有一个 `serve_forever` 任务。

示例:

```
async def client_connected(reader, writer):
    # 使用 reader/writer 流与客户端通信。 例如:
    await reader.readline()

async def main(host, port):
```

(续下页)

(接上页)

```
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

Added in version 3.7.

is_serving()

如果服务器正在接受新连接的状态，返回 `True` 。

Added in version 3.7.

async wait_closed()

等待直到*close()* 方法完成且所有活动的连接结束。

sockets

由类套接字对象组成的列表 `asyncio.trsock.TransportSocket`，服务器将监听这些对象。

在 3.7 版本发生变更: 在 Python 3.7 之前 `Server.sockets` 会直接返回内部的服务器套接字列表。在 3.7 版则会返回该列表的副本。

事件循环实现

asyncio 带有两种不同的事件循环实现: *SelectorEventLoop* 和*ProactorEventLoop*。

在默认情况下 asyncio 将被配置为使用*EventLoop*。

class asyncio.**SelectorEventLoop**

基于*selectors* 模块的*AbstractEventLoop* 的子类。

使用给定平台中最高效的可用 *selector*。也可以手动配置要使用的特定 selector:

```
import asyncio
import selectors

class MyPolicy(asyncio.DefaultEventLoopPolicy):
   def new_event_loop(self):
      selector = selectors.SelectSelector()
      return asyncio.SelectorEventLoop(selector)

asyncio.set_event_loop_policy(MyPolicy())
```

Availability: Unix, Windows.

class asyncio.**ProactorEventLoop**

使用"I/O Completion Ports" (IOCP) 的针对 Windows 的*AbstractEventLoop* 子类。

Availability: Windows.

> **参见**
>
> 有关 I/O 完成端口的 MSDN 文档。

class asyncio.**EventLoop**

在给定平台上可用的最高效的*AbstractEventLoop* 子类的别名。

它在 Unix 上是*SelectorEventLoop* 的别名而在 Windows 上是*ProactorEventLoop* 的别名。

Added in version 3.13.

class asyncio.**AbstractEventLoop**

asyncio 兼容事件循环的抽象基类。

事件循环方法集 一节列出了 AbstractEventLoop 的替代实现应当要定义的所有方法。

例子

请注意本节中的所有示例都 **有意地**演示了如何使用低层级的事件循环 API，例如*loop.run_forever()* 和*loop.call_soon()*。现代的 asyncio 应用很少需要以这样的方式编写；请考虑使用高层级的函数例如*asyncio.run()*。

call_soon() 的 Hello World 示例。

一个使用*loop.call_soon()* 方法来安排回调的示例。回调会显示 "Hello World" 然后停止事件循环:

```
import asyncio

def hello_world(loop):
    """打印 'Hello World' 并停止事件循环的回调"""
    print('Hello World')
    loop.stop()

loop = asyncio.new_event_loop()

# 将对 hello_world() 的调用加入计划任务
loop.call_soon(hello_world, loop)

# 阻塞调用被 loop.stop() 中断
try:
    loop.run_forever()
finally:
    loop.close()
```

参见

一个类似的*Hello World* 示例，使用协程和*run()* 函数创建。

使用 call_later() 来展示当前的日期

一个每秒刷新显示当前日期的示例。回调使用*loop.call_later()* 方法在 5 秒后将自身重新加入计划日程，然后停止事件循环:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.new_event_loop()

# 将对 display_date() 的第一次调用加入计划任务
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# 被 loop.stop() 中断的阻塞调用
```

(续下页)

(接上页)

```
try:
    loop.run_forever()
finally:
    loop.close()
```

参见

一个类似的*current date* 示例，使用协程和`run()` 函数创建。

监控一个文件描述符的读事件

使用`loop.add_reader()` 方法，等到文件描述符收到一些数据，然后关闭事件循环：

```
import asyncio
from socket import socketpair

# 创建一对已连接的文件描述符
rsock, wsock = socketpair()

loop = asyncio.new_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # 已完成：注销文件描述符
    loop.remove_reader(rsock)

    # 停止事件循环
    loop.stop()

# 为读取事件注册文件描述符
loop.add_reader(rsock, reader)

# 模拟从网络接收数据
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # 运行事件循环
    loop.run_forever()
finally:
    # 已完成。 关闭套接字和事件循环。
    rsock.close()
    wsock.close()
    loop.close()
```

参见

- 一个类似的示例，使用传输、协议和`loop.create_connection()` 方法创建。
- 另一个类似的示例，使用了高层级的`asyncio.open_connection()` 函数和流。

为 SIGINT 和 SIGTERM 设置信号处理器

（这个 `signals` 示例只适用于 Unix。）

使用`loop.add_signal_handler()` 方法为信号`SIGINT` 和`SIGTERM` 注册处理器：

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

19.1.9 Futures

源代码: Lib/asyncio/futures.py, Lib/asyncio/base_futures.py

Future 对象用来链接 **底层回调式代码**和高层异步/等待式代码。

Future 函数

asyncio.**isfuture**(*obj*)

如果 *obj* 为下面任意对象，返回 `True`：

- 一个*asyncio.Future* 类的实例，
- 一个*asyncio.Task* 类的实例，
- 带有 `_asyncio_future_blocking` 属性的类似 Future 的对象。

Added in version 3.5.

asyncio.**ensure_future**(*obj*, *, *loop=None*)

返回：

- *obj* 参数会是保持原样，如果 *obj* 是*Future*、*Task* 或类似 Future 的对象 (*isfuture()* 用于测试。)
- 封装了 *obj* 的*Task* 对象，如果 *obj* 是一个协程 (使用*iscoroutine()* 进行检测)；在此情况下该协程将通过 `ensure_future()` 加入执行计划。
- 等待 *obj* 的*Task* 对象，如果 *obj* 是一个可等待对象 (*inspect.isawaitable()* 用于测试)

如果 *obj* 不是上述对象会引发一个*TypeError* 异常。

> **重要**
>
> 查看*create_task()* 函数，它是创建新任务的首选途径。
>
> 保存一个指向此函数的结果的引用，以避免任务在执行期间消失。

在 3.5.1 版本发生变更: 这个函数接受任意*awaitable* 对象。

自 3.10 版本弃用: 如果 *obj* 不是 Future 类对象同时未指定 *loop* 并且没有正在运行的事件循环则会发出弃用警告。

asyncio.**wrap_future**(*future*, *, *loop=None*)

将一个*concurrent.futures.Future* 对象封装到*asyncio.Future* 对象中。

自 3.10 版本弃用: 如果 *future* 不是 Future 类对象同时未指定 *loop* 并且没有正在运行的事件循环则会发出弃用警告。

Future 对象

class asyncio.**Future**(*, *loop=None*)

一个 Future 代表一个异步运算的最终结果。线程不安全。

Future 是一个*awaitable* 对象。协程可以等待 Future 对象直到它们有结果或设置了异常，或者直到它们被取消。一个 Future 可被等待多次并且结果相同。

通常 Future 用于支持底层回调式代码 (例如在协议实现中使用 asyncio *transports*) 与高层异步/等待式代码交互。

经验告诉我们永远不要面向用户的接口暴露 Future 对象，同时建议使用*loop.create_future()* 来创建 Future 对象。这种方法可以让 Future 对象使用其它的事件循环实现，它可以注入自己的优化实现。

在 3.7 版本发生变更: 加入对*contextvars* 模块的支持。

自 3.10 版本弃用: 如果未指定 *loop* 并且没有正在运行的事件循环则会发出弃用警告。

result()

返回 Future 的结果。

如果 Future 状态为 完成，并由*set_result()* 方法设置一个结果，则返回这个结果。

如果 Future 状态为 完成，并由*set_exception()* 方法设置一个异常，那么这个方法会引发异常。

如果 Future 已 取消，方法会引发一个*CancelledError* 异常。

如果 Future 的结果还不可用，此方法会引发一个*InvalidStateError* 异常。

set_result(*result*)

将 Future 标记为 完成并设置结果。

如果 Future 已经 完成则抛出一个*InvalidStateError* 错误。

set_exception(*exception*)

将 Future 标记为 完成并设置一个异常。

如果 Future 已经 完成则抛出一个*InvalidStateError* 错误。

done()

如果 Future 为已 完成则返回 `True` 。

如果 Future 为 取消或调用*set_result()* 设置了结果或调用*set_exception()* 设置了异常，那么它就是 完成。

cancelled()

如果 Future 已 取消则返回 `True`

这个方法通常在设置结果或异常前用来检查 Future 是否已 取消。

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)

添加一个在 Future 完成时运行的回调函数。

调用 *callback* 时，Future 对象是它的唯一参数。

如果调用这个方法时 Future 已经 完成，回调函数会被*loop.call_soon()* 调度。

可选键值类的参数 *context* 允许 *callback* 运行在一个指定的自定义*contextvars.Context* 对象中。如果没有提供 *context* ，则使用当前上下文。

可以用*functools.partial()* 给回调函数传递参数，例如:

```
# 当 "fut" 已完成时则调用 'print("Future:", fut)'。
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

在 3.7 版本发生变更: 加入键值类形参 *context*。请参阅 **PEP 567** 查看更多细节。

remove_done_callback(*callback*)

从回调列表中移除 *callback* 。

返回被移除的回调函数的数量，通常为 1，除非一个回调函数被添加多次。

cancel(*msg=None*)

取消 Future 并调度回调函数。

如果 Future 已经 完成或 取消，返回 `False` 。否则将 Future 状态改为 取消并在调度回调函数后返回 `True` 。

在 3.9 版本发生变更: 增加了 *msg* 形参。

exception()

返回 Future 已设置的异常。

只有 Future 在 完成时才返回异常（或者 `None` ，如果没有设置异常）。

如果 Future 已 取消，方法会引发一个*CancelledError* 异常。

如果 Future 还没 完成，这个方法会引发一个*InvalidStateError* 异常。

get_loop()

返回 Future 对象已绑定的事件循环。

Added in version 3.7.

这个例子创建一个 Future 对象，创建和调度一个异步任务去设置 Future 结果，然后等待其结果:

```
async def set_after(fut, delay, value):
    # 休眠 *delay* 秒。
    await asyncio.sleep(delay)

    # 将 *value* 设为 Future 对象 *fut* 的结果。
    fut.set_result(value)

async def main():
    # 获取当前事件循环。
    loop = asyncio.get_running_loop()

    # 新建一个 Future 对象。
    fut = loop.create_future()

    # 在一个并行的任务中运行 "set_after()" 协程。
    # 在这里我们使用低层级的 "loop.create_task()" API
    # 因为我们手头已经拥有一个对事件循环的引用。
    # 在其他情况下我们可以使用 "asyncio.create_task()"。
    loop.create_task(
```

(续下页)

(接上页)

```
        set_after(fut, 1, '... world'))

    print('hello ...')

    # 等待直到 *fut* 得出结果 (1 秒) 并打印它。
    print(await fut)

asyncio.run(main())
```

重要

该 Future 对象是为了模仿*concurrent.futures.Future* 类。主要差异包含：

- 与 asyncio 的 Future 不同，*concurrent.futures.Future* 实例不是可等待对象。
- *asyncio.Future.result()* 和*asyncio.Future.exception()* 不接受 *timeout* 参数。
- Future 没有完成时*asyncio.Future.result()* 和*asyncio.Future.exception()* 抛出一个*InvalidStateError* 异常。
- 使用*asyncio.Future.add_done_callback()* 注册的回调函数不会立即调用，而是被*loop.call_soon()* 调度。
- asyncio Future 不能兼容*concurrent.futures.wait()* 和*concurrent.futures.as_completed()* 函数。
- *asyncio.Future.cancel()* 接受一个可选的 `msg` 参数，但*concurrent.futures.Future.cancel()* 无此参数。

19.1.10 传输和协议

前言

传输和协议会被像*loop.create_connection()* 这类 **底层**事件循环接口使用。它们使用基于回调的编程风格支持网络或 IPC 协议（如 HTTP）的高性能实现。

基本上，传输和协议应只在库和框架上使用，而不应该在高层的异步应用中使用它们。

本文档包含*Transports* 和*Protocols* 。

概述

在最顶层，传输只关心 **怎样**传送字节内容，而协议决定传送 **哪些**字节内容 (还要在一定程度上考虑何时)。

也可以这样说：从传输的角度来看，传输是套接字 (或类似的 I/O 终端) 的抽象，而协议是应用程序的抽象。

换另一种说法，传输和协议一起定义网络 I/0 和进程间 I/O 的抽象接口。

传输对象和协议对象总是一对一关系：协议调用传输方法来发送数据，而传输在接收到数据时调用协议方法传递数据。

大部分面向连接的事件循环方法 (如*loop.create_connection()*) 通常接受 *protocol_factory* 参数为接收到的链接创建 协议对象，并用 传输对象来表示。这些方法一般会返回 `(transport, protocol)` 元组。

目录

本文档包含下列小节:

- 传输 部分记载异步 IO *BaseTransport* 、*ReadTransport* 、*WriteTransport* 、*Transport* 、*DatagramTransport* 和*SubprocessTransport* 等类。
- The *Protocols* section documents asyncio *BaseProtocol*, *Protocol*, *BufferedProtocol*, *DatagramProtocol*, and *SubprocessProtocol* classes.

- 例子 部分展示怎样使用传输、协议和底层事件循环接口。

传输

源码: Lib/asyncio/transports.py

传输属于*asyncio*模块中的类，用来抽象各种通信通道。

传输对象总是由异步 *IO* 事件循环 实例化。

异步 IO 实现 TCP、UDP、SSL 和子进程管道的传输。传输上可用的方法由传输的类型决定。

传输类属于线程不安全 。

传输层级

class asyncio.**BaseTransport**

所有传输的基类。包含所有异步 IO 传输共用的方法。

class asyncio.**WriteTransport**(*BaseTransport*)

只写链接的基础传输。

WriteTransport 类的实例由*loop.connect_write_pipe()* 事件循环方法返回，也被子进程相关的方法如*loop.subprocess_exec()* 使用。

class asyncio.**ReadTransport**(*BaseTransport*)

只读链接的基础传输。

ReadTransport 类的实例由*loop.connect_read_pipe()* 事件循环方法返回，也被子进程相关的方法如*loop.subprocess_exec()* 使用。

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)

接口代表一个双向传输，如 TCP 链接。

用户不用直接实例化传输；调用一个功能函数，给它传递协议工厂和其它需要的信息就可以创建传输和协议。

传输类实例由如*loop.create_connection()* 、*loop.create_unix_connection()* 、*loop.create_server()* 、*loop.sendfile()* 等这类事件循环方法使用或返回。

class asyncio.**DatagramTransport**(*BaseTransport*)

数据报 (UDP) 传输链接。

DatagramTransport 类实例由事件循环方法*loop.create_datagram_endpoint()* 返回。

class asyncio.**SubprocessTransport**(*BaseTransport*)

表示父进程和子进程之间连接的抽象。

SubprocessTransport 类的实例由事件循环方法*loop.subprocess_shell()* 和*loop.subprocess_exec()* 返回。

基础传输

BaseTransport.**close**()

关闭传输。

如果传输具有外发数据缓冲区，已缓存的数据将被异步地发送。之后将不会再接收更多数据。在所有已缓存的数据被发送之后，协议的*protocol.connection_lost()* 方法将被调用并以*None* 作为其参数。在传输关闭后它就不应再被使用。

BaseTransport.**is_closing**()

返回 True ，如果传输正在关闭或已经关闭。

BaseTransport.**get_extra_info**(*name*, *default=None*)

返回传输或它使用的相关资源信息。

name 是表示要获取传输特定信息的字符串。

default 是在信息不可用或传输不支持第三方事件循环实现或当前平台查询时返回的值。

例如下面代码尝试获取传输相关套接字对象:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

传输可查询信息类别:

- 套接字:
  - 'peername': 套接字链接时的远端地址，*socket.socket.getpeername()* 方法的结果(出错时为 None)
  - 'socket': *socket.socket* 实例
  - 'sockname': 套接字本地地址，*socket.socket.getsockname()* 方法的结果
- SSL 套接字
  - 'compression': 用字符串指定压缩算法，或者链接没有压缩时为 None；*ssl.SSLSocket.compression()* 的结果。
  - 'cipher': 一个三值的元组，包含使用密码的名称，定义使用的 SSL 协议的版本，使用的加密位数。*ssl.SSLSocket.cipher()* 的结果。
  - 'peercert': 远端凭证；*ssl.SSLSocket.getpeercert()* 结果。
  - 'sslcontext': *ssl.SSLContext* 实例
  - 'ssl_object': *ssl.SSLObject* 或*ssl.SSLSocket* 实例
- 管道:
  - 'pipe': 管道对象
- 子进程:
  - 'subprocess': *subprocess.Popen* 实例

BaseTransport.**set_protocol**(*protocol*)

设置一个新协议。

只有两种协议都写明支持切换才能完成切换协议。

BaseTransport.**get_protocol**()

返回当前协议。

只读传输

ReadTransport.**is_reading**()

如果传输接收到新数据时返回 True 。

Added in version 3.7.

ReadTransport.**pause_reading**()

暂停传输的接收端。*protocol.data_received()* 方法将不会收到数据，除非*resume_reading()* 被调用。

在 3.7 版本发生变更: 这个方法幂等的，它可以在传输已经暂停或关闭时调用。

ReadTransport.**resume_reading**()

恢复接收端。如果有数据可读取时，协议方法*protocol.data_received()* 将再次被调用。

在 3.7 版本发生变更: 这个方法幂等的，它可以在传输已经准备好读取数据时调用。

只写传输

WriteTransport.**abort**()

立即关闭传输，不会等待已提交的操作处理完毕。已缓存的数据将会丢失。不会接收更多数据。最终*None* 将作为协议的*protocol.connection_lost()* 方法的参数被调用。

WriteTransport.**can_write_eof**()

如果传输支持*write_eof()* 返回*True* 否则返回*False* 。

WriteTransport.**get_write_buffer_size**()

返回传输使用输出缓冲区的当前大小。

WriteTransport.**get_write_buffer_limits**()

获取写入流控制 *high* 和 *low* 高低标记位。返回元组 `(low, high)` ，*low* 和 *high* 为正字节数。

使用*set_write_buffer_limits()* 设置限制。

Added in version 3.4.2.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)

设置写入流控制 *high* 和 *low* 高低标记位。

这两个值（以字节数表示）控制何时调用协议的*protocol.pause_writing()* 和*protocol.resume_writing()* 方法。如果指明，则低水位必须小于或等于高水位。*high* 和 *low* 都不能为负值。

pause_writing() 会在缓冲区尺寸大于或等于 *high* 值时被调用。如果写入已经被暂停，*resume_writing()* 会在缓冲区尺寸小于或等于 *low* 值时被调用。

默认值是实现专属的。如果只给出了高水位值，则低水位值默认为一个小于或等于高水位值的实现传属值。将 *high* 设为零会强制将 *low* 也设为零，并使得*pause_writing()* 在缓冲区变为非空的任何时刻被调用。将 *low* 设为零会使得*resume_writing()* 在缓冲区为空时只被调用一次。对于上下限都使用零值通常是不够优化的，因为它减少了并发执行 I/O 和计算的机会。

可使用*get_write_buffer_limits()* 来获取上下限值。

WriteTransport.**write**(*data*)

将一些 *data* 字节串写入传输。

此方法不会阻塞；它会缓冲数据并安排其被异步地发出。

WriteTransport.**writelines**(*list_of_data*)

将数据字节串的列表（或任意可迭代对象）写入传输。这在功能上等价于在可迭代对象产生的每个元素上调用*write()*，但其实现可能更为高效。

WriteTransport.**write_eof**()

在刷新所有已缓冲数据之后关闭传输的写入端。数据仍可以被接收。

如果传输（例如 SSL）不支持半关闭的连接，此方法会引发*NotImplementedError*。

数据报传输

DatagramTransport.**sendto**(*data*, *addr=None*)

将 *data* 字节串发送到 *addr* (基于传输的目标地址) 所给定的远端对等方。如果 *addr* 为*None*，则将数据发送到传输创建时给定的目标地址。

此方法不会阻塞；它会缓冲数据并安排其被异步地发出。

在 3.13 版本发生变更: 调用此方法时可以传入一个空字节串对象来发送零长度的数据报。用于流量控制的缓冲区大小计算也会被更新以计入数据报的标头。

DatagramTransport.**abort**()

立即关闭传输，不会等待已提交的操作执行完毕。已缓存的数据将会丢失。不会接收更多的数据。协议的*protocol.connection_lost()* 方法最终将附带*None* 作为参数被调用。

子进程传输

SubprocessTransport.**get_pid**()

将子进程的进程 ID 以整数形式返回。

SubprocessTransport.**get_pipe_transport**(*fd*)

返回对应于整数文件描述符 *fd* 的通信管道的传输:

- 0: 标准输入 (*stdin*) 的可读流式传输，如果子进程创建时未设置 stdin=PIPE 则为*None*
- 1: 标准输出 (*stdout*) 的可写流式传输，如果子进程创建时未设置 stdout=PIPE 则为*None*
- 2: 标准错误 (*stderr*) 的可写流式传输，如果子进程创建时未设置 stderr=PIPE 则为*None*
- 其他 *fd*: *None*

SubprocessTransport.**get_returncode**()

返回整数形式的进程返回码，或者如果还未返回则为*None*，这类似于*subprocess.Popen.returncode* 属性。

SubprocessTransport.**kill**()

杀死子进程。

在 POSIX 系统中，函数会发送 SIGKILL 到子进程。在 Windows 中，此方法是*terminate()* 的别名。

另请参见*subprocess.Popen.kill()*。

SubprocessTransport.**send_signal**(*signal*)

发送 *signal* 编号到子进程，与*subprocess.Popen.send_signal()* 一样。

SubprocessTransport.**terminate**()

停止子进程。

在 POSIX 系统中，此方法会发送*SIGTERM* 到子进程。在 Windows 中，则会调用 Windows API 函数 TerminateProcess() 来停止子进程。

另请参见*subprocess.Popen.terminate()*。

SubprocessTransport.**close**()

通过调用*kill()* 方法来杀死子进程。

如果子进程尚未返回，并关闭 *stdin*, *stdout* 和 *stderr* 管道的传输。

协议

源码: Lib/asyncio/protocols.py

---

asyncio 提供了一组抽象基类，它们应当被用于实现网络协议。这些类被设计为与*传输* 配合使用。

抽象基础协议类的子类可以实现其中的部分或全部方法。所有这些方法都是回调：它们由传输或特定事件调用，例如当数据被接收的时候。基础协议方法应当由相应的传输来调用。

基础协议

class asyncio.**BaseProtocol**

带有所有协议的共享方法的基础协议。

class asyncio.**Protocol**(*BaseProtocol*)

用于实现流式协议（TCP, Unix 套接字等等）的基类。

class asyncio.**BufferedProtocol**(*BaseProtocol*)

用于实现可对接收缓冲区进行手动控制的流式协议的基类。

class asyncio.**DatagramProtocol**(*BaseProtocol*)

用于实现数据报（UDP）协议的基类。

class asyncio.**SubprocessProtocol**(*BaseProtocol*)

用于实现与子进程通信（单向管道）的协议的基类。

基础协议

所有 asyncio 协议均可实现基础协议回调。

连接回调

连接回调会在所有协议上被调用，每个成功的连接将恰好调用一次。所有其他协议回调只能在以下两个方法之间被调用。

BaseProtocol.**connection_made**(*transport*)

连接建立时被调用。

transport 参数是代表连接的传输。此协议负责将引用保存至对应的传输。

BaseProtocol.**connection_lost**(*exc*)

连接丢失或关闭时将被调用。

方法的参数是一个异常对象或为*None*。后者意味着收到了常规的 EOF，或者连接被连接的一端取消或关闭。

流程控制回调

流程控制回调可由传输来调用以暂停或恢复协议所执行的写入操作。

请查看*set_write_buffer_limits()* 方法的文档了解详情。

BaseProtocol.**pause_writing**()

当传输的缓冲区升至高水位以上时将被调用。

BaseProtocol.**resume_writing**()

当传输的缓冲区降到低水位以下时将被调用。

如果缓冲区大小等于高水位值，则*pause_writing()* 不会被调用：缓冲区大小必须要高于该值。

相反地，*resume_writing()* 会在缓冲区大小等于或小于低水位值时被调用。这些结束条件对于当两个水位取零值时也能确保符合预期的行为是很重要的。

流式协议

事件方法，例如*loop.create_server()*，*loop.create_unix_server()*，*loop.create_connection()*，*loop.create_unix_connection()*，*loop.connect_accepted_socket()*，*loop.connect_read_pipe()* 和*loop.connect_write_pipe()* 都接受返回流式协议的工厂。

Protocol.**data_received**(*data*)

当收到数据时被调用。*data* 为包含入站数据的非空字节串对象。

数据是否会被缓冲、分块或重组取决于具体传输。通常，你不应依赖于特定的语义而应使你的解析具有通用性和灵活性。但是，数据总是要以正确的顺序被接收。

此方法在连接打开期间可以被调用任意次数。

但是，*protocol.eof_received()* 最多只会被调用一次。一旦 `eof_received()` 被调用，`data_received()` 就不会再被调用。

Protocol.**eof_received**()

当发出信号的另一端不再继续发送数据时（例如通过调用*transport.write_eof()*，如果另一端也使用 asyncio 的话）被调用。

此方法可能返回假值 (包括 None)，在此情况下传输将会自行关闭。相反地，如果此方法返回真值，将以所用的协议来确定是否要关闭传输。由于默认实现是返回 None，因此它会隐式地关闭连接。

某些传输，包括 SSL 在内，并不支持半关闭的连接，在此情况下从该方法返回真值将导致连接被关闭。

状态机：

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

缓冲流协议

Added in version 3.7.

带缓冲的协议可与任何支持流式协议 的事件循环方法配合使用。

BufferedProtocol 实现允许显式手动分配和控制接收缓冲区。随后事件循环可以使用协议提供的缓冲区来避免不必要的数据复制。这对于接收大量数据的协议来说会有明显的性能提升。复杂的协议实现能显著地减少缓冲区分配的数量。

以下回调是在*BufferedProtocol* 实例上被调用的:

BufferedProtocol.**get_buffer**(*sizehint*)

调用后会分配新的接收缓冲区。

sizehint 是推荐的返回缓冲区最小尺寸。返回小于或大于 *sizehint* 推荐尺寸的缓冲区也是可接受的。当设为 -1 时，缓冲区尺寸可以是任意的。返回尺寸为零的缓冲区则是错误的。

get_buffer() 必须返回一个实现了 缓冲区协议的对象。

BufferedProtocol.**buffer_updated**(*nbytes*)

用接收的数据更新缓冲区时被调用。

nbytes 是被写入到缓冲区的字节总数。

BufferedProtocol.**eof_received**()

请查看*protocol.eof_received()* 方法的文档。

在连接期间*get_buffer()* 可以被调用任意次数。但是，*protocol.eof_received()* 最多只能被调用一次，如果被调用，则在此之后*get_buffer()* 和*buffer_updated()* 不能再被调用。

状态机：

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

数据报协议

数据报协议实例应当由传递给*loop.create_datagram_endpoint()* 方法的协议工厂来构造。

DatagramProtocol.**datagram_received**(*data*, *addr*)

当接收到数据报时被调用。*data* 是包含传入数据的字节串对象。*addr* 是发送数据的对等端地址；实际的格式取决于具体传输。

DatagramProtocol.**error_received**(*exc*)

当前一个发送或接收操作引发*OSError* 时被调用。*exc* 是*OSError* 的实例。

此方法会在当传输（例如 UDP）检测到无法将数据报传给接收方等极少数情况下被调用。而在大多数情况下，无法送达的数据报将被静默地丢弃。

备注

在 BSD 系统（macOS, FreeBSD 等等）上，数据报协议不支持流控制，因为没有可靠的方式来检测因写入多过包所导致的发送失败。

套接字总是显示为'ready' 且多余的包会被丢弃。有一定的可能性会引发*OSError* 并设置 errno 为*errno.ENOBUFS*；如果此异常被引发，它将被报告给*DatagramProtocol.error_received()*，在其他情况下则会被忽略。

子进程协议

子进程协议实例应当由传递给*loop.subprocess_exec()* 和*loop.subprocess_shell()* 方法的协议工厂函数来构造。

SubprocessProtocol.**pipe_data_received**(*fd*, *data*)

当子进程向其 stdout 或 stderr 管道写入数据时被调用。

fd 是以整数表示的管道文件描述符。

data 是包含已接收数据的非空字节串对象。

SubprocessProtocol.**pipe_connection_lost**(*fd*, *exc*)

与子进程通信的其中一个管道关闭时被调用。

fd 以整数表示的已关闭文件描述符。

SubprocessProtocol.**process_exited**()

子进程退出时被调用。

它可以在*pipe_data_received()* 和*pipe_connection_lost()* 方法之前被调用。

例子

TCP 回显服务器

使用*loop.create_server()* 方法创建 TCP 回显服务器，发回已接收的数据，并关闭连接:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()
```

(续下页)

(接上页)

```
async def main():
    # 获取指向事件循环的引用
    # 因为我们准备使用低层级 API。
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        EchoServerProtocol,
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

参见

使用流的 *TCP* 回显服务器 示例，使用了高层级的*asyncio.start_server()* 函数。

TCP 回显客户端

使用*loop.create_connection()* 方法的 TCP 回显客户端，发送数据并等待，直到连接被关闭:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
    # 获取指向事件循环的引用
    # 因为我们准备使用低层级 API。
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # 等待发出连接丢失的协议信号
    # 并关闭传输。
    try:
```

(续下页)

(接上页)

```
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

参见

使用流的 TCP 回显客户端 示例，使用了高层级的*asyncio.open_connection()* 函数。

UDP 回显服务器

使用*loop.create_datagram_endpoint()* 方法的 UDP 回显服务器，发回已接收的数据:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # 获取对事件循环的引用
    # 因为我们打算使用低层级的 APIs。
    loop = asyncio.get_running_loop()

    # 将创建一个协议实例来为所有客户端请求提供服务。
    transport, protocol = await loop.create_datagram_endpoint(
        EchoServerProtocol,
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # 服务持续 1 小时。
    finally:
        transport.close()


asyncio.run(main())
```

UDP 回显客户端

使用*loop.create_datagram_endpoint()* 方法的 UDP 回显客户端，发送数据并在收到回应时关闭传输:

```
import asyncio


class EchoClientProtocol:
```

(续下页)

(接上页)

```
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # 获取一个对事件循环的引用
    # 因为我们计划使用低层级的 API。
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

链接已存在的套接字

附带一个协议使用*loop.create_connection()* 方法，等待直到套接字接收数据:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport
```

(续下页)

(接上页)

```
    def data_received(self, data):
        print("Received:", data.decode())

        # 已完成：关闭传输；
        # connection_lost() 将自动被调用。
        self.transport.close()

    def connection_lost(self, exc):
        # 套接字已被关闭
        self.on_con_lost.set_result(True)


async def main():
    # 获取指向事件循环的引用
    # 因为我们准备使用低层级 API。
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # 创建一对已连接的套接字
    rsock, wsock = socket.socketpair()

    # 注册套接字以等待数据。
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # 模拟从网络接收数据。
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

参见

使用低层级的*loop.add_reader()* 方法来注册一个 FD 的监视文件描述符以读取事件 示例。

使用在协程中通过*open_connection()* 函数创建的高层级流的注册一个打开的套接字以等待使用流的数据 示例。

loop.subprocess_exec() 与 SubprocessProtocol

一个使用子进程协议来获取子进程的输出并等待子进程退出的示例。

这个子进程是由*loop.subprocess_exec()* 方法创建的:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()
        self.pipe_closed = False
        self.exited = False
```

(续下页)

(接上页)

```
    def pipe_connection_lost(self, fd, exc):
        self.pipe_closed = True
        self.check_for_exit()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exited = True
        # process_exited() 方法可以在
        # pipe_connection_lost() 方法之前被调用:
        # 等待直到两个方法都已被调用。
        self.check_for_exit()

    def check_for_exit(self):
        if self.pipe_closed and self.exited:
            self.exit_future.set_result(True)

async def get_date():
    # 获取对事件循环的引用因为我们打算使用
    # 低层级 API。
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # 创建由 DateProtocol 控制的子进程;
    # 重定向标准输出到一个管道。
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # 使用协议的 process_exited() 方法
    # 等待子进程退出。
    await exit_future

    # 关闭 stdout 管道。
    transport.close()

    # 读取由协议的 pipe_data_received() 方法所收集的
    # 输出。
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

另请参阅使用高层级 API 编写的相同示例。

19.1.11 策略

事件循环策略是一个用于获取和设置当前事件循环 的全局对象，还可以创建新的事件循环。默认策略可以可以被替换 为内置替代策略 以使用不同的事件循环实现，或者替换为可以覆盖这些行为的自定义策略。

策略对象 可为每个 *context* 获取和设置单独的事件循环。在默认情况下是分线程，不过自定义策略可以按不同的方式定义 *context*。

自定义事件循环策略可以控制*get_event_loop()*, *set_event_loop()* 和*new_event_loop()* 的行为。

策略对象应该实现*AbstractEventLoopPolicy* 抽象基类中定义的 API。

获取和设置策略

可以使用下面函数获取和设置当前进程的策略:

asyncio.**get_event_loop_policy**()
返回当前进程域的策略。

asyncio.**set_event_loop_policy**(*policy*)
将 *policy* 设置为当前进程域策略。

如果 *policy* 设为 None 将恢复默认策略。

策略对象

抽象事件循环策略基类定义如下:

class asyncio.**AbstractEventLoopPolicy**
异步策略的抽象基类。

get_event_loop()
为当前上下文获取事件循环。

返回一个实现*AbstractEventLoop* 接口的事件循环对象。

该方法永远不应返回 None。

在 3.6 版本发生变更.

set_event_loop(*loop*)
将当前上下文的事件循环设置为 *loop* 。

new_event_loop()
创建并返回一个新的事件循环对象。

该方法永远不应返回 None。

get_child_watcher()
获取子进程监视器对象。

返回一个实现*AbstractChildWatcher* 接口的监视器对象。

该函数仅支持 Unix。

自 3.12 版本弃用.

set_child_watcher(*watcher*)
将当前子进程监视器设置为 *watcher* 。

该函数仅支持 Unix。

自 3.12 版本弃用.

asyncio 附带下列内置策略:

class asyncio.**DefaultEventLoopPolicy**
默认的 asyncio 策略。在 Unix 上使用*SelectorEventLoop* 而在 Windows 上使用*ProactorEventLoop*。

不需要手动安装默认策略。asyncio 已配置成自动使用默认策略。

在 3.8 版本发生变更: 在 Windows 上，现在默认会使用*ProactorEventLoop*。

自 3.12 版本弃用: 现在默认 asyncio 策略的*get_event_loop()* 方法将在没有正在运行的事件循环而决定创建一个事件循环时发出*DeprecationWarning*。在未来的某个 Python 发布版中这将改为发出错误。

class asyncio.**WindowsSelectorEventLoopPolicy**

一个使用*SelectorEventLoop* 事件循环实现的替代事件循环策略。

Availability: Windows.

class asyncio.**WindowsProactorEventLoopPolicy**

使用*ProactorEventLoop* 事件循环实现的另一种事件循环策略。

Availability: Windows.

进程监视器

进程监视器允许定制事件循环如何监视 Unix 子进程。具体来说，事件循环需要知道子进程何时退出。

在 asyncio 中子进程由*create_subprocess_exec()* 和*loop.subprocess_exec()* 函数创建。

asyncio 定义了*AbstractChildWatcher* 抽象基类，子监视器必须要实现它，并具有四种不同实现: *ThreadedChildWatcher* (已配置为默认使用), *MultiLoopChildWatcher*, *SafeChildWatcher* 和*FastChildWatcher*。

请参阅子进程和线程 部分。

以下两个函数可用于自定义子进程监视器实现，它将被 asyncio 事件循环使用:

asyncio.**get_child_watcher**()

返回当前策略的当前子监视器。

自 3.12 版本弃用.

asyncio.**set_child_watcher**(*watcher*)

将当前策略的子监视器设置为 *watcher* 。*watcher* 必须实现*AbstractChildWatcher* 基类定义的方法。

自 3.12 版本弃用.

备注

第三方事件循环实现可能不支持自定义子监视器。对于这样的事件循环，禁止使用*set_child_watcher()* 或不起作用。

class asyncio.**AbstractChildWatcher**

add_child_handler(*pid*, *callback*, **args*)

注册一个新的子处理回调函数。

安排 `callback(pid, returncode, *args)` 在进程的 PID 与 *pid* 相等时调用。指定另一个同进程的回调函数替换之前的回调处理函数。

回调函数 *callback* 必须是线程安全。

remove_child_handler(*pid*)

删除进程 PID 与 *pid* 相等的进程的处理函数。

处理函数成功删除时返回 `True` ，没有删除时返回 `False` 。

attach_loop(*loop*)

给一个事件循环绑定监视器。

如果监视器之前已绑定另一个事件循环，那么在绑定新循环前会先解绑原来的事件循环。

注意：循环有可能是 `None` 。

is_active()

如果监视器已准备好使用则返回 True。

使用 *不活动的*当前子监视器生成子进程将引发*RuntimeError*。

Added in version 3.8.

close()

关闭监视器。

必须调用这个方法以确保相关资源会被清理。

自 3.12 版本弃用.

class asyncio.**ThreadedChildWatcher**

此实现会为每个生成的子进程启动一具新的等待线程。

即使是当 asyncio 事件循环运行在非主 OS 线程上时它也能可靠地工作。

当处理大量子进程时不存在显著的开销 (每次子进程结束时为 *O*(1))，但当每个进程启动一个线程时则需要额外的内存。

此监视器会默认被使用。

Added in version 3.8.

class asyncio.**MultiLoopChildWatcher**

此实现会在实例化时注册一个 SIGCHLD 信号处理程序。这可能会破坏为 SIGCHLD 信号安装自定义处理程序的第三方代码。

此监视器会在收到 SIGCHLD 信号时通过显式地轮询每个进程来避免干扰其他代码生成的进程。

该监视器一旦被安装就不会限制从不同线程运行子进程。

该解决方案是安全的，但在处理大量进程时会有显著的开销 (每收到一个 SIGCHLD 时为 *O*(*n*))。

Added in version 3.8.

自 3.12 版本弃用.

class asyncio.**SafeChildWatcher**

该实现会使用主线程中的活动事件循环来处理 SIGCHLD 信号。如果主线程没有正在运行的事件循环，则其他线程无法生成子进程 (会引发*RuntimeError*)。

此监视器会在收到 SIGCHLD 信号时通过显式地轮询每个进程来避免干扰其他代码生成的进程。

该解决方案与*MultiLoopChildWatcher* 一样安全并具有相同的 *O*(*n*) 复杂度，但需要主线程有正在运行的事件循环才能工作。

自 3.12 版本弃用.

class asyncio.**FastChildWatcher**

这种实现直接调用 os.waitpid(-1) 来获取所有已结束的进程，可能会中断其它代码🄵生进程并等待它们结束。

在处理大量子进程时没有明显的开销 (每次子进程结束时为 *O*(1))。

该解决方案需要主线程有正在运行的事件循环才能工作，这与*SafeChildWatcher* 一样。

自 3.12 版本弃用.

class asyncio.**PidfdChildWatcher**

这个实现会轮询处理文件描述符 (pidfds) 以等待子进程终结。在某些方面，*PidfdChildWatcher* 是一个“理想的”子进程监视器实现。它不需要使用信号或线程，不会介入任何在事件循环以外发起的进程，并能随事件循环发起的子进程数量进行线性伸缩。其主要缺点在于 pidfds 是 Linux 专属的，并且仅在较近版本的核心（5.3+）上可用。

Added in version 3.9.

自定义策略

要实现一个新的事件循环策略，建议子类化*DefaultEventLoopPolicy* 并重写需要定制行为的方法，例如:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """获取事件循环。

        这可能为 None 或是一个 EventLoop 的实例。
        """
        loop = super().get_event_loop()
        # 对 loop 执行一些操作 ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

19.1.12 平台支持

asyncio 模块被设计为可移植的, 但由于平台的底层架构和功能，一些平台存在细微的差异和限制。

所有平台

- *loop.add_reader()* 和*loop.add_writer()* 不能用来监视文件 I/O。

Windows

源代码: Lib/asyncio/proactor_events.py, Lib/asyncio/windows_events.py, Lib/asyncio/windows_utils.py

---

在 3.8 版本发生变更: 在 Windows 上，*ProactorEventLoop* 现在是默认的事件循环。

Windows 上的所有事件循环都不支持以下方法:

- 不支持*loop.create_unix_connection()* 和*loop.create_unix_server()*。*socket.AF_UNIX* 套接字族是 Unix 专属的。
- 不支持*loop.add_signal_handler()* 和*loop.remove_signal_handler()* 。

SelectorEventLoop 有下列限制:

- *SelectSelector* 只被用于等待套接字事件：它支持套接字且最多支持 512 个套接字。
- *loop.add_reader()* 和*loop.add_writer()* 只接受套接字处理回调函数 (如管道、文件描述符等都不支持)。
- 因为不支持管道，所以*loop.connect_read_pipe()* 和*loop.connect_write_pipe()* 方法没有实现。
- 不支持*Subprocesses* ，也就是*loop.subprocess_exec()* 和*loop.subprocess_shell()* 方法没有实现。

ProactorEventLoop 有下列限制:

- 不支持*loop.add_reader()* 和*loop.add_writer()* 方法。

通常 Windows 上单调时钟的分辨率约为 15.6 毫秒。最佳分辨率是 0.5 毫秒。分辨率依赖于具体的硬件 (HPET 的可用性) 和 Windows 的设置。

Windows 的子进程支持

在 Windows 上，默认的事件循环*ProactorEventLoop* 支持子进程，而*SelectorEventLoop* 则不支持。

也不支持*policy.set_child_watcher()* 函数，*ProactorEventLoop* 有不同的机制来监视子进程。

macOS

完整支持流行的 macOS 版本。

macOS <= 10.8

在 macOS 10.6, 10.7 和 10.8 上，默认的事件循环使用*selectors.KqueueSelector*，在这些版本上它并不支持字符设备。可以手工配置*SelectorEventLoop* 来使用*SelectSelector* 或*PollSelector* 以在这些较老版本的 macOS 上支持字符设备。例如:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

19.1.13 扩展

asyncio 扩展的主要方向是编写自定义的 事件循环类。asyncio 具有可以被用来简化此任务的辅助工具。

> **备注**
>
> 第三方应当小心谨慎地重用现有的异步代码，新的 Python 版本可以自由地打破 API 的 内部部分的向下兼容性。

编写自定义事件循环

asyncio.AbstractEventLoop 声明了大量的方法。从头开始全部实现它们将是一件烦琐的工作。

一个事件循环可以通过从 `asyncio.BaseEventLoop` 继承来自动地获得许多常用方法的实现。

相应地，继承者应当实现多个在 `asyncio.BaseEventLoop` 中已声明但未实现的 私有方法。

例如，`loop.create_connection()` 会检查参数，解析 DNS 地址，并调用应当由继承方类来实现的 `loop._make_socket_transport()`。`_make_socket_transport()` 方法未被写入文档并被视为 内部 API。

Future 和 Task 私有构造器

asyncio.Future 和*asyncio.Task* 不应该被直接实例化，请使用对应的*loop.create_future()*，*loop.create_task()* 或*asyncio.create_task()* 工厂函数。

但是，第三方 事件循环可能会 重用内置的 Future 和 Task 实现以自动获得复杂且高度优化的代码。

出于这个目的，下面列出了相应的 私有构造器:

`Future.`**`__init__`**(*, *loop=None*)

创建一个内置的 Future 实例。

loop 是一个可选的事件循环实例。

`Task.`**`__init__`**(*coro*, *, *loop=None*, *name=None*, *context=None*)

创建一个内置的 Task 实例。

loop 是一个可选的事件循环实例。其余参数会在*loop.create_task()* 说明中加以描述。

在 3.11 版本发生变更: 添加了 *context* 参数。

Task 生命周期支持

第三方任务实现应当调用下列函数以使任务对*asyncio.all_tasks()* 和*asyncio.current_task()* 可见:

asyncio.**_register_task**(*task*)

注册一个新的 *task* 并由 *asyncio* 管理。

调用来自任务构造器的函数。

asyncio.**_unregister_task**(*task*)

从 *asyncio* 内置结构体中注销 *task*。

此函数应当在任务将要结束时被调用。

asyncio.**_enter_task**(*loop*, *task*)

将当前任务切换为 *task* 参数。

在执行嵌入的 *coroutine* (`coroutine.send()` 或 `coroutine.throw()`) 的一部分之前调用此函数。

asyncio.**_leave_task**(*loop*, *task*)

将当前任务从 *task* 切换回 `None`。

在 `coroutine.send()` 或 `coroutine.throw()` 执行之后调用此函数。

19.1.14 高层级 API 索引

这个页面列举了所有能用于 async/wait 的高层级 asyncio API 集。

任务

运行异步程序，创建 Task 对象，等待多件事运行超时的公共集。

run()	创建事件循环，运行一个协程，关闭事件循环。
Runner	一个能够简化多次异步函数调用操作的上下文管理器。
Task	Task 对象
TaskGroup	持有一组任务的上下文管理器。它提供了一种等待分组中所有任务完成的方便可靠的方式。
create_task()	启动一个异步 Task，然后将其返回。
current_task()	返回当前 Task 对象
all_tasks()	返回一个事件循环的所有尚未完成的任务。
await *sleep()*	休眠几秒。
await *gather()*	并发执行所有事件的调度和等待。
await *wait_for()*	有超时控制的运行。
await *shield()*	屏蔽取消操作
await *wait()*	完成情况的监控器
timeout()	设置超时运行。在 `wait_for` 不适合的情况下会很有用。
to_thread()	在不同的 OS 线程中异步地运行一个函数。
run_coroutine_threadsafe()	从其他 OS 线程中调度一个协程。
for in *as_completed()*	用 `for` 循环监控完成情况。

例子

- 使用 *asyncio.gather()* 并行运行.
- 使用 *asyncio.wait_for()* 强制超时.
- 撤销协程.
- *asyncio.sleep()* 的用法.

- 请主要参阅协程与任务文档.

队列集

队列集被用于多个异步 Task 对象的运行调度，实现连接池以及发布/订阅模式。

Queue	先进先出队列
PriorityQueue	优先级队列。
LifoQueue	后进先出队列。

例子

- 使用 *asyncio.Queue* 在多个并发任务间分配工作量.
- 请参阅队列集文档.

子进程集

用于生成子进程和运行 shell 命令的工具包。

await *create_subprocess_exec()*	创建一个子进程。
await *create_subprocess_shell()*	运行一个 shell 命令。

例子

- 执行一个 *shell* 命令.
- 请参阅子进程 *APIs* 相关文档.

流

用于网络 IO 处理的高级 API 集。

await *open_connection()*	建立一个 TCP 连接。
await *open_unix_connection()*	建立一个 Unix socket 连接。
await *start_server()*	启动 TCP 服务。
await *start_unix_server()*	启动一个 Unix 套接字服务。
StreamReader	接收网络数据的高级 async/await 对象。
StreamWriter	发送网络数据的高级 async/await 对象。

例子

- *TCP* 客户端样例.
- 请参阅*streams APIs* 文档。

同步

能被用于 Task 对象集的，类似线程的同步基元组件。

Lock	互斥锁。
Event	事件对象。
Condition	条件对象
Semaphore	信号量
BoundedSemaphore	有界的信号量。
Barrier	一个 Barrier 对象。

例子

- *asyncio.Event 的用法*.
- *使用 asyncio.Barrier*。
- 请参阅 asyncio 文档*synchronization primitives*.

异常

asyncio.CancelledError	当一个 Task 对象被取消的时候被引发。请参阅*Task.cancel()*。
asyncio.BrokenBarrierError	当一个 Barrier 对象被破坏时引发。另请参阅*Barrier.wait()*。

例子

- *在取消请求发生的运行代码中如何处理 CancelledError 异常*.
- 请参阅完整的*asyncio 专用异常* 列表.

19.1.15 低层级 API 索引

本页列出所有低层级的 asyncio API。

获取事件循环

asyncio.get_running_loop()	获取当前运行的事件循环 **首选**函数。
asyncio.get_event_loop()	获取一个事件循环实例（正在运行的事件循环或通过当前策略确定的当前事件循环）。
asyncio.set_event_loop()	通过当前策略将事件循环设置当前事件循环。
asyncio.new_event_loop()	创建一个新的事件循环。

例子

- *使用 asyncio.get_running_loop()*。

事件循环方法集

另请参阅有关*事件循环方法集* 的主文档章节。

生命周期

loop.run_until_complete()	运行一个期程/任务/可等待对象直到完成。
loop.run_forever()	一直运行事件循环。
loop.stop()	停止事件循环。
loop.close()	关闭事件循环。
loop.is_running()	返回 `True`，如果事件循环正在运行。
loop.is_closed()	返回 `True`，如果事件循环已经被关闭。
`await` *loop.shutdown_asyncgens()*	关闭异步生成器。

调试

loop.set_debug()	开启或禁用调试模式。
loop.get_debug()	获取当前测试模式。

调度回调函数

loop.call_soon()	尽快调用回调。
loop.call_soon_threadsafe()	*loop.call_soon()* 方法线程安全的变体。
loop.call_later()	在给定时间 之后调用回调函数。
loop.call_at()	在 指定时间调用回调函数。

线程/进程池

await *loop.run_in_executor()*	在*concurrent.futures* 执行器中运行一个独占 CPU 或其它阻塞函数。
loop.set_default_executor()	设置*loop.run_in_executor()* 默认执行器。

任务与期程

loop.create_future()	创建一个*Future* 对象。
loop.create_task()	将协程当作*Task* 一样调度。
loop.set_task_factory()	设置*loop.create_task()* 使用的工厂，它将用来创建*Tasks* 。
loop.get_task_factory()	获取*loop.create_task()* 使用的工厂，它用来创建*Tasks* 。

DNS

await *loop.getaddrinfo()*	异步版的*socket.getaddrinfo()* 。
await *loop.getnameinfo()*	异步版的*socket.getnameinfo()* 。

网络和 IPC

await *loop.create_connection()*	打开一个 TCP 链接。
await *loop.create_server()*	创建一个 TCP 服务。
await *loop.create_unix_connection()*	打开一个 Unix socket 连接。
await *loop.create_unix_server()*	创建一个 Unix socket 服务。
await *loop.connect_accepted_socket()*	将*socket* 包装成 `(transport, protocol)` 对。
await *loop.create_datagram_endpoint()*	打开一个数据报 (UDP) 连接。
await *loop.sendfile()*	通过传输通道发送一个文件。
await *loop.start_tls()*	将一个已建立的链接升级到 TLS。
await *loop.connect_read_pipe()*	将管道读取端包装成 `(transport, protocol)` 对。
await *loop.connect_write_pipe()*	将管道写入端包装成 `(transport, protocol)` 对。

套接字

await *loop.sock_recv()*	从*socket* 接收数据。
await *loop.sock_recv_into()*	从*socket* 接收数据到一个缓冲区中。
await *loop.sock_recvfrom()*	从*socket* 接收数据报。
await *loop.sock_recvfrom_into()*	从*socket* 接收数据报并放入缓冲区。
await *loop.sock_sendall()*	发送数据到*socket* 。
await *loop.sock_sendto()*	通过*socket* 向给定的地址发送数据报。
await *loop.sock_connect()*	链接 await *loop.sock_connect()* 。
await *loop.sock_accept()*	接受一个*socket* 链接。
await *loop.sock_sendfile()*	利用*socket* 发送一个文件。
loop.add_reader()	开始对一个文件描述符的可读性的监视。
loop.remove_reader()	停止对一个文件描述符的可读性的监视。
loop.add_writer()	开始对一个文件描述符的可写性的监视。
loop.remove_writer()	停止对一个文件描述符的可写性的监视。

Unix 信号

loop.add_signal_handler()	给*signal* 添加一个处理回调函数。
loop.remove_signal_handler()	删除*signal* 的处理回调函数。

子进程集

loop.subprocess_exec()	衍生一个子进程
loop.subprocess_shell()	从终端命令衍生一个子进程。

错误处理

loop.call_exception_handler()	调用异常处理器。
loop.set_exception_handler()	设置一个新的异常处理器。
loop.get_exception_handler()	获取当前异常处理器。
loop.default_exception_handler()	默认异常处理器实现。

例子

- 使用 *asyncio.new_event_loop()* 和 *loop.run_forever()*.
- 使用 *loop.call_later()*.
- 使用 `loop.create_connection()` 实现*echo* 客户端.
- 使用 `loop.create_connection()` 去链接 *socket*.
- 使用 *add_reader()* 监听 *FD*(文件描述符) 的读取事件.
- 使用 *loop.add_signal_handler()*.
- 使用 *loop.add_signal_handler()*。

传输

所有传输都实现以下方法:

transport.close()	关闭传输。
transport.is_closing()	返回 `True`，如果传输正在关闭或已经关闭。
transport.get_extra_info()	请求传输的相关信息。
transport.set_protocol()	设置一个新协议。
transport.get_protocol()	返回当前协议。

传输可以接收数据 (TCP 和 Unix 链接，管道等)。它通过*loop.create_connection()*，*loop.create_unix_connection()*，*loop.connect_read_pipe()* 等方法返回。

读取传输

transport.is_reading()	返回 `True`，如果传输正在接收。
transport.pause_reading()	暂停接收。
transport.resume_reading()	继续接收。

传输可以发送数据 (TCP 和 Unix 链接，管道等)。它通过*loop.create_connection()*，*loop.create_unix_connection()*，*loop.connect_write_pipe()* 等方法返回。

写入传输

transport.write()	向传输写入数据。
transport.write()	向传输写入缓冲。
transport.can_write_eof()	返回*True*，如果传输支持发送 EOF。
transport.write_eof()	在冲洗已缓冲的数据后关闭传输和发送 EOF。
transport.abort()	立即关闭传输。
transport.get_write_buffer_size()	返回当前输出缓冲区的大小。
transport.get_write_buffer_limits()	返回写入流控制的高位标记位和低位标记位。
transport.set_write_buffer_limits()	设置新的写入流控制的高位标记位和低位标记位。

由*loop.create_datagram_endpoint()* 返回的传输:

数据报传输

transport.sendto()	发送数据到远程链接端。
transport.abort()	立即关闭传输。

基于子进程的底层抽象传输，它由*loop.subprocess_exec()* 和*loop.subprocess_shell()* 返回:

子进程传输

transport.get_pid()	返回子进程的进程 ID。
transport.get_pipe_transport()	返回请求通信管道 (*stdin*, *stdout*, 或 *stderr*) 的传输。
transport.get_returncode()	返回子进程的返回代号。
transport.kill()	杀死子进程。
transport.send_signal()	发送一个信号到子进程。
transport.terminate()	停止子进程。
transport.close()	杀死子进程并关闭所有管道。

协议

协议类可以由下面 **回调方法**实现：

callback *connection_made()*	连接建立时被调用。
callback *connection_lost()*	连接丢失或关闭时将被调用。
callback *pause_writing()*	传输的缓冲区超过高位标记位时被调用。
callback *resume_writing()*	传输的缓冲区传送到低位标记位时被调用。

流协议 (TCP, Unix 套接字, 管道)

callback *data_received()*	接收到数据时被调用。
callback *eof_received()*	接收到 EOF 时被调用。

缓冲流协议

callback *get_buffer()*	调用后会分配新的接收缓冲区。
callback *buffer_updated()*	用接收的数据更新缓冲区时被调用。
callback *eof_received()*	接收到 EOF 时被调用。

数据报协议

callback *datagram_received()*	接收到数据报时被调用。
callback *error_received()*	前一个发送或接收操作引发*OSError* 时被调用。

子进程协议

callback *pipe_data_received()*	子进程向 *stdout* 或 *stderr* 管道写入数据时被调用。
callback *pipe_connection_lost()*	与子进程通信的其中一个管道关闭时被调用。
callback *process_exited()*	当子进程退出时被调用。可在*pipe_data_received()* 和*pipe_connection_lost()* 方法之前被调用。

事件循环策略

策略是改变*asyncio.get_event_loop()* 这类函数行为的一个底层机制。更多细节可以查阅策略部分。

访问策略

asyncio.get_event_loop_policy()	返回当前进程域的策略。
asyncio.set_event_loop_policy()	设置一个新的进程域策略。
AbstractEventLoopPolicy	策略对象的基类。

19.1.16 用 asyncio 开发

异步编程与传统的“顺序”编程不同。

本页列出常见的错误和陷阱，并解释如何避免它们。

Debug 模式

默认情况下，asyncio 以生产模式运行。为了简化开发，asyncio 还有一种 *debug 模式 * 。

有几种方法可以启用异步调试模式:

- 将 PYTHONASYNCIODEBUG 环境变量设置为 1 。
- 使用*Python 开发模式*。
- 将 debug=True 传递给*asyncio.run()* 。
- 调用*loop.set_debug()* 。

除了启用调试模式外，还要考虑:

- 将*asyncio logger* 的级别设为*logging.DEBUG*，例如下面的代码片段可以在应用程序启动时运行:

```
logging.basicConfig(level=logging.DEBUG)
```

- 配置*warnings* 模块以显示*ResourceWarning* 警告。一种方法是使用 -W default 命令行选项。

启用调试模式时:

- asyncio 检查*未被等待的协程* 并记录他们；这将消除“被遗忘的等待”问题。
- 许多非线程安全的异步 APIs (例如*loop.call_soon()* 和*loop.call_at()* 方法)，如果从错误的线程调用，则会引发异常。
- 如果执行 I/O 操作花费的时间太长，则记录 I/O 选择器的执行时间。
- 执行时间超过 100 毫秒的回调会被写入日志。*loop.slow_callback_duration* 属性可用于设置以秒为单位的将被视为“缓慢”的最小执行持续时间。

并发性和多线程

事件循环在线程中运行 (通常是主线程)，并在其线程中执行所有回调和任务。当一个任务在事件循环中运行时，没有其他任务可以在同一个线程中运行。当一个任务执行一个 await 表达式时，正在运行的任务被挂起，事件循环执行下一个任务。

要调度来自另一 OS 线程的*callback*，应该使用*loop.call_soon_threadsafe()* 方法。例如:

```
loop.call_soon_threadsafe(callback, *args)
```

几乎所有异步对象都不是线程安全的，这通常不是问题，除非在任务或回调函数之外有代码可以使用它们。如果需要这样的代码来调用低级异步 API，应该使用*loop.call_soon_threadsafe()* 方法，例如:

```
loop.call_soon_threadsafe(fut.cancel)
```

要从不同的 OS 线程调度一个协程对象，应该使用*run_coroutine_threadsafe()* 函数。它返回一个*concurrent.futures.Future* 。查询结果:

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# 随后在另一个 OS 线程中:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# 等待结果:
result = future.result()
```

为了能处理信号事件循环必须在主线程中运行。

方法*loop.run_in_executor()* 可以和*concurrent.futures.ThreadPoolExecutor* 一起使用，用于在一个不同的操作系统线程中执行阻塞代码，并避免阻塞运行事件循环的那个操作系统线程。

目前没有办法直接从另一个进程 (如使用*multiprocessing* 启动的进程) 安排协程或回调。事件循环方法集 小节列出了一些可以从管道读取并监视文件描述符而不会阻塞事件循环的 API。此外，asyncio 的子进程

API 提供了一种启动进程并在事件循环中与其通信的办法。最后，之前提到的*loop.run_in_executor()* 方法也可以配合*concurrent.futures.ProcessPoolExecutor* 使用以在另一个进程中执行代码。

运行阻塞的代码

不应该直接调用阻塞 (CPU 绑定) 代码。例如，如果一个函数执行 1 秒的 CPU 密集型计算，那么所有并发异步任务和 IO 操作都将延迟 1 秒。

可以用执行器在不同的线程甚至不同的进程中运行任务，以避免使用事件循环阻塞 OS 线程。请参阅*loop.run_in_executor()* 方法了解详情。

日志记录

asyncio 使用*logging* 模块，所有日志记录都是通过 "asyncio" logger 执行的。

默认的日志级别是*logging.INFO*，它可以被方便地调整:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

网络日志会阻塞事件循环。建议使用一个单独进程来处理日志或者使用非阻塞式 IO。例如，可以参阅 blocking-handlers。

检测 never-awaited 协同程序

当协程函数被调用而不是被等待时 (即执行 `coro()` 而不是 `await coro()`) 或者协程没有通过*asyncio.create_task()* 被排入计划日程，asyncio 将会发出一条*RuntimeWarning*:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

输出:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

调试模式的输出:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

通常的修复方法是等待协程或者调用*asyncio.create_task()* 函数:

```
async def main():
    await test()
```

检测就再也没异常

如果调用*Future.set_exception()*，但不等待 Future 对象，将异常传播到用户代码。在这种情况下，当 Future 对象被垃圾收集时，asyncio 将发出一条日志消息。

未处理异常的例子:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

输出:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

激活调试模式 以获取任务创建处的跟踪信息:

```
asyncio.run(main(), debug=True)
```

调试模式的输出:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

备注

asyncio 的源代码可以在 Lib/asyncio/ 中找到。

19.2 socket --- 低层级的网络接口

源代码: Lib/socket.py

这个模块提供了访问 BSD 套接字的接口。在所有现代 Unix 系统、Windows、macOS 和其他一些平台上可用。

备注

一些行为可能因平台不同而异，因为调用的是操作系统的套接字 API。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

这个 Python 接口是将 Unix 系统调用和套接字库接口直接转写为 Python 的面向对象风格：函数`socket()`返回一个 *套接字对象*，其方法是对各种套接字系统调用的实现。形参类型相比 C 接口更高级一些：如同在 Python 文件上的 `read()` 和 `write()` 操作那样，接受操作的缓冲区分配是自动进行的，发送操作的缓冲区长度则是隐式的。

参见

模块*socketserver*
用于简化网络服务端编写的类。

模块*ssl*
套接字对象的 TLS/SSL 封装。

19.2.1 套接字协议族

根据系统以及构建选项，此模块提供了各种套接字协议簇。

特定的套接字对象需要的地址格式将根据此套接字对象被创建时指定的地址族被自动选择。套接字地址表示如下：

- 一个绑定在文件系统节点上的`AF_UNIX`套接字的地址表示为一个字符串，使用文件系统字符编码和 `'surrogateescape'` 错误回调方法（see **PEP 383**）。一个地址在 Linux 的抽象命名空间被返回为带有初始的 null 字节的*字节类对象*；注意在这个命名空间种的套接字可能与普通文件系统套接字通信，所以打算运行在 Linux 上的程序可能需要解决两种地址类型。当传递为参数时，一个字符串或字节类对象可以用于任一类型的地址。

  在 3.3 版本发生变更: 之前，`AF_UNIX` 套接字路径被假设使用 UTF-8 编码。

  在 3.5 版本发生变更: 现在接受可写的*字节类对象*。

- 一对 `(host, port)` 被用作`AF_INET` 地址族，其中 *host* 是一个表示互联网域名标记形式的主机名例如 `'daring.cwi.nl'` 或者 IPv4 地址例如 `'100.50.200.5'` 的字符串，而 *port* 是一个整数值。

  - 对于 IPv4 地址，有两种可接受的特殊形式被用来代替一个主机地址：`''` 代表 `INADDR_ANY`，用来绑定到所有接口；字符串 `'<broadcast>'` 代表 `INADDR_BROADCAST`。此行为不兼容 IPv6，因此，如果你的 Python 程序打算支持 IPv6，则可能需要避开这些。

- 对于`AF_INET6` 地址族，使用一个四元组 `(host, port, flowinfo, scope_id)`，其中 *flowinfo* 和 *scope_id* 代表了 C 库 `struct sockaddr_in6` 中的 `sin6_flowinfo` 和 `sin6_scope_id` 成员。对于`socket` 模块中的方法，*flowinfo* 和 *scope_id* 可以被省略，只为了向后兼容。注意，省略 *scope_id* 可能会导致操作带有领域 (Scope) 的 IPv6 地址时出错。

  在 3.7 版本发生变更: 对于多播地址（其 *scope_id* 起作用），地址中可以不包含 `%scope_id`（或 `zone id`）部分，这部分是多余的，可以放心省略（推荐）。

- `AF_NETLINK` 套接字由一对 `(pid, groups)` 表示。

- 指定 `AF_TIPC` 地址族可以使用仅 Linux 支持的 TIPC 协议。TIPC 是一种开放的、非基于 IP 的网络协议，旨在用于集群计算环境。其地址用元组表示，其中的字段取决于地址类型。一般元组形式为 `(addr_type, v1, v2, v3 [, scope])`，其中：

  - *addr_type* 取 `TIPC_ADDR_NAMESEQ`、`TIPC_ADDR_NAME` 或 `TIPC_ADDR_ID` 中的一个。

  - *scope* 取 `TIPC_ZONE_SCOPE`、`TIPC_CLUSTER_SCOPE` 和 `TIPC_NODE_SCOPE` 中的一个。

  - 如果 *addr_type* 为 TIPC_ADDR_NAME，那么 *v1* 是服务器类型，*v2* 是端口标识符，*v3* 应为 0。

    如果 *addr_type* 为 TIPC_ADDR_NAMESEQ，那么 *v1* 是服务器类型，*v2* 是端口号下限，而 *v3* 是端口号上限。

    如果 *addr_type* 为 TIPC_ADDR_ID，那么 *v1* 是节点 (node)，*v2* 是 ref，*v3* 应为 0。

- *AF_CAN* 地址族使用元组 (interface,)，其中 *interface* 是表示网络接口名称的字符串，如 'can0'。网络接口名 '' 可以用于接收本族所有网络接口的数据包。
  - *CAN_ISOTP* 协议接受一个元组 (interface, rx_addr, tx_addr)，其中两个额外参数都是无符号长整数，都表示 CAN 标识符（标准或扩展标识符）。
  - *CAN_J1939* 协议接受一个元组 (interface, name, pgn, addr)，其中额外参数有：表示 ECU 名称的 64 位无符号整数，表示参数组号 (Parameter Group Number, PGN) 的 32 位无符号整数，以及表示地址的 8 位整数。
- PF_SYSTEM 协议族的 SYSPROTO_CONTROL 协议使用一个字符串或元组 (id, unit)。这个字符串是使用动态分配 ID 的内核控件名称。如果 ID 和内核控件的单元编号都已知或者使用了已注册的 ID 则可以使用元组。

  Added in version 3.3.

- AF_BLUETOOTH 支持以下协议和地址格式：
  - BTPROTO_L2CAP 接受 (bdaddr, psm)，其中 bdaddr 为字符串格式的蓝牙地址，psm 是一个整数。
  - BTPROTO_RFCOMM 接受 (bdaddr, channel)，其中 bdaddr 为字符串格式的蓝牙地址，channel 是一个整数。
  - BTPROTO_HCI 将接受依赖于具体操作系统的格式。
    * 在 Linux 上它接受一个元组 (device_id,) 其中 device_id 是一个指明 Bluetooth 设备号的整数。
    * 在 FreeBSD, NetBSD 和 DragonFly BSD 上它接受 bdaddr 其中 bdaddr 是字符串形式的蓝牙地址。

    在 3.2 版本发生变更: 添加了对 NetBSD 和 DragonFlyBSD 的支持。

    在 3.13.3 版本发生变更: 添加了 FreeBSD 支持。
  - BTPROTO_SCO 接受 bdaddr 其中 bdaddr 是字符串或*bytes* 对象形式的蓝牙地址。(例如 '12:23:34:45:56:67' 或 b'12:23:34:45:56:67') 此协议在 FreeBSD 下不受支持。
- *AF_ALG* 是一个仅 Linux 可用的、基于套接字的接口，用于连接内核加密算法。算法套接字可用包括 2 至 4 个元素的元组来配置 (type, name [, feat [, mask]])，其中：
  - *type* 是表示算法类型的字符串，如 aead、hash、skcipher 或 rng。
  - *name* 是表示算法类型和操作模式的字符串，如 sha256、hmac(sha256)、cbc(aes) 或 drbg_nopr_ctr_aes256。
  - *feat* 和 *mask* 是无符号 32 位整数。

  Availability: Linux >= 2.6.38.

  某些算法类型需要更新的内核。

  Added in version 3.6.

- *AF_VSOCK* 用于支持虚拟机与宿主机之间的通讯。该套接字用 (CID, port) 元组表示，其中 Context ID (CID) 和 port 都是整数。

  Availability: Linux >= 3.9

  参见 *vsock(7)*

  Added in version 3.7.

- *AF_PACKET* 是一个直接连接网络设备的低层级接口。地址以元组 `(ifname, proto[, pkttype[, hatype[, addr]]])` 表示，其中：
  - *ifname* - 指定设备名称的字符串。
  - *proto* - 以太网协议号。可以为*ETH_P_ALL* 表示捕获所有协议，某个*ETHERTYPE_* 常量* 或者任何其他以太网协议号。
  - *pkttype* - 指定数据包类型的整数（可选）：
    - `PACKET_HOST` （默认）- 寻址到本地主机的数据包。
    - `PACKET_BROADCAST` - 物理层广播的数据包。
    - `PACKET_MULTICAST` - 发送到物理层多播地址的数据包。
    - `PACKET_OTHERHOST` - 被（处于混杂模式的）网卡驱动捕获的、发送到其他主机的数据包。
    - `PACKET_OUTGOING` - 来自本地主机的、回环到一个套接字的数据包。
  - *hatype* - 可选整数，指定 ARP 硬件地址类型。
  - *addr* - 可选的类字节串对象，用于指定硬件物理地址，其解释取决于各设备。

  Availability: Linux >= 2.2.
- *AF_QIPCRTR* 是一个仅 Linux 可用的、基于套接字的接口，用于与高通平台中协处理器上运行的服务进行通信。该地址簇用一个 `(node, port)` 元组表示，其中 *node* 和 *port* 为非负整数。

  Availability: Linux >= 4.7.

  Added in version 3.8.
- `IPPROTO_UDPLITE` 是一种 UDP 的变体，允许指定数据包的哪一部分计算入校验码内。它添加了两个可以修改的套接字选项。`self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length)` 修改传出数据包的哪一部分计算入校验码内，而 `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length)` 将过滤掉计算入校验码的数据太少的数据包。在这两种情况下，`length` 都应在 `range(8, 2**16, 8)` 范围内。

  对于 IPv4，应使用 `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE)` 来构造这样的套接字；对于 IPv6，应使用 `socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE)` 来构造这样的套接字。

  Availability: Linux >= 2.6.20, FreeBSD >= 10.1

  Added in version 3.9.
- *AF_HYPERV* 是 Windows 专属的用于同 Hyper-V 主机和客户机通信的基于套接字的接口。其地址族以一个 `(vm_id, service_id)` 元组表示，其中 `vm_id` 和 `service_id` 均为 UUID 字符串。

  `vm_id` 为虚拟机标识号或者如果目标不是一台特定的虚拟机则为已知 VMID 值的集合。在 `socket` 上定义的已知 VMID 常量有：
  - `HV_GUID_ZERO`
  - `HV_GUID_BROADCAST`
  - `HV_GUID_WILDCARD` - 用于绑定自身并接受来自所有分区的连接。
  - `HV_GUID_CHILDREN` - 用于绑定自身并接受来自子分区的连接。
  - `HV_GUID_LOOPBACK` - 用作指向自身的目标。
  - `HV_GUID_PARENT` - 当用作绑定时接受来自父分区的连接。当用作地址目标时它将连接到父分区。will connect to the parent partition.

  `service_id` 是已注册服务的服务标识号。

  Added in version 3.12.

如果你在 IPv4/v6 套接字地址的 *host* 部分中使用了一个主机名，此程序可能会表现不确定行为，因为 Python 使用 DNS 解析返回的第一个地址。套接字地址在实际的 IPv4/v6 中以不同方式解析，根据 DNS 解析和/或 host 配置。为了确定行为，在 *host* 部分中使用数字的地址。

所有错误都会引发异常。普通异常将针对无效的参数类型和内存不足等情况被引发。与套接字或地址语义有关的错误则会引发*OSError* 或它的某个子类。

可以用*setblocking()* 设置非阻塞模式。一个基于超时的 generalization 通过*settimeout()* 支持。

19.2.2 模块内容

socket 模块包含下列元素。

异常

exception socket.**error**

一个被弃用的*OSError* 的别名。

在 3.3 版本发生变更: 根据 **PEP 3151**，这个类是*OSError* 的别名。

exception socket.**herror**

OSError 的子类，本异常通常表示与地址相关的错误，比如那些在 POSIX C API 中使用了 *h_errno* 的函数，包括*gethostbyname_ex()* 和*gethostbyaddr()*。附带的值是一对 `(h_errno, string)`，代表库调用返回的错误。*h_errno* 是一个数字，而 *string* 表示 *h_errno* 的描述，它们由 C 函数 `hstrerror()` 返回。

在 3.3 版本发生变更: 此类是*OSError* 的子类。

exception socket.**gaierror**

OSError 的子类，该异常由*getaddrinfo()* 和*getnameinfo()* 引发以表示与地址相关的错误。附带的值是一个 `(error, string)` 对，代表库调用所返回的错误。*string* 代表 *error* 的描述，如 `gai_strerror()` C 函数所返回的值。数字值 *error* 将与本模块中定义的某个 `EAI_*` 常量相匹配。

在 3.3 版本发生变更: 此类是*OSError* 的子类。

exception socket.**timeout**

TimeoutError 的已被弃用的别名。

OSError 的子类，当套接字发生超时，且事先已调用过*settimeout()*（或隐式地通过*setdefaulttimeout()*）启用了超时，则会抛出此异常。附带的值是一个字符串，其值总是”timed out”。

在 3.3 版本发生变更: 此类是*OSError* 的子类。

在 3.10 版本发生变更: 这个类是*TimeoutError* 的别名。

常量

AF_* 和 SOCK_* 常量现在都在 `AddressFamily` 和 `SocketKind` 这两个*IntEnum* 集合内。

Added in version 3.4.

socket.**AF_UNIX**

socket.**AF_INET**

socket.**AF_INET6**

这些常量表示地址（和协议）族，被用作传给*socket()* 的第一个参数。如果*AF_UNIX* 常量未定义则该协议将不受支持。根据具体系统可能会有更多的常量可用。

socket.**AF_UNSPEC**

AF_UNSPEC 表示*getaddrinfo()* 应当为任何可被使用的地址族返回套接字地址（无论是 IPv4, IPv6 还是其他）。

socket.**SOCK_STREAM**

socket.**SOCK_DGRAM**

socket.**SOCK_RAW**

socket.**SOCK_RDM**

socket.**SOCK_SEQPACKET**

这些常量表示套接字类型，被用作传给*socket()* 的第二个参数。根据具体系统可能会有更多的常量可用。(只有*SOCK_STREAM* 和*SOCK_DGRAM* 是普遍适用的。)

socket.**SOCK_CLOEXEC**

socket.**SOCK_NONBLOCK**

这两个常量（如果已定义）可以与上述套接字类型结合使用，允许你设置这些原子性相关的 flags (从而避免可能的竞争条件和单独调用的需要)。

> **参见**
>
> 安全文件描述符处理 提供了更详尽的解释。

Availability: Linux >= 2.6.27.

Added in version 3.2.

SO_*

socket.**SOMAXCONN**

MSG_*

SOL_*

SCM_*

IPPROTO_*

IPPORT_*

INADDR_*

IP_*

IPV6_*

EAI_*

AI_*

NI_*

TCP_*

许多这样的常量，记录在 Unix 有关套接字和/或 IP 协议的文档中，也在 socket 模块中有定义。它们通常被用于传给套接字对象的*setsockopt()* 和*getsockopt()* 等方法的参数中。在大多数情况下，只有那些在 Unix 头文件中有定义的符号会在本模块中定义；对于部分符号，还提供了默认值。

在 3.6 版本发生变更: 添加了 SO_DOMAIN, SO_PROTOCOL, SO_PEERSEC, SO_PASSSEC, TCP_USER_TIMEOUT, TCP_CONGESTION。

在 3.6.5 版本发生变更: 在 Windows 上，如果 Windows 运行时支持，则 TCP_FASTOPEN、TCP_KEEPCNT 可用。

在 3.7 版本发生变更: 添加了 TCP_NOTSENT_LOWAT。

在 Windows 上，如果 Windows 运行时支持，则 TCP_KEEPIDLE、TCP_KEEPINTVL 可用。

在 3.10 版本发生变更: 添加了 IP_RECVTOS。还添加了 TCP_KEEPALIVE。这个常量在 MacOS 上可以与在 Linux 上使用 TCP_KEEPIDLE 的相同方式被使用。

在 3.11 版本发生变更: 添加了 TCP_CONNECTION_INFO。在 MacOS 上此常量可以与在 Linux 和 BSD 上使用 TCP_INFO 的相同方式来使用。

在 3.12 版本发生变更: 增加了 SO_RTABLE 和 SO_USER_COOKIE。这些常量分别在 OpenBSD 和 FreeBSD 可按与 SO_MARK 在 Linux 上相同的方式被使用。还增加了来自 Linux 的缺失的 TCP 套接字选项: TCP_MD5SIG, TCP_THIN_LINEAR_TIMEOUTS, TCP_THIN_DUPACK, TCP_REPAIR, TCP_REPAIR_QUEUE, TCP_QUEUE_SEQ, TCP_REPAIR_OPTIONS, TCP_TIMESTAMP, TCP_CC_INFO, TCP_SAVE_SYN, TCP_SAVED_SYN, TCP_REPAIR_WINDOW, TCP_FASTOPEN_CONNECT, TCP_ULP, TCP_MD5SIG_EXT, TCP_FASTOPEN_KEY, TCP_FASTOPEN_NO_COOKIE, TCP_ZEROCOPY_RECEIVE,

TCP_INQ, TCP_TX_DELAY。增加了 IP_PKTINFO, IP_UNBLOCK_SOURCE, IP_BLOCK_SOURCE, IP_ADD_SOURCE_MEMBERSHIP, IP_DROP_SOURCE_MEMBERSHIP。

在 3.13 版本发生变更: 增加了 SO_BINDTOIFINDEX。在 Linux 上此常量可按照与 SO_BINDTODEVICE 相同的用法来使用，但是要通过网络接口的索引号而不是其名称。

socket.**AF_CAN**

socket.**PF_CAN**

SOL_CAN_*

CAN_*

此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

Availability: Linux >= 2.6.25, NetBSD >= 8.

Added in version 3.3.

在 3.11 版本发生变更: 添加了 NetBSD 支持。

在 3.13.3 (unreleased) 版本发生变更: 在 Linux 上恢复缺失的 CAN_RAW_ERR_FILTER。

socket.**CAN_BCM**

CAN_BCM_*

CAN 协议簇内的 CAN_BCM 是广播管理器（Bbroadcast Manager -- BCM）协议，广播管理器常量在 Linux 文档中有所记载，在本 socket 模块中也有定义。

Availability: Linux >= 2.6.25.

> **备注**
>
> CAN_BCM_CAN_FD_FRAME 旗标仅在 Linux >= 4.8 时可用。

Added in version 3.4.

socket.**CAN_RAW_FD_FRAMES**

在 CAN_RAW 套接字中启用 CAN FD 支持，默认是禁用的。它使应用程序可以发送 CAN 和 CAN FD 帧。但是，从套接字读取时，也必须同时接受 CAN 和 CAN FD 帧。

此常量在 Linux 文档中有所记载。

Availability: Linux >= 3.6.

Added in version 3.5.

socket.**CAN_RAW_JOIN_FILTERS**

加入已应用的 CAN 过滤器，这样只有与所有 CAN 过滤器匹配的 CAN 帧才能传递到用户空间。

此常量在 Linux 文档中有所记载。

Availability: Linux >= 4.1.

Added in version 3.9.

socket.**CAN_ISOTP**

CAN 协议簇中的 CAN_ISOTP 就是 ISO-TP (ISO 15765-2) 协议。ISO-TP 常量在 Linux 文档中有所记载。

Availability: Linux >= 2.6.25.

Added in version 3.7.

socket.**CAN_J1939**

CAN 协议族中的 CAN_J1939 即 SAE J1939 协议。J1939 常量记录在 Linux 文档中。

Availability: Linux >= 5.4.

Added in version 3.9.

socket.**AF_DIVERT**

socket.**PF_DIVERT**

这两个常量，记录在 FreeBSD divert(4) 手册页中，同样已在 socket 模块中定义。

Availability: FreeBSD >= 14.0.

Added in version 3.12.

socket.**AF_PACKET**

socket.**PF_PACKET**

PACKET_*

此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

Availability: Linux >= 2.2.

socket.**ETH_P_ALL**

ETH_P_ALL 可在*socket* 构造器中用作*AF_PACKET* 族的 *proto* 以便捕获每个包，无论是使用什么协议。

要了解详情，请参阅 *packet(7)* 手册页。

Availability: Linux.

Added in version 3.12.

socket.**AF_RDS**

socket.**PF_RDS**

socket.**SOL_RDS**

RDS_*

此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

Availability: Linux >= 2.6.30.

Added in version 3.3.

socket.**SIO_RCVALL**

socket.**SIO_KEEPALIVE_VALS**

socket.**SIO_LOOPBACK_FAST_PATH**

RCVALL_*

Windows 的 WSAIoctl() 的常量。这些常量用于套接字对象的*ioctl()* 方法的参数。

在 3.6 版本发生变更: 添加了 SIO_LOOPBACK_FAST_PATH。

TIPC_*

TIPC 相关常量，与 C socket API 导出的常量一致。更多信息请参阅 TIPC 文档。

socket.**AF_ALG**

socket.**SOL_ALG**

ALG_*

用于 Linux 内核加密算法的常量。

Availability: Linux >= 2.6.38.

Added in version 3.6.

socket.**AF_VSOCK**

socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**

VMADDR*

SO_VM*

用于 Linux 宿主机/虚拟机通讯的常量。

Availability: Linux >= 4.8.

Added in version 3.7.

socket.**AF_LINK**

Availability: BSD, macOS.

Added in version 3.4.

socket.**has_ipv6**

本常量为一个布尔值，该值指示当前平台是否支持 IPv6。

socket.**BDADDR_ANY**

socket.**BDADDR_LOCAL**

这些是字符串常量，包含蓝牙地址，这些地址具有特殊含义。例如，当用 BTPROTO_RFCOMM 指定绑定套接字时，*BDADDR_ANY* 表示“任何地址”。

socket.**HCI_FILTER**

socket.**HCI_TIME_STAMP**

socket.**HCI_DATA_DIR**

配合 BTPROTO_HCI 使用。HCI_FILTER 仅在 Linux 和 FreeBSD 上可用。HCI_TIME_STAMP 和 HCI_DATA_DIR 仅在 Linux 上可用。

socket.**AF_QIPCRTR**

高通 IPC 路由协议的常数，用于与提供远程处理器的服务进行通信。

Availability: Linux >= 4.7.

socket.**SCM_CREDS2**

socket.**LOCAL_CREDS**

socket.**LOCAL_CREDS_PERSISTENT**

LOCAL_CREDS 和 LOCAL_CREDS_PERSISTENT 可与 SOCK_DGRAM, SOCK_STREAM 套接字一起使用，等价于 Linux/DragonFlyBSD SO_PASSCRED，其中 LOCAL_CREDS 会在首次读取时发送凭证，LOCAL_CREDS_PERSISTENT 会在每次读取时发送，随后必须为后者使用 SCM_CREDS2 作为消息类型。

Added in version 3.11.

Availability: FreeBSD.

socket.**SO_INCOMING_CPU**

用于优化 CPU 定位的常量，应与 SO_REUSEPORT 配合使用。

Added in version 3.11.

Availability: Linux >= 3.9

socket.**AF_HYPERV**

socket.**HV_PROTOCOL_RAW**

socket.**HVSOCKET_CONNECT_TIMEOUT**

socket.**HVSOCKET_CONNECT_TIMEOUT_MAX**

socket.**HVSOCKET_CONNECTED_SUSPEND**

socket.**HVSOCKET_ADDRESS_FLAG_PASSTHRU**

socket.**HV_GUID_ZERO**

socket.**HV_GUID_WILDCARD**

socket.**HV_GUID_BROADCAST**

socket.**HV_GUID_CHILDREN**

socket.**HV_GUID_LOOPBACK**

socket.**HV_GUID_PARENT**

用于 Windows Hyper-V 宿主机/客户机通信的套接字的常量。

Availability: Windows.

Added in version 3.12.

socket.**ETHERTYPE_ARP**

socket.**ETHERTYPE_IP**

socket.**ETHERTYPE_IPV6**

socket.**ETHERTYPE_VLAN**

IEEE 802.3 协议号 常量。

Availability: Linux, FreeBSD, macOS.

Added in version 3.12.

socket.**SHUT_RD**

socket.**SHUT_WR**

socket.**SHUT_RDWR**

这些常量将由套接字对象的*shutdown()* 方法使用。

Availability: not WASI.

函数

创建套接字

下列函数都能创建套接字对象.

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

使用给定的地址族、套接字类型和协议号创建一个新的套接字。地址族应为*AF_INET* (默认值), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET* 或*AF_RDS* 之一。套接字类型应为*SOCK_STREAM* (默认值), *SOCK_DGRAM*, *SOCK_RAW* 或其他可能的 SOCK_ 常量之一。协议号通常为零并且可以省略，或在协议族为*AF_CAN* 的情况下，协议应为 CAN_RAW, *CAN_BCM*, *CAN_ISOTP* 或*CAN_J1939* 之一。

如果指定了 *fileno*，那么将从指定的文件描述符中自动检测 *family*, *type* 和 *proto* 的值。自动检测可被调用此函数时显式传入的 *family*, *type* 或 *proto* 参数所覆盖。这只会影响 Python 表示诸如*socket.getpeername()* 函数的返回值的方式而不会影响实际的 OS 资源。与*socket.fromfd()* 不同，*fileno* 将返回同样的套接字而不是其副本。这将有助于使用*socket.close()* 来关闭已分离的套接字。

新创建的套接字是不可继承的。

引发一个审计事件 socket.__new__ 并附带参数 self, family, type, protocol。

在 3.3 版本发生变更: 添加了 AF_CAN 簇。添加了 AF_RDS 簇。

在 3.4 版本发生变更: 添加了 CAN_BCM 协议。

在 3.4 版本发生变更: 返回的套接字现在是不可继承的。

在 3.7 版本发生变更: 添加了 CAN_ISOTP 协议。

在 3.7 版本发生变更: 当将*SOCK_NONBLOCK* 或*SOCK_CLOEXEC* 旗标位应用于 *type* 时它们将被清除，且*socket.type* 将不会反映它们。它们仍然会被传递给底层的系统 socket() 调用。因而，

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

仍将在支持 SOCK_NONBLOCK 的系统上创建一个非阻塞的套接字，但是 sock.type 会被置为 socket.SOCK_STREAM。

在 3.9 版本发生变更: 添加了 CAN_J1939 协议。

在 3.10 版本发生变更: 添加了 IPPROTO_MPTCP 协议。

socket.**socketpair**([*family*[, *type*[, *proto*]]])

使用给定的地址族、套接字类型和协议号构建一对已连接的套接字对象。地址族、套接字类型和协议号与上述*socket()* 函数中的相同。默认地址族为定义于平台中的*AF_UNIX*；如未定义，则默认为*AF_INET*。

新创建的套接字都是不可继承的。

在 3.2 版本发生变更: 现在，返回的套接字对象支持全部套接字 API，而不是全部 API 的一个子集。

在 3.4 版本发生变更: 返回的套接字现在都是不可继承的。

在 3.5 版本发生变更: 添加了 Windows 支持。

socket.**create_connection**(*address*, *timeout=GLOBAL_DEFAULT*, *source_address=None*, *, *all_errors=False*)

连接到一个在互联网 *address* (以 `(host, port)` 2 元组表示) 上侦听的 TCP 服务，并返回套接字对象。这是一个相比*socket.connect()* 层级更高的函数：如果 *host* 是非数字的主机名，它将尝试将其解析为*AF_INET* 和*AF_INET6*，然后依次尝试连接到所有可能的地址直到连接成功。这使编写兼容 IPv4 和 IPv6 的客户端变得很容易。

传入可选参数 *timeout* 可以在套接字实例上设置超时（在尝试连接前）。如果未提供 *timeout*，则使用由*getdefaulttimeout()* 返回的全局默认超时设置。

如果提供了 *source_address*，它必须为二元组 `(host, port)`，以便套接字在连接之前绑定为其源地址。如果 host 或 port 分别为" 或 0，则使用操作系统默认行为。

当无法创建连接时，将会引发一个异常。在默认情况下，它将是来自列表中最后一个地址的异常。如果 *all_errors* 为 `True`，它将是一个包含所有尝试错误的*ExceptionGroup*。

在 3.2 版本发生变更: 添加了 *source_address* 参数

在 3.11 版本发生变更: 添加了 *all_errors*。

socket.**create_server**(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

创建绑定到 *address* 的 TCP 套接字（一个 `(host, port)` 2 元组）并返回该套接字对象的便捷函数。

family 应当为*AF_INET* 或*AF_INET6*。*backlog* 是传递给*socket.listen()* 的队列大小；当未指定时，将选择一个合理的默认值。*reuse_port* 指定是否要设置 `SO_REUSEPORT` 套接字选项。

如果 *dualstack_ipv6* 为真值，*family* 为*AF_INET6* 并且平台支持则套接字将能同时接受 IPv4 和 IPv6 连接，否则它将引发*ValueError*。大多数 POSIX 平台和 Windows 都应该支持此功能。当此功能被启用时*socket.getpeername()* 在进行 IPv4 连接时返回的将是一个被表示为映射到 IPv4 的 IPv6 地址的 IPv6 地址。如果 *dualstack_ipv6* 为假值则它将在默认启用此功能的平台上（例如 Linux）显式禁用此功能。该形参可与*has_dualstack_ipv6()* 结合使用：

```
import socket

addr = ("", 8080)  # 所有接口，端口 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

备注

在 POSIX 平台上，设置 `SO_REUSEADDR` 套接字选项是为了立即重用以前绑定在同一 *address* 上并保持 TIME_WAIT 状态的套接字。

Added in version 3.8.

socket.**has_dualstack_ipv6**()

如果平台支持创建 IPv4 和 IPv6 连接都可以处理的 TCP 套接字，则返回 `True`。

Added in version 3.8.

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

复制文件描述符 *fd* (由文件对象的*fileno()* 方法返回的整数) 并根据结果构建一个套接字对象。地址族、套接字类型和协议号与上述*socket()* 函数中的相同。文件描述符应指向一个套接字，但不会检查这一点 --- 如果文件描述符是无效的则对该对象的后续操作可能会失败。本函数很少被用到，

但在将套接字作为标准输入或输出传给给程序 (如 Unix inet 进程启动的服务器) 时可以用来获取或设置套接字选项。套接字将被假定为阻塞模式。

新创建的套接字是不可继承的。

在 3.4 版本发生变更: 返回的套接字现在是不可继承的。

socket.**fromshare**(*data*)

根据*socket.share()* 方法获得的数据实例化套接字。套接字将处于阻塞模式。

Availability: Windows.

Added in version 3.3.

socket.**SocketType**

这是一个 Python 类型对象，表示套接字对象的类型。它等同于 type(socket(...))。

其他功能

socket 模块还提供多种网络相关服务：

socket.**close**(*fd*)

关闭一个套接字文件描述符。它类似于*os.close()*，但专用于套接字。在某些平台上（特别是在 Windows 上），*os.close()* 对套接字文件描述符无效。

Added in version 3.7.

socket.**getaddrinfo**(*host*, *port*, *family=AF_UNSPEC*, *type=0*, *proto=0*, *flags=0*)

此函数对下层系统的 C 函数 getaddrinfo 进行了包装。

将 *host*/*port* 参数转换为 5 元组的序列，其中包含创建（连接到某服务的）套接字所需的所有参数。*host* 是域名，是字符串格式的 IPv4/v6 地址或 None。*port* 是字符串格式的服务名称，如 'http'、端口号（数字）或 None。传入 None 作为 *host* 和 *port* 的值，相当于将 NULL 传递给底层 C API。

可以选择指定 *family*, *type* 和 *proto* 参数以提供选项并对所返回的地址列表进行限制。为其传入默认值（分别为*AF_UNSPEC*, 0 和 0）则不会对结果进行限制。请参阅下面的注释了解详情。

flags 参数可以是 AI_* 常量中的一个或多个，并会影响结果的计算和返回。例如，AI_NUMERICHOST 将禁用域名解析并将在 *host* 为域名时引发错误。

本函数返回一个列表，其中的 5 元组具有以下结构：

```
(family, type, proto, canonname, sockaddr)
```

在这些元组中，*family*, *type*, *proto* 都是整数且其作用是被传给*socket()* 函数。如果 AI_CANONNAME 是 *flags* 参数的一部分则 *canonname* 将为表示 *host* 的规范名称的字符串；否则 *canonname* 将为空。*sockaddr* 是一个描述套接字地址的元组，其具体格式取决于返回的 *family* (对于*AF_INET* 将为 (address, port) 2 元组，对于*AF_INET6* 将为 (address, port, flowinfo, scope_id) 4 元组)，其作用是被传给*socket.connect()* 方法。

> **备注**
>
> 如果你想要使用来自 getaddrinfo() 的结果创建套接字（而不是使用提取 *canonname* 等方式），可以考虑通过 *type* (例如*SOCK_STREAM* 或*SOCK_DGRAM*) 和/或你的应用程序能处理的 *proto* (例如 IPPROTO_TCP 或 IPPROTO_UDP) 来限制结果。
>
> 对于 *family*, *type*, *proto* 和 *flags* 使用默认值时的行为取决于具体的系统。
>
> 许多系统（例如，大多数 Linux 配置）将返回一个由所有匹配的地址组成的已排序列表。这些地址通常应按顺序被尝试直到有一个连接成功（可能是并行尝试，例如使用 Happy Eyeballs 算法）。在这些情况下，限制 *type* 和/或 *proto* 有助于消除不成功或不可用的连接尝试。
>
> 但是，某些系统将只返回一个地址。(例如，在 Solaris 和 AIX 配置上即有报告此种情况。) 在这些系统上，限制 *type* 和/或 *proto* 有助于确保此地址是可用的。

引发一个审计事件 `socket.getaddrinfo` 并附带参数 `host`, `port`, `family`, `type`, `protocol`。

下面的示例获取了 TCP 连接地址信息，假设该连接通过 80 端口连接至 `example.org` （如果系统未启用 IPv6，则结果可能会不同）：

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(socket.AF_INET6, socket.SOCK_STREAM,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (socket.AF_INET, socket.SOCK_STREAM,
 6, '', ('93.184.216.34', 80))]
```

在 3.2 版本发生变更: 现在可以使用关键字参数的形式来传递参数。

在 3.7 版本发生变更: 对于 IPv6 多播地址，表示地址的字符串将不包含 `%scope_id` 部分。

`socket.`**`getfqdn`**(*[name]*)

返回 *name* 的完整限定域名。如果 *name* 被省略或为空，则将其解读为本地主机。要查找完整限定名称，将先检查*gethostbyaddr()* 所返回的主机名，然后是主机的别名（如果存在）。包括句点的第一个名称将会被选择。对于没有完整限定域名而提供了 *name* 的情况，则会将其原样返回。如果 *name* 为空或等于 `'0.0.0.0'`，则返回来自*gethostname()* 的主机名。

`socket.`**`gethostbyname`**(*hostname*)

将主机名转换为 IPv4 地址格式。IPv4 地址以字符串格式返回，如 `'100.50.200.5'`。如果主机名本身是 IPv4 地址，则原样返回。更完整的接口请参考*gethostbyname_ex()*。*gethostbyname()* 不支持 IPv6 名称解析，应使用*getaddrinfo()* 来支持 IPv4/v6 双协议栈。

引发一个审计事件 `socket.gethostbyname` 并附带参数 `hostname`。

Availability: not WASI.

`socket.`**`gethostbyname_ex`**(*hostname*)

将一个主机名转换为 IPv4 地址格式的扩展接口。返回一个 3 元组 `(hostname, aliaslist, ipaddrlist)` 其中 *hostname* 是主机的首选主机名，*aliaslist* 是同一地址的备选主机名列表（可能为空），而 *ipaddrlist* 是同一主机上同一接口的 IPv4 地址列表（通常为单个地址但并不总是如此）。*gethostbyname_ex()* 不支持 IPv6 名称解析，应当改用*getaddrinfo()* 来提供 IPv4/v6 双栈支持。

引发一个审计事件 `socket.gethostbyname` 并附带参数 `hostname`。

Availability: not WASI.

`socket.`**`gethostname`**()

返回一个字符串，包含当前正在运行 Python 解释器的机器的主机名。

引发一个不带参数的审计事件 `socket.gethostname`。

注意：*gethostname()* 并不总是返回全限定域名，必要的话请使用*getfqdn()*。

Availability: not WASI.

`socket.`**`gethostbyaddr`**(*ip_address*)

返回一个 3 元组 `(hostname, aliaslist, ipaddrlist)` 其中 *hostname* 是响应给定 *ip_address* 的首选主机名，*aliaslist* 是同一地址的备选主机名列表（可能为空），而 *ipaddrlist* 是同一主机上同一接口的 IPv4/v6 地址列表（很可能仅包含一个地址）。要查询完整限定域名，请使用函数*getfqdn()*。*gethostbyaddr()* 同时支持 IPv4 和 IPv6。

引发一个审计事件 `socket.gethostbyaddr` 并附带参数 `ip_address`。

Availability: not WASI.

`socket.`**`getnameinfo`**(*sockaddr*, *flags*)

将套接字地址 *sockaddr* 转换为一个 2 元组 `(host, port)`。根据 *flags* 的设置，结果可能包含 *host* 中的完整限定域名或数字形式的地址。类似地，*port* 可以包含字符串形式的端口名或数字形式的端口号。

对于 IPv6 地址，如果 *sockaddr* 包含有意义的 *scope_id*，则 `%scope_id` 会被附加到主机部分。这种情况通常发生在多播地址上。

关于 *flags* 的更多信息可参阅 *getnameinfo(3)*。

引发一个审计事件 socket.getnameinfo 并附带参数 sockaddr。

Availability: not WASI.

socket.**getprotobyname**(*protocolname*)

将一个互联网协议名称 (如 'icmp') 转写为能被作为 (可选的) 第三个参数传给 *socket()* 函数的常量。这通常仅对以"raw" 模式 (*SOCK_RAW*) 打开的套接字来说是必要的；对于正常的套接字模式，当协议名称被省略或为零时会自动选择正确的协议。

Availability: not WASI.

socket.**getservbyname**(*servicename*[, *protocolname*])

将一个互联网服务名称和协议名称转换为该服务的端口号。如果给出了可选的协议名称，它应为 'tcp' 或 'udp'，否则将匹配任意的协议。

引发一个审计事件 socket.getservbyname 并附带参数 servicename, protocolname。

Availability: not WASI.

socket.**getservbyport**(*port*[, *protocolname*])

将一个互联网端口号和协议名称转换为该服务的服务名称。如果给出了可选的协议名称，它应为 'tcp' 或 'udp'，否则将匹配任意的协议。

引发一个审计事件 socket.getservbyport 并附带参数 port, protocolname。

Availability: not WASI.

socket.**ntohl**(*x*)

将 32 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**ntohs**(*x*)

将 16 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

在 3.10 版本发生变更: 如果 *x* 不能转为 16 位无符号整数则会引发 *OverflowError*。

socket.**htonl**(*x*)

将 32 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**htons**(*x*)

将 16 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

在 3.10 版本发生变更: 如果 *x* 不能转为 16 位无符号整数则会引发 *OverflowError*。

socket.**inet_aton**(*ip_string*)

将一个 IPv4 地址从以点号分为四段的字符串格式（例如'123.45.67.89'）转换为 32 位的紧凑二进制格式，长度为四个字符的字节串对象。这在与使用标准 C 库并且需要 in_addr 类型对象的程序通信时很有用处，该类型就是此函数所返回的 32 位的紧凑二进制格式 C 类型。

inet_aton() 也接受句点数少于三的字符串，详情请参阅 Unix 手册 *inet(3)*。

如果传入本函数的 IPv4 地址字符串无效，则抛出 *OSError*。注意，具体什么样的地址有效取决于 inet_aton() 的底层 C 实现。

inet_aton() 不支持 IPv6，在 IPv4/v6 双协议栈下应使用 *inet_pton()* 来代替。

socket.**inet_ntoa**(*packed_ip*)

将一个 32 位紧凑 IPv4 地址 (长度为四个字节的 *bytes-like object*) 转换为标准的以点号四分段字符串表示形式 (例如'123.45.67.89')。这在与使用标准 C 库并且需要 in_addr 类型对象的程序通信时很有用处，该类型就是此函数接受作为参数的 32 位的紧凑二进制格式 C 类型。

如果传入本函数的字节序列长度不是 4 个字节，则抛出*OSError*。*inet_ntoa()* 不支持 IPv6，在 IPv4/v6 双协议栈下应使用*inet_ntop()* 来代替。

在 3.5 版本发生变更: 现在接受可写的*字节类对象*。

socket.**inet_pton**(*address_family*, *ip_string*)

将基于特定地址族字符串格式的 IP 地址转换为紧凑的二进制格式。*inet_pton()* 在一个库或网络协议需要 `in_addr` (类似于*inet_aton()*) 或 `in6_addr` 类型的对象时很有用处。

目前 *address_family* 支持*AF_INET* 和*AF_INET6*。如果 IP 地址字符串 *ip_string* 无效，则抛出*OSError*。注意，具体什么地址有效取决于 *address_family* 的值和 `inet_pton()` 的底层实现。

Availability: Unix, Windows.

在 3.4 版本发生变更: 添加了 Windows 支持

socket.**inet_ntop**(*address_family*, *packed_ip*)

将一个紧凑的 IP 地址 (长度为多个字节的*bytes-like object*) 转换为标准的基于特定地址族的字符串表示形式 (例如 `'7.10.0.5'` 或 `'5aef:2b::8'`)。*inet_ntop()* 在一个库或网络协议返回 `in_addr` (类似于*inet_ntoa()*) 或 `in6_addr` 类型的对象时很有用处。

目前 *address_family* 支持*AF_INET* 和*AF_INET6*。如果字节对象 *packed_ip* 与指定的地址簇长度不符，则抛出*ValueError*。针对*inet_ntop()* 调用的错误则抛出*OSError*。

Availability: Unix, Windows.

在 3.4 版本发生变更: 添加了 Windows 支持

在 3.5 版本发生变更: 现在接受可写的*字节类对象*。

socket.**CMSG_LEN**(*length*)

返回给定 *length* 所关联数据的辅助数据项的总长度（不带尾部填充）。此值通常用作*recvmsg()* 接收一个辅助数据项的缓冲区大小，但是 **RFC 3542** 要求可移植应用程序使用*CMSG_SPACE()*，以此将尾部填充的空间计入，即使该项在缓冲区的最后。如果 *length* 超出允许范围，则抛出*OverflowError*。

Availability: Unix, not WASI.

大多数 Unix 平台。

Added in version 3.3.

socket.**CMSG_SPACE**(*length*)

返回*recvmsg()* 所需的缓冲区大小，以接收给定 *length* 所关联数据的辅助数据项，带有尾部填充。接收多个项目所需的缓冲区空间是关联数据长度的*CMSG_SPACE()* 值的总和。如果 *length* 超出允许范围，则抛出*OverflowError*。

请注意，某些系统可能支持辅助数据，但不提供本函数。还需注意，如果使用本函数的结果来设置缓冲区大小，可能无法精确限制可接收的辅助数据量，因为可能会有其他数据写入尾部填充区域。

Availability: Unix, not WASI.

大多数 Unix 平台。

Added in version 3.3.

socket.**getdefaulttimeout**()

返回用于新套接字对象的默认超时（以秒为单位的浮点数）。值 `None` 表示新套接字对象没有超时。首次导入 socket 模块时，默认值为 `None`。

socket.**setdefaulttimeout**(*timeout*)

设置用于新套接字对象的默认超时（以秒为单位的浮点数）。首次导入 socket 模块时，默认值为 `None`。可能的取值及其各自的含义请参阅*settimeout()*。

socket.**sethostname**(*name*)

将计算机的主机名设置为 *name*。如果权限不足将抛出*OSError*。

引发一个*审计事件* `socket.sethostname` 并附带参数 `name`。

Availability: Unix, not Android.

Added in version 3.3.

socket.**if_nameindex**()

返回一个列表，包含网络接口（网卡）信息二元组（整数索引，名称字符串）。系统调用失败则抛出*OSError*。

Availability: Unix, Windows, not WASI.

Added in version 3.3.

在 3.8 版本发生变更: 添加了 Windows 支持。

备注

在 Windows 中网络接口在不同上下文中具有不同的名称（所有名称见对应示例）：

- UUID: `{FB605B73-AAC2-49A6-9A2F-25416AEA0573}`
- 名称: `ethernet_32770`
- 友好名称: `vEthernet (nat)`
- 描述: `Hyper-V Virtual Ethernet Adapter`

此函数返回列表中第二种形式的名称，在此示例中为 `ethernet_32770`。

socket.**if_nametoindex**(*if_name*)

返回网络接口名称相对应的索引号。如果没有所给名称的接口，则抛出*OSError*。

Availability: Unix, Windows, not WASI.

Added in version 3.3.

在 3.8 版本发生变更: 添加了 Windows 支持。

参见

”Interface name” 为*if_nameindex()* 中所描述的名称。

socket.**if_indextoname**(*if_index*)

返回网络接口索引号相对应的接口名称。如果没有所给索引号的接口，则抛出*OSError*。

Availability: Unix, Windows, not WASI.

Added in version 3.3.

在 3.8 版本发生变更: 添加了 Windows 支持。

参见

”Interface name” 为*if_nameindex()* 中所描述的名称。

socket.**send_fds**(*sock*, *buffers*, *fds*[, *flags*[, *address*]])

将文件描述符列表 *fds* 通过一个*AF_UNIX* 套接字 *sock* 进行发送。*fds* 形参是由文件描述符组成的序列。请查看*sendmsg()* 获取这些形参的文档说明。

Availability: Unix, Windows, not WASI.

支持*sendmsg()* 和 `SCM_RIGHTS` 机制的 Unix 平台。

Added in version 3.9.

socket.**recv_fds**(*sock*, *bufsize*, *maxfds*[, *flags*])

接收至多 *maxfds* 个来自*AF_UNIX* 套接字 *sock* 的文件描述符。返回 `(msg, list(fds), flags, addr)`。请查看*recvmsg()* 获取这些形参的文档说明。

Availability: Unix, Windows, not WASI.

支持*sendmsg()* 和 `SCM_RIGHTS` 机制的 Unix 平台。

Added in version 3.9.

> **备注**
>
> 位于文件描述符列表末尾的任何被截断整数。

19.2.3 套接字对象

套接字对象具有以下方法。除了*makefile()*，其他都与套接字专用的 Unix 系统调用相对应。

在 3.2 版本发生变更: 添加了对上下文管理器 协议的支持。退出上下文管理器与调用*close()* 等效。

socket.**accept**()

接受一个连接。此 socket 必须绑定到一个地址上并且监听连接。返回值是一个 `(conn, address)` 对，其中 *conn* 是一个 新的套接字对象，用于在此连接上收发数据，*address* 是连接另一端的套接字所绑定的地址。

新创建的套接字是不可继承的。

在 3.4 版本发生变更: 该套接字现在是不可继承的。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**bind**(*address*)

将套接字绑定到 *address*。套接字必须尚未绑定。(*address* 的格式取决于地址簇——参见上文)

引发一个审计事件 `socket.bind` 并附带参数 `self`、`address`。

Availability: not WASI.

socket.**close**()

将套接字标记为已关闭。底层的系统资源（例如文件描述符）也将在*makefile()* 创建的所有文件对象关闭时被关闭。一旦上述情况发生，将来对该套接字对象的所有操作都将失败。远端将不会接收到新的数据（在队列中的数据被清空之后）。

垃圾回收时，套接字会自动关闭，但建议显式*close()* 它们，或在它们周围使用 `with` 语句。

在 3.6 版本发生变更: 现在，如果底层的 `close()` 调用出错，会抛出*OSError*。

> **备注**
>
> *close()* 会释放与连接相关联的资源但不一定立即关闭连接。如果你想要及时关闭连接，请在*close()* 之前调用*shutdown()*。

socket.**connect**(*address*)

连接到 *address* 处的远程套接字。(*address* 的格式取决于地址簇——参见上文)

如果连接被信号中断，则本方法将等待直至连接完成，或者如果信号处理器未引发异常并且套接字被阻塞或已超时则会在超时后引发*TimeoutError*。对于非阻塞型套接字，如果连接被信号中断则本方法将引发*InterruptedError* 异常（或信号处理器所引发的异常）。

引发一个审计事件 `socket.connect` 并附带参数 `self`、`address`。

在 3.5 版本发生变更: 本方法现在将等待，直到连接完成，而不是在以下情况抛出 *InterruptedError* 异常。该情况为，连接被信号中断，信号处理程序未抛出异常，且套接字阻塞中或已超时（具体解释请参阅 **PEP 475** ）。

Availability: not WASI.

socket.**connect_ex**(*address*)

类似于 `connect(address)`，但是对于 C 级别的 `connect()` 调用返回的错误，本函数将返回错误指示器，而不是抛出异常（对于其他问题，如 “找不到主机”，仍然可以抛出异常）。如果操作成功，则错误指示器为 `0`，否则为 `errno` 变量的值。这对支持如异步连接很有用。

引发一个审计事件 `socket.connect` 并附带参数 `self`、`address`。

Availability: not WASI.

socket.**detach**()

将套接字对象置于关闭状态，而底层的文件描述符实际并不关闭。返回该文件描述符，使其可以重新用于其他目的。

Added in version 3.2.

socket.**dup**()

创建套接字的副本。

新创建的套接字是不可继承的。

在 3.4 版本发生变更: 该套接字现在是不可继承的。

Availability: not WASI.

socket.**fileno**()

返回套接字的文件描述符（一个小整数），失败返回 -1。配合 *select.select()* 使用很有用。

在 Windows 下，此方法返回的小整数在允许使用文件描述符的地方无法使用（如 *os.fdopen()* ）。Unix 无此限制。

socket.**get_inheritable**()

获取套接字文件描述符或套接字句柄的可继承标志：如果子进程可以继承套接字则为 `True`，否则为 `False`。

Added in version 3.4.

socket.**getpeername**()

返回套接字连接到的远程地址。举例而言，这可以用于查找远程 IPv4/v6 套接字的端口号。（返回的地址格式取决于地址簇——参见上文。）部分系统不支持此函数。

socket.**getsockname**()

返回套接字本身的地址。举例而言，这可以用于查找 IPv4/v6 套接字的端口号。（返回的地址格式取决于地址簇——参见上文。）

socket.**getsockopt**(*level*, *optname*[, *buflen*])

返回给定套接字选项的值 (参见 Unix 手册页 *getsockopt(2)*)。所需的符号常量 (*SO_** 等) 在本模块中定义。如果未指定 *buflen*，则会假定该选项为整数值并且将由此函数返回其整数值。如果指定了 *buflen*，则它定义了用于存放选项值的缓冲区的最大长度，且该缓冲区将作为字节对象返回。调用方需要执行对缓冲区内容的解码（请参阅可选的内置模块 *struct* 了解如何对编码为字节串的 C 结构体进行解码）。

Availability: not WASI.

socket.**getblocking**()

如果套接字处于阻塞模式，返回 `True`，非阻塞模式返回 `False`。

这等价于检测 `socket.gettimeout() != 0`。

Added in version 3.7.

socket.**gettimeout**()

返回套接字操作相关的超时秒数（浮点数），未设置超时则返回 None。它反映最后一次调用 *setblocking()* 或 *settimeout()* 后的设置。

socket.**ioctl**(*control*, *option*)

平台
Windows

ioctl() 方法是 WSAIoctl 系统接口的有限接口。请参考 Win32 文档 以获取更多信息。

在其他平台上，可以使用通用的 *fcntl.fcntl()* 和 *fcntl.ioctl()* 函数，它们接受套接字对象作为第一个参数。

当前仅支持以下控制码：SIO_RCVALL、SIO_KEEPALIVE_VALS 和 SIO_LOOPBACK_FAST_PATH。

在 3.6 版本发生变更: 添加了 SIO_LOOPBACK_FAST_PATH。

socket.**listen**([*backlog*])

启动一个服务器用于接受连接。如果指定 *backlog*，则它最低为 0（小于 0 会被置为 0），它指定系统允许暂未 accept 的连接数，超过后将拒绝新连接。未指定则自动设为合理的默认值。

Availability: not WASI.

在 3.5 版本发生变更: *backlog* 参数现在是可选的。

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)

返回一个与套接字相关联的 *file object*。返回对象的具体类型取决于传给 *makefile()* 的参数。这些参数的解读方式与内置的 *open()* 函数相同，区别在于 *mode* 值仅支持 'r' (默认), 'w', 'b' 或它们的组合。

套接字必须处于阻塞模式，它可以有超时，但是如果发生超时，文件对象的内部缓冲区可能会以不一致的状态结尾。

关闭 *makefile()* 返回的文件对象不会关闭原始套接字，除非所有其他文件对象都已关闭且在套接字对象上调用了 *socket.close()*。

> **备注**
>
> 在 Windows 上，由 *makefile()* 创建的文件型对象无法作为带文件描述符的文件对象使用，如无法作为 *subprocess.Popen()* 的流参数。

socket.**recv**(*bufsize*[, *flags*])

从套接字接收数据。返回值是一个代表所接收数据的字节串对象。可一次性接收的最大数据量由 *bufsize* 指定。返回空字节串对象表示客户端已断开连接。请参阅 Unix 手册页 *recv(2)* 了解可选参数 *flags* 的含义；它默认为零。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**recvfrom**(*bufsize*[, *flags*])

从套接字接收数据。返回值是一对 (bytes, address)，其中 *bytes* 是字节对象，表示接收到的数据，*address* 是发送端套接字的地址。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。(*address* 的格式取决于地址簇——参见上文)

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

在 3.7 版本发生变更: 对于多播 IPv6 地址，*address* 的第一项不会再包含 %scope_id 部分。要获得完整的 IPv6 地址请使用 *getnameinfo()*。

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

从套接字接收普通数据（至多 *bufsize* 字节）和辅助数据。*ancbufsize* 参数设置用于接收辅助数据的内部缓冲区的大小（以字节为单位），默认为 0，表示不接收辅助数据。可以使用*CMSG_SPACE()* 或*CMSG_LEN()* 计算辅助数据缓冲区的合适大小，无法放入缓冲区的项目可能会被截断或丢弃。*flags* 参数默认为 0，其含义与*recv()* 中的相同。

返回值是一个四元组：(data, ancdata, msg_flags, address)。*data* 项是一个*bytes* 对象，用于保存接收到的非辅助数据。*ancdata* 项是零个或多个元组 (cmsg_level, cmsg_type, cmsg_data) 组成的列表，表示接收到的辅助数据（控制消息）：*cmsg_level* 和 *cmsg_type* 是分别表示协议级别和协议类型的整数，而 *cmsg_data* 是保存相关数据的*bytes* 对象。*msg_flags* 项由各种标志按位或组成，表示接收消息的情况，详细信息请参阅系统文档。如果接收端套接字断开连接，则 *address* 是发送端套接字的地址（如果有），否则该值无指定。

在某些系统上，可以使用*sendmsg()* 和*recvmsg()* 通过*AF_UNIX* 套接字在进程之间传递文件描述符。当使用此功能时 (通常仅限于*SOCK_STREAM* 套接字)，*recvmsg()* 将在其附带数据中返回 (socket.SOL_SOCKET, socket.SCM_RIGHTS, fds) 形式的项，其中 *fds* 是一个代表新文件描述符的原生 as a binary array of the native C int 类型的二进制数组形式的*bytes* 对象。如果*recvmsg()* 在系统调用返回后引发了异常，它将首先尝试关闭通过此机制接收到的任何文件描述符。

对于仅接收到一部分的辅助数据项，一些系统没有指示其截断长度。如果某个项目可能超出了缓冲区的末尾，*recvmsg()* 将发出*RuntimeWarning*，并返回其在缓冲区内的部分，前提是该对象被截断于关联数据开始后。

在支持 SCM_RIGHTS 机制的系统上，下方的函数将最多接收 *maxfds* 个文件描述符，返回消息数据和包含描述符的列表（同时忽略意外情况，如接收到无关的控制消息）。另请参阅*sendmsg()*。

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # 整数数组
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # 添加数据，忽略任何在末尾被截断的整数。
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.itemsize)])
    return msg, list(fds)
```

Availability: Unix.

大多数 Unix 平台。

Added in version 3.3.

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

从套接字接收普通数据和辅助数据，其行为与*recvmsg()* 相同，但将非辅助数据分散到一系列缓冲区中，而不是返回新的字节对象。*buffers* 参数必须是可迭代对象，它迭代出可供写入的缓冲区 (如*bytearray* 对象)，这些缓冲区将被连续的非辅助数据块填充，直到数据全部写完或缓冲区用完为止。在允许使用的缓冲区数量上，操作系统可能会有限制（*sysconf()* 的 SC_IOV_MAX 值）。*ancbufsize* 和 *flags* 参数的含义与*recvmsg()* 中的相同。

返回值为四元组：(nbytes, ancdata, msg_flags, address)，其中 *nbytes* 是写入缓冲区的非辅助数据的字节总数，而 *ancdata*、*msg_flags* 和 *address* 与*recvmsg()* 中的相同。

示例:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
```

(续下页)

(接上页)

```
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

Availability: Unix.

大多数 Unix 平台。

Added in version 3.3.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

从套接字接收数据，将其写入 *buffer* 而不是创建新的字节串。返回值是一对 `(nbytes, address)`，其中 *nbytes* 是收到的字节数，*address* 是发送端套接字的地址。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。（*address* 的格式取决于地址簇——参见上文）

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

从套接字接收至多 *nbytes* 个字节，将其写入缓冲区而不是创建新的字节串。如果 *nbytes* 未指定（或指定为 0），则接收至所给缓冲区的最大可用大小。返回接收到的字节数。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。

socket.**send**(*bytes*[, *flags*])

发送数据给套接字。本套接字必须已连接到远程套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。本方法返回已发送的字节数。应用程序要负责检查所有数据是否已发送，如果仅传输了部分数据，程序需要自行尝试传输其余数据。有关该主题的更多信息，请参考 socket-howto。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendall**(*bytes*[, *flags*])

发送数据给套接字。本套接字必须已连接到远程套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。与 *send()* 不同，本方法持续从 *bytes* 发送数据，直到所有数据都已发送或发生错误为止。成功后会返回 `None`。出错后会抛出一个异常，此时并没有办法确定成功发送了多少数据。

在 3.5 版本发生变更: 每次成员发送数据后，套接字超时将不再重置。目前的套接字超时是发送所有数据的最大总持续时间。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags*, *address*)

发送数据给套接字。本套接字不应连接到远程套接字，而应由 *address* 指定目标套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。本方法返回已发送的字节数。（*address* 的格式取决于地址簇——参见上文。）

引发一个审计事件 `socket.sendto` 并附带参数 `self`, `address`。

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

将普通数据和辅助数据发送给套接字，将从一系列缓冲区中收集非辅助数据，并将其拼接为一条消息。*buffers* 参数指定的非辅助数据应为可迭代的字节类对象（如 *bytes* 对象），在允许使用的缓冲区数量上，操作系统可能会有限制（*sysconf()* 的 `SC_IOV_MAX` 值）。*ancdata* 参数指定的辅助数据（控制消息）应为可迭代对象，迭代出零个或多个 `(cmsg_level, cmsg_type, cmsg_data)` 元组，其中 *cmsg_level* 和 *cmsg_type* 是分别指定协议级别和协议类型的整数，而 *cmsg_data* 是保存相关数据的字节类对象。请注意，某些系统（特别是没有 *CMSG_SPACE()* 的系统）可能每次调用仅支持发送一条控制消息。*flags* 参数默认为 0，与 *send()* 中的含义相同。如果 *address* 指定为除 `None` 以外的值，它将作为消息的目标地址。返回值是已发送的非辅助数据的字节数。

在支持 SCM_RIGHTS 机制的系统上，下方的函数通过一个*AF_UNIX* 套接字来发送文件描述符列表 *fds*。另请参阅*recvmsg()*。

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.array("i",␣
↪fds))])
```

Availability: Unix, not WASI.

大多数 Unix 平台。

引发一个审计事件 `socket.sendmsg` 并附带参数 `self`, `address`。

Added in version 3.3.

在 3.5 版本发生变更: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

`socket.`**`sendmsg_afalg`**(*[msg,]*, op[, iv[, assoclen[, flags]]]*)

为*AF_ALG* 套接字定制的*sendmsg()* 版本。可为*AF_ALG* 套接字设置模式、IV、AEAD 关联数据的长度和标志位。

Availability: Linux >= 2.6.38.

Added in version 3.6.

`socket.`**`sendfile`**(*file*, *offset=0*, *count=None*)

使用高性能的*os.sendfile* 发送文件，直到达到文件的 EOF 为止，返回已发送的字节总数。*file* 必须是一个以二进制模式打开的常规文件对象。如果*os.sendfile* 不可用（如 Windows）或 *file* 不是常规文件，将使用*send()* 代替。*offset* 指示从哪里开始读取文件。如果指定了 *count*，它确定了要发送的字节总数，而不会持续发送直到达到文件的 EOF。返回时或发生错误时，文件位置将更新，在这种情况下，*file.tell()* 可用于确定已发送的字节数。套接字必须为*SOCK_STREAM* 类型。不支持非阻塞的套接字。

Added in version 3.5.

`socket.`**`set_inheritable`**(*inheritable*)

设置套接字文件描述符或套接字句柄的可继承标志。

Added in version 3.4.

`socket.`**`setblocking`**(*flag*)

设置套接字为阻塞或非阻塞模式：如果 *flag* 为 false，则将套接字设置为非阻塞，否则设置为阻塞。

本方法是某些*settimeout()* 调用的简写：

- `sock.setblocking(True)` 相当于 `sock.settimeout(None)`
- `sock.setblocking(False)` 相当于 `sock.settimeout(0.0)`

在 3.7 版本发生变更: 本方法不再对*socket.type* 属性设置*SOCK_NONBLOCK* 标志。

`socket.`**`settimeout`**(*value*)

为阻塞套接字的操作设置超时。*value* 参数可以是非负浮点数，表示秒，也可以是 `None`。如果赋为一个非零值，那么如果在操作完成前超过了超时时间 *value*，后续的套接字操作将抛出*timeout* 异常。如果赋为 0，则套接字将处于非阻塞模式。如果指定为 `None`，则套接字将处于阻塞模式。

更多信息请查阅关于套接字超时的说明。

在 3.7 版本发生变更: 本方法不再修改*socket.type* 属性的*SOCK_NONBLOCK* 标志。

`socket.`**`setsockopt`**(*level*, *optname*, *value:* int)

`socket.`**`setsockopt`**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)

设置给定套接字选项的值 (参见 Unix 手册页 *setsockopt(2)*)。所需的符号常量已定义在本模块中 (SO_* 等 <socket-unix-constants>)。该值可以是整数、None 或表示缓冲区的*bytes-like object*。在后一种情况下将由调用者确保字节串中包含正确的数据位 (请参阅可选的内置模块*struct* 了解如何将 C 结构体编码为字节串)。当 *value* 设为 None 时，*optlen* 参数是必须的。这等价于调用 setsockopt() C 函数并设置 optval=NULL 和 optlen=optlen。

在 3.5 版本发生变更: 现在接受可写的字节类对象。

在 3.6 版本发生变更: 添加了 setsockopt(level, optname, None, optlen: int) 调用形式。

Availability: not WASI.

socket.**shutdown**(*how*)

关闭一半或全部的连接。如果 *how* 为*SHUT_RD*，则后续不再允许接收。如果 *how* 为*SHUT_WR*，则后续不再允许发送。如果 *how* 为*SHUT_RDWR*，则后续的发送和接收都不允许。

Availability: not WASI.

socket.**share**(*process_id*)

复制套接字，并准备将其与目标进程共享。目标进程必须以 *process_id* 形式提供。然后可以利用某种形式的进程间通信，将返回的字节对象传递给目标进程，还可以使用*fromshare()* 在新进程中重新创建套接字。一旦本方法调用完毕，就可以安全地将套接字关闭，因为操作系统已经为目标进程复制了该套接字。

Availability: Windows.

Added in version 3.3.

注意此处没有 read() 或 write() 方法，请使用不带 *flags* 参数的*recv()* 和*send()* 来替代。

套接字对象还具有以下（只读）属性，这些属性与传入*socket* 构造函数的值相对应。

socket.**family**

套接字的协议簇。

socket.**type**

套接字的类型。

socket.**proto**

套接字的协议。

19.2.4 关于套接字超时的说明

一个套接字对象可以处于以下三种模式之一：阻塞、非阻塞或超时。套接字默认以阻塞模式创建，但是可以调用*setdefaulttimeout()* 来更改。

- 在 *blocking mode*（阻塞模式）中，操作将阻塞，直到操作完成或系统返回错误（如连接超时）。
- 在 *非阻塞模式*中，如果操作无法立即完成则该操作将失败（不幸的是它所附带的错误将依赖于具体系统）：来自*select* 模块的函数可被用来获知一个套接字是否可以读取或写入。
- 在 *timeout mode*（超时模式）下，如果无法在指定的超时内完成操作（抛出*timeout* 异常），或如果系统返回错误，则操作将失败。

备注

在操作系统层级，*超时模式*下的套接字在内部都被设为非阻塞模式。同时，阻塞和超时模式会在指向同一个网络端点的文件描述符和套接字对象之间共享。这一实现细节可能导致明显的后果，例如当你决定使用套接字的*fileno()* 的时候。

超时与 connect 方法

connect() 操作也受超时设置的约束，通常建议在调用*connect()* 之前调用*settimeout()*，或将超时参数直接传递给*create_connection()*。但是，无论 Python 套接字超时设置如何，系统网络栈都有可能返回自带的连接超时错误。

超时与 accept 方法

如果*getdefaulttimeout()* 的值不是*None*，则*accept()* 方法返回的套接字将继承该超时值。若是 None，返回的套接字行为取决于侦听套接字的设置：

- 如果侦听套接字处于 *阻塞模式*或 *超时模式*，则*accept()* 返回的套接字处于 *阻塞模式*；
- 如果侦听套接字处于 *非阻塞模式*，那么*accept()* 返回的套接字是阻塞还是非阻塞取决于操作系统。如果要确保跨平台时的正确行为，建议手动覆盖此设置。

19.2.5 示例

以下是四个使用 TCP/IP 协议的最小示例程序：一个将收到的所有数据原样回馈的服务器（仅服务一个客户端），和一个使用该服务器的客户端。请注意服务器必须按*socket()*，*bind()*，*listen()*，*accept()* 的顺序执行（可能需要重复执行*accept()* 以便服务多个客户端），而客户端仅需要按*socket()*，*connect()* 的顺序执行。还要注意服务器不是在侦听的套接字上发送*sendall()*/*recv()* 而是在由*accept()* 返回的新套接字上发送。

前两个示例仅支持 IPv4。

```
# Echo server program
import socket

HOST = ''                 # 该符号名表示所有可用接口
PORT = 50007              # 任意非特权端口
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# 回显客户端程序
import socket

HOST = 'daring.cwi.nl'    # 远端主机
PORT = 50007              # 与服务器所用端口相同
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

接下来的两个例子与上面两个很想像，但同时支持 IPv4 和 IPv6。服务端将监听第一个可用的地址族（它本应同时监听两个地址族）。在大多数支持 IPv6 的系统中，IPv6 将有优先权并且服务端可能不会接受 IPv4 流量。客户端将尝试连接到作为名称解析结果被返回的所有地址，并将流量发送给第一个成功连接的地址。

```
# 回显服务端程序
import socket
import sys
```

(续下页)

(接上页)

```
HOST = None               # 该符号名表示所有可用接口
PORT = 50007              # 任意非特权端口
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# 回显客户端程序
import socket
import sys

HOST = 'daring.cwi.nl'    # 远端主机
PORT = 50007              # 与服务器所用端口相同
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

下面的例子演示了如何在 Windows 上使用原始套接字编写一个非常简单的网络嗅探器。这个例子需要管理员权限来修改接口:

```
import socket

# 公共网络接口
HOST = socket.gethostbyname(socket.gethostname())

# 创建一个原始套接字并将其绑定到公共接口
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# 包括 IP 标头
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# 接收所有数据包
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# 接收一个数据包
print(s.recvfrom(65565))

# 禁用混杂模式
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

下面的例子演示了如何使用 socket 接口与采用原始套接字协议的 CAN 网络进行通信。要改为通过广播管理器协议来使用 CAN，则要用以下方式打开一个 socket:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

在绑定 (CAN_RAW) 或连接 (*CAN_BCM*) 套接字之后，你将可以在套接字对象上正常地使用*socket.send()* 和*socket.recv()* 操作（及其同类操作）。

最后一个例子可能需要特别的权限:

```
import socket
import struct


# CAN 帧打包/解包 (参见 <linux/can.h> 中的 'struct can_frame')

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# 创建一个原始套接字并将其绑定到 'vcan0' 接口
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
```

(续下页)

(接上页)

```
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

多次运行一个示例，且每次执行之间等待时间过短，可能导致这个错误:

```
OSError: [Errno 98] Address already in use
```

这是因为前一次运行使套接字处于 TIME_WAIT 状态，无法立即重用。

要防止这种情况，需要设置一个*socket* 旗标 socket.SO_REUSEADDR:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

SO_REUSEADDR 标志告诉内核将处于 TIME_WAIT 状态的本地套接字重新使用，而不必等到固有的超时到期。

参见

关于套接字编程（C 语言）的介绍，请参阅以下文章:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*，作者 Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*，作者 Samuel J. Leffler et al,

两篇文章都在 UNIX 开发者手册，补充文档 1（第 PS1:7 和 PS1:8 节）中。那些特定于平台的参考资料，它们包含与套接字有关的各种系统调用，也是套接字语义细节的宝贵信息来源。对于 Unix，请参考手册页。对于 Windows，请参阅 WinSock（或 Winsock 2）规范。如果需要支持 IPv6 的 API，读者可能希望参考 **RFC 3493**，标题为 Basic Socket Interface Extensions for IPv6。

19.3 ssl --- 套接字对象的 TLS/SSL 包装器

源代码: Lib/ssl.py

该模块提供了对传输层安全（通常称为" 安全套接字层"）加密和网络套接字的对等认证设施的访问，包括客户端和服务器端。该模块使用 OpenSSL 库。它可以在所有现代 Unix 系统、Windows 、macOS 和可能的其他平台上使用，只要 OpenSSL 安装在该平台上。

备注

某些行为可能依赖于具体平台，因为调用了操作系统的套接字 API. 已安装的 OpenSSL 版本也可能会导致不同的行为。比如，TLSv1.3 是 OpenSSL 1.1.1 版才提供的。

警告

在阅读*安全考量* 前不要使用此模块。这样做可能会导致虚假的安全感，因为 ssl 模块的默认设置不一定适合你的应用程序。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

文档本文档记录 `ssl` 模块的对象和函数；更多关于 TLS,SSL, 和证书的信息，请参阅下方的“详情”选项

本模块提供了一个类*ssl.SSLSocket*，它派生自*socket.socket* 类型，并提供类似套接字的包装器，也能够使用 SSL 对通过套接字的数据进行加密和解密。它支持一些额外方法例如 `getpeercert()`，该方法可以从连接的另一端获取证书，还有 `cipher()`，该方法可获取安全连接所使用的密码，以及 `get_verified_chain()`、`get_unverified_chain()`，它们可获取证书链。

对于更复杂的应用程序，*ssl.SSLContext* 类有助于管理设置项和证书，进而可以被使用*SSLContext.wrap_socket()* 方法创建的 SSL 套接字继承。

在 3.5.3 版本发生变更: 更新以支持和 OpenSSL 1.1.0 的链接

在 3.6 版本发生变更: OpenSSL 0.9.8、1.0.0 和 1.0.1 已过时，将不再被支持。在 ssl 模块未来的版本中，最低需要 OpenSSL 1.0.2 或 1.1.0。

在 3.10 版本发生变更: **PEP 644** 已经实现。ssl 模块需要 OpenSSL 1.1.1 以上版本的支持。

使用废弃的常量和函数会导致废弃警告。

19.3.1 方法、常量和异常处理

套接字创建

SSLSocket 的实例必须使用*SSLContext.wrap_socket()* 方法来创建。辅助函数*create_default_context()* 将返回一个使用安全的默认设置的新上下文。

客户端套接字实例，采用默认上下文和 IPv4/IPv6 双栈:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

客户端套接字示例，带有自定义上下文和 IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT 需要有效的证书链和主机名
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

服务器套接字实例，在 localhost 上监听 IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

上下文创建

便捷函数，可以帮助创建*SSLContext* 对象，用于常见的目的。

ssl.**create_default_context**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

返回一个新的*SSLContext* 对象，使用给定 *purpose* 的默认设置。该设置由*ssl* 模块选择，并且通常是代表一个比直接调用*SSLContext* 构造器时更高的安全等级。

cafile、*capath*、*cadata* 代表用于进行证书核验的可选受信任 CA 证书，与*SSLContext.load_verify_locations()* 的一致。如果三个参数均为*None*，此函数可以转而选择信任系统的默认 CA 证书。

设置为: *PROTOCOL_TLS_CLIENT* 或*PROTOCOL_TLS_SERVER*, *OP_NO_SSLv2* 和*OP_NO_SSLv3* 带有不含 RC4 及未认证的高强度加密密码套件。传入*SERVER_AUTH* 作为 *purpose* 将把*verify_mode* 设为*CERT_REQUIRED* 并加载 CA 证书（若至少给出 *cafile*, *capath* 或 *cadata* 之一）或使用*SSLContext.load_default_certs()* 加载默认的 CA 证书。

当*keylog_filename* 受支持并且设置了环境变量 SSLKEYLOGFILE 时，*create_default_context()* 会启用密钥日志记录。

此上下文的默认设置包括*VERIFY_X509_PARTIAL_CHAIN* 和*VERIFY_X509_STRICT*。这使得下层的 OpenSSL 实现的行为与符合 **RFC 5280** 的实现更为相似，代价则是与较旧的 X.509 证书存在少量不兼容。

备注

协议、选项、密码和其他设置可随时更改为更具约束性的值而无须事先弃用。这些值代表了兼容性和安全性之间的合理平衡。

如果你的应用需要特定的设置，你应当创建一个*SSLContext* 并自行应用设置。

备注

如果你发现当某些较旧的客户端或服务器尝试与用此函数创建的*SSLContext* 进行连接时收到了报错提示"Protocol or cipher suite mismatch"，这可能是因为它们只支持 SSL3.0 而它被此函数用*OP_NO_SSLv3* 排除掉了。SSL3.0 被广泛认为 完全不可用。如果你仍希望继续使用此函数但仍允许 SSL 3.0 连接，你可以使用以下代码重新启用它们:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

备注

此上下文默认会启用*VERIFY_X509_STRICT*，它可能拒绝下层 OpenSSL 实现在其他情况下应当会接受的 **RFC 5280** 之前的证书或格式错误的证书。虽然不建议禁用此功能，但你可以使用以下方式做到这一点:

```
ctx = ssl.create_default_context()
ctx.verify_flags &= ~ssl.VERIFY_X509_STRICT
```

Added in version 3.4.

在 3.4.4 版本发生变更: RC4 被从默认密码字符串中丢弃。

在 3.6 版本发生变更: ChaCha20/Poly1305 被添加到默认密码字符串中。

3DES 被从默认密码字符串中丢弃。

在 3.8 版本发生变更: 增加了对密钥日志记录至 SSLKEYLOGFILE 的支持。

在 3.10 版本发生变更: 当前上下文使用*PROTOCOL_TLS_CLIENT* 或*PROTOCOL_TLS_SERVER* 协议而非通用的*PROTOCOL_TLS*。

在 3.13 版本发生变更: 此上下文现在会在其默认验证旗标中使用*VERIFY_X509_PARTIAL_CHAIN* 和*VERIFY_X509_STRICT*。

异常

exception ssl.**SSLError**

引发此异常以提示来自下层 SSL 实现（目前由 OpenSSL 库提供）的错误。它表示在下层网络连接之上叠加的高层级加密和验证层存在某种问题。此错误是*OSError* 的一个子类型。*SSLError* 实例的错误和消息是由 OpenSSL 库提供的。

在 3.3 版本发生变更: *SSLError* 曾经是*socket.error* 的一个子类型。

library

一个字符串形式的助记符，用来指明发生错误的 OpenSSL 子模块，例如 SSL, PEM 或 X509。可能的取值范围依赖于 OpenSSL 的版本。

Added in version 3.3.

reason

一个字符串形式的助记符，用来指明发生错误的原因，例如 CERTIFICATE_VERIFY_FAILED。可能的取值范围依赖于 OpenSSL 的版本。

Added in version 3.3.

exception ssl.**SSLZeroReturnError**

SSLError 的子类，当尝试读取或写入且 SSL 连接已被完全关闭时会被引发。请注意这并不意味着下层的传输（读取 TCP）已被关闭。

Added in version 3.3.

exception ssl.**SSLWantReadError**

SSLError 的子类，当尝试读取或写入数据，但在请求被满足之前还需要在下层的 TCP 传输上接收更多数据时会被非阻塞型 *SSL* 套接字 引发。

Added in version 3.3.

exception ssl.**SSLWantWriteError**

SSLError 的子类，当尝试读取或写入数据，但在请求被满足之前还需要在下层的 TCP 传输上发送更多数据时会被非阻塞型 *SSL* 套接字 引发。

Added in version 3.3.

exception ssl.**SSLSyscallError**

SSLError 的子类，当尝试在 SSL 套接字上执行操作时遇到系统错误时会被引发。不幸的是，没有简单的方式能检查原始 errno 编号。

Added in version 3.3.

exception ssl.**SSLEOFError**

SSLError 的子类，当 SSL 连接被突然终止时会被引发。通常，当遇到此错误时你不应再尝试重用下层的传输。

Added in version 3.3.

exception ssl.**SSLCertVerificationError**

SSLError 的子类，当证书验证失败时会被引发。

Added in version 3.7.

verify_code

一个数字形式的错误编号，用于表示验证错误。

verify_message

用于表示验证错误的人类可读的字符串。

exception ssl.**CertificateError**

SSLCertVerificationError 的别名。

在 3.7 版本发生变更: 此异常现在是*SSLCertVerificationError* 的别名。

随机生成

ssl.**RAND_bytes**(*num*)

返回 *num* 个高加密强度伪随机字节数据。如果 PRNG 未使用足够的数据作为随机种子或者如果当前 RAND 方法不支持该操作则会引发*SSLError*。*RAND_status()* 可被用来检查 PRNG 的状态而*RAND_add()* 可被用来为 PRNG 设置随机种子。

对于几乎所有应用程序都更推荐使用*os.urandom()*。

请阅读维基百科文章 Cryptographically secure pseudorandom number generator (CSPRNG) 以了解对于高加密强度生成器的具体要求。

Added in version 3.3.

ssl.**RAND_status**()

如果 SSL 伪随机数生成器已使用‘足够的’随机性作为种子则返回 `True`，否则返回 `False`。你可以使用 `ssl.RAND_egd()` 和*ssl.RAND_add()* 来增加伪随机数生成器的随机性。

ssl.**RAND_add**(*bytes*, *entropy*)

将给定的 *bytes* 混合到 SSL 伪随机数生成器中。参数 *entropy* (浮点数) 是字符串中包含的熵值的下限（因此可以始终使用 `0.0`)。请参阅 **RFC 1750** 了解有关熵源的更多信息。

在 3.5 版本发生变更: 现在接受可写的*字节类对象*。

证书处理

ssl.**cert_time_to_seconds**(*cert_time*)

返回距离 Unix 纪元零时的秒数，给定的 `cert_time` 字符串代表来自证书的”notBefore” 或”notAfter” 日期值，采用 `"%b %d %H:%M:%S %Y %Z"` strptime 格式（C 区域)。

以下为示例代码:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

”notBefore” 或”notAfter” 日期值必须使用 GMT (**RFC 5280**)。

在 3.5 版本发生变更: 将输入时间解读为 UTC 时间，基于输入字符串中指明的’GMT’ 时区。在之前使用的是本地时区。返回一个整数（不带输入格式中秒的分数部分）

ssl.**get_server_certificate**(*addr*, *ssl_version=PROTOCOL_TLS_CLIENT*, *ca_certs=None*[, *timeout*])

给定使用 SSL 保护的服务器的地址 `addr`，形式为一个 (*hostname*, *port-number*) 对，获取该服务器的证书，并返回为 PEM 编码的字符串。如果指定了 `ssl_version`，则使用该版本的 SSL 协议尝试连接该服务器。如果指定了 *ca_certs*，它应当是一个包含根证书列表的文件，与*SSLContext.load_verify_locations()* 中 *cafile* 形参所使用的格式相同。该调用将尝试根据该根证书集来难服务器的证书，如果验证失败则调用也将失败。可以通过 `timeout` 形参来指定超时限制。

在 3.3 版本发生变更: 此函数现在是 IPv6 兼容的。-compatible.

在 3.5 版本发生变更: 默认的 *ssl_version* 从*PROTOCOL_SSLv3* 改为*PROTOCOL_TLS* 以保证与现代服务器的最大兼容性。

在 3.10 版本发生变更: 加入 *timeout* 参数。

ssl.**DER_cert_to_PEM_cert**(*DER_cert_bytes*)

根据给定的 DER 编码字节块形式的证书，返回同一证书的 PEM 编码字符串版本。

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)

根据给定的 ASCII PEM 字符串形式的证书，返回同一证书的 DER 编码字节序列。

ssl.**get_default_verify_paths**()

返回包含 OpenSSL 的默认 cafile 和 capath 的路径的命名元组。此路径与*SSLContext.set_default_verify_paths()* 所使用的相同。返回值是一个*named tuple* `DefaultVerifyPaths`:

- `cafile` - 解析出的 cafile 路径或者如果文件不存在则为 `None`,
- `capath` - 解析出的 capath 路径或者如果目录不存在则为 `None`,
- `openssl_cafile_env` - 指向一个 cafile 的 OpenSSL 环境键,
- `openssl_cafile` - 一个 cafile 的硬编码路径,
- `openssl_capath_env` - 指向一个 capath 的 OpenSSL 环境键,
- `openssl_capath` - 一个 capath 目录的硬编码路径

Added in version 3.4.

ssl.**enum_certificates**(*store_name*)

从 Windows 的系统证书库中检索证书。*store_name* 可以是 `CA`, `ROOT` 或 `MY` 中的一个。Windows 也可能会提供额外的证书库。

此函数返回一个包含 (cert_bytes, encoding_type, trust) 元组的列表。encoding_type 指明 cert_bytes 的编码格式。它可以为 `x509_asn` 以表示 X.509 ASN.1 数据或是 `pkcs_7_asn` 以表示 PKCS#7 ASN.1 数据。trust 以 OIDS 集合的形式指明证书的目的，或者如果证书对于所有目的都可以信任则为 `True`。

示例:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability: Windows.

Added in version 3.4.

ssl.**enum_crls**(*store_name*)

Windows 的系统证书库中检索 CRL。*store_name* 可以是 `CA`, `ROOT` 或 `MY` 中的一个。Windows 也可能会提供额外的证书库。

此函数返回一个包含 (cert_bytes, encoding_type, trust) 元组的列表。encoding_type 指明 cert_bytes 的编码格式。它可以为 `x509_asn` 以表示 X.509 ASN.1 数据或是 `pkcs_7_asn` 以表示 PKCS#7 ASN.1 数据。

Availability: Windows.

Added in version 3.4.

常量

所有常量现在都是*enum.IntEnum* 或*enum.IntFlag* 多项集的成员。

Added in version 3.6.

ssl.**CERT_NONE**

SSLContext.verify_mode 可能的取值。*PROTOCOL_TLS_CLIENT* 除外，这是默认的模式。对于客户端套接字，几乎任何证书都会被接受。验证错误，如不受信任或过期的证书等，会被忽略并且不会中止 TLS/SSL 握手。

在服务器模式下，不会从客户端请求任何证书，因此客户端不会发送任何用于客户端证书身份验证的证书。

参见下文对于*安全考量* 的讨论。

ssl.**CERT_OPTIONAL**

SSLContext.verify_mode 可能的取值。在客户端模式下，*CERT_OPTIONAL* 具有与*CERT_REQUIRED* 相同的含义。对于客户端套接字推荐改用*CERT_REQUIRED*。

在服务器模式下，客户端证书请求会被发送给客户端。客户端可以忽略请求也可以发送一个证书以执行 TLS 客户端证书身份验证。如果客户端选择发送证书，则将对其执行验证。任何验证错误都将立即中止 TLS 握手。

使用此设置要求将一组有效的 CA 证书传递给*SSLContext.load_verify_locations()*。

ssl.**CERT_REQUIRED**

SSLContext.verify_mode 可能的取值。在此模式下，需要从套接字连接的另一端获取证书；如果未提供证书，或验证失败则将引发*SSLError*。此模式 **不能**在客户端模式下对证书进行验证因为它不会匹配主机名。*check_hostname* 也必须被启用以验证证书的真实性。*PROTOCOL_TLS_CLIENT* 会使用*CERT_REQUIRED* 并默认启用*check_hostname*。

对于服务器套接字，此模式会提供强制性的 TLS 客户端证书验证。客户端证书请求会被发送给客户端并且客户端必须提供有效且受信任的证书。

使用此设置要求将一组有效的 CA 证书传递给*SSLContext.load_verify_locations()*。

class ssl.**VerifyMode**

CERT_* 常量的*enum.IntEnum* 多项集。

Added in version 3.6.

ssl.**VERIFY_DEFAULT**

SSLContext.verify_flags 可能的取值。在此模式下，证书吊销列表（CRL）并不会被检查。OpenSSL 默认不要求也不验证 CRL。

Added in version 3.4.

ssl.**VERIFY_CRL_CHECK_LEAF**

SSLContext.verify_flags 可能的取值。在此模式下，只会检查对等证书而不检查任何中间 CA 证书。此模式要求提供由对等证书颁发者（其直接上级 CA）签名的有效 CRL。如果未使用*SSLContext.load_verify_locations* 加载正确的 CRL，则验证将失败。

Added in version 3.4.

ssl.**VERIFY_CRL_CHECK_CHAIN**

SSLContext.verify_flags 可能的取值。在此模式下，会检查对等证书链中所有证书的 CRL。

Added in version 3.4.

ssl.**VERIFY_X509_STRICT**

SSLContext.verify_flags 可能的取值，用于禁用已损坏 X.509 证书的绕过操作。

Added in version 3.4.

ssl.**VERIFY_ALLOW_PROXY_CERTS**

SSLContext.verify_flags 的可能取值，启用代理证书验证。

Added in version 3.10.

ssl.**VERIFY_X509_TRUSTED_FIRST**

SSLContext.verify_flags 可能的取值。它指示 OpenSSL 在构建用于验证某个证书的信任链时首选受信任的证书。此旗标将默认被启用。

Added in version 3.4.4.

ssl.**VERIFY_X509_PARTIAL_CHAIN**

SSLContext.verify_flags 的可能取值。它指示 OpenSSL 接受信任存储中的中间 CA 作为信任锚，与自我签名的根 CA 证书的方式相同。这样就能信任中间 CA 颁发的证书，而不一定非要去信任其祖先的根 CA。

Added in version 3.10.

class ssl.**VerifyFlags**

VERIFY_* 常量的*enum.IntFlag* 多项集。

Added in version 3.6.

ssl.**PROTOCOL_TLS**

选择客户端和服务器均支持的最高协议版本。此选项名称并不准确，实际上"SSL" 和"TLS" 协议均可被选择。

Added in version 3.6.

自 3.10 版本弃用: TLS 客户端和服务器需要不同的默认设置来实现安全通信。通用的 TLS 协议常量已废弃，而采用*PROTOCOL_TLS_CLIENT* 和*PROTOCOL_TLS_SERVER*。

ssl.**PROTOCOL_TLS_CLIENT**

自动协商为客户端和服务器都支持的最高版本协议，并配置当前上下文客户端的连接。该协议默认启用*CERT_REQUIRED* 和*check_hostname*。

Added in version 3.6.

ssl.**PROTOCOL_TLS_SERVER**

自动协商为客户端和服务器都支持的最高版本协议，并配置上下文服务器端的连接。

Added in version 3.6.

ssl.**PROTOCOL_SSLv23**

PROTOCOL_TLS 的别名。

自 3.6 版本弃用: 请改用*PROTOCOL_TLS*。

ssl.**PROTOCOL_SSLv3**

选择 SSL 版本 3 作为通道加密协议。

如果 OpenSSL 是用 no-ssl3 选项编译的，则该协议不可用。

> **警告**
>
> SSL 版本 3 并不安全。极不建议使用它。

自 3.6 版本弃用: OpenSSL 已经废弃了所有特定于版本的协议。请换用带有*SSLContext.minimum_version* 和*SSLContext.maximum_version* 的默认协议*PROTOCOL_TLS_SERVER* 或*PROTOCOL_TLS_CLIENT* 。

ssl.**PROTOCOL_TLSv1**

选择 TLS 版本 1.0 作为通道加密协议。

自 3.6 版本弃用: OpenSSL 已经废弃了所有特定于版本的协议。

ssl.**PROTOCOL_TLSv1_1**

选择 TLS 版本 1.1 作为通道加密协议。仅适用于 openssl 版本 1.0.1+。

Added in version 3.4.

自 3.6 版本弃用: OpenSSL 已经废弃了所有特定于版本的协议。

ssl.**PROTOCOL_TLSv1_2**

选用 TLS 1.2 版本作为隧道加密协议。只适用于 openssl 1.0.1 以上版本。

Added in version 3.4.

自 3.6 版本弃用: OpenSSL 已经废弃了所有特定于版本的协议。

ssl.**OP_ALL**

对存在于其他 SSL 实现中的各种缺陷启用绕过操作。默认会设置此选项。没有必要设置与 OpenSSL 的 SSL_OP_ALL 常量同名的旗标。

Added in version 3.2.

ssl.**OP_NO_SSLv2**

阻止 SSLv2 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 SSLv2 作为协议版本。

Added in version 3.2.

自 3.6 版本弃用: SSLv2 已被弃用

ssl.**OP_NO_SSLv3**

阻止 SSLv3 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 SSLv3 作为协议版本。

Added in version 3.2.

自 3.6 版本弃用: SSLv3 已被弃用

ssl.**OP_NO_TLSv1**

阻止 TLSv1 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1 作为协议版本。

Added in version 3.2.

自 3.7 版本弃用: 此选项自 OpenSSL 1.1.0 起已被弃用，请改用新的*SSLContext.minimum_version* 和*SSLContext.maximum_version*。

ssl.**OP_NO_TLSv1_1**

阻止 TLSv1.1 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1.1 作为协议版本。仅适用于 openssl 版本 1.0.1+。

Added in version 3.4.

自 3.7 版本弃用: 此选项自 OpenSSL 1.1.0 起已被弃用。

ssl.**OP_NO_TLSv1_2**

阻止 TLSv1.2 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1.2 作为协议版本。仅适用于 openssl 版本 1.0.1+。

Added in version 3.4.

自 3.7 版本弃用: 此选项自 OpenSSL 1.1.0 起已被弃用。

ssl.**OP_NO_TLSv1_3**

阻止 TLSv1.3 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1.3 作为协议版本。TLS 1.3 适用于 OpenSSL 1.1.1 或更新的版本。当 Python 编译是基于较旧版本的 OpenSSL 时，该旗标默认为 *0*。

Added in version 3.6.3.

自 3.7 版本弃用: 此选项自 OpenSSL 1.1.0 起已被弃用。它被添加到 2.7.15 和 3.6.3 是为了向下兼容 OpenSSL 1.0.2。

ssl.**OP_NO_RENEGOTIATION**

禁用所有 TLSv1.2 和更早版本的重协商操作。不发送 HelloRequest 消息，并忽略通过 ClientHello 发起的重协商请求。

此选项仅适用于 OpenSSL 1.1.0h 及更新的版本。

Added in version 3.7.

ssl.**OP_CIPHER_SERVER_PREFERENCE**

使用服务器的密码顺序首选项，而不是客户端的首选项。此选项在客户端套接字和 SSLv2 服务器套接字上无效。

Added in version 3.3.

ssl.**OP_SINGLE_DH_USE**

防止对于单独 SSL 会话重用相同的 DH 密钥。这会提升前向保密性但需要更多的计算资源。此选项仅适用于服务器套接字。

Added in version 3.3.

ssl.**OP_SINGLE_ECDH_USE**

防止对于单独 SSL 会话重用相同的 ECDH 密钥。这会提升前向保密性但需要更多的计算资源。此选项仅适用于服务器套接字。

Added in version 3.3.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**

在 TLS 1.3 握手中发送虚拟更改密码规格（CCS）消息以使得 TLS 1.3 连接看起来更像是 TLS 1.2 连接。

此选项仅适用于 OpenSSL 1.1.1 及更新的版本。

Added in version 3.8.

ssl.**OP_NO_COMPRESSION**

在 SSL 通道上禁用压缩。这适用于应用协议支持自己的压缩方案的情况。

Added in version 3.3.

class ssl.**Options**

OP_* 常量的*enum.IntFlag* 多项集。

ssl.**OP_NO_TICKET**

阻止客户端请求会话凭据。

Added in version 3.6.

ssl.**OP_IGNORE_UNEXPECTED_EOF**

忽略 TLS 连接的意外关闭。

此选项仅适用于 OpenSSL 3.0.0 及更新的版本。

Added in version 3.10.

ssl.**OP_ENABLE_KTLS**

启用内核 TLS。为了利用该特征，OpenSSL 编译时必须附带对它的支持，并且协商的密码套件和扩展必须被它所支持（受支持项的列表可能因平台和内核版本而有所变化）。

请注意当启用内核 TLS 时某些加解密操作将由内核直接执行而不是通过任何可用的 OpenSSL 提供程序。这可能并不是你想要的，例如，当应用程序要求所有加解密操作由 FIPS 提供程序执行时。

此选项仅适用于 OpenSSL 3.0.0 及更新的版本。

Added in version 3.12.

ssl.**OP_LEGACY_SERVER_CONNECT**

允许只在 OpenSSL 和未打补丁的服务器之间进行旧式的不安全协商。

Added in version 3.12.

ssl.**HAS_ALPN**

OpenSSL 库是否具有对 **RFC 7301** 中描述的 *应用层协议协商* TLS 扩展的内置支持。

Added in version 3.5.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**

OpenSSL 库是否具有对不检测目标通用名称的内置支持且*SSLContext.hostname_checks_common_name* 为可写状态。

Added in version 3.7.

ssl.**HAS_ECDH**

OpenSSL 库是否具有对基于椭圆曲线的 Diffie-Hellman 密钥交换的内置支持。此常量应当为真值，除非发布者明确地禁用了此功能。

Added in version 3.3.

ssl.**HAS_SNI**

OpenSSL 库是否具有对 *服务器名称提示扩展*（在 **RFC 6066** 中定义）的内置支持。

Added in version 3.2.

ssl.**HAS_NPN**

OpenSSL 库是否具有对 *应用层协议协商* 中描述的 *下一协议协商*的内置支持。当此常量为真值时，你可以使用*SSLContext.set_npn_protocols()* 方法来公告你想要支持的协议。

Added in version 3.3.

ssl.**HAS_SSLv2**

OpenSSL 库是否具有对 SSL 2.0 协议的内置支持。

Added in version 3.7.

ssl.**HAS_SSLv3**

OpenSSL 库是否具有对 SSL 3.0 协议的内置支持。

Added in version 3.7.

ssl.**HAS_TLSv1**

OpenSSL 库是否具有对 TLS 1.0 协议的内置支持。

Added in version 3.7.

ssl.**HAS_TLSv1_1**

OpenSSL 库是否具有对 TLS 1.1 协议的内置支持。

Added in version 3.7.

ssl.**HAS_TLSv1_2**

OpenSSL 库是否具有对 TLS 1.2 协议的内置支持。

Added in version 3.7.

ssl.**HAS_TLSv1_3**

OpenSSL 库是否具有对 TLS 1.3 协议的内置支持。

Added in version 3.7.

ssl.**HAS_PSK**

OpenSSL 库是否具有对 TLS-PSK 的内置支持。

Added in version 3.13.

ssl.**CHANNEL_BINDING_TYPES**

受支持的 TLS 通道绑定类型组成的列表。此列表中的字符串可被用作传给*SSLSocket.get_channel_binding()* 的参数。

Added in version 3.3.

ssl.**OPENSSL_VERSION**

解释器所加载的 OpenSSL 库的版本字符串:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Added in version 3.2.

ssl.**OPENSSL_VERSION_INFO**

代表 OpenSSL 库的版本信息的五个整数所组成的元组:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Added in version 3.2.

ssl.**OPENSSL_VERSION_NUMBER**

OpenSSL 库的原始版本号，以单个整数表示:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

Added in version 3.2.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**

ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**

ALERT_DESCRIPTION_*

来自 **RFC 5246** 等文档的警报描述。IANA TLS Alert Registry 中包含了这个列表及对定义其含义的 RFC 引用。

被用作*SSLContext.set_servername_callback()* 中的回调函数的返回值。

Added in version 3.4.

class ssl.**AlertDescription**

ALERT_DESCRIPTION_* 常量的*enum.IntEnum* 多项集。

Added in version 3.6.

Purpose.**SERVER_AUTH**

用于*create_default_context()* 和*SSLContext.load_default_certs()* 的参数。表示上下文可用于验证网络服务器（因此，它将被用于创建客户端套接字）。

Added in version 3.4.

Purpose.**CLIENT_AUTH**

用于*create_default_context()* 和*SSLContext.load_default_certs()* 的参数。表示上下文可用于验证网络客户（因此，它将被用于创建服务器端套接字）。

Added in version 3.4.

class ssl.**SSLErrorNumber**

SSL_ERROR_* 常量的*enum.IntEnum* 多项集。

Added in version 3.6.

class ssl.**TLSVersion**

SSLContext.maximum_version 和*SSLContext.minimum_version* 中的 SSL 和 TLS 版本的*enum.IntEnum* 多项集。

Added in version 3.7.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**

受支持的最低和最高 SSL 或 TLS 版本。这些常量被称为魔术常量。它们的值并不反映可用的最低和最高 TLS/SSL 版本。

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

TLSVersion.**TLSv1_2**

TLSVersion.**TLSv1_3**

SSL 3.0 至 TLS 1.3。

自 3.10 版本弃用: 所有*TLSVersion* 成员，除*TLSVersion.TLSv1_2* 和*TLSVersion.TLSv1_3* 之外均已废弃。

19.3.2 SSL 套接字

class ssl.**SSLSocket**(*socket.socket*)

SSL 套接字提供了套接字对象 的下列方法:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (但不允许传入非零的 `flags` 参数)
- *send()*, *sendall()* (具有同样的限制)
- *sendfile()* (但*os.sendfile* 将仅用于纯文本套接字，在其他情况下将使用*send()*)
- *shutdown()*

但是，由于 SSL（和 TLS）协议在 TCP 之上具有自己的框架，因此 SSL 套接字抽象在某些方面可能与常规的 OS 层级套接字存在差异。特别是要查看非阻塞型套接字说明。

SSLSocket 的实例必须使用*SSLContext.wrap_socket()* 方法来创建。

在 3.5 版本发生变更: 新增了 `sendfile()` 方法。

在 3.5 版本发生变更: `shutdown()` 不会在每次接收或发送字节数据后重置套接字超时。现在套接字超时为关闭的最大总持续时间。

自 3.6 版本弃用: 直接创建*SSLSocket* 实例的做法已被弃用，请使用*SSLContext.wrap_socket()* 来包装套接字。

在 3.7 版本发生变更: *SSLSocket* 的实例必须使用*wrap_socket()* 来创建。在较早的版本中，直接创建实例是可能的。但这从未被记入文档或是被正式支持。

在 3.10 版本发生变更: Python 内部现在使用 SSL_read_ex 和 SSL_write_ex。这些函数支持读取和写入大于 2GB 的数据。写入零长数据不再出现违反协议的错误。

SSL 套接字还具有下列方法和属性:

SSLSocket.**read**(*len=1024*, *buffer=None*)

从 SSL 套接字读取至多 *len* 个字节的数据并将结果作为 bytes 实例返回。如果指定了 *buffer*，则改为读取到缓冲区，并返回所读取的字节数。

如果套接字为非阻塞型 则会引发*SSLWantReadError* 或*SSLWantWriteError* 且读取将阻塞。

由于在任何时候重新协商都是可能的，因此调用*read()* 也可能导致写入操作。

在 3.5 版本发生变更: 套接字超时在每次接收或发送字节数据后不会再被重置。现在套接字超时为读取至多 *len* 个字节数据的最大总持续时间。

自 3.6 版本弃用: 请使用 recv() 来代替*read()*。

SSLSocket.**write**(*buf*)

将 *buf* 写入到 SSL 套接字并返回所写入的字节数。*buf* 参数必须为支持缓冲区接口的对象。

如果套接字为非阻塞型 则会引发*SSLWantReadError* 或*SSLWantWriteError* 且读取将阻塞。

由于在任何时候重新协商都是可能的，因此调用*write()* 也可能导致读取操作。

在 3.5 版本发生变更: 套接字超时在每次接收或发送字节数据后不会再被重置。现在套接字超时为写入 *buf* 的最大总持续时间。

自 3.6 版本弃用: 请使用 send() 来代替*write()*。

备注

read() 和*write()* 方法是读写未加密的应用级数据，并将其解密/加密为带加密的线路级数据的低层级方法。这些方法需要有激活的 SSL 连接，即握手已完成而*SSLSocket.unwrap()* 尚未被调用。

通常你应当使用套接字 API 方法例如*recv()* 和*send()* 来代替这些方法。

SSLSocket.**do_handshake**()

执行 SSL 设置握手。

在 3.4 版本发生变更: 当套接字的*context* 的*check_hostname* 属性为真值时此握手方法还会执行 match_hostname()。

在 3.5 版本发生变更: 套接字超时在每次接收或发送字节数据时不会再被重置。现在套接字超时为握手的最大总持续时间。

在 3.7 版本发生变更: 主机名或 IP 地址会在握手期间由 OpenSSL 进行匹配。函数 match_hostname() 不再被使用。在 OpenSSL 拒绝主机名或 IP 地址的情况下，握手将提前被中止并向对等方发送 TLS 警告消息。

SSLSocket.**getpeercert**(*binary_form=False*)

如果连接另一端的对等方没有证书，则返回 None。如果 SSL 握手还未完成，则会引发*ValueError*。

如果 binary_form 形参为*False*，并且从对等方接收到了证书，此方法将返回一个*dict* 实例。如果证书未通过验证，则字典将为空。如果证书通过验证，它将返回由多个密钥组成的字典，其中包括 subject (证书颁发给的主体) 和 issuer (颁发证书的主体)。如果证书包含一个 *Subject Alternative Name* 扩展的实例 (see **RFC 3280**)，则字典中还将有一个 subjectAltName 键。

subject 和 issuer 字段都是包含在证书中相应字段的数据结构中给出的相对专有名称（RDN）序列的元组，每个 RDN 均为 name-value 对的序列。这里是一个实际的示例:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

如果 binary_form 形参为*True*，并且提供了证书，此方法会将整个证书的 DER 编码形式作为字节序列返回，或者如果对等方未提供证书则返回*None*。对等方是否提供证书取决于 SSL 套接字的角色:

- 对于客户端 SSL 套接字，服务器将总是提供证书，无论是否需要进行验证;
- 对于服务器 SSL 套接字，客户端将仅在服务器要求时才提供证书；因此如果你使用了*CERT_NONE* (而不是*CERT_OPTIONAL* 或*CERT_REQUIRED*) 则*getpeercert()* 将返回*None*。

另请参阅*SSLContext.check_hostname*。

在 3.2 版本发生变更: 返回的字典包括额外的条目例如 issuer 和 notBefore。

在 3.4 版本发生变更: 如果握手未完成则会引发*ValueError*。返回的字典包括额外的 X509v3 扩展条目例如 crlDistributionPoints, caIssuers 和 OCSP URI。

在 3.9 版本发生变更: IPv6 地址字符串不再附带末尾换行符。

SSLSocket.**get_verified_chain**()

返回由 SSL 通道另一端以 DER 编码字节列表形式提供的经过验证的证书链。如果证书验证被禁用则此方法的行为与*get_unverified_chain()* 相同。

Added in version 3.13.

SSLSocket.**get_unverified_chain**()

返回由 SSL 通道另一端以 DER 编码字节列表形式提供的原始证书链。

Added in version 3.13.

SSLSocket.**cipher**()

返回由三个值组成的元组，其中包含所使用的密码名称，定义其使用方式的 SSL 协议版本，以及所使用的加密比特位数。如果尚未建立连接，则返回 None。

SSLSocket.**shared_ciphers**()

返回在客户端和服务器均可用的密码列表。所返回列表的每个条目都是由三个值组成的元组其中包含密码名称、定义其使用方式的 SSL 协议版本，以及密码所使用的加密比特位数量。如果连接尚未建立或套接字为客户端套接字则*shared_ciphers()* 将返回 None。

Added in version 3.5.

SSLSocket.**compression**()

以字符串形式返回所使用的压缩算法，或者如果连接没有使用压缩则返回 None。

如果高层级的协议支持自己的压缩机制，你可以使用*OP_NO_COMPRESSION* 来禁用 SSL 层级的压缩。

Added in version 3.3.

SSLSocket.**get_channel_binding**(*cb_type='tls-unique'*)

为当前连接获取字节串形式的通道绑定数据。如果尚未连接或握手尚未完成则返回 None。

cb_type 形参允许选择需要的通道绑定类型。有效的通道绑定类型在*CHANNEL_BINDING_TYPES* 列表中列出。目前只支持由 **RFC 5929** 所定义的'tls-unique' 通道绑定。如果请求了一个不受支持的通道绑定类型则将引发*ValueError*。

Added in version 3.3.

SSLSocket.**selected_alpn_protocol**()

返回在 TLS 握手期间所选择的协议。如果*SSLContext.set_alpn_protocols()* 未被调用，如果另一方不支持 ALPN，如果此套接字不支持任何客户端所用的协议，或者如果握手尚未发生，则将返回 None。

Added in version 3.5.

SSLSocket.**selected_npn_protocol**()

返回在 Return the higher-level protocol that was selected during the TLS/SSL 握手期间所选择的高层级协议。如果*SSLContext.set_npn_protocols()* 未被调用，或者如果另一方不支持 NPN，或者如果握手尚未发生，则将返回 None。

Added in version 3.3.

自 3.10 版本弃用: NPN 已被 ALPN 取代。

SSLSocket.**unwrap**()

执行 SSL 关闭握手，这会从下层的套接字中移除 TLS 层，并返回下层的套接字对象。这可被用来通过一个连接将加密操作转为非加密。返回的套接字应当总是被用于同连接另一方的进一步通信，而不是原始的套接字。

SSLSocket.**verify_client_post_handshake**()

向一个 TLS 1.3 客户端请求握手后身份验证（PHA）。只有在初始 TLS 握手之后且双方都启用了 PHA 的情况下才能为服务器端套接字的 TLS 1.3 连接启用 PHA，参见*SSLContext.post_handshake_auth*。

此方法不会立即执行证书交换。服务器端会在下一次写入事件期间发送 CertificateRequest 并期待客户端在下一次读取事件期间附带证书进行响应。

如果有任何前置条件未被满足（例如非 TLS 1.3，PHA 未启用），则会引发*SSLError*。

> **备注**
>
> 仅在 OpenSSL 1.1.1 且 TLS 1.3 被启用时可用。没有 TLS 1.3 支持，此方法将引发*NotImplementedError*。

Added in version 3.8.

SSLSocket.**version**()

以字符串形式返回由连接协商确定的实际 SSL 协议版本，或者如果未建立安全连接则返回 None。在撰写本文档时，可能的返回值包括 "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" 和 "TLSv1.2"。最新的 OpenSSL 版本可能会定义更多的返回值。

Added in version 3.5.

SSLSocket.**pending**()

返回在连接上等待被读取的已解密字节数。

SSLSocket.**context**

该 SSL 套接字所关联的*SSLContext* 对象。

Added in version 3.2.

SSLSocket.**server_side**

一个布尔值，对于服务器端套接字为 True 而对于客户端套接字则为 False。

Added in version 3.2.

SSLSocket.**server_hostname**

服务器的主机名: *str* 类型，对于服务器端套接字或者如果构造器中未指定主机名则为 None。

Added in version 3.2.

在 3.7 版本发生变更: 现在该属性将始终为 ASCII 文本。当 server_hostname 为一个国际化域名（IDN）时，该属性现在会保存为 A 标签形式 ("xn--pythn-mua.org") 而非 U 标签形式 ("pythön.org")。

SSLSocket.**session**

用于 SSL 连接的*SSLSession*。该会话将在执行 TLS 握手后对客户端和服务器端套接字可用。对于客户端套接字该会话可以在调用*do_handshake()* 之前被设置以重用一个会话。

Added in version 3.6.

SSLSocket.**session_reused**

Added in version 3.6.

19.3.3 SSL 上下文

Added in version 3.2.

SSL 上下文可保存各种比单独 SSL 连接寿命更长的数据，例如 SSL 配置选项，证书和私钥等。它还可为服务器端套接字管理缓存，以加快来自相同客户端的重复连接。

class ssl.**SSLContext**(*protocol=None*)

创建一个新的 SSL 上下文。你可以传入 *protocol*，它必须为此模块中定义的 PROTOCOL_* 常量之一。该形参指定要使用哪个 SSL 协议版本。通常，服务器会选择一个特定的协议版本，而客户端必须适应服务器的选择。大多数版本都不能与其他版本互操作。如果未指定，则默认值为*PROTOCOL_TLS*；它提供了与其他版本的最大兼容性。

这个表显示了客户端（横向）的哪个版本能够连接服务器（纵向）的哪个版本。

客户端 / 服务器	SSLv2	SSLv3	TLS[3]	TLSv1	TLSv1.1	TLSv1.2
SSLv2	是	否	否[1]	否	否	否
SSLv3	否	是	否[2]	否	否	否
TLS (SSLv23)[3]	否[1]	否[2]	是	是	是	是
TLSv1	否	否	是	是	否	否
TLSv1.1	否	否	是	否	是	否
TLSv1.2	否	否	是	否	否	是

备注

参见

create_default_context() 让*ssl* 为特定目标选择安全设置。

在 3.6 版本发生变更: 上下文会使用安全的默认值来创建。默认设置的选项有*OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* 和*OP_NO_SSLv3* (*PROTOCOL_SSLv3* 除外)。初始密码套件列表只包含 HIGH 密码，而不包含 NULL 密码和 MD5 密码。

[3] TLS 1.3 协议在 OpenSSL >= 1.1.1 中设置*PROTOCOL_TLS* 时可用。没有专门针对 TLS 1.3 的 PROTOCOL 常量。
[1] *SSLContext* 默认设置*OP_NO_SSLv2* 以禁用 SSLv2。
[2] *SSLContext* 默认设置*OP_NO_SSLv3* 以禁用 SSLv3。

自 3.10 版本弃用: 不带协议参数的*SSLContext* 已废弃。将来，上下文类会要求使用*PROTOCOL_TLS_CLIENT* 或*PROTOCOL_TLS_SERVER* 协议。

在 3.10 版本发生变更: 现在默认的密码套件只包含安全的 AES 和 ChaCha20 密码，具有前向保密性和安全级别 2。禁止使用少于 2048 位的 RSA 和 DH 密钥以及少于 224 位的 ECC 密钥。*PROTOCOL_TLS* 、*PROTOCOL_TLS_CLIENT* 和*PROTOCOL_TLS_SERVER* 至少使用 TLS 1.2 版本。

备注

SSLContext 一旦被某个连接使用它将只支持有限的变异。在内部信任存储中添加新证书是允许的，但是更改密码、验证设置或 mTLS 证书则可能导致令人吃惊的行为。

备注

SSLContext 被设计为可由多个连接共享和使用。因此，只要在被某个连接使用后不重新配置那么它就是线程安全的。

SSLContext 对象具有以下方法和属性:

SSLContext.**cert_store_stats**()

获取以字典表示的有关已加载的 X.509 证书数量，被标记为 CA 证书的 X.509 证书数量以及证书吊销列表的统计信息。

具有一个 CA 证书和一个其他证书的上下文示例:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Added in version 3.4.

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)

加载一个私钥及对应的证书。*certfile* 字符串必须为以 PEM 格式表示的单个文件路径，该文件中包含证书以及确立证书真实性所需的任意数量的 CA 证书。如果存在 *keyfile* 字符串，它必须指向一个包含私钥的文件。否则私钥也将从 *certfile* 中提取。请参阅证书 中的讨论来了解有关如何将证书存储至 *certfile* 的更多信息。

password 参数可以是一个函数，调用时将得到用于解密私钥的密码。它在私钥被加密且需要密码时才会被调用。它调用时将不带任何参数，并且应当返回一个字符串、字节串或字节数组。如果返回值是一个字符串，在用它解密私钥之前它将以 UTF-8 进行编码。或者也可以直接将字符串、字节串或字节数组值作为 *password* 参数提供。如果私钥未被加密且不需要密码则它将被忽略。

如果未指定 *password* 参数且需要一个密码，将会使用 OpenSSL 内置的密码提示机制来交互式地提示用户输入密码。

如果私钥不能匹配证书则会引发*SSLError*。

在 3.3 版本发生变更: 新增可选参数 *password*。

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

从默认位置加载一组默认的" 证书颁发机构" (CA) 证书。在 Windows 上它将从 CA 和 ROOT 系统存储中加载 CA 证书。在所有系统上它会调用*SSLContext.set_default_verify_paths()* 。将来该方法也可能会从其他位置加载 CA 证书。

purpose 旗标指明要加载哪种 CA 证书。默认设置*Purpose.SERVER_AUTH* 将加载被标记且被信任用于 TLS Web 服务器验证（客户端套接字）的证书。*Purpose.CLIENT_AUTH* 则会加载用于在服务器端进行客户端证书验证的 CA 证书。

Added in version 3.4.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

当*verify_mode*不为*CERT_NONE*时加载一组用于验证其他对等方证书的" 证书颁发机构" (CA) 证书。必须至少指定 *cafile* 或 *capath* 中的一个。

此方法还可加载 PEM 或 DER 格式的证书吊销列表 (CRL)，为此必须正确配置*SSLContext.verify_flags*。

如果存在 *cafile* 字符串，它应为 PEM 格式的级联 CA 证书文件的路径。请参阅证书 中的讨论来了解有关如何处理此文件中的证书的更多信息。

如果存在 *capath* 字符串，它将为包含多个 PEM 格式的 CA 证书的目录的路径，并遵循 OpenSSL 专属布局。

如果存在 *cadata* 对象，它应为一个或多个 PEM 编码的证书的 ASCII 字符串或者 DER 编码的证书的*bytes-like object*。与 *capath* 一样 PEM 编码的证书之外的多余行会被忽略，但至少要有一个证书。

在 3.4 版本发生变更: 新增可选参数 *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)

获取已离开法人" 证书颁发机构" (CA) 证书列表。如果 `binary_form` 形参为*False*则每个列表条目都是一个类似于*SSLSocket.getpeercert()*输出的字典。在其他情况下此方法将返回一个 DER 编码的证书的列表。返回的列表不包含来自 *capath* 的证书，除非 SSL 连接请求并加载了一个证书。

备注

capath 目录中的证书不会被加载，除非它们已至少被使用过一次。

Added in version 3.4.

SSLContext.**get_ciphers**()

获取已启用密码的列表。该列表将按密码的优先级排序。参见*SSLContext.set_ciphers()*。

示例:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

Added in version 3.6.

SSLContext.**set_default_verify_paths**()

从构建 OpenSSL 库时定义的文件系统路径中加载一组默认的" 证书颁发机构" (CA) 证书。不幸的是，没有一种简单的方式能知道此方法是否执行成功：如果未找到任何证书也不会返回错误。不过，当 OpenSSL 库是作为操作系统的一部分被提供时，它的配置应当是正确的。

SSLContext.**set_ciphers**(*ciphers*)

为使用此上下文创建的套接字设置可用密码。它应当为 OpenSSL 密码列表格式 的字符串。如果没有可被选择的密码（由于编译时选项或其他配置禁止使用已指定的所有密码），则将引发*SSLError*。

> **备注**
>
> 在连接后，SSL 套接字的*SSLSocket.cipher()* 方法将给出当前所选择的密码。
>
> TLS 1.3 密码套件不能通过*set_ciphers()* 禁用。

SSLContext.**set_alpn_protocols**(*protocols*)

指定在 SSL/TLS 握手期间套接字应当通告的协议。它应为由 ASCII 字符串组成的列表，例如 `['http/1.1', 'spdy/2']`，按首选顺序排列。协议的选择将在握手期间发生，并依据 **RFC 7301** 来执行。在握手成功后，*SSLSocket.selected_alpn_protocol()* 方法将返回已达成一致的协议。

如果*HAS_ALPN* 为 `False` 则此方法将引发*NotImplementedError*。

Added in version 3.5.

SSLContext.**set_npn_protocols**(*protocols*)

指定在 Specify which protocols the socket should advertise during the SSL/TLS 握手期间套接字应当通告的协议。它应为由字符串组成的列表，例如 `['http/1.1', 'spdy/2']`，按首选顺序排列。协议的选择将在握手期间发生，并将依据 应用层协议协商 来执行。在握手成功后，*SSLSocket.selected_npn_protocol()* 方法将返回已达成一致的协议。

如果*HAS_NPN* 为 `False` 则此方法将引发*NotImplementedError*。

Added in version 3.3.

自 3.10 版本弃用: NPN 已被 ALPN 取代。

SSLContext.**sni_callback**

注册一个回调函数，当 TLS 客户端指定了一个服务器名称提示时，该回调函数将在 SSL/TLS 服务器接收到 TLS Client Hello 握手消息后被调用。服务器名称提示机制的定义见 **RFC 6066** section 3 - Server Name Indication。

每个 `SSLContext` 只能设置一个回调。如果 *sni_callback* 被设置为 `None` 则会禁用回调。对该函数的后续调用将禁用之前注册的回调。

此回调函数将附带三个参数来调用；第一个参数是*ssl.SSLSocket*，第二个参数是代表客户端准备与之通信的服务器的字符串 (或者如果 TLS Client Hello 不包含服务器名称则为*None*) 而第三个参数是原来的*SSLContext*。服务器名称参数为文本形式。对于国际化域名，服务器名称是一个 IDN A 标签 (`"xn--pythn-mua.org"`)。

此回调的一个典型用法是将*ssl.SSLSocket* 的*SSLSocket.context* 属性修改为一个*SSLContext* 类型的新对象，该对象代表与服务器相匹配的证书链。

由于 TLS 连接处于早期协商阶段，因此仅能使用有限的方法和属性例如*SSLSocket.selected_alpn_protocol()* 和*SSLSocket.context*。*SSLSocket.getpeercert()*，*SSLSocket.get_verified_chain()*，*SSLSocket.get_unverified_chain()* *SSLSocket.cipher()* 和*SSLSocket.compression()* 方法要求 TLS 连接已过 TLS Client Hello 步骤因而不会返回有意义的值也不能安全地调用它们。

sni_callback 函数必须返回 `None` 以允许 TLS 协商继续进行。如果想要 TLS 失败，则可以返回常量*ALERT_DESCRIPTION_**。其他返回值将导致 TLS 的致命错误*ALERT_DESCRIPTION_INTERNAL_ERROR*。

如果从 *sni_callback* 函数引发了异常，则 TLS 连接将终止并发出 TLS 致命警告消息*ALERT_DESCRIPTION_HANDSHAKE_FAILURE*。

如果 OpenSSL library 库在构建时定义了 OPENSSL_NO_TLSEXT 则此方法将返回*NotImplementedError*。

Added in version 3.7.

SSLContext.**set_servername_callback**(*server_name_callback*)

这是被保留用于向下兼容的旧式 API。在可能的情况下，你应当改用*sni_callback*。给出的 *server_name_callback* 类似于 *sni_callback*，不同之处在于当服务器主机名是 IDN 编码的国际化域名时，*server_name_callback* 会接收到一个已编码的 U 标签 ("pythön.org")。

如果发生了服务器名称解码错误。TLS 连接将终止并向客户端发出*ALERT_DESCRIPTION_INTERNAL_ERROR* 最严重 TLS 警告消息。

Added in version 3.4.

SSLContext.**load_dh_params**(*dhfile*)

加载密钥生成参数用于 Diffie-Hellman (DH) 密钥交换。使用 DH 密钥交换能以消耗（服务器和客户端的）计算资源为代价提升前向保密性。*dhfile* 参数应当为指向一个包含 PEM 格式的 DH 形参的文件的路径。

此设置不会应用于客户端套接字。你还可以使用*OP_SINGLE_DH_USE* 选项来进一步提升安全性。

Added in version 3.3.

SSLContext.**set_ecdh_curve**(*curve_name*)

为基于椭圆曲线的 Elliptic Curve-based Diffie-Hellman (ECDH) 密钥交换设置曲线名称。ECDH 显著快于常规 DH 同时据信同样安全。*curve_name* 形参应为描述某个知名椭圆曲线的字符串，例如受到广泛支持的曲线 prime256v1。

此设置不会应用于客户端套接字。你还可以使用*OP_SINGLE_ECDH_USE* 选项来进一步提升安全性。

如果*HAS_ECDH* 为 False 则此方法将不可用。

Added in version 3.3.

参见

SSL/TLS & Perfect Forward Secrecy
Vincent Bernat。

SSLContext.**wrap_socket**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

包装一个现有的 Python 套接字 *sock* 并返回一个*SSLContext.sslsocket_class* 的实例 (默认为*SSLSocket*)。返回的 SSL 套接字会关联到相应上下文、设置及证书。*sock* 必须是一个*SOCK_STREAM* 套接字；其他套接字类型均不受支持。

形参 server_side 是一个布尔值，它标明希望从该套接字获得服务器端行为还是客户端行为。

对于客户端套接字，上下文的构造会延迟执行；如果下层的套接字尚未连接，上下文的构造将在对套接字调用 connect() 之后执行。对于服务器端套接字，如果套接字没有远端对等方，它会被视为一个监听套接字，并且服务器端 SSL 包装操作会在通过 accept() 方法所接受的客户端连接上自动执行。此方法可能会引发*SSLError*。

在客户端连接上，可选形参 *server_hostname* 指定所要连接的服务的主机名。这允许单个服务器托管具有单独证书的多个基于 SSL 的服务，很类似于 HTTP 虚拟主机。如果 *server_side* 为真值则指定 *server_hostname* 将引发*ValueError*。

形参 do_handshake_on_connect 指明是否要在调用 socket.connect() 之后自动执行 SSL 握手，还是要通过唤起*SSLSocket.do_handshake()* 方法让应用程序显式地调用它。显式地调用*SSLSocket.do_handshake()* 可给予程序对握手中所涉及的套接字 I/O 阻塞行为的控制。

形参 suppress_ragged_eofs 指明 SSLSocket.recv() 方法应当如何从连接的另一端发送非预期的 EOF 信号。如果指定为*True* (默认值)，它将返回正常的 EOF (空字节串对象) 来响应从下层套接字引发的非预期的 EOF 错误；如果指定为*False*，它将向调用方引发异常。

session，参见*session*。

要将*SSLSocket* 包装在另一个*SSLSocket* 中，请使用*SSLContext.wrap_bio()*。

在 3.5 版本发生变更: 总是允许传送 server_hostname，即使 OpenSSL 没有 SNI。

在 3.6 版本发生变更: 增加了 *session* 参数。

在 3.7 版本发生变更: 此方法返回*SSLContext.sslsocket_class* 的实例而不是硬编码的*SSLSocket*。

SSLContext.**sslsocket_class**

SSLContext.wrap_socket() 的返回类型，默认为*SSLSocket*。该属性可以在类实例上被重载以便返回自定义的*SSLSocket* 的子类。

Added in version 3.7.

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

包装 BIO 对象 *incoming* 和 *outgoing* 并返回一个*SSLContext.sslobject_class* (默认为*SSLObject*) 的实例。SSL 例程将从 BIO 中读取输入数据并将数据写入到 outgoing BIO。

server_side, *server_hostname* 和 *session* 形参具有与*SSLContext.wrap_socket()* 中相同的含义。

在 3.6 版本发生变更: 增加了 *session* 参数。

在 3.7 版本发生变更: 此方法返回*SSLContext.sslobject_class* 的实例而不是硬编码的*SSLObject*。

SSLContext.**sslobject_class**

SSLContext.wrap_bio() 的返回类型，默认为*SSLObject*。该属性可以在类实例上被重载以便返回自定义的*SSLObject* 的子类。

Added in version 3.7.

SSLContext.**session_stats**()

获取由该上下文创建或管理的 SSL 会话的统计数据。返回将每个 信息块 映射到其数字值的字典。例如，下面是自该上下文创建以来会话缓存中命中和未命中的总计数:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

是否要将匹配*SSLSocket.do_handshake()* 中对等方证书的主机名。该上下文的*verify_mode* 必须被设为*CERT_OPTIONAL* 或*CERT_REQUIRED*，并且你必须将 *server_hostname* 传给*wrap_socket()* 以便匹配主机名。启用主机名检查会自动将*verify_mode* 从*CERT_NONE* 设为*CERT_REQUIRED*。只要启用了主机名检查就无法将其设回*CERT_NONE*。*PROTOCOL_TLS_CLIENT* 协议默认启用主机名检查。对于其他协议，则必须显式地启用主机名检查。

示例:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Added in version 3.4.

在 3.7 版本发生变更: 现在当主机名检查被启用且*verify_mode* 为*CERT_NONE* 时*verify_mode* 会自动更改为*CERT_REQUIRED*。在之前版本中同样的操作将失败并引发*ValueError*。

SSLContext.**keylog_filename**

每当生成或接收到密钥时，将 TLS 密钥写入到一个密钥日志文件。密钥日志文件的设计仅适用于调试目的。文件的格式由 NSS 指明并为许多流量分析工具例如 Wireshark 所使用。日志文件会以追加模式打开。写入操作会在线程之间同步，但不会在进程之间同步。

Added in version 3.8.

SSLContext.**maximum_version**

一个代表所支持的最高 TLS 版本的*TLSVersion* 枚举成员。该值默认为*TLSVersion.MAXIMUM_SUPPORTED*。这个属性对于*PROTOCOL_TLS*，*PROTOCOL_TLS_CLIENT* 和*PROTOCOL_TLS_SERVER* 以外的其他协议来说都是只读的。

maximum_version，*minimum_version* 和*SSLContext.options* 等属性都会影响上下文所支持的 SSL 和 TLS 版本。这个实现不会阻止无效的组合。例如一个*options* 为*OP_NO_TLSv1_2* 而*maximum_version* 设为*TLSVersion.TLSv1_2* 的上下文将无法建立 TLS 1.2 连接。

Added in version 3.7.

SSLContext.**minimum_version**

与*SSLContext.maximum_version* 类似，区别在于它是所支持的最低版本或为*TLSVersion.MINIMUM_SUPPORTED*。

Added in version 3.7.

SSLContext.**num_tickets**

控制一个*PROTOCOL_TLS_SERVER* 上下文的 TLS 1.3 会话凭据数量。这个设置不会影响 TLS 1.0 至 1.2 的连接。

Added in version 3.8.

SSLContext.**options**

一个代表此上下文中所启用的 SSL 选项集的整数。默认值为*OP_ALL*，但你也可以通过在选项间进行 OR 运算来指定其他选项例如*OP_NO_SSLv2*。

在 3.6 版本发生变更: *SSLContext.options* 返回*Options* 旗标:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

自 3.7 版本弃用: 自 OpenSSL 1.1.0 起，所有 OP_NO_SSL* 和 OP_NO_TLS* 选项已被弃用，请改用新的*SSLContext.minimum_version* 和*SSLContext.maximum_version*。

SSLContext.**post_handshake_auth**

启用 TLS 1.3 握手后客户端身份验证。握手后验证默认是被禁用的，服务器只能在初始握手期间请求 TLS 客户端证书。当启用时，服务器可以在握手之后的任何时候请求 TLS 客户端证书。

当在客户端套接字上启用时，客户端会向服务器发信号说明它支持握手后身份验证。

当在服务器端套接字上启用时，*SSLContext.verify_mode* 也必须被设为*CERT_OPTIONAL* 或*CERT_REQUIRED*。实际的客户端证书交换会被延迟直至*SSLSocket.verify_client_post_handshake()* 被调用并执行了一些 I/O 操作后再进行。

Added in version 3.8.

SSLContext.**protocol**

构造上下文时所选择的协议版本。这个属性是只读的。

SSLContext.**hostname_checks_common_name**

在没有目标替代名称扩展的情况下*check_hostname* 是否要回退为验证证书的通用名称（默认为真值）。

Added in version 3.7.

在 3.10 版本发生变更: 此旗标在 OpenSSL 1.1.1l 之前的版本上不起作用。Python 3.8.9, 3.9.3 和 3.10 包括了针对之前版本的变通处理。

SSLContext.**security_level**

一个代表上下文 安全级别 的整数。该属性是只读的。

Added in version 3.10.

SSLContext.**verify_flags**

证书验证操作的标志位。可以用“或”的方式组合在一起设置*VERIFY_CRL_CHECK_LEAF* 这类标志。默认情况下，OpenSSL 既不需要也不验证证书吊销列表（CRL）。

Added in version 3.4.

在 3.6 版本发生变更: *SSLContext.verify_flags* 返回*VerifyFlags* 旗标:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

是否要尝试验证其他对等方的证书以及如果验证失败应采取何种行为。该属性值必须为*CERT_NONE*, *CERT_OPTIONAL* 或*CERT_REQUIRED* 之一。

在 3.6 版本发生变更: *SSLContext.verify_mode* 返回*VerifyMode* 枚举:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

SSLContext.**set_psk_client_callback**(*callback*)

在客户端连接上启用 TLS-PSK（预共享密钥）验证。

一般来说，基于证书的身份验证应当优先于此方法。

形参 `callback` 是一个签名为 `def callback(hint: str | None) -> tuple[str | None, bytes]` 的可调用对象。`hint` 形参是服务器所发送的可选身份标识。返回值是一个 (client-identity, psk) 形式的元组。其中 client-identity 是一个可选的字符串，它可被服务器用来为客户选择相应的 PSK。该字符串在使用 UTF-8 编码时必须小于等于 `256` 个八位字节。PSK 是一个代表预共享密钥的*bytes-like object*。返回一个零长度的 PSK 以拒绝连接。

将 `callback` 设为*None* 将移除任何现有的回调。

备注

当使用 TLS 1.3 时:

- `hint` 形参值将始终为*None*。
- client-identity 必须是一个非空字符串。

用法示例:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# 一个简单的 lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_client_callback(lambda hint: (None, psk))

# 一个使用来自服务器的提示的表:
psk_table = { 'ServerId_1': bytes.fromhex('c0ffee'),
              'ServerId_2': bytes.fromhex('facade')
}
```

(续下页)

(接上页)

```
def callback(hint):
    return 'ClientId_1', psk_table.get(hint, b'')
context.set_psk_client_callback(callback)
```

如果*HAS_PSK* 为 False 则此方法将引发*NotImplementedError*。

Added in version 3.13.

SSLContext.**set_psk_server_callback**(*callback*, *identity_hint=None*)

在服务器端连接上启用 TLS-PSK（预共享密钥验证）。

一般来说，基于证书的身份验证应当优先于此方法。

形参 callback 是一个签名为 def callback(identity: str | None) -> bytes 的可调用对象。identity 形参是客户端所发送的可选身份标识，可用于选择相应的 PSK。返回值是一个代表预共享密钥的*bytes-like object*。返回一个零长度的 PSK 以拒绝连接。

将 callback 设为*None* 将移除任何现有的回调。

形参 identity_hint 是发送给客户端的可选身份标识字符串。该字符串在使用 UTF-8 编码时必须小于等于 256 个八位字节。

备注

当使用 TLS 1.3 时 identity_hint 形参将不会被发送给客户端。

用法示例:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# 一个简单的 lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_server_callback(lambda identity: psk)

# 一个使用客户端标识的表:
psk_table = { 'ClientId_1': bytes.fromhex('c0ffee'),
              'ClientId_2': bytes.fromhex('facade')
}
def callback(identity):
    return psk_table.get(identity, b'')
context.set_psk_server_callback(callback, 'ServerId_1')
```

如果*HAS_PSK* 为 False 则此方法将引发*NotImplementedError*。

Added in version 3.13.

19.3.4 证书

总的来说证书是公钥/私钥系统的一个组成部分。在这个系统中，每个 *主体* (可能是一台机器、一个人或者一个组织) 都会分配到唯一的包含两部分的加密密钥。一部分密钥是公开的，称为 *公钥*；另一部分密钥是保密的，称为 *私钥*。这两个部分是互相关联的，就是说如果你用其中一个部分来加密一条消息，你将能用并且 **只能**用另一个部分来解密它。

在一个证书中包含有两个主体的相关信息。它包含 *目标方* 的名称和目标方的公钥。它还包含由第二个主体 *颁发方* 所发布的声明：目标方的身份与他们所宣称的一致，包含的公钥也确实是目标方的公钥。颁发方的声明使用颁发方的私钥进行签名，该私钥的内容只有颁发方自己才知道。但是，任何人都可以找到颁发方的公钥，用它来解密这个声明，并将其与证书中的其他信息进行比较来验证颁发方声明的真实性。证书还包含有关其有效期限的信息。这被表示为两个字段，即”notBefore” 和”notAfter”。

在 Python 中应用证书时，客户端或服务器可以用证书来证明自己的身份。还可以要求网络连接的另一方提供证书，提供的证书可以用于验证以满足客户端或服务器的验证要求。如果验证失败，连接尝试可被设置为引发一个异常。验证是由下层的 OpenSSL 框架来自动执行的；应用程序本身不必关注其内部的机制。但是应用程序通常需要提供一组证书以允许此过程的发生。

Python 使用文件来包含证书。它们应当采用"PEM" 格式 (参见 **RFC 1422**)，这是一种带有头部行和尾部行的 base-64 编码包装形式:

```
-----BEGIN CERTIFICATE-----
... (使用 base64 PEM 编码的证书) ...
-----END CERTIFICATE-----
```

证书链

包含证书的 Python 文件可以包含一系列的证书，有时被称为 *证书链*。这个证书链应当以" 作为" 客户端或服务器的主体的专属证书打头，然后是证书颁发方的证书，然后是 上述证书的颁发方的证书，证书链就这样不断上溯直到你得到一个 自签名的证书，即具有相同目标方和颁发方的证书，有时也称为 *根证书*。在证书文件中这些证书应当被拼接为一体。例如，假设我们有一个包含三个证书的证书链，以我们的服务器证书打头，然后是为我们的服务器证书签名的证书颁发机构的证书，最后是为证书颁发机构的证书颁发证书的机构的根证书:

```
-----BEGIN CERTIFICATE-----
... (你的服务器的证书)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (CA 的证书)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (CA 的颁发者的根证书)...
-----END CERTIFICATE-----
```

CA 证书

如果你想要求对连接的另一方的证书进行验证，你必须提供一个"CA 证书" 文件，其中包含了你愿意信任的每个颁发方的证书链。同样地，这个文件的内容就是这些证书链拼接在一起的结果。为了进行验证，Python 将使用它在文件中找到的第一个匹配的证书链。可以通过调用*SSLContext.load_default_certs()* 来使用系统平台的证书文件，这可以由*create_default_context()* 自动完成。

合并的密钥和证书

私钥往往与证书存储在相同的文件中；在此情况下，只需要将 `certfile` 形参传给*SSLContext.load_cert_chain()*。如果私钥是与证书一起存储的，则它应当放在证书链的第一个证书之前:

```
-----BEGIN RSA PRIVATE KEY-----
... (使用 base64 编码格式的私钥) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (使用 base64 PEM 编码格式的证书) ...
-----END CERTIFICATE-----
```

自签名证书

如果你准备创建一个提供 SSL 加密连接服务的服务器，你需要为该服务获取一份证书。有许多方式可以获取合适的证书，例如从证书颁发机构购买。另一种常见做法是生成自签名证书。生成自签名证书的最简单方式是使用 OpenSSL 软件包，代码如下所示:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
```

(续下页)

(接上页)

```
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

自签名证书的缺点在于它是它自身的根证书，因此不会存在于别人的已知（且信任的）根证书缓存当中。

19.3.5 例子

检测 SSL 支持

要检测一个 Python 安装版中是否带有 SSL 支持，用户代码应当使用以下例程:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # 执行需要 SSL 支持的操作
```

客户端操作

这个例子创建了一个 SSL 上下文并使用客户端套接字的推荐安全设置，包括自动证书验证:

```
>>> context = ssl.create_default_context()
```

如果你喜欢自行调整安全设置，你可能需要从头创建一个上下文（但是请请注意避免不正确的设置）：

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(这段代码假定你的操作系统将所有 CA 证书打包存放于 `/etc/ssl/certs/ca-bundle.crt`；如果不是这样，你将收到报错信息，必须修改此位置)

PROTOCOL_TLS_CLIENT 协议配置用于证书验证和主机名验证的上下文。*verify_mode* 设为*CERT_REQUIRED* 而*check_hostname* 设为 `True`。所有其他协议都会使用不安全的默认值创建 SSL 上下文。

当你使用此上下文去连接服务器时，*CERT_REQUIRED* 和*check_hostname* 会验证服务器证书；它将确认服务器证书使用了某个 CA 证书进行签名，检查签名是否正确，并验证其他属性例如主机名的有效性和身份真实性:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

你可以随后获取该证书:

```
>>> cert = conn.getpeercert()
```

可视化检查显示证书能够证明目标服务 (即 HTTPS 主机 www.python.org) 的身份:

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

现在 SSL 通道已建立并已验证了证书，你可以继续与服务器对话了:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
```

(续下页)

(接上页)

```
 b'Connection: close',
 b'',
 b'']
```

参见下文对于安全考量 的讨论。

服务器端操作

对于服务器操作，通常你需要在文件中存放服务器证书和私钥各一份。你将首先创建一个包含密钥和证书的上下文，这样客户端就能检查你的身份真实性。然后你将打开一个套接字，将其绑定到一个端口，在其上调用 `listen()`，并开始等待客户端连接:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.example.com', 10023))
bindsocket.listen(5)
```

当有客户端连接时，你将在套接字上调用 `accept()` 以从另一端获取新的套接字，并使用上下文的*SSLContext.wrap_socket()* 方法来为连接创建一个服务器端 SSL 套接字:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

随后你将从 `connstream` 读取数据并对其进行处理，直至你结束与客户端的会话（或客户端结束与你的会话）:

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # 空数据表明客户端已结束与我们的通信
    while data:
        if not do_something(connstream, data):
            # 我们将假定当我们结束与客户端的通信时
            # do_something 将返回 False
            break
        data = connstream.recv(1024)
    # 结束与客户端的通信
```

并返回至监听新的客户端连接（当然，真正的服务器应当会在单独的线程中处理每个客户端连接，或者将套接字设为非阻塞模式 并使用事件循环）。

19.3.6 关于非阻塞套接字的说明

在非阻塞模式下 SSL 套接字的行为与常规套接字略有不同。当使用非阻塞模式时，你需要注意下面这些事情:

- 如果一个 I/O 操作会阻塞，大多数*SSLSocket* 方法都将引发*SSLWantWriteError* 或*SSLWantReadError* 而非*BlockingIOError*。如果有必要在下层套接字上执行读取操作将引发*SSLWantReadError*，在下层套接字上执行写入操作则将引发*SSLWantWriteError*。请注意尝试写入到 SSL 套接字可能需要先从下层套接字 读取，而尝试从 SSL 套接字 读取则可能需要先向下层套接字 写入。

在 3.5 版本发生变更: 在较早的 Python 版本中，`SSLSocket.send()` 方法会返回零值而非引发*SSLWantWriteError* 或*SSLWantReadError*。

- 调用*select()* 将告诉你可以从 OS 层级的套接字读取（或向其写入），但这并不意味着在上面的 SSL 层有足够的数据。例如，可能只有部分 SSL 帧已经到达。因此，你必须准备好处理 `SSLSocket.recv()` 和 `SSLSocket.send()` 失败的情况，并在再次调用*select()* 之后重新尝试。
- 相反地，由于 SSL 层具有自己的帧机制，一个 SSL 套接字可能仍有可读取的数据而*select()* 并不知道这一点。因此，你应当先调用 `SSLSocket.recv()` 取走所有潜在的可用数据，然后只在必要时对*select()* 调用执行阻塞。

  (当然，类似的保留规则在使用其他原语例如*poll()*，或*selectors* 模块中的原语时也适用)
- SSL 握手本身将是非阻塞的: *SSLSocket.do_handshake()* 方法必须不断重试直至其成功返回。下面是一个使用*select()* 来等待套接字就绪的简短例子:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

参见

asyncio 模块支持*非阻塞 SSL 套接字* 并提供了更高层级的*流 API*。它会使用*selectors* 模块来轮询事件并处理*SSLWantWriteError*, *SSLWantReadError* 和*BlockingIOError* 等异常。它还会异步地执行 SSL 握手。handshake asynchronously as well.

19.3.7 内存 BIO 支持

Added in version 3.5.

自从 SSL 模块在 Python 2.6 起被引入之后，*SSLSocket* 类提供了两个互相关联但彼此独立的功能分块:

- SSL 协议处理
- 网络 IO

网络 IO API 与*socket.socket* 所提供的功能一致，*SSLSocket* 也是从那里继承而来的。这允许 SSL 套接字被用作常规套接字的替代，使得向现有应用程序添加 SSL 支持变得非常容易。

将 SSL 协议处理与网络 IO 结合使用通常都能运行良好，但在某些情况下则不能。此情况的一个例子是 async IO 框架，该框架要使用不同的 IO 多路复用模型而非 (基于就绪状态的) " 在文件描述器上执行选择/轮询" 模型，该模型是*socket.socket* 和内部 OpenSSL 套接字 IO 例程正常运行的假设前提。这种情况在该模型效率不高的 Windows 平台上最为常见。为此还提供了一个*SSLSocket* 的简化形式，称为*SSLObject*。

class ssl.**SSLObject**

SSLSocket 的简化形式，表示一个不包含任何网络 IO 方法的 SSL 协议实例。这个类通常由想要通过内存缓冲区为 SSL 实现异步 IO 的框架作者来使用。

这个类在低层级 SSL 对象上实现了一个接口，与 OpenSSL 所实现的类似。此对象会捕获 SSL 连接的状态但其本身不提供任何网络 IO。IO 需要通过单独的"BIO" 对象来执行，该对象是 OpenSSL 的 IO 抽象层。

这个类没有公有构造器。*SSLObject* 实例必须使用*wrap_bio()* 方法来创建。此方法将创建*SSLObject* 实例并将其绑定到一个 BIO 对。其中 *incoming* BIO 用来将数据从 Python 传递到 SSL 协议实例，而 *outgoing* BIO 用来进行数据反向传递。

可以使用以下方法:

- *context*
- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *get_verified_chain()*
- *get_unverified_chain()*
- *selected_alpn_protocol()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *verify_client_post_handshake()*
- *unwrap()*
- *get_channel_binding()*
- *version()*

与*SSLSocket*相比，此对象缺少下列特性:

- 任何形式的网络 IO; `recv()` 和 `send()` 仅对下层的*MemoryBIO*缓冲区执行读取和写入。
- 不存在 *do_handshake_on_connect* 机制。你必须总是手动调用*do_handshake()*来开始握手操作。
- 不存在对 *suppress_ragged_eofs* 的处理。所有违反协议的文件结束条件将通过*SSLEOFError*异常来报告。
- 方法*unwrap()*的调用不返回任何东西，不会如 SSL 套接字那样返回下层的套接字。
- *server_name_callback* 回调被传给*SSLContext.set_servername_callback()*时将获得一个*SSLObject*实例而非*SSLSocket*实例作为其第一个形参。

有关*SSLObject*用法的一些说明:

- 在*SSLObject*上的所有 IO 都是非阻塞的。这意味着例如*read()*在其需要比 incoming BIO 可用的更多数据时将会引发*SSLWantReadError*。

在 3.7 版本发生变更: *SSLObject*的实例必须使用*wrap_bio()*来创建。在较早的版本中，直接创建该实例是可能的。但这从未被写入文档或是被正式支持。

SSLObject 会使用内存缓冲区与外部世界通信。*MemoryBIO*类提供了可被用于此目的的内存缓冲区。它包装了一个 OpenSSL 内存 BIO (Basic IO) 对象:

class ssl.**MemoryBIO**

一个可被用来在 Python 和 SSL 协议实例之间传递数据的内存缓冲区。

pending

返回当前存在于内存缓冲区的字节数。

eof

一个表明内存 BIO 目前是否位于文件末尾的布尔值。

read(*n=-1*)

从内存缓冲区读取至多 *n* 个字节。如果 *n* 未指定或为负值，则返回全部字节数据。

write(*buf*)

将字节数据从 *buf* 写入到内存 BIO。*buf* 参数必须为支持缓冲区协议的对象。

返回值为写入的字节数，它总是与 *buf* 的长度相等。

write_eof()

将一个 EOF 标记写入到内存 BIO。在此方法被调用以后，再调用*write()* 将是非法的。属性*eof* will 在缓冲区当前的所有数据都被读取之后将变为真值。

19.3.8 SSL 会话

Added in version 3.6.

class ssl.**SSLSession**

session 所使用的会话对象。

id

time

timeout

ticket_lifetime_hint

has_ticket

19.3.9 安全考量

最佳默认值

针对 **客户端使用**，如果你对于安全策略没有任何特殊要求，则强烈推荐你使用*create_default_context()* 函数来创建你的 SSL 上下文。它将加载系统的受信任 CA 证书，启用证书验证和主机名检查，并尝试合理地选择安全的协议和密码设置。

例如，以下演示了你应当如何使用*smtplib.SMTP* 类来创建指向一个 SMTP 服务器的受信任且安全的连接:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

如果连接需要客户端证书，可使用*SSLContext.load_cert_chain()* 来添加。

作为对比，如果你通过自行调用*SSLContext* 构造器来创建 SSL 上下文，它默认将不会启用证书验证和主机名检查。如果你这样做，请阅读下面的段落以达到良好的安全级别。

手动设置

验证证书

当直接调用*SSLContext* 构造器时，默认值为*CERT_NONE*。由于它不会验证对端的身份，因而是不安全的，特别是在大部分时间里你都希望能确保与你通信的服务器的可靠性的客户端模式下。为此，当在客户端模式下，强烈建议使用*CERT_REQUIRED*。但是，仅靠它本身是不够的；你还必须检查服务器证书，它可通过调用*SSLSocket.getpeercert()* 来获取，确定它与目标服务相匹配。对于许多协议和应用程序来说，服务可通过主机名来进行标识。这种通用检查会在*SSLContext.check_hostname* 已启用时自动执行。

在 3.7 版本发生变更: 主机名匹配现在是由 OpenSSL 来执行的。Python 不会再使用 `match_hostname()`。

在服务器模式下，如果你想要使用 SSL 层来验证客户端（而不是使用更高层级的验证机制），你也必须要指定*CERT_REQUIRED* 并以类似方式检查客户端证书。

协议版本

SSL 版本 2 和 3 被认为是不安全的因而使用它们会有风险。如果你想要客户端和服务器之间有最大的兼容性，推荐使用*PROTOCOL_TLS_CLIENT* 或*PROTOCOL_TLS_SERVER* 作为协议版本。SSLv2 和 SSLv3 默认会被禁用。

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

上面创建的 SSL 上下文将只允许与服务器进行 TLSv1.3 及更高版本（如果你的系统支持）的连接。在默认情况下*PROTOCOL_TLS_CLIENT* 将使用证书验证和主机名检查。你必须将证书加载到上下文中。

密码选择

如果你有更高级的安全要求，也可通过*SSLContext.set_ciphers()* 方法在协商 SSL 会话时对启用的加密进行微调。从 Python 3.2.3 开始，ssl 模块默认禁用了某些较弱的加密，但你还可能希望进一步限制加密选项。请确保仔细阅读有关 加密列表格式 的 OpenSSL 文档。如果你想要检查给定的加密列表启用了哪些加密，可以使用*SSLContext.get_ciphers()* 或你所用系统的 `openssl ciphers` 命令。

多进程

如果使用此模块作为多进程应用的一部分（例如，使用*multiprocessing* 或*concurrent.futures* 模块），请注意 OpenSSL 的内部随机数字生成器并不能正确处理分叉的进程。应用程序必须修改父进程的 PRNG 状态，如果它们要使用任何包含*os.fork()* 的 SSL 特征的话。任何对*RAND_add()* 或*RAND_bytes()* 的成功调用都可以做到这一点。

19.3.10 TLS 1.3

Added in version 3.7.

TLS 1.3 协议的行为与低版本的 TLS/SSL 略有不同。某些 TLS 1.3 新特性还不可用。

- TLS 1.3 使用一组不同的加密套件集。默认情况下所有 AES-GCM 和 ChaCha20 加密套件都会被启用。*SSLContext.set_ciphers()* 方法还不能启用或禁用任何 TLS 1.3 加密，但*SSLContext.get_ciphers()* 会返回它们。
- 会话凭据不再会作为初始握手的组成部分被发送而是以不同的方式来处理。*SSLSocket.session* 和*SSLSession* 与 TLS 1.3 不兼容。
- 客户端证书在初始握手期间也不会再被验证。服务器可以在任何时候请求证书。客户端会在它们从服务器发送或接收应用数据时处理证书请求。
- 早期数据、延迟的 TLS 客户端证书请求、签名算法配置和密钥重生成等 TLS 1.3 特性尚未被支持。

参见

Class *socket.socket*
下层*socket* 类的文档

SSL/TLS 高强度加密：概述
Apache HTTP Server 文档介绍

RFC 1422: 因特网电子邮件的隐私加强：第二部分：基于证书的密钥管理
Steve Kent

RFC 4086: 确保安全的随机性要求
Donald E., Jeffrey I. Schiller

RFC 5280: 互联网 X.509 公钥基础架构证书和证书吊销列表 (CRL) 配置文件
D. Cooper

RFC 5246: 传输层安全性 (TLS) 协议版本 1.2
T. Dierks et. al.

RFC 6066: 传输层安全性 (TLS) 的扩展
D. Eastlake

IANA TLS: 传输层安全性 (TLS) 的参数
IANA

RFC 7525: 传输层安全性 (TLS) 和数据报传输层安全性 (DTLS) 的安全使用建议
IETF

Mozilla 的服务器端 TLS 建议
Mozilla

19.4 `select` --- 等待 I/O 完成

该模块提供了对 `select()` 和 `poll()` 函数的访问，这在大多数操作系统上都是可用的，`devpoll()` 在 Solaris 及其衍生系统上可用，`epoll()` 在 Linux 2.5+ 上可用，而 `kqueue()` 在大多数 BSD 上可用。注意在 Windows 上，它仅适用于套接字；在其他操作系统上，它还适用于其他文件类型（特别是在 Unix 上，它还适用于管道）。它不能被用在常规文件上确定一个文件自其最后一次被读取后大小是否有增长。

备注

selectors 模块是在 *select* 模块原型的基础上进行高级且高效的 I/O 复用。推荐用户改用 *selectors* 模块，除非用户希望对 OS 级的函数原型进行精确控制。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅 *WebAssembly 平台* 了解详情。

该模块定义以下内容：

exception `select.error`

一个被弃用的 *OSError* 的别名。

在 3.3 版本发生变更: 根据 **PEP 3151**，这个类是 *OSError* 的别名。

`select.devpoll()`

（仅支持 Solaris 及其衍生版本）返回一个 `/dev/poll` 轮询对象，请参阅下方 */dev/poll 轮询对象* 获取 devpoll 对象所支持的方法。

`devpoll()` 对象与实例化时允许的文件描述符数量相关联。如果你的程序减少该值，`devpoll()` 将会失败。如果你的程序增加该值，`devpoll()` 可能会返回不完整的活动文件描述符列表。

新的文件描述符是 *不可继承的*。

Added in version 3.3.

在 3.4 版本发生变更: 新的文件描述符现在是不可继承的。

`select.epoll`(*sizehint=-1*, *flags=0*)

（仅支持 Linux 2.5.44 或更高版本）返回一个 edge poll 对象，该对象可作为 I/O 事件的边缘触发或水平触发接口。

sizehint 通知 epoll 预计要注册的事件数量。该值必须为正数，或为 `-1` 以使用默认值。它仅在 `epoll_create1()` 不可用的旧系统上会被使用，在其他情况下它没有任何作用（尽管仍会检查其值）。

flags 已经弃用且完全被忽略。但是，如果提供该值，则它必须是 `0` 或 `select.EPOLL_CLOEXEC`，否则会抛出 `OSError` 异常。

请参阅下方*边缘触发和水平触发的轮询 (epoll) 对象* 获取 epoll 对象所支持的方法。

`epoll` 对象支持上下文管理器：当在 `with` 语句中使用时，新建的文件描述符会在运行至语句块结束时自动关闭。

新的文件描述符是*不可继承的*。

在 3.3 版本发生变更: 增加了 *flags* 参数。

在 3.4 版本发生变更: 增加了对 `with` 语句的支持。新的文件描述符现在是不可继承的。

自 3.4 版本弃用: *flags* 参数。现在默认采用 `select.EPOLL_CLOEXEC` 标志。使用*os.set_inheritable()* 来让文件描述符可继承。

`select.`**`poll`**`()`

（部分操作系统不支持）返回一个 poll 对象，该对象支持注册和注销文件描述符，支持对描述符进行轮询以获取 I/O 事件。请参阅下方*Poll 对象* 获取 poll 对象所支持的方法。

`select.`**`kqueue`**`()`

（仅支持 BSD）返回一个内核队列对象，请参阅下方*Kqueue 对象* 获取 kqueue 对象所支持的方法。

新的文件描述符是*不可继承的*。

在 3.4 版本发生变更: 新的文件描述符现在是不可继承的。

`select.`**`kevent`**`(`*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*`)`

（仅支持 BSD）返回一个内核事件对象，请参阅下方*Kevent 对象* 获取 kevent 对象所支持的方法。

`select.`**`select`**`(`*rlist*, *wlist*, *xlist*[, *timeout*]`)`

这是一个明白直观的 Unix `select()` 系统调用接口。前三个参数是产生“可等待对象”的可迭代对象：可以是代表文件描述符的整数，或是带有名为*fileno()* 的返回这样的整数的无形参方法的对象:

- *rlist*：等待，直到可以开始读取
- *wlist*：等待，直到可以开始写入
- *xlist*：等待“异常情况”（请参阅当前系统的手册，以获取哪些情况称为异常情况）

允许空的可迭代对象，但是否接受三个空的可迭代对象则取决于具体平台。（已知在 Unix 上可行但在 Windows 上不可行。）可选的 *timeout* 参数以一个浮点数表示超时秒数。当省略 *timeout* 参数时该函数将阻塞直到至少有一个文件描述符准备就绪。超时值为零表示执行轮询且永不阻塞。

返回值是三个列表，包含已就绪对象，返回的三个列表是前三个参数的子集。当超时时间已到且没有文件描述符就绪时，返回三个空列表。

可迭代对象中可接受的对象类型有 Python *文件对象* (例如 `sys.stdin` 以及*open()* 或*os.popen()* 所返回的对象)，由*socket.socket()* 返回的套接字对象等。你也可以自定义一个 *wrapper* 类，只要它具有适当的*fileno()* 方法（该方法要确实返回一个文件描述符，而不能只是一个随机整数）。

> **备注**
>
> 在 Windows 上不接受文件对象，但可以接受套接字。在 Windows 上，底层的 `select()` 函数由 WinSock 库提供，且不会处理不是源自 WinSock 的文件描述符。

在 3.5 版本发生变更: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出*InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

select.**PIPE_BUF**

当*select()*、poll() 或本模块中的其他接口报告管道已准备就绪可以写入时，可以在不阻塞该管道的情况下的最小字节数。它这不适用于其他文件型对象，例如如套接字。

POSIX 上须保证该值不小于 512。

Availability: Unix

Added in version 3.2.

19.4.1 /dev/poll 轮询对象

Solaris 及其衍生版本具有 /dev/poll。而 select() 为 *O*(最高文件描述符) 并且 poll() 为 *O*(文件描述符数量), /dev/poll 为 *O*(活动的文件描述符)。

/dev/poll 的行为非常接近标准 poll() 对象。

devpoll.**close**()

关闭轮询对象的文件描述符。

Added in version 3.4.

devpoll.**closed**

如果轮询对象已关闭，则返回 True。

Added in version 3.4.

devpoll.**fileno**()

返回轮询对象的文件描述符对应的数字。

Added in version 3.4.

devpoll.**register**(*fd*[, *eventmask*])

在轮询对象中注册文件描述符。这样，将来调用*poll()* 方法时将检查文件描述符是否有未处理的 I/O 事件。*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。文件对象已经实现了 fileno()，因此它们也可以用作参数。

eventmask 是可选的位掩码，用于描述要检查的事件类型。这些常量与 poll() 对象的相同。默认值是常量 POLLIN、POLLPRI 和 POLLOUT 的组合。

> **警告**
>
> 注册已注册的文件描述符不会报错，但结果是未定义的。适当的做法是先注销或修改它。这是与:c:func:*!poll* 的一个重要区别。

devpoll.**modify**(*fd*[, *eventmask*])

此方法先执行*unregister()* 后执行*register()*。直接执行此操作效率（稍微）高一些。

devpoll.**unregister**(*fd*)

删除轮询对象正在跟踪的某个文件描述符。与*register()* 方法类似，*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。

尝试删除从未注册过的文件描述符将被安全地忽略。

devpoll.**poll**([*timeout*])

轮询已注册的文件描述符的集合，并返回一个列表，列表可能为空，也可能有多个 (fd, event) 2 元组，其中包含了要报告事件或错误的描述符。*fd* 是文件描述符，*event* 是一个位掩码，表示该描述符所报告的事件 --- POLLIN 表示等待输入，POLLOUT 表示该描述符可以写入，依此类推。空列表表示调用超时，没有任何文件描述符报告事件。如果指定了 *timeout*，它将指定系统等待事件时，等待多长时间后返回（以毫秒为单位）。如果 *timeout* 被省略、为 -1 或为*None*，则本调用将阻塞，直到轮询对象发生事件为止。

在 3.5 版本发生变更: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出*InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

19.4.2 边缘触发和水平触发的轮询 (epoll) 对象

https://linux.die.net/man/4/epoll

eventmask

常量	含意
EPOLLIN	可读
EPOLLOUT	可写
EPOLLPRI	紧急数据读取
EPOLLERR	在关联的文件描述符上有错误情况发生
EPOLLHUP	关联的文件描述符已挂起
EPOLLET	设置触发方式为边缘触发，默认为水平触发
EPOLLONESH	设置 one-shot 模式。触发一次事件后，该描述符会在轮询对象内部被禁用。
EPOLLEXCLU	当已关联的描述符发生事件时，仅唤醒一个 epoll 对象。默认（如果未设置此标志）是唤醒所有轮询该描述符的 epoll 对象。
EPOLLRDHUP	流套接字的对侧关闭了连接或关闭了写入到一半的连接。
EPOLLRDNOR	等同于 EPOLLIN
EPOLLRDBAN	可以读取优先数据带。
EPOLLWRNOR	等同于 EPOLLOUT
EPOLLWRBAN	可以写入优先级数据。
EPOLLMSG	忽略

Added in version 3.6: 增加了 EPOLLEXCLUSIVE。仅支持 Linux Kernel 4.5 或更高版本。

epoll.**close**()

关闭用于控制 epoll 对象的文件描述符。

epoll.**closed**

如果 epoll 对象已关闭，则返回 True。

epoll.**fileno**()

返回文件描述符对应的数字，该描述符用于控制 epoll 对象。

epoll.**fromfd**(*fd*)

根据给定的文件描述符创建 epoll 对象。

epoll.**register**(*fd*[, *eventmask*])

在 epoll 对象中注册一个文件描述符。

epoll.**modify**(*fd*, *eventmask*)

修改一个已注册的文件描述符。

epoll.**unregister**(*fd*)

从 epoll 对象中删除一个已注册的文件描述符。

在 3.9 版本发生变更: 此方法不会再忽略*EBADF* 错误。

epoll.**poll**(*timeout=None*, *maxevents=-1*)

等待事件发生，timeout 是浮点数，单位为秒。

在 3.5 版本发生变更: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出*InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

19.4.3 Poll 对象

大多数 Unix 系统都支持 poll() 系统调用，它为网络服务器提供了更好的可伸缩性，可以同时为大量客户端提供服务。poll() 的可伸缩性更好是因为该系统只须列出要关注的文件描述符，而 select() 则会构建一个位映射表，打开这个要关注的描述符所对应的比特位，然后再次线性扫描整个位映射表。select() 的复杂度为 *O*(最高文件描述符)，而 poll() 则为 *O*(文件描述符的数量)。

poll.**register**(*fd*[, *eventmask*])

在轮询对象中注册文件描述符。这样，将来调用*poll()* 方法时将检查文件描述符是否有未处理的 I/O 事件。*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。文件对象已经实现了 fileno()，因此它们也可以用作参数。

eventmask 是可选的位掩码，用于指定要检查的事件类型，它可以是常量 POLLIN、POLLPRI 和 POLLOUT 的组合，如下表所述。如果未指定本参数，默认将会检查所有 3 种类型的事件。

常量	含意
POLLIN	有要读取的数据
POLLPRI	有紧急数据需要读取
POLLOUT	准备输出：写不会阻塞
POLLERR	某种错误条件
POLLHUP	挂起
POLLRDHUP	流套接字的对侧关闭了连接，或关闭了写入到一半的连接
POLLNVAL	无效的请求：描述符未打开

注册已注册过的文件描述符不会报错，且等同于只注册一次该描述符。

poll.**modify**(*fd*, *eventmask*)

修改一个已注册的文件描述符，等同于 register(fd, eventmask)。尝试修改未注册的文件描述符会抛出*OSError* 异常，错误码为 ENOENT。

poll.**unregister**(*fd*)

删除轮询对象正在跟踪的某个文件描述符。与*register()* 方法类似，*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。

尝试删除从未注册过的文件描述符会抛出*KeyError* 异常。

poll.**poll**([*timeout*])

轮询已注册的文件描述符的集合，并返回一个列表，列表可能为空，也可能有多个 (fd, event) 2 元组，其中包含了要报告事件或错误的描述符。*fd* 是文件描述符，*event* 是一个位掩码，表示该描述符所报告的事件 --- POLLIN 表示等待输入，POLLOUT 表示该描述符可以写入，依此类推。空列表表示调用超时，没有任何文件描述符报告事件。如果指定了 *timeout*，它将指定系统等待事件时，等待多长时间后返回（以毫秒为单位）。如果 *timeout* 被省略、为 -1 或为*None*，则本调用将阻塞，直到轮询对象发生事件为止。

在 3.5 版本发生变更: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出*InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

19.4.4 Kqueue 对象

kqueue.**close**()

关闭用于控制 kqueue 对象的文件描述符。

kqueue.**closed**

如果 kqueue 对象已关闭，则返回 True。

kqueue.**fileno**()

返回文件描述符对应的数字，该描述符用于控制 epoll 对象。

kqueue.**fromfd**(*fd*)

根据给定的文件描述符创建 kqueue 对象。

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist

Kevent 的低级接口

- changelist 必须是一个可迭代对象，迭代出 kevent 对象，否则置为 None。
- max_events 必须是 0 或一个正整数。

- timeout 单位为秒（一般为浮点数），默认为 None，即永不超时。

在 3.5 版本发生变更: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

19.4.5 Kevent 对象

https://man.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**

用于区分事件的标识值。其解释取决于筛选器，但该值通常是文件描述符。在构造函数中，该标识值可以是整数或带有 *fileno()* 方法的对象。kevent 在内部存储整数。

kevent.**filter**

内核筛选器的名称。

常量	含意
KQ_FILTER_READ	获取描述符，并在有数据可读时返回
KQ_FILTER_WRITE	获取描述符，并在有数据可写时返回
KQ_FILTER_AIO	AIO 请求
KQ_FILTER_VNODE	当在 *fflag* 中监视的一个或多个请求事件发生时返回
KQ_FILTER_PROC	监视进程 ID 上的事件
KQ_FILTER_NETDEV	观察网络设备上的事件 [在 macOS 上不可用]
KQ_FILTER_SIGNAL	每当监视的信号传递到进程时返回
KQ_FILTER_TIMER	建立一个任意的计时器

kevent.**flags**

筛选器操作。

常量	含意
KQ_EV_ADD	添加或修改事件
KQ_EV_DELETE	从队列中删除事件
KQ_EV_ENABLE	Permitscontrol() 返回事件
KQ_EV_DISABLE	禁用事件
KQ_EV_ONESHOT	在第一次发生后删除事件
KQ_EV_CLEAR	检索事件后重置状态
KQ_EV_SYSFLAGS	内部事件
KQ_EV_FLAG1	内部事件
KQ_EV_EOF	筛选特定 EOF 条件
KQ_EV_ERROR	请参阅返回值

kevent.**fflags**

筛选特定标志。

KQ_FILTER_READ 和 KQ_FILTER_WRITE 筛选标志：

常量	含意
KQ_NOTE_LOWAT	套接字缓冲区的低水线

KQ_FILTER_VNODE 筛选标志：

常量	含意
KQ_NOTE_DELETE	已调用 *unlink()*
KQ_NOTE_WRITE	发生写入
KQ_NOTE_EXTEND	文件已扩展
KQ_NOTE_ATTRIB	属性已更改
KQ_NOTE_LINK	链接计数已更改
KQ_NOTE_RENAME	文件已重命名
KQ_NOTE_REVOKE	对文件的访问权限已被撤销

KQ_FILTER_PROC filter flags:

常量	含意
KQ_NOTE_EXIT	进程已退出
KQ_NOTE_FORK	该进程调用了 *fork()*
KQ_NOTE_EXEC	进程已执行新进程
KQ_NOTE_PCTRLMASK	内部筛选器标志
KQ_NOTE_PDATAMASK	内部筛选器标志
KQ_NOTE_TRACK	跨 *fork()* 执行进程
KQ_NOTE_CHILD	在 *NOTE_TRACK* 的子进程上返回
KQ_NOTE_TRACKERR	无法附加到子对象

KQ_FILTER_NETDEV 过滤器旗标 (在 macOS 上不可用):

常量	含意
KQ_NOTE_LINKUP	链接已建立
KQ_NOTE_LINKDOWN	链接已断开
KQ_NOTE_LINKINV	链接状态无效

kevent.**data**
筛选特定数据。

kevent.**udata**
用户自定义值。

19.5 selectors --- 高层级 I/O 复用

Added in version 3.4.

源码: Lib/selectors.py

19.5.1 概述

此模块允许高层级且高效率的 I/O 复用，它建立在*select* 模块原型的基础之上。推荐用户改用此模块，除非他们希望对所使用的 OS 层级原型进行精确控制。

它定义了一个*BaseSelector* 抽象基类，以及多个具体实现 (*KqueueSelector*, *EpollSelector*...)，它们可被用于在多个文件对象上等待 I/O 就绪通知。在下文中，“文件对象”是指任何具有*fileno()* 方法的对象，或是一个原始文件描述符。参见*file object*。

DefaultSelector 是一个指向当前平台上可用的最高效实现的别名：这应为大多数用户的默认选择。

备注

受支持的文件对象类型取决于具体平台：在 Windows 上，支持套接字但不支持管道，而在 Unix 上两者均受支持（某些其他类型也可能受支持，例如 fifo 或特殊文件设备等）。

参见

select
低层级的 I/O 多路复用模块。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅 *WebAssembly 平台* 了解详情。

19.5.2 类

类的层次结构:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

下文中，*events* 一个位掩码，指明哪些 I/O 事件要在给定的文件对象上执行等待。它可以是以下模块级常量的组合:

常量	含意
selectors.**EVENT_READ**	可读
selectors.**EVENT_WRITE**	可写

class selectors.**SelectorKey**

SelectorKey 是一个*namedtuple*，用来将文件对象关联到其下层的文件描述符、选定事件掩码和附加数据等。它会被某些*BaseSelector* 方法返回。

fileobj
已注册的文件对象。

fd
下层的文件描述符。

events
必须在此文件对象上被等待的事件。

data
可选的关联到此文件对象的不透明数据：例如，这可被用来存储各个客户端的会话 ID。

class selectors.**BaseSelector**

一个*BaseSelector*，用来在多个文件对象上等待 I/O 事件就绪。它支持文件流注册、注销，以及在这些流上等待 I/O 事件的方法。它是一个抽象基类，因此不能被实例化。请改用*DefaultSelector*，或者*SelectSelector*, *KqueueSelector* 等。如果你想要指明使用某个实现，并且你的平台支持它的话。*BaseSelector* 及其具体实现支持*context manager* 协议。

abstractmethod register(*fileobj*, *events*, *data=None*)

注册一个用于选择的文件对象，在其上监视 I/O 事件。

fileobj 是要监视的文件对象。它可以是整数形式的文件描述符或者具有 `fileno()` 方法的对象。*events* 是要监视的事件的位掩码。*data* 是一个不透明对象。

这将返回一个新的*SelectorKey* 实例，或在出现无效事件掩码或文件描述符时引发*ValueError*，或在文件对象已被注册时引发*KeyError*。

abstractmethod unregister(*fileobj*)

注销对一个文件对象的选择，移除对它的监视。在文件对象被关闭之前应当先将其注销。

fileobj 必须是之前已注册的文件对象。

这将返回已关联的*SelectorKey* 实例，或者如果 *fileobj* 未注册则会引发*KeyError*。It will raise *ValueError* 如果 *fileobj* 无效（例如它没有 `fileno()` 方法或其 `fileno()` 方法返回无效值）。

modify(*fileobj*, *events*, *data=None*)

更改已注册文件对象所监视的事件或所附带的数据。

这等价于 `BaseSelector.unregister(fileobj)` 加 `BaseSelector.register(fileobj, events, data)`，区别在于它可以被更高效地实现。

这将返回一个新的*SelectorKey* 实例，或在出现无效事件掩码或文件描述符时引发*ValueError*，或在文件对象未被注册时引发*KeyError*。

abstractmethod select(*timeout=None*)

等待直到有已注册的文件对象就绪，或是超过时限。

如果 `timeout > 0`，这指定以秒数表示的最大等待时间。如果 `timeout <= 0`，调用将不会阻塞，并将报告当前就绪的文件对象。如果 *timeout* 为 `None`，调用将阻塞直到某个被监视的文件对象就绪。

这将返回由 `(key, events)` 元组构成的列表，每项各表示一个就绪的文件对象。

key 是对应于就绪文件对象的*SelectorKey* 实例。*events* 是在此文件对象上等待的事件位掩码。

> **备注**
>
> 如果当前进程收到一个信号，此方法可在任何文件对象就绪之前或超出时限时返回：在此情况下，将返回一个空列表。

在 3.5 版本发生变更: 现在当被某个信号中断时，如果信号处理程序没有引发异常，选择器会用重新计算的超时值进行重试（请查看 **PEP 475** 其理由），而不是在超时之前返回空的事件列表。

close()

关闭选择器。

必须调用这个方法以确保下层资源会被释放。选择器被关闭后将不可再使用。

get_key(*fileobj*)

返回关联到某个已注册文件对象的键。

此方法将返回关联到文件对象的*SelectorKey* 实例，或在文件对象未注册时引发*KeyError*。

abstractmethod get_map()

返回从文件对象到选择器键的映射。

这将返回一个将已注册文件对象映射到与其相关联的*SelectorKey* 实例的*Mapping* 实例。

class selectors.**DefaultSelector**

默认的选择器类，使用当前平台上可用的最高效实现。这应为大多数用户的默认选择。

class selectors.**SelectSelector**

基于*select.select()* 的选择器。

class selectors.**PollSelector**

基于*select.poll()* 的选择器。

class selectors.**EpollSelector**

基于*select.epoll()* 的选择器。

fileno()

此方法将返回由下层*select.epoll()* 对象所使用的文件描述符。

class selectors.**DevpollSelector**

基于*select.devpoll()* 的选择器。

fileno()

此方法将返回由下层*select.devpoll()* 对象所使用的文件描述符。

Added in version 3.5.

class selectors.**KqueueSelector**

基于*select.kqueue()* 的选择器。

fileno()

此方法将返回由下层*select.kqueue()* 对象所使用的文件描述符。

19.5.3 例子

下面是一个简单的回显服务器实现:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # 应当已就绪
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # 应当已就绪
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # 希望不会阻塞
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

19.6 signal --- 设置异步事件处理器

源代码: Lib/signal.py

该模块提供了在 Python 中使用信号处理程序的机制。

19.6.1 一般规则

signal.signal() 函数允许定义在接收到信号时执行的自定义处理程序。少量的默认处理程序已经设置：*SIGPIPE* 被忽略（因此管道和套接字上的写入错误可以报告为普通的 Python 异常）以及如果父进程没有更改*SIGINT*，则其会被翻译成*KeyboardInterrupt* 异常。

一旦设置，特定信号的处理程序将保持安装，直到它被显式重置（Python 模拟 BSD 样式接口而不管底层实现），但*SIGCHLD* 的处理程序除外，它遵循底层实现。

在 WebAssembly 平台上，信号是模拟实现的因而其行为有所不同。某些函数和信号在这些平台上将不可用。

执行 Python 信号处理程序

Python 信号处理程序不会在低级（C ）信号处理程序中执行。相反，低级信号处理程序设置一个标志，告诉*virtual machine* 稍后执行相应的 Python 信号处理程序（例如在下一个*bytecode* 指令）。这会导致：

- 捕获同步错误是没有意义的，例如*SIGFPE* 或*SIGSEGV*，它们是由 C 代码中的无效操作引起的。Python 将从信号处理程序返回到 C 代码，这可能会再次引发相同的信号，导致 Python 显然的挂起。从 Python 3.3 开始，你可以使用*faulthandler* 模块来报告同步错误。
- 纯 C 中实现的长时间运行的计算（例如在大量文本上的正则表达式匹配）可以在任意时间内不间断地运行，而不管接收到任何信号。计算完成后将调用 Python 信号处理程序。
- 如果处理器引发了异常，它将在主线程中“凭空”被引发。请参阅*下面的注释* 讨论相关细节。

信号与线程

Python 信号处理程序总是会在主 Python 主解释器的主线程中执行，即使信号是在另一个线程中接收的。这意味着信号不能被用作线程间通信的手段。你可以改用*threading* 模块中的同步原语。

此外，只有主解释器的主线程才被允许设置新的信号处理程序。

19.6.2 模块内容

在 3.5 版本发生变更: 下面列出的信号 (SIG*), 处理器 (*SIG_DFL*, *SIG_IGN*) 和信号掩码 (*SIG_BLOCK*, *SIG_UNBLOCK*, *SIG_SETMASK*) 相关的常量会被转成*enums* (分别为*Signals*, *Handlers* 和*Sigmasks*)。*getsignal()*, *pthread_sigmask()*, *sigpending()* 和*sigwait()* 函数将以*Signals* 对象形式返回人类可读的*enums*。

signal 模块定义了三个枚举:

class signal.**Signals**

enum.IntEnum 是 SIG* 常量和 CTRL_* 常量的多项集。

Added in version 3.5.

class signal.**Handlers**

enum.IntEnum 是常量*SIG_DFL* 和*SIG_IGN* 的多项集。

Added in version 3.5.

class signal.**Sigmasks**

enum.IntEnum 是常量*SIG_BLOCK*, *SIG_UNBLOCK* 和*SIG_SETMASK* 的多项集。

Availability: Unix.

请参阅手册页面 *sigprocmask(2)* 和 *pthread_sigmask(3)* 了解更多信息。

Added in version 3.5.

在*signal* 模块中定义的变量是：

signal.**SIG_DFL**

这是两种标准信号处理选项之一；它只会执行信号的默认函数。例如，在大多数系统上，对于 SIGQUIT 的默认操作是转储核心并退出，而对于*SIGCHLD* 的默认操作是简单地忽略它。

signal.**SIG_IGN**

这是另一个标准信号处理程序，它将简单地忽略给定的信号。

signal.**SIGABRT**

来自 *abort(3)* 的中止信号。

signal.**SIGALRM**

来自 *alarm(2)* 的计时器信号。

Availability: Unix.

signal.**SIGBREAK**

来自键盘的中断 (CTRL + BREAK)。

Availability: Windows.

signal.**SIGBUS**

总线错误 (非法的内存访问)。

Availability: Unix.

signal.**SIGCHLD**

子进程被停止或终结。

Availability: Unix.

signal.**SIGCLD**

SIGCHLD 的别名。

Availability: not macOS.

signal.**SIGCONT**

如果进程当前已停止则继续执行它

Availability: Unix.

signal.**SIGFPE**

浮点异常。例如除以零。

> **参见**
>
> 当除法或求余运算的第二个参数为零时会引发*ZeroDivisionError* 。

signal.**SIGHUP**

在控制终端上检测到挂起或控制进程的终止。

Availability: Unix.

signal.**SIGILL**

非法指令。

signal.**SIGINT**
来自键盘的中断 (CTRL + C)。

默认的动作是引发*KeyboardInterrupt*。

signal.**SIGKILL**
终止信号。

它不能被捕获、阻塞或忽略。

Availability: Unix.

signal.**SIGPIPE**
损坏的管道：写入到没有读取器的管道。

默认的动作是忽略此信号。

Availability: Unix.

signal.**SIGSEGV**
段错误：无效的内存引用。

signal.**SIGSTKFLT**

协处理器上的栈错误。Linux 内核不会引发此信号：它只能在用户空间中被引发。

Availability: Linux.

在信号可用的架构上。参见手册页面 *signal(7)* 了解更多信息。

Added in version 3.11.

signal.**SIGTERM**
终结信号。

signal.**SIGUSR1**
用户自定义信号 1。

Availability: Unix.

signal.**SIGUSR2**
用户自定义信号 2。

Availability: Unix.

signal.**SIGWINCH**
窗口调整大小信号。

Availability: Unix.

SIG*
所有信号编号都是符号化定义的。例如，挂起信号被定义为*signal.SIGHUP*；变量的名称与 C 程序中使用的名称相同，具体见 <signal.h>。'signal()' 的 Unix 手册页面列出了现有的信号 (在某些系统上这是 *signal(2)*，在其他系统中此列表则是在 *signal(7)* 中)。请注意并非所有系统都会定义相同的信号名称集；只有系统所定义的名称才会由此模块来定义。

signal.**CTRL_C_EVENT**
对应于 Ctrl+C 击键事件的信号。此信号只能用于*os.kill()*。

Availability: Windows.

Added in version 3.2.

signal.**CTRL_BREAK_EVENT**
对应于 Ctrl+Break 击键事件的信号。此信号只能用于*os.kill()*。

Availability: Windows.

Added in version 3.2.

signal.**NSIG**

比最高的信号编号值多一。请使用*valid_signals()* 来获取有效的信号编号。

signal.**ITIMER_REAL**

实时递减间隔计时器，并在到期时发送*SIGALRM* 。

signal.**ITIMER_VIRTUAL**

仅在进程执行时递减间隔计时器，并在到期时发送 SIGVTALRM 。

signal.**ITIMER_PROF**

当进程执行时以及当系统替进程执行时都会减小间隔计时器。这个计时器与 ITIMER_VIRTUAL 相配结，通常被用于分析应用程序在用户和内核空间中花费的时间。SIGPROF 会在超期时被发送。

signal.**SIG_BLOCK**

pthread_sigmask() 的 *how* 形参的一个可能的值，表明信号将会被阻塞。

Added in version 3.3.

signal.**SIG_UNBLOCK**

pthread_sigmask() 的 *how* 形参的是个可能的值，表明信号将被解除阻塞。

Added in version 3.3.

signal.**SIG_SETMASK**

pthread_sigmask() 的 *how* 形参的一个可能的值，表明信号掩码将要被替换。

Added in version 3.3.

signal 模块定义了一个异常:

exception signal.**ItimerError**

作为来自下层*setitimer()* 或*getitimer()* 实现错误的信号被引发。如果将无效的定时器或负的时间值传给*setitimer()* 就导致这个错误。此错误是*OSError* 的子类型。

Added in version 3.3: 此错误是*IOError* 的子类型，现在则是*OSError* 的别名。

signal 模块定义了以下函数:

signal.**alarm**(*time*)

如果 *time* 值非零，则此函数将要求将一个*SIGALRM* 信号在 *time* 秒内发往进程。任何在之前排入计划的警报都会被取消（在任何时刻都只能有一个警报被排入计划）。后续的返回值将是任何之前设置的警报被传入之前的秒数。如果 *time* 值为零，则不会将任何警报排入计划，并且任何已排入计划的警报都会被取消。如果返回值为零，则目前没有任何警报被排入计划。

Availability: Unix.

请参阅手册页面 *alarm(2)* 了解更多信息。

signal.**getsignal**(*signalnum*)

返回当前用于信号 *signalnum* 的信号处理程序。返回值可以是一个 Python 可调用对象，或是特殊值*signal.SIG_IGN*, *signal.SIG_DFL* 或*None* 之一。在这里，*signal.SIG_IGN* 表示信号在之前被忽略，*signal.SIG_DFL* 表示之前在使用默认的信号处理方式，而 None 表示之前的信号处理程序未由 Python 安装。

signal.**strsignal**(*signalnum*)

返回信号 *signalnum* 的描述信息，例如”Interrupt” 对应*SIGINT*。如果 *signalnum* 没有描述信息则返回*None*。如果 *signalnum* 无效则引发*ValueError*。

Added in version 3.8.

signal.**valid_signals**()

返回本平台上的有效信号编号集。这可能会少于 range(1, NSIG)，如果某些信号被系统保留作为内部使用的话。

Added in version 3.8.

signal.**pause**()

使进程休眠直至接收到一个信号；然后将会调用适当的处理程序。返回空值。

Availability: Unix.

请参阅手册页面 *signal(2)* 了解更多信息。

另请参阅*sigwait()*, *sigwaitinfo()*, *sigtimedwait()* 和*sigpending()*。

signal.**raise_signal**(*signum*)

向调用方进程发送一个信号。返回空值。

Added in version 3.8.

signal.**pidfd_send_signal**(*pidfd*, *sig*, *siginfo=None*, *flags=0*)

发送信号 *sig* 到文件描述符 *pidfd* 所指向的进程。Python 目前不支持 *siginfo* 形参；它必须为 None。提供 *flags* 参数是为了将来扩展；当前未定义旗标值。

更多信息请参阅 *pidfd_send_signal(2)* 手册页面。

Availability: Linux >= 5.1, Android >= *build-time* API level 31

Added in version 3.9.

signal.**pthread_kill**(*thread_id*, *signalnum*)

将信号 *signalnum* 发送至与调用者在同一进程中另一线程 *thread_id*。目标线程可被用于执行任何代码（Python 或其它）。但是，如果目标线程是在执行 Python 解释器，则 Python 信号处理程序将由主解释器的主线程来执行。因此，将信号发送给特定 Python 线程的唯一作用在于强制让一个正在运行的系统调用失败并抛出*InterruptedError*。

使用*threading.Thread* 对象的*threading.get_ident()* 或*ident* 属性为 *thread_id* 获取合适的值。

如果 *signalnum* 为 0，则不会发送信号，但仍然会执行错误检测；这可被用来检测目标线程是否仍在运行。

引发一个审计事件 signal.pthread_kill 并附带参数 thread_id, signalnum。

Availability: Unix.

请参阅手册页面 *pthread_kill(3)* 了解更多信息。

另请参阅*os.kill()*。

Added in version 3.3.

signal.**pthread_sigmask**(*how*, *mask*)

获取和/或修改调用方线程的信号掩码。信号掩码是一组传送过程目前为调用者而阻塞的信号集。返回旧的信号掩码作为一组信号。

该调用的行为取决于 *how* 的值，具体见下。

- *SIG_BLOCK*: 被阻塞信号集是当前集与 *mask* 参数的并集。
- *SIG_UNBLOCK*: *mask* 中的信号会从当前已阻塞信号集中被移除。允许尝试取消对一个非阻塞信号的阻塞。
- *SIG_SETMASK*: 已阻塞信号集会被设为 *mask* 参数的值。

mask 是一个信号编号集合 (例如 {*signal.SIGINT*, *signal.SIGTERM*})。请使用*valid_signals()* 表示包含所有信号的完全掩码。

例如，signal.pthread_sigmask(signal.SIG_BLOCK, []) 会读取调用方线程的信号掩码。

SIGKILL 和 SIGSTOP 不能被阻塞。

Availability: Unix.

请参阅手册页面 *sigprocmask(2)* 和 *pthread_sigmask(3)* 了解更多信息。

另请参阅*pause()*, *sigpending()* 和*sigwait()*。

Added in version 3.3.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)

设置由 *which* 指明的给定间隔计时器 (*signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* 或 *signal.ITIMER_PROF* 之一) 在 *seconds* 秒 (接受浮点数值，为与 *alarm()* 之差) 之后开始并在每 *interval* 秒间隔时 (如果 *interval* 不为零) 启动。由 *which* 指明的间隔计时器可通过将 *seconds* 设为零来清空。

当一个间隔计时器启动时，会有信号发送至进程。所发送的具体信号取决于所使用的计时器；*signal.ITIMER_REAL* 将发送 *SIGALRM*, *signal.ITIMER_VIRTUAL* 将发送 SIGVTALRM, 而 *signal.ITIMER_PROF* 将发送 SIGPROF.

原有的值会以元组: (delay, interval) 的形式被返回。

尝试传入无效的计时器将导致 *ItimerError*。

Availability: Unix.

signal.**getitimer**(*which*)

返回由 *which* 指明的给定间隔计时器当前的值。

Availability: Unix.

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)

将唤醒文件描述符设为 *fd*。当某个你的程序已注册了信号处理器的信号被接收时，该信号的编号会以单个字节的形式写入 fd。如果你没有为你关注的信号注册信号处理器，则不会有任何内容被写入唤醒文件描述符。这可以被某个库用来唤醒一次 poll 或 select 调用，以允许该信号被完整地处理。

原有的唤醒 fd 会被返回（或者如果未启用文件描述符唤醒则返回 -1）。如果 *fd* 为 -1，文件描述符唤醒会被禁用。如果不为 -1，则 *fd* 必须为非阻塞型。需要由库来负责在重新调用 poll 或 select 之前从 *fd* 移除任何字节数据。

当启用线程用时，此函数只能从*主解释器的主线程* 被调用；尝试从另一线程调用它将导致 *ValueError* 异常被引发。

使用此函数有两种通常的方式。在两种方式下，当有信号到达时你都是用 fd 来唤醒，但之后它们在确定达到的一个或多个信号 *which* 时存在差异。

在第一种方式下，我们从 fd 的缓冲区读取数据，这些字节值会给你信号编号。这种方式很简单，但在少数情况下会发生问题：通常 fd 将有缓冲区空间大小限制，如果信号到达得太多且太快，缓冲区可能会爆满，有些信号可能丢失。如果你使用此方式，则你应当设置 warn_on_full_buffer=True，当信号丢失时这至少能将警告消息打印到 stderr。

在第二种方式下，我们 *只* 会将唤醒 fd 用于唤醒，而忽略实际的字节值。在此情况下，我们所关心的只有 fd 的缓冲区为空还是不为空；爆满的缓冲区完全不会导致问题。如果你使用此方式，则你应当设置 warn_on_full_buffer=False，这样你的用户就不会被虚假的警告消息所迷惑。

在 3.5 版本发生变更: 在 Windows 上，此函数现在也支持套接字处理。

在 3.7 版本发生变更: 添加了 warn_on_full_buffer 形参。

signal.**siginterrupt**(*signalnum*, *flag*)

更改系统调用重启行为：如果 *flag* 为 *False*，系统调用将在被信号 *signalnum* 中断时重启，否则系统调用将被中断。返回空值。

Availability: Unix.

请参阅手册页面 *siginterrupt(3)* 了解更多信息。

请注意使用 *signal()* 安装信号处理器将会通过隐式地调用 siginterrupt() 并为给定信号的 *flag* 设置真值来将重启行为重置为可中断的。

signal.**signal**(*signalnum*, *handler*)

将信号 *signalnum* 的处理程序设为函数 *handler*。*handler* 可以为接受两个参数（见下）的 Python 可调用对象，或者为特殊值 *signal.SIG_IGN* 或 *signal.SIG_DFL* 之一。之前的信号处理程序将被返回（参见上文 *getsignal()* 的描述）。(更多信息请参阅 Unix 手册页面 *signal(2)*。)

当启用线程用时，此函数只能从*主解释器的主线程* 被调用；尝试从另一线程调用它将导致 *ValueError* 异常被引发。

handler 将附带两个参数调用：信号编号和当前堆栈帧 (None 或一个帧对象；有关帧对象的描述请参阅 类型层级结构描述或者参阅*inspect* 模块中的属性描述)。

在 Windows 上，*signal()* 调用只能附带*SIGABRT*, *SIGFPE*, *SIGILL*, *SIGINT*, *SIGSEGV*, *SIGTERM* 或*SIGBREAK*。任何其他值都将引发*ValueError*。请注意不是所有系统都定义了同样的信号名称集合；如果一个信号名称未被定义为 SIG* 模块层级常量则将引发*AttributeError*。

signal.**sigpending**()

检查正在等待传送给调用方线程的信号集合（即在阻塞期间被引发的信号）。返回正在等待的信号集合。

Availability: Unix.

请参阅手册页面 *sigpending(2)* 了解更多信息。

另请参阅*pause()*, *pthread_sigmask()* 和*sigwait()*。

Added in version 3.3.

signal.**sigwait**(*sigset*)

挂起调用方线程的执行直到信号集合 *sigset* 中指定的信号之一被传送。此函数会接受该信号（将其从等待信号列表中移除），并返回信号编号。

Availability: Unix.

请参阅手册页面 *sigwait(3)* 了解更多信息。

另请参阅*pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* 和*sigtimedwait()*。

Added in version 3.3.

signal.**sigwaitinfo**(*sigset*)

挂起调用方线程的执行直到信号集合 *sigset* 中指定的信号之一被传送。此函数会接受该信号并将其从等待信号列表中移除。如果 *sigset* 中的信号之一已经在等待调用方线程，此函数将立即返回并附带有关该信号的信息。被传送信号的信号处理程序不会被调用。如果该函数被某个不在 *sigset* 中的信号中断则会引发*InterruptedError*。

返回值是一个代表 siginfo_t 结构体所包含数据的对象，具体为: si_signo, si_code, si_errno, si_pid, si_uid, si_status, si_band。

Availability: Unix.

请参阅手册页面 *sigwaitinfo(2)* 了解更多信息。

另请参阅*pause()*, *sigwait()* 和*sigtimedwait()*。

Added in version 3.3.

在 3.5 版本发生变更: 当被某个不在 *sigset* 中的信号中断时本函数将进行重试并且信号处理程序不会引发异常（请参阅 **PEP 475** 了解其理由）。

signal.**sigtimedwait**(*sigset*, *timeout*)

与*sigwaitinfo()* 类似，但会接受一个额外的 *timeout* 参数来指定超时限制。如果将 *timeout* 指定为 0，则会执行轮询。如果发生超时则返回*None*。

Availability: Unix.

请参阅手册页面 *sigtimedwait(2)* 了解更多信息。

另请参阅*pause()*, *sigwait()* 和*sigwaitinfo()*。

Added in version 3.3.

在 3.5 版本发生变更: 现在当此函数被某个不在 *sigset* 中的信号中断时将以计算出的 *timeout* 进行重试并且信号处理程序不会引发异常（请参阅 **PEP 475** 了解其理由）。

19.6.3 例子

这是一个最小示例程序。它使用*alarm()* 函数来限制等待打开一个文件所花费的时间；这在文件为无法开启的串行设备时会很有用处，此情况通常会导致*os.open()* 无限期地挂起。解决办法是在打开文件之前设置 5 秒钟的 alarm；如果操作耗时过长，将会发送 alarm 信号，并且处理程序会引发一个异常。

```
import signal, os

def handler(signum, frame):
    signame = signal.Signals(signum).name
    print(f'Signal handler called with signal {signame} ({signum})')
    raise OSError("Couldn't open device!")

# 设置信号处理器及 5 秒警报
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# open() 可能会无限挂起
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # 禁用警报
```

19.6.4 对于 SIGPIPE 的说明

将你的程序用管道输出到工具例如 *head(1)* 将会导致*SIGPIPE* 信号在其标准输出的接收方提前关闭时被发送到你的进程。这将引发一个异常例如 `BrokenPipeError: [Errno 32] Broken pipe`。要处理这种情况，请对你的入口点进行包装以捕获此异常，如下所示:

```
import os
import sys

def main():
    try:
        # 模拟大量输出 (你的代码将替换此循环)
        for x in range(10000):
            print("y")
        # 在此刷新输出以在此 try 代码块内部时
        # 强制让 SIGPIPE 被触发。
        sys.stdout.flush()
    except BrokenPipeError:
        # Python 在退出时刷新标准流；将剩余的输出
        # 重定向到 devnull 以避免关闭时引发新的 BrokenPipeError
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python 退出时在 EPIPE 上输出错误码 1

if __name__ == '__main__':
    main()
```

请不要将*SIGPIPE* 的处置方式设为*SIG_DFL* 以避免*BrokenPipeError*。这样做还会在你的程序所写入的任何套接字连接中断时导致你的程序异常退出。

19.6.5 有关信号处理器和异常的注释

如果一个信号处理器引发了异常，该异常将被传播到主线程并可能在任何*bytecode* 指令之后被引发，在执行期间的任何时候都可能出现*KeyboardInterrupt*。大多数 Python 代码，包括标准库的代码都不能对此进行健壮性处理，因此*KeyboardInterrupt* (或由信号处理器所导致的任何其他异常) 可能会在极少数情况下使程序处于非预期的状态。

为了展示这个问题，请考虑以下代码:

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # 如果 KeyboardInterrupt 在这里发生，将保持一切正常
        self.lock.acquire()
        # 如果 KeyboardInterrupt 在这里发生，__exit__ 将不会被调用
        ...
        # KeyboardInterrupt 可能会在该函数返回之前发生

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

对于许多程序，特别是那些在遇到*KeyboardInterrupt* 只需直接退出的程序来说，这不是个问题，但是高复杂度或要求高可靠性的应用程序则应当避免由于信号处理器引发异常。他们还应当避免将捕获*KeyboardInterrupt* 作为程序关闭的优雅方式。相反地，他们应当安装自己的*SIGINT* 处理器。下面是一个避免了*KeyboardInterrupt* 的 HTTP 服务器示例:

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

19.7 mmap --- 内存映射文件支持

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

内存映射文件对象的行为既像*bytearray* 又像*文件对象*。你可以在大部分接受*bytearray* 的地方使用 mmap 对象；例如，你可以使用*re* 模块来搜索一个内存映射文件。你也可以通过执行 `obj[index] = 97` 来修改单个字节，或者通过对切片赋值来修改一个子序列: `obj[i1:i2] = b'...'`。你还可以在文件的当前位置开始读取和写入数据，并使用 `seek()` 前往另一个位置。

内存映射文件是由*mmap*构造器创建的，它在 Unix 和在 Windows 上会有所不同。无论在哪种情况下你都必须为一个打开用于更新的文件提供文件描述符。如果你想要映射一个已有的 Python 文件对象，请使用其*fileno()*方法来为 *fileno* 形参获取正确的值。否则，你可以使用*os.open()*函数来打开这个文件，它会直接返回一个文件描述符（结束时仍然需要关闭该文件）。

备注

如果要为可写的缓冲文件创建内存映射，则应当首先*flush()*该文件。这确保了对缓冲区的本地修改在内存映射中可用。

对于 Unix 和 Windows 版本的构造函数，可以将 *access* 指定为可选的关键字参数。*access* 接受以下四个值之一：ACCESS_READ，ACCESS_WRITE 或 ACCESS_COPY 分别指定只读，直写或写时复制内存，或 ACCESS_DEFAULT 推迟到 *prot*。*access* 可以在 Unix 和 Windows 上使用。如果未指定 *access*，则 Windows mmap 返回直写映射。这三种访问类型的初始内存值均取自指定的文件。向 ACCESS_READ 内存映射赋值会引发*TypeError*异常。向 ACCESS_WRITE 内存映射赋值会影响内存和底层的文件。向 ACCESS_COPY 内存映射赋值会影响内存，但不会更新底层的文件。

在 3.7 版本发生变更: 添加了 ACCESS_DEFAULT 常量。

要映射匿名内存，应将 -1 作为 fileno 和 length 一起传递。

class mmap.**mmap**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*, *offset=0*)

(Windows 版本) 映射被文件句柄 *fileno* 指定的文件的 *length* 个字节，并创建一个 mmap 对象。如果 *length* 大于当前文件大小，则文件将扩展为包含 *length* 个字节。如果 *length* 为 0，则映射的最大长度为当前文件大小。如果文件为空，Windows 会引发异常（你无法在 Windows 上创建空映射）。

如果指定了 *tagname* 并且不为 None，则将是一个为映射提供标签名称的字符串。Windows 允许对同一文件设置许多不同的映射。如果指定一个现有标签的名称，则将打开该标签，否则将创建一个具有该名称的新标签。如果此形参被省略或为 None，则创建的映射将不带名称。避免使用 *tagname* 形参将有助于使你的代码在 Unix 和 Windows 之间可移植。

offset 可以被指定为非负整数偏移量。mmap 引用将相对于从文件开头的偏移。*offset* 默认为 0。*offeset* 必须是 ALLOCATIONGRANULARITY 的倍数。

引发一个审计事件 mmap.__new__ 并附带参数 fileno, length, access, offset。

class mmap.**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE | PROT_READ*, *access=ACCESS_DEFAULT*, *offset=0*, *, *trackfd=True*)

(Unix 版本) 映射文件描述符 *fileno* 指定的文件的 *length* 个字节，并返回一个 mmap 对象。如果 *length* 为 0，则当调用*mmap*时，映射的最大长度将为文件的当前大小。

flags 指明映射的性质。*MAP_PRIVATE* 会创建私有的写入时拷贝映射，因此对 mmap 对象内容的修改将为该进程所私有。而*MAP_SHARED* 会创建与其他映射同一文件区域的进程所共享的映射。默认值为*MAP_SHARED*。某些系统还具有额外的可用旗标，完整列表会在*MAP_* 常量* 中指明。

如果指明了 *prot*，它将给出所需的内存保护方式；最有用的两个值是 PROT_READ 和 PROT_WRITE，分别指明页面为可读或可写。*prot* 默认为 PROT_READ | PROT_WRITE。

可以指定 *access* 作为替代 *flags* 和 *prot* 的可选关键字形参。同时指定 *flags*, *prot* 和 *access* 将导致错误。请参阅上文中 *access* 的描述了解有关如何使用此形参的信息。

offset 可以被指定为非负整数偏移量。mmap 引用将相对于从文件开头的偏移。*offset* 默认为 0。*offset* 必须是 ALLOCATIONGRANULARITY 的倍数，它在 Unix 系统上等价于 PAGESIZE。

如果 *trackfd* 为 False，则由 *fileno* 指定的文件描述符将不会被复制，而结果 mmap 对象将不会被关联到映射的下层文件。这意味着*size()*和*resize()*方法将会失败。此模式适用于限制打开文件描述符的数量。

为了确保已创建内存映射的有效性，描述符 *fileno* 所指定的文件在 macOS 上会与物理后备存储进行内部自动同步。

在 3.13 版本发生变更: 增加了 *trackfd* 形参。

这个例子演示了使用*mmap*的简单方式:

```
import mmap

# 写入一个简单的示例文件
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # 对文件进行内存映射，大小为 0 表示整个文件
    mm = mmap.mmap(f.fileno(), 0)
    # 通过标准文件方法读取内容
    print(mm.readline())  # prints b"Hello Python!\n"
    # 通过切片标记方式读取内容
    print(mm[:5])  # 打印 b"Hello"
    # 使用切片标记方式更新内容；
    # 请注意新内容必须为相同的大小
    mm[6:] = b" world!\n"
    # ... 并使用标准文件方法再次读取
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # 关闭映射
    mm.close()
```

mmap 也可以在 `with` 语句中被用作上下文管理器:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Added in version 3.2: 上下文管理器支持。

下面的例子演示了如何创建一个匿名映射并在父进程和子进程之间交换数据。:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

引发一个审计事件 `mmap.__new__` 并附带参数 `fileno`, `length`, `access`, `offset`。

映射内存的文件对象支持以下方法:

close()

关闭 mmap。后续调用该对象的其他方法将导致引发 ValueError 异常。此方法将不会关闭打开的文件。

closed

如果文件已关闭则返回 `True`。

Added in version 3.2.

find(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在对象内被找到的最小索引号，使得 *sub* 被包含在 [*start*, *end*] 范围中。可选参数 *start* 和 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

在 3.5 版本发生变更: 现在接受可写的字节类对象。

flush([*offset*[, *size*]])

将对文件的内存副本的修改刷新至磁盘。如果不使用此调用则无法保证在对象被销毁前将修改写回存储。如果指定了 *offset* 和 *size*，则只将对指定范围内字节的修改刷新至磁盘；在其他情况下，映射的全部范围都会被刷新。*offset* 必须为 PAGESIZE 或 ALLOCATIONGRANULARITY 的倍数。

返回 None 以表示成功。当调用失败时将引发异常。

在 3.8 版本发生变更: 在之前版本中，成功时将返回非零值；在 Windows 下当发生错误时将返回零。在 Unix 下成功时将返回零值；当发生错误时将引发异常。

madvise(*option*[, *start*[, *length*]])

将有关内存区域的建议 *option* 发送至内核，从 *start* 开始扩展 *length* 个字节。*option* 必须为系统中可用的*MADV_* 常量* 之一。如果省略 *start* 和 *length*，则会包含整个映射。在某些系统中（包括 Linux），*start* 必须为 PAGESIZE 的倍数。

可用性: 具有 madvise() 系统调用的系统。

Added in version 3.8.

move(*dest*, *src*, *count*)

将从偏移量 *src* 开始的 *count* 个字节拷贝到目标索引号 *dest*。如果 mmap 创建时设置了 ACCESS_READ，则调用 move 将引发*TypeError* 异常。

read([*n*])

返回一个*bytes*，其中包含从当前文件位置开始的至多 *n* 个字节。如果参数省略，为 None 或负数，则返回从当前文件位置开始直至映射结尾的所有字节。文件位置会被更新为返回字节数据之后的位置。

在 3.3 版本发生变更: 参数可被省略或为 None。

read_byte()

将当前文件位置上的一个字节以整数形式返回，并让文件位置前进 1。

readline()

返回一个单独的行，从当前文件位置开始直到下一个换行符。文件位置会被更新为返回字节数据之后的位置。

resize(*newsize*)

改变映射和下层文件的大小，如果存在的话。

改变具有 ACCESS_READ 或 ACCESS_COPY 访问权限的已创建映射的大小，将引发一个*TypeError* 异常。改变 *trackfd* 被设为 False 的已创建映射的大小，将引发一个*ValueError* 异常。

在 Windows 上: 如果存在其他针对相同名称文件的映射则改变映射大小将引发*OSError*。改变匿名映射（即针对分页文件）的大小将静默地创建一个新映射并将原始数据复制到对应新大小的长度。

在 3.11 版本发生变更: 如果在持有另一个映射允许在 Windows 上针对匿名映射改变大小的情况下尝试改变大小则会正确地报告失败

rfind(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在对象内被找到的最大索引号，使得 *sub* 被包含在 [*start*, *end*] 范围中。可选参数 *start* 和 *end* 会被解读为切片表示法。如果未找到则返回 -1。

在 3.5 版本发生变更: 现在接受可写的字节类对象。

seek(*pos*[, *whence*])

设置文件的当前位置。*whence* 参数为可选项并且默认为 os.SEEK_SET 或 0 (绝对文件定位)；其他值还有 os.SEEK_CUR 或 1 (相对当前位置查找) 和 os.SEEK_END 或 2 (相对文件末尾查找)。

在 3.13 版本发生变更: 返回一个新的绝对位置值而非 None。

seekable()

返回文件是否支持定位，返回值将始终为 `True`。

Added in version 3.13.

size()

返回文件的长度，该数值可以大于内存映射区域的大小。

tell()

返回文件指针的当前位置。

write(*bytes*)

将 *bytes* 中的字节数据写入文件指针当前位置的内存并返回写入的字节总数 (一定不小于 `len(bytes)`，因为如果写入失败，将会引发*ValueError*)。在字节数据被写入后文件位置将会更新。如果 mmap 创建时设置了 `ACCESS_READ`，则向其写入将引发*TypeError* 异常。

在 3.5 版本发生变更: 现在接受可写的字节类对象。

在 3.6 版本发生变更: 现在会返回写入的字节总数。

write_byte(*byte*)

将整数值 *byte* 写入文件指针当前位置的内存；文件位置前进 `1`。如果 mmap 创建时设置了 `ACCESS_READ`，则向其写入将引发*TypeError* 异常。

19.7.1 MADV_* 常量

mmap.**MADV_NORMAL**

mmap.**MADV_RANDOM**

mmap.**MADV_SEQUENTIAL**

mmap.**MADV_WILLNEED**

mmap.**MADV_DONTNEED**

mmap.**MADV_REMOVE**

mmap.**MADV_DONTFORK**

mmap.**MADV_DOFORK**

mmap.**MADV_HWPOISON**

mmap.**MADV_MERGEABLE**

mmap.**MADV_UNMERGEABLE**

mmap.**MADV_SOFT_OFFLINE**

mmap.**MADV_HUGEPAGE**

mmap.**MADV_NOHUGEPAGE**

mmap.**MADV_DONTDUMP**

mmap.**MADV_DODUMP**

mmap.**MADV_FREE**

mmap.**MADV_NOSYNC**

mmap.**MADV_AUTOSYNC**

mmap.**MADV_NOCORE**

mmap.**MADV_CORE**

mmap.**MADV_PROTECT**

mmap.**MADV_FREE_REUSABLE**

mmap.**MADV_FREE_REUSE**

这些选项可被传给*mmap.madvise()*。不是每个选项都存在于每个系统中。

可用性: 具有 madvise() 系统调用的系统。

Added in version 3.8.

19.7.2 MAP_* 常量

mmap.**MAP_SHARED**

mmap.**MAP_PRIVATE**

mmap.**MAP_32BIT**

mmap.**MAP_ALIGNED_SUPER**

mmap.**MAP_ANON**

mmap.**MAP_ANONYMOUS**

mmap.**MAP_CONCEAL**

mmap.**MAP_DENYWRITE**

mmap.**MAP_EXECUTABLE**

mmap.**MAP_HASSEMAPHORE**

mmap.**MAP_JIT**

mmap.**MAP_NOCACHE**

mmap.**MAP_NOEXTEND**

mmap.**MAP_NORESERVE**

mmap.**MAP_POPULATE**

mmap.**MAP_RESILIENT_CODESIGN**

mmap.**MAP_RESILIENT_MEDIA**

mmap.**MAP_STACK**

mmap.**MAP_TPRO**

mmap.**MAP_TRANSLATED_ALLOW_EXECUTE**

mmap.**MAP_UNIX03**

以下是可被传给*mmap.mmap()* 的各种旗标。*MAP_ALIGNED_SUPER* 仅在 FreeBSD 上可用而*MAP_CONCEAL* 仅在 OpenBSD 上可用。请注意某些选项在某些系统上可能不存在。

在 3.10 版本发生变更: 增加了*MAP_POPULATE* 常量。

Added in version 3.11: 增加了*MAP_STACK* 常量。

Added in version 3.12: 增加了*MAP_ALIGNED_SUPER* 和*MAP_CONCEAL* 常量。

Added in version 3.13: 增加了*MAP_32BIT*, *MAP_HASSEMAPHORE*, *MAP_JIT*, *MAP_NOCACHE*, *MAP_NOEXTEND*, *MAP_NORESERVE*, *MAP_RESILIENT_CODESIGN*, *MAP_RESILIENT_MEDIA*, *MAP_TPRO*, *MAP_TRANSLATED_ALLOW_EXECUTE* 和*MAP_UNIX03* 等常量。

CHAPTER 20

互联网数据处理

本章介绍了一些支持处理因特网上常用数据格式的模块。

20.1 `email` --- 电子邮件与 MIME 处理包

源代码: Lib/email/__init__.py

`email` 包是一个用于管理电子邮件消息的库。它 并非专门被设计用来执行向 SMTP (**RFC 2821**), NNTP 或其他服务器发送电子邮件消息；这些是`smtplib` 等模块的功能。`email` 包试图尽可能地遵循 RFC，支持 **RFC 5322** 和 **RFC 6532**，以及与 MIME 相关的各个 RFC 例如 **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183** 和 **RFC 2231**。

email 包的总体结构可以分为三个主要组件，另外还有第四个组件用于控制其他组件的行为。

这个包的中心组件是代表电子邮件消息的 “对象模型”。应用程序主要通过在`message` 子模块中定义的对象模型接口与这个包进行交互。应用程序可以使用此 API 来询问有关现有电子邮件的问题、构造新的电子邮件，或者添加或移除自身也使用相同对象模型接口的电子邮件子组件。也就是说，遵循电子邮件消息及其 MIME 子组件的性质，电子邮件对象模型是所有提供`EmailMessage` API 的对象所构成的树状结构。

这个包的另外两个主要组件是`parser` 和`generator`。parser 接受电子邮件消息的序列化版本（字节流）并将其转换为`EmailMessage` 对象树。generator 接受`EmailMessage` 并将其转回序列化的字节流。(parser 和 generator 还能处理文本字符流，但不建议这种用法，因为这很容易导致某种形式的无效消息。

控制组件是`policy` 模块。每一个`EmailMessage`、每一个`generator` 和每一个`parser` 都有一个相关联的`policy` 对象来控制其行为。通常应用程序只有在`EmailMessage` 被创建时才需要指明控制策略，或者通过直接实例代`EmailMessage` 来新建电子邮件，或者通过使用`parser` 来解析输入流。但是策略也可以在使用`generator` 序列化消息时被更改。例如，这允许从磁盘解析通用电子邮件消息，而在将消息发送到电子邮件服务器时使用标准 SMTP 设置对其进行序列化。

email 包会尽量地对应用程序隐藏各种控制类 RFC 的细节。从概念上讲应用程序应当能够将电子邮件消息视为 Unicode 文本和二进制附件的结构化树，而不必担心在序列化时要如何表示它们。但在实际中，经常有必要至少了解一部分控制类 MIME 消息及其结构的规划，特别是 MIME ” 内容类型” 的名称和性质以及它们是如何标识多部分文档的。在大多数情况下这些知识应当仅对于更复杂的应用程序来说才是必需的，并且即便在那时它也应当仅是特定的高层级结构，而不是如何表示这些结构的细节信息。由于 MIME 内容类型被广泛应用于现代因特网软件（而非只是电子邮件），因此这对许多程序员来说将是很熟悉的概念。

以下小节描述了*email* 包的具体功能。我们会从*message* 对象模型开始，它是应用程序将要使用的主要接口，之后是*parser* 和*generator* 组件。然后我们会介绍*policy* 控制组件，它将完成对这个库的主要组件的处理。

接下来的三个小节会介绍这个包可能引发的异常以及*parser* 可能检测到的缺陷（即与 RFC 不相符）。然后我们会介绍*headerregistry* 和*contentmanager* 子组件，它们分别提供了用于更精细地操纵标题和载荷的工具。这两个组件除了包含使用与生成非简单消息的相关特性，还记录了它们的可扩展性 API，这将是高级应用程序所感兴趣的内容。

在此之后是一组使用之前小节所介绍的 API 的基本部分的示例。

前面的内容是 email 包的现代（对 Unicode 支持良好）API。从*Message* 类开始的其余小节则介绍了旧式*compat32* API，它会更直接地处理如何表示电子邮件消息的细节。*compat32* API 不会向应用程序隐藏 RFC 的相关细节，但对于需要进行此种层级操作的应用程序来说将是很有用的工具。此文档对于因向下兼容理由而仍然使用*compat32* API 的应用程序也是很适合的。

在 3.6 版本发生变更: 文档经过重新组织和撰写以鼓励使用新的*EmailMessage*/*EmailPolicy* API。

email 包文档的内容:

20.1.1 email.message: 表示电子邮件消息

源代码: Lib/email/message.py

Added in version 3.6:[1]

位于*email* 包的中心的类就是*EmailMessage* 类。这个类导入自*email.message* 模块。它是*email* 对象模型的基类。*EmailMessage* 为设置和查询头字段内容、访问信息体的内容、以及创建和修改结构化信息提供了核心功能。

一份电子邮件信息由 *标头* 和 *载荷*（又被称为 *内容*）组成。标头遵循 **RFC 5322** 或者 **RFC 6532** 风格的字段名和值，字段名和字段值之间由一个冒号隔开。这个冒号既不属于字段名，也不属于字段值。信息的载荷可能是一段简单的文字消息，也可能是一个二进制的对象，更可能是由多个拥有各自标头和载荷的子信息组成的结构化子信息序列。对于后者类型的载荷，信息的 MIME 类型将会被指明为诸如 *multipart/** 或 *message/rfc822* 的类型。

EmailMessage 对象所提供的抽象概念模型是一个头字段组成的有序字典加一个代表 **RFC 5322** 标准的信息体的 *载荷*。载荷有可能是一系列子 EmailMessage 对象的列表。你除了可以通过一般的字典方法来访问头字段名和值，还可以使用特制方法来访问头的特定字段（比如说 MIME 内容类型字段）、操纵载荷、生成信息的序列化版本、递归遍历对象树。

EmailMessage 字典型接口使用标头名称作为索引，它必须是 ASCII 值。字典的值是包含一些附加方法的字符串。标头是以保留大小写的形式存储和返回的，但字段名的匹配则是大小写不敏感的。键是保留顺序的，但与真正字典不同的是键可以重复。提供了一些额外的方法来处理包含重复键的标头。

载荷 是多样的。对于简单的信息对象，它是字符串或字节对象；对于诸如 *multipart/** 和 *message/rfc822* 信息对象的 MIME 容器文档，它是一个*EmailMessage* 对象列表。

class email.message.**EmailMessage**(*policy=default*)

如果指定了 *policy*，消息将由这个 *policy* 所指定的规则来更新和序列化信息的表达。如果没有指定 *policy*，其将默认使用*default* 策略。这个策略遵循电子邮件的 RFC 标准，除了行终止符号（RFC 要求使用 \r\n，此策略使用 Python 标准的 \n 行终止符）。请前往*policy* 的文档获取更多信息。

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)

以一段字符串的形式返回整个消息对象。若可选的 *unixfrom* 为真值，信封头将包括在返回的字符串中。*unixfrom* 默认为 False。为了保持向下兼容基类*Message* 还接受 *maxheaderlen*，但其默认为 None，这意味着在默认情况下行长度将由策略的*max_line_length* 来控制。*policy* 参数可以被用于覆盖从消息实例获取到的默认策略。这可以被用来控制该方法所产生的某些格式，因为指定的 *policy* 将被传给*Generator*。

[1] 原先在 3.4 版本中以*provisional module* 添加。过时的文档被移动至*email.message.Message: 使用 compat32 API 来表示电子邮件消息*。

扁平化信息可能会对*EmailMessage*做出修改。这是因为为了完成向字符串的转换，一些内容需要使用默认值填入（举个例子，MIME 边界字段可能会被生成或被修改）。

请注意，这个方法是为了便利而提供，不一定是适合你的应用程序的最理想的序列化信息的方法。这在你处理多封信息的时候尤甚。如果你需要使用更加灵活的 API 来序列化信息，请参见*email.generator.Generator* 。同时请注意，当*utf8* 属性为其默认值 `False` 的时候，本方法将限制其行为为生成以“7 bit clean”方式序列化的信息。

在 3.6 版本发生变更: *maxheaderlen* 没有被指定时的默认行为从默认为 0 修改为默认为策略的 *max_line_length* 值。

__str__()

与 `as_string(policy=self.policy.clone(utf8=True))` 等价。这将让 `str(msg)` 产生的字符串包含人类可读的的序列化信息内容。

在 3.4 版本发生变更: 本方法开始使用 `utf8=True` ，而非*as_string()* 的直接替身。使用 `utf8=True` 会产生类似于 **RFC 6531** 的信息表达。

as_bytes(*unixfrom=False*, *policy=None*)

以字节串对象的形式返回整个扁平化后的消息。当可选的 *unixfrom* 为真值时，返回的字符串会包含信封标头。*unixfrom* 的默认值为 `False`。*policy* 参数可被用于覆盖从消息实例获取的默认 policy。这可被用来控制该方法所产生的部分格式效果，因为指定的 *policy* 将被传递给*BytesGenerator*。

扁平化信息可能会对*EmailMessage*做出修改。这是因为为了完成向字符串的转换，一些内容需要使用默认值填入（举个例子，MIME 边界字段可能会被生成或被修改）。

请注意，这个方法是为了便利而提供，不一定是适合你的应用程序的最理想的序列化信息的方法。这在你处理多封信息的时候尤甚。如果你需要使用更加灵活的 API 来序列化信息，请参见*email.generator.BytesGenerator* 。

__bytes__()

等价于*as_bytes()*。让 `bytes(msg)` 产生一个包含已序列化消息的字节串对象。

is_multipart()

如果该信息的载荷是一个子*EmailMessage* 对象列表，返回 `True` ；否则返回 `False` 。在*is_multipart()* 返回 `True` 的场合下，载荷应当是一个字符串对象（有可能是一个使用了内容传输编码进行编码的二进制载荷）。请注意，*is_multipart()* 返回 `True` 不意味着 `msg.get_content_maintype() == 'multipart'` 也会返回 `True` 。举个例子，`is_multipart` 在*EmailMessage* 是 `message/rfc822` 类型的信息的情况下，其返回值也是 `True` 。

set_unixfrom(*unixfrom*)

将信息的信封头设置为 *unixform* ，这应当是一个字符串。（在*mboxMessage* 中有关于这个头的一段简短介绍。）

get_unixfrom()

返回消息的信封头。如果信封头从未被设置过，默认返回 `None` 。

以下方法实现了对信息的头字段进行访问的类映射接口。请留意，只是类映射接口，这与平常的映射接口（比如说字典映射）有一些语义上的不同。举个例子，在一个字典当中，键之间不可重复，但是信息头字段是可以重复的。不光如此，在字典当中调用*keys()* 方法返回的结果，其顺序没有保证；但是在一个*EmailMessage* 对象当中，返回的头字段永远以其在原信息当中出现的顺序，或以其加入信息的顺序为序。任何删了后又重新加回去的头字段总是添加在当时列表的末尾。

这些语义上的不同是刻意而为之的，是出于在绝大多数常见使用情景中都方便的初衷下设计的。

请注意在任何情况下，消息当中的任何封包标头都不会包含在映射接口当中。

__len__()

返回标头的总数，包括重复项。

__contains__(*name*)

如果消息对象中有一个名为 *name* 的字段，其返回值为 `True` 。匹配无视大小写差异，*name* 也不包含末尾的的冒号。`in` 操作符的实现中用到了这个方法，比如说：

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

返回头字段名对应的字段值。*name* 不含冒号分隔符。如果字段未找到，返回 None 。*KeyError* 异常永不抛出。

请注意，如果对应名字的字段找到了多个，具体返回哪个字段值是未定义的。请使用*get_all()* 方法获取当前匹配字段名的所有字段值。

使用标准策略（非 compat32）时，返回值是*email.headerregistry.BaseHeader* 的某个子类的一个实例。

__setitem__(*name*, *val*)

在信息头中添加名为 *name* 值为 *val* 的字段。这个字段会被添加在已有字段列表的结尾处。

请注意，这个方法 既不会覆盖 也不会删除任何字段名重名的已有字段。如果你确实想保证新字段是整个信息头当中唯一拥有 *name* 字段名的字段，你需要先把旧字段删除。例如：

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

如果*policy* 将特定标头定义为唯一的（就像标准策略所做的一样），则当这样的标头已存在时试图为其赋值此方法会引发*ValueError*。采取此种行为是出于保持一致性的考量，但不能依赖它因为在未来我们可能会选择让这样的赋值操作自动删除现有的标头。

__delitem__(*name*)

删除信息头当中字段名匹配 *name* 的所有字段。如果匹配指定名称的字段没有找到，也不会抛出任何异常。

keys()

以列表形式返回消息头中所有的字段名。

values()

以列表形式返回消息头中所有的字段值。

items()

以二元元组的列表形式返回消息头中所有的字段名和字段值。

get(*name*, *failobj=None*)

返回对应标头字段名的值。这个方法与 __getitem__() 是一样的，只是如果对应标头不存在则返回可选的 *failobj* (*failobj* 默认为 None)。

以下是一些与头有关的更多有用方法：

get_all(*name*, *failobj=None*)

返回字段名为 *name* 的所有字段值的列表。如果信息内不存在匹配的字段，返回 *failobj* （其默认值为 None ）。

add_header(*_name*, *_value*, ***_params*)

高级头字段设定。这个方法与*__setitem__()* 类似，不过你可以使用关键字参数为字段提供附加参数。*_name* 是字段名，*_value* 是字段 主值。

对于关键字参数字典 *_params* 的每个键值对而言，它的键被用作参数的名字，其中下划线被替换为短横杠（毕竟短横杠不是合法的 Python 标识符）。一般来讲，参数以 键 ="值" 的方式添加，除非值是 None 。要真的是这样的话，只有键会被添加。

如果值含有非 ASCII 字符，你可以将值写成 (CHARSET, LANGUAGE, VALUE) 形式的三元组，这样你可以人为控制字符的字符集和语言。CHARSET 是一个字符串，它为你的值的编码命名；LANGUAGE 一般可以直接设为 None ，也可以直接设为空字符串（其他可能取值参见 **RFC 2231** ）；VALUE 是一个字符串值，其包含非 ASCII 的码点。如果你没有使用三元组，你的字符串又含有非 ASCII 字符，那么它就会使用 **RFC 2231** 中，CHARSET 为 utf-8 ，LANGUAGE 为 None 的格式编码。

例如：

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

会添加一个形如下文的头字段：

```
Content-Disposition: attachment; filename="bud.gif"
```

带有非 ASCII 字符的拓展接口：

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)

替换头字段。只会替换掉信息内找到的第一个字段名匹配 *_name* 的字段值。字段的顺序不变，原字段名的大小写也不变。如果没有找到匹配的字段，抛出*KeyError* 异常。

get_content_type()

返回信息的内容类型，其形如 *maintype/subtype*，强制全小写。如果信息的 *Content-Type* 头字段不存在则返回*get_default_type()* 的返回值；如果信息的 *Content-Type* 头字段无效则返回 text/plain 。

（根据 **RFC 2045** 所述，信息永远都有一个默认类型，所以*get_content_type()* 一定会返回一个值。**RFC 2045** 定义信息的默认类型为 *text/plain* 或 *message/rfc822*，其中后者仅出现在消息头位于一个 *multipart/digest* 容器中的场合中。如果消息头的 *Content-Type* 字段所指定的类型是无效的，**RFC 2045** 令其默认类型为 *text/plain* 。）

get_content_maintype()

返回信息的主要内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *maintype* 部分。

get_content_subtype()

返回信息的子内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *subtype* 部分。

get_default_type()

返回默认的内容类型。绝大多数的信息，其默认内容类型都是 *text/plain* 。作为 *multipart/digest* 容器内子部分的信息除外，它们的默认内容类型是 *message/rfc822* 。

set_default_type(*ctype*)

设置默认的内容类型。尽管并非强制，但是 *ctype* 仍应当是 *text/plain* 或 *message/rfc822* 二者取一。默认内容类型并不存储在 *Content-Type* 头字段当中，所以设置此项的唯一作用就是决定当 *Content-Type* 头字段在信息中不存在时，get_content_type 方法的返回值。

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

在 *Content-Type* 头字段当中设置一个参数。如果该参数已于字段中存在，将其旧值替换为 *value* 。如果 *header* 是 Content-Type （默认值），并且该头字段于信息中尚未存在，则会先添加该字段，将其值设置为 *text/plain* ，并附加参数值。可选的 *header* 可以让你指定 *Content-Type* 之外的另一个头字段。

如果值包含非 ASCII 字符，其字符集和语言可以通过可选参数 *charset* 和 *language* 显式指定。可选参数 *language* 指定 **RFC 2231** 当中的语言，其默认值是空字符串。*charset* 和 *language* 都应当字符串。默认使用的是 utf8 *charset* ， *language* 为 None 。

如果 *replace* 为 False （默认值），该头字段会被移动到所有头字段列表的末尾。如果 *replace* 为 True ，字段会被原地更新。

于*EmailMessage* 对象而言，*requote* 参数已被弃用。

请注意标头现有的形参值可以通过标头值的*params* 属性来访问（例如 msg['Content-Type'].params['charset']）。

在 3.4 版本发生变更: 添加了 replace 关键字。

del_param(*param*, *header='content-type'*, *requote=True*)

从 *Content-Type* 头字段中完全移去给定的参数。头字段会被原地重写，重写后的字段不含参数和值。可选的 *header* 可以让你指定 *Content-Type* 之外的另一个字段。

于*EmailMessage* 对象而言，*requote* 参数已被弃用。

get_filename(*failobj=None*)

返回信息头当中 *Content-Disposition* 字段当中名为 filename 的参数值。如果该字段当中没有此参数，该方法会退而寻找 *Content-Type* 字段当中的 name 参数值。如果这个也没有找到，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

get_boundary(*failobj=None*)

返回信息头当中 *Content-Type* 字段当中名为 boundary 的参数值。如果字段当中没有此参数，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

set_boundary(*boundary*)

将 *Content-Type* 头字段的 boundary 参数设置为 *boundary* 。*set_boundary()* 方法永远都会在必要的时候为 *boundary* 添加引号。如果信息对象中没有 *Content-Type* 头字段，抛出*HeaderParseError* 异常。

请注意使用这个方法与直接删除旧的 *Content-Type* 头字段然后使用*add_header()* 方法添加一个带有新边界值参数的 *Content-Type* 头字段有细微差距。*set_boundary()* 方法会保留 *Content-Type* 头字段在原信息头当中的位置。

get_content_charset(*failobj=None*)

返回 *Content-Type* 头字段中的 charset 参数，强制小写。如果字段当中没有此参数，或者这个字段压根不存在，返回 *failobj* 。

get_charsets(*failobj=None*)

返回一个包含了信息内所有字符集名字的列表。如果信息是 *multipart* 类型的，那么列表当中的每一项都对应其载荷的子部分的字符集名字。否则，该列表是一个长度为 1 的列表。

列表当中的每一项都是一个字符串，其值为对应子部分的 *Content-Type* 头字段的 charset 参数值。如果该子部分没有此头字段，或者没有此参数，或者其主要 MIME 类型并非 *text* ，那么列表中的那一项即为 *failobj* 。

is_attachment()

如果信息头当中存在一个名为 *Content-Disposition* 的字段，且该字段的值为 attachment（大小写无关），返回 True 。否则，返回 False 。

在 3.4.2 版本发生变更: 为了与*is_multipart()* 方法一致，is_attachment 现在是一个方法，不再是属性了。

get_content_disposition()

如果信息的 *Content-Disposition* 头字段存在，返回其字段值；否则返回 None 。返回的值均为小写，不包含参数。如果信息遵循 **RFC 2183** 标准，则此方法的返回值只可能在 *inline* 、*attachment* 和 None 之间选择。

Added in version 3.5.

下列方法与信息内容（载荷）之访问与操控有关。

walk()

walk() 方法是一个多功能生成器。它可以被用来以深度优先顺序遍历信息对象树的所有部分和子部分。一般而言，*walk()* 会被用作 for 循环的迭代器，每一次迭代都返回其下一个子部分。

以下例子会打印出一封具有多部分结构之信息的每个部分的 MIME 类型。

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk 会遍历所有*is_multipart()* 方法返回 True 的部分之子部分，哪怕 msg.get_content_maintype() == 'multipart' 返回的是 False 。使用 _structure 除错帮助函数可以帮助我们在下面这个例子当中看清楚这一点：

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

在这里，message 的部分并非 multiparts ，但是它们真的包含子部分！is_multipart() 返回 True ， walk 也深入进这些子部分中。

get_body(*preferencelist=('related', 'html', 'plain')*)

返回信息的 MIME 部分。这个部分是最可能成为信息体的部分。

preferencelist 必须是一个字符串序列，其内容从 related 、html 和 plain 这三者组成的集合中选取。这个序列代表着返回的部分的内容类型之偏好。

在 get_body 方法被调用的对象上寻找匹配的候选者。

如果 related 未包括在 *preferencelist* 中，可考虑将所遇到的任意相关的根部分（或根部分的子部分）在该（子）部分与一个首选项相匹配时作为候选项。

当遇到一个 multipart/related 时，将检查 start 形参并且如果找到了一个匹配 *Content-ID* 的部分，在查找候选匹配时只考虑它。在其他情况下则只考虑 multipart/related 的第一个（默认的根）部分。

如果一个部分具有 *Content-Disposition* 标头，则当标头值为 inline 时将只考虑将该部分作为候选匹配。

如果没有任何候选部分匹配 *preferencelist* 中的任何首选项，则返回 None。

注: (1) 对于大多数应用来说有意义的 *preferencelist* 组合仅有 ('plain',),('html', 'plain') 以及默认的 ('related', 'html', 'plain')。(2) 由于匹配是从调用 get_body 的对象开始的，因此在 multipart/related 上调用 get_body 将返回对象本身，除非 *preferencelist* 具有非默认值。(3) 未指定 *Content-Type* 或者 *Content-Type* 标头无效的消息（或消息部分）将被当作具有 text/plain 类型来处理，这有时可能导致 get_body 返回非预期的结果。

iter_attachments()

返回包含所有不是候选"body" 部分的消息的直接子部分的迭代器。也就是说，跳过首次出现的每个 text/plain, text/html, multipart/related 或 multipart/alternative (除非通过 *Content-Disposition: attachment* 将它们显式地标记为附件)，并返回所有的其余部分。当直接应用于 multipart/related 时，将返回包含除根部分之外所有相关部分的迭代器 (即由 start 形参所指向的部分，或者当没有 start 形参或 start 形参不能匹配任何部分的 *Content-ID* 时则为第一部分)。当直接应用于 multipart/alternative 或非 multipart 时，将返回一个空迭代器。

iter_parts()

返回包含消息的所有直接子部分的迭代器，对于非 multipart 将为空对象。(另请参阅 *walk()*。)

get_content(**args*, *content_manager=None*, ***kw*)

调用 *content_manager* 的 *get_content()* 方法，将自身作为消息对象传入，并将其他参数或关键字作为额外参数传入。如果未指定 *content_manager*，则会使用当前 *policy* 所指定的 content_manager。

set_content(**args*, *content_manager=None*, ***kw*)

调用 *content_manager* 的 *set_content()* 方法，将自身作为消息传入，并将其他参数或关键字作为额外参数传入。如果未指定 *content_manager*，则会使用当前 *policy* 所指定的 content_manager。

make_related(*boundary=None*)

将非 multipart 消息转换为 multipart/related 消息，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

make_alternative(*boundary=None*)

将非 multipart 或 multipart/related 转换为 multipart/alternative，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

make_mixed(*boundary=None*)

将非 multipart, multipart/related 或 multipart-alternative 转换为 multipart/mixed，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

add_related(**args*, *content_manager=None*, ***kw*)

如果消息为 multipart/related，则创建一个新的消息对象，将所有参数传给其 *set_content()* 方法，并将其 *attach()* 到 multipart。如果消息为非 multipart，则先调用 *make_related()* 然后再继续上述步骤。如果消息为任何其他类型的 multipart，则会引发 *TypeError*。如果未指定 *content_manager*，则使用当前 *policy* 所指定的 content_manager。如果添加的部分没有 *Content-Disposition* 标头，则会添加一个值为 inline 的标头。

add_alternative(**args*, *content_manager=None*, ***kw*)

如果消息为 multipart/alternative，则创建一个新的消息对象，将所有参数传给其 *set_content()* 方法，并将其 *attach()* 到 multipart。如果消息为非 multipart 或 multipart/related，则先调用 *make_alternative()* 然后再继续上述步骤。如果消息为任何其他类型的 multipart，则会引发 *TypeError*。如果未指定 *content_manager*，则会使用当前 *policy* 所指定的 content_manager。

add_attachment(**args*, *content_manager=None*, ***kw*)

如果消息为 multipart/mixed，则创建一个新的消息对象，将所有参数传给其 *set_content()* 方法，并将其 *attach()* 到 multipart。如果消息为非 multipart，multipart/related 或 multipart/alternative，则先调用 *make_mixed()* 然后再继续上述步骤。如果未指定 *content_manager*，则使用当前 *policy* 所指定的 content_manager。如果添加的部分没有 *Content-Disposition* 标头，则会添加一个值为 attachment 的标头。此方法对于显式附件 (*Content-Disposition: attachment*) 和 inline 附件 (*Content-Disposition: inline*) 均可使用，只须向 content_manager 传入适当的选项即可。

clear()

移除所有载荷和所有标头。

clear_content()

移除载荷以及所有 *!Content-* 标头，保持所有其他标头不变并保留其原始顺序。

EmailMessage 对象具有下列实例属性:

preamble

MIME 文档格式在标头之后的空白行以及第一个多部分的分界字符串之间允许添加一些文本，通常，此文本在支持 MIME 的邮件阅读器中永远不可见，因为它处在标准 MIME 保护范围之外。但是，当查看消息的原始文本，或当在不支持 MIME 的阅读器中查看消息时，此文本会变得可见。

preamble 属性包含 MIME 文档开头部分的这些处于保护范围之外的文本。当*Parser* 在标头之后及第一个分界字符串之前发现一些文本时，它会将这些文本赋值给消息的 *preamble* 属性。当*Generator* 写出 MIME 消息的纯文本表示形式时，如果它发现消息具有 *preamble* 属性，它将在标头及第一个分界之间区域写出这些文本。请参阅*email.parser* 和*email.generator* 了解更多细节。

请注意如果消息对象没有前导文本，则 *preamble* 属性将为 None。

epilogue

epilogue 属性的作用方式与 *preamble* 相同，区别在于它包含在最后一个分界及消息结尾之间出现的文本。与*preamble* 类似，如果没有附加文本，则此属性将为 None。

defects

defects 属性包含在解析消息时发现的所有问题的列表。请参阅*email.errors* 了解可能的解析缺陷的详细描述。

class email.message.**MIMEPart**(*policy=default*)

这个类代表 MIME 消息的子部分。它与*EmailMessage* 相似，不同之处在于当*set_content()* 被调用时不会添加 *MIME-Version* 标头，因为子部分不需要有它们自己的 *MIME-Version* 标头。

备注

20.1.2 email.parser: 解析电子邮件消息

源代码: Lib/email/parser.py

使用以下两种方法的其中一种以创建消息对象结构：直接创建一个*EmailMessage* 对象，使用字典接口添加消息头，并且使用*set_content()* 和其他相关方法添加消息负载；或者通过解析一个电子邮件消息的序列化表达来创建消息对象结构。

email 包提供了一个可以理解包含 MIME 文档在内的绝大多数电子邮件文档结构的标准语法分析程序。你可以传递给语法分析程序一个字节串、字符串或者文件对象，语法分析程序会返回给你对应于该对象结构的根*EmailMessage* 实例。对于简单的、非 MIME 的消息，这个根对象的负载很可能就是一个包含了该消息文字内容的字符串。对于 MIME 消息，调用根对象的*is_multipart()* 方法会返回 True，其子项可以通过负载操纵方法来进行访问，例如*get_body()*、*iter_parts()* 还有*walk()*。

事实上你可以使用的语法分析程序接口有两种: *Parser* API 和增量式的*FeedParser* API。当你的全部消息内容都在内存当中，或者整个消息都保存在文件系统内的一个文件当中的时候，*Parser* API 非常有用。当你从可能会为了等待更多输入而阻塞的数据流当中读取消息（比如从套接字当中读取电子邮件消息）的时候，*FeedParser* 会更合适。*FeedParser* 会增量读取并解析消息，并且只有在你关闭语法分析程序的时候才会返回根对象。

请注意解析器可以进行有限的扩展，当然你也可以完全从零开始实现你自己的解析器。将*email* 包的内置解析器和*EmailMessage* 类连接起来的所有逻辑都保存在*Policy* 类中。因此自定义解析器可以根据其需要通过实现合适的 Policy 方法的自定义版本以任意方式创建消息对象树。

FeedParser API

从 email.feedparser 模块导入的*BytesFeedParser* 类提供了一个适合于增量解析电子邮件消息的 API，比如说在从一个可能会阻塞（例如套接字）的源当中读取消息文字的场合中它就会变得很有用。当然，*BytesFeedParser* 也可以用来解析一个已经完整包含在一个*bytes-like object* 、字符串或文件内的电子邮件消息，但是在这些场合下使用*BytesParser* API 可能会更加方便。这两个语法分析程序 API 的语义和最终结果是一致的。

BytesFeedParser 的 API 十分简洁易懂：你创建一个语法分析程序的实例，向它不断输入大量的字节直到尽头，然后关闭这个语法分析程序就可以拿到根消息对象了。在处理符合标准的消息的时候*BytesFeedParser* 非常准确；在处理不符合标准的消息的时候它做的也不差，但这视消息损坏的程度而定。它会向消息对象的*defects* 属性中写入它从消息中找到的问题列表。关于它能找到的所有问题类型的列表，详见*email.errors* 模块。

这里是*BytesFeedParser* 的 API：

class email.parser.**BytesFeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

创建一个*BytesFeedParser* 实例。可选的 *_factory* 参数是一个不带参数的可调用对象；如果没有被指定，就会使用 *policy* 参数的*message_factory* 属性。每当需要一个新的消息对象的时候，*_factory* 都会被调用。

如果指定了 *policy* 参数，它就会使用这个参数所指定的规则来更新消息的表达方式。如果没有设定 *policy* 参数，它就会使用*compat32* 策略。这个策略维持了对 Python 3.2 版本的 email 包的后向兼容性，并且使用*Message* 作为默认的工厂。其他策略使用*EmailMessage* 作为默认的 *_factory*。关于 *policy* 还会控制什么，参见*policy* 的文档。

注: **一定要指定 policy 关键字**。在未来版本的 Python 当中，它的默认值会变成*email.policy.default*。

Added in version 3.2.

在 3.3 版本发生变更: 添加了 *policy* 关键字。

在 3.6 版本发生变更: *_factory* 默认为策略 message_factory。

feed(*data*)

向语法分析程序输入更多数据。*data* 应当是一个包含一行或多行内容的*bytes-like object*。行内容可以是不完整的，语法分析程序会妥善的将这些不完整的行缝合在一起。每一行可以使用以下三种常见的终止符号的其中一种：回车符、换行符或回车符加换行符（三者甚至可以混合使用）。

close()

完成之前输入的所有数据的解析并返回根消息对象。如果在这个方法被调用之后仍然调用*feed()* 方法，结果是未定义的。

class email.parser.**FeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

行为跟*BytesFeedParser* 类一致，只不过向*feed()* 方法输入的内容必须是字符串。它的实用性有限，因为这种消息只有在其只含有 ASCII 文字，或者 utf8 被设置为 True 且没有二进制格式的附件的时候，才会有效。

在 3.3 版本发生变更: 添加了 *policy* 关键字。

Parser API

BytesParser 类从*email.parser* 模块导入，当消息的完整内容包含在一个*bytes-like object* 或文件中时它提供了可用于解析消息的 API。*email.parser* 模块还提供了*Parser* 用来解析字符串，以及只用来解析消息头的*BytesHeaderParser* 和*HeaderParser*，如果你只对消息头感兴趣就可以使用后两者。在这种场合下*BytesHeaderParser* 和*HeaderParser* 速度非常快，因为它们并不会尝试解析消息体，而是将载荷设为原始数据。

class email.parser.**BytesParser**(*_class=None*, ***, *policy=policy.compat32*)

创建一个*BytesParser* 实例。*_class* 和 *policy* 参数在含义和语义上与*BytesFeedParser* 的 *_factory* 和 *policy* 参数一致。

注: **一定要指定 policy 关键字**。在未来版本的 Python 当中，它的默认值会变成*email.policy.default*。

在 3.3 版本发生变更: 移除了在 2.4 版本中被弃用的 *strict* 参数。新增了 *policy* 关键字。

在 3.6 版本发生变更: *_class* 默认为策略 `message_factory`。

parse(*fp*, *headersonly=False*)

从二进制的类文件对象 *fp* 中读取全部数据、解析其字节内容、并返回消息对象。*fp* 必须同时支持*readline()* 方法和 `read()` 方法。

fp 内包含的字节内容必须是一块遵循 **RFC 5322** （如果 `utf8` 为 `True`，则为 **RFC 6532** ）格式风格的消息头和消息头延续行，并可能紧跟一个信封头。头块要么以数据结束，要么以一个空行为终止。跟着头块的是消息体（消息体内可能包含 MIME 编码的子部分，这也包括 *Content-Transfer-Encoding* 字段为 `8bit` 的子部分）。

可选的 *headersonly* 指示了是否应当在读取完消息头后就终止。默认值为 `False` ，意味着它会解析整个文件的全部内容。

parsebytes(*bytes*, *headersonly=False*)

与*parse()* 方法类似，只不过它要求输入为一个*bytes-like object* 而不是类文件对象。于一个*bytes-like object* 调用此方法相当于先将这些字节包装于一个*BytesIO* 实例中，然后调用*parse()* 方法。

可选的 *headersonly* 与*parse()* 方法中的 *headersonly* 是一致的。

Added in version 3.2.

class email.parser.**BytesHeaderParser**(*_class=None*, ***, *policy=policy.compat32*)

除了 *headersonly* 默认为 `True`，其他与*BytesParser* 类完全一样。

Added in version 3.3.

class email.parser.**Parser**(*_class=None*, ***, *policy=policy.compat32*)

这个类与*BytesParser* 一样，但是处理字符串输入。

在 3.3 版本发生变更: 移除了 *strict* 参数。添加了 *policy* 关键字。

在 3.6 版本发生变更: *_class* 默认为策略 `message_factory`。

parse(*fp*, *headersonly=False*)

从文本模式的文件型对象 *fp* 读取所有数据，解析所读取的文本，并返回根消息对象。*fp* 必须同时支持文件型对象上的*readline()* 和*read()* 方法。

除了文字模式的要求外，这个方法跟*BytesParser.parse()* 的运行方式一致。

parsestr(*text*, *headersonly=False*)

与*parse()* 方法类似，只不过它要求输入为一个字符串而不是类文件对象。于一个字符串对象调用此方法相当于先将 *text* 包装于一个*StringIO* 实例中，然后调用*parse()* 方法。

可选的 *headersonly* 与*parse()* 方法中的 *headersonly* 是一致的。

class email.parser.**HeaderParser**(*_class=None*, ***, *policy=policy.compat32*)

除了 *headersonly* 默认为 `True` ，其他与*Parser* 类完全一样。

考虑到从一个字符串或一个文件对象中创建一个消息对象是非常常见的任务，我们提供了四个方便的函数。它们于顶层*email* 包命名空间内可用。

email.**message_from_bytes**(*s*, *_class=None*, ***, *policy=policy.compat32*)

从一个*bytes-like object* 中返回消息对象。这与 `BytesParser().parsebytes(s)` 等价。可选的 *_class* 和 *policy* 参数与*BytesParser* 类的构造函数的参数含义一致。

Added in version 3.2.

在 3.3 版本发生变更: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_binary_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)

从打开的二进制*file object* 中返回消息对象。这与 `BytesParser().parse(fp)` 等价。*_class* 和 *policy* 参数与*BytesParser* 类的构造函数的参数含义一致。

Added in version 3.2.

在 3.3 版本发生变更: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_string**(*s*, *_class=None*, *, *policy=policy.compat32*)

从一个字符串中返回消息对象。这与 `Parser().parsestr(s)` 等价。*_class* 和 *policy* 参数与*Parser* 类的构造函数的参数含义一致。

在 3.3 版本发生变更: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)

从一个打开的*file object* 中返回消息对象。这与 `Parser().parse(fp)` 等价。*_class* 和 *policy* 参数与*Parser* 类的构造函数的参数含义一致。

在 3.3 版本发生变更: 移除了 *strict* 参数。添加了 *policy* 关键字。

在 3.6 版本发生变更: *_class* 默认为策略 `message_factory`。

这里是一个展示了你如何在 Python 交互式命令行中使用*message_from_bytes()* 的例子:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

附加说明

在解析语义的时候请注意:

- 大多数非 *multipart* 类型的消息都会被解析为一个带有字符串负载的消息对象。这些对象在调用*is_multipart()* 的时候会返回 `False`，调用*iter_parts()* 的时候会产生一个空列表。
- 所有 *multipart* 类型的消息都会被解析成一个容器消息对象。该对象的负载是一个子消息对象列表。外层的容器消息在调用*is_multipart()* 的时候会返回 `True`，在调用*iter_parts()* 的时候会产生一个子部分列表。
- 大多数内容类型为 *message/** （例如 *message/delivery-status* 和 *message/rfc822* ）的消息也会被解析为一个负载是长度为 1 的列表的容器对象。在它们身上调用*is_multipart()* 方法会返回 `True`，调用*iter_parts()* 所产生的单个元素会是一个子消息对象。
- 一些不遵循标准的消息在其内部关于它是否为 *multipart* 类型前后不一。这些消息可能在消息头的 *Content-Type* 字段中写明为 *multipart*，但它们的*is_multipart()* 方法的返回值可能是 `False`。如果这种消息被*FeedParser* 类解析，它们的 *defects* 属性列表当中会有一个*MultipartInvariantViolationDefect* 类的实例。关于更多信息，详见*email.errors*。

20.1.3 email.generator: 生成 MIME 文档

源代码: Lib/email/generator.py

最常见的任务之一是生成由消息对象结构体表示的电子消息消息的展平（序列化）版本。如果你想通过*smtplib.SMTP.sendmail()* 来发送你的消息或是将消息打印到控制台就会需要这样做。接受一个消息对象结构体并生成其序列化表示就是这些生成器类的工作。

与*email.parser* 模块一样，你并不会受限于已捆绑生成器的功能；你可以自己从头写一个。不过，已捆绑生成器知道如何以符合标准的方式来生成大多数电子邮件，应该能够很好地处理 MIME 和非 MIME 电子邮件消息，并且被设计为面向字节的解析和生成操作是互逆的，它假定两者都使用同样的非转换型*policy*。也就是说，通过*BytesParser* 类来解析序列化字节流然后再使用*BytesGenerator* 来重新生成序列化字节流应当得到与输入相同的结果[1]。（而另一方面，在由程序所构造的*EmailMessage* 上使用生成器可能导致对默认填入的*EmailMessage* 对象的改变。。）

[1] 此语句假定你使用了正确的 `unixfrom` 设置，并且没有针对自动调整的*email.policy* 设置调用（例如，*refold_source* 必须为 `none`，这 不是默认值）。这也不是 100% 为真的，因为如果消息不遵循 RFC 标准则有时实际原始文本的信息会在解析错误恢复时丢失。它的目标是在可能的情况下修复这些后续的边缘情况。

可以使用*Generator* 类将消息扁平化为文本（而非二进制数据）的序列化表示形式，但是由于 Unicode 无法直接表示二进制数据，因此消息有必要被转换为仅包含 ASCII 字符的数据，这将使用标准电子邮件 RFC 内容传输编码格式技术来编码电子邮件消息以便通过非“8 比特位兼容”的信道来传输。

为了适应 SMIME 签名消息的可重现处理过程，*Generator* 禁用了针对 `multipart/signed` 类型的消息部分及所有子部分的标头折叠。

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

返回一个*BytesGenerator* 对象，该对象将把提供给*flatten()* 方法的任何消息或者提供给*write()* 方法的任何经过代理转义编码的文本写入到*file-like object* *outfp*。*outfp* 必须支持接受二进制数据的 `write` 方法。

如果可选的 *mangle_from_* 为 `True`，则会将一个 `>` 字符放到消息体中恰好以字符串 `"From "` 打头，即开头文本为 `From` 加一个空格的任何行的前面。*mangle_from_* 默认为 *policy* 的*mangle_from_* 设置值 (对于*compat32* 策略为 `True` 而对于所有其他策略则为 `False`)。*mangle_from_* 被设计为在当消息以 Unix mbox 格式存储时使用 (参见*mailbox* 和 WHY THE CONTENT-LENGTH FORMAT IS BAD)。

如果 *maxheaderlen* 不为 `None`，则重新折叠任何长于 *maxheaderlen* 的标头行，或者如果为 `0`，则不重新包装任何标头。如果 *manheaderlen* 为 `None` (默认值)，则根据 *policy* 设置包装标头和其他消息行。

如果指定了 *policy*，则使用该策略来控制消息的生成。如果 *policy* 为 `None` (默认值)，则使用与传递给 `flatten` 的*Message* 或*EmailMessage* 对象相关联的策略来控制消息的生成。请参阅*email.policy* 了解有关 *policy* 所控制内容的详情。

Added in version 3.2.

在 3.3 版本发生变更: 添加了 *policy* 关键字。

在 3.6 版本发生变更: *mangle_from_* 和 *maxheaderlen* 形参的默认行为是遵循策略。

flatten(*msg*, *unixfrom=False*, *linesep=None*)

将将以 *msg* 为根的消息对象结构体的文本表示形式打印到创建*BytesGenerator* 实例时指定的输出文件。

如果*policy* 选项*cte_type* 为 `8bit` (默认值)，则会将未被修改的原始已解析消息中的任何标头拷贝到输出，其中会重新生成与原始数据相同的高比特位组字节数据，并保留具有它们的任何消息体部分的非 ASCII *Content-Transfer-Encoding*。如果 `cte_type` 为 `7bit`，则会根据需要使用兼容 ASCII 的 *Content-Transfer-Encoding* 来转换高比特位组字节数据。也就是说，将具有非 ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) 的部分转换为兼容 ASCII 的 *Content-Transfer-Encoding*，并使用 MIME `unknown-8bit` 字符集来编码标头中不符合 RFC 的非 ASCII 字节数据，以使其符合 RFC。

如果 *unixfrom* 为 `True`，则会在根消息对象的第一个 **RFC 5322** 标头之前打印 Unix mailbox 格式 (参见*mailbox*) 所使用的封包标头分隔符。如果根对象没有封包标头，则会创建一个标准标头。默认值为 `False`。请注意对于子部分来说，不会打印任何封包标头。

如果 *linesep* 不为 `None`，则会将其用作扁平化消息的所有行之间的分隔符。如果 *linesep* 为 `None` (默认值)，则使用在 *policy* 中指定的值。

clone(*fp*)

返回此*BytesGenerator* 实例的独立克隆，具有完全相同的选项设置，而 *fp* 为新的 *outfp*。

write(*s*)

使用 `ASCII` 编解码器和 `surrogateescape` 错误处理程序编码 *s*，并将其传递给传入到*BytesGenerator* 的构造器的 *outfp* 的 *write* 方法。

作为一个便捷工具，*EmailMessage* 提供了*as_bytes()* 和 `bytes(aMessage)` (即*__bytes__()*) 等方法，它们简单地生成一个消息对象的序列化二进制表示形式。更多细节请参阅*email.message*。

因为字符串无法表示二进制数据，*Generator* 类必须将任何消息中扁平化的任何二进制数据转换为兼容 ASCII 的格式，具体将其转换为兼容 ASCII 的 *Content-Transfer_Encoding*。使用电子邮件 RFC 的术语，你可以将其视作*Generator* 序列化为不” 支持 8 比特” 的 I/O 流。换句话说，大部分应用程序将需要使用*BytesGenerator*，而非*Generator*。

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

返回一个*Generator*，它将把提供给*flatten()* 方法的任何消息，或者提供给*write()* 方法的任何文本写入到*file-like object* *outfp*。*outfp* 必须支持接受字符串数据的 write 方法。

如果可选的 *mangle_from_* 为 True，则会将一个 > 字符放到消息体中恰好以字符串 "From " 打头，即开头文本为 From 加一个空格的任何行的前面。*mangle_from_* 默认为 *policy* 的*mangle_from_* 设置值 (对于*compat32* 策略为 True 而对于所有其他策略则为 False)。*mangle_from_* 被设计为在当消息以 Unix mbox 格式存储时使用 (参见*mailbox* 和 WHY THE CONTENT-LENGTH FORMAT IS BAD)。

如果 *maxheaderlen* 不为 None，则重新折叠任何长于 *maxheaderlen* 的标头行，或者如果为 0，则不重新包装任何标头。如果 *manheaderlen* 为 None (默认值)，则根据 *policy* 设置包装标头和其他消息行。

如果指定了 *policy*，则使用该策略来控制消息的生成。如果 *policy* 为 None (默认值)，则使用与传递给 flatten 的*Message* 或*EmailMessage* 对象相关联的策略来控制消息的生成。请参阅*email.policy* 了解有关 *policy* 所控制内容的详情。

在 3.3 版本发生变更: 添加了 *policy* 关键字。

在 3.6 版本发生变更: *mangle_from_* 和 *maxheaderlen* 形参的默认行为是遵循策略。

flatten(*msg*, *unixfrom=False*, *linesep=None*)

将以 *msg* 为根的消息对象结构体的文本表示形式打印到当*Generator* 实例被创建时所指定的输出文件。

如果*policy* 选项*cte_type* 为 8bit，则视同选项被设为 7bit 来生成消息。（这是必需的，因为字符串无法表示非 ASCII 字节数据。）将使用兼容 ASCII 的 *Content-Transfer-Encoding* 按需转换任何具有高比特位组的字节数据。也就是说，将具有非 ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) 的部分转换为兼容 ASCII 的 *Content-Transfer-Encoding*，并使用 MIME unknown-8bit 字符集来编码标头中不符合 RFC 的非 ASCII 字节数据，以使其符合 RFC。

如果 *unixfrom* 为 True，则会在根消息对象的第一个 **RFC 5322** 标头之前打印 Unix mailbox 格式 (参见*mailbox*) 所使用的封包标头分隔符。如果根对象没有封包标头，则会创建一个标准标头。默认值为 False。请注意对于子部分来说，不会打印任何封包标头。

如果 *linesep* 不为 None，则会将其用作扁平化消息的所有行之间的分隔符。如果 *linesep* 为 None (默认值)，则使用在 *policy* 中指定的值。

在 3.2 版本发生变更: 添加了对重编码 8bit 消息体的支持，以及 *linesep* 参数。

clone(*fp*)

返回此*Generator* 实例的独立克隆，具有完全相同的选项设置，而 *fp* 为新的 *outfp*。

write(*s*)

将 *s* 写入到传给*Generator* 的构造器的 *outfp* 的 *write* 方法。这足够为*Generator* 实际提供可用于*print()* 函数的文件类 API。

作为一个便捷工具，*EmailMessage* 提供了*as_string()* 和 str(aMessage) (即*__str__()*) 等方法，它们简单地生成一个消息对象的已格式化字符串表示形式。更多细节请参阅*email.message*。

email.generator 模块还提供了一个派生类*DecodedGenerator*，它类似于*Generator* 基类，不同之处在于非 *text* 部分不会被序列化，而是被表示为基于模板并填写了有关该部分的信息的字符串输出流的形式。

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, *, *policy=None*)

行为类似于*Generator*，不同之处在于对传给*Generator.flatten()* 的消息的任何子部分，如果该子部分的主类型为 *text*，则打印该子部分的已解码载荷，而如果其主类型不为 *text*，则不直接打印它而是使用来自该部分的信息填入字符串 *fmt* 并将填写完成的字符串打印出来。

要填入 *fmt*，则执行 fmt % part_info，其中 part_info 是由下列键和值组成的字典:

- type -- 非 *text* 部分的完整 MIME 类型
- maintype -- 非 *text* 部分的主 MIME 类型

- `subtype` -- 非 *text* 部分的子 MIME 类型
- `filename` -- 非 *text* 部分的文件名
- `description` -- 与非 *text* 部分相关联的描述
- `encoding` -- 非 *text* 部分的内容转换编码格式

如果 *fmt* 为 `None`，则使用下列默认 *fmt*:

"[忽略消息的非文本 (%(type)s) 部分，文件名%(filename)s]"

可选的 *_mangle_from_* 和 *maxheaderlen* 与 *Generator* 基类的相同。

备注

20.1.4 email.policy: 策略对象

Added in version 3.3.

源代码: Lib/email/policy.py

---

email 的主要焦点是按照各种电子邮件和 MIME RFC 的描述来处理电子邮件消息。但是电子邮件消息的基本格式（一个由名称加冒号加值的标头字段构成的区块，后面再加一个空白行和任意的‘消息体’）是在电子邮件领域以外也获得应用的格式。这些应用的规则有些与主要电子邮件 RFC 十分接近，有些则很不相同。即使是操作电子邮件，有时也可能需要打破严格的 RFC 规则，例如生成可与某些并不遵循标准的电子邮件服务器互联的电子邮件，或者是实现希望应用某些破坏标准的操作方式的扩展。

Policy 对象给予 email 包处理这些不同用例的灵活性。

Policy 对象封装了一组属性和方法用来在使用期间控制 email 包中各个组件的行为。*Policy* 实例可以被传给 email 包中的多个类和方法以更改它们的默认行为。可设置的值及其默认值如下所述。

在 email 包中的所有类会使用一个默认的策略。对于所有*parser* 类及相关的便捷函数，还有对于*Message* 类来说，它是*Compat32* 策略，通过其对应的预定义实例*compat32* 来使用。这个策略提供了与 Python3.3 版之前的 email 包的完全向下兼容性（在某些情况下，也包括对缺陷的兼容性）。

传给*EmailMessage* 的 *policy* 关键字的默认值是*EmailPolicy* 策略，表示为其预定义的实例*default*。

在创建*Message* 或*EmailMessage* 对象时，它需要一个策略。如果消息是由*parser* 创建的，则传给该解析器的策略将是它所创建的消息所使用的策略。如果消息是由程序创建的，则该策略可以在创建它的时候指定。当消息被传递给*generator* 时，生成器默认会使用来自该消息的策略，但你也可以将指定的策略传递给生成器，这将覆盖存储在消息对象上的策略。

email.parser 类和解析器便捷函数的 *policy* 关键字的默认值在未来的 Python 版本中 **将会改变**。因此在调用任何*parser* 模块所描述的类和函数时你应当 **总是显式地指定你想要使用的策略**。

本文档的第一部分介绍了*Policy* 的特性，它是一个*abstract base class*，定义了所有策略对象包括*compat32* 的共有特性。这些特性包括一些由 email 包内部调用的特定钩子方法，自定义策略可以重写这些方法以获得不同行为。第二部分描述了实体类*EmailPolicy* 和*Compat32*，它们分别实现了提供标准行为和向下兼容行为与特性的钩子。

Policy 实例是不可变的，但它们可以被克隆，接受与类构造器一致的关键字参数并返回一个新的*Policy* 实例，新实例是原实例的副本，但具有被改变的指定属性。

例如，以下代码可以被用来从一个 Unix 系统的磁盘文件中读取电子邮件消息并将其传递给系统的 `sendmail` 程序:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
...
```

(续下页)

(接上页)

```
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

这里我们让*BytesGenerator* 在创建要送入 sendmail's stdin 的二进制字串时使用符合 RFC 的行分隔字符，默认的策略将会使用 \n 行分隔符。

某些 email 包的方法接受一个 *policy* 关键字参数，允许为该方法覆盖原有策略。例如，以下代码使用了来自之前示例的 *msg* 对象的*as_bytes()* 方法并使用其运行所在平台的本机行分隔符将消息写入一个文件:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy 对象也可使用加法运算符进行组合来产生一个新策略对象，其设置是被加总对象的非默认值的组合:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

此运算不满足交换律；也就是说对象的添加顺序很重要。见以下演示:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

这是所有策略类的*abstract base class*。它提供了一些简单方法的默认实现，以及不可变特征属性，*clone()* 方法以及构造器语义的实现。

可以向策略类的构造器传入各种关键字参数。可以指定的参数是该类的任何非方法特征属性，以及实体类的任何额外非方法特征属性。在构造器中指定的值将覆盖相应属性的默认值。

这个类定义了下列特征属性，因此下列值可以被传给任何策略类的构造器:

max_line_length

序列化输出中任何行的最大长度，不计入行字符的末尾。默认值为 78，基于 **RFC 5322**。值为 0 或*None* 表示完全没有行包装。

linesep

用来在序列化输出中确定行的字符串。默认值为 \n 因为这是 Python 所使用的内部行结束符规范，但 RFC 的要求是 \r\n。

cte_type

控制可能要求使用的内容传输编码格式类型。可能的值包括:

7bit	所有数据必须为" 纯 7 比特位" (仅 ASCII)。这意味着在必要情况下数据将使用可打印引用形式或 base64 编码格式进行编码。
8bit	数据不会被限制为纯 7 比特位。标头中的数据仍要求仅 ASCII 因此将被编码（参阅下文的*fold_binary()* 和*utf8* 了解例外情况），但消息体部分可能使用 8bit CTE。

cte_type 值为 8bit 仅适用于 BytesGenerator 而非 Generator，因为字符串不能包含二进制数据。如果 Generator 运行于指定了 cte_type=8bit 的策略，它的行为将与 cte_type 为 7bit 相同。

raise_on_defect

如为*True*，则遇到的任何缺陷都将引发错误。如为*False* (默认值)，则缺陷将被传递给*register_defect()* 方法。

mangle_from_

如为*True*，则消息体中以 *"From "* 开头的行会通过在其前面放一个 > 来进行转义。当消息被生成器执行序列化时会使用此形参。默认值 t: *False*。

Added in version 3.5.

message_factory

用来构造新的空消息对象的工厂函数。在构建消息时由解析器使用。默认为 None，在此情况下会使用*Message*。

Added in version 3.6.

verify_generated_headers

如为 True (默认值)，则生成器会引发 raise *HeaderWriteError* 而不是写入一个不正确地折叠或设限的标头，以便它可以被解析为多重标头或与相邻数据合并。这样的标头可通过自定义标头类或 email 模块中的程序错误来生成。

由于它是一个安全特性，即使是在*Compat32* 策略中该值也默认为 True。为了保持向下兼容但是不安全的行为，它必须显式地被设为 False。

Added in version 3.13.

下列*Policy* 方法是由使用 email 库的代码来调用以创建具有自室外设置的策略实例:

clone(***kw*)

返回一个新的*Policy* 实例，其属性与当前实例具有相同的值，除非是那些由关键字参数给出了新值的属性。

其余的*Policy* 方法是由 email 包代码来调用的，而不应当被使用 email 包的应用程序所调用。自定义的策略必须实现所有这些方法。

handle_defect(*obj*, *defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of *MessageDefect*.

默认实现会检查*raise_on_defect* 旗标。如果其为 True，则 *defect* 会被作为异常来引发。如果其为 False (默认值)，则 *obj* 和 *defect* 会被传递给*register_defect()*。

register_defect(*obj*, *defect*)

Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of *MessageDefect*.

默认实现会调用 *obj* 的 defects 属性的 append 方法。当 email 包调用*handle_defect* 时，*obj* 通常将具有一个带 append 方法的 defects 属性。配合 email 包使用的自定义对象类型（例如自定义的 Message 对象）也应当提供这样的属性，否则在被解析消息中的缺陷将引发非预期的错误。

header_max_count(*name*)

返回名为 *name* 的标头的最大允许数量。

当添加一个标头到*EmailMessage* 或*Message* 对象时被调用。如果返回值不为 0 或 None，并且已有的名称为 *name* 的标头数量大于等于所返回的值，则会引发*ValueError*。

由于 Message.__setitem__ 的默认行为是将值添加到标头列表，因此很容易不知情地创建重复的标头。此方法允许在程序中限制可以被添加到 Message 中的特定标头的实例数量。(解析器不会考虑此限制，它将忠实地产生被解析消息中存在的任意数量的标头。)

默认实现对于所有标头名称都返回 None。

header_source_parse(*sourcelines*)

email 包调用此方法时将传入一个字符串列表，其中每个字符串以在被解析源中找到的行分隔符结束。第一行包括字段标头名称和分隔符。源中的所有空白符都会被保留。此方法应当返回 `(name, value)` 元组以保存至 `Message` 中来代表被解析的标头。

如果一个实现希望保持与现有 email 包策略的兼容性，则 *name* 应当为保留大小写形式的名称（所有字符直至':' 分隔符），而 *value* 应当为展开后的值（移除所有行分隔符，但空白符保持不变），并移除开头的空白符。

sourcelines 可以包含经替代转义的二进制数据。

此方法没有默认实现

header_store_parse(*name*, *value*)

当一个应用通过程序代码修改 `Message` (而不是由解析器创建 `Message`) 时，email 包会调用此方法并附带应用程序所提供的名称和值。此方法应当返回 `(name, value)` 元组以保存至 `Message` 中用来表示标头。

如果一个实现希望保持与现有 email 包策略的兼容性，则 *name* 和 *value* 应当为字符串或字符串的子类，它们不会修改在参数中传入的内容。

此方法没有默认实现

header_fetch_parse(*name*, *value*)

当标头被应用程序所请求时，email 包会调用此方法并附带当前保存在 `Message` 中的 *name* 和 *value*，并且无论此方法返回什么它都会被回传给应用程序作为被提取标头的值。请注意可能会有多个相同名称的标头被保存在 `Message` 中；此方法会将指定标头的名称和值返回给应用程序。

value 可能包含经替代转义的二进制数据。此方法所返回的值应当没有经替代转义的二进制数据。

此方法没有默认实现

fold(*name*, *value*)

email 包调用此方法时会附带当前保存在 `Message` 中的给定标头的 *name* 和 *value*。此方法应当返回一个代表该标头的（根据策略设置）通过处理 *name* 和 *value* 并在适当位置插入 *linesep* 字符来正确地“折叠”的字符串。请参阅 **RFC 5322** 了解有关折叠电子邮件标头的规则的讨论。

value 可能包含经替代转义的二进制数据。此方法所返回的字符串应当没有经替代转义的二进制数据。

fold_binary(*name*, *value*)

与 *fold()* 类似，不同之处在于返回的值应当为字节串对象而非字符串。

value 可能包含经替代转义的二进制数据。这些数据可以在被返回的字节串对象中被转换回二进制数据。

class email.policy.**EmailPolicy**(***kw*)

这个实体 *Policy* 提供了完全遵循当前电子邮件 RFC 的行为。这包括 (但不限于) **RFC 5322**, **RFC 2047** 以及当前的各种 MIME RFC。

此策略添加了新的标头解析和折叠算法。标头不是简单的字符串，而是带有依赖于字段类型的属性的 `str` 的子类。这个解析和折叠算法完整实现了 **RFC 2047** 和 **RFC 5322**。

message_factory 属性的默认值为 *EmailMessage*。

除了上面列出的适用于所有策略的可设置属性，此策略还添加了下列额外属性:

Added in version 3.6:[1]

utf8

如为 `False`，则遵循 **RFC 5322**，通过编码为“已编码字”来支持标头中的非 ASCII 字符。如为 `True`，则遵循 **RFC 6532** 并对标头使用 `utf-8` 编码格式。以此方式格式化的消息可被传递给支持 `SMTPUTF8` 扩展 (**RFC 6531**) 的 SMTP 服务器。

[1] 最初在 3.3 中作为暂定特性 添加。

refold_source

如果 Message 对象中标头的值源自*parser*(而非由程序设置)，此属性会表明当将消息转换回序列化形式时是否应当由生成器来重新折叠该值。可能的值如下:

none	所有源值使用原始折叠
long	具有任何长度超过 max_line_length 的行的源值将被折叠
all	所有值会被重新折叠。

默认值为 long。

header_factory

该可调用对象接受两个参数，name 和 value，其中 name 为标头字段名而 value 为展开后的标头字段值，并返回一个表示该标头的字符串子类。已提供的默认 header_factory (参见*headerregistry*) 支持对各种地址和日期 **RFC 5322** 标头字段类型及主要 MIME 标头字段类型的自定义解析。未来还将添加对其他自定义解析的支持。

content_manager

此对象至少有两个方法: get_content 和 set_content。当一个*EmailMessage* 对象的*get_content()* 或*set_content()* 方法被调用时，它会调用此对象的相应方法，将消息对象作为其第一个参数，并将传给它的任何参数或关键字作为附加参数传入。默认情况下 content_manager 会被设为*raw_data_manager*。

Added in version 3.4.

这个类提供了下列对*Policy* 的抽象方法的具体实现:

header_max_count(*name*)

返回用来表示具有给定名称的标头的专用类的*max_count* 属性的值。

header_source_parse(*sourcelines*)

此名称会被作为到':' 止的所有内容来解析。该值是通过从第一行的剩余部分去除前导空格，再将所有后续行连接到一起，并去除所有末尾回车符或换行符来确定的。

header_store_parse(*name*, *value*)

name 将会被原样返回。如果输入值具有 name 属性并可在忽略大小写的情况下匹配 *name*，则 value 也会被原样返回。在其他情况下 *name* 和 *value* 会被传递给 header_factory，并将结果标头对象作为值返回。在此情况下如果输入值包含 CR 或 LF 字符则会引发 ValueError。

header_fetch_parse(*name*, *value*)

如果值具有 name 属性，它会被原样返回。在其他情况下 *name* 和移除了所有 CR 和 LF 字符的 *value* 会被传递给 header_factory，并返回结果标头对象。任何经替代转义的字节串会被转换为 unicode 未知字符字形。

fold(*name*, *value*)

标头折叠是由*refold_source* 策略设置来控制的。当且仅当一个值没有 name 属性 (具有 name 属性就意味着它是某种标头对象) 它才会被当作是 "源值"。如果一个原值需要按照策略来重新折叠，则会通过将 *name* 和去除了所有 CR 和 LF 字符的 *value* 传递给 header_factory 来将其转换为标头对象。标头对象的折叠是通过调用其 fold 方法并附带当前策略来完成的。

源值会使用*splitlines()* 来拆分成多行。如果该值不被重新折叠，则会使用策略中的 linesep 重新合并这些行并将其返回。例外的是包含非 ascii 二进制数据的行。在此情况下无论 refold_source 如何设置该值都会被重新折叠，这会导致二进制数据使用 unknown-8bit 字符集进行 CTE 编码。

fold_binary(*name*, *value*)

如果*cte_type* 为 7bit 则与*fold()* 类似，不同之处在于返回的值是字节串。

如果*cte_type* 为 8bit，则将非 ASCII 二进制数据转换回字节串。带有二进制数据的标头不会被重新折叠，无论 refold_header 设置如何，因为无法知晓该二进制数据是由单字节字符还是多字节字符组成的。

以下*EmailPolicy* 的实例提供了适用于特定应用领域的默认值。请注意在未来这些实例（特别是 HTTP 实例）的行为可能会被调整以便更严格地遵循与其领域相关的 RFC。

email.policy.**default**

一个未改变任何默认值的 EmailPolicy 实例。此策略使用标准的 Python \n 行结束符而非遵循 RFC 的 \r\n。

email.policy.**SMTP**

适用于按照符合电子邮件 RFC 的方式来序列化消息。与 default 类似，但 linesep 被设为遵循 RFC 的 \r\n。

email.policy.**SMTPUTF8**

与 SMTP 类似但是*utf8* 为 True。适用于在不使用标头内已编码字的情况下对消息进行序列化。如果发送方或接收方地址具有非 ASCII 字符则应当只被用于 SMTP 传输 (*smtplib.SMTP.send_message()* 方法会自动如此处理)。

email.policy.**HTTP**

适用于序列化标头以在 HTTP 通信中使用。与 SMTP 类似但是 max_line_length 被设为 None (无限制)。

email.policy.**strict**

便捷实例。与 default 类似但是 raise_on_defect 被设为 True。这样可以允许通过以下写法来严格地设置任何策略:

```
somepolicy + policy.strict
```

因为所有这些*EmailPolicies*，email 包的高效 API 相比 Python 3.2 API 发生了以下几方面变化:

- 在*Message* 中设置标头将使得该标头被解析并创建一个标头对象。
- 从*Message* 提取标头将使得该标头被解析并创建和返回一个标头对象。
- 任何标头对象或任何由于策略设置而被重新折叠的标头都会使用一种完全实现了 RFC 折叠算法的算法来进行折叠，包括知道在休息需要并允许已编码字。

从应用程序的视角来看，这意味着任何通过*EmailMessage* 获得的标头都是具有附加属性的标头对象，其字符串值都是该标头的完全解码后的 unicode 值。类似地，可以使用 unicode 对象为一个标头赋予新的值，或创建一个新的标头对象，并且该策略将负责把该 unicode 字符串转换为正确的 RFC 已编码形式。

标头对象及其属性的描述见*headerregistry*。

class email.policy.**Compat32**(***kw*)

这个实体*Policy* 为向下兼容策略。它复制了 Python 3.2 中 email 包的行为。*policy* 模块还定义了该类的一个实例*compat32*，用来作为默认策略。因此 email 包的默认行为会保持与 Python 3.2 的兼容性。

下列属性具有与*Policy* 默认值不同的值:

mangle_from_

默认值为 True。

这个类提供了下列对*Policy* 的抽象方法的具体实现:

header_source_parse(*sourcelines*)

此名称会被作为到':' 止的所有内容来解析。该值是通过从第一行的剩余部分去除前导空格，再将所有后续行连接到一起，并去除所有末尾回车符或换行符来确定的。

header_store_parse(*name*, *value*)

name 和 value 会被原样返回。

header_fetch_parse(*name*, *value*)

如果 value 包含二进制数据，则会使用 unknown-8bit 字符集来将其转换为*Header* 对象。在其他情况下它会被原样返回。

fold(*name*, *value*)

标头会使用*Header*折叠算法进行折叠，该算法保留 value 中现有的换行，并将每个结果行的长度折叠至 `max_line_length`。非 ASCII 二进制数据会使用 `unknown-8bit` 字符串进行 CTE 编码。

fold_binary(*name*, *value*)

标头会使用*Header*折叠算法进行折叠，该算法保留 value 中现有的换行，并将每个结果行的长度折叠至 `max_line_length`。如果 `cte_type` 为 `7bit`，则非 ascii 二进制数据会使用 `unknown-8bit` 字符集进行 CTE 编码。在其他情况下则会使用原始的源标头，这将保留其现有的换行和所包含的任何（不符合 RFC 的）二进制数据。

`email.policy.`**compat32**

*Compat32*的实例，提供与 Python 3.2 中的 email 包行为的向下兼容性。

备注

20.1.5 `email.errors`: 异常和缺陷类

源代码: Lib/email/errors.py

以下异常类在*email.errors*模块中定义:

exception `email.errors.`**MessageError**

这是*email*包可以引发的所有异常的基类。它源自标准异常*Exception*类，这个类没有定义其他方法。

exception `email.errors.`**MessageParseError**

这是由*Parser*类引发的异常的基类。它派生自*MessageError*。*headerregistry*使用的解析器也在内部使用这个类。

exception `email.errors.`**HeaderParseError**

在解析消息的 **RFC 5322** 标头时，某些错误条件下会触发，此类派生自*MessageParseError*。如果在调用方法时内容类型未知，则*set_boundary()*方法将引发此错误。当尝试创建一个看起来包含嵌入式标头的标头时*Header*可能会针对某些 base64 解码错误引发此错误（也就是说，应该是一个没有前导空格并且看起来像标题的延续行）。

exception `email.errors.`**BoundaryError**

已弃用和不再使用的。

exception `email.errors.`**MultipartConversionError**

Raised 当*attach()*方法是在一个派生自*MIMENonMultipart*的类 (例如*MIMEImage*) 的实例上被调用时会引发该异常。*MultipartConversionError*多重继承自*MessageError*和内置的*TypeError*。

exception `email.errors.`**HeaderWriteError**

当*generator*输出标头发生错误时将被引发。

exception `email.errors.`**MessageDefect**

这是表示在解析邮件消息时出现的所有错误的基类。它派生自*ValueError*。

exception `email.errors.`**HeaderDefect**

这是表示在解析邮件标头时出现的所有错误的基类。它派生自*MessageDefect*。

以下是*FeedParser*在解析消息时可发现的缺陷列表。请注意这些缺陷会在问题被发现时加入到消息中，因此举例来说，如果某条嵌套在*multipart/alternative*中的消息具有错误的标头，该嵌套消息对象就会有一条缺陷，但外层消息对象则没有。

所有缺陷类都是*email.errors.MessageDefect*的子类。

exception `email.errors.`**NoBoundaryInMultipartDefect**

一条消息宣称有多个部分，但却没有*boundary*形参。

exception email.errors.**StartBoundaryNotFoundDefect**

在 *Content-Type* 标头中宣称的开始边界无法找到。

exception email.errors.**CloseBoundaryNotFoundDefect**

找到了开始边界，但找不到对应的结束边界。

Added in version 3.3.

exception email.errors.**FirstHeaderLineIsContinuationDefect**

消息以一个后续行作为其第一个标头行。

exception email.errors.**MisplacedEnvelopeHeaderDefect**

在标头块中间发现了一个"Unix From" 标头。

exception email.errors.**MissingHeaderBodySeparatorDefect**

在解析没有前缀空格但又不包含':' 的标头期间发现一行内容。解析将假定该行代表消息体的第一行以继续执行。

Added in version 3.3.

exception email.errors.**MalformedHeaderDefect**

找到一个缺失了冒号，或者是格式错误的标头。

自 3.3 版本弃用: 此缺陷在近几个 Python 版本中已不再使用。

exception email.errors.**MultipartInvariantViolationDefect**

一条消息宣称为 *multipart*，但找不到任何子部分。请注意当一条消息有此缺陷时，其*is_multipart()* 方法可能返回 `False`，即使其内容类型宣称为 *multipart*。

exception email.errors.**InvalidBase64PaddingDefect**

当解码一个 base64 编码的字节块时，填充的数据不正确。虽然添加了足够的填充数据以执行解码，但作为结果的已解码字节串可能无效。

exception email.errors.**InvalidBase64CharactersDefect**

当解码一个 base64 编码的字节分块时，遇到了 base64 字符表以外的字符。这些字符会被忽略，但作为结果的已解码字节串可能无效。

exception email.errors.**InvalidBase64LengthDefect**

当解码一个 base64 编码的字节分块时，非填充 base64 字符的数量无效（比 4 的倍数多 1）。已编码分块会保持原样。

exception email.errors.**InvalidDateDefect**

当解码一个无效或不可解析的数据字段时引发。原始值会被保持原样。

20.1.6 email.headerregistry: 自定义标头对象

源代码: Lib/email/headerregistry.py

Added in version 3.6:[1]

标头是由*str* 的自定义子类来表示的。用于表示给定标头的特定类则由创建标头时生效的*policy* 的*header_factory* 确定。这一节记录了 email 包为处理兼容 **RFC 5322** 的电子邮件消息所实现的特定 `header_factory`，它不仅为各种标头类型提供了自定义的标头对象，还为应用程序提供了添加其自定义标头类型的扩展机制。

当使用派生自*EmailPolicy* 的任何策略对象时，所有标头都通过*HeaderRegistry* 产生并且以*BaseHeader* 作为其最后一个基类。每个标头类都有一个由该标头类型确定的附加基类。例如，许多标头都以*UnstructuredHeader* 类作为其另一个基类。一个标头专用的第二个类是由标头名称使用存储在*HeaderRegistry* 中的查找表来确定的。所有这些都针对典型应用程序进行透明的管理，但也为修改默认行为提供了接口，以便由更复杂的应用使用。

[1] 最初在 3.3 中作为暂定模块 添加

以下各节首先记录了标头基类及其属性，然后是用于修改*HeaderRegistry* 行为的 API，最后是用于表示从结构化标头解析的数据的支持类。

class email.headerregistry.**BaseHeader**(*name*, *value*)

name 和 *value* 会从*header_factory* 调用传递给 BaseHeader。任何标头对象的字符串值都是完成解码为 unicode 的 *value*。

这个基类定义了下列只读属性:

name

标头的名称（字段在':' 之前的部分）。这就是 *name* 的*header_factory* 调用所传递的值；也就是说会保持大小写形式。

defects

一个包含*HeaderDefect* 实例的元组，这些实例报告了在解析期间发现的任何 RFC 合规性问题。email 包会尝试尽可能地检测合规性问题。请参阅*errors* 模块了解可能被报告的缺陷类型的相关讨论。

max_count

此类型标头可具有相同 name 的最大数量。None 值表示无限制。此属性的 BaseHeader 值为 None；专用的标头类预期将根据需要覆盖这个值。

BaseHeader 还提供了以下方法，它由 email 库代码调用，通常不应当由应用程序来调用。

fold(*, *policy*)

返回一个字符串，其中包含用来根据 *policy* 正确地折叠标头的*linesep* 字符。*cte_type* 为 8bit 时将被作为 7bit 来处理，因为标头不能包含任意二进制数据。如果*utf8* 为 False，则非 ASCII 数据将根据 **RFC 2047** 来编码。

BaseHeader 本身不能被用于创建标头对象。它定义了一个与每个专用标头相配合的协议以便生成标头对象。具体来说，BaseHeader 要求专用类提供一个名为 parse 的*classmethod()*。此方法的调用形式如下:

```
parse(string, kwds)
```

kwds 是包含了一个预初始化键 defects 的字典。defects 是一个空列表。parse 方法应当将任何已检测到的缺陷添加到此列表中。在返回时，kwds 字典 *必须*至少包含 decoded 和 defects 等键的值。decoded 应当是标头的字符串值（即完全解码为 unicode 的标头值）。parse 方法应当假定 *string* 可能包含 content-transfer-encoded 部分，但也应当正确地处理全部有效的 unicode 字符以便它能解析未经编码的标头值。

随后 BaseHeader 的 __new__ 会创建标头实例，并调用其 init 方法。专属类如果想要设置 BaseHeader 自身所提供的属性之外的附加属性，只需提供一个 init 方法。这样的 init 看起来应该是这样:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

也就是说，专属类放入 kwds 字典的任何额外内容都应当被移除和处理，并且 kw (和 args) 的剩余内容会被传递给 BaseHeader init 方法。

class email.headerregistry.**UnstructuredHeader**

" 非结构化" 标头是 **RFC 5322** 中默认的标头类型。任何没有指定语法的标头都会被视为是非结构化的。非结构化标头的经典例子是 *Subject* 标头。

在 **RFC 5322** 中，非结构化标头是指一段以 ASCII 字符集表示的任意文本。但是 **RFC 2047** 具有一个 **RFC 5322** 兼容机制用来将标头值中的非 ASCII 文本编码为 ASCII 字符。当包含已编码字的 *value* 被传递给构造器时，UnstructuredHeader 解析器会按照非结构化文本的 **RFC 2047** 规则将此类已编码字转换为 unicode。解析器会使用启发式机制来尝试解码一些不合规的已编码字。在此种情况下各类缺陷，例如已编码字或未编码文本中的无效字符问题等缺陷将会被注册。

此标头类型未提供附加属性。

class email.headerregistry.**DateHeader**

RFC 5322 为电子邮件标头内的日期指定了非常明确的格式。DateHeader 解析器会识别该日期格式，并且也能识别间或出现的一些“不规范”变种形式。

这个标头类型提供了以下附加属性。

datetime

如果标头值能被识别为某一种有效的日期形式，此属性将包含一个代表该日期的*datetime*实例。如果输入日期的时区被指定为 -0000 (表示它是 UTC 但不包含源时区的相关信息)，则*datetime* 将为简单型*datetime*。如果找到了特定的时区时差值 (包括 +0000)，则*datetime* 将包含一个使用*datetime.timezone* 来记录时区时差的感知型 datetime。

标头的 decoded 值是由按照 RFC 5322 对 datetime 进行格式化来确定的；也就是说，它会被设为:

```
email.utils.format_datetime(self.datetime)
```

当创建 DateHeader 时，*value* 可以为*datetime* 实例。例如这意味着以下代码是有效的并能实现人们预期的行为:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

因为这是个简单型 datetime 它将被解读为 UTC 时间戳，并且结果值的时区将为 -0000。使用来自*utils* 模块的*localtime()* 函数会更有用:

```
msg['Date'] = utils.localtime()
```

这个例子将日期标头设为使用当前时区时差值的当前时间和日期。

class email.headerregistry.**AddressHeader**

地址标头是最复杂的结构化标头类型之一。AddressHeader 类提供了适合任何地址标头的泛用型接口。

这个标头类型提供了以下附加属性。

groups

编码了在标头值中找到的地址和分组的*Group* 对象的元组。非分组成员的地址在此列表中表示为*display_name* 为 None 的单地址 Groups。

addresses

编码了来自标头值的所有单独地址的*Address* 对象的元组。如果标头值包含任何分组，则来自分组的单个地址将包含在该分组出现在值中的点上列出（也就是说，地址列表会被“展平”为一维列表）。

标头的 decoded 值将把所有已编码字解码为 unicode。*idna* 编码的域名也会被解码为 unicode。decoded 值是通过对 groups 属性的元素的*str* 值使用 ', ' 进行合并 来设置的。

可以使用*Address* 与*Group* 对象的任意组合的列表来设置一个地址标头的值。display_name 为 None 的 Group 对象将被解读为单独地址，这允许一个地址列表可以附带通过使用从源标头的 groups 属性获取的列表而保留原分组。

class email.headerregistry.**SingleAddressHeader**

AddressHeader 的子类，添加了一个额外的属性:

address

由标头值编码的单个地址。如果标头值实际上包含一个以上的地址（这在默认*policy* 下将违反 RFC），则访问此属性将导致*ValueError*。

上述类中许多还具有一个 Unique 变体 (例如 UniqueUnstructuredHeader)。其唯一差别是在 Unique 变体中*max_count* 被设为 1。

class email.headerregistry.**MIMEVersionHeader**

实际上 *MIME-Version* 标头只有一个有效的值，即 1.0。为了将来的扩展，这个标头类还支持其他的有效版本号。如果一个版本号是 RFC 2045 的有效值，则标头对象的以下属性将具有不为 None 的值:

version

字符串形式的版本号。任何空格和/或注释都会被移除。

major

整数形式的主版本号

minor

整数形式的次版本号

class email.headerregistry.**ParameterizedMIMEHeader**

MIME 标头都以前缀'Content-' 打头。每个特定标头都具有特定的值，其描述在该标头的类之中。有些也可以接受一个具有通用格式的补充形参形表。这个类被用作所有接受形参的 MIME 标头的基类。

params

一个将形参名映射到形参值的字典。

class email.headerregistry.**ContentTypeHeader**

处理 *Content-Type* 标头的*ParameterizedMIMEHeader* 类。

content_type

maintype/subtype 形式的内容类型字符串。

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**

处理 *Content-Disposition* 标头的*ParameterizedMIMEHeader* 类。

content_disposition

inline 和 attachment 是仅有的常用有效值。

class email.headerregistry.**ContentTransferEncoding**

处理 *Content-Transfer-Encoding* 标头。

cte

可用的有效值为 7bit, 8bit, base64 和 quoted-printable。更多信息请参阅 **RFC 2045**。

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)

这是由*EmailPolicy* 在默认情况下使用的工厂函数。HeaderRegistry 会使用 *base_class* 和从它所保存的注册表中获取的专用类来构建用于动态地创建标头实例的类。当给定的标头名称未在注册表中出现时，则会使用由 *default_class* 所指定的类作为专用类。当 *use_default_map* 为 True (默认值) 时，则会在初始化期间把从标头名称到类的标准映射拷贝到注册表中。*base_class* 始终会是所生成的类的 __bases__ 列表中的最后一个类。

默认的映射有:

subject
UniqueUnstructuredHeader

date
UniqueDateHeader

resent-date
DateHeader

orig-date
UniqueDateHeader

sender
UniqueSingleAddressHeader

resent-sender
SingleAddressHeader

到
UniqueAddressHeader

resent-to
AddressHeader

cc
UniqueAddressHeader

resent-cc
AddressHeader

bcc
UniqueAddressHeader

resent-bcc
AddressHeader

从
UniqueAddressHeader

resent-from
AddressHeader

reply-to
UniqueAddressHeader

mime-version
MIMEVersionHeader

content-type
ContentTypeHeader

content-disposition
ContentDispositionHeader

content-transfer-encoding
ContentTransferEncodingHeader

message-id
MessageIDHeader

`HeaderRegistry` 具有下列方法:

map_to_type(*self*, *name*, *cls*)

name 是要映射的标头名称。它将在注册表中被转换为小写形式。*cls* 是要与 *base_class* 一起被用来创建用于实例化与 *name* 相匹配的标头的类的专用类。

__getitem__(*name*)

构造并返回一个类来处理 *name* 标头的创建。

__call__(*name*, *value*)

从注册表获得与 *name* 相关联的专用标头 (如果 *name* 未在注册表中出现则使用 *default_class*) 并将其与 *base_class* 相组合以产生类，调用被构造类的构造器，传入相同的参数列表，并最终返回由此创建的类实例。

以下的类是用于表示从结构化标头解析的数据的类，并且通常会由应用程序使用以构造结构化的值并赋给特定的标头。

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

用于表示电子邮件地址的类。地址的一般形式为:

```
[display_name] <username@domain>
```

或者：

```
username@domain
```

其中每个部分都必须符合在 **RFC 5322** 中阐述的特定语法规则。

为了方便起见可以指定 *addr_spec* 来替代 *username* 和 *domain*，在此情况下 *username* 和 *domain* 将从 *addr_spec* 中解析。*addr_spec* 应当是一个正确地引用了 RFC 的字符串；如果它不是 `Address` 则将引发错误。Unicode 字符也允许使用并将在序列化时被正确地编码。但是，根据 RFC，地址的 username 部分 *不允许* 有 unicode。

display_name

地址的显示名称部分（如果有的话）并去除所有引用项。如果地址没有显示名称，则此属性将为空字符串。

username

地址的 `username` 部分，去除所有引用项。

domain

地址的 `domain` 部分。

addr_spec

地址的 `username@domain` 部分，经过正确引用处理以作为纯地址使用（上面显示的第二种形式）。此属性不可变。

__str__()

对象的 `str` 值是根据 **RFC 5322** 规则进行引用处理的地址，但不带任何非 ASCII 字符的 Content Transfer Encoding。

为了支持 SMTP (**RFC 5321**)，`Address` 会处理一种特殊情况：如果 `username` 和 `domain` 均为空字符串 (或为 `None`)，则 `Address` 的字符串值为 `<>`。

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)

用于表示地址组的类。地址组的一般形式为：

```
display_name: [address-list];
```

作为处理由组和单个地址混合构成的列表的便捷方式，`Group` 也可以通过将 *display_name* 设为 `None` 以用来表示不是某个组的一部分的单独地址并提供单独地址的列表作为 *addresses*。

display_name

组的 `display_name`。如果其为 `None` 并且恰好有一个 `Address` 在 `addresses` 中，则 `Group` 表示一个不在某个组中的单独地址。

addresses

一个可能为空的表示组中地址的包含*Address* 对象的元组。

__str__()

`Group` 的 `str` 值会根据 **RFC 5322** 进行格式化，但不带任何非 ASCII 字符的 Content Transfer Encoding。如果 `display_name` 为空值且只有一个单独 `Address` 在 `addresses` 列表中，则 `str` 值将与该单独 `Address` 的 `str` 相同。

备注

20.1.7 email.contentmanager: 管理 MIME 内容

源代码: Lib/email/contentmanager.py

Added in version 3.6:[1]

[1] 最初在 3.4 中作为暂定模块 添加

class email.contentmanager.**ContentManager**

内容管理器的基类。提供注册 MIME 内容和其他表示形式间转换器的标准注册机制，以及 get_content 和 set_content 发送方法。

get_content(*msg*, **args*, ***kw*)

基于 *msg* 的 mimetype 查找处理函数（参见下一段），调用该函数，传递所有参数，并返回调用的结果。预期的效果是处理程序将从 *msg* 中提取有效载荷并返回编码了有关被提取数据信息的对象。

要找到处理程序，将在注册表中查找以下键，找到第一个键即停止:

- 表示完整 MIME 类型的字符串 (maintype/subtype)
- 表示 maintype 的字符串
- 空字符串

如果这些键都没有产生处理程序，则为完整 MIME 类型引发一个*KeyError*。

set_content(*msg*, *obj*, **args*, ***kw*)

如果 maintype 为 multipart，则引发*TypeError*；否则基于 *obj* 的类型（参见下一段）查找处理函数，在 *msg* 上调用*clear_content()*，并调用处理函数，传递所有参数。预期的效果是处理程序将转换 *obj* 并存入 *msg*，并可能对 *msg* 进行其他更改，例如添加各种 MIME 标头来编码需要用来解释所存储数据的信息。

要找到处理程序，将获取 *obj* 的类型 (typ = type(obj))，并在注册表中查找以下键，找到第一个键即停止:

- 类型本身 (typ)
- 类型的完整限定名称 (typ.__module__ + '.' + typ.__qualname__)。
- 类型的 qualname (typ.__qualname__)
- 类型的 name (typ.__name__)。

如果未匹配到上述任何一项，则为*MRO* (typ.__mro__) 中的每个类型重复上述所有检测。最后，如果没有其他键产生处理器，则为 None 键检测处理器。如果没有 None 的处理器，则为该类型的完整限定名称引发*KeyError*。

并会添加一个 *MIME-Version* 标头，如果没有的话 (另请参见*MIMEPart*)。

add_get_handler(*key*, *handler*)

将 *handler* 函数记录为 *key* 的处理程序。对于可能的 *key* 键，请参阅*get_content()*。

add_set_handler(*typekey*, *handler*)

将 *handler* 记录为当一个匹配 *typekey* 的类型对象被传递给*set_content()* 时所要调用的函数。对于可能的 *typekey* 值，请参阅*set_content()*。

内容管理器实例

目前 email 包只提供了一个实体内容管理器*raw_data_manager*，不过在未来可能会添加更多。*raw_data_manager* 是由*EmailPolicy* 及其衍生工具所提供的*content_manager*。

email.contentmanager.**raw_data_manager**

这个内容管理器仅提供了超出*Message* 本身提供内容的最小接口：它只处理文本、原始字节串和*Message* 对象。不过相比基础 API，它具有显著的优势：在文本部分上执行 get_content 将返回一个 unicode 字符串而无需由应用程序来手动解码，set_content 为控制添加到一个部分的标头和控制内容传输编码格式提供了丰富的选项集合，并且它还启用了多种 add_ 方法，从而简化了多部分消息的创建过程。

email.contentmanager.**get_content**(*msg*, *errors='replace'*)

将指定部分的有效载荷作为字符串（对于 text 部分），*EmailMessage* 对象（对于 message/rfc822 部分）或 bytes 对象（对于所有其他非多部分类型）返回。如果是在 multipart 上调用则会引发*KeyError*。如果指定部分是一个 text 部分并且指明了 *errors*，则会在将载荷解码为 unicode 时将其用作错误处理程序。默认的错误处理程序是 replace。

email.contentmanager.**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

向 *msg* 添加标头和有效载荷:

添加一个带有 maintype/subtype 值的 *Content-Type* 标头。

- 对于 str，将 MIME maintype 设为 text，如果指定了子类型 *subtype* 则设为指定值，否则设为 plain。
- 对于 bytes，将使用指定的 *maintype* 和 *subtype*，如果未指定则会引发 *TypeError*。
- 对于 *EmailMessage* 对象，将 maintype 设为 message，并将指定的 subtype 设为 *subtype*，如果未指定则设为 rfc822。如果 *subtype* 为 partial，则引发一个错误（必须使用 bytes 对象来构造 message/partial 部分）。

如果提供了 *charset* (这只对 str 适用)，则使用指定的字符集将字符串编码为字节串。默认值为 utf-8。如果指定的 *charset* 是某个标准 MIME 字符集名称的已知别名，则会改用该标准字符集。

如果设置了 *cte*，则使用指定的内容传输编码格式对有效载荷进行编码，并将 *Content-Transfer-Encoding* 标头设为该值。可能的 *cte* 值有 quoted-printable, base64, 7bit, 8bit 和 binary。如果输入无法以指定的编码格式被编码 (例如，对于包含非 ASCII 值的输入指定 *cte* 值为 7bit)，则会引发 *ValueError*。

- 对于 str 对象，如果未设置 *cte* 则使用启发式求解来确定最紧凑的编码格式。在编码之前，将使用 *str.splitlines()* 来正规化所有行边界，确保载荷的每一行是以当前策略的 *linesep* 特性属性来结束的（即使原始字符串不是以它结束）。
- 对于 bytes 对象，如果未设置 *cte* 则使用 base64，并且不会执行之前提及的换行符转写。
- 对于 *EmailMessage*，按照 **RFC 2046**，如果为 *subtype* rfc822 请求的 *cte* 为 quoted-printable 或 base64，而为 7bit 以外的任何 *cte* 为 *subtype* external-body 则会引发一个错误。对于 message/rfc822，如果 *cte* 未指定则会使用 8bit。对于所有其他 *subtype* 值，都会使用 7bit。

> **备注**
>
> *cte* 值为 binary 实际上还不能正确工作。由 set_content 所修改的 EmailMessage 对象是正确的，但 *BytesGenerator* 不会正确地将其序列化。

如果设置了 *disposition*，它会被用作 *Content-Disposition* 标头的值。如果未设置，并且指定了 *filename*，则添加值为 attachment 的标头。如果未设置 *disposition* 并且也未指定 *filename*，则不添加标头。*disposition* 的有效值仅有 attachment 和 inline。

如果设置了 *filename*，则将其用作 *Content-Disposition* 标头的 filename 参数的值。

如果设置了 *cid*，则添加一个 *Content-ID* 标头并将其值设为 *cid*。

如果设置了 *params*，则迭代其 items 方法并使用输出的 (key, value) 结果对在 *Content-Type* 标头上设置附加参数。

如果设置了 *headers* 并且为 headername: headervalue 形式的字符串的列表或为 header 对象的列表（通过一个 name 属性与字符串相区分），则将标头添加到 *msg*。

备注

20.1.8 email: 示例

以下是一些如何使用*email* 包来读取、写入和发送简单电子邮件以及更复杂的 MIME 邮件的示例。

首先，让我们看看如何创建和发送简单的文本消息（文本内容和地址都可能包含 unicode 字符）：

```
# 导入 smtplib 以使用实际的发送函数
import smtplib

# 导入我们需要的 email 模块
from email.message import EmailMessage

# 打开 textfile 中相应名称的纯文本文件供读取。
with open(textfile) as fp:
    # 创建纯文本消息
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == 发送方 email 地址
# you == 接收方 email 地址
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# 通过我们使用的 SMTP 服务器发送消息。
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

解析 **RFC 822** 标题可以通过使用*parser* 模块中的类来轻松完成：

```
# 导入我们需要的 email 模块
#from email.parser import BytesParser
from email.parser import Parser
from email.policy import default

# 如果 email 标头保存在文件中，则取消注释这两行：
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  或者如果要从字符串中解析标头（这是不太常见的操作），使用：
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  现在标头条目将可作为字典访问：
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# 你也可以访问地址的各个部分：
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

以下是如何发送包含可能在目录中的一系列家庭照片的 MIME 消息示例：

```
# 导入 smtplib 以使用实际的发送函数。
import smtplib
```

(续下页)

(接上页)

```
# 以下是我们要用到的 email 包模块。
from email.message import EmailMessage

# 创建容器 email 消息。
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == 发送方 email 地址
# family = 所有接收方的 email 地址列表
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# 以二进制模式打开文件。 你也可以忽略子类型
# 如果你想要 MIMEImage 自动猜测的话。
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype='png')

# 通过我们自己的 SMTP 服务器发送 email。
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

以下是如何将目录的全部内容作为电子邮件消息发送的示例:[1]

```
#!/usr/bin/env python3

"""将目录的内容作为 MIME 消息来发送。"""

import os
import smtplib
# 用于根据文件扩展名来猜测 MIME 类型
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
```

(续下页)

[1] 感谢 Matthew Dixon Cowles 提供最初的灵感和示例。

(接上页)

```
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # 创建消息
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # 根据文件扩展名来猜测内容类型。
        # 编码格式将被忽略，不过我们应当检查某些简单事务
        # 例如是否为 gzip 或压缩文件。
        ctype, encoding = mimetypes.guess_file_type(path)
        if ctype is None or encoding is not None:
            # 不可以猜测，或者文件已被编码（压缩），
            # 因此我们使用基本的比特位数据类型。
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # 现在执行消息发送或存储
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

以下是如何将上述 MIME 消息解压缩到文件目录中的示例：

```
#!/usr/bin/env python3

"""将 MIME 消息解包到一个文件目录中。"""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
```

(续下页)

(接上页)

```
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* 只是一些容器
        if part.get_content_maintype() == 'multipart':
            continue
        # 应用程序真的应该对所给文件名做无害化处理
        # 以保证 email 消息不能被用来覆盖重要的文件
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

以下是如何使用备用纯文本版本创建 HTML 消息的示例。为了让事情变得更有趣，我们在 html 部分中包含了一个相关的图像，我们保存了一份我们要发送的内容到硬盘中，然后发送它。

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# 创建基本的文本消息。
msg = EmailMessage()
msg['Subject'] = "Pourquoi pas des asperges pour ce midi ?"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cette recette [1] sera sûrement un très bon repas.
```

(续下页)

(接上页)

```
[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# 增加 html 版本。 这会将消息转换为一个 multipart/alternative 容器，
# 原始文本消息作为第一部分而新的 html 消息作为第二部分。
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cette
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recette
        </a> sera sûrement un très bon repas.
    </p>
    <img src="cid:{asparagus_cid}">
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# 请注意我们需要将 <> 从 msgid 中去掉以便在 html 中使用。

# 现在添加相关图像到 html 部分中。
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# 创建我们将要发送的内容的本地副本。
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# 通过本地 SMTP 服务器发送消息。
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

如果我们发送最后一个示例中的消息，这是我们可以处理它的一种方法：

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# 导入我们需要的 email 模块
from email import policy
from email.parser import BytesParser


def magic_html_parser(html_text, partfiles):
    """返回链接到 partfiles 的经安全性处理的 html。

    重写 href="cid:...." 属性以指向 partfiles 中的文件名。
    虽然并非琐碎，这应可使用 html.parser 来实现。
    """
    raise NotImplementedError("Add the magic needed")


# 在真正的程序中你将从参数中获得文件名。
```

(续下页)

(接上页)

```
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# 现在可通过字典形式访问标头条目，并且任何非 ASCII 内容
# 都将被转换为 unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# 如果我们想要打印消息内容的预览，可以提取
# 未经格式化的载荷并打印其中前三行。 当然，
# 如果消息没有纯文本部分则打印 html 的前三行
# 可能是无用的，但这只是个概念性的示例。
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# 我们可以提取最丰富的替代项用于显示:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # 再次去除 <> 以将 cid 的 email 形式转为 html 形式。
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# 当然，许多 email 消息都有可能破坏这个简单的程序，
# 但它将能处理最普通的消息。
```

直到输出提示，上面的输出是：

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Pourquoi pas des asperges pour ce midi ?

Salut!

Cette recette [1] sera sûrement un très bon repas.
```

备注

旧式 API:

20.1.9 email.message.Message: 使用 compat32 API 来表示电子邮件消息

Message 类与*EmailMessage* 类非常相似，但没有该类所添加的方法，并且某些方法的默认行为也略有不同。我们还在这里记录了一些虽然被*EmailMessage* 类所支持但并不推荐的方法，除非你是在处理旧有代码。

在其他情况下这两个类的理念和结构都是相同的。

本文档描述了默认 (对于*Message*) 策略*Compat32* 之下的行为。如果你要使用其他策略，你应当改用*EmailMessage* 类。

电子邮件消息由多个 *标头* 和一个 *载荷* 组成。标头必须为 **RFC 5322** 风格的名称和值，其中字典名和值由冒号分隔。冒号不是字段名或字段值的组成部分。载荷可以是简单的文本消息，或是二进制对象，或是多个子消息的结构化序列，每个子消息都有自己的标头集合和自己的载荷。后一种类型的载荷是由具有 *multipart/** 或 *message/rfc822* 等 MIME 类型的消息来指明的。

Message 对象所提供了概念化模型是由标头组成的有序字典，加上用于访问标头中的特殊信息以及访问载荷的额外方法，以便能生成消息的序列化版本，并递归地遍历对象树。请注意重复的标头是受支持的，但必须使用特殊的方法来访问它们。

Message 伪字典以标头名作为索引，标头名必须为 ASCII 值。字典的值为应当只包含 ASCII 字符的字符串；对于非 ASCII 输入有一些特殊处理，但这并不总能产生正确的结果。标头以保留原大小写的形式存储和返回，但字段名称匹配对大小写不敏感。还可能会有一个单独的封包标头，也称 *Unix-From* 标头或 `From_` 标头。*载荷* 对于简单消息对象的情况是一个字符串或字节串，对于 MIME 容器文档的情况 (例如 *multipart/** 和 *message/rfc822*) 则是一个*Message* 对象。

以下是*Message* 类的方法:

class email.message.**Message**(*policy=compat32*)

如果指定了 *policy* (它必须为*policy* 类的实例) 则使用它所设置的规则来更新和序列化消息的表示形式。如果未设置 *policy*，则使用*compat32* 策略，该策略会保持对 Python 3.2 版 email 包的向下兼容性。更多信息请参阅*policy* 文档。

在 3.3 版本发生变更: 增加了 *policy* 关键字参数。

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

以展平的字符串形式返回整个消息对象。或可选的 *unixfrom* 为真值，返回的字符串会包括封包标头。*unixfrom* 的默认值是 `False`。出于保持向下兼容性的原因，*maxheaderlen* 的默认值是 `0`，因此如果你想要不同的值你必须显式地重写它（在策略中为 *max_line_length* 指定的值将被此方法忽略）。*policy* 参数可被用于覆盖从消息实例获取的默认策略。这可以用来对该方法所输出的格式进行一些控制，因为指定的 *policy* 将被传递给 `Generator`。

如果需要填充默认值以完成对字符串的转换则展平消息可能触发对*Message* 的修改（例如，MIME 边界可能会被生成或被修改）。

请注意此方法是出于便捷原因提供的，并可能无法总是以你想要的方式来格式化消息。例如，在默认情况下它不会按 Unix mbox 格式的要求对以 `From` 打头的行执行调整。为了获得更高灵活性，请实例化一个*Generator* 实例并直接使用其*flatten()* 方法。例如:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

如果消息对象包含未按照 RFC 标准进行编码的二进制数据，则这些不合规数据将被 unicode "unknown character" 码位值所替代。(另请参阅*as_bytes()* 和*BytesGenerator*。)

在 3.4 版本发生变更: 增加了 *policy* 关键字参数。

__str__()

等价于*as_string()*。让 `str(msg)` 产生一个包含已格式化消息的字符串。

as_bytes(*unixfrom=False*, *policy=None*)

以字节串对象的形式返回整个扁平化后的消息。当可选的 *unixfrom* 为真值时，返回的字符串会包括封包标头。*unixfrom* 的默认值为 `False`。*policy* 参数可被用于覆盖从消息实例获取的默认策略。这可被用来控制该方法所产生的部分格式化效果，因为指定的 *policy* 将被传递给 `BytesGenerator`。

如果需要填充默认值以完成对字符串的转换则展平消息可能触发对*Message* 的修改（例如，MIME 边界可能会被生成或被修改)。

请注意此方法是出于便捷原因提供的，并可能无法总是以你想要的方式来格式化消息。例如，在默认情况下它不会按 Unix mbox 格式的要求对以 `From` 打头的行执行调整。为了获得更高灵活性，请实例化一个*BytesGenerator* 实例并直接使用其*flatten()* 方法。例如:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Added in version 3.4.

__bytes__()

等价于*as_bytes()*。让 `bytes(msg)` 产生一个包含已格式化消息的字节串对象。

Added in version 3.4.

is_multipart()

如果该消息的载荷是一个子*Message* 对象列表则返回 `True`，否则返回 `False`。当*is_multipart()* 返回 `False` 时，载荷应当是一个字符串对象（有可能是一个 CTE 编码的二进制载荷)。(请注意*is_multipart()* 返回 `True` 并不意味着"msg.get_content_maintype() == 'multipart'" 将返回 `True`。例如，`is_multipart` 在*Message* 类型为 `message/rfc822` 时也将返回 `True`。)

set_unixfrom(*unixfrom*)

将消息的封包标头设为 *unixfrom*，这应当是一个字符串。

get_unixfrom()

返回消息的信封头。如果信封头从未被设置过，默认返回 `None` 。

attach(*payload*)

将给定的 *payload* 添加到当前载荷中，当前载荷在该调用之前必须为 `None` 或是一个*Message* 对象列表。在调用之后，此载荷将总是一个*Message* 对象列表。如果你想将此载荷设为一个标量对象（如字符串)，请改用*set_payload()*。

这是一个过时的方法。在 `EmailMessage` 类上它的功能已被*set_content()* 及相应的 `make` 和 `add` 方法所替代。

get_payload(*i=None*, *decode=False*)

返回当前的载荷，它在*is_multipart()* 为 True 时将是一个*Message* 对象列表，在*is_multipart()* 为 False 时则是一个字符串。如果该载荷是一个列表且你修改了这个列表对象，那么你就是原地修改了消息的载荷。

传入可选参数 *i* 时，如果*is_multipart()* 为 True，*get_payload()* 将返回载荷从零开始计数的第 *i* 个元素。如果 *i* 小于 0 或大于等于载荷中的条目数则将引发*IndexError*。如果载荷是一个字符串 (即*is_multipart()* 为 False) 且给出了 *i*，则会引发*TypeError*。

可选的 *decode* 是一个指明载荷是否应根据 *Content-Transfer-Encoding* 标头被解码的旗标。当其值为 True 且消息没有多个部分时，如果此标头值为 quoted-printable 或 base64 则载荷将被解码。如果使用了其他编码格式，或者找不到 *Content-Transfer-Encoding* 标头时，载荷将被原样返回（不编码）。在所有情况下返回值都是二进制数据。如果消息有多个部分且 *decode* 旗标为 True，则将返回 None。如果载荷为 base64 但内容不完全正确（如缺少填充符、存在 base64 字母表以外的字符等），则将在消息的缺陷属性中添加适当的缺陷值 (分别为*InvalidBase64PaddingDefect* 或*InvalidBase64CharactersDefect*)。

当 *decode* 为 False (默认值) 时消息体会作为字符串返回而不解码 *Content-Transfer-Encoding*。但是，对于 *Content-Transfer-Encoding* 为 8bit 的情况，会尝试使用 *Content-Type* 标头指定的 charset 来解码原始字节串，并使用 replace 错误处理程序。如果未指定 charset，或者如果指定的 charset 未被 email 包所识别，则会使用默认的 ASCII 字符集来解码消息体。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*get_content()* 和*iter_parts()* 方法所替代。

set_payload(*payload*, *charset=None*)

将整个消息对象的载荷设为 *payload*。客户端要负责确保载荷的不变性。可选的 *charset* 用于设置消息的默认字符集；详情请参阅*set_charset()*。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*set_content()* 方法所替代。

set_charset(*charset*)

将载荷的字符集设为 *charset*，它可以是*Charset* 实例 (参见*email.charset*)、字符集名称字符串或 None。如果是字符串，它将被转换为一个*Charset* 实例。如果 *charset* 是 None，charset 形参将从 *Content-Type* 标头中被删除（消息将不会进行其他修改）。任何其他值都将导致*TypeError*。

如果 *MIME-Version* 标头不存在则将被添加。如果 *Content-Type* 标头不存在，则将添加一个值为 *text/plain* 的该标头。无论 *Content-Type* 标头是否已存在，其 charset 形参都将被设为 *charset.output_charset*。如果 *charset.input_charset* 和 *charset.output_charset* 不同，则载荷将被重编码为 *output_charset*。如果 *Content-Transfer-Encoding* 标头不存在，则载荷将在必要时使用指定的*Charset* 来转换编码，并将添加一个具有相应值的标头。如果 *Content-Transfer-Encoding* 标头已存在，则会假定载荷已使用该 *Content-Transfer-Encoding* 进行正确编码并不会再被修改。

这是一个过时的方法。在 EmailMessage 类上它的功能已被 email.emailmessage.EmailMessage.set_content() 方法的 *charset* 形参所替代。

get_charset()

返回与消息的载荷相关联的*Charset* 实例。

这是一个过时的方法。在 EmailMessage 类上它将总是返回 None。

以下方法实现了用于访问消息的 **RFC 2822** 标头的类映射接口。请注意这些方法和普通映射（例如字典）接口之间存在一些语义上的不同。举例来说，在一个字典中不能有重复的键，但消息标头则可能有重复。并且，在字典中由*keys()* 返回的键的顺序是没有保证的，但在*Message* 对象中，标头总是会按它们在原始消息中的出现或后继加入顺序返回。任何已删除再重新加入的标头总是会添加到标头列表的末尾。

这些语义上的差异是有意为之且其目的是为了提供最大的便利性。

请注意在任何情况下，消息当中的任何封包标头都不会包含在映射接口当中。

在由字符串生成的模型中，任何包含非 ASCII 字节数据（违反 RFC）的标头值在通过此接口来获取时，将被表示为使用 unknown-8bit 字符集的*Header* 对象。

__len__()

返回标头的总数，包括重复项。

__contains__(*name*)

如果消息对象中有一个名为 *name* 的字段则返回 True。匹配操作对大小写不敏感并且 *name* 不应包括末尾的冒号。用于 in 运算符，例如:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

返回指定名称标头字段的值。*name* 不应包括作为字段分隔符的冒号。如果标头未找到，则返回 None；*KeyError* 永远不会被引发。

请注意如果指定名称的字段在消息标头中多次出现，具体将返回哪个字段值是未定义的。请使用*get_all()* 方法来获取所有指定名称标头的值。

__setitem__(*name*, *val*)

将具有字段名 *name* 和值 *val* 的标头添加到消息中。字段会被添加到消息的现有字段的末尾。

请注意，这个方法 既不会覆盖 也不会删除任何字段名重名的已有字段。如果你确实想保证新字段是整个信息头当中唯一拥有 *name* 字段名的字段，你需要先把旧字段删除。例如:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)

删除信息头当中字段名匹配 *name* 的所有字段。如果匹配指定名称的字段没有找到，也不会抛出任何异常。

keys()

以列表形式返回消息头中所有的字段名。

values()

以列表形式返回消息头中所有的字段值。

items()

以二元元组的列表形式返回消息头中所有的字段名和字段值。

get(*name*, *failobj=None*)

返回对应标头字段名的值。这个方法与 __getitem__() 是一样的，只是如果对应标头不存在则返回可选的 *failobj* (默认为 None)。

以下是一些有用的附加方法:

get_all(*name*, *failobj=None*)

返回字段名为 *name* 的所有字段值的列表。如果信息内不存在匹配的字段，返回 *failobj* （其默认值为 None ）。

add_header(*_name*, *_value*, ***_params*)

高级头字段设定。这个方法与*__setitem__()* 类似，不过你可以使用关键字参数为字段提供附加参数。*_name* 是字段名，*_value* 是字段 主值。

对于关键字参数字典 *_params* 中的每一项，其键会被当作形参名，并执行下划线和连字符间的转换（因为连字符不是合法的 Python 标识符）。通常，形参将以 key="value" 的形式添加，除非值为 None，在这种情况下将只添加键。如果值包含非 ASCII 字符，可将其指定为格式为 (CHARSET, LANGUAGE, VALUE) 的三元组，其中 CHARSET 为要用来编码值的字符集名称字符串，LANGUAGE 通常可设为 None 或空字符串（请参阅 **RFC 2231** 了解其他可能的取值），而 VALUE 为包含非 ASCII 码位的字符串值。如果不是传入一个三元组且值包含非 ASCII 字符，则会自动以 **RFC 2231** 格式使用 CHARSET 为 utf-8 和 LANGUAGE 为 None 对其进行编码。

以下是为示例代码:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

会添加一个形如下文的头字段:

```
Content-Disposition: attachment; filename="bud.gif"
```

使用非 ASCII 字符的示例代码:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

它的输出结果为

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)

替换一个标头。将替换在匹配 *_name* 的消息中找到的第一个标头，标头顺序和字段名大小写保持不变。如果未找到匹配的标头，则会引发*KeyError*。

get_content_type()

返回消息的内容类型。返回的字符串会强制转换为 *maintype/subtype* 的全小写形式。如果消息中没有 *Content-Type* 标头则将返回由*get_default_type()* 给出的默认类型。因为根据 **RFC 2045**，消息总是要有一个默认类型，所以*get_content_type()* 将总是返回一个值。

RFC 2045 将消息的默认类型定义为 *text/plain*，除非它是出现在 *multipart/digest* 容器内，在这种情况下其类型应为 *message/rfc822*。如果 *Content-Type* 标头指定了无效的类型，**RFC 2045** 规定其默认类型应为 *text/plain*。

get_content_maintype()

返回信息的主要内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *maintype* 部分。

get_content_subtype()

返回信息的子内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *subtype* 部分。

get_default_type()

返回默认的内容类型。绝大多数的信息，其默认内容类型都是 *text/plain* 。作为 *multipart/digest* 容器内子部分的信息除外，它们的默认内容类型是 *message/rfc822* 。

set_default_type(*ctype*)

设置默认的内容类型。*ctype* 应当为 *text/plain* 或者 *message/rfc822*，尽管这并非强制。默认的内容类型不会存储在 *Content-Type* 标头中。

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)

将消息的 *Content-Type* 形参作为列表返回。所返回列表的元素为以 '=' 号拆分出的键/值对 2 元组。'=' 左侧的为键，右侧的为值。如果形参值中没有 '=' 号，否则该将值如*get_param()* 描述并且在可选 *unquote* 为 True (默认值) 时会被取消转义。

可选的 *failobj* 是在没有 *Content-Type* 标头时要返回的对象。可选的 *header* 是要替代 *Content-Type* 被搜索的标头。

这是一个过时的方法。在 EmailMessage 类上它的功能已被标头访问方法所返回的单独标头对象的 *params* 特征属性所替代。

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)

将 *Content-Type* 标头的形参 *param* 作为字符串返回。如果消息没有 *Content-Type* 标头或者没有这样的形参，则返回 *failobj* (默认为 None)。

如果给出可选的 *header*，它会指定要替代 *Content-Type* 来使用的消息标头。

形参的键总是以大小写不敏感的方式来比较的。返回值可以是一个字符串，或者如果形参以 **RFC 2231** 编码则是一个 3 元组。当为 3 元组时，值中的元素采用 (CHARSET, LANGUAGE, VALUE) 的形式。请注意 CHARSET 和 LANGUAGE 都可以为 None，在此情况下你应当将 VALUE 当作以 us-ascii 字符集来编码。你可以总是忽略 LANGUAGE。

如果你的应用不关心形参是否以 **RFC 2231** 来编码，你可以通过调用 *email.utils.collapse_rfc2231_value()* 来展平形参值，传入来自 *get_param()* 的返回值。当值为元组时这将返回一个经适当编码的 Unicode 字符串，否则返回未经转换的原字符串。例如:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

无论在哪种情况下，形参值（或为返回的字符串，或为 3 元组形式的 VALUE 条目）总是未经转换的，除非 *unquote* 被设为 False。

这是一个过时的方法。在 EmailMessage 类上它的功能已被标头访问方法所返回的单独标头对象的 *params* 特征属性所替代。

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

在 *Content-Type* 标头中设置一个形参。如果该形参已存在于标头中，它的值将被替换为 *value*。如果此消息还未定义 *Content-Type* 标头，它将被设为 *text/plain* 且新的形参值将按 **RFC 2045** 的要求添加。

可选的 *header* 指定一个 *Content-Type* 的替代标头，并且所有形参将根据需要被转换，除非可选的 *requote* 为 False (默认为 True)。

如果指定了可选的 *charset*，形参将按照 **RFC 2231** 来编码。可选的 *language* 指定了 RFC 2231 的语言，默认为空字符串。*charset* 和 *language* 都应为字符串。

如果 *replace* 为 False （默认值），该头字段会被移动到所有头字段列表的末尾。如果 *replace* 为 True ，字段会被原地更新。

在 3.4 版本发生变更: 添加了 replace 关键字。

del_param(*param*, *header='content-type'*, *requote=True*)

从 *Content-Type* 标头中完全移除给定的形参。标头将被原地重写并不带该形参或它的值。所有的值将根据需要被转换，除非 *requote* 为 False (默认为 True)。可选的 *header* 指定 *Content-Type* 的一个替代项。

set_type(*type*, *header='Content-Type'*, *requote=True*)

设置 *Content-Type* 标头的主类型和子类型。*type* 必须为 *maintype/subtype* 形式的字符串，否则会引发 *ValueError*。

此方法可替换 *Content-Type* 标头，并保持所有形参不变。如果 *requote* 为 False，这会保持原有标头引用转换不变，否则形参将被引用转换（默认行为）。

可以在 *header* 参数中指定一个替代标头。当 *Content-Type* 标头被设置时也会添加一个 *MIME-Version* 标头。

这是一个过时的方法。在 EmailMessage 类上它的功能已被 make_ 和 add_ 方法所替代。

get_filename(*failobj=None*)

返回信息头当中 *Content-Disposition* 字段当中名为 filename 的参数值。如果该字段当中没有此参数，该方法会退而寻找 *Content-Type* 字段当中的 name 参数值。如果这个也没有找到，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照 *email.utils.unquote()* 方法去除引号。

get_boundary(*failobj=None*)

返回信息头当中 *Content-Type* 字段当中名为 boundary 的参数值。如果字段当中没有此参数，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照 *email.utils.unquote()* 方法去除引号。

set_boundary(*boundary*)

将 *Content-Type* 头字段的 boundary 参数设置为 *boundary* 。*set_boundary()* 方法永远都会在必要的时候为 *boundary* 添加引号。如果信息对象中没有 *Content-Type* 头字段，抛出*HeaderParseError* 异常。

请注意使用这个方法与删除旧的 *Content-Type* 标头并通过*add_header()* 添加一个带有新边界的新标头有细微的差异，因为*set_boundary()* 会保留 *Content-Type* 标头在原标头列表中的顺序。但是，它 不会保留原 *Content-Type* 标头中可能存在的任何连续的行。

get_content_charset(*failobj=None*)

返回 *Content-Type* 头字段中的 charset 参数，强制小写。如果字段当中没有此参数，或者这个字段压根不存在，返回 *failobj* 。

请注意此方法不同于*get_charset()*，后者会返回*Charset* 实例作为消息体的默认编码格式。

get_charsets(*failobj=None*)

返回一个包含了信息内所有字符集名字的列表。如果信息是 *multipart* 类型的，那么列表当中的每一项都对应其载荷的子部分的字符集名字。否则，该列表是一个长度为 1 的列表。

列表中的每一项都是字符串，它们是其所表示的子部分的 *Content-Type* 标头中 charset 形参的值。但是，如果该子部分没有 *Content-Type* 标头，或没有 charset 形参，或者主 MIME 类型不是 *text*，则所返回列表中的对应项将为 *failobj*。

get_content_disposition()

如果信息的 *Content-Disposition* 头字段存在，返回其字段值；否则返回 None 。返回的值均为小写，不包含参数。如果信息遵循 **RFC 2183** 标准，则此方法的返回值只可能在 *inline* 、*attachment* 和 None 之间选择。

Added in version 3.5.

walk()

walk() 方法是一个多功能生成器。它可以被用来以深度优先顺序遍历信息对象树的所有部分和子部分。一般而言，*walk()* 会被用作 for 循环的迭代器，每一次迭代都返回其下一个子部分。

以下例子会打印出一封具有多部分结构之信息的每个部分的 MIME 类型。

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk 会遍历所有*is_multipart()* 方法返回 True 的部分之子部分，哪怕 msg.get_content_maintype() == 'multipart' 返回的是 False 。使用 _structure 除错帮助函数可以帮助我们在下面这个例子当中看清楚这一点：

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
```

(续下页)

(接上页)

```
text/plain
message/delivery-status
    text/plain
    text/plain
message/rfc822
    text/plain
```

在这里，`message` 的部分并非 `multiparts`，但是它们真的包含子部分！`is_multipart()` 返回 `True`，`walk` 也深入进这些子部分中。

Message 对象也可以包含两个可选的实例属性，它们可被用于生成纯文本的 MIME 消息。

preamble

MIME 文档格式在标头之后的空白行以及第一个多部分的分界字符串之间允许添加一些文本，通常，此文本在支持 MIME 的邮件阅读器中永远不可见，因为它处在标准 MIME 保护范围之外。但是，当查看消息的原始文本，或当在不支持 MIME 的阅读器中查看消息时，此文本会变得可见。

preamble 属性包含 MIME 文档开头部分的这些处于保护范围之外的文本。当*Parser* 在标头之后及第一个分界字符串之前发现一些文本时，它会将这些文本赋值给消息的 *preamble* 属性。当*Generator* 写出 MIME 消息的纯文本表示形式时，如果它发现消息具有 *preamble* 属性，它将在标头及第一个分界之间区域写出这些文本。请参阅*email.parser* 和*email.generator* 了解更多细节。

请注意如果消息对象没有前导文本，则 *preamble* 属性将为 `None`。

epilogue

epilogue 属性的作用方式与 *preamble* 属性相同，区别在于它包含出现于最后一个分界与消息结尾之间的文本。

你不需要将 epilogue 设为空字符串以便让*Generator* 在文件末尾打印一个换行符。

defects

defects 属性包含在解析消息时发现的所有问题的列表。请参阅*email.errors* 了解可能的解析缺陷的详细描述。

20.1.10 email.mime: 从头创建电子邮件和 MIME 对象

源代码: Lib/email/mime/

此模块是旧版 (`Compat32`) 电子邮件 API 的组成部分。它的功能在新版 API 中被*contentmanager* 部分替代，但在某些应用中这些类仍可能有用，即使是在非旧版代码中。

通常，你是通过传递一个文件或一些文本到解析器来获得消息对象结构体的，解析器会解析文本并返回根消息对象。不过你也可以从头开始构建一个完整的消息结构体，甚至是手动构建单独的*Message* 对象。实际上，你也可以接受一个现有的结构体并添加新的*Message* 对象并移动它们。这为切片和分割 MIME 消息提供了非常方便的接口。

你可以通过创建*Message* 实例并手动添加附件和所有适当的标头来创建一个新的对象结构体。不过对于 MIME 消息来说，*email* 包提供了一些便捷子类来让事情变得更容易。

这些类列示如下:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)

模块: *email.mime.base*

这是*Message* 的所有 MIME 专属子类。通常你不会创建专门的*MIMEBase* 实例，尽管你可以这样做。*MIMEBase* 主要被提供用来作为更具体的 MIME 感知子类的便捷基类。

_maintype 是 *Content-Type* 的主类型 (例如 *text* 或 *image*)，而 *_subtype* 是 *Content-Type* 的次类型 (例如 *plain* 或 *gif*)。*_params* 是一个形参键/值字典并会被直接传递给*Message.add_header*。

如果指定了 *policy* (默认为 compat32 策略)，它将被传递给 Message。

MIMEBase 类总是会添加一个 Content-Type 标头 (基于 *_maintype*, *_subtype* 和 *_params*)，以及一个 MIME-Version 标头 (总是设为 1.0)。

在 3.6 版本发生变更: 添加了 *policy* 仅限关键字形参。

class email.mime.nonmultipart.**MIMENonMultipart**

模块: email.mime.nonmultipart

MIMEBase 的子类，这是用于非 multipart MIME 消息的中间基类。这个类的主要目标是避免使用 attach() 方法，该方法仅对 multipart 消息有意义。如果 attach() 被调用，则会引发 MultipartConversionError 异常。

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, ***, *policy=compat32*, ***_params*)

模块: email.mime.multipart

MIMEBase 的子类，这是用于 multipart MIME 消息的中间基类。可选的 *_subtype* 默认为 mixed，但可被用来指定消息的子类型。将会在消息对象中添加一个 multipart/*_subtype* 的 Content-Type 标头。并还将添加一个 MIME-Version 标头。

可选的 *boundary* 是多部分边界字符串。当为 None (默认值) 时，则会在必要时（例如当消息被序列化时）计算边界。

_subparts 是载荷初始子部分的序列。此序列必须可以被转换为列表。你总是可以使用 Message.attach 方法将新的子部分附加到消息中。

可选的 *policy* 参数默认为 compat32。

用于 Content-Type 标头的附加形参会从关键字参数中获取，或者传入到 *_params* 参数，该参数是一个关键字的字典。

在 3.6 版本发生变更: 添加了 *policy* 仅限关键字形参。

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: email.mime.application

MIMENonMultipart 的子类，MIMEApplication 类被用来表示主类型为 application 的 MIME 消息。*_data* 为包含原始应用程序数据的字节串。可选的 *_subtype* 指定 MIME 子类型并默认为 octet-stream。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的数据编码以便传输。这个可调用对象接受一个参数，该参数是 MIMEApplication 的实例。它应当使用 get_payload() 和 set_payload() 来将载荷改为已编码形式。它还应根据需要将任何 Content-Transfer-Encoding 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅 email.encoders 模块来查看内置编码器列表。

可选的 *policy* 参数默认为 compat32。

_params 会被直接传递给基类的构造器。

在 3.6 版本发生变更: 添加了 *policy* 仅限关键字形参。

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: email.mime.audio

MIMENonMultipart 的子类，MIMEAudio 类被用来创建主类型为 audio 的 MIME 消息。*_audiodata* 是包含原始音频数据的字节串。如果此数据可作为 au, wav, aiff 或 aifc 来解码，则其子类型将被自动包括在 Content-Type 标头中。在其他情况下你可以通过 *_subtype* 参数显式地指定音频子类型。如果无法猜测出次类型并且未给出 *_subtype*，则会引发 TypeError。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的音频数据编码以便传输。这个可调用对象接受一个参数，该参数是 MIMEAudio 的实例。它应当使用 get_payload() 和 set_payload()

来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给基类的构造器。

在 3.6 版本发生变更: 添加了 *policy* 仅限关键字形参。

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: *email.mime.image*

MIMENonMultipart 的子类，*MIMEImage* 类被用来创建主类型为 *image* 的 MIME 消息对象。*_imagedata* 是包含原始图像数据的字节串。如果此数据类型可以被检测（将尝试 jpeg, png, gif, tiff, rgb, pbm, pgm, ppm, rast, xbm, bmp, webp 和 exr 类型），则其子类型将被自动包括在 *Content-Type* 标头中。在其他情况下你可以通过 *_subtype* 参数显式地指定图像子类型。如果无法猜测出次类型并且未给出 *_subtype*，则会引发*TypeError*。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的图像数据编码以便传输。这个可调用对象接受一个参数，该参数是*MIMEImage* 的实例。它应当使用*get_payload()* 和*set_payload()* 来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给*MIMEBase* 构造器。

在 3.6 版本发生变更: 添加了 *policy* 仅限关键字形参。

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

模块: *email.mime.message*

MIMENonMultipart 的子类，*MIMEMessage* 类被用来创建主类型为 *message* 的 MIME 对象。*_msg* 将被用作载荷，并且必须为*Message* 类（或其子类）的实例，否则会引发*TypeError*。

可选的 *_subtype* 设置消息的子类型；它的默认值为 *rfc822*。

可选的 *policy* 参数默认为*compat32*。

在 3.6 版本发生变更: 添加了 *policy* 仅限关键字形参。

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)

模块: *email.mime.text*

MIMENonMultipart 的子类，*MIMEText* 类被用来创建主类型为 *text* 的 MIME 对象。*_text* 是用作载荷的字符串。*_subtype* 指定子类型并且默认为 *plain*。*_charset* 是文本的字符集并会作为参数传递给*MIMENonMultipart* 构造器；如果该字符串仅包含 ascii 码位则其默认值为 us-ascii，否则为 utf-8。*_charset* 形参接受一个字符串或是一个*Charset* 实例。

除非 *_charset* 参数被显式地设为 None，否则所创建的 MIMEText 对象将同时具有附带 charset 形参的 *Content-Type* 标头，以及 *Content-Transfer-Encoding* 标头。这意味着后续的 set_payload 调用将不再产生已编码的载荷，即使它在 set_payload 命令中被传入。你可以通过删除 Content-Transfer-Encoding 标头来“重置”此行为，在此之后的 set_payload 调用将自动编码新的载荷（并添加新的 *Content-Transfer-Encoding* 标头）。

可选的 *policy* 参数默认为*compat32*。

在 3.5 版本发生变更: *_charset* 也可接受*Charset* 实例。

在 3.6 版本发生变更: 添加了 *policy* 仅限关键字形参。

20.1.11 email.header: 国际化标头

源代码: Lib/email/header.py

此模块是旧式 (Compat32) email API 的一部分。在当前的 API 中标头的编码和解码是由*EmailMessage* 类的字典型 API 来透明地处理的。除了在旧有代码中使用，此模块在需要完全控制当编码标头时所使用的字符集时也很有用处。

本节中的其余文本是此模块的原始文档。

RFC 2822 是描述电子邮件消息格式的基础标准。它派生自更早的 **RFC 822** 标准，该标准在大多数电子邮件仅由 ASCII 字符组成时已被广泛使用。**RFC 2822** 所描述的规范假定电子邮件都只包含 7 位 ASCII 字符。

当然，随着电子邮件在全球部署，它已经变得国际化了，例如电子邮件消息中现在可以使用特定语言的专属字符集。这个基础标准仍然要求电子邮件消息只使用 7 位 ASCII 字符来进行传输，为此编写了大量 RFC 来描述如何将包含非 ASCII 字符的电子邮件编码为符合 **RFC 2822** 的格式。这些 RFC 包括 **RFC 2045**, **RFC 2046**, **RFC 2047** 和 **RFC 2231**。*email* 包在其*email.header* 和*email.charset* 模块中支持了这些标准。

如果你想在你的电子邮件标头中包括非 ASCII 字符，比如说是在 *Subject* 或 *To* 字段中，你应当使用*Header* 类并将*Message* 对象中的字段赋值为*Header* 的实例而不是使用字符串作为字段值。请从*email.header* 模块导入*Header* 类。例如:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

是否注意到这里我们是如何希望 *Subject* 字段包含非 ASCII 字符的？我们通过创建一个*Header* 实例并传入字节串编码所用的字符集来做到这一点。当后续的*Message* 实例被展平时，*Subject* 字段会正确地按 **RFC 2047** 来编码。可感知 MIME 的电子邮件阅读器将会使用嵌入的 ISO-8859-1 字符来显示此标头。

以下是*Header* 类描述:

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

创建符合 MIME 要求的标头，其中可包含不同字符集的字符串。

可选的 *s* 是初始标头值。如果为 None (默认值)，则表示初始标头值未设置。你可以在稍后使用*append()* 方法调用向标头添加新值。*s* 可以是*bytes* 或*str* 的实例，注意参阅*append()* 文档了解相关语义。

可选的 *charset* 用于两种目的：它的含义与*append()* 方法的 *charset* 参数相同。它还会为所有省略了 *charset* 参数的后续*append()* 调用设置默认字符集。如果 *charset* 在构造器中未提供（默认设置），则会将 us-ascii 字符集用作 *s* 的初始字符集以及后续*append()* 调用的默认字符集。

通过 maximum line length can be specified explicitly via *maxlinelen* 可以显式地指定行长度的最大值。要将第一行拆分为更短的值（以适应未被包括在 *s* 中的字段标头，例如 *Subject* 等）则将字段名称作为 *header_name* 传入。默认的 *maxlinelen* 为 78，而 *header_name* 的默认值为 None，表示不考虑拆分超长标头的第一行。

可选的 *continuation_ws* 必须为符合 **RFC 2822** 的折叠用空白符，通常是空格符或硬制表符。这个字符将被加缀至连续行的开头。*continuation_ws* 默认为一个空格符。

可选的 *errors* 会被直接传递给*append()* 方法。

append(*s*, *charset=None*, *errors='strict'*)

将字符串 *s* 添加到 MIME 标头。

如果给出可选的 *charset*，它应当是一个*Charset* 实例 (参见*email.charset*) 或字符集名称，该参数将被转换为一个*Charset* 实例。如果为 None (默认值) 则表示会使用构造器中给出的 *charset*。

s 可以是*bytes* 或*str* 的实例。如果它是*bytes* 的实例，则 *charset* 为该字节串的编码格式，如果字节串无法用该字符集来解码则将引发*UnicodeError*。

如果 *s* 是*str* 的实例，则 *charset* 是用来指定字符串中字符字符集的提示。

在这两种情况下，当使用 **RFC 2047** 规则产生符合 **RFC 2822** 的标头时，将使用指定字符集的输出编解码器来编码字符串。如果字符串无法使用该输出编解码器来编码，则将引发 UnicodeError。

可选的 *errors* 会在 *s* 为字节串时被作为 errors 参数传递给 decode 调用。

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)

将消息标头编码为符合 RFC 的格式，可能会对过长的行采取折行并将非 ASCII 部分以 base64 或 quoted-printable 编码格式进行封装。

可选的 *splitchars* 是一个字符串，其中包含应在正常的标头折行处理期间由拆分算法赋予额外权重的字符。这是对于 **RFC 2822** 中' 更高层级语法拆分' 的很粗略的支持：在拆分期间会首选在 splitchar 之前的拆分点，字符的优先级是基于它们在字符串中的出现顺序。字符串中可包含空格和制表符以指明当其他拆分字符未在被拆分行中出现时是否要将某个字符作为优先于另一个字符的首选拆分点。拆分字符不会影响以 **RFC 2047** 编码的行。

如果给出 *maxlinelen*，它将覆盖实例的最大行长度值。

linesep 指定用来分隔已折叠标头行的字符。它默认为 Python 应用程序代码中最常用的值 (\n)，但也可以指定为 \r\n 以便产生带有符合 RFC 的行分隔符的标头。

在 3.2 版本发生变更: 增加了 *linesep* 参数。

Header 类还提供了一些方法以支持标准运算符和内置函数。

__str__()

以字符串形式返回*Header* 的近似表示，使用不受限制的行长度。所有部分都会使用指定编码格式转换为 unicode 并适当地连接起来。任何带有 'unknown-8bit' 字符集的部分都会使用 'replace' 错误处理程序解码为 ASCII。

在 3.2 版本发生变更: 增加对 'unknown-8bit' 字符集的处理。

__eq__(*other*)

这个方法允许你对两个*Header* 实例进行相等比较。

__ne__(*other*)

这个方法允许你对两个*Header* 实例进行不等比较。

email.header 模块还提供了下列便捷函数。

email.header.**decode_header**(*header*)

在不转换字符集的情况下对消息标头值进行解码。*header* 为标头值。

这个函数返回一个 (decoded_string, charset) 对的列表，其中包含标头的每个已解码部分。对于标头的未编码部分 *charset* 为 None，在其他情况下则为一个包含已编码字符串中所指定字符集名称的小写字符串。

以下是为示例代码:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)

基于*decode_header()* 所返回的数据对序列创建一个*Header* 实例。

decode_header() 接受一个标头值字符串并返回格式为 (decoded_string, charset) 的数据对序列，其中 *charset* 是字符集名称。

这个函数接受这样的数据对序列并返回一个*Header* 实例。可选的 *maxlinelen*, *header_name* 和 *continuation_ws* 与*Header* 构造器中的含义相同。

20.1.12 email.charset: 表示字符集

源代码: Lib/email/charset.py

此模块是旧版 (Compat32) email API 的组成部分。在新版 API 中只会使用其中的别名表。

本节中的其余文本是此模块的原始文档。

此模块提供了一个*Charset* 类用来表示电子邮件消息中的字符集和字符集转换操作，以及一个字符集注册表和几个用于操作此注册表的便捷方法。*Charset* 的实例在*email* 包的其他几个模块中也有使用。

请从*email.charset* 模块导入这个类。

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)

将字符集映射到其 email 特征属性。

这个类提供了特定字符集对于电子邮件的要求的相关信息。考虑到适用编解码器的可用性，它还为字符集之间的转换提供了一些便捷例程。在给定字符集的情况下，它将尽可能地以符合 RFC 的方式在电子邮件消息中提供有关如何使用该字符集的信息。

特定字符集当在电子邮件标头或消息体中使用时必须以 quoted-printable 或 base64 来编码。某些字符集则必须被立即转换，不允许在电子邮件中使用。

可选的 *input_charset* 说明如下；它总是会被强制转为小写。在进行别名正规化后它还会被用来查询字符集注册表以找出用于该字符集的标头编码格式、消息体编码格式和输出转换编解码器。举例来说，如果 *input_charset* 为 iso-8859-1，则标头和消息体将会使用 quoted-printable 来编码并且不需要输出转换编解码器。如果 *input_charset* 为 euc-jp，则标头将使用 base64 来编码，消息体将不会被编码，但输出文本将从 euc-jp 字符集转换为 iso-2022-jp 字符集。

Charset 实例具有下列数据属性:

input_charset

指定的初始字符集。通用别名会被转换为它们的 官方电子邮件名称 (例如 latin_1 会被转换为 iso-8859-1)。默认值为 7 位 us-ascii。

header_encoding

如果字符集在用于电子邮件标头之前必须被编码，此属性将被设为 charset.QP (表示 quoted-printable 编码格式), charset.BASE64 (表示 base64 编码格式) 或 charset.SHORTEST 表示 QP 或 BASE64 编码格式中最简短的一个。在其他情况下，该属性将为 None。

body_encoding

与 *header_encoding* 一样，但是用来描述电子邮件消息体的编码格式，它实际上可以与标头编码格式不同。charset.SHORTEST 不允许被用作 *body_encoding*。

output_charset

某些字符集在用于电子邮件标头或消息体之前必须被转换。如果 *input_charset* 是这些字符集之一，该属性将包含输出将要转换的字符集名称。在其他情况下，该属性将为 None。

input_codec

用于将 *input_charset* 转换为 Unicode 的 Python 编解码器名称。如果不需要任何转换编解码器，该属性将为 None。

output_codec

用于将 Unicode 转换为 *output_charset* 的 Python 编解码器名称。如果不需要任何转换编解码器，该属性将具有与 *input_codec* 相同的值。

Charset 实例还有下列方法:

get_body_encoding()

返回用于消息体编码的内容转换编码格式。

根据所使用的编码格式返回 quoted-printable 或 base64，或是返回一个函数，在这种情况下你应当调用该函数并附带一个参数，即被编码的消息对象。该函数应当自行将 *Content-Transfer-Encoding* 标头设为适当的值。

如果 *body_encoding* 为 QP 则返回字符串 quoted-printable，如果 *body_encoding* 为 BASE64 则返回字符串 base64，并在其他情况下返回字符串 7bit。

get_output_charset()

返回输出字符集。

如果 *output_charset* 属性不为 None 则返回该属性，否则返回 *input_charset*。

header_encode(*string*)

对字符串 *string* 执行标头编码。

编码格式的类型 (base64 或 quoted-printable) 将取决于 *header_encoding* 属性。

header_encode_lines(*string*, *maxlengths*)

通过先将 *string* 转换为字节串来对其执行标头编码。

这类似于*header_encode()*，区别是字符串会被调整至参数 *maxlengths* 所给出的最大行长度，它应当是一个迭代器：该迭代器返回的每个元素将提供下一个最大行长度。

body_encode(*string*)

对字符串 *string* 执行消息体编码。

编码格式的类型 (base64 或 quoted-printable) 将取决于 *body_encoding* 属性。

Charset 类还提供了一些方法以支持标准运算和内置函数。

__str__()

将 *input_charset* 以转为小写的字符串形式返回。__repr__() 是 __str__() 的别名。

__eq__(*other*)

这个方法允许你对两个*Charset* 实例进行相等比较。

__ne__(*other*)

这个方法允许你对两个*Charset* 实例进行相等比较。

email.charset 模块还提供了下列函数用于向全局字符集、别名以及编解码器注册表添加新条目:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)

向全局注册表添加字符特征属性。

charset 是输入字符集，它必须为某个字符集的正规名称。

可选的 *header_enc* 和 *body_enc* 可以是 charset.QP 表示 quoted-printable 编码格式, charset.BASE64 表示 base64 编码格式, charset.SHORTEST 表示 quoted-printable 或 base64 编码格式中较短的一个，或者为 None 表示没有编码格式。SHORTEST 仅对 *header_enc* 有效。默认值为 None 表示没有编码格式。

可选的 *output_charset* 是输出所应当采用的字符集。当 Charset.convert() 方法被调用时将会执行从输入字符集到输出字符集的转换。默认情况下输出字符集将与输入字符集相同。

input_charset 和 *output_charset* 都必须在模块的字符集-编解码器映射中具有 Unicode 编解码器条目；使用*add_codec()* 可添加本模块还不知道的编解码器。请参阅*codecs* 模块的文档来了解更多信息。

全局字符集注册表保存在模块全局字典 CHARSETS 中。

email.charset.**add_alias**(*alias*, *canonical*)

添加一个字符集别名。*alias* 为特定的别名，例如 latin-1。*canonical* 是字符集的正规名称，例如 iso-8859-1。

全局字符集注册表保存在模块全局字典 ALIASES 中。

email.charset.**add_codec**(*charset*, *codecname*)

添加在给定字符集的字符和 Unicode 之间建立映射的编解码器。

charset 是某个字符集的正规名称。*codecname* 是某个 Python 编解码器的名称，可以被用来作为*str* 的*encode()* 方法的第二个参数。

20.1.13 email.encoders: 编码器

源代码: Lib/email/encoders.py

此模块是旧版 (Compat32) email API 的组成部分。在新版 API 中将由*set_content()* 方法的 *cte* 形参提供该功能。

此模块在 Python 3 中已弃用。这里提供的函数不应被显式地调用，因为*MIMEText* 类会在类实例化期间使用 *_subtype* 和 *_charset* 值来设置内容类型和 CTE 标头。

本节中的其余文本是此模块的原始文档。

当创建全新的*Message* 对象时，你经常需要对载荷编码以便通过兼容的邮件服务器进行传输。对于包含二进制数据的 *image/** 和 *text/** 类型的消息来说尤其如此。

email 包在其*encoders* 模块中提供了一些方便的编码器。这些编码器实际上由*MIMEAudio* 和*MIMEImage* 类构造器使用以提供默认编码格式。所有编码器函数都只接受一个参数，即要编码的消息对象。它们通常会提取有效截荷，对其进行编码，并将载荷重置为新近编码的值。它们还应当相应地设置 *Content-Transfer-Encoding* 标头。

请注意，这些函数对于多段消息没有意义。它们必须应用到各个单独的段上面，而不是整体。如果直接传递一个多段类型的消息，会产生一个*TypeError* 错误。

下面是提供的编码函数：

email.encoders.**encode_quopri**(*msg*)

将有效数据编码为经转换的可打印形式，并将 *Content-Transfer-Encoding* 标头设置为 quoted-printable[1]。当大多数实际的数据是普通的可打印数据但包含少量不可打印的字符时，这是一个很好的编码。

email.encoders.**encode_base64**(*msg*)

将有效载荷编码为 base64 形式，并将 *Content-Transfer-Encoding* 标头设为 base64。当你的载荷主要包含不可打印数据时这是一种很好用的编码格式，因为它比 quoted-printable 更紧凑。base64 编码格式的缺点是它会使文本变成人类不可读的形式。

email.encoders.**encode_7or8bit**(*msg*)

此函数并不实际改变消息的有效载荷，但它会基于载荷数据将 *Content-Transfer-Encoding* 标头相应地设为 7bit 或 8bit。

email.encoders.**encode_noop**(*msg*)

此函数什么都不会做；它甚至不会设置 *Content-Transfer-Encoding* 标头。

备注

20.1.14 email.utils: 杂项工具

源代码: Lib/email/utils.py

---

email.utils 模块提供如下几个工具

[1] 请注意使用*encode_quopri()* 编码格式还会对数据中的所有制表符和空格符进行编码。

email.utils.**localtime**(*dt=None*)

将当地时间作为感知型 datetime 对象返回。如果不带参数地调用，则返回当前时间。否则 *dt* 参数应为*datetime* 的实例，并将根据系统时区数据库将其转换为当地时区。如果 *dt* 为简单型 (即 `dt.tzinfo` 为 `None`)，它将被假定为当地时间。*isdst* 形参将被忽略。

Added in version 3.3.

Deprecated since version 3.12, will be removed in version 3.14: *isdst* 形参。

email.utils.**make_msgid**(*idstring=None*, *domain=None*)

返回一个适合作为兼容 **RFC 2822** 的 *Message-ID* 标头的字符串。可选参数 *idstring* 可传入一字符串以增强该消息 ID 的唯一性。可选参数 *domain* 可用于提供消息 ID 中字符'@' 之后的部分，其默认值是本机的主机名。正常情况下无需覆盖此默认值，但在特定情况下覆盖默认值可能会有用，比如构建一个分布式系统，在多台主机上采用一致的域名。

在 3.2 版本发生变更: 增加了关键字 *domain*

下列函数是旧（`Compat32`）电子邮件 API 的一部分。新 API 提供的解析和格式化在标头解析机制中已经被自动完成，故在使用新 API 时没有必要直接使用它们函数。

email.utils.**quote**(*str*)

返回一个新的字符串，*str* 中的反斜杠被替换为两个反斜杠，并且双引号被替换为反斜杠加双引号。

email.utils.**unquote**(*str*)

返回 *str* 被去除引用版本的字符串。如果 *str* 开头和结尾均是双引号，则这对双引号被去除。类似地，如果 *str* 开头和结尾都是尖角括号，这对尖角括号会被去除。

email.utils.**parseaddr**(*address*, ***, *strict=True*)

将地址（应为诸如 *To* 或者 *Cc* 之类包含地址的字段值）解析为构成之的 *真实名字* 和 *电子邮件地址* 部分。返回包含这两个信息的一个元组；如若解析失败，则返回一个二元组 `('', '')`。

如果 *strict* 为真值，将使用拒绝错误形式输入的严格解析器。

在 3.13 版本发生变更: 增加了 *strict* 可选形参并将默认拒绝错误形式输入。

email.utils.**formataddr**(*pair*, *charset='utf-8'*)

是*parseaddr()* 的逆操作，接受一个（*真实名字*, *电子邮件地址*）的二元组，并返回适合于 *To* or *Cc* 标头的字符串。如果第一个元素为假性值，则第二个元素将被原样返回。

可选地，如果指定 *charset*，则被视为一符合 **RFC 2047** 的编码字符集，用于编码 *真实名字* 中的非 ASCII 字符。可以是一个*str* 类的实例，或者一个*Charset* 类。默认为 `utf-8` 。

在 3.3 版本发生变更: 添加了 *charset* 选项。

email.utils.**getaddresses**(*fieldvalues*, ***, *strict=True*)

该方法返回一个与 `parseaddr()` 返回值形式相同的 2 元组。*fieldvalues* 是与*Message.get_all* 返回值形式相同的由标头字段值组成的序列。

如果 *strict* 为真值，将使用拒绝错误形式输入的严格解析器。

下面简单示例可获取一条消息的所有接收方:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

在 3.13 版本发生变更: 增加了 *strict* 可选形参并将默认拒绝错误形式输入。

email.utils.**parsedate**(*date*)

尝试根据 **RFC 2822** 的规则解析一个日期。然而，有些寄信人不严格遵守这一格式，所以这种情况下*parsedate()* 会尝试猜测其形式。*date* 是一个字符串包含了一个形如 `"Mon, 20 Nov 1995`

19:12:08 -0500" 的 **RFC 2822** 格式日期。如果日期解析成功，*parsedate()* 将返回一个九元组，可直接传递给*time.mktime()*；否则返回 None。注意返回的元组中下标为 6、7、8 的部分是无用的。

email.utils.**parsedate_tz**(*date*)

执行与*parsedate()* 相同的功能，但会返回 None 或是一个 10 元组；前 9 个元素构成一个可以直接传给*time.mktime()* 的元组，而第十个元素则是该日期的时区与 UTC (格林威治平均时 GMT 的正式名称)[1] 的时差。如果输入字符串不带时区，则所返回元组的最后一个元素将为 0，这表示 UTC。请注意结果元组的索引号 6, 7 和 8 是不可用的。

email.utils.**parsedate_to_datetime**(*date*)

format_datetime() 的逆操作。执行与*parsedate()* 相同的功能，但会在成功时返回一个*datetime*；否则如果 *date* 包含无效的值例如小时值大于 23 或时区偏移量不在 -24 和 24 时范围之内则会引发 ValueError。如果输入日期的时区值为 -0000，则 datetime 将为一个简单形 datetime，而如果日期符合 RFC 标准则它将代表一个 UTC 时间，但是并不指明日期所在消息的实际源时区。如果输入日期具有任何其他有效的时区偏移量，则 datetime 将是一个感知型 datetime 并与*timezone tzinfo* 相对应。

Added in version 3.3.

email.utils.**mktime_tz**(*tuple*)

将*parsedate_tz()* 所返回的 10 元组转换为一个 UTC 时间戳（相距 Epoch 纪元初始的秒数）。如果元组中的时区项为 None，则视为当地时间。

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)

返回 **RFC 2822** 标准的日期字符串，例如:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

可选的 *timeval* 如果给出，则是一个可被*time.gmtime()* 和*time.localtime()* 接受的浮点数时间值，否则会使用当前时间。

可选的 *localtime* 是一个旗标，当为 True 时，将会解析 *timeval*，并返回一个相对于当地时区而非 UTC 的日期值，并会适当地考虑夏令时。默认值 False 表示使用 UTC。

可选的 *usegmt* 是一个旗标，当为 True 时，将会输出一个日期字符串，其中时区表示为 ascii 字符串 GMT 而非数字形式的 -0000。这对某些协议（例如 HTTP）来说是必要的。这仅在 *localtime* 为 False 时应用。默认值为 False。

email.utils.**format_datetime**(*dt*, *usegmt=False*)

类似于 formatdate，但输入的是一个*datetime* 实例。如果实例是一个简单型 datetime，它会被视为" 不带源时区信息的 UTC"，并且使用传统的 -0000 作为时区。如果实例是一个感知型 datetime，则会使用数字形式的时区时差。如果实例是感知型且时区时差为零，则 *usegmt* 可能会被设为 True，在这种情况下将使用字符串 GMT 而非数字形式的时区时差。这提供了一种生成符合标准 HTTP 日期标头的方式。

Added in version 3.3.

email.utils.**decode_rfc2231**(*s*)

根据 **RFC 2231** 解码字符串 *s*。

email.utils.**encode_rfc2231**(*s*, *charset=None*, *language=None*)

根据 **RFC 2231** 对字符串 *s* 进行编码。可选的 *charset* 和 *language* 如果给出，则为指明要使用的字符集名称和语言名称。如果两者均未给出，则会原样返回 *s*。如果给出 *charset* 但未给出 *language*，则会使用空字符串作为 *language* 值来对字符串进行编码。

email.utils.**collapse_rfc2231_value**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)

当以 **RFC 2231** 格式来编码标头形参时，*Message.get_param* 可能返回一个包含字符集、语言和值的 3 元组。*collapse_rfc2231_value()* 会将此返回为一个 unicode 字符串。可选的 *errors* 会被传递给*str* 的*encode()* 方法的 *errors* 参数；它的默认值为 'replace'。可选的 *fallback_charset* 指定当 **RFC 2231** 标头中的字符集无法被 Python 识别时要使用的字符集；它的默认值为 'us-ascii'。

[1] 请注意时区时差的符号与同一时区的 time.timezone 变量的符号相反；后者遵循 POSIX 标准而此模块遵循 **RFC 2822**。

为方便起见，如果传给*collapse_rfc2231_value()* 的 *value* 不是一个元组，则应为一个字符串并会将其原样返回。

email.utils.**decode_params**(*params*)

根据 **RFC 2231** 解码参数列表。*params* 是一个包含 (content-type, string-value) 形式的元素的 2 元组的序列。

备注

20.1.15 email.iterators: 迭代器

源代码: Lib/email/iterators.py

通过*Message.walk* 方法来迭代消息对象树是相当容易的。*email.iterators* 模块提供了一些适用于消息对象树的高层级迭代器。

email.iterators.**body_line_iterator**(*msg*, *decode=False*)

此函数会迭代 *msg* 的所有子部分中的所有载荷，逐行返回字符串载荷。它会跳过所有子部分的标头，并且它也会跳过任何包含不为 Python 字符串的载荷的子部分。这基本上等价于使用*readline()* 从一个文件读取消息的纯文本表示形式，并跳过所有中间的标头。

可选的 *decode* 会被传递给*Message.get_payload*。

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)

此函数会迭代 *msg* 的所有子部分，只返回其中与 *maintype* 和 *subtype* 所指定的 MIME 类型相匹配的子部分。

请注意 *subtype* 是可选项；如果省略，则仅使用主类型来进行子部分 MIME 类型的匹配。*maintype* 也是可选项；它的默认值为 *text*。

因此，在默认情况下*typed_subpart_iterator()* 会返回每一个 MIME 类型为 *text/** 的子部分。

增加了以下函数作为有用的调试工具。它 *不应当* 被视为该包所支持的公共接口的组成部分。

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)

打印消息对象结构的内容类型的缩进表示形式。例如:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

可选项 *fp* 是一个作为打印输出目标的文件型对象。它必须适用于 Python 的*print()* 函数。*level* 是供内部使用的。*include_default* 如果为真值，则会同时打印默认类型。

参见

smtplib 模块
: SMTP (简单邮件传输协议) 客户端

poplib 模块
: POP (邮局协议) 客户端

imaplib 模块
: IMAP (互联网消息访问协议) 客户端

mailbox 模块
: 创建、读取和管理使用保存在磁盘中的多种标准格式的消息集的工具。

20.2 `json` --- JSON 编码器和解码器

源代码: Lib/json/__init__.py

JSON (JavaScript Object Notation)，由 **RFC 7159** (它取代了 **RFC 4627**) 和 ECMA-404 进行规范，是一个受到 JavaScript 对象字面值语法启发的轻量级数据交换格式 (虽然它并不是严格意义上的 JavaScript 子集[1])。

> **警告**
>
> 在解析来自不受信任恶劣的 JSON 数据时要小心谨慎。恶意的 JSON 字符串可能导致解码器消耗大量 CPU 和内存资源。建议对要解析的数据大小进行限制。

json 提供了与标准库*marshal* 和*pickle* 相似的 API 接口。

对基本的 Python 对象层次结构进行编码:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

紧凑编码:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

美化输出:

[1] 正如 RFC 7159 的勘误表 所说明的，JSON 允许以字符串表示字面值字符 U+2028 (LINE SEPARATOR) 和 U+2029 (PARAGRAPH SEPARATOR)，而 JavaScript (在 ECMAScript 5.1 版中) 不允许。

```
>>> import json
>>> print(json.dumps({'6': 7, '4': 5}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

对 JSON 对象解码的特殊化:

```
>>> import json
>>> def custom_json(obj):
...     if isinstance(obj, complex):
...         return {'__complex__': True, 'real': obj.real, 'imag': obj.imag}
...     raise TypeError(f'Cannot serialize object of {type(obj)}')
...
>>> json.dumps(1 + 2j, default=custom_json)
'{"__complex__": true, "real": 1.0, "imag": 2.0}'
```

JSON 解码:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

特殊 JSON 对象解码:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

扩展*JSONEncoder*:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return super().default(obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

从命令行使用*json.tool* 来验证并美化输出:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

详细文档请参见*CLI*。

备注

JSON 是 YAML 1.2 的一个子集。由该模块的默认设置所产生的 JSON（尤其是默认的 *separators* 值）也是 YAML 1.0 和 1.1 的一个子集。因此该模块也能被用作 YAML 序列化器。

备注

这个模块的编码器和解码器默认保护输入和输出的顺序。仅当底层的容器未排序时才会失去顺序。

20.2.1 基本使用

json.**dump**(*obj*, *fp*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

使用这个*Python 至 JSON 转换表* 将 *obj* 序列化为输出到 *fp* (一个支持 `.write()` 的*file-like object*) 的已格式化的 JSON 流。

备注

与*pickle* 和*marshal* 不同，JSON 不是一个具有框架的协议，所以尝试多次使用同一个 *fp* 调用*dump()* 来序列化多个对象会产生一个不合规的 JSON 文件。

参数

- **obj** (object) -- 要序列化的 Python 对象。
- **fp** (*file-like object*) -- 文件型对象 *obj* 将为序列化输出的目标。`json` 模块总是产生*str* 对象，而不是*bytes* 对象，因此 `fp.write()` 必须支持*str* 输入。
- **skipkeys** (bool) -- 如为 `True`，则不为基本类型 (*str*, *int*, *float*, *bool*, None) 的键将被跳过而不会引发*TypeError*。默认为 `False`。
- **ensure_ascii** (bool) -- 如为 `True` (默认值)，则输出将保证对所有输入的非 ASCII 字符进行转义。如为 `False`，这些字符将被原样输出。
- **check_circular** (bool) -- 如为 `False`，则对容器类型的循环引用检查会被跳过并且循环引用将导致*RecursionError* (或更糟的情况)。默认为 `True`。
- **allow_nan** (bool) -- 如为 `False`，则对超范围的*float* 值 (`nan`, `inf`, `-inf`) 进行序列化将导致*ValueError*，以严格遵循 JSON 规范。如为 `True` (默认值)，则将使用它们的 JavaScript 等价形式 (`NaN`, `Infinity`, `-Infinity`)。
- **cls** (a *JSONEncoder* subclass) -- 如果设置，则重写为一个带有*default()* 方法的自定义 JSON 编码器，用以序列化为自定义的数据类型。如为 `None` (默认值)，则使用 `JSONEncoder`。
- **indent** (int | str | *None*) -- 如为一个正整数或字符串，JSON 数组元素和对象成员将按其所指定的缩进层级美化打印。正整数表示每级缩进指定数量的空格；字符串 (如 `"\t"`) 则被用于每级缩进。如为零、负数或 `""` (空字符串)，则仅插入换行符。如为 `None` (默认值)，则使用最紧凑表示形式。

- **separators** (*tuple* / *None*) -- 一个二元组: (item_separator, key_separator)。如为 None (默认值)，则默认当 *indent* 为 None 时 *separators* 为 (', ', ': ')，否则为 (',', ': ')。要使用最紧凑形式的 JSON，可指定 (',', ':') 来去除空格。
- **default** (*callable* | None) -- 当对象无法被序列化时将被调用的函数。它应该返回一个可被 JSON 编码的版本或是引发 *TypeError*。如为 None (默认值)，则会引发 TypeError。
- **sort_keys** (bool) -- 如为 True，则字典输出将按键排序。默认为 False。

在 3.2 版本发生变更: 现允许使用字符串作为 *indent* 而不再仅仅是整数。

在 3.4 版本发生变更: 现当 *indent* 不是 None 时，采用 (',', ': ') 作为默认值。

在 3.6 版本发生变更: 所有可选形参现在都是仅限关键字参数。

json.**dumps**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, **kw*)

使用这个转换表 将 *obj* 序列化为 JSON 格式的 *str*。其参数的含义与 *dump()* 中的相同。

> **备注**
>
> JSON 中的键-值对中的键永远是 *str* 类型的。当一个对象被转化为 JSON 时，字典中所有的键都会被强制转换为字符串。这所造成的结果是字典被转换为 JSON 然后转换回字典时可能和原来的不相等。换句话说，如果 x 具有非字符串的键，则有 loads(dumps(x)) != x。

json.**load**(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, **kw*)

使用*JSON 至 Python 转换表* 将 *fp* 反序列化为 Python 对象。

参数

- **fp** (*file-like object*) -- 一个包含要执行反序列化的 JSON 文档的支持 .read() 的*text file* 或*binary file*。
- **cls** (a *JSONDecoder* subclass) -- 如果设定此参数，则使用自定义的 JSON 解码器。任何传递给 load() 方法的多余关键字参数都会被传递给 *cls* 的构造器。如果传入 None （默认值)，则会使用 JSONDecoder 。
- **object_hook** (*callable* | None) -- 如果设置，将是一个在调用时以任意被解码的对象字面值 (即一个 *dict*) 作为参数的函数。该函数的返回值将代替该 *dict* 使用。此特性可被用于实现自定义解码器，例如 JSON-RPC 类提示。默认值为 None。
- **object_pairs_hook** (*callable* | None) -- 如果设置，将是一个在调用时以对照值的有序列表进行解码的任意对象字面值作为参数的函数。该函数的返回值将代替该 *dict* 使用。此特性可被用于实现自定义解码器。如果还设置了 *object_hook*，则 *object_pairs_hook* 的优先级更高，。默认值为 None。
- **parse_float** (*callable* | None) -- 如果设置，将是一个在调用时以代表每个要解码的 JSON 浮点数的字符串作为参数的函数。如为 None (默认值)，则它等价于 float(num_str)。这可被用于将 JSON 浮点数解析为自定义数据类型，例如 *decimal.Decimal*。
- **parse_int** (*callable* | None) -- 如果设置，将是一个在调用时以代表每个要解码的 JSON 整数的字符串作为参数的函数。如为 None (默认值)，则它等价于 int(num_str)。这可被用于将 JSON 整数解析为自定义数据类型，例如 *float*。
- **parse_constant** (*callable* | None) -- 如果设置，将是一个以下列字符串之一作为参数的函数: '-Infinity', 'Infinity' 或 'NaN'。这可被用于在遇到无效的 JSON 数字时引发异常。默认值为 None。

抛出

- ***JSONDecodeError*** -- 当被反序列化的数据不是合法的 JSON 文档。
- ***UnicodeDecodeError*** -- 当被反序列化的数据不包含 UTF-8, UTF-16 或 UTF-32 编码的数据。

在 3.1 版本发生变更:

- 增加了可选的 *object_pairs_hook* 形参。
- *parse_constant* 不再调用'null'，'true'，'false'。

在 3.6 版本发生变更:

- 所有可选形参现在都是仅限关键字参数。
- *fp* 现在可以是*binary file*。输入编码应当是 UTF-8，UTF-16 或者 UTF-32。

在 3.11 版本发生变更: 现在`int()`默认的 *parse_int* 会通过解释器的整数字符串长度上限来限制整数字符串的最大长度以帮助避免拒绝服务攻击。

json.**loads**(*s*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

类似于`load()`，但不是针对文件型对象，而是使用这个转换表将 *s* (一个包含 JSON 的`str`, `bytes` 或`bytearray` 实例) 反序列化为 Python 对象。

在 3.6 版本发生变更: *s* 现在可以为`bytes` 或`bytearray` 类型。输入编码应为 UTF-8, UTF-16 或 UTF-32。

在 3.9 版本发生变更: 关键字参数 *encoding* 已被移除。

20.2.2 编码器和解码器

class json.**JSONDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

简单的 JSON 解码器。

默认情况下，解码执行以下翻译:

JSON	Python
object -- 对象	dict
array	list -- 列表
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

它还将“NaN”、“Infinity”和“-Infinity”理解为它们对应的“float”值，这超出了 JSON 规范。

object_hook 是一个可选的函数，它被调用时将传入每个要解码的 JSON 对象而其返回值将被用来替代给定的`dict`。它可被用于提供自定义的反序列化操作 (比如支持 JSON-RPC 类提示)。

object_pairs_hook 是一个可选的函数，它被调用时将传入每个以一个对照值的有序列表进行解码的 JSON 对象的结果。*object_pairs_hook* 的返回值将被用于代替原本的`dict`。此特性可被用于实现自定义的解码器。如果还定义了 *object_hook*，则 *object_pairs_hook* 的优先级更高。

在 3.1 版本发生变更: 添加了对 *object_pairs_hook* 的支持。

parse_float 是一个可选的函数，它被调用时将传入每个要解码的 JSON 浮点数。默认情况下，这相当于 `float(num_str)`。它可被用于 JSON 浮点数使用其他类型或解析器的情况 (比如`decimal.Decimal`)。

parse_int 是一个可选的函数，它被调用时将传入每个要解码的 JSON 整数。默认情况下，这相当于 `int(num_str)`。它可被用于 JSON 整数使用其他数据类型或解析器的情况 (比如`float`)。

parse_constant 是一个可选的函数，它被调用时将传入以下字符串之一: `'-Infinity'`, `'Infinity'`, `'NaN'`。它可被用于当遇到无效的 JSON 数字时引发一个异常。

如果 *strict* 为 false （默认为 `True`），那么控制字符将被允许在字符串内。在此上下文中的控制字符编码在范围 0--31 内的字符，包括 `'\t'` (制表符)，`'\n'`，`'\r'` 和 `'\0'`。

如果反序列化的数据不是有效 JSON 文档，引发*JSONDecodeError* 错误。

在 3.6 版本发生变更: 所有形参现在都是仅限关键字参数。

decode(*s*)

返回 *s* 的 Python 表示形式（包含一个 JSON 文档的*str* 实例）。

如果给定的 JSON 文档无效则将引发*JSONDecodeError*。

raw_decode(*s*)

从 *s* 中解码出 JSON 文档（以 JSON 文档开头的一个*str* 对象）并返回一个 Python 表示形式为 2 元组以及指明该文档在 *s* 中结束位置的序号。

这可以用于从一个字符串解码 JSON 文档，该字符串的末尾可能有无关的数据。

class json.**JSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

用于 Python 数据结构的可扩展 JSON 编码器。

默认支持以下对象和类型:

Python	JSON
dict	object -- 对象
list, tuple	array
str	string
int, float, int 和 float 派生的枚举	number
True	true
False	false
None	null

在 3.4 版本发生变更: 添加了对 int 和 float 派生的枚举类的支持

要将其扩展至识别其他对象，需要子类化并实现*default()*，如果可能还要实现另一个返回 `o` 的可序列化对象的方法，否则它应当调用超类实现 (来引发*TypeError*)。

如果 *skipkeys* 为假值（默认），则当尝试对不为*str*, *int*, *float*, *bool* 或 `None` 的键进行编码时将会引发*TypeError*。如果 *skipkeys* 为真值，这样的条目将被直接跳过。

如果 *ensure_ascii* 是 true （即默认值），输出保证将所有输入的非 ASCII 字符转义。如果 *ensure_ascii* 是 false，这些字符会原样输出。

如果 *check_circular* 为真值（默认），那么列表、字典和自定义的已编码对象将在编码期间进行循环引用检查以防止无限递归 (无限递归会导致*RecursionError*)。在其他情况下，将不会进行这种检查。

如果 *allow_nan* 为 true （默认），那么 `NaN`，`Infinity`，和 `-Infinity` 进行编码。此行为不符合 JSON 规范，但与大多数的基于 Javascript 的编码器和解码器一致。否则，它将是一个*ValueError* 来编码这些浮点数。

如果 *sort_keys* 为 true （默认为: `False`），那么字典的输出是按照键排序；这对回归测试很有用，以确保可以每天比较 JSON 序列化。

如果 *indent* 是一个非负整数或者字符串，那么 JSON 数组元素和对象成员会被美化输出为该值指定的缩进等级。如果缩进等级为零、负数或者 `""`，则只会添加换行符。`None` (默认值) 选择最紧凑的表达。使用一个正整数会让每一层缩进同样数量的空格。如果 *indent* 是一个字符串 (比如 `"\t"`)，那个字符串会被用于缩进每一层。

在 3.2 版本发生变更: 现允许使用字符串作为 *indent* 而不再仅仅是整数。

当被指定时，*separators* 应当是一个 (item_separator, key_separator) 元组。当 *indent* 为 None 时，默认值取 (', ', ': ')，否则取 (',', ': ')。为了得到最紧凑的 JSON 表达式，你应该指定其为 (',', ':') 以消除空白字符。

在 3.4 版本发生变更: 现当 *indent* 不是 None 时，采用 (',', ': ') 作为默认值。

当 *default* 被指定时，其应该是一个函数，每当某个对象无法被序列化时它会被调用。它应该返回该对象的一个可以被 JSON 编码的版本或者引发一个*TypeError*。如果没有被指定，则会直接引发*TypeError*。

在 3.6 版本发生变更: 所有形参现在都是仅限关键字参数。

default(*o*)

在子类中实现这种方法使其返回 *o* 的可序列化对象，或者调用基础实现（引发*TypeError* ）。

例如，要支持任意的迭代器，你可以这样实现*default()*:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # 让基类的 default 方法引发 TypeError
   return super().default(o)
```

encode(*o*)

返回 Python *o* 数据结构的 JSON 字符串表达方式。例如:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

编码给定对象 *o* ，并且让每个可用的字符串表达方式。例如:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

20.2.3 异常

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)

拥有以下附加属性的*ValueError* 的子类:

msg

未格式化的错误消息。

doc

正在解析的 JSON 文档。

pos

The start index of *doc* where parsing failed.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

Added in version 3.5.

20.2.4 标准符合性和互操作性

JSON 格式由 **RFC 7159** 和 ECMA-404 进行规范说明。这一节详细介绍了本模块与 RFC 的相符程度级别。简单起见，*JSONEncoder* 和*JSONDecoder* 子类，以及明确提到的形参以外的形参，都未被纳入考量。

此模块不严格遵循于 RFC ，它实现了一些扩展是有效的 Javascript 但不是有效的 JSON。尤其是：

- 无限和 NaN 数值是被接受并输出；
- 对象内的重复名称是接受的，并且仅使用最后一对属性-值对的值。

自从 RFC 允许符合 RFC 的语法分析程序接收不符合 RFC 的输入文本以来，这个模块的解串器在默认状态下默认符合 RFC 。

字符编码

RFC 要求使用 UTF-8 ，UTF-16 ，或 UTF-32 之一来表示 JSON ，为了最大互通性推荐使用 UTF-8 。

RFC 允许，尽管不是必须的，这个模块的序列化默认设置为 *ensure_ascii=True* ，这样消除输出以便结果字符串至容纳 ASCII 字符。

ensure_ascii 参数以外，此模块是严格的按照在 Python 对象和*Unicode strings* 间的转换定义的，并且因此不能直接解决字符编码的问题。

RFC 禁止添加字符顺序标记（BOM ）在 JSON 文本的开头，这个模块的序列化器不添加 BOM 标记在它的输出上。RFC，准许 JSON 反序列化器忽略它们输入中的初始 BOM 标记，但不要求。此模块的反序列化器引发*ValueError* 当存在初始 BOM 标记。

RFC 不会明确禁止包含字节序列的 JSON 字符串这不对应有效的 Unicode 字符（比如不成对的 UTF-16 的替代物），但是它确实指出它们可能会导致互操作性问题。默认下，模块对这样的序列接受和输出（当在原始*str* 存在时）代码点。

Infinite 和 NaN 数值

RFC 不允许 infinite 或者 NaN 数值的表达方式。尽管这样，默认情况下，此模块接受并且输出 `Infinity` ，`-Infinity`，和 `NaN` 好像它们是有效的 JSON 数字字面值

```
>>> # 这些调用均不会引发异常，但结果都不是合法的 JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # 当反序列化时也一样
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

序列化器中，*allow_nan* 参数可用于替代这个行为。反序列化器中，*parse_constant* 参数，可用于替代这个行为。

对象中的重复名称

RFC 具体说明了在 JSON 对象里的名字应该是唯一的，但没有规定如何处理 JSON 对象中的重复名称。默认下，此模块不引发异常；作为替代，对于给定名它将忽略除姓-值对之外的所有对:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

The *object_pairs_hook* parameter can be used to alter this behavior.

顶级非对象，非数组值

过时的 **RFC 4627** 指定的旧版本 JSON 要求 JSON 文本顶级值必须是 JSON 对象或数组（Python *dict* 或 *list*），并且不能是 JSON null 值，布尔值，数值或者字符串值。**RFC 7159** 移除这个限制，此模块没有并且从未在序列化器和反序列化器中实现这个限制。

无论如何，为了最大化地获取互操作性，你可能希望自己遵守该原则。

实现限制

一些 JSON 反序列化器的实现应该在以下方面做出限制：

- 可接受的 JSON 文本大小
- 嵌套 JSON 对象和数组的最高水平
- JSON 数字的范围和精度
- JSON 字符串的内容和最大长度

此模块不强制执行任何上述限制，除了相关的 Python 数据类型本身或者 Python 解释器本身的限制以外。

当序列化为 JSON，在应用中当心此类限制这可能破坏你的 JSON 。特别是，通常将 JSON 数字反序列化为 IEEE 754 双精度数字，从而受到该表示方式的范围和精度限制。这是特别相关的，当序列化非常大的 Python *int* 值时，或者当序列化”exotic” 数值类型的实例时比如*decimal.Decimal* 。

20.2.5 CLI

源代码： Lib/json/tool.py

The *json.tool* module provides a simple command line interface to validate and pretty-print JSON objects.

如果未指定可选的 `infile` 和 `outfile` 参数，则将分别使用*sys.stdin* 和*sys.stdout*:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

在 3.5 版本发生变更: 输出现在将与输入顺序保持一致。请使用*--sort-keys* 选项来将输出按照键的字母顺序排序。

命令行选项

infile

要被验证或美化打印的 JSON 文件：

```
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

如果未指定 *infile*，则从*sys.stdin* 读取。

outfile
将 *infile* 输出写入到给定的 *outfile*。在其他情况下，将写入到*sys.stdout*。

--sort-keys
将字典输出按照键的字母顺序排序。

Added in version 3.5.

--no-ensure-ascii
禁用非 ASCII 字符的转义，详情参见*json.dumps()*。

Added in version 3.9.

--json-lines
将每个输入行解析为单独的 JSON 对象。

Added in version 3.8.

--indent, --tab, --no-indent, --compact
用于空白符控制的互斥选项。

Added in version 3.9.

-h, --help
显示帮助消息。

备注

20.3 mailbox --- 操纵多种格式的邮箱

源代码： Lib/mailbox.py

本模块定义了两个类，*Mailbox* 和*Message*，用于访问和操作磁盘中的邮箱及其所包含的电子邮件。Mailbox 提供了类似字典的从键到消息的映射。Message 为*email.message* 模块的*Message* 类扩展了特定格式专属的状态和行为。支持的邮箱格式有 Maildir, mbox, MH, Babyl 和 MMDF。

参见

模块*email*
表示和操作邮件消息。

20.3.1 Mailbox 对象

class mailbox.**Mailbox**
一个邮箱，它可以被检视和修改。

Mailbox 类定义了一个接口并且它不应被实例化。而是应该让格式专属的子类继承 Mailbox 并且你的代码应当实例化一个特定的子类。

Mailbox 接口类似于字典，其中每个小键都有对应的消息。键是由 Mailbox 实例发出的，它们将由实例来使用并且只对该 Mailbox 实例有意义。键会持续标识一条消息，即使对应的消息已被修改，例如被另一条消息所替代。

可以使用类似于集合的方法*add()* 将消息添加到 Mailbox 实例并使用 del 语句或类似于集合的方法*remove()* 和*discard()* 将其移除。

Mailbox 接口语义在某些值得注意的方面与字典语义有所不同。每次请求消息时，都会基于邮箱的当前状态生成一个新的表示形式（通常为*Message* 实例）。类似地，当向 Mailbox 实例添加消息时，所提供的消息表示形式的内容将被复制。无论在哪种情况下 Mailbox 实例都不会保留对消息表示形式的引用。

默认的 Mailbox *iterator* 会迭代消息表示形式，而不像默认的字典 迭代器那样迭代键。此外，在迭代期间修改邮箱是安全且有明确定义的。在创建迭代器之后被添加到邮箱的消息将对该迭代不可见。在迭代器产出消息之前从邮箱移除的消息将被静默地跳过，但是使用来自迭代器的键也有可能导致*KeyError* 异常，如果对应的消息后来被移除的话。

警告

在修改可能同时被其他某个进程修改的邮箱时要非常小心。用于此种任务的最安全邮箱格式是*Maildir*；请尽量避免使用*mbox* 之类的单文件格式进行并发写入。如果你正在修改一个邮箱，你 必须在读取文件中的任何消息或者执行添加或删除消息等修改操作 之前通过调用*lock()* 和*unlock()* 方法来锁定它。如果未锁定邮箱则将导致丢失消息或损坏整个邮箱的风险。

Mailbox 实例具有下列方法：

add(*message*)

将 *message* 添加到邮箱并返回分配给它的键。

形参 *message* 可以是*Message* 实例、*email.message.Message* 实例、字符串、字节串或文件型对象（应当以二进制模式打开）。如果 *message* 是适当的格式专属*Message* 子类的实例（举例来说，如果它是一个*mboxMessage* 实例而这是一个*mbox* 实例），将使用其格式专属的信息。在其他情况下，则会使用合理的默认值作为格式专属的信息。

在 3.2 版本发生变更: 增加了对二进制输入的支持。

remove(*key*)

__delitem__(*key*)

discard(*key*)

从邮箱中删除对应于 *key* 的消息。

当消息不存在时，如果此方法是作为*remove()* 或*__delitem__()* 调用则会引发*KeyError* 异常，而如果此方法是作为*discard()* 调用则不会引发异常。如果下层邮箱格式支持来自其他进程的并发修改则*discard()* 的行为可能是更为适合的。

__setitem__(*key*, *message*)

将 *key* 所对应的消息替换为 *message*。如果没有与 *key* 所对应的消息则会引发*KeyError* 异常。

与*add()* 一样，形参 *message* 可以是*Message* 实例、*email.message.Message* 实例、字符串、字节串或文件型对象（应当以二进制模式打开）。如果 *message* 是适当的格式专属*Message* 子类的实例（举例来说，如果它是一个*mboxMessage* 实例而这是一个*mbox* 实例），将使用其格式专属的信息。在其他情况下，当前与 *key* 所对应的消息的格式专属信息则会保持不变。

iterkeys()

返回一个迭代所有键的*iterator*

keys()

与*iterkeys()* 类似，不同之处是返回*list* 而不是*iterator*

itervalues()

__iter__()

返回一个迭代所有消息的表示形式的*iterator*。消息会被表示为适当的格式专属*Message* 子类的实例，除非当 Mailbox 实例被初始化时指定了自定义的消息工厂函数。

备注

__iter__() 的行为与字典不同，后者是对键进行迭代。

values()

与*itervalues()* 类似，不同之处是返回*list* 而不是*iterator*

iteritems()

返回一个包含 (*key*, *message*) 对的*iterator*，其中 *key* 为键而 *message* 为消息表示形式。消息会被表示为适当的格式专属*Message* 子类的实例，除非当 Mailbox 实例被初始化时指定了自定义的消息工厂函数。

items()

与*iteritems()* 类似，不同之处是返回包含键值对的*list* 而不是键值对的*iterator*。

get(*key*, *default=None*)

__getitem__(*key*)

返回对应于 *key* 的消息的表示形式。当对应的消息不存在时，如果该方法是通过*get()* 调用则返回 *default*，而如果该方法是通过 __getitem__() 调用则会引发*KeyError*。消息会被表示为适当的格式专属*Message* 子类的实例，除非当 Mailbox 被初始化时指定了自定义的消息工厂函数。

get_message(*key*)

将对应于 *key* 的消息的表示形式作为适当的格式专属*Message* 子类的实例返回，或者如果对应的消息不存在则会引发*KeyError* 异常。

get_bytes(*key*)

返回对应于 *key* 的消息的字节表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。

Added in version 3.2.

get_string(*key*)

返回对应于 *key* 的消息的字符串表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。消息是通过*email.message.Message* 处理来将其转换为纯 7bit 表示形式的。

get_file(*key*)

返回对应于 *key* 的消息的*文件类* 表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。文件型对象的行为相当于以二进制模式打开。当不再需要此文件时应当将其关闭。

在 3.2 版本发生变更: 此文件对象实际上是一个*binary file*；之前它被不正确地以文本模式返回。并且，此*file-like object* 现在还支持*context manager* 协议：你可以使用 with 语句来自动关闭它。

> **备注**
>
> 不同于其他消息表示形式，*文件类* 表示形式并不一定独立于创建它们的 Mailbox 实例或下层的邮箱。每个子类都会提供更具体的文档。

__contains__(*key*)

如果 *key* 有对应的消息则返回 True，否则返回 False。

__len__()

返回邮箱中消息的数量。

clear()

从邮箱中删除所有消息。

pop(*key*, *default=None*)

返回对应于 *key* 的消息的表示形式并删除该消息。如果对应的消息不存在则返回 *default*。消息会被表示为适当的格式专属*Message* 子类的实例，除非当 Mailbox 实例被初始化时指定的自定义的消息工厂函数。

popitem()

返回一个任意的 (*key*, *message*) 对，其中 *key* 为键而 *message* 为消息的表示形式，并删除对应的消息。如果邮箱为空，则会引发*KeyError* 异常。消息会被表示为适当的格式专属*Message* 子类的实例，除非当 Mailbox 实例被初始化时指定了自定义的消息工厂函数。

update(*arg*)

形参 *arg* 应当是 *key* 到 *message* 的映射或 (*key*, *message*) 对的可迭代对象。用来更新邮箱以使得对于每个给定的 *key* 和 *message*，与 *key* 相对应的消息会被设为 *message*，就像通过使用 *__setitem__()* 一样。类似于 *__setitem__()*，每个 *key* 都必须在邮箱中有一个对应的消息否则将会引发 *KeyError* 异常。因此在通常情况下将 *arg* 设为 Mailbox 实例是不正确的。

> **备注**
>
> 与字典不同，关键字参数是不受支持的。

flush()

将所有待定的更改写入到文件系统。对于某些 *Mailbox* 子类来说，更改总是被立即写入因而 flush() 将不做任何事，但你仍然应当养成调用此方法的习惯。

lock()

在邮箱上获取一个独占式咨询锁以使其他进程知道不能修改它。如果锁无法被获取则会引发 *ExternalClashError*。所使用的具体锁机制取决于邮箱的格式。在对邮箱内容进行任何修改之前你应当 *总是* 锁定它。

unlock()

释放邮箱上的锁，如果存在的话。

close()

刷新邮箱，如有必要则将其解锁，并关闭所有打开的文件。对于某些 Mailbox 子类来说，此方法不会做任何事。

Maildir 对象

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)

Mailbox 的一个子类，用于 Maildir 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 None，则会使用 *MaildirMessage* 作为默认的消息表示形式。如果 *create* 为 True，则当邮箱不存在时会创建它。

如果 *create* 为 True 且 *dirname* 路径存在，它将被视为已有的 maildir 而无需尝试验证其目录布局。

使用 *dirname* 这个名称而不使用 *path* 是出于历史原因。

Maildir 是一种基于目录的邮箱格式，它是针对 qmail 邮件传输代理而发明的，现在也得到了其他程序的广泛支持。Maildir 邮箱中的消息存储在一个公共目录结构中的单独文件内。这样的设计允许 Maildir 邮箱被多个彼此无关的程序访问和修改而不会导致数据损坏，因此文件锁定操作是不必要的。

Maildir 邮箱包含三个子目录，分别是: tmp, new 和 cur。消息会不时地在 tmp 子目录中创建然后移至 new 子目录来结束投递。后续电子邮件客户端可能将消息移至 cur 子目录并将有关消息状态的信息存储在附带到其文件名的特殊”info” 小节中。

Courier 邮件传输代理所引入的文件夹风格也是受支持的。主邮箱中任何子目录只要其名称的第一个字符是 '.' 就会被视为文件夹。文件夹名称会被 Maildir 表示为不带前缀 '.' 的形式。每个文件夹自身都是一个 Maildir 邮箱但不应包含其他文件夹。逻辑嵌套关系是使用 '.' 来划定层级，例如”Archived.2005.07”。

colon

Maildir 规范要求使用在特定消息文件名中使用冒号 (':')。但是，某些操作系统不允许将此字符用于文件名，如果你希望在这些操作系统上使用类似 Maildir 的格式，你应当指定改用另一个字符。叹号 ('!') 是一个受欢迎的选择。例如:

```
import mailbox
mailbox.Maildir.colon = '!'
```

colon 属性也可以在每个实例上分别设置。

在 3.13 版本发生变更: 现在*Maildir* 会忽略以点号打头的文件。

Maildir 实例具有*Mailbox* 的所有方法及下列附加方法:

list_folders()

返回所有文件夹名称的列表。

get_folder(*folder*)

返回表示名称为 *folder* 的文件夹的 Maildir 实例。如果文件夹不存在则会引发*NoSuchMailboxError* 异常。

add_folder(*folder*)

创建一个名称为 *folder* 的文件夹并返回代表它的 Maildir 实例。

remove_folder(*folder*)

删除名称为 *folder* 的文件夹。如果文件夹包含任何消息，则将引发*NotEmptyError* 异常且该文件夹将不会被删除。

clean()

从邮箱中删除最近 36 小时内未被访问过的临时文件。Maildir 规范要求邮件阅读程序应当时常进行此操作。

get_flags(*key*)

以字符串形式返回在对应于 *key* 的消息上设置的旗标。这等同于 `get_message(key).get_flags()` 但更为快速，因为它不会打开消息文件。在迭代键时可使用此方法来确定哪些消息是值得获取的。

如果你确实有一个*MaildirMessage* 对象，请改用其*get_flags()* 方法，因为由消息的*set_flags()*, *add_flag()* 和*remove_flag()* 方法所做的修改在邮箱的*__setitem__()* 方法被调用之前都不会在这里反映出来。

Added in version 3.13.

set_flags(*key*, *flags*)

在对应于 *key* 的消息上，设置由 *flags* 指定的旗标并取消设置所有其他旗标。调用 `some_mailbox.set_flags(key, flags)` 就类似于

```
one_message = some_mailbox.get_message(key)
one_message.set_flags(flags)
some_mailbox[key] = one_message
```

但更为快速，因为它不会打开消息文件。

如果你确实有一个*MaildirMessage* 对象，请改用其*set_flags()* 方法，因为用此邮箱方法所做的修改对消息对象的方法*get_flags()* 来说将不可见。

Added in version 3.13.

add_flag(*key*, *flag*)

在对应于 *key* 的消息上，设置由 *flag* 指定的旗标而不改变其他旗标。要一次性添加多个旗标，*flag* 可以是包含多个字符的字符串。

对于是使用此方法还是使用消息对象的*add_flag()* 方法的考量与*set_flags()* 类似；参见那里的讨论。

Added in version 3.13.

remove_flag(*key*, *flag*)

在对应于 *key* 的消息上，取消设置由 *flag* 指定的旗标而不改变其他旗标。要一次性移除多个旗标，*flag* 可以是包含多个字符的字符串。

对于是使用此方法还是使用消息对象的*remove_flag()* 方法的考量与*set_flags()* 类似；参见那里的讨论。

Added in version 3.13.

get_info(*key*)

以字符串形式返回包含对应于 *key* 的消息的信息的字符串。这等同于 `get_message(key).get_info()` 但更为快速，因为它不会打开消息文件。在迭代键时可使用此方法来确定哪些消息是值得获取的。

如果你确实有一个*MaildirMessage* 对象，请改用其*get_info()* 方法，因为由消息的*set_info()* 方法所做的修改在邮箱的*__setitem__()* 方法被调用之前都不会在这里反映出来。

Added in version 3.13.

set_info(*key*, *info*)

将对应于 *key* 的消息的信息设为 *info*。调用 `some_mailbox.set_info(key, flags)` 就类似于

```
one_message = some_mailbox.get_message(key)
one_message.set_info(info)
some_mailbox[key] = one_message
```

但更为快速，因为它不会打开消息文件。

如果你确实有一个*MaildirMessage* 对象，请改用其*set_info()* 方法，因为用此邮箱方法所做的修改对消息对象的方法*get_info()* 来说将不可见。

Added in version 3.13.

`Maildir` 所实现的某些*Mailbox* 方法值得进行特别说明：

add(*message*)

__setitem__(*key*, *message*)

update(*arg*)

> **警告**
>
> 这些方法会基于当前进程 ID 来生成唯一文件名。当使用多线程时，可能发生未被检测到的名称冲突并导致邮箱损坏，除非是对线程进行协调以避免使用这些方法同时操作同一个邮箱。

flush()

对 Maildir 邮箱的所有更改都会立即被应用，所以此方法并不会做任何事情。

lock()

unlock()

Maildir 邮箱不支持（或要求）锁定操作，所以此方法并不会做任何事情。

close()

`Maildir` 实例不保留任何打开的文件并且下层的邮箱不支持锁定操作。所以此方法不会做任何事情。

get_file(*key*)

根据主机平台的不同，当返回的文件保持打开状态时可能无法修改或移除下层的消息。

> **参见**
>
> **maildir man page from Courier**
> 该格式的规格说明。描述了用于支持文件夹的通用扩展。
>
> **使用 maildir 格式**
> Maildir 发明者对它的说明。包括已更新的名称创建方案和”info” 语义的相关细节。

mbox 对象

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 mbox 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 None，则会使用*mboxMessage* 作为默认的消息表示形式。如果 *create* 为 True，则当邮箱不存在时会创建它。

mbox 格式是在 Unix 系统上存储电子邮件的经典格式。mbox 邮箱中的所有消息都存储在一个单独文件中，每条消息的开头由前五个字符为"From " 的行来指明。

还有一些 mbox 格式的变种对原始格式中发现的缺点做了改进。为了保证兼容性，mbox 只实现了原始格式，或称 *mboxo* 格式。这意味着当存储消息时，如果存在 *Content-Length* 标头，它将被忽略并且消息体中出现于行开头的任何"From " 都会被转换为">From "，但是当读取消息时">From " 则不会被转换为"From "。

mbox 所实现的某些*Mailbox* 方法值得进行特别的说明：

get_bytes(*key*, *from_=False*)

注意：此方法相比其他类有一个额外形参 (*from_*)。mbox 文件条目的第一行是 Unix "From " 行。如果 *from_* 为 False，则文件的第一行将被丢弃。the first line of the file is dropped.

get_file(*key*, *from_=False*)

在 mbox 实例上调用*flush()* 或*close()* 之后再使用文件可能产生无法预料的结果或者引发异常。

注意：此方法相比其他类有一个额外形参 (*from_*)。mbox 文件条目的第一行是 Unix "From " 行。如果 *from_* 为 False，则文件的第一行将被丢弃。the first line of the file is dropped.

get_string(*key*, *from_=False*)

注意：此方法相比其他类有一个额外形参 (*from_*)。mbox 文件条目的第一行是 Unix "From " 行。如果 *from_* 为 False，则文件的第一行将被丢弃。the first line of the file is dropped.

lock()

unlock()

使用三种锁机制 --- dot 锁，以及在受支持的情况下可用的 flock() 和 lockf() 系统调用。

参见

tin 上的 mbox 指南页面
该格式的规格说明，包括有关锁的详情。

在 Unix 上配置 Netscape Mail: 为何 Content-Length 格式是不好的
使用原始 mbox 格式而非其变种的一些理由。

"mbox" 是由多个彼此不兼容的邮箱格式构成的家族
有关 mbox 变种的历史。

MH 对象

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 MH 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 None，则会使用*MHMessage* 作为默认的消息表示形式。如果 *create* 为 True，则当邮箱不存在时会创建它。

MH 是一种基于目录的邮箱格式，它是针对 MH Message Handling System 电子邮件用户代理而发明的。在 MH 邮箱的每条消息都放在单独文件中。MH 邮箱中除了邮件消息还可以包含其他 MH 邮箱 (称为 *文件夹*)。文件夹可以无限嵌套。MH 邮箱还支持 *序列*，这是一种命名列表，用来对消息进行逻辑分组而不必将其移入子文件夹。序列是在每个文件夹中名为 .mh_sequences 的文件内定义的。

MH 类可以操作 MH 邮箱，但它并不试图模拟 **mh** 的所有行为。特别地，它并不会修改 context 或 .mh_profile 文件也不会受其影响，这两个文件是 **mh** 用来存储状态和配置数据的。

MH 实例具有*Mailbox* 的所有方法及下列附加方法：

在 3.13 版本发生变更: 不包含 .mh_sequences 文件的受支持文件夹。

list_folders()

返回所有文件夹名称的列表。

get_folder(*folder*)

返回表示名称为 *folder* 的文件夹的 MH 实例。如果文件夹不存在则会引发*NoSuchMailboxError* 异常。

add_folder(*folder*)

创建名称为 *folder* 的文件夹并返回表示它的 MH 实例。

remove_folder(*folder*)

删除名称为 *folder* 的文件夹。如果文件夹包含任何消息，则将引发*NotEmptyError* 异常且该文件夹将不会被删除。

get_sequences()

返回映射到键列表的序列名称字典。如果不存在任何序列，则返回空字典。

set_sequences(*sequences*)

根据由映射到键列表的名称组成的字典 *sequences* 来重新定义邮箱中的序列，该字典与*get_sequences()* 返回值的形式一样。

pack()

根据需要重命名邮箱中的消息以消除序号中的空缺。序列列表中的条目会做相应的修改。

> **备注**
>
> 已发送的键会因此操作而失效并且不应当被继续使用。

MH 所实现的某些*Mailbox* 方法值得进行特别的说明：

remove(*key*)

__delitem__(*key*)

discard(*key*)

这些方法会立即删除消息。通过在名称前加缀一个冒号作为消息删除标记的 MH 惯例不会被使用。

lock()

unlock()

使用三种锁机制 --- dot 锁，以及在受支持的情况下可用的 flock() 和 lockf() 系统调用。对于 MH 邮箱来说，锁定邮箱意味着锁定 .mh_sequences 文件，并且仅在执行任何对它们有影响的操作期间锁定单独消息文件。

get_file(*key*)

根据主机平台的不同，当返回的文件保持打开状态时可能无法移除下层的消息。

flush()

对 MH 邮箱的所有更改都会立即被应用，所以此方法并不会做任何事情。

close()

MH 实例不会保留任何打开的文件，所以此方法等价于*unlock()*。

> **参见**
>
> **nmh - Message Handling System**
> **nmh** 的主页，这是原始 **mh** 的更新版本。

MH & nmh: Email for Users & Programmers
使用 GPL 许可证的介绍 **mh** 与 **nmh** 的图书，包含有关该邮箱格式的各种信息。

Babyl 对象

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 Babyl 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*BabylMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

Babyl 是 Rmail 电子邮箱用户代理所使用单文件邮箱格式，包括在 Emacs 中。每条消息的开头由一个包含 Control-Underscore (`'\037'`) 和 Control-L (`'\014'`) 这两个字符的行来指明。消息的结束由下一条消息的开头来指明，或者当为最后一条消息时则由一个包含 Control-Underscore (`'\037'`) 字符的行来指明。

Babyl 邮箱中的消息带有两组标头：原始标头和所谓的可见标头。可见标头通常为原始标头经过重格式化和删减以更易读的子集。Babyl 邮箱中的每条消息都附带了一个 标签列表，即记录消息相关额外信息的短字符串，邮箱中所有的用户定义标签列表会存储于 Babyl 的选项部分。

`Babyl` 实例具有*Mailbox* 的所有方法及下列附加方法：

get_labels()
返回邮箱中使用的所有用户定义标签名称的列表。

> **备注**
>
> 邮箱中存在哪些标签会通过检查实际的消息而非查询 Babyl 选项部分的标签列表，但 Babyl 选项部分会在邮箱被修改时更新。

`Babyl` 所实现的某些*Mailbox* 方法值得进行特别的说明：

get_file(*key*)
在 Babyl 邮箱中，消息的标头并不是与消息体存储在一起的。要生成文件类表示形式，标头和消息体会被一起拷贝到一个*io.BytesIO* 实例中，它具有与文件相似的 API。因此，文件型对象实际上独立于下层邮箱，但与字符串表达形式相比并不会更节省内存。

lock()

unlock()
使用三种锁机制 --- dot 锁，以及在受支持的情况下可用的 `flock()` 和 `lockf()` 系统调用。

> **参见**
>
> **Format of Version 5 Babyl Files**
> Babyl 格式的规格说明。
>
> **Reading Mail with Rmail**
> Rmail 的帮助手册，包含了有关 Babyl 语义的一些信息。

MMDF 对象

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 MMDF 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*MMDFMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

MMDF 是一种专用于电子邮件传输代理 Multichannel Memorandum Distribution Facility 的单文件邮箱格式。每条消息使用与 mbox 消息相同的形式，但其前后各有包含四个 Control-A (`'\001'`) 字符

的行。与 mbox 格式一样，每条消息的开头由一个前五个字符为"From " 的行来指明，但当存储消息时额外出现的"From " 不会被转换为">From " 因为附加的消息分隔符可防止将这些内容误认为是后续消息的开头。

MMDF 所实现的某些*Mailbox* 方法值得进行特别的说明：

get_bytes(*key*, *from_=False*)

注意：此方法相比其他类有一个额外形参 (*from_*)。mbox 文件条目的第一行是 Unix "From " 行。如果 *from_* 为 False，则文件的第一行将被丢弃。the first line of the file is dropped.

get_file(*key*, *from_=False*)

在 MMDF 实例上调用*flush()* 或*close()* 之后再使用文件可能产生无法预料的结果或者引发异常。

注意：此方法相比其他类有一个额外形参 (*from_*)。mbox 文件条目的第一行是 Unix "From " 行。如果 *from_* 为 False，则文件的第一行将被丢弃。the first line of the file is dropped.

lock()

unlock()

使用三种锁机制 --- dot 锁，以及在受支持的情况下可用的 flock() 和 lockf() 系统调用。

参见

tin 上的 mmdf 指南页面
MMDF 格式的规格说明，来自新闻阅读器 tin 的文档。

MMDF
一篇描述 Multichannel Memorandum Distribution Facility 的维基百科文章。

20.3.2 Message 对象

class mailbox.**Message**(*message=None*)

email.message 模块的*Message* 的子类。mailbox.Message 的子类添加了特定邮箱格式专属的状态和行为。

如果省略了 *message*，则新实例会以默认的空状态被创建。如果 *message* 是一个*email.message.Message* 实例，其内容会被拷贝；此外，如果 *message* 是一个 Message 实例，则任何格式专属信息会尽可能地被拷贝。如果 *message* 是一个字符串、字节串或文件，则它应当包含兼容 **RFC 2822** 的消息，该消息会被读取和解析。文件应当以二进制模式打开，但文本模式的文件也会被接受以向下兼容。

各个子类所提供的格式专属状态和行为各有不同，但总的来说只有那些不仅限于特定邮箱的特性才会被支持（虽然这些特性可能专属于特定邮箱格式）。例如，例如，单文件邮箱格式的文件偏移量和基于目录的邮箱格式的文件名都不会被保留，因为它们都仅适用于对应的原始邮箱。但消息是否已被用户读取或标记为重要等状态则会被保留，因为它们适用于消息本身。

不要求使用 Message 实例来表示使用*Mailbox* 实例所提取到的消息。在某些情况下，生成 Message 表示形式所需的时间和内存空间可能是不可接受的。对于此类情况，Mailbox 实例还提供了字符串和文件类表示形式，并可在初始化 Mailbox 实例时设置自定义的消息工厂函数。

MaildirMessage 对象

class mailbox.**MaildirMessage**(*message=None*)

具有 Maildir 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

通常，邮件用户代理应用程序会在用户第一次打开并关闭邮箱之后将 new 子目录中的所有消息移至 cur 子目录，将这些消息记录为旧消息，无论它们是否真的已被阅读。cur 下的每条消息都有一个"info" 部分被添加到其文件名中以存储有关其状态的信息。（某些邮件阅读器还会把"info" 部分也添加到 new 下的消息中。）"info" 部分可以采用两种形式之一：它可能包含"2," 后面跟一个经标准化的旗标列表（例如"2,FR"）或者它可能包含"1," 后面跟所谓的实验性信息。Maildir 消息的标准旗标如下：

旗标	含意	说明
D	草稿	正在撰写中
F	已标记	已被标记为重要
P	已检视	转发，重新发送或退回
R	已回复	回复给
S	已查看	已阅读
T	已删除	标记为可被删除

`MaildirMessage` 实例提供了下列方法：

get_subdir()

返回"new" (如果消息应当被存储在 `new` 子目录下) 或者"cur" (如果消息应当被存储在 `cur` 子目录下)。

> **备注**
>
> 一条消息通常会在其邮箱被访问后从 `new` 移至 `cur`，无论该消息是否已被阅读。如果 msg.get_flags()" 中的 `"S"` 为 ``True`` 则说明消息 `msg` 已被阅读。

set_subdir(*subdir*)

设置消息应当被存储到的子目录。形参 *subdir* 必须为"new" 或"cur"。

get_flags()

返回一个指明当前所设旗标的字符串。如果消息符合标准的 Maildir 格式，则结果为零或按字母顺序各自出现一次的 `'D'`, `'F'`, `'P'`, `'R'`, `'S'` 和 `'T'` 的拼接。如果未设任何旗标或者如果"info" 包含实验性语义则返回空字符串。

set_flags(*flags*)

设置由 *flags* 所指定的旗标并重置所有其它旗标。

add_flag(*flag*)

设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。当前"info" 会被覆盖，无论它是否只包含实验性信息而非旗标。

remove_flag(*flag*)

重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。如果"info" 包含实验性信息而非旗标，则当前的"info" 不会被修改。

get_date()

以表示 Unix 纪元秒数的浮点数形式返回消息的发送日期。

set_date(*date*)

将消息的发送日期设为 *date*，一个表示 Unix 纪元秒数的浮点数。

get_info()

返回一个包含消息的"info" 的字符串。这适用于访问和修改实验性的"info" (即不是由旗标组成的列表)。

set_info(*info*)

将"info" 设为 *info*，这应当是一个字符串。

当一个 `MaildirMessage` 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进行下列转换：

结果状态	*mboxMessage* 或*MMDFMessage* 状态
”cur” 子目录	O 旗标
F 旗标	F 旗标
R 旗标	A 旗标
S 旗标	R 旗标
T 旗标	D 旗标

当一个 MaildirMessage 实例基于*MHMessage* 实例被创建时，将进行下列转换：

结果状态	*MHMessage* 状态
”cur” 子目录	”unseen” 序列
”cur” 子目录和 S 旗标	非”unseen” 序列
F 旗标	”flagged” 序列
R 旗标	”replied” 序列

当一个 MaildirMessage 实例基于*BabylMessage* 实例被创建时，将进行下列转换：

结果状态	*BabylMessage* 状态
”cur” 子目录	”unseen” 标签
”cur” 子目录和 S 旗标	非”unseen” 标签
P 旗标	”forwarded” 或”resent” 标签
R 旗标	”answered” 标签
T 旗标	”deleted” 标签

mboxMessage 对象

class mailbox.**mboxMessage**(*message=None*)

具有 mbox 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

mbox 邮箱中的消息会一起存储在单个文件中。发件人的信封地址和发送时间通常存储在指明每条消息的起始的以”From ” 打头的行中，不过在 mbox 的各种实现之间此数据的确切格式具有相当大的差异。指明消息状态的各种旗标，例如是否已读或标记为重要等等通常存储在 *Status* 和 *X-Status* 标头中。

传统的 mbox 消息旗标如下：

旗标	含意	说明
R	已阅读	已阅读
O	旧消息	之前已经过 MUA 检测
D	已删除	标记为可被删除
F	已标记	已被标记为重要
A	已回复	回复给

”R” 和”O” 旗标存储在 *Status* 标头中，而”D”, ”F” 和”A” 旗标存储在 *X-Status* 标头中。旗标和标头通常会按上述顺序显示。

mboxMessage 实例提供了下列方法：

get_from()

返回一个表示在 mbox 邮箱中标记消息起始的”From ” 行的字符串。开头的”From ” 和末尾的换行符会被去除。

set_from(*from_*, *time_*=*None*)

将”From ” 行设为 *from_*，这应当被指定为不带开头的”From ” 或末尾的换行符。为方便起见，可以指定 *time_* 并将经过适当的格式化再添加到 *from_*。如果指定了 *time_*，它应当是一个*time.struct_time* 实例、适合传给*time.strftime()* 的元组或者 `True` (以使用*time.gmtime()*)。

get_flags()

返回一个指明当前所设旗标的字符串。如果消息符合规范格式，则结果为零或各自出现一次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按上述顺序的拼接。

set_flags(*flags*)

设置由 *flags* 所指明的旗标并重启所有其他旗标。形参 *flags* 应当为零或各自出现多次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按任意顺序的拼接。

add_flag(*flag*)

设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

remove_flag(*flag*)

重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

当一个 `mboxMessage` 实例基于*MaildirMessage* 实例被创建时，将根据*MaildirMessage* 实例的发送日期生成”From ” 行，并进行下列转换：

结果状态	*MaildirMessage* 状态
R 旗标	S 旗标
O 旗标	”cur” 子目录
D 旗标	T 旗标
F 旗标	F 旗标
A 旗标	R 旗标

当一个 `mboxMessage` 实例基于*MHMessage* 实例被创建时，将进行下列转换：

结果状态	*MHMessage* 状态
R 旗标和 O 旗标	非”unseen” 序列
O 旗标	”unseen” 序列
F 旗标	”flagged” 序列
A 旗标	”replied” 序列

当一个 `mboxMessage` 实例基于*BabylMessage* 实例被创建时，将进行下列转换：

结果状态	*BabylMessage* 状态
R 旗标和 O 旗标	非”unseen” 标签
O 旗标	”unseen” 标签
D 旗标	”deleted” 标签
A 旗标	”answered” 标签

当一个 `mboxMessage` 实例基于*MMDFMessage* 实例被创建时，”From ” 行会被拷贝并直接对应所有旗标：

结果状态	*MMDFMessage* 状态
R 旗标	R 旗标
O 旗标	O 旗标
D 旗标	D 旗标
F 旗标	F 旗标
A 旗标	A 旗标

MHMessage 对象

class mailbox.**MHMessage**(*message=None*)

具有 MH 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

MH 消息不支持传统意义上的标记或旗标，但它们支持序列，即对任意消息的逻辑分组。某些邮件阅读程序 (但不包括标准 **mh** 和 **nmh**) 以与其他格式使用旗标类似的方式来使用序列，如下所示:

序列	说明
unseen	未阅读，但之前已经过 MUA 检测
已回复	回复给
已标记	已被标记为重要

MHMessage 实例提供了下列方法:

get_sequences()

返回一个包含此消息的序列的名称的列表。

set_sequences(*sequences*)

设置包含此消息的序列的列表。

add_sequence(*sequence*)

将 *sequence* 添加到包含此消息的序列的列表。

remove_sequence(*sequence*)

将 *sequence* 从包含此消息的序列的列表中移除。

当一个 MHMessage 实例基于*MaildirMessage* 实例被创建时，将进行下列转换:

结果状态	*MaildirMessage* 状态
”unseen” 序列	非 S 旗标
”replied” 序列	R 旗标
”flagged” 序列	F 旗标

当一个 MHMessage 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进行下列转换:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
”unseen” 序列	非 R 旗标
”replied” 序列	A 旗标
”flagged” 序列	F 旗标

当一个 MHMessage 实例基于*BabylMessage* 实例被创建时，将进行下列转换:

结果状态	*BabylMessage* 状态
”unseen” 序列	”unseen” 标签
”replied” 序列	”answered” 标签

BabylMessage 对象

class mailbox.**BabylMessage**(*message=None*)

具有 Babyl 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

某些消息标签被称为 *属性*，根据惯例被定义为具有特殊的含义。这些属性如下所示:

标签	说明
unseen	未阅读，但之前已经过 MUA 检测
deleted	标记为可被删除
filed	复制到另一个文件或邮箱
answered	回复给
forwarded	已转发
edited	已被用户修改
resent	已重发

默认情况下，Rmail 只显示可见标头。不过，`BabylMessage` 类会使用原始标头因为它们更为完整。如果需要可以显式地访问可见标头。

`BabylMessage` 实例提供了下列方法：

get_labels()

返回邮件上的标签列表。

set_labels(*labels*)

将消息上的标签列表设置为 *labels* 。

add_label(*label*)

将 *label* 添加到消息上的标签列表中。

remove_label(*label*)

从消息上的标签列表中删除 *label* 。

get_visible()

返回一个*Message* 实例，其标头为消息的可见标头而其消息体为空。

set_visible(*visible*)

将消息的可见标头设为与 *message* 中的标头一致。形参 *visible* 应当是一个*Message* 实例，*email.message.Message* 实例，字符串或文件型对象（且应当以文本模式打开）。

update_visible()

当一个 `BabylMessage` 实例的原始标头被修改时，可见标头不会自动进行对应修改。此方法将按以下方式更新可见标头：每个具有对应原始标头的可见标头会被设为原始标头的值，每个没有对应原始标头的可见标头会被移除，而任何存在于原始标头但不存在于可见标头中的 *Date*, *From*, *Reply-To*, *To*, *CC* 和 *Subject* 会被添加至可见标头。

当一个 `BabylMessage` 实例基于*MaildirMessage* 实例被创建时，将进行下列转换：

结果状态	*MaildirMessage* 状态
”unseen” 标签	非 S 旗标
”deleted” 标签	T 旗标
”answered” 标签	R 旗标
”forwarded” 标签	P 旗标

当一个 `BabylMessage` 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进行下列转换：

结果状态	*mboxMessage* 或*MMDFMessage* 状态
”unseen” 标签	非 R 旗标
”deleted” 标签	D 旗标
”answered” 标签	A 旗标

当一个 `BabylMessage` 实例基于*MHMessage* 实例被创建时，将进行下列转换：

结果状态	*MHMessage* 状态
”unseen” 标签	”unseen” 序列
”answered” 标签	”replied” 序列

MMDFMessage 对象

class mailbox.**MMDFMessage**(*message=None*)

具有 MMDF 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

与 mbox 邮箱中的消息类似，MMDF 消息会与将发件人的地址和发送日期作为以”From ” 打头的初始行一起存储。同样地，指明消息状态的旗标通常存储在 *Status* 和 *X-Status* 标头中。

传统的 MMDF 消息旗标与 mbox 消息的类似，如下所示:

旗标	含意	说明
R	已阅读	已阅读
O	旧消息	之前已经过 MUA 检测
D	已删除	标记为可被删除
F	已标记	已被标记为重要
A	已回复	回复给

”R” 和”O” 旗标存储在 *Status* 标头中，而”D”, ”F” 和”A” 旗标存储在 *X-Status* 标头中。旗标和标头通常会按上述顺序显示。

MMDFMessage 实例提供了下列方法，与*mboxMessage* 所提供的类似:

get_from()

返回一个表示在 mbox 邮箱中标记消息起始的”From ” 行的字符串。开头的”From ” 和末尾的换行符会被去除。

set_from(*from_*, *time_=None*)

将”From ” 行设为 *from_*，这应当被指定为不带开头的”From ” 或末尾的换行符。为方便起见，可以指定 *time_* 并将经过适当的格式化再添加到 *from_*。如果指定了 *time_*，它应当是一个*time.struct_time* 实例、适合传给*time.strftime()* 的元组或者 True (以使用*time.gmtime()*)。

get_flags()

返回一个指明当前所设旗标的字符串。如果消息符合规范格式，则结果为零或各自出现一次的 'R', 'O', 'D', 'F' 和 'A' 按上述顺序的拼接。

set_flags(*flags*)

设置由 *flags* 所指明的旗标并重启所有其他旗标。形参 *flags* 应当为零或各自出现多次的 'R', 'O', 'D', 'F' 和 'A' 按任意顺序的拼接。

add_flag(*flag*)

设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

remove_flag(*flag*)

重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

当一个 MMDFMessage 实例基于*MaildirMessage* 实例被创建时，”From ” 行会基于*MaildirMessage* 实例的发送日期被生成，并进行下列转换:

结果状态	*MaildirMessage* 状态
R 旗标	S 旗标
O 旗标	”cur” 子目录
D 旗标	T 旗标
F 旗标	F 旗标
A 旗标	R 旗标

当一个 MMDFMessage 实例基于*MHMessage* 实例被创建时，将进行下列转换：

结果状态	*MHMessage* 状态
R 旗标和 O 旗标	非”unseen” 序列
O 旗标	”unseen” 序列
F 旗标	”flagged” 序列
A 旗标	”replied” 序列

当一个 MMDFMessage 实例基于*BabylMessage* 实例被创建时，将进行下列转换：

结果状态	*BabylMessage* 状态
R 旗标和 O 旗标	非”unseen” 标签
O 旗标	”unseen” 标签
D 旗标	”deleted” 标签
A 旗标	”answered” 标签

当一个 MMDFMessage 实例基于*mboxMessage* 实例被创建时，”From ” 行会被拷贝并直接对应所有旗标：

结果状态	*mboxMessage* 状态
R 旗标	R 旗标
O 旗标	O 旗标
D 旗标	D 旗标
F 旗标	F 旗标
A 旗标	A 旗标

20.3.3 异常

mailbox 模块中定义了下列异常类：

exception mailbox.**Error**

所有其他模块专属异常的基类。

exception mailbox.**NoSuchMailboxError**

在期望获得一个邮箱但未找到时被引发，例如当使用不存在的路径来实例化一个*Mailbox* 子类时(且将 *create* 形参设为 False)，或是当打开一个不存在的路径时。

exception mailbox.**NotEmptyError**

在期望一个邮箱为空但不为空时被引发，例如当删除一个包含消息的文件夹时。

exception mailbox.**ExternalClashError**

在某些邮箱相关条件超出了程序控制范围导致其无法继续运行时被引发，例如当要获取的锁已被另一个程序获取时，或是当要生成的唯一性文件名已存在时等等。

exception mailbox.**FormatError**

在某个文件中的数据无法被解析时被引发，例如当一个*MH* 实例尝试读取已损坏的 .mh_sequences 文件时。

20.3.4 例子

一个打印指定邮箱中所有消息的主题的简单示例:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # 可能为 None。
    if subject and 'python' in subject.lower():
        print(subject)
```

要将所有邮件从 Babyl 邮箱拷贝到 MH 邮箱，请转换所有可转换的格式专属信息:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

这个示例将来自多个邮件列表的邮件分类放入不同的邮箱，小心避免由于其他程序的并发修改导致的邮件损坏，由于程序中断导致的邮件丢失，或是由于邮箱中消息格式错误导致的意外终止:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # 消息格式错误。 不做处理。

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # 获取要使用的邮箱
            box = boxes[name]

            # 在移除原始消息之前将副本写入磁盘。
            # 如果发生崩溃，你可能重复写入消息，
            # 但那也比完全丢失这条消息要好。
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # 移除原始消息
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # 已发现目标，停止查找。

for box in boxes.itervalues():
    box.close()
```

20.4 mimetypes --- 将文件名映射到 MIME 类型

源代码: Lib/mimetypes.py

---

mimetypes 模块可以在文件名或 URL 和关联到文件扩展名的 MIME 类型之间执行转换。所提供的转换包括从文件名到 MIME 类型和从 MIME 类型到文件扩展名；后一种转换不支持编码格式。

该模块提供了一个类和一些便捷函数。这些函数是该模块通常的接口，但某些应用程序可能也会希望使用类。

下列函数提供了此模块的主要接口。如果此模块尚未被初始化，它们将会调用*init()*，如果它们依赖于*init()* 所设置的信息的话。

mimetypes.**guess_type**(*url*, *strict=True*)

根据 *url* 给出的文件名、路径或 URL 来猜测文件的类型，URL 可以为字符串或*path-like object*。

返回值是一个元组 `(type, encoding)` 其中 *type* 在无法猜测（后缀不存在或者未知）时为 `None`，或者为 `'type/subtype'` 形式的字符串，可以作为 MIME *content-type* 标头。

encoding 在无编码格式时为 `None`，或者为程序所用的编码格式 (例如 **compress** 或 **gzip**)。它可以作为 *Content-Encoding* 标头，但 **不可**作为 *Content-Transfer-Encoding* 标头。映射是表格驱动的。编码格式前缀对大小写敏感；类型前缀会先以大小写敏感方式检测再以大小写不敏感方式检测。

可选的 *strict* 参数是一个旗标，指明要将已知 MIME 类型限制在 IANA 已注册 的官方类型之内。当 *strict* 为 `True` 时（默认值），则仅支持 IANA 类型；当 *strict* 为 `False` 时，则还支持某些附加的非标准但常用的 MIME 类型。

在 3.8 版本发生变更: 增加了对于 *url* 使用*path-like object* 的支持。

自 3.13 版本弃用: 传入文件路径而非 URL 的做法已经*soft deprecated*。此种情况改用*guess_file_type()*。

mimetypes.**guess_file_type**(*path*, ***, *strict=True*)

根据由 *path* 给出的文件路径猜测其类型。类似于*guess_type()* 函数，但可接受路径而非 URL。路径可以是字符串、字节串对象或*path-like object*。

Added in version 3.13.

mimetypes.**guess_all_extensions**(*type*, *strict=True*)

根据由 *type* 给出的文件 MIME 类型猜测其扩展名。返回值是由所有可能的文件扩展名组成的字符串列表，包括开头的点号 (`'.'`)。这些扩展名不保证能关联到任何特定的数据流，但是将会由*guess_type()* 和*guess_file_type()* 映射到 MIME 类型 *type*。

可选的 *strict* 参数具有与*guess_type()* 函数一致的含义。

mimetypes.**guess_extension**(*type*, *strict=True*)

根据由 *type* 给出的文件 MIME 类型猜测其扩展名。返回值是一个表示文件扩展名的字符串，包括开头的点号 (`'.'`)。该扩展名不保证能关联到任何特定的数据流，但是将会由*guess_type()* 和*guess_file_type()* 映射到 MIME 类型 *type*。如果不能猜测出 *type* 的扩展名，则返回 `None`。

可选的 *strict* 参数具有与*guess_type()* 函数一致的含义。

有一些附加函数和数据项可被用于控制此模块的行为。

mimetypes.**init**(*files=None*)

初始化内部数据结构。*files* 如果给出则必须是一个文件名序列，它应当被用于协助默认的类型映射。如果省略则要使用的文件名会从*knownfiles* 中获取；在 Windows 上，将会载入当前注册表设置。在 *files* 或*knownfiles* 中指定的每个文件名的优先级将高于在它之前的文件名。*init()* 允许被重复调用。

为 *files* 指定一个空列表将防止应用系统默认选项：将只保留来自内置列表的常用值。

如果 *files* 为 `None` 则内部数据结构会完全重建为其初始默认值。这是一个稳定操作并将在多次调用时产生相同的结果。

在 3.2 版本发生变更: 在之前版本中，Windows 注册表设置会被忽略。

mimetypes.**read_mime_types**(*filename*)

载入在文件 *filename* 中给定的类型映射，如果文件存在的话。返回的类型映射会是一个字典，其中的键值对为文件扩展名包括开头的点号 (`'.'`) 与 `'type/subtype'` 形式的字符串。如果文件 *filename* 不存在或无法被读取，则返回 `None`。

mimetypes.**add_type**(*type*, *ext*, *strict=True*)

添加一个从 MIME 类型 *type* 到扩展名 *ext* 的映射。当扩展名已知时，新类型将替代旧类型。当类型已知时，扩展名将被添加到已知扩展名列表。

当 *strict* 为 `True` 时（默认值），映射将被添加到官方 MIME 类型，否则添加到非标准类型。

mimetypes.**inited**

指明全局数据结构是否已被初始化的旗标。这会由 *init()* 设为 `True`。

mimetypes.**knownfiles**

通常安装的类型映射文件名列表。这些文件一般被命名为 `mime.types` 并会由不同的包安装在不同的位置。

mimetypes.**suffix_map**

将后缀映射到其他后缀的字典。它被用来允许识别已编码的文件，其编码格式和类型是由相同的扩展名来指明的。例如，`.tgz` 扩展名被映射到 `.tar.gz` 以允许编码格式和类型被分别识别。

mimetypes.**encodings_map**

映射文件扩展名到编码格式类型的字典。

mimetypes.**types_map**

映射文件扩展名到 MIME 类型的字典。

mimetypes.**common_types**

映射文件扩展名到非标准但常见的 MIME 类型的字典。

此模块一个使用示例:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

20.4.1 MimeTypes 对象

MimeTypes 类可以被用于那些需要多个 MIME 类型数据库的应用程序；它提供了与 *mimetypes* 模块所提供的类似接口。

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)

这个类表示 MIME 类型数据库。默认情况下，它提供了对与此模块其余部分一致的数据库的访问权限。这个初始数据库是此模块所提供数据库的一个副本，并可以通过使用 *read()* 或 *readfp()* 方法将附加的 `mime.types` 样式文载入到数据库中来进行扩展。如果不需要默认数据的话这个映射字典也可以在载入附加数据之前先被清空。

可选的 *filenames* 形参可被用来让附加文件被载入到默认数据库“之上”。

suffix_map

将后缀映射到其他后缀的字典。它被用来允许识别已编码的文件，其编码格式和类型是由相同的扩展名来指明的。例如，`.tgz` 扩展名被映射到 `.tar.gz` 以允许编码格式和类型被分别识别。这是在模块中定义的全局 *suffix_map* 的一个副本。

encodings_map

映射文件扩展名到编码格式类型的字典。这是在模块中定义的全局*encodings_map* 的一个副本。

types_map

包含两个字典的元组，将文件扩展名映射到 MIME 类型：第一个字典针对非标准类型而第二个字典针对标准类型。它们会由*common_types* 和*types_map* 来初始化。

types_map_inv

包含两个字典的元组，将 MIME 类型映射到文件扩展名列表：第一个字典针对非标准类型而第二个字典针对标准类型。它们会由*common_types* 和*types_map* 来初始化。

guess_extension(*type*, *strict=True*)

类似于*guess_extension()* 函数，使用存储的表作为对象的一部分。

guess_type(*url*, *strict=True*)

类似于*guess_type()* 函数，使用存储的表作为对象的一部分。

guess_file_type(*path*, *, *strict=True*)

类似于*guess_file_type()* 函数，使用存储的表作为对象的一部分。

Added in version 3.13.

guess_all_extensions(*type*, *strict=True*)

类似于*guess_all_extensions()* 函数，使用存储的表作为对象的一部分。

read(*filename*, *strict=True*)

从名称为 *filename* 的文件载入 MIME 信息。此方法使用*readfp()* 来解析文件。

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

readfp(*fp*, *strict=True*)

从打开的文件 *fp* 载入 MIME 类型信息。文件必须具有标准 `mime.types` 文件的格式。

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

read_windows_registry(*strict=True*)

从 Windows 注册表载入 MIME 类型信息。

Availability: Windows.

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

Added in version 3.2.

add_type(*type*, *ext*, *strict=True*)

添加一个从 MIME 类型 *type* 到扩展名 *ext* 的映射。当扩展名已知时，新类型将替代旧类型。当类型已知时，扩展名将被添加到已知扩展名列表。

当 *strict* 为 `True` 时（默认值），映射将被添加到官方 MIME 类型，否则添加到非标准类型。

20.5 base64 --- Base16, Base32, Base64, Base85 数据编码

源代码： Lib/base64.py

此模块提供了将二进制数据编码为可打印的 ASCII 字符以及将这种编码格式解码回二进制数据的函数。它为 **RFC 4648** 所定义的 Base16, Base32 和 Base64 算法及已成为事实标准的 Ascii85 和 Base85 编码格式提供了编码和解码函数。

RFC 4648 中的编码格式适用于编码二进制数据使得它能安全地通过电子邮件发送、用作 URL 的一部分，或者包括在 HTTP POST 请求之中。此编码格式算法与 **uuencode** 程序并不相同。

此模块提供了两个接口。较新的接口支持将字节类对象 编码为 ASCII *bytes*，以及将字节类对象 或包含 ASCII 的字符串解码为*bytes*。在 **RFC 4648** 中定义的几种 base-64 字母表（普通的以及 URL 和文件系统安全的）都受到支持。

旧的接口不提供从字符串的解码操作，但提供了操作文件对象 的编码和解码函数。旧接口只支持标准的 Base64 字母表，并且按照 **RFC 2045** 的规范每 76 个字符增加一个换行符。注意：如果你需要支持 **RFC 2045**，那么使用*email* 模块可能更加合适。

在 3.3 版本发生变更: 新的接口提供的解码函数现在已经支持只包含 ASCII 的 Unicode 字符串。

在 3.4 版本发生变更: 所有类字节对象 现在已经被所有编码和解码函数接受。添加了对 Ascii85/Base85 的支持。

新的接口提供：

base64.**b64encode**(*s*, *altchars=None*)

对*bytes-like object* *s* 进行 Base64 编码，并返回编码后的*bytes*。

可选项 *altchars* 必须是一个长度为 2 的*bytes-like object*，它指定了用于替换 + 和 / 的字符表。这允许应用程序生成对 URL 或文件系统安全的 Base64 字符串。默认值为 None，即使用标准 Base64 字符表。

如果 *altchars* 的长度不为 2 则可以断言或引发*ValueError*。如果 *altchars* 不是*bytes-like object* 则会引发*TypeError*。

base64.**b64decode**(*s*, *altchars=None*, *validate=False*)

解码 Base64 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选项 *altchars* 必须是一个长度为 2 的*bytes-like object* 或 ASCII 字符串，它指定了用于替换 + 和 / 的字符表。

如果 *s* 被不正确地填写，一个*binascii.Error* 错误将被抛出。

如果 *validate* 值为 False （默认情况），则在填充检查前，将丢弃既不在标准 base-64 字母表之中也不在备用字母表中的字符。如果 *validate* 为 True，这些非 base64 字符将导致*binascii.Error*。

有关严格 base64 检查的详情，请参阅*binascii.a2b_base64()*

如果 *altchars* 不为 2 则可以断言设定或引发*ValueError*。

base64.**standard_b64encode**(*s*)

编码*bytes-like object* *s*，使用标准 Base64 字母表并返回编码过的*bytes*。

base64.**standard_b64decode**(*s*)

解码*bytes-like object* 或 ASCII 字符串 *s*，使用标准 Base64 字母表并返回编码过的*bytes*。

base64.**urlsafe_b64encode**(*s*)

编码*bytes-like object* *s*，使用 URL 与文件系统安全的字母表，使用 - 以及 _ 代替标准 Base64 字母表中的 + 和 /。返回编码过的*bytes*。结果中可能包含 =。

base64.**urlsafe_b64decode**(*s*)

解码*bytes-like object* 或 ASCII 字符串 *s*，使用 URL 与文件系统安全的字母表，使用 - 以及 _ 代替标准 Base64 字母表中的 + 和 /。返回解码过的*bytes*

base64.**b32encode**(*s*)

用 Base32 编码*bytes-like object* *s* 并返回编码过的*bytes*

base64.**b32decode**(*s*, *casefold=False*, *map01=None*)

解码 Base32 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选的 *casefold* 是一个指定小写字幕是否可接受为输入的标志。为了安全考虑，默认值为 False。

RFC 4648 允许可以选择将数码 0 (zero) 映射为字母 O (oh)，并可以选择将数码 1 (one) 映射为字母 I (eye) 或字母 L (el)。可选参数 *map01* 在不为 None 时，指定数码 1 应当映射为哪个字母 (当 *map01* 不为 None 时，数码 0 总是被映射为字母 O)。出于安全考虑其默认值为 None，因而在输入中不允许 0 和 1。

如果 *s* 被错误地填写或输入中存在字母表之外的字符，将抛出*binascii.Error*。

base64.**b32hexencode**(*s*)

类似于*b32encode()* 但是使用 Extended Hex Alphabet，如 **RFC 4648** 所定义。

Added in version 3.10.

base64.**b32hexdecode**(*s*, *casefold=False*)

类似于*b32decode()* 但是使用 Extended Hex Alphabet，如 **RFC 4648** 所定义。

这个版本不允许数字 0（零）与字母 O（oh）和数字 1（一）与字母 I（eye）或字母 L（el）的映射，所有这些字符都包含在扩展的十六进制字母表中，不能互换。

Added in version 3.10.

base64.**b16encode**(*s*)

用 Base16 编码*bytes-like object* *s* 并返回编码过的*bytes*

base64.**b16decode**(*s*, *casefold=False*)

解码 Base16 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选的 *casefold* 是一个指定小写字幕是否可接受为输入的标志。为了安全考虑，默认值为 `False`。

如果 *s* 被错误地填写或输入中存在字母表之外的字符，将抛出*binascii.Error*。

base64.**a85encode**(*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)

用 Ascii85 编码*bytes-like object* *s* 并返回编码过的*bytes*

foldspaces 是一个可选的标志，使用特殊的短序列'y' 代替'btoa' 提供的 4 个连续空格 (ASCII 0x20)。这个特性不被" 标准" Ascii85 编码支持。

wrapcol 控制输出是否应当添加换行符 (`b'\n'`)。如其为非零值，则每个输出行将只有该值所限定长度的字符数量，不包括末尾换行符。

pad 控制在编码之前输入是否填充为 4 的倍数。请注意，`btoa` 实现总是填充。

adobe 控制编码后的字节序列是否要加上 <~ 和 ~>，这是 Adobe 实现所使用的。

Added in version 3.4.

base64.**a85decode**(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\x0b'*)

解码 Ascii85 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

foldspaces 旗标指明是否应接受'y' 短序列作为 4 个连续空格 (ASCII 0x20) 的快捷方式。此特性不被" 标准" Ascii85 编码格式所支持。

adobe 控制输入序列是否为 Adobe Ascii85 格式 (即附加 <~ 和 ~>)。

ignorechars 应当是一个*bytes-like object* 或 ASCII 字符串，其中包含要从输入中忽略的字符。这应当只包含空白字符，并且默认包含 ASCII 中所有的空白字符。

Added in version 3.4.

base64.**b85encode**(*b*, *pad=False*)

用 base85（如 git 风格的二进制 diff 数据所用格式）编码*bytes-like object* *b* 并返回编码后的*bytes*。

如果 *pad* 为真值，输入将以 `b'\0'` 填充以使其编码前长度为 4 字节的倍数。

Added in version 3.4.

base64.**b85decode**(*b*)

解码 base85 编码过的*bytes-like object* 或 ASCII 字符串 *b* 并返回解码过的*bytes*。如有必要，填充会被隐式地移除。

Added in version 3.4.

base64.**z85encode**(*s*)

使用 Z85 (如在 ZeroMQ 中所使用的) 来编码*bytes-like object* *s* 并返回已编码的*bytes*。请参阅 Z85 规范说明 了解详情。

Added in version 3.13.

base64.**z85decode**(*s*)

解码 Z85 编码的*bytes-like object* 或 ASCII 字符串 *s* 并返回已解码的*bytes*。请参阅 Z85 规范说明 了解详情。

Added in version 3.13.

旧式接口:

base64.**decode**(*input*, *output*)

解码二进制 *input* 文件的内容并将结果二进制数据写入 *output* 文件。*input* 和 *output* 必须为文件对象. *input* 将被读取直至 `input.readline()` 返回空字节串对象。

base64.**decodebytes**(*s*)

解码*bytes-like object* *s*，该对象必须包含一行或多行 base64 编码的数据，并返回已解码的*bytes*。

Added in version 3.1.

base64.**encode**(*input*, *output*)

编码二进制 *input* 文件的内容并将经 base64 编码的数据写入 *output* 文件。*input* 和 *output* 必须为文件对象。*input* 将被读取直到 `input.read()` 返回空字节串对象。*encode()* 会在每输出 76 个字节之后插入一个换行符 (`b'\n'`)，并会确保输出总是以换行符来结束，如 **RFC 2045** (MIME) 所规定的那样。

base64.**encodebytes**(*s*)

编码*bytes-like object* *s*，其中可以包含任意二进制数据，并返回包含经 base64 编码数据的*bytes*，每输出 76 个字节之后将带一个换行符 (`b'\n'`)，并会确保在末尾也有一个换行符，如 **RFC 2045** (MIME) 所规定的那样。

Added in version 3.1.

此模块的一个使用示例:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

20.5.1 安全考量

在 **RFC 4648** 中新增了安全事项部分（第 12 节）；对于要部署到生产环境的任何代码都建议充分考虑此安全事项部分。

参见

模块*binascii*
支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) 第一部分：规定并描述因特网消息体的格式的机制。
第 5.2 节，“Base64 内容转换编码格式”提供了 base64 编码格式的定义。

20.6 binascii --- 在二进制数据和 ASCII 之间进行转换

binascii 模块包含多个方法用来在二进制数据和多种 ASCII 编码的二进制数据表示形式之间进行转换。在通常情况下，你不会直接使用这些函数而是使用*base64* 这样的包装器模块作为替代。*binascii* 模块包含用 C 语言编写的供这些高层级模块使用的低层级函数以获得更快的运行速度。

备注

a2b_* 函数接受只含有 ASCII 码的 Unicode 字符串。其他函数只接受字节类对象（例如*bytes*，*bytearray* 和其他支持缓冲区协议的对象）。

在 3.3 版本发生变更: ASCII-only unicode strings are now accepted by the a2b_* functions.

binascii 模块定义了以下函数：

binascii.**a2b_uu**(*string*)

将单行 uu 编码数据转换成二进制数据并返回。uu 编码每行的数据通常包含 45 个（二进制）字节，最后一行除外。每行数据后面可能跟有空格。

binascii.**b2a_uu**(*data*, *, *backtick=False*)

将二进制数据转换为 ASCII 编码字符，返回值是转换后的行数据，包括换行符。*data* 的长度最多为 45。如果 *backtick* 为 ture，则零由 '`' 而不是空格表示。

在 3.7 版本发生变更: 增加 *backtick* 形参。

binascii.**a2b_base64**(*string*, /, *, *strict_mode=False*)

将 base64 数据块转换成二进制并以二进制数据形式返回。一次可以传递多行数据。

如果 *strict_mode* 为真值，则将只转换有效的 base64 数据。无效的 base64 数据将会引发*binascii.Error*。

有效的 base64:

- 遵循 **RFC 3548**。
- 仅包含来自 base64 字符表的字符。
- 不包含填充后的额外数据（包括冗余填充、换行符等）。
- 不以填充符打头。

在 3.11 版本发生变更: 增加了 *strict_mode* 形参。

binascii.**b2a_base64**(*data*, *, *newline=True*)

将二进制数据转换为一行用 base64 编码的 ASCII 字符串。返回值是转换后的行数据，如果 *newline* 为 true，则返回值包括换行符。该函数的输出符合：rfc：*3548*。

在 3.6 版本发生变更: 增加 *newline* 形参。

binascii.**a2b_qp**(*data*, *header=False*)

将一个引号可打印的数据块转换成二进制数据并返回。一次可以转换多行。如果可选参数 *header* 存在且为 true，则数据中的下划线将被解码成空格。

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)

将二进制数据转换为一行或多行带引号可打印编码的 ASCII 字符串。返回值是转换后的行数据。如果可选参数 *quotetabs* 存在且为真值，则对所有制表符和空格进行编码。如果可选参数 *istext* 存在且为真值，则不对新行进行编码，但将对尾随空格进行编码。如果可选参数 *header* 存在且为 true，则空格将被编码为下划线 **RFC 1522**。如果可选参数 *header* 存在且为假值，则也会对换行符进行编码；不进行换行转换编码可能会破坏二进制数据流。

binascii.**crc_hqx**(*data*, *value*)

以 *value* 作为初始 CRC 计算 *data* 的 16 位 CRC 值，返回其结果。这里使用 CRC-CCITT 生成多项式 $x^{16} + x^{12} + x^5 + 1$ ，通常表示为 0x1021。该 CRC 被用于 binhex4 格式。

binascii.**crc32**(*data*[, *value*])

计算 CRC-32，即 *data* 的无符号 32 位校验和，初始 CRC 值为 *value*。默认的初始 CRC 值为零。该算法与 ZIP 文件校验和算法一致。由于该算法被设计用作校验和算法，因此不适合用作通用哈希算法。使用方式如下：

```
print(binascii.crc32(b"hello world"))
# 或者，分成两块：
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

在 3.0 版本发生变更: 结果将总是不带符号的。

binascii.**b2a_hex**(*data*[, *sep*[, *bytes_per_sep=1*]])

binascii.**hexlify**(*data*[, *sep*[, *bytes_per_sep=1*]])

返回二进制数据 *data* 的十六进制表示形式。*data* 的每个字节都被转换为相应的 2 位十六进制表示形式。因此返回的字节对象的长度是 *data* 的两倍。

使用：*bytes.hex()* 方法也可以方便地实现相似的功能（但仅返回文本字符串）。

如果指定了 *sep*，它必须为单字符 str 或 bytes 对象。它将被插入每个 *bytes_per_sep* 输入字节之后。分隔符位置默认从输出的右端开始计数，如果你希望从左端开始计数，请提供一个负的 *bytes_per_sep* 值。

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

在 3.8 版本发生变更: 添加了 *sep* 和 *bytes_per_sep* 形参。

binascii.**a2b_hex**(*hexstr*)

binascii.**unhexlify**(*hexstr*)

返回由十六进制字符串 *hexstr* 表示的二进制数据。此函数功能与*b2a_hex()* 相反。*hexstr* 必须包含偶数个十六进制数字（可以是大写或小写），否则会引发*Error* 异常。

使用：*bytes.fromhex()* 类方法也实现相似的功能（仅接受文本字符串参数，不限制其中的空白字符）。

exception binascii.**Error**

通常是因为编程错误引发的异常。

exception binascii.**Incomplete**

数据不完整引发的异常。通常不是编程错误导致的，可以通过读取更多的数据并再次尝试来处理该异常。

参见

模块*base64*
支持在 16，32，64，85 进制中进行符合 RFC 协议的 base64 样式编码。

模块*quopri*
支持在 MIME 版本电子邮件中使用引号可打印编码。

20.7 quopri --- 编码与解码 MIME 转码的可打印数据

源代码: Lib/quopri.py

此模块会执行转换后可打印的传输编码与解码，具体定义见 **RFC 1521**: "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies"。转换后可打印的编码格式被设计用于只包含相对较少的不可打印字符的数据；如果存在大量这样的字符，通过*base64* 模块所提供的 base64 编码方案会更为紧凑，例如当发送图片文件时。

quopri.**decode**(*input*, *output*, *header=False*)

解码 *input* 文件的内容并将已解码二进制数据结果写入 *output* 文件。*input* 和 *output* 必须为二进制文件对象。如果提供了可选参数 *header* 且为真值，下划线将被解码为空格。此函数可用于解码 "Q" 编码的头数据，具体描述见 **RFC 1522**: "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text"。

quopri.**encode**(*input*, *output*, *quotetabs*, *header=False*)

编码 *input* 文件的内容并将转换后可打印的数据结果写入 *output* 文件。*input* 和 *output* 必须为二进制文件对象. *quotetabs* 是一个非可选的旗标，它控制是否要编码内嵌的空格与制表符；当为真值时将编码此类内嵌空白符，当为假值时则保持原样不进行编码。请注意出现在行尾的空格与制表符总是会被编码，具体描述见 **RFC 1521**。*header* 旗标控制空格符是否要编码为下划线，具体描述见 **RFC 1522**。

quopri.**decodestring**(*s*, *header=False*)

类似*decode()*，区别在于它接受一个源*bytes* 并返回对应的已解码*bytes*。

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)

类型*encode()*，区别在于它接受一个源*bytes* 并返回对应的已编码*bytes*。在默认情况下，它会发送 `False` 值给*encode()* 函数的 *quotetabs* 形参。

参见

模块*base64*
编码与解码 MIME base64 数据

CHAPTER 21

结构化标记处理工具

Python 支持各种模块，以处理各种形式的结构化数据标记。这包括使用标准通用标记语言（SGML）和超文本标记语言（HTML）的模块，以及使用可扩展标记语言（XML）的几个接口。

21.1 `html` --- 超文本标记语言支持

源码： Lib/html/__init__.py

该模块定义了操作 HTML 的工具。

`html.escape`(*s*, *quote=True*)

将字符串 *s* 中的字符 `&` 、< 和 > 转换为安全的 HTML 序列。如果需要在 HTML 中显示可能包含此类字符的文本，请使用此选项。如果可选的标志 *quote* 为真值，则字符 (") 和 (') 也被转换；这有助于包含在由引号分隔的 HTML 属性中，如 `<a href="...">`。

Added in version 3.2.

`html.unescape`(*s*)

将字符串 *s* 中的所有命名和数字字符引用 (例如 `>`, `>`, `>`) 转换为相应的 Unicode 字符。此函数使用 HTML 5 标准为有效和无效字符引用定义的规则，以及*HTML 5 命名字符引用列表*。

Added in version 3.4.

`html` 包中的子模块是：

- *`html.parser`* ——具有宽松解析模式的 HTML / XHTML 解析器
- *`html.entities`* -- HTML 实体定义

21.2 `html.parser` --- 简单的 HTML 和 XHTML 解析器

源代码： Lib/html/parser.py

这个模块定义了一个*`HTMLParser`* 类，为 HTML（超文本标记语言）和 XHTML 文本文件解析提供基础。

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

创建一个能解析无效标记的解析器实例。

如果 *convert_charrefs* 为 `True` (默认值)，则所有字符引用 (`script`/`style` 元素中的除外) 都会自动转换为相应的 Unicode 字符。

一个*HTMLParser* 类的实例用来接受 HTML 数据，并在标记开始、标记结束、文本、注释和其他元素标记出现的时候调用对应的方法。要实现具体的行为，请使用*HTMLParser* 的子类并重写其方法。

这个解析器不检查结束标记是否与开始标记匹配，也不会因外层元素完毕而隐式关闭了的元素引发结束标记处理。

在 3.4 版本发生变更: *convert_charrefs* 关键字参数被添加。

在 3.5 版本发生变更: *convert_charrefs* 参数的默认值现在为 `True`。

21.2.1 HTML 解析器的示例程序

下面是简单的 HTML 解析器的一个基本示例，使用*HTMLParser* 类，当遇到开始标记、结束标记以及数据的时候将内容打印出来。

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

输出是:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

21.2.2 `HTMLParser` 方法

HTMLParser 实例有下列方法:

HTMLParser.**feed**(*data*)

填充一些文本到解析器中。如果包含完整的元素，则被处理；如果数据不完整，将被缓冲直到更多的数据被填充，或者*close()* 被调用。*data* 必须为*str* 类型。

HTMLParser.**close**()

如同后面跟着一个文件结束标记一样，强制处理所有缓冲数据。这个方法能被派生类重新定义，用于在输入的末尾定义附加处理，但是重定义的版本应当始终调用基类*HTMLParser* 的*close()* 方法。

HTMLParser.**reset**()

重置实例。丢失所有未处理的数据。在实例化阶段被隐式调用。

HTMLParser.**getpos**()

返回当前行号和偏移值。

HTMLParser.**get_starttag_text**()

返回最近打开的开始标记中的文本。结构化处理时通常应该不需要这个，但在处理“已部署”的 HTML 或是在以最小改变来重新生成输入时可能会有用处（例如可以保留属性间的空格等）。

下列方法将在遇到数据或者标记元素的时候被调用。他们需要在子类中重写。基类的实现中没有任何实际操作（除了*handle_startendtag()*）：

HTMLParser.**handle_starttag**(*tag*, *attrs*)

调用此方法来处理一个元素的开始标记 (例如)。

tag 参数是小写的标记名。*attrs* 参数是一个 (name, value) 形式的列表，包含了所有在标记的 <> 括号中找到的属性。*name* 转换为小写，*value* 的引号被去除，字符和实体引用都会被替换。

实例中，对于标签 <A HREF="https://www.cwi.nl/">，这个方法将以下列形式被调用 handle_starttag('a', [('href', 'https://www.cwi.nl/')])。

html.entities 中的所有实体引用，会被替换为属性值。

HTMLParser.**handle_endtag**(*tag*)

此方法被用来处理元素的结束标记（例如： ）。

tag 参数是小写的标签名。

HTMLParser.**handle_startendtag**(*tag*, *attrs*)

类似于*handle_starttag()*，只是在解析器遇到 XHTML 样式的空标记时被调用（）。这个方法能被需要这种特殊词法信息的子类重写；默认实现仅简单调用*handle_starttag()*和*handle_endtag()*。

HTMLParser.**handle_data**(*data*)

这个方法被用来处理任意数据（例如：文本节点和 <script>...</script> 以及 <style>...</style> 中的内容）。

HTMLParser.**handle_entityref**(*name*)

这个方法被用于处理 &name; 形式的命名字符引用（例如 >），其中 *name* 是通用的实体引用（例如：'gt'）。如果 *convert_charrefs* 为 True，该方法永远不会被调用。

HTMLParser.**handle_charref**(*name*)

调用该方法来处理 &#*NNN*; 和 &#x*NNN*; 形式的十进制和十六进制数字字符引用。例如，> 的等价十进制形式为 >，而十六进制形式则为 >；在这种情况下，该方法将收到 '62' 或 'x3E'。如果 *convert_charrefs* 为 True，则此方法永远不会被调用。

HTMLParser.**handle_comment**(*data*)

这个方法在遇到注释的时候被调用（例如：<!--comment-->）。

例如，<!-- comment --> 这个注释会用 ' comment ' 作为参数调用此方法。

Internet Explorer 条件注释（condcoms）的内容也被发送到这个方法，因此，对于 <!--[if IE 9]>IE9-specific content<![endif]-->，这个方法将接收到 '[if IE 9]>IE9-specific content<![endif]'。

HTMLParser.**handle_decl**(*decl*)

这个方法用来处理 HTML doctype 申明（例如 <!DOCTYPE html>）。

decl 形参为 <!...> 标记中的所有内容（例如：'DOCTYPE html'）。

HTMLParser.**handle_pi**(*data*)

此方法在遇到处理指令的时候被调用。*data* 形参将包含整个处理指令。例如，对于处理指令 <?proc color='red'>，这个方法将以 handle_pi("proc color='red'") 形式被调用。它旨在被派生类重写；基类实现中无任何实际操作。

> **备注**
>
> *HTMLParser* 类使用 SGML 语法规则处理指令。使用 '?' 结尾的 XHTML 处理指令将导致 '?' 包含在 *data* 中。

HTMLParser.**unknown_decl**(*data*)

当解析器读到无法识别的声明时，此方法被调用。

data 形参为 <![...]> 标记中的所有内容。某些时候对派生类的重写很有用。基类实现中无任何实际操作。

21.2.3 例子

下面的类实现了一个解析器，用于更多示例的演示:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

解析一个文档类型声明:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
↪strict.dtd"
```

解析一个具有一些属性和标题的元素:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
```

(续下页)

(接上页)

```
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

`script` 和 `style` 元素中的内容原样返回，无需进一步解析:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

解析注释:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

解析命名或数字形式的字符引用，并把他们转换到正确的字符（注意：这 3 种转义都是 `'>'`）：

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

填充不完整的块给*feed()* 执行，*handle_data()* 可能会多次调用（除非 *convert_charrefs* 被设置为 `True`）：

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

解析无效的 HTML (例如：未引用的属性）也能正常运行:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

21.3 html.entities --- HTML 一般实体的定义

源码： Lib/html/entities.py

该模块定义了四个词典，*html5*、*name2codepoint*、*codepoint2name*、以及*entitydefs*。

html.entities.**html5**

将 HTML5 命名字符引用[1] 映射到等效的 Unicode 字符的字典，例如 `html5['gt;'] == '>'`。请注意，尾随的分号包含在名称中（例如 `'gt;'`），但是即使没有分号，一些名称也会被标准接受，在这种情况下，名称出现时带有和不带有 `';'`。另见*html.unescape()*。

Added in version 3.3.

html.entities.**entitydefs**

将 XHTML 1.0 实体定义映射到 ISO Latin-1 中的替换文本的字典。

html.entities.**name2codepoint**

一个将 HTML4 实体名称映射到 Unicode 代码点的字典。

html.entities.**codepoint2name**

一个将 Unicode 代码点映射到 HTML4 实体名称的字典。

备注

21.4 XML 处理模块

源码： Lib/xml/

用于处理 XML 的 Python 接口分组在 `xml` 包中。

> **警告**
>
> XML 模块对于错误或恶意构造的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞* 和*defusedxml 包* 部分。

值得注意的是*xml* 包中的模块要求至少有一个 SAX 兼容的 XML 解析器可用。在 Python 中包含 Expat 解析器，因此*xml.parsers.expat* 模块将始终可用。

xml.dom 和*xml.sax* 包的文档是 DOM 和 SAX 接口的 Python 绑定的定义。

XML 处理子模块包括:

- *xml.etree.ElementTree*：ElementTree API，一个简单而轻量级的 XML 处理器
- *xml.dom*：DOM API 定义
- *xml.dom.minidom*：最小的 DOM 实现
- *xml.dom.pulldom*：支持构建部分 DOM 树
- *xml.sax*：SAX2 基类和便利函数
- *xml.parsers.expat*：Expat 解析器绑定

[1] 参见 https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references

21.4.1 XML 漏洞

XML 处理模块对于恶意构造的数据是不安全的。攻击者可能滥用 XML 功能来执行拒绝服务攻击、访问本地文件、生成与其它计算机的网络连接或绕过防火墙。

下表概述了已知的攻击以及各种模块是否容易受到攻击。

种类	sax	etree	minidom	pulldom	xmlrpc
billion laughs	**易受攻击** (1)	**易受攻击** (1)	**易受攻击** (1)	**易受攻击** (1)	**易受攻击** (1)
quadratic blowup	**易受攻击** (1)	**易受攻击** (1)	**易受攻击** (1)	**易受攻击** (1)	**易受攻击** (1)
external entity expansion	安全 (5)	安全 (2)	安全 (3)	安全 (5)	安全 (4)
DTD retrieval	安全 (5)	安全	安全	安全 (5)	安全
decompression bomb	安全	安全	安全	安全	**易受攻击**
解析大量词元	**易受攻击** (6)	**易受攻击** (6)	**易受攻击** (6)	**易受攻击** (6)	**易受攻击** (6)

1. Expat 2.4.1 及更新的版本不易受"billion laughs" 和"quadratic blowup" 漏洞的影响。因为可能要依赖系统提供的库而仍被列为易受攻击的项。请检查 `pyexpat.EXPAT_VERSION`。
2. *`xml.etree.ElementTree`* 不会扩展外部实体并将在遇到实体时引发*`ParseError`*。
3. *`xml.dom.minidom`* 不会扩展外部实体，只是简单地返回未扩展的实体。
4. *`xmlrpc.client`* 不会扩展外部实体并将忽略它们。
5. 从 Python 3.7.1 开始，默认情况下不再处理外部通用实体。
6. Expat 2.6.0 及更新的版本不易受到因解析大量词元而导致利用指数级运行时间的拒绝服务攻击。由于对系统所提供的库的潜在依赖仍会有一些项目被列为易受攻击。请检查 `pyexpat.EXPAT_VERSION`。

billion laughs / exponential entity expansion （狂笑/递归实体扩展）
Billion Laughs 攻击 -- 也称为递归实体扩展 -- 使用多级嵌套实体。每个实体多次引用另一个实体，最终实体定义包含一个小字符串。指数级扩展导致几千 GB 的文本，并消耗大量内存和 CPU 时间。

quadratic blowup entity expansion（二次爆炸实体扩展）
二次爆炸攻击类似于 Billion Laughs 攻击；它也滥用了实体扩展。它不是嵌套实体，而是一遍又一遍地重复一个具有几千个字符的大型实体。这种攻击不如递归情况有效，但它可避免触发禁止深度嵌套实体的解析器对策。

external entity expansion
实体声明可以包含的不仅仅是替换文本。它们还可以指向外部资源或本地文件。XML 解析器访问资源并将内容嵌入到 XML 文档中。

DTD retrieval
Python 的一些 XML 库*`xml.dom.pulldom`* 从远程或本地位置检索文档类型定义。该功能与外部实体扩展问题具有相似的含义。

decompression bomb
Decompression bombs（解压炸弹，又名 ZIP bomb）适用于所有可以解析压缩 XML 流（例如 gzip 压缩的 HTTP 流或 LZMA 压缩的文件）的 XML 库。对于攻击者来说，它可以将传输的数据量减少三个量级或更多。

解析大量词元
Expat 需要重新解析未完成的词元；在没有 Expat 2.6.0 所引入的防护措施的情况下，这会导致可被用来在解析 XML 的应用程序中制造拒绝服务攻击的指数级运行时间。此问题被称为 **CVE 2023-52425**。

PyPI 上 defusedxml 的文档包含关于所有已知攻击向量的更多信息并附带示例和参考资料。

21.4.2 defusedxml 包

defusedxml 是一个纯 Python 软件包，它修改了所有 stdlib XML 解析器的子类，可以防止任何潜在的恶意操作。对于解析不受信任的 XML 数据的任何服务器代码推荐使用此软件包。该软件包还附带了关于其他 XML 漏洞（如 XPath 注入）的利用示例和扩展文档。

21.5 xml.etree.ElementTree --- ElementTree XML API

源代码： Lib/xml/etree/ElementTree.py

---

xml.etree.ElementTree 模块实现了一个简单高效的 API，用于解析和创建 XML 数据。

在 3.3 版本发生变更: 此模块将在可能的情况下使用快速实现。

自 3.3 版本弃用: `xml.etree.cElementTree` 模块已被弃用。

> **警告**
>
> *xml.etree.ElementTree* 模块对于恶意构建的数据是不安全的。如果需要解析不可信或未经身份验证的数据，请参见*XML 漏洞* 。

21.5.1 教程

这是一个使用*xml.etree.ElementTree* （简称 `ET` ）的简短教程。目标是演示模块的一些构建块和基本概念。

XML 树和元素

XML 是一种继承性的分层数据格式，最自然的表示方法是使用树。为此，`ET` 有两个类 -- *ElementTree* 将整个 XML 文档表示为一个树，*Element* 表示该树中的单个节点。与整个文档的交互（读写文件）通常在*ElementTree* 级别完成。与单个 XML 元素及其子元素的交互是在*Element* 级别完成的。

解析 XML

我们将使用虚构的 `country_data.xml` XML 文档作为本节的示例数据：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

可以通过从文件中读取来导入此数据：

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

或直接从字符串中解析：

```
root = ET.fromstring(country_data_as_string)
```

fromstring() 将 XML 从字符串直接解析为*Element*，该元素是已解析树的根元素。其他解析函数可能会创建一个*ElementTree*。确切信息请查阅文档。

作为*Element*，root 具有标签和属性字典：

```
>>> root.tag
'data'
>>> root.attrib
{}
```

还有可以迭代的子节点：

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

子级是可以嵌套的，我们可以通过索引访问特定的子级节点：

```
>>> root[0][1].text
'2008'
```

备注

并非 XML 输入的所有元素都将作为解析树的元素结束。目前，此模块跳过输入中的任何 XML 注释、处理指令和文档类型声明。然而，使用这个模块的 API 而不是从 XML 文本解析构建的树可以包含注释和处理指令，生成 XML 输出时同样包含这些注释和处理指令。可以通过将自定义*TreeBuilder*实例传递给*XMLParser*构造器来访问文档类型声明。

用于非阻塞解析的拉取 API

此模块所提供了大多数解析函数都要求在返回任何结果之前一次性读取整个文档。可以使用*XMLParser*并以增量方式添加数据，但这是在回调目标上调用方法的推送式 API。有时用户真正想要的是能够以增量方式解析 XML 而无需阻塞操作，同时享受完整的已构造*Element*对象。

针对此需求的最强大工具是*XMLPullParser*。它不要求通过阻塞式读取来获得 XML 数据，而是通过执行*XMLPullParser.feed()*调用来增量式地添加数据。要获得已解析的 XML 元素，应调用*XMLPullParser.read_events()*。下面是一个示例：

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
```

(续下页)

(接上页)

```
end
mytag text= sometext more text
```

常见的用例是针对以非阻塞方式进行的应用程序，其中 XML 是从套接字接收或从某些存储设备增量式读取的。在这些用例中，阻塞式读取是不可接受的。

因为其非常灵活，*XMLPullParser* 在更简单的用例中使用起来可能并不方便。如果你不介意你的应用程序在读取 XML 数据时造成阻塞但仍希望具有增量解析能力，可以考虑*iterparse()*。它在你读取大型 XML 文档并且不希望将它完全放去内存时会很适用。

在需要通过事件获得 即时反馈的场合中，调用方法*XMLPullParser.flush()* 将有助于减少延迟；请注意研究相关的安全说明。

查找感兴趣的元素

Element 有一些很有效的方法，可帮助递归遍历其下的所有子树（包括子级，子级的子级，等等）。例如*Element.iter()*：

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() 仅查找当前元素的直接子元素中带有指定标签的元素。*Element.find()* 找带有特定标签的 第一个子级，然后可以用*Element.text* 访问元素的文本内容。*Element.get* 访问元素的属性:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

通过使用*XPath* ，可以更精确地指定要查找的元素。

修改 XML 文件

ElementTree 提供了一种构建 XML 文档并将其写入文件的简单方法。调用*ElementTree.write()* 方法就可以实现。

创建后可以直接操作*Element* 对象。例如：使用*Element.text* 修改文本字段，使用*Element.set()* 方法添加和修改属性，以及使用*Element.append()* 添加新的子元素。

假设我们要为每个国家/地区的中添加一个排名，并在排名元素中添加一个 `updated` 属性：

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

生成的 XML 现在看起来像这样：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

可以使用*Element.remove()* 删除元素。假设我们要删除排名高于 50 的所有国家/地区:

```
>>> for country in root.findall('country'):
...     # 使用 root.findall() 以避免在遍历期间执行移除
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

请注意在迭代时进行并发修改可能会导致问题，就像在迭代并修改 Python 列表或字典时那样。因此，这个示例先通过 `root.findall()` 收集了所有匹配的元素，在此之后再对匹配项列表进行迭代。

生成的 XML 现在看起来像这样:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

构建 XML 文档

SubElement() 函数还提供了一种便捷方法来为给定元素创建新的子元素:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
```

(续下页)

(接上页)

```
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

解析带有命名空间的 XML

如果 XML 输入带有 命名空间，则具有前缀的 `prefix:sometag` 形式的标记和属性将被扩展为 `{uri}sometag`，其中 *prefix* 会被完整 *URI* 所替换。并且，如果存在 默认命名空间，则完整 URI 会被添加到所有未加前缀的标记之前。

下面的 XML 示例包含两个命名空间，一个具有前缀"fictional" 而另一个则作为默认命名空间:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

搜索和探查这个 XML 示例的一种方式是手动为 *find()* 或 *findall()* 的 xpath 中的每个标记或属性添加 URI:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

一种更好的方式是搜索带命名空间的 XML 示例创建一个字典来存放你自己的前缀并在搜索函数中使用它们:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

这两种方式都会输出:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

21.5.2 XPath 支持

此模块提供了对 XPath 表达式 的有限支持用于在树中定位元素。其目标是支持一个简化语法的较小子集；完整的 XPath 引擎超出了此模块的适用范围。

示例

下面是一个演示此模块的部分 XPath 功能的例子。我们将使用来自*解析 XML* 小节的 `countrydata` XML 文档:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# 最高层级的元素
root.findall(".")

# 最高层级下的 'country' 子元素的所有 'neighbor' 孙子元素
root.findall("./country/neighbor")

# 有一个 'year' 子元素的包含 name='Singapore' 的节点
root.findall(".//year/..[@name='Singapore']")

# 是包含 name='Singapore' 的节点的子元素的 'year' 节点
root.findall(".//*[@name='Singapore']/year")

# 是其父元素的第二个子元素的所有 'neighbor' 节点
root.findall(".//neighbor[2]")
```

对于带有命名空间的 XML，应使用通常的限定 `{namespace}tag` 标记法:

```
# 文档中所有的 dublin-core "title" 标签
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

支持的 XPath 语法

语法	含意
`tag`	选择具有给定标记的所有子元素。例如，`spam` 是选择名为 `spam` 的所有子元素，而 `spam/egg` 是在名为 `spam` 的子元素中选择所有名为 `egg` 的孙元素，`{*}spam` 是在任意（或无）命名空间中选择名为 `spam` 的标记，而 `{}*` 是只选择不在一个命名空间中的标记。 在 3.8 版本发生变更: 增加了对星号通配符的支持。
`*`	选择所有子元素，包括注释和处理说明。例如 `*/egg` 选择所有名为 `egg` 的孙元素。
`.`	选择当前节点。这在路径的开头非常有用，用于指示它是相对路径。
`//`	选择所有子元素在当前元素的所有下级中选择所有下级元素。例如，`.//egg` 是在整个树中选择所有 `egg` 元素。
`..`	选择父元素。如果路径试图前往起始元素的上级（元素的 `find` 被调用）则返回 `None`。
`[@attrib]`	选择具有给定属性的所有元素。
`[@attrib='value']`	选择给定属性具有给定值的所有元素。该值不能包含引号。
`[@attrib!='value']`	选择给定属性不具有给定值的所有元素。该值不能包含引号。 Added in version 3.10.
`[tag]`	选择所有包含 `tag` 子元素的元素。只支持直系子元素。
`[.='text']`	选择完整文本内容等于 `text` 的所有元素（包括后代）。 Added in version 3.7.
`[.!='text']`	选择完整文本内容包括其下级内容不等于给定的 `text` 的所有元素。 Added in version 3.10.
`[tag='text']`	选择所有包含名为 `tag` 的子元素的元素，这些子元素（包括后代）的完整文本内容等于给定的 `text` 。
`[tag!='text']`	选择具有名为 `tag` 的子元素的所有元素，这些子元素包括其下级元素的完整文本内容不等于给定的 `text`。 Added in version 3.10.
`[position]`	选择位于给定位置的所有元素。位置可以是一个整数 (1 表示首位)，表达式 `last()` (表示末位)，或者相对于末位的位置 (例如 `last()-1`)。

谓词（方括号内的表达式）之前必须带有标签名称，星号或其他谓词。`position` 谓词前必须有标签名称。

21.5.3 参考

函数

`xml.etree.ElementTree.`**`canonicalize`**(*xml_data=None*, *, *out=None*, *from_file=None*, ***options*)

C14N 2.0 转换功能。.

规整化是标准化 XML 输出的一种方式，它允许按字节比较和使用数字签名。它降低了 XML 序列化器所具有的自由度并改为生成更受约束的 XML 表示形式。主要限制涉及命名空间声明的位置、属性的顺序和可忽略的空白符等。

此函数接受一个 XML 数字字符串 (*xml_data*) 或文件路径或者文件型对象 (*from_file*) 作为输入，将其转换为规整形式，并在提供了 *out* 文件（类）对象的情况下将其写到该对象的话，或者如果未提供则将其作为文本字符串返回。输出文件接受文本而非字节数据。因此它应当以使用 `utf-8` 编码格式的文本模式来打开。

典型使用:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(xml_data, out=out_file)
```

(续下页)

(接上页)

```
with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

配置选项 *options* 如下:

- *with_comments*: 设为真值以包括注释 (默认为假值)
- ***strip_text*: 设为真值以去除文本内容前后的空白符**
  （默认值：否）
- ***rewrite_prefixes*: 设为真值以替换带有"n{number}" 前缀的命名空间**
  （默认值：否）
- ***qname_aware_tags*: 一组可感知限定名称的标记名称，其中的前缀**
  应当在文本内容中被替换 (默认为空值)
- ***qname_aware_attrs*: 一组可感知限定名称的属性名称，其中的前缀**
  应当在文本内容中被替换 (默认为空值)
- *exclude_attrs*: 一组不应当被序列化的属性名称
- *exclude_tags*: 一组不应当被序列化的标记名称

在上面的选项列表中，" 一组" 是指任意多项集或包含字符串的可迭代对象，排序是不必要的。

Added in version 3.8.

xml.etree.ElementTree.**Comment**(*text=None*)

注释元素工厂函数。这个工厂函数可创建一个特殊元素，它将被标准序列化器当作 XML 注释来进行序列化。注释字串可以是字节串或是 Unicode 字符串。*text* 是包含注释字串的字符串。返回一个表示注释的元素实例。

请注意*XMLParser* 会跳过输入中的注释而不会为其创建注释对象。*ElementTree* 将只在当使用某个*Element* 方法向树插入了注释节点时才会包含注释节点。

xml.etree.ElementTree.**dump**(*elem*)

将一个元素树或元素结构体写入到 sys.stdout。此函数应当只被用于调试。

实际输出格式是依赖于具体实现的。在这个版本中，它将以普通 XML 文件的格式写入。

elem 是一个元素树或单独元素。

在 3.8 版本发生变更: *dump()* 函数现在会保留用户指定的属性顺序。

xml.etree.ElementTree.**fromstring**(*text*, *parser=None*)

根据一个字符串常量解析 XML 的节。与*XML()* 类似。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准*XMLParser* 解析器。返回一个*Element* 实例。

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)

根据一个字符串片段序列解析 XML 文档。*sequence* 是包含 XML 数据片段的列表或其他序列对象。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*Element* 实例。

Added in version 3.2.

xml.etree.ElementTree.**indent**(*tree*, *space=' '*, *level=0*)

添加空格到子树来实现树的缩进效果。这可以被用来生成美化打印的 XML 输出。*tree* 可以为 Element 或 ElementTree。*space* 是对应将被插入的每个缩进层级的空格字符串，默认为两个空格符。要对已缩进的树的部分子树进行缩进，请传入初始缩进层级作为 *level*。

Added in version 3.9.

xml.etree.ElementTree.**iselement**(*element*)

检测一个对象是否为有效的元素对象。*element* 是一个元素实例。如果对象是一个元素对象则返回 `True`。

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)

增量式地将一个 XML 节解析为元素树，并向用户报告执行情况。*source* 是包含 XML 数据的文件名或*file object*。*events* 是要往回报告的事件序列。受支持的事件对应字符串有 "start", "end", "comment", "pi", "start-ns" 和 "end-ns" ("ns" 事件用于获取详细的命名空间信息)。如果省略了 *events*，则只有 "end" 事件会被报告。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。*parser* 必须为*XMLParser* 的子类并且只能使用默认的*TreeBuilder* 作为目标。返回一个提供 (event, elem) 对的*iterator*；它有一个 root 属性将在 *source* 被完整读取后指向结果 XML 树的根元素。该迭代器具有 close() 方法，可在 *source* 为文件名时关闭内部文件对象。

请注意虽然*iterparse()* 是以增量方式构建树，但它会对 *source* (或其所指定的文件) 发出阻塞式读取。因此，它不适用于不可执行阻塞式读取的应用。对于完全非阻塞式的解析，请参看*XMLPullParser*。

备注

iterparse() 只会确保当发出"start" 事件时看到了开始标记的">" 字符，因而在这个点上属性已被定义，但文本容和末尾属性还未被定义。这同样适用于元素的下级；它们可能存在也可能不存在。

如果你需要已完全填充的元素，请改为查找"end" 事件。

自 3.4 版本弃用: *parser* 参数。

在 3.8 版本发生变更: 增加了 comment 和 pi 事件。

在 3.13 版本发生变更: 增加了 close() 方法。

xml.etree.ElementTree.**parse**(*source*, *parser=None*)

将一个 XML 的节解析为元素树。*source* 是包含 XML 数据的文件名或文件对象。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*ElementTree* 实例。

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)

PI 元素工厂函数。这个工厂函数可创建一个特殊元素，它将被当作 XML 处理指令来进行序列化。*target* 是包含 PI 目标的字符串。*text* 如果给出则是包含 PI 内容的字符串。返回一个表示处理指令的元素实例。

请注意*XMLParser* 会跳过输入中的处理指令而不会为其创建 PI 对象。*ElementTree* 将只在当使用某个*Element* 方法向树插入了处理指令节点时才会包含它们。

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)

注册一个命名空间前缀。这个注册表是全局的，并且任何对应给定前缀或命名空间 URI 的现有映射都会被移除。*prefix* 是命名空间前缀。*uri* 是命名空间 URI。如果可能的话，这个命名空间中的标记和属性将附带给定的前缀来进行序列化。

Added in version 3.2.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)

子元素工厂函数。这个函数会创建一个元素实例，并将其添加到现有的元素。

元素名、属性名和属性值可以是字节串或 Unicode 字符串。*parent* 是父元素。*tag* 是子元素名。*attrib* 是一个可选的字典，其中包含元素属性。*extra* 包含额外的属性，以关键字参数形式给出。返回一个元素实例。

xml.etree.ElementTree.**tostring**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

生成一个 XML 元素的字符串表示形式，包括所有子元素。*element* 是一个*Element* 实例。*encoding*[1] 是输出编码格式 (默认为 US-ASCII)。请使用 encoding="unicode" 来生成 Unicode 字符串 (否则生成字节串)。*method* 是 "xml", "html" 或 "text" (默认为 "xml")。*xml_declaration*, *default_namespace*

[1] 包括在 XML 输出中的编码格式字符串应当符合适当的标准。例如"UTF-8" 是有效的，但"UTF8" 是无效的。请参阅 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

和 *short_empty_elements* 具有与*ElementTree.write()* 中一致的含义。返回一个包含 XML 数据（可选）已编码的字符串。

在 3.4 版本发生变更: 增加了 *short_empty_elements* 形参。

在 3.8 版本发生变更: 增加了 *xml_declaration* 和 *default_namespace* 形参。

在 3.8 版本发生变更: *tostring()* 函数现在会保留用户指定的属性顺序。

xml.etree.ElementTree.**tostringlist**(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

生成一个 XML 元素的字符串表示形式，包括所有子元素。*element* 是一个*Element* 实例。*encoding*[Page 1264, 1] 是输出编码格式（默认为 US-ASCII）。请使用 `encoding="unicode"` 来生成 Unicode 字符串（否则生成字节串）。*method* 是 `"xml"`, `"html"` 或 `"text"` (默认为 `"xml"`)。*xml_declaration*, *default_namespace* 和 *short_empty_elements* 具有与*ElementTree.write()* 中一致的含义。返回一个包含 XML 数据（可选）已编码字符串的列表。它并不保证任何特定的序列，除了 `b"".join(tostringlist(element)) == tostring(element)`。

Added in version 3.2.

在 3.4 版本发生变更: 增加了 *short_empty_elements* 形参。

在 3.8 版本发生变更: 增加了 *xml_declaration* 和 *default_namespace* 形参。

在 3.8 版本发生变更: *tostringlist()* 函数现在会保留用户指定的属性顺序。

xml.etree.ElementTree.**XML**(*text*, *parser=None*)

根据一个字符串常量解析 XML 的节。此函数可被用于在 Python 代码中嵌入“XML 字面值”。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*Element* 实例。

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)

根据一个字符串常量解析 XML 的节，并且还将返回一个将元素的 id:s 映射到元素的字典。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个包含*Element* 实例和字典的元组。

21.5.4 XInclude 支持

此模块通过*xml.etree.ElementInclude* 辅助模块提供了对 XInclude 指令 的有限支持，这个模块可被用来根据元素树的信息在其中插入子树和文本字符串。

示例

以下是一个演示 XInclude 模块用法的例子。要在当前文本中包括一个 XML 文档，请使用 `{http://www.w3.org/2001/XInclude}include` 元素并将 **parse** 属性设为 `"xml"`，并使用 **href** 属性来指定要包括的文档。

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

默认情况下，**href** 属性会被当作文件名来处理。你可以使用自定义加载器来覆盖此行为。还要注意标准辅助器不支持 XPointer 语法。

要处理这个文件，请正常加载它，并将根元素传给*xml.etree.ElementTree* 模块:

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

ElementInclude 模块使用来自 **source.xml** 文档的根元素替代 `{http://www.w3.org/2001/XInclude}include` 元素。结果看起来大概是这样:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

如果省略了 **parse** 属性，它会取默认的"xml"。要求有 href 属性。

要包括文本文档，请使用 `{http://www.w3.org/2001/XInclude}include` 元素，并将 **parse** 属性设为"text":

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

结果可能如下所示:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

21.5.5 参考

函数

`xml.etree.ElementInclude.`**`default_loader`**(*href*, *parse*, *encoding=None*)

默认的加载器。这个默认的加载器会从磁盘读取所包括的资源。*href* 是一个 URL。*parse* 是取值为"xml" 或"text" 的解析模式。*encoding* 是可选的文本编码格式。如果未给出，则编码格式为 `utf-8`。返回已扩展的资源。如果解析模式为 `"xml"`，则它是一个*Element* 实例。如果解析模式为 `"text"`，则它是一个字符串。如果加载器失败，它可以返回 `None` 或者引发异常。

`xml.etree.ElementInclude.`**`include`**(*elem*, *loader=None*, *base_url=None*, *max_depth=6*)

这个函数会在 *elem* 指向的树上原地扩展 XInclude 指令。*elem* 是根*Element* 或用于查找相应元素的*ElementTree* 实例。*loader* 是可选的资源加载器。如果省略，则它默认为*default_loader()*。如果给出，则它应当是一个实现了与*default_loader()* 相同接口的可调用对象。*base_url* 是原始文件的基准 URL, 用于求解相对的包括文件引用。*max_depth* 是递归包括的最大数量。此限制是为了减少恶意内容爆破的风险。传入 `None` 可禁用此限制。

在 3.9 版本发生变更: 增加了 *base_url* 和 *max_depth* 形参。

元素对象

`class` `xml.etree.ElementTree.`**`Element`**(*tag*, *attrib={}*, ***extra*)

元素类。这个类定义了 Element 接口，并提供了这个接口的引用实现。

元素名、属性名和属性值可以是字节串或 Unicode 字符串。*tag* 是元素名。*attrib* 是一个可选的字典，其中包含元素属性。*extra* 包含额外的属性，以关键字参数形式给出。

`tag`

一个标识此元素意味着何种数据的字符串 (换句话说，元素类型)。

`text`

`tail`

这些属性可被用于存放与元素相关联的额外数据。它们的值通常为字符串但也可以是任何应用专属的对象。如果元素是基于 XML 文件创建的，*text* 属性会存放元素的开始标记及其第一个子元素或结束标记之间的文本，或者为 `None`，而 *tail* 属性会存放元素的结束标记及下一个标记之间的文本，或者为 `None`。对于 XML 数据

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

a 元素的 *text* 和 *tail* 属性均为 `None`，*b* 元素的 *text* 为 `"1"` 而 *tail* 为 `"4"`，*c* 元素的 *text* 为 `"2"` 而 *tail* 为 `None`，*d* 元素的 *text* 为 `None` 而 *tail* 为 `"3"`。

要获取一个元素的内部文本，请参阅*itertext()*，例如 `"".join(element.itertext())`。

应用程序可以将任意对象存入这些属性。

attrib

一个包含元素属性的字典。请注意虽然 *attrib* 值总是一个真正可变的 Python 字典，但 ElementTree 实现可以选择其他内部表示形式，并只在有需要时才创建字典。为了发挥这种实现的优势，请在任何可能情况下使用下列字典方法。

以下字典类方法作用于元素属性。

clear()

重设一个元素。此方法会移除所有子元素，清空所有属性，并将 text 和 tail 属性设为 `None`。

get(*key*, *default=None*)

获取名为 *key* 的元素属性。

返回属性的值，或者如果属性未找到则返回 *default*。

items()

将元素属性以 (name, value) 对序列的形式返回。所返回属性的顺序任意。

keys()

将元素属性名称以列表的形式返回。所返回名称的顺序任意。

set(*key*, *value*)

将元素的 *key* 属性设为 *value*。

以下方法作用于元素的下级（子元素）。

append(*subelement*)

将元素 *subelement* 添加到此元素的子元素内部列表。如果 *subelement* 不是一个*Element* 则会引发*TypeError*。

extend(*subelements*)

基于一个包含多个元素的可迭代对象添加 *subelements*。如果某个子元素不是*Element* 则会引发*TypeError*。

Added in version 3.2.

find(*match*, *namespaces=None*)

查找第一个匹配 *match* 的子元素。*match* 可以是一个标记名称或者路径。返回一个元素实例或 `None`。*namespaces* 是可选的从命名空间前缀到完整名称的映射。传入 `''` 作为前缀可将表达式中所有无前缀的标记名称移动到给定的命名空间。

findall(*match*, *namespaces=None*)

根据标记名称或者路径 查找所有匹配的子元素。返回一个包含所有匹配元素按文档顺序排序的列表。*namespaces* 是可选的从命名空间前缀到完整名称的映射。传入 `''` 作为前缀可将表达式中所有无前缀的标记名称移动到给定的命名空间。

findtext(*match*, *default=None*, *namespaces=None*)

查找第一个匹配 *match* 的子元素的文本。*match* 可以是一个标记名称或者路径。反回第一个匹配的元素的文本内容，或者如果元素未找到则返回 *default*。请注意如果匹配的元素没有文本内容则会返回一个空字符串。*namespaces* 是可选的从命名空间前缀到完整名称的映射。传入 `''` 作为前缀可将表达式中所有无前缀的标记名称移动到给定的命名空间。

insert(*index*, *subelement*)

将 *subelement* 插入到此元素的给定位置中。如果 *subelement* 不是一个*Element* 则会引发*TypeError*。

iter(*tag=None*)

创建一个以当前元素为根元素的树的*iterator*。该迭代器将以文档（深度优先）顺序迭代此元素及其所有下级元素。如果 *tag* 不为 `None` 或 `'*'`，则迭代器只返回标记为 *tag* 的元素。如果树结构在迭代期间被修改，则结果是未定义的。

Added in version 3.2.

iterfind(*match*, *namespaces=None*)

根据标记名称或者路径 查找所有匹配的子元素。返回一个按文档顺序产生所有匹配元素的可迭代对象。*namespaces* 是可选的从命名空间前缀到完整名称的映射。

Added in version 3.2.

itertext()

创建一个文本迭代器。该迭代器将按文档顺序遍历此元素及其所有子元素，并返回所有内部文本。

Added in version 3.2.

makeelement(*tag*, *attrib*)

创建一个与此元素类型相同的新元素对象。请不要调用此方法，而应改用*SubElement()* 工厂函数。

remove(*subelement*)

从元素中移除 *subelement*。与 find* 方法不同的是此方法会基于实例的标识来比较元素，而不是基于标记的值或内容。

Element 对象还支持下列序列类型方法以配合子元素使用: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`。

注意：没有子元素的元素测试结果将为 `False`。在未来的 Python 发布版中，所有元素不论是否存在子元素测试结果都将为 `True`。建议改用显式的 `len(elem)` 或 `elem is not None` 测试。:

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

在 3.12 版本发生变更: 检测元素真值将引发*DeprecationWarning*。

在 Python 3.8 之前，元素的 XML 属性的序列化顺序会通过按其名称排序来强制使其可被预期。由于现在字典已保证是有序的，这个强制重排序在 Python 3.8 中已被移除以保留原本由用户代码解析或创建的属性顺序。

通常，用户代码应当尽量不依赖于特定的属性顺序，因为 XML 信息设定 明确地排除了用属性顺序转递信息的做法。代码应当准备好处理任何输入顺序。对于要求确定性的 XML 输出的情况，例如加密签名或检测数据集等，可以通过规范化*canonicalize()* 函数来进行传统的序列化。

对于规范化输出不可用但仍然要求输出特定属性顺序的情况，代码应当设法直接按要求的顺序来创建属性，以避免代码阅读者产生不匹配的感觉。如果这一点是难以做到的，可以在序列化之前应用以下写法来强制实现顺序不依赖于元素的创建:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # 调整属性顺序，例如通过排序操作
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

ElementTree 对象

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree 包装器类。这个类表示一个完整的元素层级结构，并添加了一些对于标准 XML 序列化的额外支持。

element 是根元素。如果给出 XML *file* 则将使用其内容来初始化树结构。

_setroot(*element*)

替换该树结构的根元素。这将丢弃该树结构的当前内容，并将其替换为给定的元素。请小心使用。*element* 是一个元素实例。

find(*match*, *namespaces=None*)

与*Element.find()* 类似，从树的根节点开始。

findall(*match*, *namespaces=None*)

与*Element.findall()* 类似，从树的根节点开始。

findtext(*match*, *default=None*, *namespaces=None*)

与*Element.findtext()* 类似，从树的根节点开始。

getroot()

返回这个树的根元素。

iter(*tag=None*)

创建并返回根元素的树结构迭代器。该迭代器会以节顺序遍历这个树的所有元素。*tag* 是要查找的标记（默认返回所有元素）。

iterfind(*match*, *namespaces=None*)

与*Element.iterfind()* 类似，从树的根节点开始。

Added in version 3.2.

parse(*source*, *parser=None*)

将一个外部 XML 节载入到此元素树。*source* 是一个文件名或*file object*。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回该节的根元素。

write(*file*, *encoding='us-ascii'*, *xml_declaration=None*, *default_namespace=None*, *method='xml'*, ***, *short_empty_elements=True*)

将元素树以 XML 格式写入到文件。*file* 为文件名，或是以写入模式打开的*file object*。*encoding*[Page 1264, 1] 为输出编码格式 (默认为 US-ASCII)。*xml_declaration* 控制是否要将 XML 声明添加到文件中。使用 `False` 表示从不添加，`True` 表示总是添加，`None` 表示仅在非 US-ASCII 或 UTF-8 或 Unicode 时添加 (默认为 `None`)。*default_namespace* 设置默认 XML 命名空间 (用于"xmlns")。*method* 为 `"xml"`, `"html"` 或 `"text"` (默认为 `"xml"`)。仅限关键字形参 *short_empty_elements* 控制不包含内容的元素的格式。如为 `True` (默认值)，它们会被输出为单个自结束标记，否则它们会被输出为一对开始/结束标记。

输出是一个字符串 (*str*) 或字节串 (*bytes*)。由 *encoding* 参数来控制。如果 *encoding* 为 `"unicode"`，则输出是一个字符串；否则为字节串；请注意这可能与 *file* 的类型相冲突，如果它是一个打开的*file object* 的话；请确保你不会试图写入字符串到二进制流或者反向操作。

在 3.4 版本发生变更: 增加了 *short_empty_elements* 形参。

在 3.8 版本发生变更: *write()* 方法现在会保留用户指定的属性顺序。

这是将要被操作的 XML 文件:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
```

(续下页)

(接上页)

```
    </body>
</html>
```

修改第一段中的每个链接的"target" 属性的示例:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # 在 body 中找到首次出现的标签 p
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # 返回所有链接的列表
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:              # 迭代所有已找到的链接
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName 对象

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName 包装器。这可被用来包装 QName 属性值，以便在输出中获得适当的命名空间处理。*text_or_uri* 是一个包含 QName 值的字符串，其形式为 {uri}local，或者如果给出了 tag 参数，则为 QName 的 URI 部分。如果给出了 *tag*，则第一个参数会被解读为 URI，而这个参数会被解读为本地名称。*QName* 实例是不透明的。

TreeBuilder 对象

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, *, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

通用元素结构构建器。此构建器会将包含 start, data, end, comment 和 pi 方法调用的序列转换为格式良好的元素结构。你可以通过这个类使用一个自定义 XML 解析器或其他 XML 类格式的解析器来构建元素结构。

如果给出 *element_factory*，它必须为接受两个位置参数的可调用对象：一个标记和一个属性字典。它预期会返回一个新的元素实例。

如果给出 *comment_factory* 和 *pi_factory* 函数，它们的行为应当像*Comment()* 和*ProcessingInstruction()* 函数一样创建注释和处理指令。如果未给出，则将使用默认工厂函数。当 *insert_comments* 和/或 *insert_pis* 为真值时，如果 comments/pis 在根元素之中（但不在其之外）出现则它们将被插入到树中。

close()

刷新构建器缓存，并返回最高层级的文档元素。返回一个*Element* 实例。

data(*data*)

将文本添加到当前元素。*data* 为要添加的文本。这应当是一个字节串或 Unicode 字符串。

end(*tag*)

关闭当前元素。*tag* 是元素名称。返回已关闭的元素。

start(*tag*, *attrs*)

打开一个新元素。*tag* 是元素名称。*attrs* 是包含元素属性的字典。返回打开的元素。

comment(*text*)

使用给定的 *text* 创建一条注释。如果 `insert_comments` 为真值，这还会将其添加到树结构中。

Added in version 3.8.

pi(*target*, *text*)

使用给定的 *target* 名称和 *text* 创建一条处理指令。如果 `insert_pis` 为真值，这还会将其添加到树中。

Added in version 3.8.

此外，自定义的*TreeBuilder* 对象还提供了以下方法:

doctype(*name*, *pubid*, *system*)

处理一条 doctype 声明。*name* 为 doctype 名称。*pubid* 为公有标识。*system* 为系统标识。此方法不存在于默认的*TreeBuilder* 类中。

Added in version 3.2.

start_ns(*prefix*, *uri*)

在定义了 `start()` 回调的打开元素的该回调被调用之前，当解析器遇到新的命名空间声明时都会被调用。*prefix* 对于默认命名空间为 `''` 或者在其他情况下为被声明的命名空间前缀名称。*uri* 是命名空间 URI。

Added in version 3.8.

end_ns(*prefix*)

在声明了命名空间前缀映射的元素的 `end()` 回调之后被调用，附带超出作用域的 *prefix* 的名称。

Added in version 3.8.

class xml.etree.ElementTree.**C14NWriterTarget**(*write*, *, *with_comments=False*, *strip_text=False*, *rewrite_prefixes=False*, *qname_aware_tags=None*, *qname_aware_attrs=None*, *exclude_attrs=None*, *exclude_tags=None*)

C14N 2.0 写入器。其参数与*canonicalize()* 函数的相同。这个类并不会构建树结构而是使用 *write* 函数将回调事件直接转换为序列化形式。

Added in version 3.8.

XMLParser 对象

class xml.etree.ElementTree.**XMLParser**(*, *target=None*, *encoding=None*)

这个类是此模块的低层级构建单元。它使用*xml.parsers.expat* 来实现高效、基于事件的 XML 解析。它可以通过*feed()* 方法增量式地收受 XML 数据，并且解析事件会被转换为推送式 API —— 通过在 *target* 对象上发起对回调的调用。如果省略 *target*，则会使用标准的*TreeBuilder*。如果给出了 *encoding*[Page 1264, 1] ，该值将覆盖在 XML 文件中指定的编码格式。

在 3.8 版本发生变更: 形参现在都是仅限关键字形参。*html* 参数不再受支持。argument is no longer supported.

close()

结束向解析器提供数据。返回调用在构造期间传入的 *target* 的 `close()` 方法的结果；在默认情况下，这是最高层级的文档元素。

feed(*data*)

将数据送入解析器。*data* 是编码后的数据。

flush()

触发对之前送入的未解析数据的解析，这可被用于确保更为实时的反馈，尤其是对于 Expat >=2.6.0 的情况。*flush()* 的实现会暂时禁用 Expat 的重新解析延迟（如果当前已启用）并触发重新解析。禁用重新解析延迟会带来安全性的影响；请参阅*xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* 了解详情。

请注意*flush()* 已作为安全修正被向下移植到一些较早的 CPython 发布版。如果在运行于多个 Python 版本的代码中要用到*flush()* 请使用*hasattr()* 来检查其可用性。

Added in version 3.13.

XMLParser.feed() 会为每个打开的标记调用 *target* 的 `start(tag, attrs_dict)` 方法，为每个关闭的标记调用它的 `end(tag)` 方法，并通过 `data(data)` 方法来处理数据。有关更多受支持的回调方法，请参阅*TreeBuilder* 类。*XMLParser.close()* 会调用 *target* 的 `close()` 方法。*XMLParser* 不仅仅可被用来构建树结构。下面是一个统计 XML 文件最大深度的示例:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # 解析器的目标对象
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # 针对每个开启标签执行调用。
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # 针对每个关闭标签执行调用。
...         self.depth -= 1
...     def data(self, data):
...         pass            # 我们不需要对 data 做任何操作。
...     def close(self):    # 当所有数据都已被解析时执行调用。
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser 对象

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

适用于非阻塞应用程序的拉取式解析器。它的输入侧 API 与*XMLParser* 的类似，但不是向回调目标推送调用，*XMLPullParser* 会收集一个解析事件的内部列表并让用户来读取它。*events* 是要报告的事件序列。受支持的事件字符串有 `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` 和 `"end-ns"` ("ns" 事件被用于获取详细的命名空间信息)。如果 *events* 被省略，则只报告 `"end"` 事件。

feed(*data*)

将给定的字节数据送入解析器。

flush()

触发对之前送入的未解析数据的解析，这可被用于确保更为实时的反馈，尤其是对于 Expat >=2.6.0 的情况。*flush()* 的实现会暂时禁用 Expat 的重新解析延迟（如果当前已启用）并触发重新解析。禁用重新解析延迟会带来安全性的影响；请参阅*xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* 了解详情。

请注意*flush()* 已作为安全修正被向下移植到一些较早的 CPython 发布版。如果在运行于多个 Python 版本的代码中要用到*flush()* 请使用*hasattr()* 来检查其可用性。

Added in version 3.13.

close()

通知解析器数据流已终结。不同于*XMLParser.close()*，此方法总是返回*None*。当解析器被关闭时任何还未被获取的事件仍可通过*read_events()* 被读取。

read_events()

返回包含在送入解析器的数据中遇到的事件的迭代器。此迭代器会产生 (event, elem) 对，其中 *event* 是代表事件类型的字符串 (例如 "end") 而 *elem* 是遇到的 *Element* 对象，或者以下的其他上下文值。

- start, end: 当前元素。
- comment, pi: 当前注释 / 处理指令
- start-ns: 一个指定所声明命名空间映射的元组 (prefix, uri)。
- end-ns: *None* (这可能在未来版本中改变)

在之前对 *read_events()* 的调用中提供的事件将不会被再次产生。事件仅当它们从迭代器中被取出时才会在内部队列中被消费，因此多个读取方对获取自 *read_events()* 的迭代器进行平行迭代将产生无法预料的结果。

备注

XMLPullParser 只会确保当发出"start" 事件时看到了开始标记的">" 字符，因而在这个点上属性已被定义，但文本内容和末尾属性还未被定义。这同样适用于元素的下级；它们可能存在也可能不存在。

如果你需要已完全填充的元素，请改为查找"end" 事件。

Added in version 3.4.

在 3.8 版本发生变更: 增加了 comment 和 pi 事件。

异常

class xml.etree.ElementTree.**ParseError**

XML 解析器错误，由此模块中的多个解析方法在解析失败时引发。此异常的实例的字符串表示将包含用户友好的错误消息。此外，它将具有下列可用属性:

code

来自外部解析器的数字错误代码。请参阅 *xml.parsers.expat* 的文档查看错误代码列表及它们的含义。

position

一个包含 *line*, *column* 数值的元组，指明错误发生的位置。

备注

21.6 xml.dom --- 文档对象模型 API

源代码: Lib/xml/dom/__init__.py

文档对象模型 "DOM" 是一个来自万维网联盟（W3C）的跨语言 API，用于访问和修改 XML 文档。DOM 的实现将 XML 文档以树结构表示，或者允许客户端代码从头构建这样的结构。然后它会通过一组提供通用接口的对象赋予对结构的访问权。

DOM 特别适用于进行随机访问的应用。SAX 仅允许你每次查看文档的一小部分。如果你正在查看一个 SAX 元素，你将不能访问其他元素。如果你正在查看一个文本节点，你将不能访问包含它的元素。当你编写一个 SAX 应用时，你需要在你自己的代码的某个地方记住你的程序在文档中的位置。SAX 不会帮你做这件事。并且，如果你想要在 XML 文档中向前查看，你是绝对办不到的。

有些应用程序在不能访问树的事件驱动模型中是根本无法编写的。当然你可以在 SAX 事件中自行构建某种树，但是 DOM 可以使你避免编写这样的代码。DOM 是针对 XML 数据的标准树表示形式。

文档对象模型是由 W3C 分阶段定义的，在其术语中称为“层级”。Python 中该 API 的映射大致是基于 DOM 第 2 层级的建议。

DOM 应用程序通常从将某些 XML 解析为 DOM 开始。此操作如何实现完全未被 DOM 第 1 层级所涉及，而第 2 层级也只提供了有限的改进：有一个 `DOMImplementation` 对象类，它提供对 `Document` 创建方法的访问，但却没有办法以不依赖具体实现的方式访问 XML 读取器/解析器/文档创建器。也没有当不存在 `Document` 对象的情况下访问这些方法的定义良好的方式。在 Python 中，每个 DOM 实现将提供一个函数 *getDOMImplementation()*。DOM 第 3 层级增加了一个载入/存储规格说明，它定义了与读取器的接口，但这在 Python 标准库中尚不可用。

一旦你得到了 DOM 文档对象，你就可以通过 XML 文档的属性和方法访问它的各个部分。这些属性定义在 DOM 规格说明当中；参考指南的这一部分描述了 Python 对此规格说明的解读。

W3C 提供的规格说明定义了适用于 Java, ECMAScript 和 OMG IDL 的 DOM API。这里定义的 Python 映射很大程度上是基于此规格说明的 IDL 版本，但并不要求严格映射（但具体实现可以自由地支持对 IDL 的严格映射）。请参阅一致性 一节查看有关映射要求的详细讨论。

参见

文档对象模型 (DOM) 第 2 层级规格说明
被 Python DOM API 作为基础的 W3C 建议。

文档对象模型 (DOM) 第 1 层级规格说明
被 *xml.dom.minidom* 所支持的 W3C 针对 DOM 的建议。

Python 语言映射规格说明
此文档指明了从 OMG IDL 到 Python 的映射。

21.6.1 模块内容

xml.dom 包含下列函数:

xml.dom.**registerDOMImplementation**(*name*, *factory*)

注册 *factory* 函数并使用名称 *name*。该工厂函数应当返回一个实现了 `DOMImplementation` 接口的对象。该工厂函数可每次都返回相同对象，或每次调用都返回新的对象，视具体实现的要求而定（例如该实现是否支持某些定制功能）。

xml.dom.**getDOMImplementation**(*name=None*, *features=()*)

返回一个适当的 DOM 实现。*name* 是通用名称、DOM 实现的模块名称或者 `None`。如果它不为 `None`，则会导入相应模块并在导入成功时返回一个 `DOMImplementation` 对象。如果没有给出名称，并且如果设置了 `PYTHON_DOM` 环境变量，此变量会被用来查找相应的实现。

如果未给出 name，此函数会检查可用的实现来查找具有所需特性集的一个。如果找不到任何实现，则会引发 *ImportError*。features 集必须是包含 `(feature, version)` 对的序列，它会被传给可用的 `DOMImplementation` 对象上的 `hasFeature()` 方法。

还提供了一些便捷常量:

xml.dom.**EMPTY_NAMESPACE**

该值用于指明没有命名空间被关联到 DOM 中的某个节点。它通常被作为某个节点的 `namespaceURI`，或者被用作某个命名空间专属方法的 *namespaceURI* 参数。

xml.dom.**XML_NAMESPACE**

关联到保留前缀 `xml` 的命名空间 URI，如 XML 中的命名空间 （第 4 节）所定义的。

xml.dom.**XMLNS_NAMESPACE**

命名空间声明的命名空间 URI，如 文档对象模型 (DOM) 第 2 层级核心规格说明 (第 1.1.8 节) 所定义的。

xml.dom.**XHTML_NAMESPACE**

XHTML 命名空间的 URI，如 XHTML 1.0: 扩展超文本标记语言 (第 3.1.1 节) 所定义的。

此外，*xml.dom* 还包含一个基本 Node 类和一些 DOM 异常类。此模块提供的 Node 类未实现 DOM 规格描述所定义的任何方法和属性；实际的 DOM 实现必须提供它们。提供 Node 类作为此模块的一部分并没有提供用于实际的 Node 对象的 nodeType 属性的常量；它们是位于类内而不是位于模块层级以符合 DOM 规格描述。

21.6.2 DOM 中的对象

DOM 的权威文档是来自 W3C 的 DOM 规格描述。

请注意，DOM 属性也可以作为节点而不是简单的字符串进行操作。然而，必须这样做的情况相当少见，所以这种用法还没有被写入文档。

接口	部件	目的
DOMImplementation	*DOMImplementation* 对象	底层实现的接口。
Node	节点对象	文档中大多数对象的基本接口。
NodeList	节点列表对象	节点序列的接口。
DocumentType	文档类型对象	有关处理文档所需声明的信息。
Document	*Document* 对象	表示整个文档的对象。
Element	元素对象	文档层次结构中的元素节点。
Attr	*Attr* 对象	元素节点上的属性值节点。
Comment	注释对象	源文档中注释的表示形式。
Text	*Text* 和 *CDATASection* 对象	包含文档中文本内容的节点。
ProcessingInstruction	*ProcessingInstruction* 对象	处理指令表示形式。

描述在 Python 中使用 DOM 定义的异常的小节。

DOMImplementation 对象

DOMImplementation 接口提供了一种让应用程序确定他们所使用的 DOM 中某一特性可用性的方式。DOM 第 2 级还添加了使用 DOMImplementation 来创建新的 Document 和 DocumentType 对象的能力。

DOMImplementation.**hasFeature**(*feature*, *version*)

如果字符串对 *feature* 和 *version* 所标识的特性已被实现则返回 True。

DOMImplementation.**createDocument**(*namespaceUri*, *qualifiedName*, *doctype*)

返回一个新的 Document 对象 (DOM 的根节点)，包含一个具有给定 *namespaceUri* 和 *qualifiedName* 的下级 Element 对象。*doctype* 必须为由 *createDocumentType()* 创建的 DocumentType 对象，或者为 None。在 Python DOM API 中，前两个参数也可为 None 以表示不要创建任何下级 Element。

DOMImplementation.**createDocumentType**(*qualifiedName*, *publicId*, *systemId*)

返回一个新的封装了给定 *qualifiedName*, *publicId* 和 *systemId* 字符串的 DocumentType 对象，它表示包含在 XML 文档类型声明中的信息。

节点对象

XML 文档的所有组成部分都是 Node 的子类。

Node.**nodeType**

一个代表节点类型的整数。类型符号常量在 Node 对象上: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE。这是个只读属性。

Node.**parentNode**

当前节点的上级，或者对于文档节点则为 None。该值总是一个 Node 对象或者 None。对于 Element 节点，这将为上级元素，但对于根元素例外，在此情况下它将为 Document 对象。对于 Attr 节点，它将总是为 None。这是个只读属性。

Node.**attributes**

属性对象的 NamedNodeMap。这仅对元素才有实际值；其它对象会为该属性提供 None 值。这是个只读属性。

Node.**previousSibling**

在此节点之前具有相同上级的相邻节点。例如结束标记紧接在在 *self* 元素的开始标记之前的元素。当然，XML 文档并非只是由元素组成，因此之前相邻节点可以是文本、注释或者其他内容。如果此节点是上级的第一个子节点，则该属性将为 None。这是一个只读属性。

Node.**nextSibling**

在此节点之后具有相同上级的相邻节点。另请参见*previousSibling*。如果此节点是上级的最后一个子节点，则该属性将为 None。这是一个只读属性。

Node.**childNodes**

包含在此节点中的节点列表。这是一个只读属性。

Node.**firstChild**

节点的第一个下级，如果有的话，否则为 None。这是个只读属性。

Node.**lastChild**

节点的最后一个下级，如果有的话，否则为 None。这是个只读属性。

Node.**localName**

tagName 在冒号之后的部分，如果有冒号的话，否则为整个 tagName。该值为一个字符串。

Node.**prefix**

tagName 在冒号之前的部分，如果有冒号的话，否则为空字符串。该值为一个字符串或者为 None。

Node.**namespaceURI**

关联到元素名称的命名空间。这将是一个字符串或为 None。这是个只读属性。

Node.**nodeName**

这对于每种节点类型具有不同的含义；请查看 DOM 规格说明来了解详情。你总是可以从其他特征属性例如元素的 tagName 特征属性或属性的 name 特征属性获取你能从这里获取的信息。对于所有节点类型，这个属性的值都将是一个字符串或为 None。这是一个只读属性。

Node.**nodeValue**

这对于每种节点类型具有不同的含义；请查看 DOM 规格说明来了解详情。具体情况与*nodeName*的类似。该值是一个字符串或为 None。

Node.**hasAttributes**()

如果该节点具有任何属性则返回 True。

Node.**hasChildNodes**()

如果该节点具有任何子节点则返回 True。

Node.**isSameNode**(*other*)

如果 *other* 指向的节点就是此节点则返回 True。这对于使用了任何代理架构的 DOM 实现来说特别有用（因为多个对象可能指向相同节点）。

> **备注**
>
> 这是基于已提议的 DOM 第 3 等级 API，目前尚处于“起草”阶段，但这个特定接口看来并不存在争议。来自 W3C 的修改将不会影响 Python DOM 接口中的这个方法（不过针对它的任何新 W3C API 也将受到支持）。

Node.**appendChild**(*newChild*)

在子节点列表末尾添加一个新的子节点，返回 *newChild*。如果节点已存在于树结构中，它将先被移除。

Node.**insertBefore**(*newChild*, *refChild*)

在现有的子节点之前插入一个新的子节点。它必须属于 *refChild* 是这个节点的子节点的情况；如果不是，则会引发*ValueError*。*newChild* 会被返回。如果 *refChild* 为 None，它会将 *newChild* 插入到子节点列表的末尾。

Node.**removeChild**(*oldChild*)

移除一个子节点。*oldChild* 必须是这个节点的子节点；如果不是，则会引发*ValueError*。成功时 *oldChild* 会被返回。如果 *oldChild* 将不再被继续使用，则将调用它的 unlink() 方法。

Node.**replaceChild**(*newChild*, *oldChild*)

将一个现有节点替换为新的节点。这必须属于 *oldChild* 是该节点的子节点的情况；如果不是，则会引发*ValueError*。

Node.**normalize**()

合并相邻的文本节点以便将所有文本段存储为单个 Text 实例。这可以简化许多应用程序处理来自 DOM 树文本的操作。

Node.**cloneNode**(*deep*)

克隆此节点。设置 *deep* 表示也克隆所有子节点。此方法将返回克隆的节点。

节点列表对象

NodeList 代表一个节点列表。在 DOM 核心建议中这些对象有两种使用方式: 由 Element 对象提供作为其子节点列表，以及由 Node 的 getElementsByTagName() 和 getElementsByTagNameNS() 方法通过此接口返回对象来表示查询结果。

DOM 第 2 层级建议为这些对象定义一个方法和一个属性:

NodeList.**item**(*i*)

从序列中返回第 *i* 项，如果序列不为空的话，否则返回 None。索引号 *i* 不允许小于零或大于等于序列的长度。

NodeList.**length**

序列中的节点数量。

此外，Python DOM 接口还要求提供一些额外支持来允许将 NodeList 对象用作 Python 序列。所有 NodeList 实现都必须包括对 __len__() 和 __getitem__() 的支持；这样 NodeList 就允许使用 for 语句进行迭代并能正确地支持*len()* 内置函数。

如果一个 DOM 实现支持文档的修改，则 NodeList 实现还必须支持 __setitem__() 和 __delitem__() 方法。

文档类型对象

有关一个文档所声明的标注和实体的信息（包括解析器所使用并能提供信息的外部子集）可以从 DocumentType 对象获取。文档的 DocumentType 可从 Document 对象的 doctype 属性中获取；如果一个文档没有 DOCTYPE 声明，则该文档的 doctype 属性将被设为 None 而非此接口的一个实例。

DocumentType 是 Node 是专门化，并增加了下列属性:

DocumentType.**publicId**

文档类型定义的外部子集的公有标识。这将为一个字符串或者为 None。

DocumentType.**systemId**

文档类型定义的外部子集的系统标识。这将为一个字符串形式的 URI，或者为 None。

DocumentType.**internalSubset**

一个给出来自文档的完整内部子集的字符串。这不包括子集外面的圆括号。如果文档没有内部子集，则应为 None。

DocumentType.**name**

DOCTYPE 声明中给出的根元素名称，如果有的话。

DocumentType.**entities**

这是给出外部实体定义的 NamedNodeMap。对于多次定义的实体名称，则只提供第一次的定义（其他的会按照 XML 建议被忽略）。这可能为 None，如果解析器未提供此信息，或者如果未定义任何实体的话。

DocumentType.**notations**

这是给出标注定义的 NamedNodeMap。对于多次定义的标注，则只提供第一次的定义（其他的会按照 XML 建议被忽略）。这可能为 None，如果解析器未提供此信息，或者如果未定义任何标注的话。

Document 对象

Document 代表一个完整的 XML 文档，包括其组成元素、属性、处理指令和注释等。请记住它会继承来自 Node 的属性。

Document.**documentElement**

文档唯一的根元素。

Document.**createElement**(*tagName*)

创建并返回一个新的元素节点。当元素被创建时不会被插入到文档中。你需要通过某个其他方法例如 insertBefore() 或 appendChild() 来显示地插入它。

Document.**createElementNS**(*namespaceURI*, *tagName*)

创建并返回一个新的带有命名空间的元素。*tagName* 可以带有前缀。当元素被创建时不会被插入到文档中。你需要通过某个其他方法例如 insertBefore() 或 appendChild() 来显式地插入它。

Document.**createTextNode**(*data*)

创建并返回一个包含作为形参被传入的数据的文本节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createComment**(*data*)

创建并返回一个包含作为形参被传入的数据的注释节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createProcessingInstruction**(*target*, *data*)

创建并返回一个包含作为形参被传入的 *target* 和 *data* 的处理指令节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createAttribute**(*name*)

创建并返回一个属性节点。此方法不会将属性节点关联到任何特定的元素。你必须在正确的 Element 对象上使用 setAttributeNode() 来使用新创建的属性实例。

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)

创建并返回一个带有命名空间的属性节点。*tagName* 可以带有前缀。此方法不会将属性节点关联到任何特定的元素。你必须在正确的 Element 对象上使用 setAttributeNode() 来使用新创建的属性实例。

Document.**getElementsByTagName**(*tagName*)

搜索全部具有特定元素类型名称的后继元素（直接下级、下级的下级等等）。

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)

搜索全部具有特定命名空间 URI 和 localname 的后继元素（直接下级、下级的下级等等）。localname 是命名空间在前缀之后的部分。

元素对象

Element 是 Node 的子类，因此会继承该类的全部属性。

Element.**tagName**

元素类型名称。在使用命名空间的文档中它可能包含冒号。该值是一个字符串。

Element.**getElementsByTagName**(*tagName*)

与 Document 类中的对应方法相同。

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)

与 Document 类中的对应方法相同。

Element.**hasAttribute**(*name*)

如果元素带有名称为 *name* 的属性则返回 True。

Element.**hasAttributeNS**(*namespaceURI*, *localName*)

如果元素带有名称为 *namespaceURI* 加 *localName* 的属性则返回 True。

Element.**getAttribute**(*name*)

将名称为 *name* 的属性的值作为字符串返回。如果指定属性不存在，则返回空字符串，就像该属性没有对应的值一样。

Element.**getAttributeNode**(*attrname*)

返回名称为 *attrname* 的属性对应的 Attr 节点。

Element.**getAttributeNS**(*namespaceURI*, *localName*)

将名称为 *namespaceURI* 加 *localName* 的属性的值作为字符串返回。如果指定属性不存在，则返回空字符串，就像该属性没有对应的值一样。

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)

将给定 *namespaceURI* 加 *localName* 的属性的值作为节点返回。

Element.**removeAttribute**(*name*)

移除指定名称的节点。如果没有匹配的属性，则会引发*NotFoundErr*。

Element.**removeAttributeNode**(*oldAttr*)

从属性列表中移除并返回 *oldAttr*，如果该属性存在的话。如果 *oldAttr* 不存在，则会引发*NotFoundErr*。

Element.**removeAttributeNS**(*namespaceURI*, *localName*)

移除指定名称的属性。请注意它是使用 localName 而不是 qname。如果没有匹配的属性也不会引发异常。

Element.**setAttribute**(*name*, *value*)

将属性值设为指定的字符串。

Element.**setAttributeNode**(*newAttr*)

将一个新的属性节点添加到元素，当匹配到 name 属性时如有必要会替换现有的属性。如果发生了替换，将返回原有属性节点。如果 *newAttr* 已经被使用，则会引发*InuseAttributeErr*。

Element.**setAttributeNodeNS**(*newAttr*)

将一个新的属性节点添加到元素，当匹配到 namespaceURI 和 localName 属性时如有必要会替换现有的属性。如果发生了替换，将返回原有属性节点。如果 *newAttr* 已经被使用，则会引发*InuseAttributeErr*。

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)

将属性值设为 *namespaceURI* 和 *qname* 所给出的字符串。请注意 qname 是整个属性名称。这与上面的方法不同。

Attr 对象

Attr 继承自 Node，因此会继承其全部属性。

Attr.**name**

属性名称。在使用命名空间的文档中可能会包括冒号。

Attr.**localName**

名称在冒号之后的部分，如果有的话，否则为完整名称。这是个只读属性。

Attr.**prefix**
: 名称在冒号之前的部分，如果有冒号的话，否则为空字符串。

Attr.**value**
: 属性的文本值。这与 nodeValue 属性同义。

NamedNodeMap 对象

NamedNodeMap 不是继承自 Node。

NamedNodeMap.**length**
: 属性列表的长度。

NamedNodeMap.**item**(*index*)
: 返回特定带有索引号的属性。获取属性的顺序是强制规定的，但在 DOM 的生命期内会保持一致。其中每一项均为属性节点。可使用 value 属性获取其值。

还有一些试验性方法给予这个类更多的映射行为。你可以使用它们或者使用 Element 对象上标准化的 getAttribute*() 方法族。

注释对象

Comment 代表 XML 文档中的注释。它是 Node 的子类，但不能拥有下级节点。

Comment.**data**
: 注释的内容是一个字符串。该属性包含在开头 <!-- 和末尾 --> 之间的所有字符，但不包括这两个符号。

Text 和 CDATASection 对象

Text 接口代表 XML 文档中的文本。如果解析器和 DOM 实现支持 DOM 的 XML 扩展，则包裹在 CDATA 标记的节中的部分会被存储到 CDATASection 对象中。这两个接口很相似，但是提供了不同的 nodeType 属性值。

这些接口扩展了 Node 接口。它们不能拥有下级节点。

Text.**data**
: 字符串形式的文本节点内容。

> **备注**
>
> CDATASection 节点的使用并不表示该节点代表一个完整的 CDATA 标记节，只是表示该节点的内容是 CDATA 节的一部分。单个 CDATA 节可以由文档树中的多个节点来表示。没有什么办法能确定两个相邻的 CDATASection 节点是否代表不同的 CDATA 标记节。

ProcessingInstruction 对象

代表 XML 文档中的处理指令。它继承自 Node 接口并且不能拥有下级节点。

ProcessingInstruction.**target**
: 到第一个空格符为止的处理指令内容。这是个只读属性。

ProcessingInstruction.**data**
: 在第一个空格符之后的处理指令内容。

异常

DOM 第 2 层级推荐定义一个异常*DOMException*，以及多个变量用来允许应用程序确定发生了何种错误。*DOMException* 实例带有*code* 属性用来提供特定异常所对应的值。

Python DOM 接口提供了一些常量，但还扩展了异常集以使 DOM 所定义的每个异常代码都存在特定的异常。接口的具体实现必须引发正确的特定异常，它们各自带有正确的*code* 属性值。

exception xml.dom.**DOMException**

所有特定 DOM 异常所使用的异常基类。该异常类不可被直接实例化。

exception xml.dom.**DomstringSizeErr**

当指定范围的文本不能适配一个字符串时被引发。此异常在 Python DOM 实现中尚不可用，但可从不是以 Python 编写的 DOM 实现中接收。

exception xml.dom.**HierarchyRequestErr**

当尝试插入一个节点但该节点类型不被允许时被引发。

exception xml.dom.**IndexSizeErr**

当一个方法的索引或大小参数为负值或超出允许的值范围时被引发。

exception xml.dom.**InuseAttributeErr**

当尝试插入一个 `Attr` 节点但该节点已存在于文档中的某处时被引发。

exception xml.dom.**InvalidAccessErr**

当某个参数或操作在底层对象中不受支持时被引发。

exception xml.dom.**InvalidCharacterErr**

当某个字符串参数包含的字符在使用它的上下文中不被 XML 1.0 标准建议所允许时引发。例如，尝试创建一个元素类型名称中带有空格的 `Element` 节点将导致此错误被引发。

exception xml.dom.**InvalidModificationErr**

当尝试修改某个节点的类型时被引发。

exception xml.dom.**InvalidStateErr**

当尝试使用未定义或不再可用的的对象时被引发。

exception xml.dom.**NamespaceErr**

如果试图以 XML 中的命名空间 建议所不允许的方式修改任何对象，则会引发此异常。

exception xml.dom.**NotFoundErr**

当某个节点不存在于被引用的上下文中时引发的异常。例如，`NamedNodeMap.removeNamedItem()` 将在所传入的节点不在于于映射中时引发此异常。

exception xml.dom.**NotSupportedErr**

当具体实现不支持所请求的对象类型或操作时被引发。

exception xml.dom.**NoDataAllowedErr**

当为某个不支持数据的节点指定数据时被引发。

exception xml.dom.**NoModificationAllowedErr**

当尝试修改某个不允许修改的对象（例如只读节点）时被引发。

exception xml.dom.**SyntaxErr**

当指定了无效或非法的字符串时被引发。

exception xml.dom.**WrongDocumentErr**

当将某个节点插入非其当前所属的另一个文档，并且具体实现不支持从一个文档向一个文档迁移节点时被引发。

DOM 建议映射中针对上述异常而定义的异常代码如下表所示:

常量	异常
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

21.6.3 一致性

本节描述了 Python DOM API、W3C DOM 建议以及 Python 的 OMG IDL 映射之间的一致性要求和关系。

类型映射

将根据下表，将 DOM 规范中使用的 IDL 类型映射为 Python 类型。

IDL 类型	Python 类型
`boolean`	`bool` 或 `int`
`int`	`int`
`long int`	`int`
`unsigned int`	`int`
`DOMString`	`str` 或 `bytes`
`null`	`None`

访问器方法

从 OMG IDL 到 Python 的映射以类似于 Java 映射的方式定义了针对 IDL `attribute` 声明的访问器函数。映射以下 IDL 声明

```
readonly attribute string someValue;
         attribute string anotherValue;
```

会产生三个访问器函数: `someValue` 的"get" 方法 (`_get_someValue()`)，以及 `anotherValue` 的"get" 和"set" 方法 (`_get_anotherValue()` 和 `_set_anotherValue()`)。特别地，该映射不要求 IDL 属性像普通 Python 属性那样可访问: `object.someValue` 并非必须可用，并可能引发*AttributeError*。

但是，Python DOM API 则确实要求普通属性访问可用。这意味着由 Python IDL 解译器生成的典型代理有可能会不可用，如果 DOM 对象是通过 CORBA 来访问则在客户端可能需要有包装对象。虽然这确实要求为 CORBA DOM 客户端进行额外的考虑，但具有从 Python 通过 CORBA 使用 DOM 经验的实现并不会认为这是个问题。已经声明了 `readonly` 的属性不必在所有 DOM 实现中限制写入访问。

在 Python DOM API 中，访问器函数不是必须的。如果提供，则它们应当采用由 Python IDL 映射所定义的形式，但这些方法会被认为不必要，因为这些属性可以从 Python 直接访问。永远都不要为 `readonly` 属性提供"set" 访问器。

IDL 定义没有完全体现 W3C DOM API 的要求，如特定对象的概念，又如 `getElementsByTagName()` 的返回值为"live" 等。Python DOM API 并不强制具体实现执行这些要求。

21.7 xml.dom.minidom --- 最小化的 DOM 实现

源代码: Lib/xml/dom/minidom.py

---

xml.dom.minidom 是文档对象模型接口的最小化实现，具有与其他语言类似的 API。它的目标是比完整 DOM 更简单并且更为小巧。对于 DOM 还不十分熟悉的用户则应当考虑改用*xml.etree.ElementTree* 模块来进行 XML 处理。

> **警告**
>
> *xml.dom.minidom* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

DOM 应用程序通常会从将某个 XML 解析为 DOM 开始。使用*xml.dom.minidom* 时，这是通过各种解析函数来完成的:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # 解析指定名称的 XML 文件

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # 解析打开的文件

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

parse() 函数可接受一个文件名或者打开的文件对象。

xml.dom.minidom.**parse**(*filename_or_file*, *parser=None*, *bufsize=None*)

根据给定的输入返回一个 Document。*filename_or_file* 可以是一个文件名，或是一个文件型对象。如果给定 *parser* 则它必须是一个 SAX2 解析器对象。此函数将修改解析器的处理程序并激活命名空间支持；其他解析器配置（例如设置一个实体求解器）必须已经提前完成。

如果你将 XML 存放为字符串形式，则可以改用*parseString()* 函数:

xml.dom.minidom.**parseString**(*string*, *parser=None*)

返回一个代表 *string* 的 Document。此方法会为指定字符串创建一个*io.StringIO* 对象并将其传递给*parse()*。

两个函数均返回一个代表文档内容的 Document 对象。object representing the content of the document.

parse() 和*parseString()* 函数所做的是将 XML 解析器连接到一个"DOM 构建器"，它可以从任意 SAX 解析器接收解析事件并将其转换为 DOM 树结构。这两个函数的名称可能有些误导性，但在学习此接口时是很容易掌握的。文档解析操作将在这两个函数返回之前完成；简单地说这两个函数本身并不提供解析器实现。

你也可以通过在一个"DOM 实现" 对象上调用方法来创建 Document。此对象可通过调用*xml.dom* 包或者*xml.dom.minidom* 模块中的 getDOMImplementation() 函数来获取。一旦你获得了一个 Document，你就可以向它添加子节点来填充 DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

一旦你得到了 DOM 文档对象，你就可以通过其属性和方法访问对应 XML 文档的各个部分。这些属性定义在 DOM 规格说明当中；文档对象的主要特征属性是 `documentElement`。它给出了 XML 文档中的主元素：即包含了所有其他元素的元素。以下是一个示例程序:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

当你完成对一个 DOM 树的处理时，你可以选择调用 `unlink()` 方法来鼓励尽早清除已不再需要的对象。`unlink()` 是针对 DOM API 的*xml.dom.minidom* 专属扩展，它会将特定节点及其下级标记为不再有用。在其他情况下，Python 的垃圾回收器将负责最终处理树结构中的对象。

参见

文档对象模型 (DOM) 第 1 层级规格说明
被*xml.dom.minidom* 所支持的 W3C 针对 DOM 的建议。

21.7.1 DOM 对象

Python 的 DOM API 定义被作为*xml.dom* 模块文档的一部分给出。这一节列出了该 API 和*xml.dom.minidom* 之间的差异。

`Node.`**`unlink`**`()`

破坏 DOM 的内部引用以便它能在没有循环 GC 的 Python 版本上垃圾回收器回收。即使在循环 GC 可用的时候，使用此方法也可让大量内存更快变为可用，因此当 DOM 对象不再被需要时尽早调用它们的这个方法是很好的做法。此方法只须在 `Document` 对象上调用，但也可以在下级节点上调用以丢弃该节点的下级节点。

你可以通过使用 `with` 语句来避免显式调用此方法。以下代码会在 `with` 代码块退出时自动取消链接 *dom*:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # 操作 dom。
```

`Node.`**`writexml`**`(`*writer*, *indent=''*, *addindent=''*, *newl=''*, *encoding=None*, *standalone=None*`)`

将 XML 写入到写入器对象。写入器接受文本而非字节串作为输入，它应当具有与文件对象接口相匹配的 `write()` 方法。*indent* 形参是当前节点的缩进层级。*addindent* 形参是用于当前节点的下级节点的缩进量。*newl* 形参指定用于一行结束的字符串。

对于 `Document` 节点，可以使用附加的关键字参数 *encoding* 来指定 XML 标头的编码格式字段。

类似地，显式指明 *standalone* 参数将会使单独的文档声明被添加到 XML 文档的开头部分。如果将该值设为 `True`，则会添加 `standalone="yes"`，否则它会被设为 `"no"`。未指明该参数将使文档声明被省略。

在 3.8 版本发生变更: *writexml()* 方法现在会保留用户指定的属性顺序。

在 3.9 版本发生变更: 增加了 *standalone* 形参。

`Node.`**`toxml`**`(`*encoding=None*, *standalone=None*`)`

返回一个包含 XML DOM 节点所代表的 XML 的字符串或字节串。

带有显式的 *encoding*[1] 参数时，结果为使用指定编码格式的字节串。没有 *encoding* 参数时，结果为 Unicode 字符串，并且结果字符串中的 XML 声明将不指定编码格式。使用 UTF-8 以外的编码格式对此字符串进行编码通常是不正确的，因为 UTF-8 是 XML 的默认编码格式。

standalone 参数的行为与*writexml()* 中的完全一致。

在 3.8 版本发生变更: *toxml()* 方法现在会保留用户指定的属性顺序。

在 3.9 版本发生变更: 增加了 *standalone* 形参。

[1] 包括在 XML 输出中的编码格式名称应当遵循适当的标准。例如，"UTF-8" 是有效的，但"UTF8" 在 XML 文档的声明中是无效的，即使 Python 接受其作为编码格式名称。详情参见 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

Node.**toprettyxml**(*indent='\t'*, *newl='\n'*, *encoding=None*, *standalone=None*)

返回文档的美化打印版本。*indent* 指定缩进字符串并默认为制表符；*newl* 指定标示每行结束的字符串并默认为 `\n`。

encoding 参数的行为类似于 `toxml()` 的对应参数。

standalone 参数的行为与 `writexml()` 中的完全一致。

在 3.8 版本发生变更: `toprettyxml()` 方法现在会保留用户指定的属性顺序。

在 3.9 版本发生变更: 增加了 *standalone* 形参。

21.7.2 DOM 示例

此示例程序是个相当实际的简单程序示例。在这个特定情况中，我们没有过多地利用 DOM 的灵活性。

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print(f"<title>{getText(title.childNodes)}</title>")

def handleSlideTitle(title):
```

(续下页)

(接上页)

```
    print(f"<h2>{getText(title.childNodes)}</h2>")

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print(f"<li>{getText(point.childNodes)}</li>")

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print(f"<p>{getText(title.childNodes)}</p>")

handleSlideshow(dom)
```

21.7.3 minidom 和 DOM 标准

xml.dom.minidom 模块实际上是兼容 DOM 1.0 的 DOM 并带有部分 DOM 2 特性（主要是命名空间特性）。

Python 中 DOM 接口的用法十分直观。会应用下列映射规则:

- 接口是通过实例对象来访问的。应用程序不应实例化这些类本身；它们应当使用 `Document` 对象提供的创建器函数。派生的接口支持上级接口的所有操作（和属性），并添加了新的操作。
- 操作以方法的形式使用。因由 DOM 只使用 `in` 形参，参数是以正常顺序传入的（从左至右）。不存在可选参数。`void` 操作返回 `None`。
- IDL 属性会映射到实例属性。为了兼容针对 Python 的 OMG IDL 语言映射，属性 `foo` 也可通过访问器方法 `_get_foo()` 和 `_set_foo()` 来访问。`readonly` 属性不可被修改；运行时并不强制要求这一点。
- `short int`, `unsigned int`, `unsigned long long` 和 `boolean` 类型都会映射为 Python 整数类型。
- `DOMString` 类型会映射为 Python 字符串。*xml.dom.minidom* 支持字节串或字符串，但通常是产生字符串。`DOMString` 类型的值也可以为 `None`，W3C 的 DOM 规格说明允许其具有 IDL `null` 值。
- `const` 声明会映射为它们各自的作用域内的变量（例如 `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`）；它们不可被修改。
- `DOMException` 目前不被*xml.dom.minidom* 所支持。*xml.dom.minidom* 会改为使用标准 Python 异常例如*TypeError* 和*AttributeError*。
- `NodeList` 对象是使用 Python 内置列表类型来实现的。这些对象提供了 DOM 规格说明中定义的接口，但在较早版本的 Python 中它们不支持官方 API。相比在 W3C 建议中定义的接口，它们要更加的"Pythonic"。

下列接口未在*xml.dom.minidom* 中实现:

- `DOMTimeStamp`
- `EntityReference`

这些接口所反映的 XML 文档信息对于大多数 DOM 用户来说没有什么帮助。

备注

21.8 `xml.dom.pulldom` --- 对构建部分 DOM 树的支持

源代码: Lib/xml/dom/pulldom.py

xml.dom.pulldom 模块提供了一个“拉取解析器”，它能在必要时被用于产生文件的可访问 DOM 的片段。其基本概念包括从输入的 XML 流拉取“事件”并处理它们。与同样地同时应用了事件驱动处理模型加回调函数的 SAX 不同，拉取解析器的用户要负责显式地从流拉取事件，并循环遍历这些事件直到处理结束或者发生了错误条件。

警告

xml.dom.pulldom 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

在 3.7.1 版本发生变更: SAX 解析器默认不再处理一般外部实体以提升在默认情况下的安全性。要启用外部实体处理，请传入一个自定义的解析器实例:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

示例:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

`event` 是一个常量，可以取下列值之一:

- `START_ELEMENT`
- `END_ELEMENT`
- `COMMENT`
- `START_DOCUMENT`
- `END_DOCUMENT`
- `CHARACTERS`
- `PROCESSING_INSTRUCTION`
- `IGNORABLE_WHITESPACE`

`node` 是一个 `xml.dom.minidom.Document`, `xml.dom.minidom.Element` 或 `xml.dom.minidom.Text` 类型的对象。

由于文档是被当作“展平”的事件流来处理的，文档“树”会被隐式地遍历并且无论所需元素在树中的深度如何都会被找到。换句话说，不需要考虑层级问题，例如文档节点的递归搜索等，但是如果元素的内容很重要，则有必要保留一些上下文相关的状态（例如记住任意给定点在文档中的位置）或者使用*DOMEventStream.expandNode()* 方法并切换到 DOM 相关的处理过程。

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)

xml.sax.handler.ContentHandler 的子类。

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)

xml.sax.handler.ContentHandler 的子类。

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)

基于给定的输入返回一个*DOMEventStream*。*stream_or_string* 可以是一个文件名，或是一个文件型对象。*parser* 如果给出，则必须是一个*XMLReader* 对象。此函数将改变解析器的文档处理程序并激活命名空间支持；其他解析器配置（例如设置实体解析器）必须在之前已完成。

如果你将 XML 存放为字符串形式，则可以改用*parseString()* 函数:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)

返回一个*DOMEventStream* 来表示 (Unicode) *string*。

xml.dom.pulldom.**default_bufsize**

将 *bufsize* 形参的默认值设为*parse()*。

此变量的值可在调用*parse()* 之前修改并使新值生效。

21.8.1 DOMEventStream 对象

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)

在 3.11 版本发生变更: 对 __getitem__() 方法的支持已被移除。

getEvent()

返回一个元组，其中包含 *event* 和 xml.dom.minidom.Document 形式的当前 *node* 如果 event 等于 START_DOCUMENT，包含 xml.dom.minidom.Element 如果 event 等于 START_ELEMENT 或 END_ELEMENT 或者 xml.dom.minidom.Text 如果 event 等于 CHARACTERS。当前 node 不包含有关其子节点的信息，除非*expandNode()* 被调用。

expandNode(*node*)

将 *node* 的所有子节点扩展到 *node* 中。例如:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # 以下语句只打印 '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # 以下语句将打印节点所有的子节点 '<p>Some text <div>and more</div></p>'
        print(node.toxml())
```

reset()

21.9 xml.sax --- SAX2 解析器支持

源代码: Lib/xml/sax/__init__.py

---

xml.sax 包提供多个模块，它们在 Python 上实现了用于 XML (SAX) 接口的简单 API。这个包本身为 SAX API 用户提供了一些最常用的 SAX 异常和便捷函数。

> **警告**
>
> *xml.sax* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

在 3.7.1 版本发生变更: SAX 解析器默认不会再处理通用外部实体以便提升安全性。在此之前，解析器会创建网络连接来获取远程文件或是从 DTD 和实体文件系统中加载本地文件。此特性可通过在解析器对象上调用*setFeature()* 对象并传入参数*feature_external_ges* 来重新启用。

可用的便捷函数如下所列:

xml.sax.**make_parser**(*parser_list=[]*)

创建并返回一个 SAX *XMLReader* 对象。将返回第一个被找到的解析器。如果提供了 *parser_list*，它必须为一个包含字符串的可迭代对象，这些字符串指定了具有名为 `create_parser()` 函数的模块。在 *parser_list* 中列出的模块将在默认解析器列表中的模块之前被使用。

在 3.8 版本发生变更: *parser_list* 参数可以是任意可迭代对象，而不一定是列表。

xml.sax.**parse**(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)

创建一个 SAX 解析器并用它来解析文档。用于传入文档的 *filename_or_stream* 可以是一个文件名或文件对象。*handler* 形参必须是一个 SAX *ContentHandler* 实例。如果给出了 *error_handler*，则它必须是一个 SAX *ErrorHandler* 实例；如果省略，则对于任何错误都将引发*SAXParseException*。此函数没有返回值；所有操作必须由传入的 *handler* 来完成。

xml.sax.**parseString**(*string*, *handler*, *error_handler=handler.ErrorHandler()*)

类似于*parse()*，但解析对象是作为形参传入的缓冲区 *string*。*string* 必须为*str* 实例或者*bytes-like object*。

在 3.5 版本发生变更: 增加了对*str* 实例的支持。

典型的 SAX 应用程序会使用三种对象：读取器、处理器和输入源。“读取器”在此上下文中与解析器同义，即某个从输入源读取字节或字符，并产生事件序列的代码段。事件随后将被分发给处理器对象，即由读取器唤起处理器上的某个方法。因此 SAX 应用程序必须获取一个读取器对象，创建或打开输入源，创建处理器，并一起连接到这些对象。作为准备工作的最后一步，将调用读取器来解析输入内容。在解析过程中，会根据来自输入数据的结构化和语义化事件来调用处理器对象上的方法。

就这些对象而言，只有接口部分是需要关注的；它们通常不是由应用程序本身来实例化。由于 Python 没有显式的接口标记法，它们的正式引入形式是类，但应用程序可能会使用并非从已提供的类继承而来的实现。*InputSource*, *Locator*, Attributes, AttributesNS 以及*XMLReader* 接口是在*xml.sax.xmlreader* 模块中定义的。处理器接口是在*xml.sax.handler* 中定义的。为了方便起见，*InputSource* (它往往会被直接实例化) 和处理器类也可以从*xml.sax* 获得。这些接口的描述见下文。

除了这些类，*xml.sax* 还提供了如下异常类。

exception xml.sax.**SAXException**(*msg*, *exception=None*)

封装某个 XML 错误或警告。这个类可以包含来自 XML 解析器或应用程序的基本错误或警告信息：它可以被子类化以提供额外的功能或是添加本地化信息。请注意虽然在*ErrorHandler* 接口中定义的处理器可以接收该异常的实例，但是并不要求实际引发该异常 --- 它也可以被用作信息的容器。

当实例化时，*msg* 应当是适合人类阅读的错误描述。如果给出了可选的 *exception* 形参，它应当为 None 或者解析代码所捕获的异常并会被作为信息传递出去。

这是其他 SAX 异常类的基类。

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)

SAXException 的子类，针对解析错误引发。这个类的实例会被传递给 SAX *ErrorHandler* 接口的方法来提供关于解析错误的信息。这个类支持 SAX *Locator* 接口以及*SAXException* 接口。

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)

SAXException 的子类，当 SAX *XMLReader* 遇到不可识别的特性或属性时引发。SAX 应用程序和扩展可能会出于类似目的而使用这个类。

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)

SAXException 的子类，当 SAX *XMLReader* 被要求启用某个不受支持的特性，或者将某个属性设为具体实现不支持的值时引发。SAX 应用程序和扩展可能会出于类似目的而使用这个类。

参见

SAX: The Simple API for XML
这个网站是 SAX API 定义的焦点。它提供了一个 Java 实现以及在线文档。还包括其他实现的链接和历史信息。

xml.sax.handler 模块
: 应用程序所提供对象的接口定义。

xml.sax.saxutils 模块
: 可在 SAX 应用程序中使用的便捷函数。

xml.sax.xmlreader 模块
: 解析器所提供对象的接口定义。

21.9.1 SAXException 对象

SAXException 异常类支持下列方法:

SAXException.**getMessage**()
: 返回描述错误条件的适合人类阅读的消息。

SAXException.**getException**()
: 返回一个封装的异常对象或者 None。

21.10 xml.sax.handler --- SAX 处理器的基类

源代码: Lib/xml/sax/handler.py

SAX API 定义了五种处理器：内容处理器、DTD 处理器、错误处理器、实体解析器以及词法处理器。应用程序通常只需要实现他们感兴趣的事件对应的接口；他们可以在单个对象或多个对象中实现这些接口。处理器的实现应当继承自 *xml.sax.handler* 模块所提供的基类，以便所有方法都能获得默认的实现。

class xml.sax.handler.**ContentHandler**
: 这是 SAX 中的主回调接口，也是对应用程序来说最重要的一个接口。此接口中事件的顺序反映了文档中信息的顺序。

class xml.sax.handler.**DTDHandler**
: 处理 DTD 事件。

  这个接口仅指定了基本解析（未解析的实体和属性）所需的那些 DTD 事件。

class xml.sax.handler.**EntityResolver**
: 用于解析实体的基本接口。如果你创建了实现此接口的对象，然后用你的解析器注册该对象，该解析器将调用你的对象中的方法来解析所有外部实体。

class xml.sax.handler.**ErrorHandler**
: 解析器用来向应用程序表示错误和警告的接口。这个对象的方法控制错误是要立即转换为异常还是以某种其他该来来处理。

class xml.sax.handler.**LexicalHandler**
: 解析器用来代表低频度事件的接口，这些事件可能是许多应用程序都不感兴趣的。

除了这些类，*xml.sax.handler* 还提供了表示特性和属性名称的符号常量。

xml.sax.handler.**feature_namespaces**
: 值: "http://xml.org/sax/features/namespaces"
  true: 执行命名空间处理。
  false: 可选择不执行命名空间处理 (这意味着 namespace-prefixes; default)。
  access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_namespace_prefixes**

值: "http://xml.org/sax/features/namespace-prefixes"
true: 报告原始的带前缀名称和用于命名空间声明的属性。
false: 不报告用于命名空间声明的属性，可选择不报告原始的带前缀名称（默认）。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_string_interning**

值: "http://xml.org/sax/features/string-interning"
true: 所有元素名称、前缀、属性名称、命名空间 URI 以及本地名称都使用内置的 intern 函数进行内化。
false: 名称不要求被内化，但也可以被内化（默认）。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_validation**

值: "http://xml.org/sax/features/validation"
true: 报告所有的验证错误（包括 external-general-entities 和 external-parameter-entities）。
false: 不报告验证错误。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_external_ges**

值: "http://xml.org/sax/features/external-general-entities"
true: 包括所有的外部通用（文本）实体。
false: 不包括外部通用实体。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_external_pes**

值: "http://xml.org/sax/features/external-parameter-entities"
true: 包括所有的外部参数实体，也包括外部 DTD 子集。
false: 不包括任何外部参数实体，也不包括外部 DTD 子集。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**all_features**

全部特性列表。

xml.sax.handler.**property_lexical_handler**

值: "http://xml.org/sax/properties/lexical-handler"
数据类型: xml.sax.handler.LexicalHandler (在 Python 2 中不受支持)
描述: 可选的扩展处理器，用于注释等词法事件。
访问: 读/写

xml.sax.handler.**property_declaration_handler**

值: "http://xml.org/sax/properties/declaration-handler"
数据类型: xml.sax.sax2lib.DeclHandler (在 Python 2 中不受支持)
描述: 可选的扩展处理器，用于标注和未解析实体以外的 DTD 相关事件。
访问: 读/写

xml.sax.handler.**property_dom_node**

值: "http://xml.org/sax/properties/dom-node"
数据类型: org.w3c.dom.Node (在 Python 2 中不受支持)
描述: 在解析时，如果这是一个 DOM 迭代器则为当前被访问的 DOM 节点；不在解析时，则将根 DOM 节点用于迭代。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**property_xml_string**

值: "http://xml.org/sax/properties/xml-string"
数据类型: Bytes
描述: 作为当前事件来源的字符串字面值。
访问: 只读

xml.sax.handler.**all_properties**
已知属性名称列表。

21.10.1 ContentHandler 对象

用户应当子类化*ContentHandler* 来支持他们的应用程序。以下方法会由解析器在输入文档的适当事件上调用:

ContentHandler.**setDocumentLocator**(*locator*)
由解析器调用来给予应用程序一个定位器以确定文档事件来自何处。

强烈建议（虽然不是绝对的要求）SAX 解析器提供一个定位器：如果提供的话，它必须在唤起 DocumentHandler 接口的任何其他方法之前通过唤起此方法来提供定位器。

定位器允许应用程序确定任何文档相关事件的结束位置，即使解析器没有报告错误。通常，应用程序将使用这些信息来报告它自己的错误（例如未匹配到应用程序业务规则的字符内容）。定位器所返回的信息可能不足以与搜索引擎配合使用。

请注意定位器只有在发唤起此接口中的事件时才会返回正确的信息。应用程序不应试图在其他任何时刻使用它。

ContentHandler.**startDocument**()
接收一个文档开始的通知。

SAX 解析器将只唤起这个方法一次，并且会在调用这个接口或 DTDHandler 中的任何其他方法之前(*setDocumentLocator()* 除外)。

ContentHandler.**endDocument**()
接收一个文档结束的通知。

SAX 解析器将只唤起这个方法一次，并且它将是在解析过程中最后被唤起的方法。解析器在（因不可恢复的错误）放弃解析或到达输入的终点之前不应唤起这个方法。

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)
开始一个前缀 URI 命名空间映射的范围。

来自此事件的信息对于一般命名空间处理来说是不必要的：当 feature_namespaces 特性被启用时（默认）SAX XML 读取器将自动为元素和属性名称替换前缀。

但是也存在一些情况，当应用程序需要在字符数据或属性值中使用前缀，而它们无法被安全地自动扩展；*startPrefixMapping()* 和*endPrefixMapping()* 事件会向应用程序提供信息以便在这些上下文内部扩展前缀，如果有必要的话。

请注意*startPrefixMapping()* 和*endPrefixMapping()* 事件并不保证能够相对彼此被正确地嵌套：所有*startPrefixMapping()* 事件都将在对应的*startElement()* 事件之前发生，而所有*endPrefixMapping()* 事件都将在对应的*endElement()* 事件之后发生，但它们的并不保证一致。

ContentHandler.**endPrefixMapping**(*prefix*)
结束一个前缀 URI 映射的范围。

请参看*startPrefixMapping()* 了解详情。此事件将总是会在对应的*endElement()* 事件之后发生，但*endPrefixMapping()* 事件的顺序则并没有保证。

ContentHandler.**startElement**(*name*, *attrs*)

在非命令空间模式下指示一个元素的开始。

name 形参包含字符串形式的元素类型原始 XML 1.0 名称而 *attrs* 形参存放包含元素属性的 Attributes 接口对象 (参见*Attributes 接口*)。作为 *attrs* 传入的对象可能被解析器所重用；维持一个对它的引用不是保持属性副本的可靠方式。要保持这些属性的一个副本，请使用 *attrs* 对象的*copy()* 方法。

ContentHandler.**endElement**(*name*)

在非命名空间模式下指示一个元素的结束。

name 形参包含元素类型的名称，与*startElement()* 事件的一样。

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)

在命名空间模式下指示一个元素的开始。

name 形参包含以 (uri, localname) 元组表示的元素类型名称，*qname* 形参包含源文档中使用的原始 XML 1.0 名称，而 *attrs* 形参存放包含元素属性的 AttributesNS 接口实例 (参见*AttributesNS 接口*)。如果没有命名空间被关联到元素，则 *name* 的 *uri* 部分将为 None。作为 *attrs* 传入的对象可能被解析器所重用；维持一个对它的引用不是保持属性副本的可靠方式。要保持这些属性的一个副本，请使用 *attrs* 对象的*copy()* 方法。

解析器可将 *qname* 形参设为 None，除非 feature_namespace_prefixes 特性已被激活。

ContentHandler.**endElementNS**(*name*, *qname*)

在命名空间模式下指示一个元素的结束。

name 形参包含元素类型的名称，与*startElementNS()* 方法的一样，*qname* 形参也是类似的。

ContentHandler.**characters**(*content*)

接收字符数据的通知。

解析器将调用此方法来报告每一个字符数据分块。SAX 解析器可以将所有连续字符数据返回为一个单独分块，或者将其拆成几个分块；但是，在任意单个事件中的所有字符都必须来自同一个外部实体以便定位器提供有用的信息。

content 可以是一个字符串或字节串实例；expat 读取器模块总是会产生字符串。

> **备注**
>
> Python XML 特别关注小组所提供的早期 SAX 1 接口针对此方法使用了一个更类似于 Java 的接口。由于 Python 所使用的大多数解析器都没有利用老式的接口，因而选择了更简单的签名来替代它。要将旧代码转换为新接口，请使用 *content* 而不要通过旧的 *offset* 和 *length* 形参来对内容进行切片。

ContentHandler.**ignorableWhitespace**(*whitespace*)

接收元素内容中可忽略空白符的通知。

验证解析器必须使用此方法来报告每个可忽略的空白符分块（参见 W3C XML 1.0 建议第 2.10 节）：非验证解析器如果能够解析并使用内容模型的话也可以使用此方法。

SAX 解析器可以将所有连续字符数据返回为一个单独分块，或者将其拆成几个分块；但是，在任意单个事件中的所有字符都必须来自同一个外部实体以便定位器提供有用的信息。

ContentHandler.**processingInstruction**(*target*, *data*)

接受一条处理指令的通知。

解析器将为已找到的每条处理指令唤起该方法一次：请注意处理指令可能出现在主文档元素之前或之后。

SAX 解析器绝不应当使用此方法来报告 XML 声明（XML 1.0 第 2.8 节）或文本声明（XML 1.0 第 4.3.1 节）。

ContentHandler.**skippedEntity**(*name*)

接收一个已跳过实体的通知。

解析器将为每个已跳过实体唤起此方法一次。非验证处理程序可能会跳过未看到声明的实体（例如，由于实体是在一个外部 because, for example, the entity was declared in an external DTD 子集中声明的）。所有处理程序都可以跳过外部实体，具体取决于 `feature_external_ges` 和 `feature_external_pes` 属性的值。

21.10.2 DTDHandler 对象

DTDHandler 实例提供了下列方法:

DTDHandler.**notationDecl**(*name*, *publicId*, *systemId*)

处理标注声明事件。

DTDHandler.**unparsedEntityDecl**(*name*, *publicId*, *systemId*, *ndata*)

处理未解析的实体声明事件。

21.10.3 EntityResolver 对象

EntityResolver.**resolveEntity**(*publicId*, *systemId*)

求解一个实体的系统标识符并返回一个字符串形式的系统标识符作为读取源，或是一个 InputSource 作为读取源。默认的实现会返回 *systemId*。

21.10.4 ErrorHandler 对象

带有这个接口的对象被用于接收来自 *XMLReader* 的错误和警告信息。如果你创建了一个实现此接口的对象，然后用你的 *XMLReader* 注册这个对象，则解析器将调用你的对象中的这个方法来报告所有的警告和错误。有三个可用的错误级别：警告、（或许）可恢复的错误和不可恢复的错误。所有方法都接受 *SAXParseException* 作为唯一的形参。错误和警告可以通过引发所传入的异常对象来转换为异常。

ErrorHandler.**error**(*exception*)

当解析器遇到一个可恢复的错误时调用。如果此方法没有引发异常，则解析可能会继续，但是应用程序不能预期获得更多的文档信息。允许解析器继续可能会允许在输入文档中发现额外的错误。

ErrorHandler.**fatalError**(*exception*)

当解析器遇到一个不可恢复的错误时调用；在此方法返回时解析应当终止。

ErrorHandler.**warning**(*exception*)

当解析器向应用程序提供次要警告信息时调用。在此方法返回时解析应当继续，并且文档信息将继续被传递给应用程序。在此方法中引发异常将导致解析结束。

21.10.5 LexicalHandler 对象

可选的词法事件 SAX2 处理器。

这个处理器被用来获取一个 XML 文档的相关词法信息。词法信息包括描述所使用的文档编码格式和嵌入文档中的 XML 注释，以及 DTD 和任何 CDATA 部分的节边界。词法处理器的使用方式与内容处理器相同。

通过使用带有属性标识符 `'http://xml.org/sax/properties/lexical-handler'` 的 setProperty 方法来设置一个 XMLReader 的 LexicalHandler。

LexicalHandler.**comment**(*content*)

报告在文档中任何地方（包括 DTD 和文档元素以外）的注释。

LexicalHandler.**startDTD**(*name*, *public_id*, *system_id*)

如果文档有关联的 DTD 则报告 DTD 声明的开始。

LexicalHandler.**endDTD**()

报告 DTD 声明的结束。

LexicalHandler.**startCDATA**()

报告 CDATA 标记部分的开始。

CDATA 标记部分的内容将通过字符处理器来报告。

LexicalHandler.**endCDATA**()

报告 CDATA 标记部分的结束。

21.11 xml.sax.saxutils --- SAX 工具集

源代码: Lib/xml/sax/saxutils.py

---

xml.sax.saxutils 模块包含一些在创建 SAX 应用程序时十分有用的类和函数，它们可以被直接使用，或者是作为基类使用。

xml.sax.saxutils.**escape**(*data*, *entities={}*)

对数据字符串中的 '&', '<' 和 '>' 进行转义。

你可以通过传入一个字典作为可选的 *entities* 形参来对其他字符串数据进行转义。字典的键和值必须为字符串；每个键将被替换为其所对应的值。字符 '&', '<' 和 '>' 总是会被转义，即使提供了 *entities*。

> **备注**
>
> 此函数应当仅用于对无法直接在 XML 中使用的字符进行转义。请不要将此函数用作通用的字符串转换函数。

xml.sax.saxutils.**unescape**(*data*, *entities={}*)

对字符串数据中的 '&', '<' 和 '>' 进行反转义。

你可以通过传入一个字典作为可选的 *entities* 形参来对其他数据字符串进行转义。字典的键和值必须都为字符串；每个键将被替换为所对应的值。'&', '<' 和 '>' 将总是保持不被转义，即使提供了 *entities*。

xml.sax.saxutils.**quoteattr**(*data*, *entities={}*)

类似于*escape()*，但还会对 *data* 进行处理以将其用作属性值。返回值是 *data* 加上任何额外要求的替换的带引号版本。*quoteattr()* 将基于 *data* 的内容选择一个引号字符，以尽量避免在字符串中编码任何引号字符。如果单双引号字符在 *data* 中都存在，则双引号字符将被编码并且 *data* 将使用双引号来标记。结果字符串可被直接用作属性值:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

此函数适用于为 HTML 或任何使用引用实体语法的 SGML 生成属性值。

class xml.sax.saxutils.**XMLGenerator**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)

这个类通过将 SAX 事件写回到 XML 文档来实现*ContentHandler* 接口。换句话说，使用*XMLGenerator* 作为内容处理程序将重新产生所解析的原始文档。*out* 应当为一个文件型对象，它默认将为 *sys.stdout*。*encoding* 为输出流的编码格式，它默认将为 'iso-8859-1'。*short_empty_elements* 控制不包含内容的元素的格式化：如为 False (默认值) 则它们会以开始/结束标记对的形式被发送，如果设为 True 则它们会以单个自结束标记的形式被发送。

在 3.2 版本发生变更: 增加了 *short_empty_elements* 形参。

class xml.sax.saxutils.**XMLFilterBase**(*base*)

这个类被设计用来分隔*XMLReader* 和客户端应用的事件处理程序。在默认情况下，它除了将请求传送给读取器并将事件传送给处理程序之外什么都不做，但其子类可以重载特定的方法以在传送它们的时候修改事件流或配置请求。

xml.sax.saxutils.**prepare_input_source**(*source*, *base=''*)

此函数接受一个输入源和一个可选的基准 URL 并返回一个经过完整解析可供读取的*InputSource*对象。输入源的给出形式可以是字节串、文件型对象或*InputSource*对象；解析器将使用此函数来针对它们的*parse()*方法实现多态 *source* 参数。

21.12 xml.sax.xmlreader --- 用于 XML 解析器的接口

源代码: Lib/xml/sax/xmlreader.py

SAX 解析器实现了*XMLReader*接口。它们是在一个 Python 模块中实现的，该模块必须提供一个 create_parser() 函数。该函数由*xml.sax.make_parser()*不带参数地唤起来创建新的解析器对象。

class xml.sax.xmlreader.**XMLReader**

可由 SAX 解析器继承的基类。

class xml.sax.xmlreader.**IncrementalParser**

在某些情况下，最好不要一次性地解析输入源，而是在可用的时候分块送入。请注意读取器通常不会读取整个文件，它同样也是分块读取的；并且 parse() 在处理完整个文档之前不会返回。所以如果不希望 parse() 出现阻塞行为则应当使用这些接口。

当解析器被实例化时它已准备好立即开始接受来自 feed 方法的数据。在通过调用 close 方法结束解析时 reset 方法也必须被调用以使解析器准备好接受新的数据，无论它是来自于 feed 还是使用 parse 方法。

请注意这些方法 不可在解析期间被调用，即在 parse 被调用之后及其返回之前。

默认情况下，该类还使用 IncrementalParser 接口的 feed, close 和 reset 方法来实现 XMLReader 接口的 parse 方法以方便 SAX 2.0 驱动的编写者。

class xml.sax.xmlreader.**Locator**

用于关联一个 SAX 事件与一个文档位置的接口。定位器对象只有在调用 DocumentHandler 的方法期间才会返回有效的结果；在其他任何时候，结果都是不可预测的。如果信息不可用，这些方法可能返回 None。

class xml.sax.xmlreader.**InputSource**(*system_id=None*)

*XMLReader*读取实体所需信息的封装。

这个类可能包括了关于公有标识符、系统标识符、字节流（可能带有字符编码格式信息）和/或一个实体的字符流的信息。

应用程序将创建这个类的对象以便在*XMLReader.parse()*方法中使用或是用于从 EntityResolver.resolveEntity 返回值。

*InputSource*属于应用程序，*XMLReader*不能修改从应用程序传递给它的*InputSource*对象，但它可以创建副本并进行修改。

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)

这是 Attributes 接口（参见*Attributes 接口*一节）的具体实现。这是一个 startElement() 调用中的元素属性的字典类对象。除了最有用处的字典操作，它还支持接口所描述的一些其他方法。该类的对象应当由读取器来实例化；*attrs* 必须为包含从属性名到属性值的映射的字典类对象。

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)

可感知命名空间的*AttributesImpl*变体形式，它将被传递给 startElementNS()。它派生自*AttributesImpl*，但会将属性名称解读为 *namespaceURI* 和 *localname* 二元组。此外，它还提供了一些期望接收在原始文档中出现的限定名称的方法。这个类实现了 AttributesNS 接口（参见*AttributesNS 接口*一节）。

21.12.1 XMLReader 对象

XMLReader 接口支持下列方法:

XMLReader.**parse**(*source*)

处理输入源，产生 SAX 事件。*source* 对象可以是一个系统标识符（标识输入源的字符串 -- 通常为文件名或 URL），*pathlib.Path* 或*路径类* 对象，或者是*InputSource* 对象。当*parse()* 返回时，输入会被全部处理完成，解析器对象可以被丢弃或重置。

在 3.5 版本发生变更: 添加了对字符流的支持。

在 3.8 版本发生变更: 增加了对路径类对象的支持。

XMLReader.**getContentHandler**()

返回当前的*ContentHandler*。

XMLReader.**setContentHandler**(*handler*)

设置当前的*ContentHandler*。如果没有设置*ContentHandler*，内容事件将被丢弃。

XMLReader.**getDTDHandler**()

返回当前的*DTDHandler*。

XMLReader.**setDTDHandler**(*handler*)

设置当前的*DTDHandler*。如果没有设置*DTDHandler*，DTD 事件将被丢弃。

XMLReader.**getEntityResolver**()

返回当前的*EntityResolver*。

XMLReader.**setEntityResolver**(*handler*)

设置当前的*EntityResolver*。如果没有设置*EntityResolver*，尝试解析一个外部实体将导致打开该实体的系统标识符，并且如果它不可用则操作将失败。

XMLReader.**getErrorHandler**()

返回当前的*ErrorHandler*。

XMLReader.**setErrorHandler**(*handler*)

设置当前的错误处理器。如果没有设置*ErrorHandler*，错误将作为异常被引发，并将打印警告信息。

XMLReader.**setLocale**(*locale*)

允许应用程序为错误和警告设置语言区域。

SAX 解析器不要求为错误和警告提供本地化信息；但是如果它们无法支持所请求的语言区域，则必须引发一个 SAX 异常。应用程序可以在解析的中途请求更改语言区域。

XMLReader.**getFeature**(*featurename*)

返回 *featurename* 特性的当前设置。如果特性无法被识别，则会引发 SAXNotRecognizedException。在*xml.sax.handler* 模块中列出了常见的特性名称。

XMLReader.**setFeature**(*featurename*, *value*)

将 *featurename* 设为 *value*。如果特性无法被识别，则会引发 SAXNotRecognizedException。如果特性或其设置不被解析器所支持，则会引发 *SAXNotSupportedException*。

XMLReader.**getProperty**(*propertyname*)

返回 *propertyname* 属性的当前设置。如果属性无法被识别，则会引发 SAXNotRecognizedException。在*xml.sax.handler* 模块中列出了常见的属性名称。

XMLReader.**setProperty**(*propertyname*, *value*)

将 *propertyname* 设为 *value*。如果属性无法被识别，则会引发 SAXNotRecognizedException。如果属性或其设置不被解析器所支持，则会引发 *SAXNotSupportedException*。

21.12.2 IncrementalParser 对象

IncrementalParser 的实例额外提供了下列方法:

IncrementalParser.**feed**(*data*)

处理 *data* 的一个分块。

IncrementalParser.**close**()

确定文档的结尾。这将检查只能在结尾处检查的格式是否良好的条件，唤起处理程序，并可能会清理在解析期间分配的资源。

IncrementalParser.**reset**()

此方法会在调用 close 来重置解析器以便其准备好解析新的文档之后被调用。在 close 之后未调用 reset 即调用 parse 或 feed 的结果是未定义的。

21.12.3 Locator 对象

Locator 的实例提供了下列方法:

Locator.**getColumnNumber**()

返回当前事件开始位置的列号。

Locator.**getLineNumber**()

返回当前事件开始位置的行号。

Locator.**getPublicId**()

返回当前事件的公有标识符。

Locator.**getSystemId**()

返回当前事件的系统标识符。

21.12.4 InputSource 对象

InputSource.**setPublicId**(*id*)

设置该*InputSource* 的公有标识符。

InputSource.**getPublicId**()

返回此*InputSource* 的公有标识符。

InputSource.**setSystemId**(*id*)

设置此*InputSource* 的系统标识符。

InputSource.**getSystemId**()

返回此*InputSource* 的系统标识符。

InputSource.**setEncoding**(*encoding*)

设置此*InputSource* 的字符编码格式。

编码格式必须是 XML 编码声明可接受的字符串（参见 XML 建议规范第 4.3.3 节）。

如果*InputSource* 还包含一个字符流则*InputSource* 的 encoding 属性会被忽略。

InputSource.**getEncoding**()

获取此 InputSource 的字符编码格式。

InputSource.**setByteStream**(*bytefile*)

设置此输入源的字节流（为*binary file* 对象）。

如果还指定了一个字符流被则 SAX 解析器会忽略此设置，但它将优先使用字节流而不是自己打开一个 URI 连接。

如果应用程序知道字节流的字符编码格式，它应当使用 setEncoding 方法来设置它。

InputSource.**getByteStream**()

获取此输入源的字节流。

getEncoding 方法将返回该字节流的字符编码格式，如果未知则返回 None。

InputSource.**setCharacterStream**(*charfile*)

设置此输入源的字符流 (为*text file* 对象)。

如果指定了一个字符流，SAX 解析器将忽略任何字节流并且不会尝试打开一个指向系统标识符的 URI 连接。

InputSource.**getCharacterStream**()

获取此输入源的字符流。

21.12.5 Attributes 接口

Attributes 对象实现了一部分映射协议，包括 copy(), get(), __contains__(), items(), keys() 和 values() 等方法。还提供了下列方法:

Attributes.**getLength**()

返回属性的数量。

Attributes.**getNames**()

返回属性的名称。

Attributes.**getType**(*name*)

返回属性 *name* 的类型，通常为 'CDATA'。

Attributes.**getValue**(*name*)

返回属性 *name* 的值。

21.12.6 AttributesNS 接口

此接口是 Attributes 接口（参见*Attributes 接口* 章节）的一个子类型。那个接口所支持的所有方法在 AttributesNS 对象上也都可用。

下列方法也是可用的:

AttributesNS.**getValueByQName**(*name*)

返回一个限定名称的值。

AttributesNS.**getNameByQName**(*name*)

返回限定名称 *name* 的 (namespace, localname) 对。

AttributesNS.**getQNameByName**(*name*)

返回 (namespace, localname) 对的限定名称。

AttributesNS.**getQNames**()

返回所有属性的限定名称。

21.13 xml.parsers.expat --- 使用 Expat 进行快速 XML 解析

警告

pyexpat 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

xml.parsers.expat 模块是针对 Expat 非验证 XML 解析器的 Python 接口。此模块提供了一个扩展类型 xmlparser，它代表一个 XML 解析器的当前状态。在创建一个 xmlparser 对象之后，该对象的各个属性可被设置为相应的处理器函数。随后当将一个 XML 文档送入解析器时，就会为该 XML 文档中的字符数据和标记调用处理器函数。

此模块使用 pyexpat 模块来提供对 Expat 解析器的访问。直接使用 pyexpat 模块的方式已被弃用。

此模块提供了一个异常和一个类型对象:

exception xml.parsers.expat.**ExpatError**

此异常会在 Expat 报错时被引发。请参阅*ExpatError 异常* 一节了解有关解读 Expat 错误的更多信息。

exception xml.parsers.expat.**error**

ExpatError 的别名。

xml.parsers.expat.**XMLParserType**

来自*ParserCreate()* 函数的返回值的类型。

xml.parsers.expat 模块包含两个函数:

xml.parsers.expat.**ErrorString**(*errno*)

返回给定错误号 *errno* 的解释性字符串。

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)

创建并返回一个新的 xmlparser 对象。如果指定了 *encoding*，它必须为指定 XML 数据所使用的编码格式名称的字符串。Expat 支持的编码格式没有 Python 那样多，而且它的编码格式库也不能被扩展；它支持 UTF-8, UTF-16, ISO-8859-1 (Latin1) 和 ASCII。如果给出了 *encoding*[1] 则它将覆盖隐式或显式指定的文档编码格式。

可以选择让 Expat 为你做 XML 命名空间处理，这是通过提供 *namespace_separator* 值来启用的。该值必须是一个单字符的字符串；如果字符串的长度不合法则将引发*ValueError* (None 被视为等同于省略)。当命名空间处理被启用时，属于特定命名空间的元素类型名称和属性名称将被展开。传递给 The element name passed to the 元素处理器 StartElementHandler 和 EndElementHandler 的元素名称将为命名空间 URI，命名空间分隔符和名称的本地部分的拼接。如果命名空间分隔符是一个零字节 (chr(0)) 则命名空间 URI 和本地部分将被直接拼接而不带任何分隔符。

举例来说，如果 *namespace_separator* 被设为空格符 (' ') 并对以下文档进行解析:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler 将为每个元素获取以下字符串:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

由于 pyexpat 所使用的 Expat 库的限制，被返回的 xmlparser 实例只能被用来解析单个 XML 文档。请为每个文档调用 ParserCreate 来提供单独的解析器实例。

参见

The Expat XML Parser
Expat 项目的主页。

[1] 包括在 XML 输出中的编码格式字符串应当符合适当的标准。例如"UTF-8" 是有效的，但"UTF8" 是无效的。请参阅 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

21.13.1 XMLParser 对象

xmlparser 对象具有以下方法:

xmlparser.**Parse**(*data*[, *isfinal*])

解析字符串 *data* 的内容，调用适当的处理函数来处理解析后的数据。在对此方法的最后一次调用时 *isfinal* 必须为真值；它允许以片段形式解析单个文件，而不是提交多个文件。*data* 在任何时候都可以为空字符串。

xmlparser.**ParseFile**(*file*)

解析从对象 *file* 读取的 XML 数据。*file* 仅需提供 `read(nbytes)` 方法，当没有更多数据可读时将返回空字符串。

xmlparser.**SetBase**(*base*)

设置要用于解析声明中的系统标识符的相对 URI 的基准。解析相对标识符的任务会留给应用程序进行：这个值将作为 *base* 参数传递给*ExternalEntityRefHandler()*, *NotationDeclHandler()* 和*UnparsedEntityDeclHandler()* 函数。

xmlparser.**GetBase**()

返回包含之前调用*SetBase()* 所设置的基准位置的字符串，或者如果未调用*SetBase()* 则返回 None。

xmlparser.**GetInputContext**()

将生成当前事件的输入数据以字符串形式返回。数据为包含文本的实体的编码格式。如果被调用时未激活事件处理器，则返回值将为 None。

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])

创建一个“子”解析器，可被用来解析由父解析器解析的内容所引用的外部解析实体。*context* 形参应当是传递给*ExternalEntityRefHandler()* 处理函数的字符串，具体如下所述。子解析器创建时*ordered_attributes* 和*specified_attributes* 会被设为此解析器的值。

xmlparser.**SetParamEntityParsing**(*flag*)

控制参数实体（包括外部 DTD 子集）的解析。可能的 *flag* 值有 XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE 和 XML_PARAM_ENTITY_PARSING_ALWAYS。如果该旗标设置成功则返回真值。

xmlparser.**UseForeignDTD**([*flag*])

调用时将 *flag* 设为真值（默认）将导致 Expat 调用*ExternalEntityRefHandler* 时将所有参数设为*None* 以允许加载替代的 DTD。如果文档不包含文档类型声明，*ExternalEntityRefHandler* 仍然会被调用，但*StartDoctypeDeclHandler* 和*EndDoctypeDeclHandler* 将不会被调用。

为 *flag* 传入假值将将撤消之前传入真值的调用，除此之外没有其他影响。

此方法只能在调用*Parse()* 或*ParseFile()* 方法之前被调用；在已调用过这两个方法之后调用它会导致引发*ExpatError* 且*code* 属性被设为 `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`。

xmlparser.**SetReparseDeferralEnabled**(*enabled*)

> **警告**
>
> 调用 `SetReparseDeferralEnabled(False)` 会对安全产生影响，详情见下文；在使用 `SetReparseDeferralEnabled` 方法之前请务必了解这些后果。

Expat 2.6.0 引入了一种名为“重新解析延迟”的安全机制，这种机制可避免因重新解析大量词元的指数级运行时间而导致的拒绝服务，而是会在默认情况下延迟对未完成词元的重新解析直至达到足够的输入量。由于这种延迟，已注册的处理器有可能—具体取决于推送到 Expat 的输入块大小—不会在向解析器推送新输入后立即被调用。如果希望获得即时反馈并接管防止大量词元因大量词元导致的拒绝服务的责任，可以调用 `SetReparseDeferralEnabled(False)` 暂时或完全禁用当前 Expat 解析器实例的重新解析延迟。调用 `SetReparseDeferralEnabled(True)` 可以再次启用重新解析延迟。

请注意*SetReparseDeferralEnabled()* 已作为安全修正被向下移植到一些较早的 CPython 发布版。如果在运行于多个 Python 版本的代码中要用到*SetReparseDeferralEnabled()* 请使用*hasattr()* 来检查其可用性。

Added in version 3.13.

xmlparser.**GetReparseDeferralEnabled**()

返回当前是否为给定的 Expat 解析器实例启用了重新解析延迟。

Added in version 3.13.

xmlparser 对象具有下列属性:

xmlparser.**buffer_size**

当*buffer_text* 为真值时所使用的缓冲区大小。可以通过将此属性赋一个新的整数值来设置一个新的缓冲区大小。当大小发生改变时，缓冲区将被刷新。

xmlparser.**buffer_text**

将此属性设为真值会使得 xmlparser 对象缓冲 Expat 所返回的文本内容以尽可能地避免多次调用*CharacterDataHandler()* 回调。这可以显著地提升性能，因为 Expat 通常会将字符数据在每个行结束的位置上进行分块。此属性默认为假值，并可在任何时候被更改。请注意当其为假值时，不包含换行符的数据也可能被分块。

xmlparser.**buffer_used**

当*buffer_text* 被启用时，缓冲区中存储的字节数。这些字节数据表示以 UTF-8 编码的文本。当*buffer_text* 为假值时此属性没有任何实际意义。

xmlparser.**ordered_attributes**

将该属性设为非零整数会使得各个属性被报告为列表而非字典。各个属性会按照在文档文本中的出现顺序显示。对于每个属性，将显示两个列表条目：属性名和属性值。（该模块的较旧版本也使用了此格式。）默认情况下，该属性为假值；它可以在任何时候被更改。

xmlparser.**specified_attributes**

如果设为非零整数，解析器将只报告在文档实例中指明的属性而不报告来自属性声明的属性。设置此属性的应用程序需要特别小心地使用从声明中获得的附加信息以符合 XML 处理程序的行为标准。默认情况下，该属性为假值；它可以在任何时候被更改。

下列属性包含与 xmlparser 对象遇到的最近发生的错误有关联的值，并且一旦对 Parse() 或 ParseFile() 的调用引发了*xml.parsers.expat.ExpatError* 异常就将只包含正确的值。

xmlparser.**ErrorByteIndex**

错误发生位置的字节索引号。

xmlparser.**ErrorCode**

指明问题的的数字代码。该值可被传给*ErrorString()* 函数，或是与在 errors 对象中定义的常量之一进行比较。

xmlparser.**ErrorColumnNumber**

错误发生位置的列号。

xmlparser.**ErrorLineNumber**

错误发生位置的行号。

下列属性包含 xmlparser 对象中关联到当前解析位置的值。在回调报告解析事件期间它们将指示生成事件的字符序列的第一个字符的位置。当在回调的外部被调用时，所指示的位置将恰好位于最后的解析事件之后（无论是否存在关联的回调）。

xmlparser.**CurrentByteIndex**

解析器输入的当前字节索引号。

xmlparser.**CurrentColumnNumber**

解析器输入的当前列号。

xmlparser.**CurrentLineNumber**

解析器输入的当前行号。

可被设置的处理器列表。要在一个 xmlparser 对象 *o* 上设置处理器，请使用 o.handlername = func。*handlername* 必须从下面的列表中获取，而 *func* 必须为接受正确数量参数的可调用对象。所有参数均为字符串，除非另外指明。

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)

当解析 XML 声明时被调用。XML 声明是 XML 建议适用版本、文档文本的编码格式，以及可选的“独立”声明的（可选）声明。*version* 和 *encoding* 将为字符串，而 *standalone* 在文档被声明为独立时将为 1，在文档被声明为非独立时将为 0，或者在 standalone 短语被省略时则为 -1。这仅适用于 Expat 的 1.95.0 或更新版本。

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)

当 Expat 开始解析文档类型声明 (<!DOCTYPE ...) 时被调用。*doctypeName* 会完全按所显示的被提供。*systemId* 和 *publicId* 形参给出所指定的系统和公有标识符，如果被省略则为 None。如果文档包含内部文档声明子集则 *has_internal_subset* 将为真值。这要求 Expat 1.2 或更新的版本。

xmlparser.**EndDoctypeDeclHandler**()

当 Expat 完成解析文档类型声明时被调用。这要求 Expat 1.2 或更新版本。

xmlparser.**ElementDeclHandler**(*name*, *model*)

为每个元素类型声明调用一次。*name* 为元素类型名称，而 *model* 为内容模型的表示形式。

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)

为一个元素类型的每个已声明属性执行调用。如果一个属性列表声明声明了三个属性，这个处理器会被调用三次，每个属性一次。*elname* 是声明所适用的元素的名称而 *attname* 是已声明的属性的名称。属性类型是作为 *type* 传入的字符串；可能的值有 'CDATA', 'ID', 'IDREF', ... *default* 给出了当属性未被文档实例所指明时该属性的默认值，或是为 None，如果没有默认值 (#IMPLIED 值) 的话。如果属性必须在文档实例中给出，则 *required* 将为真值。这要求 Expat 1.95.0 或更新的版本。

xmlparser.**StartElementHandler**(*name*, *attributes*)

在每个元素开始时调用。*name* 是包含元素名称的字符串，而 *attributes* 是元素的属性。如果*ordered_attributes* 为真值，则属性为列表形式 (完整描述参见*ordered_attributes*)。否则为将名称映射到值的字典。

xmlparser.**EndElementHandler**(*name*)

在每个元素结束时调用。

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)

在每次处理指令时调用。

xmlparser.**CharacterDataHandler**(*data*)

针对字符数据调用。此方法将被用于普通字符数据、CDATA 标记的内容以及可忽略的空白符。需要区别这几种情况的应用程序可以使用*StartCdataSectionHandler*, *EndCdataSectionHandler* 和*ElementDeclHandler* 回调来收集必要的信息。请注意字符数据即使很短可能会被分块并且你可能会收到多个对*CharacterDataHandler()* 的调用。请将*buffer_text* 实例属性设为 True 来避免此情况。

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)

针对未解析（NDATA）实体声明调用。此方法仅存在于 Expat 库的 1. 2 版；对于更新的版本，请改用*EntityDeclHandler*。(下层 Expat 库中的对应函数已被声明为过时。)

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)

针对所有实体声明被调用。对于形参和内部实体，*value* 将为给出实体的声明内容的字符串；对于外部实体将为 None。*notationName* 形参对于已解析实体将为 None，对于未解析实体则为标注的名称。如果实体为形参实体则 *is_parameter_entity* 将为真值而如果为普通实体则为假值（大多数应用程序只需要关注普通实体）。此方法仅从 1.95.0 版 Expat 库开始才可用。

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)

针对标注声明被调用。*notationName*, *base*, *systemId* 和 *publicId* 如果给出则均应为字符串。如果省略公有标识符，则 *publicId* 将为 None。

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)

当一个元素包含命名空间声明时被调用。命名空间声明会在为声明所在的元素调用*StartElementHandler* 之前被处理。

xmlparser.**EndNamespaceDeclHandler**(*prefix*)

当到达包含命名空间声明的元素的关闭标记时被调用。此方法会按照调用*StartNamespaceDeclHandler* 以指明每个命名空间作用域的开始的逆顺序为元素上的每个命名空间声明调用一次。对这个处理器的调用是在相应的*EndElementHandler* 之后针对元素的结束而进行的。

xmlparser.**CommentHandler**(*data*)

针对注释被调用。*data* 是注释的文本，不包括开头的 '<!--' 和末尾的 '-->'。

xmlparser.**StartCdataSectionHandler**()

在一个 CDATA 节的开头被调用。需要此方法和*EndCdataSectionHandler* 以便能够标识 CDATA 节的语法开始和结束。

xmlparser.**EndCdataSectionHandler**()

在一个 CDATA 节的末尾被调用。

xmlparser.**DefaultHandler**(*data*)

针对 XML 文档中没有指定适用处理器的任何字符被调用。这包括了所有属于可被报告的结构的一部分，但未提供处理器的字符。

xmlparser.**DefaultHandlerExpand**(*data*)

这与*DefaultHandler()* 相同，但不会抑制内部实体的扩展。实体引用将不会被传递给默认处理器。

xmlparser.**NotStandaloneHandler**()

当 XML 文档未被声明为独立文档时被调用。这种情况发生在出现外部子集或对参数实体的引用，但 XML 声明没有在 XML 声明中将 standalone 设为 yes 的时候。如果这个处理器返回 0，那么解析器将引发 XML_ERROR_NOT_STANDALONE 错误。如果这个处理器没有被设置，那么解析器就不会为这个条件引发任何异常。

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)

为对外部实体的引用执行调用。*base* 为当前的基准，由之前对*SetBase()* 的调用设置。公有和系统标识符 *systemId* 和 *publicId* 如果给出则[F]为字符串；如果公有标识符未给出，则 *publicId* 将为 None。*context* 是仅根据以下说明来使用的不透明值。

对于要解析的外部实体，这个处理器必须被实现。它负责使用 ExternalEntityParserCreate(context) 来创建子解析器，通过适当的回调将其初始化，并对实体进行解析。这个处理器应当返回一个整数；如果它返回 0，则解析器将引发 XML_ERROR_EXTERNAL_ENTITY_HANDLING 错误，否则解析将会继续。

如果未提供这个处理器，外部实体会由*DefaultHandler* 回调来报告，如果提供了该回调的话。

21.13.2 ExpatError 异常

ExpatError 异常包含几个有趣的属性:

ExpatError.**code**

Expat 对于指定错误的内部错误号。*errors.messages* 字典会将这些错误号映射到 Expat 的错误消息。例如:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
```

(续下页)

(接上页)

```
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

errors 模块也提供了一些错误消息常量和一个将这些消息映射回错误码的字典*codes*，参见下文。

ExpatError.**lineno**

检测到错误所在的行号。首行的行号为 1。

ExpatError.**offset**

错误发生在行中的字符偏移量。首列的列号为 0。

21.13.3 示例

以下程序定义了三个处理器，会简单地打印出它们的参数。：

```
import xml.parsers.expat

# 3 处理器函数
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

来自这个程序的输出是:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

21.13.4 内容模型描述

内容模型是使用嵌套的元组来描述的。每个元素包含四个值：类型、限定符、名称和一个子元组。子元组就是附加的内容模型描述。

前两个字段的值是在*xml.parsers.expat.model* 模块中定义的常量。这些常量可分为两组：模型类型组和限定符组。

模型类型组中的常量有:

xml.parsers.expat.model.**XML_CTYPE_ANY**
模型名称所指定的元素被声明为具有 ANY 内容模型。

xml.parsers.expat.model.**XML_CTYPE_CHOICE**
命名元素允许从几个选项中选择；这被用于 (A | B | C) 形式的内容模型。

xml.parsers.expat.model.**XML_CTYPE_EMPTY**
被声明为 EMPTY 的元素具有此模型类型。

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**
代表彼此相连的一系列模型的模型用此模型类型来指明。这被用于 (A, B, C) 形式的模型。

限定符组中的常量有:

xml.parsers.expat.model.**XML_CQUANT_NONE**
未给出限定符，这样它可以只出现一次，例如 A。

xml.parsers.expat.model.**XML_CQUANT_OPT**
模型是可选的：它可以出现一次或完全不出现，例如 A?。

xml.parsers.expat.model.**XML_CQUANT_PLUS**
模型必须出现一次或多次 (例如 A+)。

xml.parsers.expat.model.**XML_CQUANT_REP**
模型必须出现零次或多次，例如 A*。

21.13.5 Expat 错误常量

下列常量是在*xml.parsers.expat.errors* 模块中提供的。这些常量在有错误发生时解读被引发的 ExpatError 异常对象的某些属性时很有用处。出于保持向下兼容性的理由，这些常量的值是错误消息而不是数字形式的错误代码，为此你可以将它的*code* 属性和 errors.codes[errors.XML_ERROR_*CONSTANT_NAME*] 进行比较。

errors 模块具有以下属性:

xml.parsers.expat.errors.**codes**
将字符串描述映射到其错误代码的字典。

Added in version 3.2.

xml.parsers.expat.errors.**messages**
将数字形式的错误代码映射到其字符串描述的字典。

Added in version 3.2.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
属性值中指向一个外部实体而非内部实体的实体引用。

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
指向一个在 XML 不合法的字符的字符引用 (例如，字符 0 或'&#0;')。

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
指向一个使用标注声明，因而无法被解析的实体的实体引用。

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
一个属性在一个开始标记中被使用超过一次。

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
当一个输入字节无法被正确分配给一个字符时引发；例如，在 UTF-8 输入流中的 NUL 字节 (值为 0)。

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
在文档元素之后出现空白符以外的内容。

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
在输入数据开始位置以外的地方发现 XML 声明。

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
文档不包含任何元素（XML 要求所有文档都包含恰好一个最高层级元素）。

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat 无法在内部分配内存。

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**
在不被允许的位置发现一个参数实体引用。

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**
在输入中发出一个不完整的字符。

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**
一个实体引用包含了对同一实体的另一个引用；可能是通过不同的名称，并可能是间接的引用。

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**
遇到了某个未指明的语法错误。

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**
一个结束标记不能匹配到最内层的未关闭开始标记。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**
某些记号（例如开始标记）在流结束或遇到下一个记号之前还未关闭。

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**
对一个未定义的实体进行了引用。

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**
文档编码格式不被 Expat 所支持。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**
一个 CDATA 标记节还未关闭。

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**
解析器确定文档不是“独立的”但它却在 XML 声明中声明自己是独立的，并且 NotStandaloneHandler 被设置为返回 0。

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
请求了一个需要已编译 DTD 支持的操作，但 Expat 被配置为不带 DTD 支持。此错误应当绝对不会被*xml.parsers.expat* 模块的标准构建版本所报告。

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
在解析开始之后请求一个只能在解析开始之前执行的行为改变。此错误（目前）只能由 UseForeignDTD() 所引发。

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**
当命名空间处理被启用时发现一个未声明的前缀。

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**
文档试图移除与某个前缀相关联的命名空间声明。

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**
一个参数实体包含不完整的标记。

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**
文档完全未包含任何文档元素。

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**
解析一个外部实体中的文本声明时出现错误。

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**
在公有 id 中发现不被允许的字符。

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**
在挂起的解析器上请求执行操作，但未获得允许。这包括提供额外输入或停止解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**
在解析器未被挂起的时候执行恢复解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_ABORTED**
此错误不应当被报告给 Python 应用程序。

xml.parsers.expat.errors.**XML_ERROR_FINISHED**
在一个已经完成解析输入的解析器上请求执行操作，但未获得允许。这包括提供额外输入或停止解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XML**
有人试图撤销保留的命名空间前缀 `xml` 或将其绑定到另一个命名空间 URI。

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XMLNS**
有人试图声明或撤销保留的命名空间前缀 `xmlns`。

xml.parsers.expat.errors.**XML_ERROR_RESERVED_NAMESPACE_URI**
有人试图将一个保留的命名空间前缀 `xml` 和 `xmlns` 的 URI 绑定到另一个命名空间前缀。

xml.parsers.expat.errors.**XML_ERROR_INVALID_ARGUMENT**
此错误不应当被报告给 Python 应用程序。

xml.parsers.expat.errors.**XML_ERROR_NO_BUFFER**
此错误不应当被报告给 Python 应用程序。

xml.parsers.expat.errors.**XML_ERROR_AMPLIFICATION_LIMIT_BREACH**
输入放大系数的限制（来自 DTD 和实体）已被突破。

备注

CHAPTER 22

互联网协议和支持

本章介绍的模块实现了互联网协议以及相关技术支持。它们都是用 Python 实现的。这些模块大多需要依赖于系统的模块*socket* 作为前提，该模块在大多数流行系统平台上都受到支持。下面是一份概览:

22.1 webbrowser --- 方便的 Web 浏览器控制工具

源码： Lib/webbrowser.py

webbrowser 模块提供了一个高层级接口，允许向用户显示基于 Web 的文档。在大多数情况下，只需调用此模块的*open()* 函数就可以了。

在 Unix 下，图形浏览器在 X11 下是首选，但如果图形浏览器不可用或 X11 显示不可用，则将使用文本模式浏览器。如果使用文本模式浏览器，则调用进程将阻塞，直到用户退出浏览器。

如果存在环境变量 BROWSER ，则将其解释为*os.pathsep* 分隔的浏览器列表，以便在平台默认值之前尝试。当列表部分的值包含字符串 %s 时，它被解释为一个文字浏览器命令行，用于替换 %s 的参数 URL ；如果该部分不包含 %s，则它只被解释为要启动的浏览器的名称。[1]

对于非 Unix 平台，或者当 Unix 上有远程浏览器时，控制过程不会等待用户完成浏览器，而是允许远程浏览器在显示界面上维护自己的窗口。如果 Unix 上没有远程浏览器，控制进程将启动一个新的浏览器并等待。

在 iOS 上，BROWSER 环境变量以及任何控制自动唤起、浏览器首选项和新选项卡/窗口创建的参数都将被忽略。Web 页面将 *总是* 在用户首选的浏览器中打开一个新选项卡，并且此浏览器将被切换到前台。在 iOS 上使用*webbrowser* 模块需要有*ctypes* 模块。如果*ctypes* 不可用，对*open()* 的调用将会失败。

脚本 **webbrowser** 可作为该模块的命令行接口来使用。它接受一个 URL 作为参数。它还接受下列可选形参:

-n, --new-window

在新的浏览器窗口中打开 URL，如果可能的话。

-t, --new-tab

在新的浏览器选项卡中打开 URL。

这些选项自然是互斥的。用法示例:

[1] 这里命名的不带完整路径的可执行文件将在 PATH 环境变量给出的目录中搜索。

```
python -m webbrowser -t "https://www.python.org"
```

Availability: not WASI, not Android.

定义了以下异常：

exception webbrowser.**Error**

发生浏览器控件错误时引发异常。

定义了以下函数：

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)

使用默认浏览器显示 *url*。如果 *new* 为 0，则尽可能在同一浏览器窗口中打开 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。如果 *autoraise* 为 “True”，则会尽可能置前窗口（请注意，在许多窗口管理器下，无论此变量的设置如何，都会置前窗口）。

如果浏览器成功启动则返回 `True`，否则返回 `False`。

请注意，在某些平台上，尝试使用此函数打开文件名，可能会起作用并启动操作系统的关联程序。但是，这种方式不被支持也不可移植。

引发一个审计事件 `webbrowser.open` 并附带参数 `url`。

webbrowser.**open_new**(*url*)

如果可能，在默认浏览器的新窗口中打开 *url*，否则，在唯一的浏览器窗口中打开 *url*。

如果浏览器成功启动则返回 `True`，否则返回 `False`。

webbrowser.**open_new_tab**(*url*)

如果可能，在默认浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

如果浏览器成功启动则返回 `True`，否则返回 `False`。

webbrowser.**get**(*using=None*)

返回浏览器类型为 *using* 指定的控制器对象。如果 *using* 为 `None`，则返回适用于调用者环境的默认浏览器的控制器。

webbrowser.**register**(*name*, *constructor*, *instance=None*, *, *preferred=False*)

注册 *name* 浏览器类型。注册浏览器类型后，*get()* 函数可以返回该浏览器类型的控制器。如果没有提供 *instance*，或者为 `None`，*constructor* 将在没有参数的情况下被调用，以在需要时创建实例。如果提供了 *instance*，则永远不会调用 *constructor*，并且可能是 `None`。

将 *preferred* 设置为 `True` 使得这个浏览器成为*get()* 不带参数调用的首选结果。否则，只有在您计划设置 `BROWSER` 变量，或使用与您声明的处理程序的名称相匹配的非空参数调用*get()* 时，此入口点才有用。

在 3.7 版本发生变更: 添加了仅关键字参数 *preferred*。

预定义了许多浏览器类型。此表给出了可以传递给*get()* 函数的类型名称以及控制器类的相应实例化，这些都在此模块中定义。

类型名	类名	备注
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'epiphany'	Epiphany('epiphany')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'opera'	Opera()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSXOSAScript('default')	(3)
'safari'	MacOSXOSAScript('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	
'iosbrowser'	IOSBrowser	(4)

注释:

(1) "Konqueror" 是用于 Unix 的 KDE 桌面环境的文件管理器，只有在运行了 KDE 时才有意义。某些能可靠地检测 KDE 的方法会很有用处；仅检查 KDEDIR 变量是不够的。还要注意即使在使用 KDE 2 的 **konqueror** 命令时会使用也会使用"kfm" 这一名称 --- 该实现会为运行 Konqueror 选择最佳的策略。

(2) 仅限 Windows 平台。

(3) 仅限于 macOS。

(4) 仅限于 iOS。

Added in version 3.2: 在 Mac 上会使用新增的 MacOSXOSAScript 类而不是之前的 MacOSX 类。它增加了对打开当前未被设为 OS 默认首选项的浏览器的支持。

Added in version 3.3: 添加了对 Chrome/Chromium 的支持。

在 3.12 版本发生变更: 对某些过时浏览器的支持已被移除。被移除的浏览器包括 Grail, Mosaic, Netscape, Galeon, Skipstone, Iceape 和 Firefox 35 及以下的版本。

在 3.13 版本发生变更: 添加了对 iOS 的支持。

以下是一些简单的例子:

```
url = 'https://docs.python.org/'

# 在新选项卡中打开 URL，如果已打开一个浏览器窗口的话。
webbrowser.open_new_tab(url)

# 在新窗口中打开 URL，如有可能将启动窗口。
webbrowser.open_new(url)
```

22.1.1 浏览器控制器对象

浏览器控制器提供了*name* 属性，和以下三个与模块级便捷函数相对应的方法:

controller.**name**

浏览器依赖于系统的名称。

controller.**open**(*url*, *new=0*, *autoraise=True*)

使用此控制器处理的浏览器显示 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。

controller.**open_new**(*url*)

如果可能，在此控制器处理的浏览器的新窗口中打开 *url* ，否则，在唯一的浏览器窗口中打开 *url* 。别名*open_new()*。

controller.**open_new_tab**(*url*)

如果可能，在此控制器处理的浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

备注

22.2 wsgiref --- WSGI 工具和参考实现

源代码: Lib/wsgiref

Web 服务器网关接口（WSGI）是 Web 服务器软件和用 Python 编写的 Web 应用程序之间的标准接口。具有标准接口能够让支持 WSGI 的应用程序与多种不同的 Web 服务器配合使用。

只有 Web 服务器和编程框架的开发者才需要了解 WSGI 设计的每个细节和边界情况。你不需要了解 WSGI 的每个细节而只需要安装一个 WSGI 应用程序或编写使用现有框架的 Web 应用程序。

wsgiref 是 WSGI 规范的一个参考实现，可被用于将 WSGI 支持添加到 Web 服务器或框架中。它提供了操作 WSGI 环境变量和响应标头的工具，实现 WSGI 服务器的基类，可部署 WSGI 应用程序的演示性 HTTP 服务器，用于静态类型检查的类型，以及一个用于检查 WSGI 服务器和应用程序是否符合 WSGI 规范的验证工具 (**PEP 3333**)。

参看 wsgi.readthedocs.io 获取有关 WSGI 的更多信息，以及教程和其他资源的链接。

22.2.1 wsgiref.util -- WSGI 环境工具

本模块提供了多种工具函数配合 WSGI 环境使用。WSGI 环境就是一个包含 **PEP 3333** 所描述的 HTTP 请求变量的字典。所有接受 *environ* 形参的函数都会预期得到一个符合 WSGI 规范的字典；请参阅 **PEP 3333** 来了解相关规范的细节并请参阅*WSGIEnvironment* 来了解可被用于类型标注的类型别名。

wsgiref.util.**guess_scheme**(*environ*)

返回对于 wsgi.url_scheme 应为”http” 还是”https” 的猜测，具体方式是在 *environ* 中检查 HTTPS 环境变量。返回值是一个字符串。

此函数适用于创建一个包装了 CGI 或 CGI 类协议例如 FastCGI 的网关。通常，提供这种协议的服务器将包括一个 HTTPS 变量并会在通过 SSL 接收请求时将其值设为”1”, ”yes” 或”on”。这样，此函数会在找到上述值时返回”https”，否则返回”http”。

wsgiref.util.**request_uri**(*environ*, *include_query=True*)

使用 **PEP 3333** 的”URL Reconstruction” 一节中的算法返回完整的请求 URL，可能包括查询字符串。如果 *include_query* 为假值，则结果 URI 中将不包括查询字符串。

wsgiref.util.**application_uri**(*environ*)

类似于*request_uri()*，区别在于 PATH_INFO 和 QUERY_STRING 变量会被忽略。结果为请求所指定的应用程序对象的基准 URI。

wsgiref.util.**shift_path_info**(*environ*)

将单个名称从 PATH_INFO 变换为 SCRIPT_NAME 并返回该名称。*environ* 目录将被原地 修改；如果你需要保留原始 PATH_INFO 或 SCRIPT_NAME 不变请使用一个副本。

如果 PATH_INFO 中没有剩余的路径节，则返回 None。

通常，此例程被用来处理请求 URI 路径的每个部分，比如说将路径当作是一系列字典键。此例程会修改传入的环境以使其适合唤起位于目标 URI 上的其他 WSGI 应用程序，如果有一个 WSGI 应用程序位于 /foo，而请求 URI 路径为 /foo/bar/baz，且位于 /foo 的 WSGI 应用程序调用

了*shift_path_info()*，它将获得字符串"bar"，而环境将被更新以适合传递给位于 /foo/bar 的 WSGI 应用程序。也就是说，SCRIPT_NAME 将由 /foo 变为 /foo/bar，而 PATH_INFO 将由 /bar/baz 变为 /baz。

当 PATH_INFO 只是"/" 时，此例程会返回一个空字符串并在 SCRIPT_NAME 末尾添加一个斜杠，虽然空的路径节通常会被忽略，并且 SCRIPT_NAME 通常也不以斜杠作为结束。此行为是有意为之的，用来确保应用程序在使用此例程执行对象遍历时能区分以 /x 结束的和以 /x/ 结束的 URI。

wsgiref.util.**setup_testing_defaults**(*environ*)

以简短的默认值更新 *environ* 用于测试目的。

此例程会添加 WSGI 所需要的各种参数，包括 HTTP_HOST，SERVER_NAME，SERVER_PORT，REQUEST_METHOD，SCRIPT_NAME，PATH_INFO 以及 **PEP 3333** 中定义的所有 wsgi.* 变量。它只提供默认值，而不会替换这些变量的现有设置。

此例程的目的是让 WSGI 服务器的单元测试以及应用程序设置测试环境更为容易。它不应该被实际的 WSGI 服务器或应用程序所使用，因为它用的是假数据！

用法示例 (另请参阅*demo_app()* 中的其他示例):

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# 一个相对简单的 WSGI 应用程序。 它将打印出
# 由 setup_testing_defaults 更新之后的环境字典
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

除了上述的环境函数，*wsgiref.util* 模块还提供了以下辅助工具:

wsgiref.util.**is_hop_by_hop**(*header_name*)

如果'header_name' 是 **RFC 2616** 所定义的 HTTP/1.1 "Hop-by-Hop" 标头则返回 True。

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)

一个*wsgiref.types.FileWrapper* 协议的具体实现被用来将文件型对象转换为*iterator*。结果对象将为*iterable*。当对象被迭代时，可选的 *blksize* 形参将被反复地传给 *filelike* 对象的 read() 方法以获取字节器并产生输出。当 read() 返回空字节串时，迭代将结束并且不可再恢复。

如果 *filelike* 具有 close() 方法，返回的对象也将具有 close() 方法，并且它将在被调用时唤起 *filelike* 对象的 close() 方法。

用法示例:

```
from io import StringIO
from wsgiref.util import FileWrapper

# 我们使用一个 StringIO 缓冲区作为文件型对象
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)
```

(续下页)

(接上页)

```
for chunk in wrapper:
    print(chunk)
```

在 3.11 版本发生变更: 对 __getitem__() 方法的支持已被移除。

22.2.2 wsgiref.headers -- WSGI 响应标头工具

此模块提供了一个单独的类*Headers*，可以方便地使用一个映射类接口操作 WSGI 响应标头。

class wsgiref.headers.**Headers**([*headers*])

创建一个包装了 *headers* 的映射类对象，它必须为如 **PEP 3333** 所描述的由标头名称/值元组构成的列表。*headers* 的默认值为一个空列表。

Headers 对象支持典型的映射操作包括 __getitem__(), *get()*, __setitem__(), *setdefault()*, __delitem__() 和 __contains__()。对于以上各个方法，映射的键都是标头名称（大小写不敏感），而值则是关联到该标头名称的第一个值。设置一个标头会移除该标头的任何现有值，再将一个新值添加到所包装的标头列表末尾。标头的现有顺序通常会保持不变，新的标头会被添加到所包装的列表末尾。

与字典不同，当你试图获取或删除所包装的标头列表中不存在的键时*Headers* 对象不会引发错误。获取一个不存在的标头只是返回 None，而删除一个不存在的标头则没有任何影响。

Headers 对象还支持 keys(), values() 以及 items() 方法。如果存在具有多个值的键则 keys() 和 items() 所返回的列表可能包括多个相同的键。对*Headers* 对象执行 len() 的结果与 items() 的长度相同，也与所包装的标头列表的长度相同。实际上，items() 方法只是返回所包装的标头列表的一个副本。

在*Headers* 对象上调用 bytes() 将返回适用于作为 HTTP 响应标头来传输的已格式化字节串。每个标头附带以逗号加空格分隔的值放置在一行中。每一行都以回车符加换行符结束，且该字节串会以一个空行作为结束。

除了映射接口和格式化特性，*Headers* 对象还具有下列方法用来查询和添加多值标头，以及添加具有 MIME 参数的标头:

get_all(*name*)

返回包含指定标头的所有值的列表。

返回的列表项将按它们在原始标头列表中的出现或被添加到实例中的顺序排序，并可能包含重复项。任何被删除并重新插入的字段总是会被添加到标头列表末尾。如果给定名称的字段不存在，则返回一个空列表。

add_header(*name*, *value*, ***_params*)

添加一个（可能有多个值）标头，带有通过关键字参数指明的可选的 MIME 参数。

name 是要添加的标头字段。可以使用关键字参数来为标头字段设置 MIME 参数。每个参数必须为字符串或 None。参数名中的下划线会被转换为连字符，因为连字符不可在 Python 标识符中出现，但许多 MIME 参数名都包括连字符。如果参数值为字符串，它会以 name="value" 的形式被添加到标头值参数中。如果为 None，则只会添加参数名。(这适用于没有值的 MIME 参数。）示例用法:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

以上代码将添加一个这样的标头:

```
Content-Disposition: attachment; filename="bud.gif"
```

在 3.5 版本发生变更: *headers* 形参是可选的。

22.2.3 wsgiref.simple_server -- 一个简单的 WSGI HTTP 服务器

此模块实现了一个简单的 HTTP 服务器 (基于*http.server*) 来发布 WSGI 应用程序。每个服务器实例会在给定的主机名和端口号上发布一个 WSGI 应用程序。如果你想在一个主机名和端口号上发布多个应用程序，你应当创建一个通过解析 PATH_INFO 来选择每个请求要唤起哪个应用程序的 WSGI 应用程序。(例如，使用*wsgiref.util* 中的 shift_path_info() 函数。)

wsgiref.simple_server.**make_server**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

创建一个新的 WSGI 服务器并在 *host* 和 *port* 上进行监听，接受对 *app* 的连接。返回值是所提供的 *server_class* 的实例，并将使用指定的 *handler_class* 来处理请求。*app* 必须是一个如 **PEP 3333** 所定义的 WSGI 应用程序对象。

用法示例:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # 响应请求直到进程被杀死
    httpd.serve_forever()

    # 或者：服务一次请求，然后退出
    httpd.handle_request()
```

wsgiref.simple_server.**demo_app**(*environ*, *start_response*)

此函数是一个小巧但完整的 WSGI 应用程序，它返回一个包含消息"Hello world!" 以及 *environ* 形参中提供的键/值对的文本页面。它适用于验证 WSGI 服务器 (例如*wsgiref.simple_server*) 是否能够正确地运行一个简单的 WSGI 应用程序。

start_response 可调用对象必须遵循*StartResponse* 协议。

class wsgiref.simple_server.**WSGIServer**(*server_address*, *RequestHandlerClass*)

创建一个*WSGIServer* 实例。*server_address* 应当是一个 (host,port) 元组，而 *RequestHandlerClass* 应当是*http.server.BaseHTTPRequestHandler* 的子类，它将被用来处理请求。

你通常不需要调用此构造器，因为*make_server()* 函数能为你处理所有的细节。

WSGIServer 是*http.server.HTTPServer* 的子类，因此它所有的方法 (例如 serve_forever() 和 handle_request()) 都是可用的。*WSGIServer* 还提供了以下 WSGI 专属的方法:

set_app(*application*)

将可调用对象 *application* 设为将要接受请求的 WSGI 应用程序。

get_app()

返回当前设置的应用程序可调用对象。

但是，你通常不需要使用这些附加的方法，因为*set_app()* 通常会由*make_server()* 来调用，而*get_app()* 主要是针对请求处理器实例的。

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)

为给定的 *request* 创建一个 HTTP 处理器 (例如套接字)，*client_address* (一个 (host,port) 元组)，以及 *server* (*WSGIServer* 实例)。

你不需要直接创建该类的实例；它们会根据*WSGIServer* 对象的需要自动创建。但是，你可以子类化该类并将其作为 *handler_class* 提供给*make_server()* 函数。一些可能在子类中重载的相关方法:

get_environ()

返回对应于一个请求的*WSGIEnvironment* 字典。默认实现会拷贝*WSGIServer* 对象的 base_environ 字典属性的内容然后添加从 HTTP 请求获取的各种标头。对此方法的每次调用都应当返回一个新的包含 **PEP 3333** 所规定的所有相关 CGI 环境变量的字典。

get_stderr()

返回应被用作 wsgi.errors 流的对象。默认实现将只返回 sys.stderr。

handle()

处理 HTTP 请求。默认的实现会使用*wsgiref.handlers* 类创建一个处理器实例来实现实际的 WSGI 应用程序接口。

22.2.4 wsgiref.validate --- WSGI 一致性检查器

当创建新的 WSGI 应用程序对象、框架、服务器或中间件时，使用*wsgiref.validate* 来验证新代码的一致性是很有用的。此模块提供了一个创建 WSGI 应用程序对象的函数来验证 WSGI 服务器或网关与 WSGI 应用程序对象之间的通信，以便检查双方的协议一致性。

请注意这个工具并不保证完全符合 **PEP 3333**；此模块没有报告错误并不一定表示不存在错误。但是，如果此模块确实报告了错误，那么几乎可以肯定服务器或应用程序不是 100% 符合要求的。

此模块是基于 Ian Bicking 的"Python Paste" 库的 paste.lint 模块。

wsgiref.validate.**validator**(*application*)

包装 *application* 并返回一个新的 WSGI 应用程序对象。返回的应用程序将转发所有请求到原始的 *application*，并将检查 *application* 和唤起它的服务器是否都符合 WSGI 规范和 **RFC 2616**。

任何被检测到的不一致性都会导致引发*AssertionError*；但是请注意，如何对待这些错误取决于具体服务器。例如，*wsgiref.simple_server* 和其他基于*wsgiref.handlers* 的服务器（它们没有重载错误处理方法来做其他操作）将简单地输出一条消息报告发生了错误，并将回溯转给 sys.stderr 或者其他错误数据流。

这个包装器也可能会使用*warnings* 模块生成输出来指明存在问题但实际上未被 **PEP 3333** 所禁止的行为。除非它们被 Python 命令行选项或*warnings* API 所屏蔽，否则任何此类警告都将被写入到 sys.stderr (不是 wsgi.errors，除非它们恰好是同一个对象)。

用法示例:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# 我们的可调用对象与标准是故意不兼容的，
# 因此验证器将会出错
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP 状态
    headers = [('Content-type', 'text/plain')]  # HTTP 标头
    start_response(status, headers)

    # 这将会出错因为我们需要返回一个列表，
    # 验证器将会提醒我们
    return b"Hello World"

# 这是包装在验证器中国应用程序
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

22.2.5 wsgiref.handlers -- 服务器/网关基类

此模块提供了用于实现 WSGI 服务器和网关的处理器基类。这些基类可处理大部分与 WSGI 应用程序通信的工作，只要给予它们一个带有输入、输出和错误流的 CGI 类环境。

class wsgiref.handlers.**CGIHandler**

通过 sys.stdin, sys.stdout, sys.stderr 和 os.environ 发起基于 CGI 的调用。这在当你有一个 WSGI 应用程序并想将其作为 CGI 脚本运行时很有用处。只需唤起 CGIHandler().run(app)，其中 app 是你想要唤起的 WSGI 应用程序。

该类是*BaseCGIHandler* 的子类，它将设置 wsgi.run_once 为真值，wsgi.multithread 为假值，wsgi.multiprocess 为真值，并总是使用*sys* 和*os* 来获取所需的 CGI 流和环境。

class wsgiref.handlers.**IISCGIHandler**

CGIHandler 的一个专门化替代，用于在 Microsoft 的 IIS Web 服务器上部署，无需设置 config allowPathInfo 选项 (IIS>=7) 或 metabase allowPathInfoForScriptMappings (IIS<7)。

默认情况下，IIS 给出的 PATH_INFO 会与前面的 SCRIPT_NAME 重复，导致想要实现路由的 WSGI 应用程序出现问题。这个处理器会去除任何这样的重复路径。

IIS 可被配置为传递正确的 PATH_INFO，但这会导致另一个 PATH_TRANSLATED 出错的问题。幸运的是这个变量很少被使用并且不被 WSGI 所保证。但是在 IIS<7 上，这个设置只能在 vhost 层级上进行，影响到所有其他脚本映射，其中许多在受 PATH_TRANSLATED 问题影响时都会中断运行。因此 IIS<7 的部署几乎从不附带这样的修正（即使 IIS7 也很少使用它，因为它仍然不带 UI）。

CGI 代码没有办法确定该选项是否已设置，因此提供了一个单独的处理器类。它的用法与*CGIHandler* 相同，即通过调用 IISCGIHandler().run(app)，其中 app 是你想要唤起的 WSGI 应用程序。

Added in version 3.2.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

类似于*CGIHandler*，但是改用*sys* 和*os* 模块，CGI 环境和 I/O 流会被显式地指定。*multithread* 和 *multiprocess* 值被用来为处理器实例所运行的任何应用程序设置 wsgi.multithread 和 wsgi.multiprocess 旗标。

该类是*SimpleHandler* 的子类，旨在用于 HTTP " 原始服务器" 以外的软件。如果你在编写一个网关协议实现（例如 CGI, FastCGI, SCGI 等等），它使用 Status: 标头来发布 HTTP 状态，你可能会想要子类化该类而不是*SimpleHandler*。

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

类似于*BaseCGIHandler*，但被设计用于 HTTP 原始服务器。如果你在编写一个 HTTP 服务器实现，你可能会想要子类化该类而不是*BaseCGIHandler*。

该类是*BaseHandler* 的子类。它重载了 __init__(), *get_stdin()*, *get_stderr()*, *add_cgi_vars()*, *_write()* 和*_flush()* 方法以支持通过构造器显式地设置环境和流。所提供的环境和流存储在 stdin, stdout, stderr 和 environ 属性中。

stdout 的*write()* 方法应该完整写入每个数据块，与*io.BufferedIOBase* 一样。

class wsgiref.handlers.**BaseHandler**

这是适用于运行 WSGI 应用程序的抽象基类。每个实例将处理一个单独的 HTTP 请求，不过在原则上你也可以创建一个可针对多个请求重用的子类。

BaseHandler 实例只有一个方法提供给外部使用:

run(*app*)

运行指定的 WSGI 应用程序 *app*。

所有的*BaseHandler* 其他方法都是在运行应用程序过程中由该方法唤起的，因此主要是为了允许定制运行过程。

以下方法必须在子类中被重载:

_write(*data*)

缓冲字节数据 *data* 以便传输给客户端。如果此方法真的传输了数据也是可以的；*BaseHandler* 只有在底层系统真有这样的区分时才会区分写入和刷新操作以提高效率。

_flush()

强制将缓冲的数据传输给客户端。如果此方法无任何操作也是可以的（例如，*_write()* 实际上已发送了数据）。

get_stdin()

返回一个兼容*InputStream* 的适合用作当前所处理请求的 wsgi.input 的对象。

get_stderr()

返回一个兼容*ErrorStream* 的适合用作当前所处理请求的 wsgi.errors 的对象。

add_cgi_vars()

将当前请求的 CGI 变量插入到 environ 属性。

以下是一些你可能会想要重载的其他方法。但这只是个简略的列表，它不包括每个可被重载的方法。你应当在在尝试创建自定义的*BaseHandler* 子类之前参阅文档字符串和源代码来了解更多信息。

用于自定义 WSGI 环境的属性和方法:

wsgi_multithread

用于 wsgi.multithread 环境变量的值。它在*BaseHandler* 中默认为真值，但在其他子类中可能有不同的默认值（或是由构造器来设置）。

wsgi_multiprocess

用于 wsgi.multiprocess 环境变量的值。它在*BaseHandler* 中默认为真值，但在其他子类中可能有不同的默认值（或是由构造器来设置）。

wsgi_run_once

用于 wsgi.run_once 环境变量的值。它在*BaseHandler* 中默认为假值，但在*CGIHandler* 中默认为真值。

os_environ

要包括在每个请求的 WSGI 环境中的默认环境变量。在默认情况下，这是当*wsgiref.handlers* 被导入时的 os.environ 的一个副本，但也可以在类或实例层级上创建它们自己的子类。请注意该字典应当被当作是只读的，因为默认值会在多个类和实际之间共享。

server_software

如果设置了*origin_server* 属性，该属性的值会被用来设置默认的 SERVER_SOFTWARE WSGI 环境变量，并且还会被用来设置 HTTP 响应中默认的 Server: 标头。它会被不是 HTTP 原始服务器的处理器所忽略 (例如*BaseCGIHandler* 和*CGIHandler*)。

在 3.3 版本发生变更: 名称”Python” 会被替换为实现专属的名称如”CPython”, ”Jython” 等等。

get_scheme()

返回当前请求所使用的 URL 方案。默认的实现会使用来自*wsgiref.util* 的 guess_scheme() 函数根据当前请求的 environ 变量来猜测方案应该是”http” 还是”https”。

setup_environ()

将 environ 属性设为填充了完整内容的 WSGI 环境。默认的实现会使用上述的所有方法和属性，加上*get_stdin()*, *get_stderr()* 和*add_cgi_vars()* 方法以及*wsgi_file_wrapper* 属性。它还会在 SERVER_SOFTWARE 不存在时插入该键，只要*origin_server* 属性为真值并且设置了*server_software* 属性。

用于自定义异常处理的方法和属性:

log_exception(*exc_info*)

将 *exc_info* 元素记录到服务器日志。*exc_info* 是一个 (type, value, traceback) 元组。默认的实现会简单地将回溯信息写入请求的 wsgi.errors 流并刷新它。子类可以重写此方法来修改格式或重设输出目标，通过邮件将回溯信息发给管理员，或执行任何其他符合要求的动作。

traceback_limit

要包括在默认*log_exception()* 方法的回溯信息输出中的最大帧数。如果为 None，则会包括所有的帧。

error_output(*environ*, *start_response*)

此方法是一个为用户生成错误页面的 WSGI 应用程序。它仅当将标头发送给客户端之前发生错误时会被唤起。

此方法可以使用 `sys.exception()` 来访问当前错误信息，并应当在调用它时将该信息传给 *start_response* (如 **PEP 3333** 的"Error Handling" 一节所描述的)。具体而言，*start_response* 可调用对象应当遵循 *StartResponse* 协议

默认的实现只是使用 *error_status*, *error_headers*, 和 *error_body* 属性来生成一个输出页面。子类可以重写此方法来产生更动态化的错误输出。

但是请注意，从安全角度看来是不建议将诊断信息暴露给任何用户的；理想的做法是你应当通过特别处理来启用诊断输出，因此默认的实现并不包括这样的内容。

error_status

用于错误响应的 HTTP 状态。这应当是一个在 **PEP 3333** 中定义的字符串；它默认为代码 500 以相应的消息。

error_headers

用于错误响应的 HTTP 标头。这应当是由 WSGI 响应标头 (`(name, value)` 元组) 构成的列表，如 **PEP 3333** 所定义的。默认的列表只是将内容类型设为 `text/plain`。

error_body

错误响应体。这应当是一个 HTTP 响应体字节串。它默认为纯文本"A server error occurred. Please contact the administrator."

用于 **PEP 3333** 的" 可选的平台专属文件处理" 特性的方法和属性:

wsgi_file_wrapper

一个 `wsgi.file_wrapper` 工厂对象，兼容 *wsgiref.types.FileWrapper*，或者为 `None`。该属性的默认值是 *wsgiref.util.FileWrapper* 类。

sendfile()

重载以实现平台专属的文件传输。此方法仅在应用程序的返回值是由 *wsgi_file_wrapper* 属性指定的类的实例时会被调用。如果它能够成功地传输文件则应当返回真值，以使得默认的传输代码将不会被执行。此方法的默认实现只会返回假值。

杂项方法和属性:

origin_server

该属性在处理器的 *_write()* 和 *_flush()* 被用于同客户端直接通信而不是通过需要 HTTP 状态为某种特殊 `Status:` 标头的 CGI 类网关协议时应当被设为真值

该属性在 *BaseHandler* 中默认为真值，但在 *BaseCGIHandler* 和 *CGIHandler* 中则为假值。

http_version

如果 *origin_server* 为真值，则该字符串属性会被用来设置给客户端的响应的 HTTP 版本。它的默认值为 `"1.0"`。

wsgiref.handlers.**read_environ**()

将来自 `os.environ` 的 CGI 变量转码为 **PEP 3333** "bytes in unicode" 字符串，返回一个新的字典。此函数被 *CGIHandler* 和 *IISCGIHandler* 用来替代直接使用 `os.environ`，后者不一定在所有使用 Python 3 的平台和 Web 服务器上都符合 WSGI 标准 -- 特别是当 OS 的实际环境为 Unicode 时 (例如 Windows)，或者当环境为字节数据，但被 Python 用来解码它的系统编码格式不是 ISO-8859-1 时 (例如使用 UTF-8 的 Unix 系统)。

如果你要实现自己的基于 CGI 的处理器，你可能会想要使用此例程而不是简单地从 `os.environ` 直接拷贝值。

Added in version 3.2.

22.2.6 wsgiref.types -- 用于静态类型检查的 WSGI 类型

本模块提供了多种类型用于在 **PEP 3333** 中所描述的静态类型检查。

Added in version 3.11.

class wsgiref.types.**StartResponse**

一个描述 **start_response()** 可迭代对象的*typing.Protocol*。(**PEP 3333**)

wsgiref.types.**WSGIEnvironment**

一个描述 WSGI 环境字典的类型别名。

wsgiref.types.**WSGIApplication**

一个描述 WSGI 应用程序可迭代对象的类型别名。

class wsgiref.types.**InputStream**

一个描述 **WSGI 输入流** 的*typing.Protocol*。

class wsgiref.types.**ErrorStream**

一个描述 **WSGI 错误流** 的*typing.Protocol*。

class wsgiref.types.**FileWrapper**

一个描述 **文件包装器** 的*typing.Protocol*。请参阅*wsgiref.util.FileWrapper* 查看此协议的一个具体实现。

22.2.7 例子

下面是一个可运行的"Hello World" WSGI 应用程序，其中 *start_response* 可调用对象应当遵循*StartResponse* 协议:

```
"""
每个 WSGI 应用程序必须有一个应用程序对象 —— 即一个将接受
两个参数的可调用对象。 为达成此目的，我们准备使用一个函数
（请注意并非仅限使用函数，例如你也可以使用一个类）。
传给该函数的第一个参数是一个包含 CGI 风格的环境变量的字典
而第二个变量就是上述的可调用对象。
"""
from wsgiref.simple_server import make_server


def hello_world_app(environ, start_response):
    status = "200 OK"  # HTTP Status
    headers = [("Content-type", "text/plain; charset=utf-8")]  # HTTP 标头
    start_response(status, headers)

    # 返回的对象将被打印出来
    return [b"Hello World"]

with make_server("", 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # 运行服务直到进程被杀掉
    httpd.serve_forever()
```

一个发布当前目录的 WSGI 应用程序示例，接受通过命令行指定可选的目录和端口号 (默认值: 8000):

```
"""
基于 wsgiref 的小型 web 服务器。 接受一个服务路径和
一个可选的端口号 (默认为 8000)，然后尝试发布文件。
MIME 类型将根据文件名来猜测，如果文件未找到则会
引发 404 错误。
"""
import mimetypes
```

(续下页)

(接上页)

```
import os
import sys
from wsgiref import simple_server, util


def app(environ, respond):
    # 获取文件名和 MIME 类型
    fn = os.path.join(path, environ["PATH_INFO"][1:])
    if "." not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, "index.html")
    mime_type = mimetypes.guess_file_type(fn)[0]

    # 如果文件存在则返回 200 OK, 否则返回 404 Not Found
    if os.path.exists(fn):
        respond("200 OK", [("Content-Type", mime_type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond("404 Not Found", [("Content-Type", "text/plain")])
        return [b"not found"]


if __name__ == "__main__":
    # 从命令行参数获取路径和端口
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000

    # 创建并启动服务直至被 control-c 中断
    httpd = simple_server.make_server("", port, app)
    print(f"Serving {path} on port {port}, control-C to stop")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()
```

22.3 `urllib` --- URL 处理模块

源代码: Lib/urllib/

`urllib` 是一个收集了多个涉及 URL 的模块的包：

- *`urllib.request`* 打开和读取 URL
- *`urllib.error`* 包含*`urllib.request`* 抛出的异常
- *`urllib.parse`* 用于解析 URL
- *`urllib.robotparser`* 用于解析 `robots.txt` 文件

22.4 `urllib.request` --- 用于打开 URL 的可扩展库

源码： Lib/urllib/request.py

`urllib.request` 模块定义了适用于在各种复杂情况下打开 URL（主要为 HTTP）的函数和类 --- 例如基本认证、摘要认证、重定向、cookies 及其它。

参见

对于更高级别的 HTTP 客户端接口，建议使用 Requests 。

警告

在 macOS 将此模块用于包含 *os.fork()* 的程序是不安全的，因为 macOS 的 *getproxies()* 实现使用了高层级的系统 API。可将环境变量 `no_proxy` 设为 `*` 以避免此问题 (即 `os.environ["no_proxy"] = "*"`)。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅 *WebAssembly 平台* 了解详情。

urllib.request 模块定义了以下函数：

urllib.request.**urlopen**(*url*, *data=None*, [*timeout*,]*, *context=None*)

打开 *url*，它可以是一个包含有效的、被正确编码的 URL 的字符串，或是一个 *Request* 对象。

data 必须是一个对象，用于给出要发送到服务器的附加数据，若不需要发送数据则为 `None`。详情请参阅 *Request* 。

urllib.request 模块采用 HTTP/1.1 协议，并且在其 HTTP 请求中包含 `Connection:close` 头部信息。

timeout 为可选参数，用于指定阻塞操作（如连接尝试）的超时时间，单位为秒。如未指定，将使用全局默认超时参数）。本参数实际仅对 HTTP、HTTPS 和 FTP 连接有效。

如果给定了 *context* 参数，则必须是一个 *ssl.SSLContext* 实例，用于描述各种 SSL 参数。更多详情请参阅 *HTTPSConnection* 。

本函数总会返回一个对象，该对象可作为 *context manager* 使用，带有 *url*、*headers* 和 *status* 属性。有关这些属性的更多详细信息，请参阅 *urllib.response.addinfourl* 。

对于 HTTP 和 HTTPS 的 URL 而言，本函数将返回一个稍经修改的 *http.client.HTTPResponse* 对象。除了上述 3 个新的方法之外，还有 msg 属性包含了与 *reason* 属性相同的信息---服务器返回的原因描述文字，而不是 *HTTPResponse* 的文档所述的响应头部信息。

对于 FTP、文件、数据的 URL，以及由传统的 *URLopener* 和 *FancyURLopener* 类处理的请求，本函数将返回一个 *urllib.response.addinfourl* 对象。

协议发生错误时，将会引发 *URLError* 。

请注意，如果没有处理函数对请求进行处理，则有可能会返回 `None` 。尽管默认安装的全局 *OpenerDirector* 会用 *UnknownHandler* 来确保不会发生这种情况。

另外，如果检测到设置了代理（例如，当设置了 `http_proxy` 之类的 `*_proxy` 环境变量时），将默认安装 *ProxyHandler* 并确保通过代理来处理请求。

Python 2.6 以下版本中留存的 `urllib.urlopen` 函数已停止使用了；*urllib.request.urlopen()* 对应于传统的 `urllib2.urlopen` 。对代理服务的处理是通过将字典参数传给 `urllib.urlopen` 来完成的，可以用 *ProxyHandler* 对象获取到代理处理函数。

默认会为 `urllib.Request` 引发一条审计事件，其参数 `fullurl`、`data`、`headers`、`method` 均取自请求对象。

在 3.2 版本发生变更: 增加了 *cafile* 与 *capath*。

现在将在可能的情况下支持 HTTPS 虚拟主机（也就是说，如果 *ssl.HAS_SNI* 为真值）。

data 可以是一个可迭代对象。

在 3.3 版本发生变更: 增加了 *cadefault*。

在 3.4.3 版本发生变更: 增加了 *context*。

在 3.10 版本发生变更: 当未给出 *context* 时 HTTPS 连接现在会发送一个带有 `http/1.1` 协议指示符的 ALPN 扩展。自定义 *context* 应当使用 *set_alpn_protocols()* 来设置 ALPN 协议。

在 3.13 版本发生变更: 移除了 *cafile*, *capath* 和 *cadefault* 形参：请改用 *context* 形参。

`urllib.request.`**`install_opener`**(*opener*)

安装一个 *OpenerDirector* 实例，作为默认的全局打开函数。仅当 urlopen 用到该打开函数时才需要安装；否则，只需调用 *OpenerDirector.open()* 而不是 *urlopen()*。代码不会检查是否真的属于 *OpenerDirector* 类，所有具备适当接口的类都能适用。

`urllib.request.`**`build_opener`**([*handler*, ...])

返回一个 *OpenerDirector* 实例，以给定顺序把处理函数串联起来。处理函数可以是 *BaseHandler* 的实例，也可以是 *BaseHandler* 的子类（这时构造函数必须允许不带任何参数的调用）。以下类的实例将位于 处理函数之前，除非 处理函数已包含这些类、其实例或其子类: *ProxyHandler*（如果检测到代理设置） 、*UnknownHandler* 、*HTTPHandler* 、*HTTPDefaultErrorHandler* 、*HTTPRedirectHandler* 、*FTPHandler* 、*FileHandler* 、*HTTPErrorProcessor* 。

若 Python 安装时已带了 SSL 支持（指可以导入 *ssl* 模块），则还会加入 *HTTPSHandler* 。

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

`urllib.request.`**`pathname2url`**(*path*)

将给定的本地路径转换为一个 `file:` URL。此函数会使用 *quote()* 函数对路径进行编码。出于历史原因，返回值将省略 `file:` 类别前缀。下面的例子演示该函数在 Windows 上的使用:

```
>>> from urllib.request import pathname2url
>>> path = 'C:\\Program Files'
>>> 'file:' + pathname2url(path)
'file:///C:/Program%20Files'
```

`urllib.request.`**`url2pathname`**(*url*)

将给定的 `file:` URL 转换为一个本地路径。此函数会使用 *unquote()* 对 URL 进行解码。出于历史原因，给定的值 必须省略 `file:` 类别前缀。下面的例子演示该函数在 Windows 上的使用:

```
>>> from urllib.request import url2pathname
>>> url = 'file:///C:/Program%20Files'
>>> url2pathname(url.removeprefix('file:'))
'C:\\Program Files'
```

`urllib.request.`**`getproxies`**()

此辅助函数将返回一个将各个方案映射到代理服务器 URL 的字典。它会先为所有操作系统以大小写不敏感的方式扫描名为 `<scheme>_proxy` 的环境变量，当无法找到时，则会在 macOS 上从系统配置中而在 Windows 上从 Windows 系统注册表中查找代理信息。如果同时存在小写和大写形式的环境变量（且内容不一致），则会首先小写形式。

> **备注**
>
> 如果存在环境变量 `REQUEST_METHOD` ，通常表示脚本运行于 CGI 环境中，则环境变量 `HTTP_PROXY`（大写的 `_PROXY`）将会被忽略。这是因其可以由客户端用 HTTP 头部信息 “Proxy:” 注入。若要在 CGI 环境中使用 HTTP 代理，请显式使用 `ProxyHandler ，或确保变量名称为小写（或至少是 `_proxy` 后缀）。

提供了以下类:

class `urllib.request.`**`Request`**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)

URL 请求对象的抽象类。

url 应为一个包含有效的、被正确编码的 URL 的字符串。

data 必须是一个对象，用于给定发往服务器的附加数据，若无需此类数据则为 `None` 。目前唯一用到 *data* 的只有 HTTP 请求。支持的对象类型包括字节串、类文件对象和可遍历的类字节串对象。如果没有提供 `Content-Length` 和 `Transfer-Encoding` 头部字段，*HTTPHandler* 会根据 *data* 的类型设置这些头部字段。`Content-Length` 将用于发送字节对象，而 **RFC 7230** 第 3.3.1 节中定义的 `Transfer-Encoding: chunked` 将用于发送文件和其他可遍历对象。

对于 HTTP POST 请求方法而言，*data* 应该是标准 *application/x-www-form-urlencoded* 格式的缓冲区。*urllib.parse.urlencode()* 函数的参数为映射对象或二元组序列，并返回一个该编码格式的 ASCII 字符串。在用作 *data* 参数之前，应将其编码为字节串。

headers 应当是一个字典，并将被视同附带了每个键和值作为参数去调用*add_header()*。这通常被用于” 伪装” `User-Agent` 标头值，浏览器会使用标头值来标识自己 -- 某些 HTTP 服务器只允许来自普通浏览器的请求而不允许来自脚本的请求。例如，Mozilla Firefox 可能将自己标识为 `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`，而*urllib*的默认用户代理字符串则是 `"Python-urllib/2.6"` (在 Python 2.6 中)。所有发送的标头键都使用驼峰命名法。

如果给出了 *data* 参数则应当包括一个合适的 `Content-Type` 标头。如果未提供此标头并且 *data* 不为 `None`，则会添加 `Content-Type: application/x-www-form-urlencoded` 作为默认值。

接下来的两个参数，只对第三方 HTTP cookie 的处理才有用：

origin_req_host 应为发起初始会话的请求主机，定义参见 **RFC 2965** 。默认指为 `http.cookiejar.request_host(self)` 。这是用户发起初始请求的主机名或 IP 地址。假设请求是针对 HTML 文档中的图片数据发起的，则本属性应为对包含图像的页面发起请求的主机。

unverifiable 应该标示出请求是否无法验证，定义参见 **RFC 2965** 。默认值为 `False` 。所谓无法验证的请求，是指用户没有机会对请求的 URL 做验证。例如，如果请求是针对 HTML 文档中的图像，用户没有机会去许可能自动读取图像，则本参数应为 True。

method 应为一个指明要使用的 HTTP 请求方法的字符串 (例如 `'HEAD'`)。如果提供，其值将存储在*method*属性中并由*get_method()*使用。如果 *data* 为 `None` 则默认为 `'GET'`，否则为 `'POST'`。子类可以通过在类自身设置*method*属性来指明不同的默认方法。

> **备注**
>
> 如果 data 对象无法分多次传递其内容（比如文件或只能生成一次内容的可迭代对象）并且由于 HTTP 重定向或身份验证而发生请求重试行为，则该请求不会正常工作。*data* 是紧挨着头部信息发送给 HTTP 服务器的。现有库不支持 HTTP 100-continue 的征询。

在 3.3 版本发生变更: Request 类增加了*Request.method*参数。

在 3.4 版本发生变更: 默认*Request.method*可以在类中标明。

在 3.6 版本发生变更: 如果给出了 `Content-Length` ，且 *data* 既不为 `None` 也不是字节串对象，则不会触发错误。而会退而求其次采用分块传输的编码格式。

class urllib.request.**OpenerDirector**

OpenerDirector 类通过串接在一起的*BaseHandler*打开 URL，并负责管理 handler 链及从错误中恢复。

class urllib.request.**BaseHandler**

这是所有已注册 handler 的基类，只做了简单的注册机制。

class urllib.request.**HTTPDefaultErrorHandler**

为 HTTP 错误响应定义的默认 handler，所有出错响应都会转为*HTTPError*异常。

class urllib.request.**HTTPRedirectHandler**

一个用于处理重定向的类。

class urllib.request.**HTTPCookieProcessor**(*cookiejar=None*)

一个用于处理 HTTP Cookies 的类。

class urllib.request.**ProxyHandler**(*proxies=None*)

让请求转往代理服务。如果给出了 *proxies*，则它必须是一个将协议名称映射到代理 URL 的字典。默认是从环境变量 <protocol>_proxy 中读取代理列表。如果没有设置代理服务的环境变量，则在 Windows 环境下代理设置会从注册表的 Internet Settings 部分获取，而在 macOS 环境下代理信息会从 System Configuration Framework 获取。

若要禁用自动检测出来的代理，请传入空的字典对象。

环境变量 no_proxy 可用于指定不必通过代理访问的主机；应为逗号分隔的主机名后缀列表，可加上 :port ，例如 cern.ch,ncsa.uiuc.edu,some.host:8080 。

> **备注**
>
> 如果设置了 REQUEST_METHOD 变量，则会忽略 HTTP_PROXY ；参阅*getproxies()* 文档。

class urllib.request.**HTTPPasswordMgr**

维护 (realm, uri) -> (user, password) 映射数据库。

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**

维护 (realm, uri) -> (user, password) 映射数据库。realm 为 None 视作全匹配，若没有其他合适的安全区域就会检索它。

class urllib.request.**HTTPPasswordMgrWithPriorAuth**

HTTPPasswordMgrWithDefaultRealm 的一个变体，也带有 uri -> is_authenticated 映射数据库。可被 BasicAuth 处理函数用于确定立即发送身份认证凭据的时机，而不是先等待 401 响应。

Added in version 3.5.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)

这是一个帮助完成 HTTP 身份认证的混合类，对远程主机和代理都适用。参数 *password_mgr* 应与*HTTPPasswordMgr* 兼容；关于必须支持哪些接口，请参阅*HTTPPasswordMgr 对象* 对象的章节。如果 *password_mgr* 还提供 is_authenticated 和 update_authenticated 方法（请参阅*HTTPPasswordMgrWithPriorAuth 对象* 对象），则 handler 将对给定 URI 用到 is_authenticated 的结果，来确定是否随请求发送身份认证凭据。如果该 URI 的 is_authenticated 返回 True，则发送凭据。如果 is_authenticated 为 False ，则不发送凭据，然后若收到 401 响应，则使用身份认证凭据重新发送请求。如果身份认证成功，则调用 update_authenticated 设置该 URI 的 is_authenticated 为 True，这样后续对该 URI 或其所有父 URI 的请求将自动包含该身份认证凭据。

Added in version 3.5: 增加了对 is_authenticated 的支持。

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)

处理远程主机的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。如果给出错误的身份认证方式，HTTPBasicAuthHandler 将会触发*ValueError* 。

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)

处理有代理服务时的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)

这是一个帮助完成 HTTP 身份认证的混合类，对远程主机和代理都适用。参数 *password_mgr* 应与*HTTPPasswordMgr* 兼容；关于必须支持哪些接口，请参阅*HTTPPasswordMgr 对象* 的章节。

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)

处理远程主机的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。如果同时添加了 digest 身份认证 handler 和 basic 身份认证 handler，则会首先尝试 digest 身份认证。如果 digest 身份认证再返回 40x 响应，会再发送到 basic 身份验证 handler 进行处理。如果给出 Digest 和 Basic 之外的身份认证方式，本 handler 方法将会触发*ValueError* 。

在 3.3 版本发生变更: 碰到不支持的认证方式时，将会触发*ValueError* 。

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)

处理有代理服务时的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。

class urllib.request.**HTTPHandler**

用于打开 HTTP URL 的 handler 类。

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)

用于打开 HTTPS URL 的 handler 类。*context* 和 *check_hostname* 的含义与*http.client.HTTPSConnection* 的一样。

在 3.2 版本发生变更: 添加 *context* 和 *check_hostname* 参数。

class urllib.request.**FileHandler**

打开本地文件。

class urllib.request.**DataHandler**

打开数据 URL。

Added in version 3.4.

class urllib.request.**FTPHandler**

打开 FTP URL。

class urllib.request.**CacheFTPHandler**

打开 FTP URL，并将打开的 FTP 连接存入缓存，以便最大程度减少延迟。

class urllib.request.**UnknownHandler**

处理所有未知类型 URL 的兜底类。

class urllib.request.**HTTPErrorProcessor**

处理出错的 HTTP 响应。

22.4.1 Request 对象

以下方法介绍了*Request* 的公开接口，因此子类可以覆盖所有这些方法。这里还定义了几个公开属性，客户端可以利用这些属性了解经过解析的请求。

Request.**full_url**

传给构造函数的原始 URL。

在 3.4 版本发生变更.

Request.full_url 是一个带有 setter、getter 和 deleter 的属性。读取*full_url* 属性将会返回附带片段（fragment）的初始请求 URL。

Request.**type**

URI 方式。

Request.**host**

URI 权限，通常是整个主机，但也有可能带有冒号分隔的端口号。

Request.**origin_req_host**

请求的原始主机，不含端口。

Request.**selector**

URI 路径。若*Request* 使用代理，selector 将会是传给代理的完整 URL。

Request.**data**

请求的数据体，未给出则为 None 。

在 3.4 版本发生变更: 现在如果修改*Request.data* 的值，则会删除之前设置或计算过的 “Content-Length” 头部信息。

Request.**unverifiable**

布尔值，标识本请求是否属于 **RFC 2965** 中定义的无法验证的情况。

Request.**method**

要采用的 HTTP 请求方法。默认为*None*，表示*get_method()* 将对方法进行正常处理。设置本值可以覆盖*get_method()* 中的默认处理过程，设置方式可以是在*Request* 的子类中给出默认值，也可以通过 *method* 参数给*Request* 构造函数传入一个值。

Added in version 3.3.

在 3.4 版本发生变更: 现在可以在子类中设置默认值；而之前只能通过构造函数的实参进行设置。

Request.**get_method**()

返回表示 HTTP 请求方法的字符串。如果*Request.method* 不为 None ，则返回其值。否则若*Request.data* 为则返回 'GET'，不为 None 则返回 'POST' 。只对 HTTP 请求有效。

在 3.3 版本发生变更: 现在 get_method 会兼顾*Request.method* 的值。

Request.**add_header**(*key*, *val*)

向请求添加一个标头。标头目前会被所有处理器忽略但只有 HTTP 处理器是例外，该处理器会将它们加入发给服务器的标头列表中。请注意同名的标头只能有一个，当 *key* 发生冲突时后续的调用将会覆盖之前的调用。目前，这并不会造成 HTTP 功能的损失，因为所有可多次使用而仍有意义的标头都有（特定标头专属的）方式来获得与仅使用一个标头时相同的功能。请注意使用此方法添加的标头也会被添加到重定向的请求中。

Request.**add_unredirected_header**(*key*, *header*)

添加一项不会被加入重定向请求的头部信息。

Request.**has_header**(*header*)

返回本实例是否带有命名头部信息（对常规数据和非重定向数据都会检测）。

Request.**remove_header**(*header*)

从本请求实例中移除指定命名的头部信息（对常规数据和非重定向数据都会检测）。

Added in version 3.4.

Request.**get_full_url**()

返回构造器中给定的 URL。

在 3.4 版本发生变更.

返回*Request.full_url*

Request.**set_proxy**(*host*, *type*)

连接代理服务器，为当前请求做准备。*host* 和 *type* 将会取代本实例中的对应值，selector 将会是构造函数中给出的初始 URL。

Request.**get_header**(*header_name*, *default=None*)

返回给定头部信息的数据。如果该头部信息不存在，返回默认值。

Request.**header_items**()

返回头部信息，形式为（名称, 数据）的元组列表。

在 3.4 版本发生变更: 自 3.3 起已弃用的下列方法已被删除：add_data、has_data、get_data、get_type、get_host、get_selector、get_origin_req_host 和 is_unverifiable 。

22.4.2 OpenerDirector 对象

OpenerDirector 实例有以下方法：

OpenerDirector.**add_handler**(*handler*)

handler 应为*BaseHandler* 的实例。将检索以下类型的方法，并将其添加到对应的处理链中（注意 HTTP 错误是特殊情况）。请注意，下文中的 *protocol* 应替换为要处理的实际协议，例如 http_response() 将是 HTTP 协议响应处理函数。并且 *type* 也应替换为实际的 HTTP 代码，例如 http_error_404() 将处理 HTTP 404 错误。

- `<protocol>_open()` --- 表明该处理器知道如何打开 *protocol* URL。

  更多信息请参阅*BaseHandler.<protocol>_open()* 。

- `http_error_<type>()` --- 表明该处理器知道如何处理 HTTP 错误代码 *type* 对应的 HTTP 错误。

  更多信息请参阅*BaseHandler.http_error_<nnn>()* 。

- `<protocol>_error()` --- 表明该处理器知道如何处理来自 (非 `http`) *protocol* 的错误。

- `<protocol>_request()` --- 表明该处理器知道如何预处理 *protocol* 请求。

  更多信息请参阅*BaseHandler.<protocol>_request()* 。

- `<protocol>_response()` --- 表明该处理器知道如何后继处理 *protocol* 响应。

  更多信息请参阅*BaseHandler.<protocol>_response()* 。

`OpenerDirector.`**`open`**(*url*, *data=None*[, *timeout*])

打开给定的 *url* (可以是一个请求对象或一个字符串)，可以选择传入给定的 *data*。参数、返回值和被引发的异常均与*urlopen()* 的相同 (它只是简单地在当前安装的全局*OpenerDirector* 上调用*open()* 方法)。可选的 *timeout* 形参指定了针对阻塞操作例如连接尝试的超时值 (如果未指明，则将使用全局默认的超时设置)。超时特性仅适用于 HTTP, HTTPS 和 FTP 连接。

`OpenerDirector.`**`error`**(*proto*, **args*)

处理一个给定协议的错误。这将调用针对给定协议的已注册错误处理器并附带给定的参数 (这是协议专属的)。HTTP 协议是一种特殊情况，它使用 HTTP 响应码来确定具体的错误处理器；请参阅错误处理器类的 `http_error_<type>()` 方法。

返回值和异常均与*urlopen()* 相同。

OpenerDirector 对象分 3 个阶段打开 URL：

每个阶段中调用这些方法的次序取决于 handler 实例的顺序。

1. 每个具有名称为 `<protocol>_request()` 的方法的错误处理器都会调用该方法来对请求进行预处理。
2. 具有名称为 `<protocol>_open()` 的方法的错误处理器将被调用以处理请求。这一阶段将在错误处理器返回非*None* 值 (即一个响应) 或者引发异常 (通常为*URLError*) 时结束。异常将被允许传播。

   实际上，以上算法会先尝试名为*default_open()* 的方法。如果这些方法全都返回*None*，则会对名为 `<protocol>_open()` 的方法重复此算法。如果这些方法也全都返回*None*，则会继承对名为*unknown_open()* 的方法重复此算法。

   请注意，这些方法的代码可能会调用*OpenerDirector* 父实例的*open()* 和*error()* 方法。
3. 每个具有名称为 `<protocol>_response()` 的方法的错误处理器都会调用该方法来对响应进行后续处理。

22.4.3 BaseHandler 对象

BaseHandler 对象提供了一些直接可用的方法，以及其他一些可供派生类使用的方法。以下是可供直接使用的方法：

`BaseHandler.`**`add_parent`**(*director*)

将 director 加为父 OpenerDirector。

`BaseHandler.`**`close`**()

移除所有父 OpenerDirector。

以下属性和方法仅供*BaseHandler* 的子类使用：

备注

以下约定已被采纳：定义 `<protocol>_request()` 或 `<protocol>_response()` 方法的子类应当命名为 `*Processor`；所有其他子类应当命名为 `*Handler`。

`BaseHandler.`**`parent`**

一个可用的*OpenerDirector*，可用于以其他协议打开 URI，或处理错误。

`BaseHandler.`**`default_open`**(*req*)

本方法在*BaseHandler* 中 未予定义，但其子类若要捕获所有 URL 则应进行定义。

如果实现了本方法，则它将被上级*OpenerDirector* 所调用。它应当返回一个如*OpenerDirector* 的*open()* 方法的返回值所描述的文件型对象，或是返回 `None`。它应当引发*URLError*，除非发生真正的异常 (例如，*MemoryError* 就不应被映射为*URLError*)。

本方法将会在所有协议的 open 方法之前被调用。

`BaseHandler.<protocol>_open(req)`

本方法在*BaseHandler* 中 未予定义，但其子类若要处理给定协议的 URL 则应进行定义。

此方法如果被定义，它将被上级*OpenerDirector* 调用。返回值应当与*default_open()* 的相同。

`BaseHandler.`**`unknown_open`**(*req*)

本方法在*BaseHandler* 中 未予定义，但其子类若要捕获并打开所有未注册 handler 的 URL，则应进行定义。

若实现了本方法，将会被*parent* 属性指向的父*OpenerDirector* 调用。返回值和*default_open()* 的一样。

`BaseHandler.`**`http_error_default`**(*req*, *fp*, *code*, *msg*, *hdrs*)

本方法在*BaseHandler* 中 未予定义，但其子类若要为所有未定义 handler 的 HTTP 错误提供一个兜底方法，则应进行重写。*OpenerDirector* 会自动调用本方法，获取错误信息，而通常在其他时候不应去调用。

req 会是一个*Request* 对象，*fp* 是一个带有 HTTP 错误体的文件型对象，*code* 是三位数的错误码，*msg* 是供用户阅读的解释信息，*hdrs* 则是一个包含出错头部信息的字典对象。

返回值和触发的异常应与*urlopen()* 的相同。

`BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)`

nnn 应为三位数的 HTTP 错误码。本方法在*BaseHandler* 中也未予定义，但当子类的实例发生代码为 *nnn* 的 HTTP 错误时，若方法存在则会被调用。

子类应该重写本方法，以便能处理相应的 HTTP 错误。

参数、返回值和被引发的异常应当与*http_error_default()* 的相同。

`BaseHandler.<protocol>_request(req)`

本方法在*BaseHandler* 中 未予定义，但其子类若要对给定协议的请求进行预处理，则应进行定义。

若实现了本方法，将会被父*OpenerDirector* 调用。*req* 将为*Request* 对象。返回值应为*Request* 对象。

`BaseHandler.<protocol>_response(req, response)`

本方法在*BaseHandler* 中 未予定义，但其子类若要对给定协议的请求进行后处理，则应进行定义。

若实现了本方法，将会被父*OpenerDirector* 调用。*req* 将为*Request* 对象。*response* 应实现与*urlopen()* 返回值相同的接口。返回值应实现与*urlopen()* 返回值相同的接口。

22.4.4 HTTPRedirectHandler 对象

> **备注**
>
> 某些 HTTP 重定向操作需要本模块的客户端代码提供的功能。这时会触发*HTTPError*。有关各种重定向代码的确切含义，请参阅 **RFC 2616** 。
>
> 如果发给 HTTPRedirectHandler 的重定向 URL 不是 HTTP, HTTPS 或 FTP URL 则出于安全考虑将会引发*HTTPError* 异常。

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

返回一个*Request* 或 None 作为对重定义的响应。此方法将在服务器接收到重定向请求时由 http_error_30*() 方法的默认实现执行调用。如果确实应当发生重定向，则返回一个新的*Request* 以允许 http_error_30*() 重定向到 *newurl*。在其他情况下，如果没有其他处理器来处理此 URL 则会引发*HTTPError*，或者如果此方法不能处理但或许还有其他处理器会处理则返回 None。

> **备注**
>
> 本方法的默认实现代码并未严格遵循 **RFC 2616**，即 POST 请求的 301 和 302 响应不得在未经用户确认的情况下自动进行重定向。现实情况下，浏览器确实允许自动重定向这些响应，将 POST 更改为 GET ，于是默认实现代码就复现了这种处理方式。

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)

重定向到 Location: 或 URI: URL。当得到 HTTP 'moved permanently' 响应时，本方法会被父级*OpenerDirector* 调用。

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)

与*http_error_301()* 相同，不过是发生 "found" 响应时的调用。

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)

与*http_error_301()* 相同，不过是发生 "see other" 响应时的调用。

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)

与*http_error_301()* 一样，但是针对' 临时重定向' 响应进行调用。它不允许将请求方法从 POST 改为 GET。

HTTPRedirectHandler.**http_error_308**(*req*, *fp*, *code*, *msg*, *hdrs*)

与*http_error_301()* 一样，但是针对' 永久重定向' 响应进行调用。它不允许将请求方法从 POST 改为 GET。

Added in version 3.11.

22.4.5 HTTPCookieProcessor 对象

HTTPCookieProcessor 的实例具备一个属性：

HTTPCookieProcessor.**cookiejar**

cookie 存放在*http.cookiejar.CookieJar* 中。

22.4.6 ProxyHandler 对象

ProxyHandler.<protocol>_open(request)

ProxyHandler 将有对应每种 *protocol* 的 <protocol>_open() 方法，在构造函数给出的 *proxies* 字典中包含相应的代理。通过调用 request.set_proxy()，本方法将把请求修改为通过代理，并调用链中的下一个处理器来实际执行协议。

22.4.7 HTTPPasswordMgr 对象

以下方法*HTTPPasswordMgr* 和*HTTPPasswordMgrWithDefaultRealm* 对象均有提供。

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)

uri 可以是单个 URI，也可以是 URI 列表。*realm*、*user* 和 *passwd* 必须是字符串。这使得在为 *realm* 和超级 URI 进行身份认证时，`(user, passwd)` 可用作认证令牌。

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)

为给定 realm 和 URI 获取用户名和密码。如果没有匹配的用户名和密码，本方法将会返回 `(None, None)`。

对于*HTTPPasswordMgrWithDefaultRealm* 对象，如果给定 *realm* 没有匹配的用户名和密码，将搜索 realm `None`。

22.4.8 HTTPPasswordMgrWithPriorAuth 对象

这是*HTTPPasswordMgrWithDefaultRealm* 的扩展，以便对那些需要一直发送认证凭证的 URI 进行跟踪。

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)

realm、*uri*、*user*、*passwd* 的含义与*HTTPPasswordMgr.add_password()* 的相同。*is_authenticated* 为给定 URI 或 URI 列表设置 `is_authenticated` 标志的初始值。如果 *is_authenticated* 设为 `True`，则会忽略 *realm*。

HTTPPasswordMgrWithPriorAuth.**find_user_password**(*realm*, *authuri*)

与*HTTPPasswordMgrWithDefaultRealm* 对象的相同。

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)

更新给定 *uri* 或 URI 列表的 `is_authenticated` 标志。

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)

返回给定 URI `is_authenticated` 标志的当前状态。

22.4.9 AbstractBasicAuthHandler 对象

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

通过获取用户名和密码并重新尝试请求，以处理身份认证请求。*authreq* 应该是请求中包含 realm 的头部信息名称，*host* 指定了需要进行身份认证的 URL 和路径，*req* 应为 (已失败的) *Request* 对象，*headers* 应该是出错的头部信息。

host 要么是一个认证信息（例如 `"python.org"`），要么是一个包含认证信息的 URL（如 `"http://python.org/"`）。不论是哪种格式，认证信息中都不能包含用户信息（因此，`"python.org"` 和 `"python.org:80"` 没问题，而 `"joe:password@python.org"` 则不行）。

22.4.10 HTTPBasicAuthHandler 对象

HTTPBasicAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)

如果可用的话，请用身份认证信息重试请求。

22.4.11 ProxyBasicAuthHandler 对象

ProxyBasicAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)

如果可用的话，请用身份认证信息重试请求。

22.4.12 AbstractDigestAuthHandler 对象

AbstractDigestAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

authreq 应为请求中有关 realm 的头部信息名称，*host* 应为需要进行身份认证的主机，*req* 应为（已失败的）*Request* 对象，*headers* 则应为出错的头部信息。

22.4.13 HTTPDigestAuthHandler 对象

HTTPDigestAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)

如果可用的话，请用身份认证信息重试请求。

22.4.14 ProxyDigestAuthHandler 对象

ProxyDigestAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)

如果可用的话，请用身份认证信息重试请求。

22.4.15 HTTPHandler 对象

HTTPHandler.**http_open**(*req*)

发送 HTTP 请求，根据 `req.has_data()` 的结果，可能是 GET 或 POST 格式。

22.4.16 HTTPSHandler 对象

HTTPSHandler.**https_open**(*req*)

发送 HTTPS 请求，根据 `req.has_data()` 的结果，可能是 GET 或 POST 格式。

22.4.17 FileHandler 对象

FileHandler.**file_open**(*req*)

若无主机名或主机名为 `'localhost'`，则打开本地文件。

在 3.2 版本发生变更: 本方法仅适用于本地主机名。当给出一个远程主机名时，将会引发*URLError*。

22.4.18 DataHandler 对象

DataHandler.**data_open**(*req*)

读取一个数据 URL。这种 URL 在 URL 本身就包含了已编码内容。数据 URL 语法是在 **RFC 2397** 中规定的。当前的实现会忽略 base64 编码的数据 URL 中的空格以便 URL 可以被包装在任何其所在的源文件中。但是即使某些浏览器不会在意 base64 编码的数据 URL 末尾缺失的填充字符，当前的实现仍会在此情况下引发*ValueError*。

22.4.19 FTPHandler 对象

FTPHandler.**ftp_open**(*req*)

打开由 *req* 给出的 FTP 文件。登录时的用户名和密码总是为空。

22.4.20 CacheFTPHandler 对象

CacheFTPHandler 对象即为加入以下方法的*FTPHandler* 对象：

CacheFTPHandler.**setTimeout**(*t*)

设置连接超时为 *t* 秒。

CacheFTPHandler.**setMaxConns**(*m*)

设置已缓存的最大连接数为 *m* 。

22.4.21 UnknownHandler 对象

UnknownHandler.**unknown_open**()

触发*URLError* 异常。

22.4.22 HTTPErrorProcessor 对象

`HTTPErrorProcessor.`**`http_response`**(*request*, *response*)

处理出错的 HTTP 响应。

对于 200 错误码，响应对象应立即返回。

对于除 200 以外的错误代码，会仅通过*OpenerDirector.error()* 将任务传给 `http_error_<type>()` 处理器方法。最终，如果没有其他处理器来处理该错误则*HTTPDefaultErrorHandler* 将引发*HTTPError*。

`HTTPErrorProcessor.`**`https_response`**(*request*, *response*)

HTTPS 出错响应的处理。

与*http_response()* 方法相同。

22.4.23 例子

urllib-howto 中给出了更多的示例。

以下示例将抓取 python.org 主页并显示前 300 个字节的内容:

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!doctype html>\n<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <!
↪[endif]-->\n<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]--
↪>\n<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">
```

请注意，urlopen 将返回字节对象。这是因为 urlopen 无法自动确定由 HTTP 服务器收到的字节流的编码。通常，只要能确定或猜出编码格式，就应将返回的字节对象解码为字符串。

以下 HTML 规范文档 https://html.spec.whatwg.org/#charset 列出了 HTML 或 XML 文档可用来指明其编码格式信息的多种方式。

要了解更多信息，请参阅 W3C 文档: https://www.w3.org/International/questions/qa-html-encoding-declarations.

由于 python.org 网站如 meta 标记所指明的那样是使用 *utf-8* 编码格式，我们将用它来解码字节串对象:

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!-
```

不使用*context manager* 方式也可以获得同样的结果:

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> try:
...     print(f.read(100).decode('utf-8'))
... finally:
...     f.close()
...
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!--
```

以下示例将会把数据流发送给某 CGI 的 stdin，并读取返回数据。请注意，该示例只能工作于 Python 装有 SSL 支持的环境。

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

上述示例中的 CGI 代码如下所示：

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

下面是利用*Request* 发送 PUT 请求的示例：

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

基本 HTTP 认证示例：

```
import urllib.request
# 创建一个带有 Basic HTTP Authentication 支持的 OpenerDirector...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...并全局安装以便其能配合 urlopen 使用。
urllib.request.install_opener(opener)
with urllib.request.urlopen('http://www.example.com/login.html') as f:
    print(f.read().decode('utf-8'))
```

build_opener() 默认提供了许多处理器，包括*ProxyHandler*。在默认情况下，*ProxyHandler* 会使用名为 <scheme>_proxy 的环境变量，其中 <scheme> 是对应的 URL 方案。例如，可以读取 http_proxy 环境变量可获得 HTTP 代理的 URL.

这个示例将默认的*ProxyHandler* 替换为使用以编程方式提供的代理 URL，并通过*ProxyBasicAuthHandler* 添加代理认证支持。

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# 这次，我们不安装 OpenerDirector，而是直接使用它：
with opener.open('http://www.example.com/login.html') as f:
   print(f.read().decode('utf-8'))
```

添加 HTTP 头部信息：

可利用*Request* 构造函数的 *headers* 参数，或者是：

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
```

(续下页)

(接上页)

```
req.add_header('Referer', 'http://www.python.org/')
# 自定义默认的 User-Agent 标头值:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
with urllib.request.urlopen(req) as f:
    print(f.read().decode('utf-8'))
```

OpenerDirector 自动会在每个*Request* 中加入一项 *User-Agent* 头部信息。若要修改，请参见以下语句：

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
with opener.open('http://www.example.com/') as f:
   print(f.read().decode('utf-8'))
```

另请记得，当*Request* 传给*urlopen()* （或*OpenerDirector.open()*）时，会加入一些标准的头部信息（*Content-Length* 、*Content-Type* 和 *Host*）。

以下会话示例用 GET 方法读取包含参数的 URL。

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例换用 POST 方法。请注意 urlencode 输出结果先被编码为字节串 data，再送入 urlopen。

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例显式指定了 HTTP 代理，以覆盖环境变量中的设置：

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

以下示例根本不用代理，也覆盖了环境变量中的设置：

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

22.4.24 已停用的接口

以下函数和类是由 Python 2 模块 urllib（相对早于 urllib2 ）移植过来的。将来某个时候可能会停用。

urllib.request.**urlretrieve**(*url*, *filename=None*, *reporthook=None*, *data=None*)

将一个 URL 形式的网络对象复制为本地文件。如果 URL 指向一个本地文件，则必须提供文件名

才会复制对象。返回一个元组 (filename, headers) 其中 *filename* 为保存该对象的本地文件名，而 *headers* 是由*urlopen()* 返回的对象的 info() 方法的返回结果（对于远程对象）。可引发的异常与*urlopen()* 的相同。

第二个参数指定文件的保存位置（若未给出，则会是名称随机生成的临时文件）。第三个参数是个可调用对象，在建立网络连接时将会调用一次，之后每次读完数据块后会调用一次。该可调用对象将会传入 3 个参数：已传输的块数、块的大小（以字节为单位）和文件总的大小。如果面对的是老旧 FTP 服务器，文件大小参数可能会是 -1，这些服务器响应读取请求时不会返回文件大小。

以下例子演示了大部分常用场景：

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

如果 *url* 使用 http: 方式的标识符，则可能给出可选的 *data* 参数来指定一个 POST 请求 (通常的请求类型为 GET)。*data* 参数必须是标准 *application/x-www-form-urlencoded* 格式的字节串对象；参见*urllib.parse.urlencode()* 函数。

urlretrieve() 在检测到可用数据少于预期大小（即由 *Content-Length* 标头所报告的大小）时将引发*ContentTooShortError*。例如，这可能会在下载被中断时发生。when it detects that the amount of data available was less than the expected amount (which is the size reported by aheader). This can occur, for example, when the download is interrupted.

Content-Length 会被视为大小的下限：如果存在更多的可用数据，urlretrieve 会读取更多的数据，但是如果可用数据少于该值，则会引发异常。

在此情况下你仍然能够获取已下载的数据，它将保存在异常实例的 content 属性中。

如果未提供 *Content-Length* 标头，urlretrieve 就无法检查它所下载的数据大小，只是简单地返回它。在这种情况下你只能假定下载是成功的。

urllib.request.**urlcleanup**()

清理之前调用*urlretrieve()* 时可能留下的临时文件。

class urllib.request.**URLopener**(*proxies=None*, ***x509*)

自 3.3 版本弃用.

用于打开和读取 URL 的基类。除非你需要支持使用 http:, ftp: 或 file: 以外的方式来打开对象，那你也许可以使用*FancyURLopener*。

在默认情况下，*URLopener* 类会发送一个内容为 urllib/VVV 的 *User-Agent* 标头，其中 *VVV* 是*urllib* 的版本号。应用程序可以通过子类化*URLopener* 或*FancyURLopener* 并在子类定义中将类属性*version* 设为适当的字符串值来定义自己的 *User-Agent* 标头。

可选的 *proxies* 形参应当是一个将方式名称映射到代理 URL 的字典，如为空字典则会完全关闭代理。它的默认值为 None，在这种情况下如果存在环境代理设置则将使用它，正如上文*urlopen()* 的定义中所描述的。

一些归属于 *x509* 的额外关键字形参可在使用 https: 方式时被用于客户端的验证。支持通过关键字 *key_file* 和 *cert_file* 来提供 SSL 密钥和证书；对客户端验证的支持需要提供这两个形参。

如果服务器返回错误代码则*URLopener* 对象将引发*OSError* 异常。

open(*fullurl*, *data=None*)

使用适当的协议打开 *fullurl*。此方法会设置缓存和代理信息，然后调用适当的打开方法并附带其输入参数。如果方式无法被识别，则会调用*open_unknown()*。*data* 参数的含义与*urlopen()* 中的 *data* 参数相同。

此方法总是会使用*quote()* 来对 *fullurl* 进行转码。

open_unknown(*fullurl*, *data=None*)

用于打开未知 URL 类型的可重载接口。

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)

提取 *url* 的内容并将其存放到 *filename* 中。返回值为元组，由一个本地文件名和一个包含响应标头 (对于远程 URL) 的*email.message.Message* 对象或者 None (对于本地 URL)。之后调用方必须打开并读取 *filename* 的内容。如果 *filename* 未给出并且 URL 指向一个本地文件，则会返回输入文件名。如果 URL 非本地并且 *filename* 未给出，则文件名为带有与输入 URL 的路径末尾部分后缀相匹配的后缀的*tempfile.mktemp()* 的输出。如果给出了 *reporthook*，它必须为接受三个数字形参的函数：数据分块编号、读入分块的最大数据量和下载的总数据量 (未知则为 -1)。它将在开始时和从网络读取每个数据分块之后被调用。对于本地 URL *reporthook* 会被忽略。

如果 *url* 使用 http: 方式的标识符，则可能给出可选的 *data* 参数来指定一个 POST 请求 (通常的请求类型为 GET)。*data* 参数必须为标准的 *application/x-www-form-urlencoded* 格式；参见*urllib.parse.urlencode()* 函数。

version

指明打开器对象的用户代理名称的变量。以便让*urllib* 告诉服务器它是某个特定的用户代理，请在子类中将其作为类变量来设置或是在调用基类构造器之前在构造器中设置。

class urllib.request.**FancyURLopener**(...)

自 3.3 版本弃用.

FancyURLopener 子类化了*URLopener* 以提供对以下 HTTP 响应代码的默认处理: 301, 302, 303, 307 和 401。对于上述的 30x 响应代码，会使用 *Location* 标头来获取实际 URL。对于 401 响应代码 (authentication required)，则会执行基本 HTTP 验证。对于 30x 响应代码，递归层数会受 *maxtries* 属性值的限制，该值默认为 10。

对于所有其他响应代码，将会调用*http_error_default()* 方法，你可以在子类中重写此方法来正确处理错误。

备注

根据 **RFC 2616** 的说明，对 POST 请求的 301 和 302 响应不可在未经用户确认的情况下自动进行重定向。在现实情况下，浏览器确实允许自动重定义这些响应，将 POST 更改为 GET，于是*urllib* 就会复现这种行为。

传给此构造器的形参与*URLopener* 的相同。

备注

当执行基本验证时，*FancyURLopener* 实例会调用其*prompt_user_passwd()* 方法。默认的实现将向用户询问控制终端所要求的信息。如有必要子类可以重写此方法来支持更适当的行为。

FancyURLopener 类附带了一个额外方法，它应当被重载以提供适当的行为:

prompt_user_passwd(*host*, *realm*)

返回指定的安全体系下在给定的主机上验证用户所需的信息。返回的值应当是一个元组 (user, password)，它可被用于基本验证。

该实现会在终端上提示此信息；应用程序应当重写此方法以使用本地环境下适当的交互模型。

22.4.25 urllib.request 的限制

- 目前，仅支持下列协议: HTTP (0.9 和 1.0 版), FTP, 本地文件, 以及数据 URL。

  在 3.4 版本发生变更: 增加了对数据 URL 的支持。

- *urlretrieve()* 的缓存特性已被禁用，等待有人有时间去正确地解决过期时间标头的处理问题。
- 应当有一个函数来查询特定 URL 是否在缓存中。

- 为了保持向下兼容性，如果某个 URL 看起来是指向本地文件但该文件无法被打开，则该 URL 会使用 FTP 协议来重新解读。这有时可能会导致令人迷惑的错误消息。
- *urlopen()* 和*urlretrieve()* 函数在等待网络连接建立时会导致任意长时间的延迟。这意味着在不使用线程的情况下搭建一个可交互的 Web 客户端是非常困难的。
- 由*urlopen()* 或*urlretrieve()* 返回的数据就是服务器所返回的原始数据。这可以是二进制数据（如图片）、纯文本或 HTML 代码等。HTTP 协议在响应标头中提供了类型信息，这可以通过读取 *Content-Type* 标头来查看。如果返回的数据是 HTML，你可以使用*html.parser* 模块来解析它。
- 处理 FTP 协议的代码无法区分文件和目录。这在尝试读取指向不可访问的 URL 时可能导致意外的行为。如果 URL 以一个 / 结束，它会被认为指向一个目录并将作相应的处理。但是如果读取一个文件的尝试导致了 550 错误（表示 URL 无法找到或不可访问，这常常是由于权限原因），则该路径会被视为一个目录以便处理 URL 是指定一个目录但略去了末尾 / 的情况。这在你尝试获取一个因其设置了读取权限因而无法访问的文件时会造成误导性的结果；FTP 代码将尝试读取它，因 550 错误而失败，然后又为这个不可读取的文件执行目录列表操作。如果需要细粒度的控制，请考虑使用*ftplib* 模块，子类化*FancyURLopener*，或是修改 *_urlopener* 来满足你的需求。

22.5 `urllib.response` --- urllib 使用的 Response 类

urllib.response 模块定义了一些函数和提供最小化文件类接口包括 `read()` 和 `readline()` 等的类。此模块定义的函数会由*urllib.request* 模块在内部使用。典型的响应对象是一个*urllib.response.addinfourl* 实例:

class urllib.response.**addinfourl**

url

已读取资源的 URL，通常用于确定是否进行了重定向。

headers

以*EmailMessage* 实例的形式返回响应的标头。

status

Added in version 3.9.

由服务器返回的状态码。

geturl()

自 3.9 版本弃用: 已弃用，建议改用*url*。

info()

自 3.9 版本弃用: 已弃用，建议改用*headers*。

code

自 3.9 版本弃用: 已弃用，建议改用*status*。

getcode()

自 3.9 版本弃用: 已弃用，建议改用*status*。

22.6 `urllib.parse` --- 将 URL 解析为组件

源代码: Lib/urllib/parse.py

该模块定义了一个标准接口，用于将统一资源定位符（URL）字符串拆分为不同部分（协议、网络位置、路径等），或将各个部分组合回 URL 字符串，并将“相对 URL”转换为基于给定的“基准 URL”的绝对 URL。

该模块被设计为匹配针对相对统一资源定位符的互联网 RFC。它支持下列 URL 类别: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `itms-services`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtsps`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`。

包括 itms-services URL 类别可以防止 app 为 macOS 和 iOS App 商店传递 Apple 的 App 商店评论进程。对 itms-services 类别的处理在 iOS 上总是会被移除；在 macOS 上，如果 CPython 编译时附带了 --with-app-store-compliance 选项则它可以被移除。

urllib.parse 模块定义的函数可分为两个主要门类: URL 解析和 URL 转码。这些函数将在以下各节中详细说明。

本模块的函数使用已弃用的术语 netloc (或 net_loc)，它是在 **RFC 1808** 中引入的。但是，此术语在 **RFC 3986** 引入术语 authority 作为其替代后已过时。netloc 仅为向下兼容而继续被使用。

22.6.1 URL 解析

URL 解析函数用于将一个 URL 字符串分割成其组成部分，或者将 URL 的多个部分组合成一个 URL 字符串。

urllib.parse.**urlparse**(*urlstring*, *scheme=''*, *allow_fragments=True*)

将一个 URL 解析为六个部分，返回一个包含 6 项的*named tuple*。这对应于 URL 的主要结构: scheme://netloc/path;parameters?query#fragment。每个元组项均为字符串，可能为空字符串。这些部分不会再被拆分为更小的部分（例如，netloc 将为单个字符串），并且% 转义不会被扩展。上面显示的分隔符不会出现在结果中，只有 *path* 部分的开头斜杠例外，它如果存在则会被保留。例如:

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?query#fragment")
ParseResult(scheme='scheme', netloc='netloc', path='/path;parameters', params='',
            query='query', fragment='fragment')
>>> o = urlparse("http://docs.python.org:80/3/library/urllib.parse.html?"
...              "highlight=params#url-parsing")
>>> o
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params'
```

根据 **RFC 1808** 中的语法规范，urlparse 仅在 netloc 前面正确地附带了'//' 的情况下才会识别它。否则输入会被当作是一个相对 URL 因而以路径的组成部分开头。

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

scheme 参数给出了默认的协议，只有在 URL 未指定协议的情况下才会被使用。它应该是与 *urlstring* 相同的类型（文本或字节串），除此之外默认值 '' 也总是被允许，并会在适当情况下自动转换为 b''。

如果 *allow_fragments* 参数为假值，则片段标识符不会被识别。它们会被解析为路径、参数或查询部分，在返回值中 fragment 会被设为空字符串。

返回值是一个*named tuple*，这意味着它的条目可以通过索引或作为命名属性来访问，这些属性是：

属性	索引	值	值（如果不存在）
scheme	0	URL 协议说明符	*scheme* 参数
netloc	1	网络位置部分	空字符串
path	2	分层路径	空字符串
params	3	最后路径元素的参数	空字符串
query	4	查询组件	空字符串
fragment	5	片段标识符	空字符串
username		用户名	*None*
password		密码	*None*
hostname		主机名（小写）	*None*
port		端口号为整数（如果存在）	*None*

如果在 URL 中指定了无效的端口，读取 port 属性将引发*ValueError*。有关结果对象的更多信息请参阅结构化解析结果 一节。

在 netloc 属性中不匹配的方括号将引发*ValueError*。

如果 netloc 属性中的字符在 NFKC 规范化下（如 IDNA 编码格式所使用的）被分解成 /, ?, #, @ 或 : 则将引发*ValueError*。如果在解析之前 URL 就被分解，则不会引发错误。

与所有具名元组的情况一样，该子类还有一些特别有用的附加方法和属性。其中一个方法是 _replace()。_replace() 方法将返回一个新的 ParseResult 对象来将指定字段替换为新的值。

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

警告

urlparse() 不会执行验证。请参阅*URL 解析安全* 了解详情。

在 3.2 版本发生变更: 添加了 IPv6 URL 解析功能。

在 3.3 版本发生变更: 现在会针对所有 URL 方案解析此片段（除非 *allow_fragments* 为假值），以符合 **RFC 3986** 规范。在之前版本中，存在一个支持片段的方案的允许名单。

在 3.6 版本发生变更: 超范围的端口号现在会引发*ValueError*，而不是返回*None*。

在 3.8 版本发生变更: 在 NFKC 规范化下会影响 netloc 解析的字符现在将引发*ValueError*。

urllib.parse.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

解析以字符串参数形式（类型为 *application/x-www-form-urlencoded* 的数据）给出的查询字符串。返回字典形式的数据。结果字典的键为唯一的查询变量名而值为每个变量名对应的值列表。

可选参数 *keep_blank_values* 是一个旗标，指明是否要将以百分号转码的空值作为空字符串处理。真值表示空值应当被保留作为空字符串。默认的假值表示空值会被忽略并将其视作未包括的值。

可选参数 *strict_parsing* 是一个旗标，指明要如何处理解析错误。如为假值（默认），错误会被静默地忽略。如为真值，错误会引发*ValueError* 异常。

可选的 *encoding* 和 *errors* 形参指定如何将以百分号编码的序列解码为 Unicode 字符，即作为*bytes.decode()* 方法所接受的数据。

可选参数 *max_num_fields* 是要读取的最大字段数量的。如果设置，则如果读取的字段超过 *max_num_fields* 会引发`ValueError`。

可选参数 *separator* 是用来分隔查询参数的符号。默认为 `&`。

使用`urllib.parse.urlencode()` 函数 (并将 `doseq` 形参设为 `True`) 将这样的字典转换为查询字符串。

在 3.2 版本发生变更: 增加了 *encoding* 和 *errors* 形参。

在 3.8 版本发生变更: 增加了 *max_num_fields* 形参。

在 3.10 版本发生变更: 增加了 *separator* 形参，默认值为 `&`。Python 在早于 Python 3.10 的版本中允许使用 `;` 和 `&` 作为查询参数分隔符。此设置已被改为只允许单个分隔符键，并以 `&` 作为默认的分隔符。

urllib.parse.**parse_qsl**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

解析以字符串参数形式（类型为 `application/x-www-form-urlencoded` 的数据）给出的查询字符串。数据以字段名和字段值对列表的形式返回。

可选参数 *keep_blank_values* 是一个旗标，指明是否要将以百分号转码的空值作为空字符串处理。真值表示空值应当被保留作为空字符串。默认的假值表示空值会被忽略并将其视作未包括的值。

可选参数 *strict_parsing* 是一个旗标，指明要如何处理解析错误。如为假值（默认），错误会被静默地忽略。如为真值，错误会引发`ValueError` 异常。

可选的 *encoding* 和 *errors* 形参指定如何将以百分号编码的序列解码为 Unicode 字符，即作为`bytes.decode()` 方法所接受的数据。

可选参数 *max_num_fields* 是要读取的最大字段数量的。如果设置，则如果读取的字段超过 *max_num_fields* 会引发`ValueError`。

可选参数 *separator* 是用来分隔查询参数的符号。默认为 `&`。

使用`urllib.parse.urlencode()` 函数将这样的名值对列表转换为查询字符串。

在 3.2 版本发生变更: 增加了 *encoding* 和 *errors* 形参。

在 3.8 版本发生变更: 增加了 *max_num_fields* 形参。

在 3.10 版本发生变更: 增加了 *separator* 形参，默认值为 `&`。Python 在早于 Python 3.10 的版本中允许使用 `;` 和 `&` 作为查询参数分隔符。此设置已被改为只允许单个分隔符键，并以 `&` 作为默认的分隔符。

urllib.parse.**urlunparse**(*parts*)

根据 `urlparse()` 所返回的元组来构造一个 URL。*parts* 参数可以是任何包含六个条目的可迭代对象。构造的结果可能是略有不同但保持等价的 URL，如果被解析的 URL 原本包含不必要的分隔符（例如，带有空查询的 `?`；RFC 已声明这是等价的）。

urllib.parse.**urlsplit**(*urlstring*, *scheme=''*, *allow_fragments=True*)

此函数类似于`urlparse()`，但不会拆分来自 URL 的参数。此函数通常应当在需要允许将参数应用到 URL 的 *path* 部分的每个分节的较新的 URL 语法的情况下 (参见 **RFC 2396**) 被用来代替`urlparse()`。需要使用一个拆分函数来拆分路径分节和参数。此函数将返回包含 5 个条目的*named tuple*:

```
(寻址方案，网络位置，路径，查询，片段标识符)。
```

返回值是一个*named tuple*，它的条目可以通过索引或作为命名属性来访问:

属性	索引	值	值（如果不存在）
scheme	0	URL 协议说明符	*scheme* 参数
netloc	1	网络位置部分	空字符串
path	2	分层路径	空字符串
query	3	查询组件	空字符串
fragment	4	片段标识符	空字符串
username		用户名	*None*
password		密码	*None*
hostname		主机名（小写）	*None*
port		端口号为整数（如果存在）	*None*

如果在 URL 中指定了无效的端口，读取 `port` 属性将引发*ValueError*。有关结果对象的更多信息请参阅结构化解析结果 一节。

在 `netloc` 属性中不匹配的方括号将引发*ValueError*。

如果 `netloc` 属性中的字符在 NFKC 规范化下（如 IDNA 编码格式所使用的）被分解成 `/`, `?`, `#`, `@` 或 `:` 则将引发*ValueError*。如果在解析之前 URL 就被分解，则不会引发错误。

按照针对 RFC 3986 进行更新的 WHATWG spec，打头的 C0 控制符和空格符将从 URL 中去除。任意位置上的 `\n`, `\r` 和制表符 `\t` 等字符也将从 URL 中去除。at any position.

警告

urlsplit() 不会执行验证。请参阅*URL 解析安全* 了解详情。

在 3.6 版本发生变更: 超范围的端口号现在会引发*ValueError*，而不是返回*None*。

在 3.8 版本发生变更: 在 NFKC 规范化下会影响 netloc 解析的字符现在将引发*ValueError*。

在 3.10 版本发生变更: ASCII 换行符和制表符会从 URL 中被去除。

在 3.12 版本发生变更: 打头的 WHATWG C0 控制符和空格符将从 URL 中去除。

`urllib.parse.`**`urlunsplit`**(*parts*)

将*urlsplit()* 所返回的元组中的元素合并为一个字符串形式的完整 URL。*parts* 参数可以是任何包含五个条目的可迭代对象。其结果可能是略有不同但保持等价的 URL，如果被解析的 URL 原本包含不必要的分隔符（例如，带有空查询的?；RFC 已声明这是等价的）。

`urllib.parse.`**`urljoin`**(*base*, *url*, *allow_fragments=True*)

通过合并一个" 基准 URL" (*base*) 和另一个 URL (*url*) 来构造一个完整 ("absolute") URL。在非正式情况下，这将使用基准 URL 的各部分，特别是地址协议、网络位置和 (一部分) 路径来提供相对 URL 中缺失的部分。例如:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

allow_fragments 参数具有与*urlparse()* 中的对应参数一致的含义与默认值。

备注

如果 *url* 为绝对 URL (即以 `//` 或 `scheme://` 打头)，则 *url* 的主机名和/或协议将出现在结果中。例如:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

如果你不想要那样的行为，请使用*urlsplit()* 和*urlunsplit()* 对 *url* 进行预处理，移除可能存在的 *scheme* 和 *netloc* 部分。

警告

因为可以将一个绝对 URL 作为 `url` 形参传入，对由攻击者控制的 `url` 使用 `urljoin` 通常是 **不安全的**。例如，在 `urljoin("https://website.com/users/", username)` 中，如果 `username` 可以包含一个绝对 URL，`urljoin` 的结果也将为绝对 URL。

在 3.5 版本发生变更: 更新行为以匹配 **RFC 3986** 中定义的语义。

`urllib.parse.`**`urldefrag`**(*url*)

如果 *url* 包含片段标识符，则返回不带片段标识符的 *url* 修改版本。如果 *url* 中没有片段标识符，则返回未经修改的 *url* 和一个空字符串。

返回值是一个*named tuple*，它的条目可以通过索引或作为命名属性来访问:

属性	索引	值	值（如果不存在）
`url`	0	不带片段的 URL	空字符串
`fragment`	1	片段标识符	空字符串

请参阅*结构化解析结果* 一节了解有关结果对象的更多信息。

在 3.2 版本发生变更: 结果为已构造好的对象而不是一个简单的 2 元组。-tuple.

`urllib.parse.`**`unwrap`**(*url*)

从已包装的 URL (即被格式化为 `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` 或 `scheme://host/path` 的字符串) 中提取 URL。如果 *url* 不是一个已包装的 URL，它将被原样返回。

22.6.2 URL 解析安全

urlsplit() 和*urlparse()* API 不会对输入进行 **验证**。它们可能不会因其他应用程序认为不合法的输入而引发错误。它们还可能在其他地方认为不是 URL 的输入上成功运行。它们的目标是达成实际的功能而不是保持纯净。

他们在非正常的输入上可能不会引发异常，而是以空字符串的形式返回某些部分。或者可能会包含某些不应包含的部分。

我们建议这些 API 的用户在任何使用的值具有安全意义的地方应用防御性代码。在你的代码中进行某些验证之后再信任被返回的组件。这个 `scheme` 合理吗？那个 `path` 正确吗？那个 `hostname` 是否存在怪异之处？等等。

一个 URL 由哪些内容组成并没有通用的良好定义。不同应用程序有不同的需求和想要的约束。举例来说现有的 WHATWG spec 描述了面向用户的 Web 客户端如 Web 浏览器的需求。而 **RFC 3986** 则更为一般化。这些函数涵盖了这两种领域的某些部分，但称不上能兼容任何一种。这些 API 和早于这两个标准的现有用户代码对于其他特定行为的期望使得我们对 API 行为的更改变得非常谨慎。

22.6.3 解析 ASCII 编码字节

这些 URL 解析函数最初设计只用于操作字符串。但在实践中，它也能够操作经过正确转码和编码的 ASCII 字节序列形式的 URL。相应地，此模块中的 URL 解析函数既可以操作*str* 对象也可以操作*bytes* 和*bytearray* 对象。

如果传入*str* 数据，结果将只包含*str* 数据。如果传入*bytes* 或*bytearray* 数据，则结果也将只包含*bytes* 数据。

试图在单个函数调用中混用*str* 数据和*bytes* 或*bytearray* 数据将导致引发*TypeError*，而试图传入非 ASCII 字节值则将引发*UnicodeDecodeError*。

为了支持结果对象在*str* 和*bytes* 之间方便地转换，所有来自 URL 解析函数的返回值都会提供 encode() 方法 (当结果包含*str* 数据) 或 decode() 方法 (当结果包含*bytes* 数据)。这些方法的签名与*str* 和*bytes* 的对应方法相匹配 (不同之处在于其默认编码格式是 'ascii' 而非 'utf-8')。每个方法会输出包含相应类型的*bytes* 数据 (对于 encode() 方法) 或*str* 数据 (对于 decode() 方法) 的值。

对于某些需要在有可能不正确地转码的包含非 ASCII 数据的 URL 上进行操作的应用程序来说，在唤起 URL 解析方法之前必须自行将字节串解码为字符。

在本节中描述的行为仅适用于 URL 解析函数。URL 转码函数在产生和消耗字节序列时使用它们自己的规则，详情参见单独 URL 转码函数的文档。

在 3.2 版本发生变更: URL 解析函数现在接受 ASCII 编码的字节序列

22.6.4 结构化解析结果

urlparse(), *urlsplit()* 和*urldefrag()* 函数的结果对象是*tuple* 类型的子类。这些子类中增加了在那些函数的文档中列出的属性，之前小节中描述的编码和解码支持，以及一个附加方法:

urllib.parse.SplitResult.**geturl**()

以字符串形式返回原始 URL 的重合并版本。这可能与原始 URL 有所不同，例如协议的名称可能被正规化为小写字母、空的组成部分可能被丢弃。特别地，空的参数、查询和片段标识符将会被移除。

对于*urldefrag()* 的结果，只有空的片段标识符会被移除。对于*urlsplit()* 和*urlparse()* 的结果，所有被记录的改变都会被应用到此方法所返回的 URL 上。

如果是通过原始的解析方法传回则此方法的结果会保持不变:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

下面的类提供了当在*str* 对象上操作时对结构化解析结果的实现:

class urllib.parse.**DefragResult**(*url*, *fragment*)

用于*urldefrag()* 结果的实体类，包含有*str* 数据。encode() 方法会返回一个*DefragResultBytes* 实例。

Added in version 3.2.

class urllib.parse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

用于*urlparse()* 结果的实体类，包含有*str* 数据。encode() 方法会返回一个*ParseResultBytes* 实例。

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)

用于*urlsplit()* 结果的实体类，包含有*str* 数据。encode() 方法会返回一个*SplitResultBytes* 实例。

下面的类提供了当在*bytes* 或*bytearray* 对象上操作时对解析结果的实现:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)

用于*urldefrag()* 结果的实体类，包含有*bytes* 数据。decode() 方法会返回一个*DefragResult* 实例。

Added in version 3.2.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

用于*urlparse()* 结果的实体类，包含有*bytes* 数据。decode() 方法会返回一个*ParseResult* 实例。

Added in version 3.2.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)

用于*urlsplit()* 结果的实体类，包含有*bytes* 数据。decode() 方法会返回一个*SplitResult* 实例。

Added in version 3.2.

22.6.5 URL 转码

URL 转码函数的功能是接收程序数据并通过对特殊字符进行转码并正确编码非 ASCII 文本来将其转为可以安全地用作 URL 组成部分的形式。它们还支持逆转此操作以便从作为 URL 组成部分的内容中重建原始数据，如果上述的 URL 解析函数还未覆盖此功能的话。

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)

使用 %*xx* 转义符替换 *string* 中的特殊字符。字母、数字和 '_.-~' 等字符一定不会被转码。在默认情况下，此函数只对 URL 的路径部分进行转码。可选的 *safe* 形参额外指定不应被转码的 ASCII 字符 --- 其默认值为 '/'。

string 可以是*str* 或*bytes* 对象。

在 3.7 版本发生变更: 从 **RFC 2396** 迁移到 **RFC 3986** 以转码 URL 字符串。"~" 现在已被包括在非保留字符集中。

可选的 *encoding* 和 *errors* 形参指明如何处理非 ASCII 字符，与*str.encode()* 方法所接受的值一样。*encoding* 默认为 'utf-8'。*errors* 默认为 'strict'，表示不受支持的字符将引发*UnicodeEncodeError*。如果 *string* 为*bytes* 则不可提供 *encoding* 和 *errors*，否则将引发*TypeError*。

请注意 quote(string, safe, encoding, errors) 等价于 quote_from_bytes(string.encode(encoding, errors), safe)。

例如: quote('/El Niño/') 将产生 '/El%20Ni%C3%B1o/'。

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)

类似于*quote()*，但还会使用加号来替换空格，如在构建放入 URL 的查询字符串时对于转码 HTML 表单值时所要求的那样。原始字符串中的加号会被转义，除非它们已包括在 *safe* 中。它也不会将 *safe* 的默认值设为 '/'。

例如: quote_plus('/El Niño/') 将产生 '%2FEl+Ni%C3%B1o%2F'。

urllib.parse.**quote_from_bytes**(*bytes*, *safe='/'*)

类似于*quote()*，但是接受*bytes* 对象而非*str*，并且不执行从字符串到字节串的编码。

例如: quote_from_bytes(b'a&\xef') 将产生 'a%26%EF'。

urllib.parse.**unquote**(*string*, *encoding='utf-8'*, *errors='replace'*)

将 %*xx* 转义符替换为等效的单字符。可选的 *encoding* 和 *errors* 形参指定如何将以百分号编码的序列解码为 Unicode 字符，即*bytes.decode()* 方法所接受的形式。

string 可以是*str* 或*bytes* 对象。

encoding 默认为 'utf-8'。*errors* 默认为 'replace'，表示无效的序列将被替换为占位字符。

例如: unquote('/El%20Ni%C3%B1o/') 将产生 '/El Niño/'。

在 3.9 版本发生变更: *string* 形参支持 bytes 和 str 对象（之前仅支持 str）。

urllib.parse.**unquote_plus**(*string*, *encoding='utf-8'*, *errors='replace'*)

类似于*unquote()*，但还会将加号替换为空格，如反转码表单值所要求的。

string 必须为*str*。

例如: unquote_plus('/El+Ni%C3%B1o/') 将产生 '/El Niño/'。

urllib.parse.**unquote_to_bytes**(*string*)

用等价的单八位形式替换 %*xx* 转义码，并返回一个*bytes* 对象。

string 可以是*str* 或*bytes* 对象。

如果它是*str*，则 *string* 中未转义的非 ASCII 字符会被编码为 UTF-8 字节串。

例如: unquote_to_bytes('a%26%EF') y 将产生 b'a&\xef'。

urllib.parse.**urlencode**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)

将一个包含有*str* 或*bytes* 对象的映射对象或二元组序列转换为以百分号编码的 ASCII 文本字符串。如果所产生的字符串要被用作*urlopen()* 函数的 POST 操作的 *data*，则它应当被编码为字节串，否则它将导致*TypeError*。

结果字符串是一系列 key=value 对，由 '&' 字符进行分隔，其中 *key* 和 *value* 都已使用 *quote_via* 函数转码。在默认情况下，会使用*quote_plus()* 来转码值，这意味着空格会被转码为 '+' 字符而'/' 字符会被转码为 %2F，即遵循 GET 请求的标准 (application/x-www-form-urlencoded)。另一个可以作为 *quote_via* 传入的替代函数是*quote()*，它将把空格转码为 %20 并且不编码'/' 字符。为了最大程度地控制要转码的内容，请使用 quote 并指定 *safe* 的值。

当使用二元组序列作为 *query* 参数时，每个元组的第一个元素为键而第二个元素为值。值元素本身也可以为一个序列，在那种情况下，如果可选的形参 *doseq* 的值为 True，则每个键的值序列元素生成单个 key=value 对（以 '&' 分隔）。被编码的字符串中的参数顺序将与序列中的形参元素顺序相匹配。

safe, *encoding* 和 *errors* 形参会被传递给 *quote_via* (*encoding* 和 *errors* 形参仅在查询元素为*str* 时会被传递)。

为了反向执行这个编码过程，此模块提供了*parse_qs()* 和*parse_qsl()* 来将查询字符串解析为 Python 数据结构。

请参考*urllib 示例* 来了解如何使用*urllib.parse.urlencode()* 方法来生成 URL 的查询字符串或 POST 请求的数据。

在 3.2 版本发生变更: 查询支持字节和字符串对象。

在 3.5 版本发生变更: 增加了 *quote_via* 形参。

参见

WHATWG - URL 现有标准
定义 URL、域名、IP 地址、application/x-www-form-urlencoded 格式及其 API 的工作组。

RFC 3986 - 统一资源标识符
这是当前的标准 (STD66)。任何对于 urllib.parse 模块的修改都必须遵循该标准。某些偏离也可能会出现，这大都是出于向下兼容的目的以及特定的经常存在于各主要浏览器上的实际解析需求。

RFC 2732 - URL 中的 IPv6 Addresses 地址显示格式。
这指明了 IPv6 URL 的解析要求。

RFC 2396 - 统一资源标识符（URI）：通用语法
描述统一资源名称 (URN) 和统一资源定位符 (URL) 通用语义要求的文档。

RFC 2368 - mailto URL 模式。
mailto URL 模式的解析要求。

RFC 1808 - 相对统一资源定位符
这个请求注释包括联结绝对和相对 URL 的规则，其中包括大量控制边界情况处理的" 异常示例"。

RFC 1738 - 统一资源定位符 (URL)
这指明了绝对 URL 的正式语义和句法。

22.7 urllib.error --- 由 urllib.request 引发的异常类

源代码： Lib/urllib/error.py

---

urllib.error 模块为*urllib.request* 所引发的异常定义了异常类。基础异常类是*URLError*。

下列异常会被*urllib.error* 按需引发：

exception urllib.error.**URLError**

处理程序在遇到问题时会引发此异常（或其派生的异常）。它是*OSError* 的一个子类。

reason

此错误的原因。它可以是一个消息字符串或另一个异常实例。

在 3.3 版本发生变更: *URLError* 曾经是*IOError* 的子类型，现在它是*OSError* 的一个别名。

exception urllib.error.**HTTPError**(*url*, *code*, *msg*, *hdrs*, *fp*)

虽然是一个异常（*URLError* 的一个子类），*HTTPError* 也可以作为一个非异常的文件类返回值（与*urlopen()* 返所回的对象相同）。这适用于处理特殊 HTTP 错误例如作为认证请求的时候。

url

包含请求 URL。是 *filename* 属性的别名。

code

一个 HTTP 状态码，具体定义见 **RFC 2616**。这个数字的值对应于存放在*http.server.BaseHTTPRequestHandler.responses* 代码字典中的某个值。

reason

这通常是一个解释本次错误原因的字符串。为 *msg* 属性的别名。

headers

导致*HTTPError* 的特定 HTTP 请求的 HTTP 响应头。为 *hdrs* 属性的别名。

Added in version 3.4.

fp

可供读取 HTTP 错误消息体的文件型对象。

exception urllib.error.**ContentTooShortError**(*msg*, *content*)

此异常会在*urlretrieve()* 函数检测到已下载的数据量少于预期量（由 *Content-Length* 标头给出）时被引发。

content

已下载（并可能被截断）的数据。

22.8 urllib.robotparser --- 用于 robots.txt 的解析器

源代码： Lib/urllib/robotparser.py

此模块提供了一个单独的类*RobotFileParser*，它可以回答关于某个特定用户代理能否在发布了 `robots.txt` 文件的网站抓取特定 URL 的问题。有关 `robots.txt` 文件结构的更多细节，请参阅 http://www.robotstxt.org/orig.html。

class urllib.robotparser.**RobotFileParser**(*url=''*)

这个类提供了一些可以读取、解析和回答关于 *url* 上的 `robots.txt` 文件的问题的方法。

set_url(*url*)

设置指向 `robots.txt` 文件的 URL。

read()
读取 robots.txt URL 并将其输入解析器。

parse(*lines*)
解析行参数。

can_fetch(*useragent*, *url*)
如果允许 *useragent* 按照被解析 robots.txt 文件中的规则来获取 *url* 则返回 True。

mtime()
返回最近一次获取 robots.txt 文件的时间。这适用于需要定期检查 robots.txt 文件更新情况的长时间运行的网页爬虫。

modified()
将最近一次获取 robots.txt 文件的时间设置为当前时间。

crawl_delay(*useragent*)
为指定的 *useragent* 从 robots.txt 返回 Crawl-delay 形参。如果此形参不存在或不适用于指定的 *useragent* 或者此形参的 robots.txt 条目存在语法错误，则返回 None。

Added in version 3.6.

request_rate(*useragent*)
以*named tuple* RequestRate(requests, seconds) 的形式从 robots.txt 返回 Request-rate 形参的内容。如果此形参不存在或不适用于指定的 *useragent* 或者此形参的 robots.txt 条目存在语法错误，则返回 None。

Added in version 3.6.

site_maps()
以*list()* 的形式从 robots.txt 返回 Sitemap 形参的内容。如果此形参不存在或者此形参的 robots.txt 条目存在语法错误，则返回 None。

Added in version 3.8.

下面的例子演示了*RobotFileParser* 类的基本用法:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

22.9 http --- HTTP 模块

源代码: Lib/http/__init__.py

---

http 是一个包，它收集了多个用于处理超文本传输协议的模块:

- *http.client* 是一个底层的 HTTP 协议客户端；对于高层级的 URL 访问请使用*urllib.request*
- *http.server* 包含基于*socketserver* 的基本 HTTP 服务类

- *http.cookies* 包含一些有用来实现通过 cookies 进行状态管理的工具
- *http.cookiejar* 提供了 cookies 的持久化

http 模块还定义了下列枚举来帮助你使用 http 相关的代码:

class http.**HTTPStatus**

Added in version 3.5.

enum.IntEnum 的子类，它定义了组 HTTP 状态码，原理短语以及用英语书写的长描述文本。

用法:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
HTTPStatus.OK
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[HTTPStatus.CONTINUE, HTTPStatus.SWITCHING_PROTOCOLS, ...]
```

22.9.1 HTTP 状态码

已支持的，IANA 注册的状态码 在*http.HTTPStatus* 中可用的有:

双字母代码	映射名	详情
100	CONTINUE: 继续	HTTP Semantics RFC 9110, Section 15.2.1
101	SWITCHING_PROTOCOLS	HTTP Semantics RFC 9110, Section 15.2.2
102	PROCESSING	WebDAV RFC 2518, 10.1 节
103	EARLY_HINTS	用于指定提示 RFC 8297 的 HTTP 状态码
200	OK	HTTP Semantics RFC 9110, Section 15.3.1
201	CREATED	HTTP Semantics RFC 9110, Section 15.3.2
202	ACCEPTED	HTTP Semantics RFC 9110, Section 15.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP Semantics RFC 9110, Section 15.3.4
204	NO_CONTENT: 没有内容	HTTP Semantics RFC 9110, Section 15.3.5
205	RESET_CONTENT	HTTP Semantics RFC 9110, Section 15.3.6
206	PARTIAL_CONTENT	HTTP Semantics RFC 9110, Section 15.3.7
207	MULTI_STATUS	WebDAV RFC 4918, 11.1 节
208	ALREADY_REPORTED	WebDAV Binding Extensions RFC 5842, 7.1
226	IM_USED	Delta Encoding in HTTP RFC 3229, 10.4.1
300	MULTIPLE_CHOICES: 有多种资源可选择	HTTP Semantics RFC 9110, Section 15.4.1
301	MOVED_PERMANENTLY: 永久移动	HTTP Semantics RFC 9110, Section 15.4.2
302	FOUND: 临时移动	HTTP Semantics RFC 9110, Section 15.4.3
303	SEE_OTHER: 已经移动	HTTP Semantics RFC 9110, Section 15.4.4
304	NOT_MODIFIED: 没有修改	HTTP Semantics RFC 9110, Section 15.4.5
305	USE_PROXY: 使用代理	HTTP Semantics RFC 9110, Section 15.4.6
307	TEMPORARY_REDIRECT: 临时重定向	HTTP Semantics RFC 9110, Section 15.4.8
308	PERMANENT_REDIRECT: 永久重定向	HTTP Semantics RFC 9110, Section 15.4.9
400	BAD_REQUEST: 错误请求	HTTP Semantics RFC 9110, Section 15.5.1
401	UNAUTHORIZED: 未授权	HTTP Semantics RFC 9110, Section 15.5.2
402	PAYMENT_REQUIRED: 保留，将来使用	HTTP Semantics RFC 9110, Section 15.5.3
403	FORBIDDEN: 禁止	HTTP Semantics RFC 9110, Section 15.5.4
404	NOT_FOUND: 没有找到	HTTP Semantics RFC 9110, Section 15.5.5

表 1 – 接上页

双字母代码	映射名	详情
405	METHOD_NOT_ALLOWED：该请求方法不允许	HTTP Semantics RFC 9110, Section 15.5.6
406	NOT_ACCEPTABLE：不可接受	HTTP Semantics RFC 9110, Section 15.5.7
407	PROXY_AUTHENTICATION_REQUIRED：要求使用代理验明正身	HTTP Semantics RFC 9110, Section 15.5.8
408	REQUEST_TIMEOUT：请求超时	HTTP Semantics RFC 9110, Section 15.5.9
409	CONFLICT：冲突	HTTP Semantics RFC 9110, Section 15.5.10
410	GONE：已经不在了	HTTP Semantics RFC 9110, Section 15.5.11
411	LENGTH_REQUIRED：长度要求	HTTP Semantics RFC 9110, Section 15.5.12
412	PRECONDITION_FAILED：前提条件错误	HTTP Semantics RFC 9110, Section 15.5.13
413	CONTENT_TOO_LARGE	HTTP Semantics RFC 9110, Section 15.5.14
414	URI_TOO_LONG	HTTP Semantics RFC 9110, Section 15.5.15
415	UNSUPPORTED_MEDIA_TYPE：不支持的媒体格式	HTTP Semantics RFC 9110, Section 15.5.16
416	RANGE_NOT_SATISFIABLE	HTTP Semantics RFC 9110, Section 15.5.17
417	EXPECTATION_FAILED：期望失败	HTTP Semantics RFC 9110, Section 15.5.18
418	IM_A_TEAPOT	HTCPCP/1.0 RFC 2324, Section 2.3.2
421	MISDIRECTED_REQUEST	HTTP Semantics RFC 9110, Section 15.5.20
422	UNPROCESSABLE_CONTENT	HTTP Semantics RFC 9110, Section 15.5.21
423	LOCKED：锁着	WebDAV RFC 4918, 11.3 节
424	FAILED_DEPENDENCY：失败的依赖	WebDAV RFC 4918, 11.4 节
425	TOO_EARLY	使用 HTTP RFC 8470 中的早期数据
426	UPGRADE_REQUIRED：升级需要	HTTP Semantics RFC 9110, Section 15.5.22
428	PRECONDITION_REQUIRED：先决条件要求	Additional HTTP Status Codes RFC 6585
429	TOO_MANY_REQUESTS：太多的请求	Additional HTTP Status Codes RFC 6585
431	REQUEST_HEADER_FIELDS_TOO_LARGE：请求头太大	Additional HTTP Status Codes RFC 6585
451	UNAVAILABLE_FOR_LEGAL_REASONS	HTTP 状态码用于报告法律障碍 RFC 772
500	INTERNAL_SERVER_ERROR：内部服务错误	HTTP Semantics RFC 9110, Section 15.6.1
501	NOT_IMPLEMENTED：不可执行	HTTP Semantics RFC 9110, Section 15.6.2
502	BAD_GATEWAY：无效网关	HTTP Semantics RFC 9110, Section 15.6.3
503	SERVICE_UNAVAILABLE：服务不可用	HTTP Semantics RFC 9110, Section 15.6.4
504	GATEWAY_TIMEOUT：网关超时	HTTP Semantics RFC 9110, Section 15.6.5
505	HTTP_VERSION_NOT_SUPPORTED：HTTP 版本不支持	HTTP Semantics RFC 9110, Section 15.6.6
506	VARIANT_ALSO_NEGOTIATES：服务器存在内部配置错误	透明内容协商在：HTTP RFC 2295, 8.1 节
507	INSUFFICIENT_STORAGE：存储不足	WebDAV RFC 4918, 11.5 节
508	LOOP_DETECTED：循环检测	WebDAV Binding Extensions RFC 5842, 7.2
510	NOT_EXTENDED：不扩展	WebDAV Binding Extensions RFC 5842, 7.2
511	NETWORK_AUTHENTICATION_REQUIRED：要求网络身份验证	Additional HTTP Status Codes RFC 6585, 6

为了保持向后兼容性，枚举值也以常量形式出现在*http.client* 模块中，。枚举名等于常量名 (例如 `http.HTTPStatus.OK` 也可以是 `http.client.OK`)。

在 3.7 版本发生变更: 添加了 `421 MISDIRECTED_REQUEST` 状态码。

Added in version 3.8: 添加了 `451 UNAVAILABLE_FOR_LEGAL_REASONS` 状态码。

Added in version 3.9: 增加了 `103 EARLY_HINTS`, `418 IM_A_TEAPOT` 和 `425 TOO_EARLY` 状态码

在 3.13 版本发生变更: 实现了针对状态常量的 RFC9110 命名。旧常量名被保留用于向下兼容。

22.9.2 HTTP 状态类别

Added in version 3.12.

这些枚举值具有一些用于指明 HTTP 状态类别的特征属性:

特征属性	表示	详情
is_informational	100 <= status <= 199	HTTP Semantics RFC 9110, Section 15
is_success	200 <= status <= 299	HTTP Semantics RFC 9110, Section 15
is_redirection	300 <= status <= 399	HTTP Semantics RFC 9110, Section 15
is_client_error	400 <= status <= 499	HTTP Semantics RFC 9110, Section 15
is_server_error	500 <= status <= 599	HTTP Semantics RFC 9110, Section 15

用法：

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK.is_success
True
>>> HTTPStatus.OK.is_client_error
False
```

class http.**HTTPMethod**

Added in version 3.11.

一个*enum.StrEnum*的子类，它定义了一组 HTTP 方法以及用英文书写的描述。

用法：

```
>>> from http import HTTPMethod
>>>
>>> HTTPMethod.GET
<HTTPMethod.GET>
>>> HTTPMethod.GET == 'GET'
True
>>> HTTPMethod.GET.value
'GET'
>>> HTTPMethod.GET.description
'Retrieve the target.'
>>> list(HTTPMethod)
[<HTTPMethod.CONNECT>,
 <HTTPMethod.DELETE>,
 <HTTPMethod.GET>,
 <HTTPMethod.HEAD>,
 <HTTPMethod.OPTIONS>,
 <HTTPMethod.PATCH>,
 <HTTPMethod.POST>,
 <HTTPMethod.PUT>,
 <HTTPMethod.TRACE>]
```

22.9.3 HTTP 方法

已支持的，IANA 注册的方法 在*http.HTTPMethod*中可用的有：

方法	映射名	详情
GET	GET	HTTP Semantics RFC 9110, Section 9.3.1
HEAD	HEAD	HTTP Semantics RFC 9110, Section 9.3.2
POST	POST	HTTP Semantics RFC 9110, Section 9.3.3
PUT	PUT	HTTP Semantics RFC 9110, Section 9.3.4
DELETE	DELETE	HTTP Semantics RFC 9110, Section 9.3.5
CONNECT	CONNECT	HTTP Semantics RFC 9110, Section 9.3.6
OPTIONS	OPTIONS	HTTP Semantics RFC 9110, Section 9.3.7
TRACE	TRACE	HTTP Semantics RFC 9110, Section 9.3.8
PATCH	PATCH	HTTP/1.1 RFC 5789

22.10 http.client --- HTTP 协议客户端

源代码： Lib/http/client.py

这个模块定义了实现 HTTP 和 HTTPS 协议客户端的类。它通常不直接使用 --- 模块*urllib.request* 会用它来处理使用 HTTP 和 HTTPS 的 URL。

参见

对于更高层级的 HTTP 客户端接口，建议使用 Requests 包。

备注

HTTPS 支持仅在编译 Python 时启用了 SSL 支持的情况下（通过*ssl* 模块）可用。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

该模块支持以下类：

class http.client.**HTTPConnection**(*host*, *port=None*, [*timeout*,]*source_address=None*, *blocksize=8192*)

HTTPConnection 的实例代表与 HTTP 服务器的一个连接事务。它在实例化时应当传入一个主机和可选的端口号。若未传入端口号，则如果主机字符串的形式为 `host:port` 则会从中提取端口，否则将使用默认的 HTTP 端口（80）。如果给出了可选的 *timeout* 形参，则阻塞操作（如连接尝试）将在指定的秒数之后超时（如果未给出，则使用全局默认超时设置）。可选的 *source_address* 形参可以是一个 (host, port) 元组，用作进行 HTTP 连接的源地址。可选的 *blocksize* 形参以字节为单位设置缓冲区的大小，用来发送文件类消息体。

举个例子，以下调用都是创建连接到同一主机和端口的服务器的实例：

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

在 3.2 版本发生变更: 添加了 *source_address* 参数

在 3.4 版本发生变更: 移除了 *strict* 形参。不再支持 HTTP 0.9 风格的“简单响应”。

在 3.7 版本发生变更: 添加了 *blocksize* 参数。

class http.client.**HTTPSConnection**(*host*, *port=None*, *, [*timeout*,]*source_address=None*, *context=None*, *blocksize=8192*)

HTTPConnection 的子类，使用 SSL 与安全服务器进行通信。默认端口为 `443`。如果指定了 *context*，它必须为一个描述 SSL 各选项的*ssl.SSLContext* 实例。

请参阅*安全考量* 了解有关最佳实践的更多信息。

在 3.2 版本发生变更: 添加了 *source_address*, *context* 和 *check_hostname*。

在 3.2 版本发生变更: 这个类现在会在可能的情况下（即当*ssl.HAS_SNI* 为真值时）支持 HTTPS 虚拟主机。

在 3.4 版本发生变更: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的“简单响应”。

在 3.4.3 版本发生变更: 目前这个类在默认情况下会执行所有必要的证书和主机检查。要回复到先前的非验证行为，可以将 `ssl._create_unverified_context()` 传给 *context* 形参。

在 3.8 版本发生变更: 该类现在对于默认的 *context* 或在传入 *cert_file* 并附带自定义 *context* 时会启用 TLS 1.3 *ssl.SSLContext.post_handshake_auth*。

在 3.10 版本发生变更: 现在这个类在未给出 *context* 的时候会发送一个带有协议指示符 `http/1.1` 的 ALPN 扩展。自定义 *context* 应当使用 *set_alpn_protocols()* 来设置 ALPN 协议。

在 3.12 版本发生变更: 已弃用的 *key_file*, *cert_file* 和 *check_hostname* 形参已被移除。

class http.client.**HTTPResponse**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

在成功连接后返回类的实例，而不是由用户直接实例化。

在 3.4 版本发生变更: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的 “简单响应”。

这个模块定义了以下函数:

http.client.**parse_headers**(*fp*)

从一个代表 HTTP 请求/响应的文件指针 *fp* 解析标头。该文件必须是一个 *BufferedIOBase* 读取器（即不为文本）并且必须提供有效的 **RFC 2822** 样式标头。

该函数返回 *http.client.HTTPMessage* 的实例，带有头部各个字段，但不带正文数据（与 *HTTPResponse.msg* 和 *http.server.BaseHTTPRequestHandler.headers* 一样）。返回之后，文件指针 *fp* 已为读取 HTTP 正文做好准备了。

> **备注**
>
> *parse_headers()* 不会解析 HTTP 消息的开始行；只会解析各 `Name: value` 行。文件必须为读取这些字段做好准备，所以在调用该函数之前，第一行应该已经被读取过了。

下列异常可以适当地被引发:

exception http.client.**HTTPException**

此模块中其他异常的基类。它是 *Exception* 的一个子类。

exception http.client.**NotConnected**

HTTPException 的一个子类。

exception http.client.**InvalidURL**

HTTPException 的一个子类，如果给出了一个非数字或为空值的端口就会被引发。

exception http.client.**UnknownProtocol**

HTTPException 的一个子类。

exception http.client.**UnknownTransferEncoding**

HTTPException 的一个子类。

exception http.client.**UnimplementedFileMode**

HTTPException 的一个子类。

exception http.client.**IncompleteRead**

HTTPException 的一个子类。

exception http.client.**ImproperConnectionState**

HTTPException 的一个子类。

exception http.client.**CannotSendRequest**

ImproperConnectionState 的一个子类。

exception http.client.**CannotSendHeader**

ImproperConnectionState 的一个子类。

exception http.client.**ResponseNotReady**

ImproperConnectionState 的一个子类。

exception http.client.**BadStatusLine**

HTTPException 的一个子类。如果服务器反馈了一个我们不理解的 HTTP 状态码就会被引发。

exception http.client.**LineTooLong**

HTTPException 的一个子类。如果在 HTTP 协议中从服务器接收到过长的行就会被引发。

exception http.client.**RemoteDisconnected**

ConnectionResetError 和*BadStatusLine* 的一个子类。当尝试读取响应时的结果是未从连接读取到数据时由*HTTPConnection.getresponse()* 引发，表明远端已关闭连接。

Added in version 3.5: 在此之前引发的异常为*BadStatusLine*('')。

此模块中定义的常量为：

http.client.**HTTP_PORT**

HTTP 协议默认的端口号 (总是 80)。

http.client.**HTTPS_PORT**

HTTPS 协议默认的端口号 (总是 443)。

http.client.**responses**

这个字典把 HTTP 1.1 状态码映射到 W3C 名称。

例如：http.client.responses[http.client.NOT_FOUND] 是 'NOT FOUND （未发现）。

本模块中可用的 HTTP 状态码常量可以参见*HTTP 状态码* 。

22.10.1 HTTPConnection 对象

HTTPConnection 实例拥有以下方法：

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)

这将使用 HTTP 请求方法 *method* 和请求 URI *url* 将服务器发送一个请求。所提供的 *url* 必须是符合 **RFC 2616 §5.1.2** 规范的绝对路径（除非是连接到一个 HTTP 代理服务器或者使用 OPTIONS 或 CONNECT 方法）。

如果给定 *body*，那么给定的数据会在信息头完成之后发送。它可能是一个 字符串，一个*bytes-like object*，一个打开的*file object*，或者*bytes* 迭代器。如果 *body* 是字符串，它会按 HTTP 默认的 ISO-8859-1 编码。如果是一个字节类对象，它会按原样发送。如果是*file object*，文件的内容会被发送，这个文件对象应该至少支持 read() 方法。如果这个文件对象是一个*io.TextIOBase* 实例，由 read() 方法返回的数据会按 ISO-8859-1 编码，否则由 read() 方法返回的数据会按原样发送。如果 *body* 是一个迭代器，迭代器中的元素会被发送，直到迭代器耗尽。

headers 参数应为由要与请求一同发送的额外 HTTP 标头组成的映射。必须提供一个 **主机标头** 以符合 **RFC 2616 §5.1.2** 规范（除非是连接到一个 HTTP 代理服务器或者使用 OPTIONS 或 CONNECT 方法）。

如果 *headers* 既不包含 Content-Length 也没有 Transfer-Encoding，但存在请求正文，那么这些头字段中的一个会自动设定。如果 *body* 是 None，那么对于要求正文的方法 (PUT，POST，和 PATCH)，Content-Length 头会被设为 0。如果 *body* 是字符串或者类似字节的对象，并且也不是文件，Content-Length 头会设为正文的长度。任何其他类型的 *body* （一般是文件或迭代器）会按块编码，这时会自动设定 Transfer-Encoding 头以代替 Content-Length。

在 *headers* 中指定 Transfer-Encoding 时，*encode_chunked* 是唯一相关的参数。如果 *encode_chunked* 为 False，HTTPConnection 对象会假定所有的编码都由调用代码处理。如果为 True，正文会按块编码。

例如，要对 https://docs.python.org/3/ 执行一个 GET 请求：

```
>>> import http.client
>>> host = "docs.python.org"
>>> conn = http.client.HTTPSConnection(host)
>>> conn.request("GET", "/3/", headers={"Host": host})
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200 OK
```

> **备注**
>
> HTTP 协议在 1.1 版中添加了块传输编码。除非明确知道 HTTP 服务器可以处理 HTTP 1.1，调用者要么必须指定 Content-Length，要么必须传入 *str* 或字节类对象，注意该对象不能是表达 body 的文件。

在 3.2 版本发生变更: *body* 现在可以是可迭代对象了。

在 3.6 版本发生变更: 如果 Content-Length 和 Transfer-Encoding 都没有在 *headers* 中设置，文件和可迭代的 *body* 对象现在会按块编码。添加了 *encode_chunked* 参数。不会尝试去确定文件对象的 Content-Length。

`HTTPConnection.`**`getresponse`**`()`

应当在发送一个请求从服务器获取响应时被调用。返回一个 *HTTPResponse* 的实例。

> **备注**
>
> 请注意你必须在读取了整个响应之后才能向服务器发送新的请求。

在 3.5 版本发生变更: 如果引发了 *ConnectionError* 或其子类，*HTTPConnection* 对象将在发送新的请求时准备好重新连接。

`HTTPConnection.`**`set_debuglevel`**`(`*level*`)`

设置调试等级。默认的调试等级为 `0`，意味着不会打印调试输出。任何大于 `0` 的值将使得所有当前定义的调试输出被打印到 stdout。`debuglevel` 会被传给任何新创建的 *HTTPResponse* 对象。

Added in version 3.1.

`HTTPConnection.`**`set_tunnel`**`(`*host*, *port=None*, *headers=None*`)`

为 HTTP 连接隧道设置主机和端口。这将允许通过代理服务器运行连接。

host 和 *port* 参数指明隧道连接的端点（即 CONNECT 请求所包含的地址，而 不是代理服务器的地址）。

headers 参数应为一个随 CONNECT 请求发送的额外 HTTP 标头的映射。

在 HTTP/1.1 被用于 HTTP CONNECT 隧道请求时，根据相应的 RFC，必须提供一个 HTTP `Host:` 标头，以匹配作为 CONNECT 请求的目标提供的请求目标 authority-form。如果未通过 headers 参数提供 HTTP `Host:` 标头，则会自动生成并传送一个标头。

例如，要通过一个运行于本机 8080 端口的 HTTPS 代理服务器隧道，我们应当向 *HTTPSConnection* 构造器传入代理的地址，并将我们最终想要访问的主机地址传给 *set_tunnel()* 方法:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

Added in version 3.2.

在 3.12 版本发生变更: HTTP CONNECT 隧道请求使用 HTTP/1.1 协议，它是从 HTTP/1.0 协议升级而来。`Host:` HTTP 标头是 HTTP/1.1 所必需的，因此如果未在 headers 参数中提供则会自动生成并传送一个标头。

`HTTPConnection.`**`get_proxy_response_headers`**`()`

返回一个由从代理服务器接收的响应标头映射到 CONNECT 请求的字典。

如果未发送 CONNECT 请求，该方法将返回 `None`。

Added in version 3.12.

HTTPConnection.**connect**()

当对象被创建后连接到指定的服务器。默认情况下，如果客户端还未建立连接，此函数会在发送请求时自动被调用。

引发一个审计事件 `http.client.connect` 并附带参数 `self`, `host`, `port`。

HTTPConnection.**close**()

关闭到服务器的连接。

HTTPConnection.**blocksize**

用于发送文件类消息体的缓冲区大小。

Added in version 3.7.

作为对使用上述*request()*方法的替代，你也可以通过使用以下四个函数来一步步地发送你的请求。

HTTPConnection.**putrequest**(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)

应为连接服务器之后首先调用的函数。将向服务器发送一行数据，包含 *method* 字符串、*url* 字符串和 HTTP 版本（`HTTP/1.1`）。若要禁止自动发送 `Host:` 或 `Accept-Encoding:` 头部信息（比如需要接受其他编码格式的内容），请将 *skip_host* 或 *skip_accept_encoding* 设为非 False 值。

HTTPConnection.**putheader**(*header*, *argument*[, ...])

向服务器发送一个 **RFC 822** 格式的头部。将向服务器发送一行由头、冒号和空格以及第一个参数组成的数据。如果还给出了其他参数，将在后续行中发送，每行由一个制表符和一个参数组成。

HTTPConnection.**endheaders**(*message_body=None*, ***, *encode_chunked=False*)

向服务器发送一个空行，表示头部文件结束。可选的 *message_body* 参数可用于传入一个与请求相关的消息体。

如果 *encode_chunked* 为 `True`，则对 *message_body* 的每次迭代结果将依照 **RFC 7230** 3.3.1 节的规范进行分块编码。数据如何编码取决于 *message_body* 的类型。如果 *message_body* 实现了 buffer 接口，编码将生成一个数据块。如果 *message_body* 是*collections.abc.Iterable*，则 *message_body* 的每次迭代都会产生一个块。如果 *message_body* 为*file object*，那么每次调用 `.read()` 都会产生一个数据块。在 *message_body* 结束后，本方法立即会自动标记分块编码数据的结束。

> **备注**
>
> 由于分块编码的规范要求，迭代器本身产生的空块将被分块编码器忽略。这是为了避免目标服务器因错误编码而过早终止对请求的读取。

在 3.6 版本发生变更: 增加了分块编码支持和 *encode_chunked* 形参。

HTTPConnection.**send**(*data*)

发送数据到服务器。本函数只应在调用*endheaders()*方法之后且调用*getresponse()*之前直接调用。

引发一个审计事件 `http.client.send` 并附带参数 `self`, `data`。

22.10.2 HTTPResponse 对象

HTTPResponse 实例封装了来自服务器的 HTTP 响应。通过它可以访问请求头和响应体。响应是可迭代对象，可在 with 语句中使用。

在 3.5 版本发生变更: 现在已实现了*io.BufferedIOBase*接口，并且支持所有的读取操作。

HTTPResponse.**read**([*amt*])

读取并返回响应体，或后续 *amt* 个字节。

HTTPResponse.**readinto**(*b*)

读取响应体的后续 len(b) 个字节到缓冲区 *b*。返回读取的字节数。

Added in version 3.3.

HTTPResponse.**getheader**(*name*, *default=None*)

返回标头 *name* 的值，或者如果没有匹配 *name* 的标头则返回 *default*。如果名为 *name* 的标头不止一个，则返回以', ' 连接的所有值。如果 *default* 是任何不为单个字符串的可迭代对象，则其元素同样会以逗号连接的形式返回。

HTTPResponse.**getheaders**()

返回 (header, value) 元组构成的列表。

HTTPResponse.**fileno**()

返回底层套接字的 `fileno`。

HTTPResponse.**msg**

包含响应头的*http.client.HTTPMessage* 实例。*http.client.HTTPMessage* 是*email.message* 的子类。

HTTPResponse.**version**

服务器采用的 HTTP 协议版本。10 代表 HTTP/1.0，11 代表 HTTP/1.1。

HTTPResponse.**url**

已读取资源的 URL，通常用于确定是否进行了重定向。

HTTPResponse.**headers**

响应的头部信息，形式为*email.message.EmailMessage* 的实例。

HTTPResponse.**status**

由服务器返回的状态码。

HTTPResponse.**reason**

服务器返回的原因短语。

HTTPResponse.**debuglevel**

一个调试钩子。如果*debuglevel* 大于零，状态信息将在读取和解析响应数据时打印输出到 stdout。

HTTPResponse.**closed**

如果流被关闭，则为 `True`。

HTTPResponse.**geturl**()

自 3.9 版本弃用: 已弃用，建议用*url*。

HTTPResponse.**info**()

自 3.9 版本弃用: 已弃用，建议用*headers*。

HTTPResponse.**getcode**()

自 3.9 版本弃用: 已弃用，建议用*status* 。

22.10.3 例子

下面是使用 `GET` 方法的会话示例:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # 这将返回全部内容。
>>> # 下面的例子演示了分块读取数据。
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
```

(续下页)

(接上页)

```
...
>>> # 无效请求的例子
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

以下是使用 HEAD 方法的会话示例。请注意，HEAD 方法从不返回任何数据。

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

下面是一个使用 POST 方法的会话示例:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="https://bugs.python.org/issue12524">https://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

客户端 HTTP PUT 请求与 POST 请求非常相似。区别仅在于服务器端 HTTP 服务器将允许通过 PUT 请求创建资源。应该注意自定义的 HTTP 方法也可以在*urllib.request.Request* 中通过设置适当的方法属性来进行处理。下面是一个使用 PUT 方法的会话示例:

```
>>> # 在这里创建一个 HTTP 请求
>>> # 其中 BODY 的内容为资源 http://localhost:8080/file
>>> # 的附件表示形式
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

22.10.4 HTTPMessage 对象

class http.client.**HTTPMessage**(*email.message.Message*)

http.client.HTTPMessage 的实例存有 HTTP 响应的头部信息。利用*email.message.Message* 类实现。

22.11 ftplib --- FTP 协议客户端

源代码： Lib/ftplib.py

本模块定义了*FTP* 类和一些相关项目。*FTP* 类实现了 FTP 协议的客户端。你可以用这个类来编写执行各种自动化 FTP 任务的 Python 程序，例如镜像其他 FTP 服务器等。它还被*urllib.request* 模块用来处理使用 FTP 的 URL。有关 FTP (文件传输协议) 的更多信息，请参阅 **RFC 959**。

默认编码为 UTF-8，遵循 **RFC 2640**。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

以下是使用*ftplib* 模块的会话示例:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')  # 连接到主机，默认端口
>>> ftp.login()                     # 用户 anonymous，密码 anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # 更改为 "debian" 目录
'250 Directory successfully changed.'
>>> ftp.retrlines('LIST')           # 列出目录内容
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

22.11.1 参考

FTP 对象

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*, *, *encoding='utf-8'*)

返回一个*FTP* 类的新实例。

参数

- **host** (str) -- 要连接的主机名。如果给出，则将由构造器隐式地调用 connect(host)。
- **user** (str) -- 如果给出 The username to log in with (default: 'anonymous'.)，则将由构造器隐式地调用 login(host, passwd, acct)。
- **passwd** (str) -- The password to use when logging in. If not given, and if *passwd* is the empty string or "-", a password will be automatically generated.
- **acct** (str) -- Account information to be used for the ACCT FTP command. Few systems implement this. See RFC-959 for more details.

- **timeout** (*float | None*) -- 用于阻塞操作如*connect()* 的以秒数表示的超时值（默认：全局默认超时设置值）。
- **source_address** (*tuple | None*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (`str`) -- The encoding for directories and filenames (default: `'utf-8'`).

FTP 类支持 `with` 语句，例如：

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

在 3.2 版本发生变更: 添加了对 `with` 语句的支持。

在 3.3 版本发生变更: 添加了 *source_address* 参数。

在 3.9 版本发生变更: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。添加了 *encoding* 参数，且为了遵循 **RFC 2640**，该参数默认值从 Latin-1 改为了 UTF-8。

某些 `FTP` 方法有两种形式：一种用于处理文本文件而另一种用于二进制文件。这些方法的名称与所用的命令相对应，文本版之后跟 `lines` 而二进制版之后跟 `binary`。

FTP 实例具有下列方法:

set_debuglevel(*level*)

将实例的调试级别设为一个*int* 值。这将控制所打印的调试输出数据量。调试级别包括：

- `0` (默认): 无调试输出。
- `1`: 产生中等的调试输出数据量，通常为每个请求一行。
- `2` 或更高: 产生最大的调试输出数据量，记录在控制连接中发送和接收的每一行。

connect(*host=''*, *port=0*, *timeout=None*, *source_address=None*)

连接到给定的主机和端口。此函数应当只为每个实例调用一次；如果在创建*FTP* 实例时给出了 *host* 参数则不应调用此函数。所有其他 `FTP` 方法只能在连接成功建立之后被调用。

参数

- **host** (`str`) -- 要连接的主机。
- **port** (`int`) -- 要连接的 TCP 端口 (默认值: `21`，如 FTP 协议规范所指明的)。很少有必要指定不同的端口号。
- **timeout** (*float | None*) -- 针对连接尝试的以秒数表示的超时值（默认值：全局默认超时设置值）。
- **source_address** (*tuple | None*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

引发一个审计事件 `ftplib.connect` 并附带参数 `self`, `host`, `port`。

在 3.3 版本发生变更: 添加了 *source_address* 参数。

getwelcome()

返回服务器发送的欢迎消息，作为连接开始的回复。（该消息有时包含与用户有关的免责声明或帮助信息。）

login(*user='anonymous'*, *passwd=''*, *acct=''*)

登录到已连接的 FTP 服务器。在建立连接后，此函数应当只为每个实例调用一次；如果在创建*FTP* 实例时给出了 *host* 和 *user* 参数则不应调用该函数。大多数 FTP 命令只有在客户端已登录后才允许使用。

参数

- **user** (str) -- The username to log in with (default: `'anonymous'`).
- **passwd** (str) -- The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (str) -- Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.

abort()

中止正在进行的文件传输。本方法并不总是有效，但值得一试。

sendcmd(*cmd*)

将一条简单的命令字符串发送到服务器，返回响应的字符串。

引发一个审计事件 `ftplib.sendcmd` 并附带参数 `self`, `cmd`。

voidcmd(*cmd*)

将一条简单的命令字符串发送到服务器并对响应进行处理。如果响应代码对应执行成功（代码在 200--299 范围内）则返回响应字符串。在其他情况下则引发*error_reply*。

引发一个审计事件 `ftplib.sendcmd` 并附带参数 `self`, `cmd`。

retrbinary(*cmd*, *callback*, *blocksize=8192*, *rest=None*)

以二进制传输模式获取一个文件。

参数

- **cmd** (str) -- 一个正确的 `RETR` 命令: `"RETR filename"`。
- **callback** (*callable*) -- 一个针对所接收的每个数据块被调用的单形参可调用对象，其唯一参数即*bytes* 类型的数据。
- **blocksize** (int) -- 在为进行实际传输而创建的底层*socket* 对象上读取的块尺寸最大值。这也对应于将要传给 *callback* 的数据尺寸最大值。默认为 `8192`。
- **rest** (int) -- 要发送给服务器的 `REST` 命令。参见*transfercmd()* 方法的 *rest* 形参的文档。

retrlines(*cmd*, *callback=None*)

以在初始化时由 *encoding* 形参指定的编码格式获取文件或目录列表。*cmd* 应为一个适当的 `RETR` 命令 (参见*retrbinary()*) 或是像 `LIST` 或 `NLST` 这样的命令 (通常即字符串 `'LIST'`)。`LIST` 将获取一个包含文件名及文件相关信息的列表。`NLST` 将获取一个文件名的列表。*callback* 函数将针对每一行被调用并附带一个包含去除了末尾 CRLF 的行的字符串参数。默认的 *callback* 会将行内容打印到*sys.stdout*。

set_pasv(*val*)

如果 *val* 为 true，则打开“被动”模式，否则禁用被动模式。默认下被动模式是打开的。

storbinary(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)

以二进制传输模式存储一个文件。

参数

- **cmd** (str) -- 一个正确的 `STOR` 命令: `"STOR filename"`。
- **fp** (*file object*) -- 一个被读取直至 EOF 的文件对象（以二进制模式打开），使用其*read()* 方法以大小为 *blocksize* 的块来提供要存储的数据。
- **blocksize** (int) -- 读取块的大小。默认为 `8192`。

- **callback** (*callable*) -- 一个针对所发送的每个数据块被调用的单形参可调用对象，其唯一参数即*bytes* 类型的数据。
- **rest** (int) -- 要发送给服务器的 REST 命令。参见*transfercmd()* 方法的 *rest* 形参的文档。

在 3.2 版本发生变更: 增加了 *rest* 形参。

storlines(*cmd*, *fp*, *callback=None*)

以文本行模式存储文件。*cmd* 应为恰当的 STOR 命令 (请参阅*storbinary()*)。*fp* 是一个文件对象 (以二进制模式打开)，将使用它的*readline()* 方法读取它的每一行，用于提供要存储的数据，直到遇到 EOF。可选参数 *callback* 是单参数函数，在每行发送后都会以该行作为参数来调用它。

transfercmd(*cmd*, *rest=None*)

在 FTP 数据连接上开始传输数据。如果传输处于活动状态，传输命令由 *cmd* 指定，需发送 EPRT 或 PORT 命令，然后接受连接 (accept)。如果服务器是被动服务器，需发送 EPSV 或 PASV 命令，连接到服务器 (connect)，然后启动传输命令。两种方式都将返回用于连接的套接字。

如果传入了可选参数 *rest*，则一条 REST 命令会被发送到服务器，并以 *rest* 作为参数。*rest* 通常表示请求文件中的字节偏移量，它告诉服务器重新开始发送文件的字节，从请求的偏移量处开始，跳过起始字节。但是请注意，*transfercmd()* 方法会将 *rest* 转换为字符串，但是不检查字符串的内容，转换用的编码是在初始化时指定的 *encoding* 参数。如果服务器无法识别 REST 命令，将引发*error_reply* 异常。如果发生这种情况，只需不带 *rest* 参数调用*transfercmd()*。

ntransfercmd(*cmd*, *rest=None*)

类似于*transfercmd()*，但返回一个元组，包括数据连接和数据的预计大小。如果预计大小无法计算，则返回的预计大小为 None。*cmd* 和 *rest* 的含义与*transfercmd()* 中的相同。

mlsd(*path=''*, *facts=[]*)

使用 MLSD 命令以标准格式列出目录内容 (**RFC 3659**)。如果省略 *path* 则使用当前目录。*facts* 是字符串列表，表示所需的信息类型（如 `["type", "size", "perm"]`）。返回一个生成器对象，每个在 path 中找到的文件都将在该对象中生成两个元素的元组。第一个元素是文件名，第二个元素是该文件的 facts 的字典。该字典的内容受 *facts* 参数限制，但不能保证服务器会返回所有请求的 facts。

Added in version 3.3.

nlst(*argument*[, ...])

返回一个文件名列表，文件名由 NLST 命令返回。可选参数 *argument* 是待列出的目录（默认为当前服务器目录）。可以使用多个参数，将非标准选项传递给 NLST 命令。

> **备注**
>
> 如果目标服务器支持相关命令，那么*mlsd()* 提供的 API 更好。

dir(*argument*[, ...])

生成一个目录列表即 LIST 命令所返回的结果，将其打印到标准输出。可选的 *argument* 是要列出的目录（默认为当前服务器目录）。可以使用多个参数将非标准选项传给 LIST 命令。如果最后一个参数是个函数，它将被用作 *callback* 函数，与*retrlines()* 的类似；默认将打印到*sys.stdout*。此方法将返回 None。

> **备注**
>
> 如果目标服务器支持相关命令，那么*mlsd()* 提供的 API 更好。

rename(*fromname*, *toname*)

将服务器上的文件 *fromname* 重命名为 *toname*。

delete(*filename*)

将服务器上名为 *filename* 的文件删除。如果删除成功，返回响应文本，如果删除失败，在权限错误时引发*error_perm*，在其他错误时引发*error_reply*。

cwd(*pathname*)

设置服务器端的当前目录。

mkd(*pathname*)

在服务器上创建一个新目录。

pwd()

返回服务器上当前目录的路径。

rmd(*dirname*)

将服务器上名为 *dirname* 的目录删除。

size(*filename*)

请求服务器上名为 *filename* 的文件大小。成功后以整数返回文件大小，未成功则返回 None。注意，SIZE 不是标准命令，但通常许多服务器的实现都支持该命令。

quit()

向服务器发送 QUIT 命令并关闭连接。这是关闭一个连接的“礼貌”方式，但是如果服务器对 QUIT 命令的响应带有错误消息则这会引发一个异常。这意味着对*close()* 方法的调用，它将使得*FTP* 实例对后继调用无效（见下文）。

close()

单方面关闭连接。这不该被应用于已经关闭的连接，例如成功调用*quit()* 之后的连接。在此调用之后*FTP* 实例不应被继续使用（在调用*close()* 或*quit()* 之后你不能通过再次唤起*login()* 方法重新打开连接）。

FTP_TLS 对象

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, ***, *context=None*, *timeout=None*, *source_address=None*, *encoding='utf-8'*)

一个为 FTP 添加如 **RFC 4217** 所描述的 TLS 支持的*FTP* 的子类。连接到 21 端口在身份验证之前隐式地确保 FTP 控制连接的安全。

备注

用户必须通过调用*prot_p()* 方法显式地确保数据连接的安全。

参数

- **host** (str) -- 要连接的主机名。如果给出，则将由构造器隐式地调用 connect(host)。
- **user** (str) -- 如果给出 The username to log in with (default: 'anonymous').，则将由构造器隐式地调用 login(host, passwd, acct)。
- **passwd** (str) -- The password to use when logging in. If not given, and if *passwd* is the empty string or "-", a password will be automatically generated.
- **acct** (str) -- Account information to be used for the ACCT FTP command. Few systems implement this. See RFC-959 for more details.
- **context** (*ssl.SSLContext*) -- 一个允许将 SSL 配置选项、证书和私钥打包至一个单独的、可以长久存在的结构体中的 SSL 上下文对象。请参阅安全考量 了解相关的最佳实践。
- **timeout** (float | *None*) -- 一个用于阻塞操作如*connect()* 的以秒数表示的超时值（默认值：全局默认超时设置）。

- **source_address** (*tuple* / *None*) -- A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (*str*) -- The encoding for directories and filenames (default: `'utf-8'`).

Added in version 3.2.

在 3.3 版本发生变更: 增加了 *source_address* 形参。

在 3.4 版本发生变更: 该类现在支持使用*ssl.SSLContext.check_hostname* 和 服务器名称提示 (参见*ssl.HAS_SNI*) 进行主机名检测。

在 3.9 版本发生变更: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。添加了 *encoding* 参数，且为了遵循 **RFC 2640**，该参数默认值从 Latin-1 改为了 UTF-8。

在 3.12 版本发生变更: 已弃用的 *keyfile* 和 *certfile* 形参已被移除。

以下是使用*FTP_TLS* 类的会话示例:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi',
↪'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore', 'libpuzzle
↪', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables', 'php-jenkins-
↪hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench', 'pincaster', 'ping',
↪'posto', 'pub', 'public', 'public_keys', 'pure-ftpd', 'qscan', 'qtc', 'sharedance',
↪'skycache', 'sound', 'tmp', 'ucarp']
```

FTP_TLS 类继承自*FTP*，它定义了下列额外方法和属性:

ssl_version
要使用的 SSL 版本 (默认为*ssl.PROTOCOL_SSLv23*)。

auth()
通过使用 TLS 或 SSL 来设置一个安全控制连接，具体取决于*ssl_version* 属性是如何设置的。

在 3.4 版本发生变更: 该方法现在支持使用*ssl.SSLContext.check_hostname* 和 服务器名称提示 (参见*ssl.HAS_SNI*) 进行主机名称检测。

ccc()
将控制通道回复为纯文本。这适用于发挥知道如何使用非安全 FTP 处理 NAT 而无需打开固定端口的防火墙的优势。

Added in version 3.3.

prot_p()
设置加密数据连接。

prot_c()
设置明文数据连接。

模块变量

exception ftplib.**error_reply**
从服务器收到意外答复时，将引发本异常。

exception ftplib.**error_temp**
收到表示临时错误的错误代码（响应代码在 400--499 范围内）时，将引发本异常。

exception ftplib.**error_perm**
收到表示永久性错误的错误代码（响应代码在 500--599 范围内）时，将引发本异常。

exception ftplib.**error_proto**

从服务器收到不符合 FTP 响应规范的答复，比如以数字 1--5 开头时，将引发本异常。

ftplib.**all_errors**

所有异常的集合（一个元组），由于 FTP 连接出现问题（并非调用者的编码错误），*FTP* 实例的方法可能会引发这些异常。该集合包括上面列出的四个异常以及*OSError* 和*EOFError*。

参见

***netrc* 模块**

.netrc 文件格式解析器。FTP 客户端在响应用户之前，通常使用 .netrc 文件来加载用户认证信息。

22.12 poplib --- POP3 协议客户端

源代码： Lib/poplib.py

本模块定义了一个*POP3* 类，该类封装了到 POP3 服务器的连接过程，并实现了 **RFC 1939** 中定义的协议。*POP3* 类同时支持 **RFC 1939** 中最小的和可选的命令集。*POP3* 类还支持在 **RFC 2595** 中引入的 STLS 命令，用于在已建立的连接上启用加密通信。

本模块额外提供一个*POP3_SSL* 类，在连接到 POP3 服务器时，该类为使用 SSL 作为底层协议层提供了支持。

注意，尽管 POP3 具有广泛的支持，但它已经过时。POP3 服务器的实现质量差异很大，而且大多很糟糕。如果邮件服务器支持 IMAP，则最好使用*imaplib.IMAP4* 类，因为 IMAP 服务器一般实现得更好。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

poplib 模块提供了两个类：

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])

本类实现实际的 POP3 协议。实例初始化时，连接就会建立。如果省略 *port*，则使用标准 POP3 端口（110）。可选参数 *timeout* 指定连接尝试的超时时间（以秒为单位，如果未指定超时，将使用全局默认超时设置）。

引发一个审计事件 poplib.connect 并附带参数 self, host, port。

所有命令都会引发一个审计事件 poplib.putline，附带参数 self 和 line，其中 line 是即将发送到远程主机的字节串。

在 3.9 版本发生变更: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *, *timeout=None*, *context=None*)

一个*POP3* 的子类，它使用经 SSL 加密的套接字连接到服务器。如果端口 *port* 未指定，则使用 995，它是标准的 POP3-over-SSL 端口。*timeout* 的作用与*POP3* 构造函数中的相同。*context* 是一个可选的*ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读安全考量 以获取最佳实践。

引发一个审计事件 poplib.connect 并附带参数 self, host, port。

所有命令都会引发一个审计事件 poplib.putline，附带参数 self 和 line，其中 line 是即将发送到远程主机的字节串。

在 3.2 版本发生变更: 添加了 *context* 参数。

在 3.4 版本发生变更: 该类现在支持使用*ssl.SSLContext.check_hostname* 和 服务器名称提示 (参见*ssl.HAS_SNI*) 进行主机名检测。

在 3.9 版本发生变更: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。

在 3.12 版本发生变更: 已弃用的 *keyfile* 和 *certfile* 形参已被移除。

定义了一个异常，它是作为*poplib* 模块的属性定义的：

exception poplib.**error_proto**

此模块的所有错误都将引发本异常（来自*socket* 模块的错误不会被捕获）。异常的原因将以字符串的形式传递给构造函数。

参见

***imaplib* 模块**
标准的 Python IMAP 模块。

有关 Fetchmail 的常见问题
fetchmail POP/IMAP 客户端的“常见问题”收集了 POP3 服务器之间的差异和 RFC 不兼容的信息，如果要编写基于 POP 协议的应用程序，这可能会很有用。

22.12.1 POP3 对象

所有 POP3 命令均以同名的方法表示，使用小写形式；大多数方法返回的是服务器所发送的响应文本。

POP3 实例具有下列方法:

POP3.**set_debuglevel**(*level*)

设置实例的调试级别，它控制着调试信息的数量。默认值 `0` 不产生调试信息。值 `1` 产生中等数量的调试信息，通常每个请求产生一行。大于或等于 `2` 的值产生的调试信息最多，FTP 控制连接上发送和接收的每一行都将被记录下来。

POP3.**getwelcome**()

返回 POP3 服务器发送的问候语字符串。

POP3.**capa**()

查询服务器支持的功能，这些功能在 **RFC 2449** 中有说明。返回一个 `{'name': ['param'...]}` 形式的字典。

Added in version 3.4.

POP3.**user**(*username*)

发送 user 命令，返回的响应应该指示需要一个密码。

POP3.**pass_**(*password*)

发送密码，响应包括邮件消息计数和邮箱大小。注意：服务器上的邮箱将被锁定直到*quit()* 被调用。

POP3.**apop**(*user*, *secret*)

使用更安全的 APOP 身份验证登录到 POP3 服务器。

POP3.**rpop**(*user*)

使用 RPOP 身份验证（类似于 Unix r-命令）登录到 POP3 服务器。

POP3.**stat**()

获取邮箱状态。结果为 2 个整数组成的元组：`(message count, mailbox size)`。

POP3.**list**([*which*])

请求消息列表，结果的形式为 `(response, ['mesg_num octets', ...], octets)`。如果设置了 *which*，它表示需要列出的消息。

POP3.**retr**(*which*)

检索编号为 *which* 的整条消息，并设置其已读标志位。结果的形式为 `(response, ['line', ...], octets)`。

POP3.**dele**(*which*)

将编号为 *which* 的消息标记为待删除。在多数服务器上，删除操作直到 QUIT 才会实际执行（主要例外是 Eudora QPOP，它在断开连接时执行删除，故意违反了 RFC）。

POP3.**rset**()

移除邮箱中的所有待删除标记。

POP3.**noop**()

什么都不做。可以用于保持活动状态。

POP3.**quit**()

登出：提交更改，解除邮箱锁定，断开连接。

POP3.**top**(*which*, *howmuch*)

检索编号为 *which* 的消息，范围是消息头加上消息头往后数 *howmuch* 行。结果的形式为 `(response, ['line', ...], octets)`。

本方法使用 POP3 TOP 命令，不同于 RETR 命令，它不设置邮件的已读标志位。不幸的是，TOP 在 RFC 中说明不清晰，且在小众的服务器软件中往往不可用。信任并使用它之前，请先手动对目标 POP3 服务器测试本方法。

POP3.**uidl**(*which=None*)

返回消息摘要（唯一 ID）列表。如果指定了 *which*，那么结果将包含那条消息的唯一 ID，形式为 `'response mesgnum uid`，如果未指定，那么结果为列表 `(response, ['mesgnum uid', ...], octets)`。

POP3.**utf8**()

尝试切换至 UTF-8 模式。成功则返回服务器的响应，失败则引发 *error_proto* 异常。在 **RFC 6856** 中有说明。

Added in version 3.5.

POP3.**stls**(*context=None*)

在活动连接上开启 TLS 会话，在 **RFC 2595** 中有说明。仅在用户身份验证前允许这样做。

context 参数是一个 *ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读 *安全考量* 以获取最佳实践。

此方法支持通过 *ssl.SSLContext.check_hostname* 和 服务器名称指示 (参见 *ssl.HAS_SNI*) 进行主机名检测。

Added in version 3.4.

POP3_SSL 实例没有额外方法。该子类的接口与其父类的相同。

22.12.2 POP3 示例

以下是一个最短示例（不带错误检查），该示例将打开邮箱，检索并打印所有消息:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

模块的最后有一段测试，其中包含的用法示例更加广泛。

22.13 imaplib --- IMAP4 协议客户端

源代码： Lib/imaplib.py

本模块定义了三个类：*IMAP4* 、*IMAP4_SSL* 和*IMAP4_stream* 。这三个类封装了与 IMAP4 服务器的连接并实现了 **RFC 2060** 当中定义的大多数 IMAP4rev1 客户端协议。其与 IMAP4（**RFC 1730** ）服务器后向兼容，但是 STATUS 指令在 IMAP4 中不支持。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

imaplib 模块提供了三个类，其中*IMAP4* 是基类：

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*, *timeout=None*)

这个类实现了实际的 IMAP4 协议。当其实例被初始化时会创建连接并确定协议版本 (IMAP4 或 IMAP4rev1)。如果未指明 *host*，则会使用 '' (本地主机)。如果省略 *port*，则会使用标准的 IMAP4 端口 (143)。可选的 *timeout* 形参指定连接尝试的超时秒数。如果未指定超时或为 None，则会使用全局默认的套接字超时。

IMAP4 类支持 with 语句。当这样使用时，IMAP4 LOGOUT 命令会在 with 语句退出时自动发出。例如:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

在 3.5 版本发生变更: 添加了对 with 语句的支持。

在 3.9 版本发生变更: 添加了可选的 *timeout* 形参。

有三个异常被定义为*IMAP4* 类的属性:

exception IMAP4.**error**

任何错误都将引发该异常。异常的原因会以字符串的形式传递给构造器。

exception IMAP4.**abort**

IMAP4 服务器错误会导致引发该异常。这是*IMAP4.error* 的子类。请注意关闭此实例并实例化一个新实例通常将会允许从该异常中恢复。

exception IMAP4.**readonly**

当一个可写邮箱的状态被服务器修改时会引发此异常。此异常是*IMAP4.error* 的子类。某个其他客户端现在会具有写入权限，将需要重新打开该邮箱以重新获得写入权限。

另外还有一个针对安全连接的子类:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, ***, *ssl_context=None*, *timeout=None*)

这是一个派生自*IMAP4* 的子类，它使用经 SSL 加密的套接字进行连接 (为了使用这个类你需要编译时附带 SSL 支持的 socket 模块)。如果未指定 *host*，则会使用 '' (本地主机)。如果省略了 *port*，则会使用标准的 IMAP4-over-SSL 端口 (993)。*ssl_context* 是一个*ssl.SSLContext* 对象，它允许将 SSL 配置选项、证书和私钥打包放入一个单独的 (可以长久存在的) 结构体中。请阅读*安全考量* 以获取最佳实践。

可选的 *timeout* 形参指明连接尝试的超时秒数。如果超时值未给出或为 None，则会使用全局默认的套接字超时设置。

在 3.3 版本发生变更: 增加了 *ssl_context* 形参。

在 3.4 版本发生变更: 该类现在支持使用*ssl.SSLContext.check_hostname* 和 *服务器名称提示* (参见*ssl.HAS_SNI*) 进行主机名检测。

在 3.9 版本发生变更: 添加了可选的 *timeout* 形参。

在 3.12 版本发生变更: 已弃用的 *keyfile* 和 *certfile* 形参已被移除。

第二个子类允许由子进程所创建的连接:

class imaplib.**IMAP4_stream**(*command*)

这是一个派生自*IMAP4* 的子类，它可以连接 stdin/stdout 文件描述符，此种文件是通过向 subprocess.Popen() 传入 *command* 来创建的。

定义了下列工具函数:

imaplib.**Internaldate2tuple**(*datestr*)

解析一个 IMAP4 INTERNALDATE 字符串并返回对应的本地时间。返回值是一个*time.struct_time* 元组或者如果字符串格式错误则为 None。

imaplib.**Int2AP**(*num*)

将一个整数转换为使用字符集 [A .. P] 的字节串表示形式。

imaplib.**ParseFlags**(*flagstr*)

将一个 IMAP4 FLAGS 响应转换为包含单独旗标的元组。

imaplib.**Time2Internaldate**(*date_time*)

将 *date_time* 转换为 IMAP4 INTERNALDATE 表示形式。返回值是以下形式的字符串: "DD-Mmm-YYYY HH:MM:SS +HHMM" (包括双引号)。*date_time* 参数可以是一个代表距离纪元起始的秒数 (如*time.time()* 的返回值) 的数字 (整数或浮点数)，一个代表本地时间的 9 元组，一个*time.struct_time* 实例 (如*time.localtime()* 的返回值)，一个感知型的*datetime.datetime* 实例，或一个双引号字符串。在最后一种情况下，它会被假定已经具有正确的格式。

请注意 IMAP4 消息编号会随邮箱的改变而改变；特别是在使用 EXPUNGE 命令执行删除后剩余的消息会被重新编号。因此高度建议通过 UID 命令来改用 UID。

模块的最后有一段测试，其中包含的用法示例更加广泛。

参见

描述该协议的文档，实现该协议的服务器源代码，由华盛顿大学 IMAP 信息中心提供 (**源代码**) https://github.com/uw-imap/imap (**不再维护**)。

22.13.1 IMAP4 对象

所有 IMAP4rev1 命令都是以相同名称的方法来表示的，可以为大写或小写形式。

命令的所有参数都会被转换为字符串，只有 AUTHENTICATE 例外，而 APPEND 的最后一个参数会被作为 IMAP4 字面值传入。如有必要 (字符串包含 IMAP4 协议中的敏感字符并且未加圆括号或双引号) 每个字符串都会被转码。但是，LOGIN 命令的 *password* 参数总是会被转码。如果你想让某个参数字符串免于被转码 (例如: STORE 的 *flags* 参数) 则要为该字符串加上圆括号 (例如: r'(\Deleted)')。

每条命令均返回一个元组: (type, [data, ...]) 其中 *type* 通常为 'OK' 或 'NO'，而 *data* 为来自命令响应的文本，或为来自命令的规定结果。每个 *data* 均为 bytes 或者元组。如果为元组，则其第一部分是响应的标头，而第二部分将包含数据 (例如: 'literal' 值)。

以下命令的 *message_set* 选项为指定要操作的一条或多条消息的字符串。它可以是一个简单的消息编号 ('1')，一段消息编号区间 ('2:4')，或者一组以逗号分隔的非连续区间 ('1:3,6:9')。区间可以包含一个星号来表示无限的上界 ('3:*')。

IMAP4 实例具有下列方法:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)

将 *message* 添加到指定的邮箱。

IMAP4.**authenticate**(*mechanism*, *authobject*)

认证命令 --- 要求对响应进行处理。

mechanism 指明要使用哪种认证机制——它应当在实例变量 `capabilities` 中以 `AUTH=mechanism` 的形式出现。

authobject 必须是一个可调用对象:

```
data = authobject(response)
```

它将被调用以便处理服务器连续响应；传给它的 *response* 参数将为 `bytes` 类型。它应当返回 base64 编码的 `bytes` 数据并发送给服务器。或者在客户端中止响应时返回 `None` 并应改为发送 `*`。

在 3.5 版本发生变更: 字符串形式的用户名和密码现在会被执行 `utf-8` 编码而不限于 ASCII 字符。

`IMAP4.`**`check`**`()`

为服务器上的邮箱设置检查点。

`IMAP4.`**`close`**`()`

关闭当前选定的邮箱。已删除的消息会从可写邮箱中被移除。在 `LOGOUT` 之前建议执行此命令。

`IMAP4.`**`copy`**`(`*message_set*, *new_mailbox*`)`

将 *message_set* 消息拷贝到 *new_mailbox* 的末尾。

`IMAP4.`**`create`**`(`*mailbox*`)`

新建名为 *mailbox* 新邮箱。

`IMAP4.`**`delete`**`(`*mailbox*`)`

删除名为 *mailbox* 的旧邮箱。

`IMAP4.`**`deleteacl`**`(`*mailbox*, *who*`)`

删除邮箱上某人的 ACL (移除任何权限)。

`IMAP4.`**`enable`**`(`*capability*`)`

启用 *capability* (参见 **RFC 5161**)。大多数功能都不需要被启用。目前只有 `UTF8=ACCEPT` 功能受到支持 (参见 **RFC 6855**)。

Added in version 3.5: *`enable()`* 方法本身，以及 **RFC 6855** 支持。

`IMAP4.`**`expunge`**`()`

从选定的邮箱中永久移除被删除的条目。为每条被删除的消息各生成一个 `EXPUNGE` 响应。返回包含按接收时间排序的 `EXPUNGE` 消息编号的列表。

`IMAP4.`**`fetch`**`(`*message_set*, *message_parts*`)`

获取消息（的各个部分）。*message_parts* 应为加圆标号的消息部分名称字符串，例如: `"(UID BODY[TEXT])"`。返回的数据是由消息部分封包和数据组成的元组。

`IMAP4.`**`getacl`**`(`*mailbox*`)`

获取 *mailbox* 的 `ACL`。此方法是非标准的，但是被 `Cyrus` 服务器所支持。

`IMAP4.`**`getannotation`**`(`*mailbox*, *entry*, *attribute*`)`

提取 *mailbox* 的特定 `ANNOTATION`。此方法是非标准的，但是被 `Cyrus` 服务器所支持。

`IMAP4.`**`getquota`**`(`*root*`)`

获取 `quota` *root* 的资源使用和限制。此方法是 rfc2087 定义的 IMAP4 QUOTA 扩展的组成部分。

`IMAP4.`**`getquotaroot`**`(`*mailbox*`)`

获取指定 *mailbox* 的 `quota roots` 列表。此方法是 rfc2087 定义的 IMAP4 QUOTA 扩展的组成部分。

`IMAP4.`**`list`**`(`[*directory*[, *pattern*]]`)`

列出 *directory* 中与 *pattern* 相匹配的邮箱名称。*directory* 默认为最高层级的电邮文件夹，而 *pattern* 默认为匹配任何文本。返回的数据包含 `LIST` 响应列表。

`IMAP4.`**`login`**`(`*user*, *password*`)`

使用纯文本密码标识客户。*password* 将被转码。

IMAP4.**login_cram_md5**(*user*, *password*)

在标识用户以保护密码时强制使用 CRAM-MD5 认证。将只在服务器 CAPABILITY 响应包含 AUTH=CRAM-MD5 阶段时才有效。

IMAP4.**logout**()

关闭对服务器的连接。返回服务器 BYE 响应。

在 3.8 版本发生变更: 此方法不会再忽略静默的任意异常。

IMAP4.**lsub**(*directory=""*, *pattern='*'*)

列出 directory 中抽取的与 pattern 相匹配的邮箱。*directory* 默认为最高层级目录而 *pattern* 默认为匹配任何邮箱。返回的数据为消息部分封包和数据的元组。

IMAP4.**myrights**(*mailbox*)

显示某个邮箱的本人 ACL (即本人在邮箱中的权限)。

IMAP4.**namespace**()

返回 **RFC 2342** 中定义的 IMAP 命名空间。

IMAP4.**noop**()

将 NOOP 发送给服务器。

IMAP4.**open**(*host*, *port*, *timeout=None*)

打开连接到 *host* 上 *port* 的套接字。可选的 *timeout* 形参指定连接尝试的超时秒数。如果超时值未给出或为 None，则会使用全局默认的套接字超时。另外请注意如果 *timeout* 形参被设为零，它将引发*ValueError* 以拒绝创建非阻塞套接字。此方法会由*IMAP4* 构造器隐式地调用。此方法所建立的连接对象将在*IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()* 和*IMAP4.shutdown()* 等方法中被使用。你可以重写此方法。

引发一个审计事件 imaplib.open 并附带参数 self, host, port。

在 3.9 版本发生变更: 加入 *timeout* 参数。

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)

获取消息被截断的部分。返回的数据是由消息部分封包和数据组成的元组。

IMAP4.**proxyauth**(*user*)

作为 *user* 进行认证。允许经权限的管理员通过代理进入任意用户的邮箱。

IMAP4.**read**(*size*)

从远程服务器读取 *size* 字节。你可以重写此方法。

IMAP4.**readline**()

从远程服务器读取一行。你可以重写此方法。

IMAP4.**recent**()

提示服务器进行更新。如果没有新消息则返回的数据为 None，否则为 RECENT 响应的值。

IMAP4.**rename**(*oldmailbox*, *newmailbox*)

将名为 *oldmailbox* 的邮箱重命名为 *newmailbox*。

IMAP4.**response**(*code*)

如果收到响应 *code* 则返回其数据，否则返回 None。返回给定的代码，而不是普通的类型。

IMAP4.**search**(*charset*, *criterion*[, ...])

在邮箱中搜索匹配的消息。*charset* 可以为 None，在这种情况下在发给服务器的请求中将不指定 CHARSET。IMAP 协议要求至少指定一个标准；当服务器返回错误时将会引发异常。*charset* 为 None 对应使用*enable()* 命令启用了 UTF8=ACCEPT 功能的情况。

示例:

```
# M 是一个已连接的 IMAP4 实例...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# 或者:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)

选择一个邮箱。返回的数据是 *mailbox* 中消息的数量 (EXISTS 响应)。默认的 *mailbox* 为 'INBOX'。如果设置了 *readonly* 旗标，则不允许修改该邮箱。

IMAP4.**send**(*data*)

将 data 发送给远程服务器。你可以重写此方法。

引发一个审计事件 imaplib.send 并附带参数 self, data。

IMAP4.**setacl**(*mailbox*, *who*, *what*)

发送 *mailbox* 的 ACL。此方法是非标准的，但是被 Cyrus 服务器所支持。

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])

设置 *mailbox* 的 ANNOTATION。此方法是非标准的，但是被 Cyrus 服务器所支持。

IMAP4.**setquota**(*root*, *limits*)

设置 quota *root* 的资源限制为 *limits*。此方法是 rfc2087 定义的 IMAP4 QUOTA 扩展的组成部分。

IMAP4.**shutdown**()

关闭在 open 中建立的连接。此方法会由 *IMAP4.logout()* 隐式地调用。你可以重写此方法。

IMAP4.**socket**()

返回用于连接服务器的套接字实例。

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])

sort 命令是 search 的变化形式，带有结果排序语句。返回的数据包含以空格分隔的匹配消息编号列表。

sort 命令在 *search_criterion* 参数之前还有两个参数；一个带圆括号的 *sort_criteria* 列表，和搜索的 *charset*。请注意不同于 search，搜索的 *charset* 参数是强制性的。还有一个 uid sort 命令与 sort 对应，如同 uid search 与 search 对应一样。sort 命令首先在邮箱中搜索匹配给定搜索条件的消息，使用 charset 参数来解读搜索条件中的字符串。然后它将返回所匹配消息的编号。

这是一个 IMAP4rev1 扩展命令。

IMAP4.**starttls**(*ssl_context=None*)

发送一个 STARTTLS 命令。*ssl_context* 参数是可选的并且应为一个 *ssl.SSLContext* 对象。这将在 IMAP 连接上启用加密。请阅读安全考量 来了解最佳实践。

Added in version 3.2.

在 3.4 版本发生变更: 该方法现在支持使用 *ssl.SSLContext.check_hostname* 和 服务器名称提示 (参见 *ssl.HAS_SNI*) 进行主机名称检测。

IMAP4.**status**(*mailbox*, *names*)

针对 *mailbox* 请求指定的状态条件。

IMAP4.**store**(*message_set*, *command*, *flag_list*)

改变邮箱中消息的旗标处理。*command* 由 **RFC 2060** 的 6.4.6 小节指明，应为"FLAGS", "+FLAGS" 或"-FLAGS" 之一，并可选择附带".SILENT" 后缀。

例如，要在所有消息上设置删除旗标:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

> **备注**
>
> 创建包含']' (例如: "[test]") 的旗标并违反 RFC 3501 (即 IMAP 协议)。不过，imaplib 在历史上曾经允许创建这样的标签，并且流行的 IMAP 服务器，如 Gmail，都接受并会产生这样的旗标。有些非 Python 程序也会创建这样的旗标。虽然它违反 RFC 并且 IMAP 客户端和服务器应当严格规范，但是 imaplib 出于向下兼容的考虑仍然继承允许创建这样的标签，并且像在 Python 3.6 中一样，会在当其被服务器所发送时处理它们，因为这能提升在真实世界中的兼容性。

IMAP4.**subscribe**(*mailbox*)

订阅新邮箱。

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])

`thread` 命令是 `search` 的变化形式，带有针对结果的消息串句法。返回的数据包含以空格分隔的消息串成员列表。

消息串成员由零个或多个消息编号组成，以空格分隔，标示了连续的上下级关系。

Thread 命令在 *search_criterion* 参数之前还有两个参数；一个 *threading_algorithm*，以及搜索使用的 *charset*。请注意不同于 `search`，搜索使用的 *charset* 参数是强制性的。还有一个 `uid thread` 命令与 `thread` 对应，如同 `uid search` 与 `search` 对应一样。`thread` 命令首先在邮箱中搜索匹配给定搜索条件的消息，使用 *charset* 参数来解读搜索条件中的字符串。然后它将按照指定的消息串算法返回所匹配的消息串。

这是一个 `IMAP4rev1` 扩展命令。

IMAP4.**uid**(*command*, *arg*[, ...])

执行 command arg 并附带用 UID 所标识的消息，而不是用消息编号。返回与命令对应的响应。必须至少提供一个参数；如果不提供任何参数，服务器将返回错误并引发异常。

IMAP4.**unsubscribe**(*mailbox*)

取消订阅原有邮箱。

IMAP4.**unselect**()

imaplib.IMAP4.unselect() 会释放关联到选定邮箱的服务器资源并将服务器返回到已认证状态。此命令会执行与*imaplib.IMAP4.close()* 相同的动作，区别在于它不会从当前选定邮箱中永久性地移除消息。

Added in version 3.9.

IMAP4.**xatom**(*name*[, ...])

允许服务器在 `CAPABILITY` 响应中通知简单的扩展命令。

在*IMAP4* 的实例上定义了下列属性:

IMAP4.**PROTOCOL_VERSION**

在服务器的 `CAPABILITY` 响应中最新的受支持协议。

IMAP4.**debug**

控制调试输出的整数值。初始值会从模块变量 `Debug` 中获取。大于三的值表示将追踪每一条命令。

IMAP4.**utf8_enabled**

通常为 `False` 的布尔值，但也可以被设为 `True`，如果成功地为 `UTF8=ACCEPT` 功能发送了*enable()* 命令的话。

Added in version 3.5.

22.13.2 IMAP4 示例

以下是一个最短示例（不带错误检查），该示例将打开邮箱，检索并打印所有消息:

```
import getpass, imaplib

M = imaplib.IMAP4(host='example.org')
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

22.14 smtplib --- SMTP 协议客户端

源代码： Lib/smtplib.py

---

smtplib 模块定义了一个 SMTP 客户端会话对象，该对象可将邮件发送到互联网上任何带有 SMTP 或 ESMTP 监听程序的计算机。关于 SMTP 和 ESMTP 操作的更多细节请参阅 **RFC 821** (简单邮件传输协议) 和 **RFC 1869** (SMTP 服务扩展)。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅 *WebAssembly 平台* 了解详情。

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*, [*timeout*,]*source_address=None*)

SMTP 实例是对 SMTP 连接的封装。它提供了支持全部 SMTP 和 ESMTP 操作的方法。如果给出了可选的 *host* 和 *port* 形参，则会在初始化期间调用 SMTP *connect()* 方法并附带这些形参。如果指定了 *local_hostname*，它将在 HELO/EHLO 命令中被用作本地主机的 FQDN。在其他情况下，会使用 *socket.getfqdn()* 来找到本地主机名。如果 *connect()* 调用返回了表示成功代码以外的任何信息，则会引发 *SMTPConnectError*。可选的 *timeout* 形参指定了阻塞操作如连接尝试的超时秒数（如果未指定，则将使用全局默认超时设置）。如果达到超时限制，将会引发 *TimeoutError*。可选的 *source_address* 形参允许在有多张网卡的计算机中绑定到某些特定的源地址，和/或绑定到某个特定的源 TCP 端口。它接受一个 2 元组 `(host, port)` 作为在连接之前要绑定为其源地址的套接字。如果省略 (或者如果 *host* 或 *port* 分别为 `''` 和/或 `0`) 则将使用 OS 的默认行为。

正常使用时，只需要初始化或 connect 方法，*sendmail()* 方法，再加上 *SMTP.quit()* 方法即可。下文包括了一个示例。

SMTP 类支持 `with` 语句。当这样使用时，`with` 语句一退出就会自动发出 SMTP `QUIT` 命令。例如:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

所有命令都会引发一个审计事件 `smtplib.SMTP.send`，附带参数 `self` 和 `data`，其中 `data` 是即将发送到远程主机的字节串。

在 3.3 版本发生变更: 添加了对 `with` 语句的支持。

在 3.3 版本发生变更: 添加了 *source_address* 参数。

Added in version 3.5: 现在已支持 SMTPUTF8 扩展 (**RFC 6531**)。

在 3.9 版本发生变更: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发 *ValueError* 来阻止该操作。

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *, [*timeout*,]*context=None*, *source_address=None*)

SMTP_SSL 实例与*SMTP* 实例的行为完全相同。在开始连接就需要 SSL，且 `starttls()` 不适合的情况下，应该使用*SMTP_SSL*。如果未指定 *host*，则使用 localhost。如果 *port* 为 0，则使用标准 SMTP-over-SSL 端口（465）。可选参数 *local_hostname*、*timeout* 和 *source_address* 的含义与*SMTP* 类中的相同。可选参数 *context* 是一个*SSLContext* 对象，可以从多个方面配置安全连接。请阅读安全考量 以获取最佳实践。

在 3.3 版本发生变更: 增加了 *context*。

在 3.3 版本发生变更: 添加了 *source_address* 参数。

在 3.4 版本发生变更: 该类现在支持使用*ssl.SSLContext.check_hostname* 和 服务器名称提示 (参见*ssl.HAS_SNI*) 进行主机名检测。

在 3.9 版本发生变更: 如果 *timeout* 形参被设为零，则它将引发*ValueError* 来阻止创建非阻塞的套接字

在 3.12 版本发生变更: 已弃用的 *keyfile* 和 *certfile* 形参已被移除。

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

LMTP 协议与 ESMTP 非常相似，它很大程度上基于标准 SMTP 客户端。将 Unix 套接字用于 LMTP 是很常见的，因此 `connect()` 方法必须支持它以及常规的 host:port 服务器。可选参数 *local_hostname* 和 *source_address* 的含义与*SMTP* 类中的相同。要指定 Unix 套接字，你必须使用绝对路径作为 *host*，即以'/' 开头。

支持使用常规的 SMTP 机制来进行认证。当使用 Unix 套接字时，LMTP 通常不支持或要求任何认证，但你的情况可能会有所不同。

在 3.9 版本发生变更: 添加了可选的 *timeout* 形参。

同样地定义了一组精心选择的异常:

exception smtplib.**SMTPException**

OSError 的子类，它是本模块提供的所有其他异常的基类。

在 3.4 版本发生变更: SMTPException 已成为*OSError* 的子类

exception smtplib.**SMTPServerDisconnected**

当服务器意外断开连接，或在*SMTP* 实例连接到服务器之前尝试使用它时将引发此异常。

exception smtplib.**SMTPResponseException**

包括 SMTP 错误代码的所有异常的基类。这些异常会在 SMTP 服务器返回错误代码时在实例中生成。错误代码存放在错误的 `smtp_code` 属性中，并且 `smtp_error` 属性会被设为错误消息。

exception smtplib.**SMTPSenderRefused**

发送方地址被拒绝。除了在所有*SMTPResponseException* 异常上设置的属性，还会将'sender' 设为代表拒绝方 SMTP 服务器的字符串。

exception smtplib.**SMTPRecipientsRefused**

所有接收方地址被拒绝。每个接收方的错误可通过属性 `recipients` 来访问，该属性是一个字典，其元素顺序与*SMTP.sendmail()* 所返回的一致。

exception smtplib.**SMTPDataError**

SMTP 服务器拒绝接收消息数据。

exception smtplib.**SMTPConnectError**

在建立与服务器的连接期间发生了错误。

exception smtplib.**SMTPHeloError**

服务器拒绝了我们的 `HELO` 消息。

exception smtplib.**SMTPNotSupportedError**

尝试的命令或选项不被服务器所支持。

Added in version 3.5.

exception smtplib.**SMTPAuthenticationError**

SMTP 认证出现问题。最大的可能是服务器不接受所提供的用户名/密码组合。

参见

RFC 821 - 简单邮件传输协议
SMTP 的协议定义。该文件涵盖了 SMTP 的模型、操作程序和协议细节。

RFC 1869 - SMTP 服务扩展
定义了 SMTP 的 ESMTP 扩展。这描述了一个用新命令扩展 SMTP 的框架，支持动态发现服务器所提供的命令，并定义了一些额外的命令。

22.14.1 SMTP 对象

一个*SMTP* 实例拥有以下方法：

SMTP.**set_debuglevel**(*level*)

设置调试输出级别。如果 *level* 的值为 1 或 True，就会产生连接的调试信息，以及所有发送和接收服务器的信息。如果 *level* 的值为 2，则这些信息会被加上时间戳。

在 3.5 版本发生变更: 添调试级别 2 。

SMTP.**docmd**(*cmd*, *args=''*)

向服务器发送一条命令 *cmd* 。可选的参数 *args* 被简单地串联到命令中，用一个空格隔开。

这将返回一个由数字响应代码和实际响应行组成的 2 元组（多行响应被连接成一个长行）。

在正常操作中，应该没有必要明确地调用这个方法。它被用来实现其他方法，对于测试私有扩展可能很有用。

如果在等待回复的过程中，与服务器的连接丢失，*SMTPServerDisconnected* 将被触发。

SMTP.**connect**(*host='localhost'*, *port=0*)

连接到某个主机的某个端口。默认是连接到 localhost 的标准 SMTP 端口（25）上。如果主机名以冒号 (':') 结尾，后跟数字，则该后缀将被删除，且数字将视作要使用的端口号。如果在实例化时指定了 host，则构造函数会自动调用本方法。返回包含响应码和响应消息的 2 元组，它们由服务器在其连接响应中发送。

引发一个审计事件 smtplib.connect 并附带参数 self, host, port。

SMTP.**helo**(*name=''*)

使用 HELO 向 SMTP 服务器表明自己的身份。hostname 参数默认为本地主机的完全合格域名。服务器返回的消息被存储为对象的 helo_resp 属性。

在正常操作中，应该没有必要明确调用这个方法。它将在必要时被*sendmail()* 隐式调用。

SMTP.**ehlo**(*name=''*)

使用 EHLO 向 ESMTP 服务器表明自己的身份。hostname 参数默认为本地主机的完全合格域名。检查 ESMTP 选项的响应，并存储它们供*has_extn()* 使用。同时设置几个信息属性：服务器返回的消息被存储为 ehlo_resp 属性，does_esmtp 根据服务器是否支持 ESMTP 被设置为 True 或 False，而 esmtp_features 将是一个字典，包含这个服务器支持的 SMTP 服务扩展的名称，以及它们的参数（如果有）。

除非你想在发送邮件前使用*has_extn()*，否则应该没有必要明确调用这个方法。它将在必要时被*sendmail()* 隐式调用。

SMTP.**ehlo_or_helo_if_needed**()

如果这个会话中没有先前的 EHLO 或 HELO 命令，该方法会调用*ehlo()* 和/或*helo()* 。它首先尝试 ESMTP EHLO 。

SMTPHeloError
服务器没有正确回复 HELO 问候。

SMTP.**has_extn**(*name*)

如果 *name* 在服务器返回的 SMTP 服务扩展集合中，返回 *True*，否则为 *False*。大小写被忽略。

SMTP.**verify**(*address*)

使用 SMTP VRFY 检查此服务器上的某个地址是否有效。如果用户地址有效则返回一个由代码 250 和完整 **RFC 822** 地址（包括人名）组成的元组。否则返回 400 或更大的 SMTP 错误代码以及一个错误字符串。

> **备注**
>
> 许多网站都禁用 SMTP VRFY 以阻止垃圾邮件。

SMTP.**login**(*user*, *password*, *, *initial_response_ok=True*)

登录到一个需要认证的 SMTP 服务器。参数是用于认证的用户名和密码。如果会话在之前没有执行过 EHLO 或 HELO 命令，此方法会先尝试 ESMTP EHLO。如果认证成功则此方法将正常返回，否则可能引发以下异常:

SMTPHeloError
服务器没有正确回复 HELO 问候。

SMTPAuthenticationError
服务器不接受所提供的用户名/密码组合。

SMTPNotSupportedError
服务器不支持 AUTH 命令。

SMTPException
未找到适当的认证方法。

smtplib 所支持的每种认证方法只要被服务器声明支持就会被依次尝试。请参阅 *auth()* 获取受支持的认证方法列表。*initial_response_ok* 会被传递给 *auth()*。

可选的关键字参数 *initial_response_ok* 对于支持它的认证方法，是否可以与 AUTH 命令一起发送 **RFC 4954** 中所规定的“初始响应”，而不是要求回复/响应。

在 3.5 版本发生变更: 可能会引发 *SMTPNotSupportedError*，并添加 *initial_response_ok* 形参。

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)

为指定的认证机制 *mechanism* 发送 SMTP AUTH 命令，并通过 *authobject* 处理回复响应。

mechanism 指定要使用何种认证机制作为 AUTH 命令的参数；可用的值是在 esmtp_features 的 auth 元素中列出的内容。

authobject 必须为接受一个可选的单独参数的可调用对象:

```
data = authobject(challenge=None)
```

如果可选的关键字参数 *initial_response_ok* 为真值，则将先不带参数地调用 authobject()。它可以返回 **RFC 4954** ” 初始响应” ASCII str，其内容将被编码并使用下述的 AUTH 命令来发送。如果 authobject() 不支持初始响应（例如由于要求一个回复），它应当将 None 作为附带 challenge=None 调用的返回值。如果 *initial_response_ok* 为假值，则 authobject() 将不会附带 None 被首先调用。

如果初始响应检测返回了 None，或者如果 *initial_response_ok* 为假值，则将调用 authobject() 来处理服务器的回复响应；它所传递的 *challenge* 参数将为一个 bytes。它应当返回用 base64 进行编码的 ASCII str *data* 并发送给服务器。

SMTP 类提供的 authobjects 针对 CRAM-MD5, PLAIN 和 LOGIN 等机制；它们的名称分别是 SMTP.auth_cram_md5, SMTP.auth_plain 和 SMTP.auth_login。它们都要求将 user 和 password 这两个 SMTP 实例属性设为适当的值。

用户代码通常不需要直接调用 auth，而是调用 *login()* 方法，它将按上述顺序依次尝试上述每一种机制。auth 被公开以便辅助实现 *smtplib* 没有（或尚未）直接支持的认证方法。

Added in version 3.5.

SMTP.**starttls**(*, *context=None*)

将 SMTP 连接设为 TLS (传输层安全) 模式。后续的所有 SMTP 命令都将被加密。你应当随即再次调用*ehlo()*。

如果提供了 *keyfile* 和 *certfile*，它们会被用来创建*ssl.SSLContext*。

可选的 *context* 形参是一个*ssl.SSLContext* 对象；它是使用密钥文件和证书的替代方式，如果指定了该形参则 *keyfile* 和 *certfile* 都应为 None。

如果这个会话中没有先前的 EHLO or HELO 命令，该方法会首先尝试 ESMTP EHLO。

在 3.12 版本发生变更: 已弃用的 *keyfile* 和 *certfile* 形参已被移除。

SMTPHeloError
服务器没有正确回复 HELO 问候。

SMTPNotSupportedError
服务器不支持 STARTTLS 扩展。

RuntimeError
SSL/TLS 支持在你的 Python 解释器上不可用。

在 3.3 版本发生变更: 增加了 *context*。

在 3.4 版本发生变更: 此方法现在支持使用 SSLContext.check_hostname 和 服务器名称指示符 (参见*HAS_SNI*) 进行主机名检测。

在 3.5 版本发生变更: 因缺少 STARTTLS 支持而引发的错误现在是*SMTPNotSupportedError* 子类而不是*SMTPException* 基类。

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

发送邮件。必要参数是一个 **RFC 822** 发件地址字符串，一个 **RFC 822** 收件地址字符串列表（裸字符串将被视为含有 1 个地址的列表），以及一个消息字符串。调用者可以将 ESMTP 选项列表（如 8bitmime）作为 *mail_options* 传入，用于 MAIL FROM 命令。需要与所有 RCPT 命令一起使用的 ESMTP 选项（如 DSN 命令）可以作为 *rcpt_options* 传入。（如果需要对不同的收件人使用不同的 ESMTP 选项，则必须使用底层的方法来发送消息，如 mail(), rcpt() 和 data()。）

> **备注**
>
> *from_addr* 和 *to_addrs* 形参被用来构造传输代理所使用的消息封包。sendmail 不会以任何方式修改消息标头。

msg 可以是一个包含 ASCII 范围内字符的字符串，或是一个字节串。字符串会使用 ascii 编解码器编码为字节串，并且单独的 \r 和 \n 字符会被转换为 \r\n 字符序列。字节串则不会被修改。

如果在此之前本会话没有执行过 EHLO 或 HELO 命令，此方法会先尝试 ESMTP EHLO。如果服务器执行了 ESMTP，消息大小和每个指定的选项将被传递给它（如果指定的选项属于服务器声明的特性集）。如果 EHLO 失败，则将尝试 HELO 并屏蔽 ESMTP 选项。

如果邮件被至少一个接收方接受则此方法将正常返回。在其他情况下它将引发异常。也就是说，如果此方法没有引发异常，则应当会有人收到你的邮件。如果此方法没有引发异常，它将返回一个字典，其中的条目对应每个拒绝的接收方。每个条目均包含由服务器发送的 SMTP 错误代码和相应错误消息所组成的元组。

如果 SMTPUTF8 包括在 *mail_options* 中，并且被服务器所支持，则 *from_addr* 和 *to_addrs* 可能包含非 ASCII 字符。

此方法可能引发以下异常:

SMTPRecipientsRefused
所有收件人都被拒绝。无人收到邮件。该异常的 recipients 属性是一个字典，其中有被拒绝收件人的信息（类似于至少有一个收件人接受邮件时所返回的信息）。

SMTPHeloError
服务器没有正确回复 HELO 问候。

SMTPSenderRefused
服务器不接受 *from_addr*。

SMTPDataError
服务器回复了一个意外的错误代码（而不是拒绝收件人）。

SMTPNotSupportedError
在 *mail_options* 中给出了 SMTPUTF8 但是不被服务器所支持。

除非另有说明，即使在引发异常之后连接仍将被打开。

在 3.2 版本发生变更: *msg* 可以为字节串。

在 3.5 版本发生变更: 增加了 SMTPUTF8 支持，并且如果指定了 SMTPUTF8 但是不被服务器所支持则可能会引发*SMTPNotSupportedError*。

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

本方法是一种快捷方法，用于带着消息调用*sendmail()*，消息由*email.message.Message* 对象表示。参数的含义与*sendmail()* 中的相同，除了 *msg*，它是一个 Message 对象。

如果 *from_addr* 为 None 或 *to_addrs* 为 None ，那么 send_message 将根据 **RFC 5322**，从 *msg* 头部提取地址填充下列参数：如果头部存在 *Sender* 字段，则用它填充 *from_addr*，不存在则用 *From* 字段填充 *from_addr*。*to_addrs* 组合了 *msg* 中的 *To*, *Cc* 和 *Bcc* 字段的值（字段存在的情况下）。如果一组 *Resent-** 头部恰好出现在 message 中，那么就忽略常规的头部，改用 *Resent-** 头部。如果 message 包含多组 *Resent-** 头部，则引发*ValueError*，因为无法明确检测出哪一组 *Resent-* 头部是最新的。

send_message 使用*BytesGenerator* 来序列化 *msg* 并以 \r\n 作为 *linesep*，然后调用*sendmail()* 来传输结果消息。无论 *from_addr* 和 *to_addrs* 的值是什么，send_message 都不会传输 *msg* 中可能出现的 *Bcc* 或 *Resent-Bcc* 标头。如果 *from_addr* 和 *to_addrs* 中的任何地址包含非 ASCII 字符并且服务器没有声明 SMTPUTF8 支持，则会引发*SMTPNotSupportedError*。在其他情况下 Message 将克隆其*policy* 来执行序列化并将*utf8* 属性设为 True，且会把 SMTPUTF8 和 BODY=8BITMIME 添加到 *mail_options* 中。

Added in version 3.2.

Added in version 3.5: 支持国际化地址 (SMTPUTF8)。

SMTP.**quit**()
终结 SMTP 会话并关闭连接。返回 SMTP QUIT 命令的结果。

与标准 SMTP/ESMTP 命令 HELP, RSET, NOOP, MAIL, RCPT 和 DATA 对应的低层级方法也是受支持的。通常不需要直接调用这些方法，因此它们没有被写入本文档。相关细节请参看模块代码。

22.14.2 SMTP 示例

这个例子提示用户输入消息封包所需的地址 (’To’ 和’From’ 地址)，以及要发送的消息。请注意包括在消息中的标头必须包括在输入的消息中；这个例子不对 **RFC 822** 标头进行任何处理。具体来说，’To’ 和’From’ 地址必须显式地包括在消息标头中:

```
import smtplib

def prompt(title):
    return input(title).strip()

from_addr = prompt("From: ")
to_addrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# 在开始时添加 From: 和 To: 标头!
lines = [f"From: {from_addr}", f"To: {', '.join(to_addrs)}", ""]
```

(续下页)

(接上页)

```
while True:
    try:
        line = input()
    except EOFError:
        break
    else:
        lines.append(line)

msg = "\r\n".join(lines)
print("Message length is", len(msg))

server = smtplib.SMTP("localhost")
server.set_debuglevel(1)
server.sendmail(from_addr, to_addrs, msg)
server.quit()
```

备注

通常，你将需要使用*email* 包的特性来构造电子邮件消息，然后你可以通过*send_message()* 来发送它，参见*email: 示例*。

22.15 uuid --- 根据 RFC 4122 定义的 UUID 对象

源代码： Lib/uuid.py

这个模块提供了不可变的*UUID* 对象 (*UUID* 类) 和*uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* 等函数用于生成 **RFC 4122** 所定义的第 1, 3, 4 和 5 版 UUID。

如果你想要的只是一个唯一的 ID，你可能应该调用*uuid1()* 或*uuid4()*。注意*uuid1()* 可能会损害隐私，因为它创建了一个包含计算机网络地址的 UUID。*uuid4()* 可以创建一个随机 UUID。

根据底层平台的支持情况，*uuid1()* 可能会也可能不会返回“安全“的 UUID。安全的 UUID 是指使用同步方法生成的 UUID，此方法可确保没有两个进程能获得相同的 UUID。*UUID* 的所有实例都有一个能够中转关于 UUID 安全性的任何信息的*is_safe* 属性，它可使用下列枚举：

class uuid.**SafeUUID**

Added in version 3.7.

safe

该 UUID 是由平台以多进程安全的方式生成的。

unsafe

UUID 不是以多进程安全的方式生成的。

unknown

该平台不提供 UUID 是否安全生成的信息。

class uuid.**UUID**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, ***, *is_safe=SafeUUID.unknown*)

用一串 32 位十六进制数字、一串大端序 16 个字节作为 *bytes* 参数、一串 16 个小端序字节作为 *bytes_le* 参数、一个由六个整数组成的元组（32 位 *time_low*，16 位 *time_mid*，16 位 *time_hi_version*，8 位 *clock_seq_hi_variant*，8 位 *clock_seq_low*，48 位 *node*）作为 *fields* 参数，或者一个 128 位整数作为 *int* 参数创建一个 UUID。当给出一串十六进制数字时，大括号、连字符和 URN 前缀都是可选的。例如，这些表达式都产生相同的 UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

必须给出 *hex*、*bytes*、*bytes_le*、*fields* 或 *int* 中的唯一一个。*version* 参数是可选的；如果给定，产生的 UUID 将根据 **RFC 4122** 设置其变体和版本号，覆盖给定的 *hex*、*bytes*、*bytes_le*、*fields* 或 *int* 中的位。

UUID 对象的比较是通过比较它们的 *UUID.int* 属性进行的。与非 UUID 对象的比较会引发 *TypeError*。

`str(uuid)` 返回一个 `12345678-1234-5678-1234-567812345678` 形式的字符串，其中 32 位十六进制数字代表 UUID。

UUID 实例有这些只读的属性:

UUID.bytes
UUID 是一个 16 字节的字符串（包含 6 个大端字节序的整数字段）。

UUID.bytes_le
UUID 是一个 16 字节的字符串（其中 *time_low*、*time_mid* 和 *time_hi_version* 为小端字节顺序）。

UUID.fields
以元组形式存放的 UUID 的 6 个整数域，有六个单独的属性和两个派生属性:

域	含意
UUID.time_low	UUID 的前 32 位。
UUID.time_mid	UUID 后续的 16 位。
UUID.time_hi_version	UUID 后续的 16 位。
UUID.clock_seq_hi_variant	UUID 后续的 8 位。
UUID.clock_seq_low	UUID 后续的 8 位。
UUID.node	UUID 的最后 48 位。
UUID.time	60 比特位的时间戳。
UUID.clock_seq	14 比特位的序列编号。

UUID.hex
UUID 是一个 32 字符的小写十六进制数码字符串。

UUID.int
UUID 是一个 128 位的整数。

UUID.**urn**

在 **RFC 4122** 中定义的 URN 形式的 UUID。

UUID.**variant**

UUID 的变体，它决定了 UUID 的内部布局。这将是*RESERVED_NCS*，*RFC_4122*，*RESERVED_MICROSOFT* 或*RESERVED_FUTURE* 中的一个。

UUID.**version**

UUID 版本号（1 到 5，只有当变体为*RFC_4122* 时才有意义）。

UUID.**is_safe**

一个*SafeUUID* 的枚举，表示平台是否以多进程安全的方式生成 UUID。

Added in version 3.7.

uuid 模块定义了以下函数：

uuid.**getnode**()

获取 48 位正整数形式的硬件地址。第一次运行时，它可能会启动一个单独的程序，这可能会相当慢。如果所有获取硬件地址的尝试都失败了，我们会按照 **RFC 4122** 中的建议，选择一个随机的 48 位数字，其多播位 (第一个八进制数的最小有效位) 设置为 1。“硬件地址”是指一个网络接口的 MAC 地址。在一台有多个网络接口的机器上，普遍管理的 MAC 地址 (即第一个八位数的第二个最小有效位是 未设置的) 将比本地管理的 MAC 地址优先，但没有其他排序保证。

在 3.7 版本发生变更: 普遍管理的 MAC 地址优于本地管理的 MAC 地址，因为前者保证是全球唯一的，而后者则不是。

uuid.**uuid1**(*node=None*, *clock_seq=None*)

根据主机 ID、序列号和当前时间生成一个 UUID。如果没有给出 *node*，则使用*getnode()* 来获取硬件地址。如果给出了 *clock_seq*，它将被用作序列号；否则将选择一个随机的 14 比特位序列号。

uuid.**uuid3**(*namespace*, *name*)

根据命名空间标识符（一个 UUID）和名称（一个*bytes* 对象或将使用 UTF-8 进行编码的字符串）的 MD5 哈希值来生成一个 UUID。

uuid.**uuid4**()

生成一个随机的 UUID。

uuid.**uuid5**(*namespace*, *name*)

根据命名空间标识符（一个 UUID）和名称（一个*bytes* 对象或将使用 UTF-8 进行编码的字符串）的 SHA-1 哈希值来生成一个 UUID。

uuid 模块定义了以下命名空间标识符，供*uuid3()* 或*uuid5()* 使用。

uuid.**NAMESPACE_DNS**

当指定这个命名空间时，*name* 字符串是一个完全限定的域名。

uuid.**NAMESPACE_URL**

当指定这个命名空间时，*name* 字符串是一个 URL。

uuid.**NAMESPACE_OID**

当指定这个命名空间时，*name* 字符串是一个 ISO OID。

uuid.**NAMESPACE_X500**

当指定这个命名空间时，*name* 字符串是 DER 或文本输出格式的 X.500 DN。

uuid 模块为*variant* 属性的可能值定义了以下常量：

uuid.**RESERVED_NCS**

为 NCS 兼容性保留。

uuid.**RFC_4122**

指定 **RFC 4122** 中给出的 UUID 布局。

uuid.**RESERVED_MICROSOFT**
为微软的兼容性保留。

uuid.**RESERVED_FUTURE**
保留给未来的定义。

参见

RFC 4122 - 通用唯一标识符（UUID）URN 命名空间
本规范定义了 UUID 的统一资源名称空间，UUID 的内部格式，以及生成 UUID 的方法。

22.15.1 命令行用法

Added in version 3.12.

uuid 模块可以在命令行下作为脚本来执行。

```
python -m uuid [-h] [-u {uuid1,uuid3,uuid4,uuid5}] [-n NAMESPACE] [-N NAME]
```

可以接受以下选项：

-h, **--help**
显示帮助信息并退出。

-u <uuid>

--uuid <uuid>
指定要用于生成 uuid 的函数名称。默认会使用*uuid4()*。

-n <namespace>

--namespace <namespace>
该命名空间是一个 UUID 或者 @ns 其中 ns 是一个以命名空间名称来定位的知名预定义 UUID。例如 @dns, @url, @oid 和 @x500。仅对于*uuid3()* / *uuid5()* 等函数是必需的。

-N <name>

--name <name>
用作生成 uuid 的一部分的名称。仅对于*uuid3()* / *uuid5()* 等函数是必需的。

22.15.2 示例

下面是一些*uuid* 模块的典型使用例子:

```
>>> import uuid

>>> # 基于主机 ID 和当前时间生成一个 UUID
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # 使用命名空间 UUID 和特定名称的 MD5 哈希值生成一个 UUID
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # 生成一个随机 UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # 使用命名空间 UUID 和特定名称的 SHA-1 哈希值生成一个 UUID
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')
```

(续下页)

(接上页)

```
>>> # 基于十六进制数码字符串（忽略花括号和连字符）生成一个 UUID
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # 将一个 UUID 转换为标准格式的十六进制数码字符串
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # 获取 UUID 的 16 字节原始字节串
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # 基于 16 字节形式的字符串获取 UUID
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

22.15.3 命令行示例

下面是一些*uuid*命令行接口的典型用法示例:

```
# 生成一个随机的 uuid - 默认将使用 uuid4()
$ python -m uuid

# 使用 uuid1() 生成一个 uuid
$ python -m uuid -u uuid1

# 使用 uuid5 生成一个 uuid
$ python -m uuid -u uuid5 -n @url -N example.com
```

22.16 socketserver --- 用于网络服务器的框架

源代码: Lib/socketserver.py

---

*socketserver*模块简化了编写网络服务器的任务。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台*了解详情。

该模块具有四个基础实体服务器类:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

该类使用互联网 TCP 协议，它可以提供客户端与服务器之间的连续数据流。如果 *bind_and_activate* 为真值，该类的构造器会自动尝试唤起*server_bind()*和*server_activate()*。其他形参会被传递给*BaseServer*基类。

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

该类使用数据包，即一系列离散的信息分包，它们可能会无序地到达或在传输中丢失。该类的形参与*TCPServer*的相同。

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

这两个更常用的类与 TCP 和 UDP 类相似，但使用 Unix 域套接字；它们在非 Unix 系统平台上不可用。它们的形参与*TCPServer*的相同。

这四个类会同步地处理请求；每个请求必须完成才能开始下一个请求。这就不适用于每个请求要耗费很长时间来完成的情况，或者因为它需要大量的计算，又或者它返回了大量的数据而客户端处理起来很缓慢。解决方案是创建单独的进程或线程来处理每个请求；*ForkingMixIn*和*ThreadingMixIn*混合类可以被用于支持异步行为。

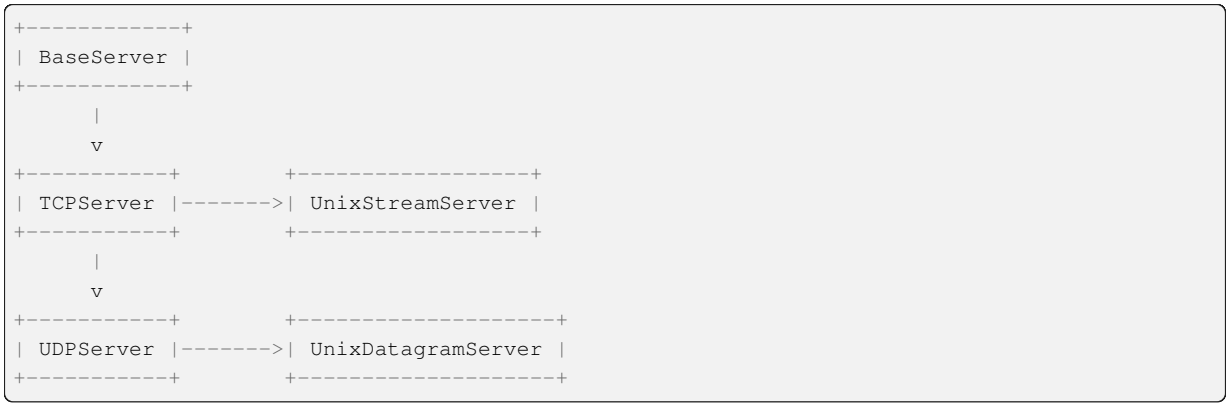
创建一个服务器需要分几个步骤进行。首先，你必须通过子类化*BaseRequestHandler* 类并重载其*handle()* 方法来创建一个请求处理器类；这个方法将处理传入的请求。其次，你必须实例化某个服务器类，将服务器地址和请求处理器类传给它。建议在 `with` 语句中使用该服务器。然后再调用服务器对象的*handle_request()* 或*serve_forever()* 方法来处理一个或多个请求。最后，调用*server_close()* 来关闭套接字（除非你使用了 `with` 语句）。

当从*ThreadingMixIn* 继承线程连接行为时，你应当显式地声明你希望在突然关机时你的线程采取何种行为。*ThreadingMixIn* 类定义了一个属性 *daemon_threads*，它指明服务器是否应当等待线程终止。如果你希望线程能自主行动你应当显式地设置这个旗标；默认值为*False*，表示 Python 将不会在*ThreadingMixIn* 所创建的所有线程都退出之前退出。

服务器类具有同样的外部方法和属性，无论它们使用哪种网络协议。

22.16.1 服务器创建的说明

在继承图中有五个类，其中四个代表四种类型的同步服务器:



请注意*UnixDatagramServer* 是派生自*UDPServer*，而不是派生自*UnixStreamServer* --- IP 和 Unix 流服务器的唯一区别地址族。

class socketserver.**ForkingMixIn**

class socketserver.**ThreadingMixIn**

每种服务器类型的分叉和线程版本都可以使用这些混合类来创建。例如，*ThreadingUDPServer* 的创建方式如下:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

混合类先出现，因为它重载了*UDPServer* 中定义的一个方法。设置各种属性也会改变下层服务器机制的行为。

ForkingMixIn 和下文提及的分叉类仅在支持*fork()* 的 POSIX 系统平台上可用。

block_on_close

ForkingMixIn.server_close 会等待直到所有子进程完成，除非*block_on_close* 属性为 `False`。

ThreadingMixIn.server_close 会等待直到所有非守护程序类线程完成，除非*block_on_close* 属性为 `False`。

daemon_threads

对于*ThreadingMixIn* 可通过将*ThreadingMixIn.daemon_threads* 设为 `True` 来使用守护线程从而无需等待线程完成。

在 3.7 版本发生变更: *ForkingMixIn.server_close* 和*ThreadingMixIn.server_close* 现在会等待直到所有子进程和非守护类线程完成。新增了一个*ForkingMixIn.block_on_close* 类属性用来选择 3.7 版之前的行为。

class socketserver.**ForkingTCPServer**

class socketserver.**ForkingUDPServer**

class socketserver.**ThreadingTCPServer**

class socketserver.**ThreadingUDPServer**

class socketserver.**ForkingUnixStreamServer**

class socketserver.**ForkingUnixDatagramServer**

class socketserver.**ThreadingUnixStreamServer**

class socketserver.**ThreadingUnixDatagramServer**

这些类都是使用混合类来预定义的。

Added in version 3.12: 增加了 ForkingUnixStreamServer 和 ForkingUnixDatagramServer 类。

要实现一个服务，你必须从*BaseRequestHandler* 派生一个类并重定义其*handle()* 方法。然后你可以通过组合某种服务器类型与你的请求处理器类来运行各种版本的服务。请求处理器类对于数据报和流服务必须是不相同的。这可以通过使用处理器子类*StreamRequestHandler* 或*DatagramRequestHandler* 来隐藏。

当然，你仍然需要动点脑筋！举例来说，如果服务包含可能被不同请求所修改的内存状态则使用分叉服务器是没有意义的，因为在子进程中的修改将永远不会触及保存在父进程中的初始状态并传递到各个子进程。在这种情况下，你可以使用线程服务器，但你可能必须使用锁来保护共享数据的一致性。

另一方面，如果你是在编写一个所有数据保存在外部（例如文件系统）的 HTTP 服务器，同步类实际上将在正在处理某个请求的时候“失聪” -- 如果某个客户端在接收它所请求的所有数据时很缓慢这可能会是非常长的时间。这时线程或分叉服务器会更为适用。

在某些情况下，合适的做法是同步地处理请求的一部分，但根据请求数据在分叉的子进程中完成处理。这可以通过使用一个同步服务器并在请求处理器类*handle()* 中进行显式分叉来实现。

另一种可以在既不支持线程也不支持*fork()* 的环境（或者对于本服务来说这两者开销过大或不适用）中处理多个同时请求的方式是维护一个显式的部分完成的请求表并使用*selectors* 来决定接下来要处理哪个请求（或者是否要处理一个新传入的请求）。这对于流式服务来说特别重要，因为每个客户端可能会连接很长的时间（如果不能使用线程或子进程）。

22.16.2 Server 对象

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

这是本模块中所有 Server 对象的超类。它定义了下文给出的接口，但没有实现大部分的方法，它们应在子类中实现。两个形参存储在对应的*server_address* 和*RequestHandlerClass* 属性中。

fileno()

返回服务器正在监听的套接字的以整数表示的文件描述符。此函数最常被传递给*selectors*，以允许在同一进程中监控多个服务器。

handle_request()

处理单个请求。此函数会依次调用下列方法: *get_request()*, *verify_request()* 和*process_request()*。如果用户提供的处理器类的*handle()* 方法引发了异常，则将调用服务器的*handle_error()* 方法。如果在*timeout* 秒内未接收到请求，将会调用*handle_timeout()* 并将返回*handle_request()*。

serve_forever(*poll_interval=0.5*)

对请求进行处理直至收到显式的*shutdown()* 请求。每隔 *poll_interval* 秒对 shutdown 进行轮询。忽略*timeout* 属性。它还会调用*service_actions()*，这可被子类或混合类用来提供某个给定服务的专属操作。例如，*ForkingMixIn* 类使用*service_actions()* 来清理僵尸子进程。

在 3.3 版本发生变更: 将 service_actions 调用添加到 serve_forever 方法。

service_actions()

此方法会在 the *serve_forever()* 循环中被调用。此方法可被子类或混合类所重载以执行某个给定服务的专属操作，例如清理操作。

Added in version 3.3.

shutdown()

通知*serve_forever()* 循环停止并等待它完成。*shutdown()* 必须在*serve_forever()* 运行于不同线程时被调用否则它将发生死锁。

server_close()

清理服务器。此方法可被重载。

address_family

服务器套接字所属的协议族。常见的例子有*socket.AF_INET*, *socket.AF_INET6* 和*socket.AF_UNIX* 等。如果你想要 IPv6 服务器类请子类化此模块中的 TCP 或 UDP 服务器类并设置类属性 `address_family = AF_INET6`。

RequestHandlerClass

用户提供的请求处理器类；将为每个请求创建该类的实例。

server_address

服务器所监听的地址。地址的格式因具体协议族而不同；请参阅*socket* 模块的文档了解详情。对于互联网协议，这将是一个元组，其中包含一个表示地址的字符串，和一个表示端口号的整数，例如: `('127.0.0.1', 80)`。

socket

将由服务器用于监听入站请求的套接字对象。

服务器类支持下列类变量:

allow_reuse_address

服务器是否要允许地址的重用。默认值为*False*，并可在子类中设置以改变策略。

request_queue_size

请求队列的长度。如果处理单个请求要花费很长的时间，则当服务器正忙时到达的任何请求都会被加入队列，最多加入*request_queue_size* 个请求。一旦队列被加满，来自客户端的更多请求将收到"Connection denied" 错误。默认值为 5，但可在子类中重载。

socket_type

服务器使用的套接字类型；常见的有*socket.SOCK_STREAM* 和*socket.SOCK_DGRAM* 这两个值。

timeout

超时限制，以秒数表示，或者如果不限制超时则为*None*。如果在超时限制期间没有收到*handle_request()*，则会调用*handle_timeout()* 方法。

有多个服务器方法可被服务器基类的子类例如*TCPServer* 所重载；这些方法对服务器对象的外部用户来说并无用处。

finish_request(*request*, *client_address*)

通过实例化*RequestHandlerClass* 并调用其*handle()* 方法来实际处理请求。

get_request()

必须接受来自套接字的请求，并返回一个 2 元组，其中包含用来与客户端通信的 *new* 套接字对象，以及客户端的地址。

handle_error(*request*, *client_address*)

此函数会在*RequestHandlerClass* 实例的*handle()* 方法引发异常时被调用。默认行为是将回溯信息打印到标准错误并继续处理其他请求。

在 3.6 版本发生变更: 现在只针对派生自*Exception* 类的异常调用此方法。

handle_timeout()

此函数会在*timeout* 属性被设为*None* 以外的值并且在超出时限之后仍未收到请求时被调用。分叉服务器的默认行为是收集任何已退出的子进程状态，而在线程服务器中此方法则不做任何操作。

process_request(*request*, *client_address*)

调用*finish_request()* 来创建*RequestHandlerClass* 的实例。如果需要，此函数可创建一个新的进程或线程来处理请求；*ForkingMixIn* 和*ThreadingMixIn* 类能完成此任务。

server_activate()

由服务器的构造器调用以激活服务器。TCP 服务器的默认行为只是在服务器的套接字上唤起*listen()*。可以被重载。

server_bind()

由服务器的构造器调用以将套接字绑定到所需的地址。可以被重载。

verify_request(*request*, *client_address*)

必须返回一个布尔值；如果值为*True*，请求将被处理。而如果值为*False*，请求将被拒绝。此函数可被重载以实现服务器的访问控制。默认实现总是返回*True*。

在 3.6 版本发生变更: 添加了对*context manager* 协议的支持。退出上下文管理器与调用*server_close()* 等效。

22.16.3 请求处理器对象

class socketserver.**BaseRequestHandler**

这是所有请求处理器对象的超类。它定义了下文列出的接口。一个实体请求处理器子类必须定义新的*handle()* 方法，并可重载任何其他方法。对于每个请求都会创建一个新的子类的实例。

setup()

会在*handle()* 方法之前被调用以执行任何必要的初始化操作。默认实现不执行任何操作。

handle()

此函数必须执行为请求提供服务所需的全部操作。默认实现不执行任何操作。它有几个可用的实例属性；请求为*request*；客户端地址为*client_address*；服务器实例为*server*，如果它需要访问特定服务器信息的话。, in case it needs access to per-server information.

针对数据报或流服务的*request* 类型是不同的。对于流服务，*request* 是一个套接字对象；对于数据报服务，*request* 是一对字符串于套接字。

finish()

在*handle()* 方法之后调用以执行任何需要的清理操作。默认实现不执行任何操作。如果*setup()* 引发了异常，此函数将不会被调用。

request

将被用于同客户端通信的 新*socket.socket* 对象。

client_address

BaseServer.get_request() 所返回的客户端地址。

server

用于处理请求的*BaseServer* 对象。

class socketserver.**StreamRequestHandler**

class socketserver.**DatagramRequestHandler**

这些*BaseRequestHandler* 子类重载了*setup()* 和*finish()* 方法，并提供了*rfile* 和*wfile* 属性。

rfile

用于读取所接受请求的文件对象。支持*io.BufferedIOBase* 可读接口。

wfile

用于写入所回复内容的文件对象。支持*io.BufferedIOBase* 可写接口。

在 3.6 版本发生变更: *wfile* 也支持*io.BufferedIOBase* 可写接口。

22.16.4 例子

socketserver.TCPServer 示例

以下是服务端:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request 是连接到客户端的 TCP 套接字
        pieces = [b'']
        total = 0
        while b'\n' not in pieces[-1] and total < 10_000:
            pieces.append(self.request.recv(2000))
            total += len(pieces[-1])
        self.data = b''.join(pieces)
        print(f"Received from {self.client_address[0]}:")
        print(self.data.decode("utf-8"))
        # 发回同样的数据，但转为大写形式
        self.request.sendall(self.data.upper())
        # 在我们返回后，套接字将被关闭。

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # 创建服务器，绑定到 localhost 的 9999 端口
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # 激活服务器；它将持续运行直到你
        # 使用 Ctrl-C 中断程序
        server.serve_forever()
```

一个使用流（通过提供标准文件接口来简化通信的文件型对象）的替代请求处理器类:

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile 是由该处理器创建的文件型对象。
        # 我们现在可以使用 readline() 代替原始 recv() 调用
        # 我们自己限制为 10000 字节以避免被发送方滥用。
        self.data = self.rfile.readline(10000).rstrip()
        print(f"{self.client_address[0]} wrote:")
        print(self.data.decode("utf-8"))
        # 类似地，self.wfile 是用于写回到客户端的
        # 文件型对象
        self.wfile.write(self.data.upper())
```

区别在于第二个处理器的 `readline()` 调用将多次调用 `recv()` 直至遇到一个换行符，而第一个处理器必须使用一个 `recv()` 循环来累积数据直至遇到一个换行符。如果它只使用一个 `recv()` 而不带循环则将只返回当前已从客户端接收的内容。TCP 是基于流的：数据将按其发送顺序到达，但在客户端 `send()` 或 `sendall()` 调用和服务端需要接收它的 `recv()` 调用数量之间并没有关联。

以下是客户端:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# 创建一个套接字 (SOCK_STREAM 表示一个 TCP 套接字)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # 连接到服务器并发送数据
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data, "utf-8"))
    sock.sendall(b"\n")

    # 从服务器接收数据并关闭
    received = str(sock.recv(1024), "utf-8")

print("Sent:     ", data)
print("Received:", received)
```

这个示例程序的输出应该是像这样的:

服务器:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

客户端:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

socketserver.UDPServer 示例

以下是服务端:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print(f"{self.client_address[0]} wrote:")
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
```

(续下页)

(接上页)

```
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

以下是客户端:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM 是用于 UDP 套接字的套接字类型
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# 如你所见，没有 connect() 调用；UDP 没有连接。
# 数据是通过 sendto() 直接发给接收方的。
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     ", data)
print("Received:", received)
```

这个示例程序的输出应该是与 TCP 服务器示例相一致的。

异步混合类

要构建异步处理器，请使用*ThreadingMixIn* 和*ForkingMixIn* 类。

ThreadingMixIn 类的示例:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # 端口 0 表示选择任意一个未使用的端口
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # 启动一个服务器线程 -- 该线程将在此后
```

(续下页)

(接上页)

```
        # 为每个请求再启动一个线程
        server_thread = threading.Thread(target=server.serve_forever)
        # 在主线程终结时退出服务器线程
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

这个示例程序的输出应该是像这样的:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

ForkingMixIn 类的使用方式是相同的，区别在于服务器将为每个请求产生一个新的进程。仅在支持*fork()* 的 POSIX 系统平台上可用。

22.17 http.server --- HTTP 服务器

源代码： Lib/http/server.py

这个模块定义了用于实现 HTTP 服务器的类。

> **警告**
>
> *http.server* 不推荐用于生产环境。它仅仅实现了*basic security checks* 的要求。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

HTTPServer 是*socketserver.TCPServer* 的一个子类。它会创建和侦听 HTTP 套接字，并将请求分发给处理程序。创建和运行 HTTP 服务器的代码类似如下所示：

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)

该类基于*TCPServer* 类，并在实例变量 `server_name` 和 `server_port` 中保存 HTTP 服务器地址。处理程序可通过实例变量 `server` 访问 HTTP 服务器。

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)

该类相似于 HTTPServer，只是会利用*ThreadingMixIn* 对请求进行多线程处理。当需要对 Web 浏览器预先打开套接字进行处理时，这就很有用，这时*HTTPServer* 会一直等待请求。

Added in version 3.7.

实例化*HTTPServer* 和*ThreadingHTTPServer* 时，必须给出一个 *RequestHandlerClass*，本模块提供了该对象的三种变体：

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)

这个类用于处理到达服务器的 HTTP 请求。它本身无法响应任何实际的 HTTP 请求；它必须被子类化以处理每个请求方法（例如 GET 或 POST）。*BaseHTTPRequestHandler* 提供了许多供子类使用的类和实例变量以及方法。

这个处理器将解析请求和标头，然后调用特定请求类型对应的方法。方法名称将根据请求来构造。例如，对于请求方法 SPAM，将不带参数地调用 do_SPAM() 方法。所有相关信息会被保存在该处理器的实例变量中。子类不重写或扩展 __init__() 方法。

BaseHTTPRequestHandler 具有下列实例变量:

client_address

包含 (host, port) 形式的指向客户端地址的元组。

server

包含服务器实例。

close_connection

应当在*handle_one_request()* 返回之前设定的布尔值，指明是否要期待另一个请求，还是应当关闭连接。

requestline

包含 HTTP 请求行的字符串表示。末尾的 CRLF 会被去除。该属性应当由*handle_one_request()* 来设定。如果无有效请求行被处理，则它应当被设为空字符串。

command

包含具体的命令（请求类型）。例如 'GET'。

path

包含请求路径。如果 URL 的查询部分存在，path 会包含这个查询部分。使用 **RFC 3986** 的术语来说，在这里，path 包含 hier-part 和 query 。

request_version

包含请求的版本字符串。例如 'HTTP/1.0'。

headers

存放由*MessageClass* 类变量所指定的类的实例。该实例会解析并管理 HTTP 请求中的标头。*http.client* 中的*parse_headers()* 函数将被用来解析标头并且它需要 HTTP 请求提供一个有效的 **RFC 2822** 风格的标头。

rfile

一个*io.BufferedIOBase* 输入流，准备从可选的输入数据的开头进行读取。

wfile

包含用于写入响应并发回给客户端的输出流。在写入流时必须正确遵守 HTTP 协议以便成功地实现与 HTTP 客户端的互操作。

在 3.6 版本发生变更: 这是一个*io.BufferedIOBase* 流。

BaseHTTPRequestHandler 具有下列属性:

server_version

指定服务器软件版本。你可能会想要重写该属性。该属性的格式为多个以空格分隔的字符串，其中每个字符串的形式为 name[/version]。例如 'BaseHTTP/0.2'。

sys_version

包含 Python 系统版本，采用*version_string* 方法和*server_version* 类变量所支持的形式。例如 'Python/1.4'。

error_message_format

指定应当被*send_error()* 方法用来构建发给客户端的错误响应的格式字符串。该字符串应使用来自*responses* 的变量根据传给*send_error()* 的状态码来填充默认值。

error_content_type

指定发送给客户端的错误响应的 Content-Type HTTP 标头。默认值为 `'text/html'`。

protocol_version

指定服务器所符合的 HTTP 版本。它会在响应中发送以便让客户端知道服务器对于未来请求的通信能力。如果设置为 `'HTTP/1.1'`，服务器将允许 HTTP 持久连接；但是，你的服务器必须在所有对客户端的响应中包括一个准确的 `Content-Length` 标头 (使用*send_header()*)。为了保持向下兼容性，该设置默认为 `'HTTP/1.0'`。

MessageClass

指定一个*email.message.Message* 这样的类来解析 HTTP 标头。通常该属性不会被重写，其默认值为*http.client.HTTPMessage*。

responses

该属性包含一个整数错误代码与由短消息和长消息组成的二元组的映射。例如，`{code: (shortmessage, longmessage)}`。*shortmessage* 通常是作为消息响应中的 *message* 键，而 *longmessage* 则是作为 *explain* 键。该属性会被*send_response_only()* 和*send_error()* 方法所使用。

BaseHTTPRequestHandler 实例具有下列方法:

handle()

调用*handle_one_request()* 一次（或者如果启用了永久连接则为多次）来处理传入的 HTTP 请求。你应该永远不需要重写它；而是要实现适当的 `do_*()` 方法。

handle_one_request()

此方法将解析并请请求分配给适当的 `do_*()` 方法。你应该永远不需要重写它。

handle_expect_100()

当一个符合 HTTP/1.1 标准的服务器接收到一个 `Expect: 100-continue` 请求标头时它会以一个 `100 Continue` 加 `200 OK` 标头作为响应。如果服务器不希望客户端继续则可以通过重写来引发一个错误。例如服务器可以选择发送 `417 Expectation Failed` 作为响应标头并 `return False`。

Added in version 3.2.

send_error(*code*, *message=None*, *explain=None*)

发送并记录回复给客户端的完整的错误信息。数字形式的 *code* 指明 HTTP 错误代码，可选的 *message* 为简短的易于人类阅读的错误描述。*explain* 参数可被用于提供更详细的错误信息；它将使用*error_message_format* 属性来进行格式化并在一组完整的标头之后作为响应体被发送。*responses* 属性保存了 *message* 和 *explain* 的默认值并将在未提供值时被使用；对于未知代码这两者的默认值均为字符串 `???`。如果方法为 HEAD 或响应代码为下列值一则响应体将为空: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`。

在 3.4 版本发生变更: 错误响应包括一个 Content-Length 标头。增加了 *explain* 参数。

send_response(*code*, *message=None*)

将一个响应标头添加到标头缓冲区并记录被接受的请求。HTTP 响应行会被写入到内部缓冲区，后面是 *Server* 和 *Date* 标头。这两个标头的值将分别通过*version_string()* 和*date_time_string()* 方法获取。如果服务器不打算使用*send_header()* 方法发送任何其他标头，则*send_response()* 后面应该跟一个*end_headers()* 调用。

在 3.3 版本发生变更: 标头会被存储到内部缓冲区并且需要显式地调用*end_headers()*。

send_header(*keyword*, *value*)

将 HTTP 标头添加到内部缓冲区，它将在*end_headers()* 或*flush_headers()* 被唤起时写入输出流。*keyword* 应当指定标头关键字，并以 *value* 指定其值。请注意，在 send_header 调用结束之后，必须调用*end_headers()* 以便完成操作。

在 3.2 版本发生变更: 标头将被存入内部缓冲区。

send_response_only(*code*, *message=None*)

只发送响应标头，用于当 `100 Continue` 响应被服务器发送给客户端的场合。标头不会被缓冲而是直接发送到输出流。如果未指定 *message*，则会发送与响应 *code* 相对应的 HTTP 消息。

Added in version 3.2.

end_headers()

将一个空行（指明响应中 HTTP 标头的结束）添加到标头缓冲区并调用*flush_headers()*。

在 3.2 版本发生变更: 已缓冲的标头会被写入到输出流。

flush_headers()

最终将标头发送到输出流并清空内部标头缓冲区。

Added in version 3.3.

log_request(*code='-'*, *size='-'*)

记录一次被接受（成功）的请求。*code* 应当指定与请求相关联的 HTTP 代码。如果请求的大小可用，则它应当作为 *size* 形参传入。

log_error(...)

当请求无法完成时记录一次错误。默认情况下，它会将消息传给*log_message()*，因此它接受同样的参数 (*format* 和一些额外的值)。

log_message(*format*, ...)

将任意一条消息记录到 `sys.stderr`。此方法通常会被重写以创建自定义的错误日志记录机制。*format* 参数是标准 printf 风格的格式字符串，其中会将传给*log_message()* 的额外参数用作格式化操作的输入。每条消息日志记录的开头都会加上客户端 IP 地址和当前日期时间。

version_string()

返回服务器软件的版本字符串。该值为*server_version* 与*sys_version* 属性的组合。

date_time_string(*timestamp=None*)

返回由 *timestamp* 所给定的日期和时间（参数应为 `None` 或为*time.time()* 所返回的格式），格式化为一个消息标头。如果省略 *timestamp*，则会使用当前日期和时间。

结果看起来像 `'Sun, 06 Nov 1994 08:49:37 GMT'`。

log_date_time_string()

返回当前的日期和时间，为日志格式化

address_string()

返回客户端的地址

在 3.3 版本发生变更: 在之前版本中，会执行一次名称查找。为了避免名称解析的时延，现在将总是返回 IP 地址。

class `http.server.`**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)

这个类会为目录 *directory* 及以下的文件提供发布服务，或者如果未提供 *directory* 则为当前目录，直接将目录结构映射到 HTTP 请求。

在 3.7 版本发生变更: 增加了 *directory* 形参。

在 3.9 版本发生变更: *directory* 形参接受一个*path-like object*。

诸如解析请求之类的大量工作都是由基类*BaseHTTPRequestHandler* 完成的。本类实现了*do_GET()* 和*do_HEAD()* 函数。

以下是*SimpleHTTPRequestHandler* 的类属性。

server_version

这会是 `"SimpleHTTP/" + __version__`，其中 `__version__` 定义于模块级别。

extensions_map

将后缀映射为 MIME 类型的字典，其中包含了覆盖系统默认值的自定义映射关系。不区分大小写，因此字典键只应为小写值。

在 3.9 版本发生变更: 此字典不再填充默认的系统映射，而只包含覆盖值。

SimpleHTTPRequestHandler 类定义了以下方法：

do_HEAD()

本方法为 'HEAD' 请求提供服务：它将发送等同于 GET 请求的头文件。关于合法头部信息的更完整解释，请参阅*do_GET()* 方法。

do_GET()

通过将请求解释为相对于当前工作目录的路径，将请求映射到某个本地文件。

如果请求被映射到目录，则会依次检查该目录是否存在 index.html 或 index.htm 文件。若存在则返回文件内容；否则会调用 list_directory() 方法生成目录列表。本方法将利用*os.listdir()* 扫描目录，如果*listdir()* 失败，则返回 404 出错应答。

如果请求被映射到文件，则会打开该文件。打开文件时的任何*OSError* 异常都会被映射为 404, 'File not found' 错误。如果请求中带有 'If-Modified-Since' 标头，而在此时间点之后文件未作修改，则会发送 304, 'Not Modified' 的响应。否则会调用 guess_type() 方法猜测内容的类型，该方法会反过来用到 *extensions_map* 变量，并返回文件内容。

将会输出 'Content-type:' 头部信息，带上猜出的内容类型，然后是 'Content-Length:' 头部信息，带有文件的大小，以及 'Last-Modified:' 头部信息，带有文件的修改时间。

后面是一个空行，标志着头部信息的结束，然后输出文件的内容。如果文件的 MIME 类型以 text/ 开头，文件将以文本模式打开；否则将使用二进制模式。

示例用法请见在 Lib/http/server.py 中 test 函数的实现。

在 3.7 版本发生变更: 为 'If-Modified-Since' 头部信息提供支持。

SimpleHTTPRequestHandler 类的用法可如下所示，以便创建一个非常简单的 Web 服务，为相对于当前目录的文件提供服务：

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

SimpleHTTPRequestHandler 也可以被子类化以便增强其行为，例如通过重写类属性 index_pages 以使用不同的 index 文件名。

class http.server.**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

该类可为当前及以下目录中的文件或输出 CGI 脚本提供服务。注意，把 HTTP 分层结构映射到本地目录结构，这与*SimpleHTTPRequestHandler* 完全一样。

> **备注**
>
> 由*CGIHTTPRequestHandler* 类运行的 CGI 脚本不能进行重定向操作（HTTP 代码 302），因为在执行 CGI 脚本之前会发送代码 200（接下来就输出脚本）。这样状态码就冲突了。

然而，如果这个类猜测它是一个 CGI 脚本，那么就会运行该 CGI 脚本，而不是作为文件提供出去。只会识别基于目录的 CGI ——另有一种常用的服务器设置，即标识 CGI 脚本是通过特殊的扩展名。

如果请求指向 cgi_directories 以下的路径，do_GET() 和 do_HEAD() 函数已作修改，不是给出文件，而是运行 CGI 脚本并输出结果。

CGIHTTPRequestHandler 定义了以下数据成员：

cgi_directories

默认为 ['/cgi-bin', '/htbin']，视作 CGI 脚本所在目录。

CGIHTTPRequestHandler 定义了以下方法：

do_POST()

本方法服务于 'POST' 请求，仅用于 CGI 脚本。如果试图向非 CGI 网址发送 POST 请求，则会输出错误 501：Can only POST to CGI scripts"。

请注意，为了保证安全性，CGI 脚本将以用户 nobody 的 UID 运行。CGI 脚本运行错误将被转换为错误 403。

Deprecated since version 3.13, will be removed in version 3.15: *CGIHTTPRequestHandler* 将在 3.15 中移除。早在十多年前 CGI 就已不被认为是一个好的解决方案。目前此代码已长期未得到维护并且极少被实际使用。保留它可能导致进一步的安全考量。

22.17.1 命令行接口

http.server 也可使用解释器的 -m 开关来直接唤起。下面的例子演示了如何以当前目录作为相对起点来发布文件:

```
python -m http.server [OPTIONS] [port]
```

可以接受以下选项：

port

服务器默认监听端口为 8000。可以通过传递所需的端口号作为参数来覆盖默认值:

```
python -m http.server 9000
```

-b, --bind <address>

指定一个它要绑定的特定地址。IPv4 和 IPv6 地址均受到支持。在默认情况下，服务器会将自己绑定到所有接口。例如，以下命令将使服务器仅绑定 localhost:

```
python -m http.server --bind 127.0.0.1
```

Added in version 3.4.

在 3.8 版本发生变更: 在 --bind 选项中支持 IPv6。

-d, --directory <dir>

指定一个它要提供文件服务的目录。在默认情况下，服务器会使用当前目录。例如，以下命令将使用一个特定地址:

```
python -m http.server --directory /tmp/
```

Added in version 3.7.

-p, --protocol <version>

指定一个服务器所采用的 HTTP 版本。在默认情况下，服务器会采用 HTTP/1.0。例如，以下命令将运行一个采用 HTTP/1.1 的服务器:

```
python -m http.server --protocol HTTP/1.1
```

Added in version 3.11.

--cgi

通过在命令行传入 `--cgi` 参数，可以启用*CGIHTTPRequestHandler*：

```
python -m http.server --cgi
```

Deprecated since version 3.13, will be removed in version 3.15: *http.server* 命令行的 `--cgi` 支持已被移除因为*CGIHTTPRequestHandler* 将要被移除。

警告

CGIHTTPRequestHandler 和 `--cgi` 命令行选项不适合不受信任的客户端使用且容易被恶意利用。应当始终在安全的环境中使用。

22.17.2 安全考量

当处理请求时，*SimpleHTTPRequestHandler* 会解析符号链接，这有可能使得指定文件夹以外的文件被暴露。

较早版本的 Python 不会擦除从 `python -m http.server` 或默认的*BaseHTTPRequestHandler* . `log_message` 实现发送到 stderr 的日志消息中的控制字符。这可以允许连接到你的服务器的远程客户端向你的终端发送邪恶的控制代码。

在 3.12 版本发生变更: 控制字符会在 stderr 日志中被擦除。

22.18 `http.cookies` --- HTTP 状态管理

源代码: Lib/http/cookies.py

---

http.cookies 模块定义的类将 cookie 的概念抽象了出来，这是一种 HTTP 状态的管理机制。它既支持简单的纯字符串形式的 cookie，也为任何可序列化数据类型的 cookie 提供抽象。

之前该模块严格应用了 **RFC 2109** 和 **RFC 2068** 规范中描述的解析规则。后来人们发现 MSIE 3.0x 并未遵循这些规范中描述的字符规则；目前各种浏览器和服务器在处理 cookie 时也放宽了解析规则。因此，该模块目前使用的解析规则也没有以前那么严格了。

字符集*string.ascii_letters*, *string.digits* 和 `!#$%&'*+-.^_`|~:` 标明了本模块允许在 cookie 名称中出现的有效字符 (如*key*)。

在 3.3 版本发生变更: 允许':' 作为有效的 cookie 名称字符。

备注

当遇到无效 cookie 时会触发*CookieError*，所以若 cookie 数据来自浏览器，一定要做好应对无效数据的准备，并在解析时捕获*CookieError*。

exception http.cookies.**CookieError**

出现异常的原因，可能是不符合 **RFC 2109**： 属性不正确、*Set-Cookie* 头部信息不正确等等。

class http.cookies.**BaseCookie**([*input*])

类似字典的对象，字典键为字符串，字典值是*Morsel* 实例。请注意，在将键值关联时，首先会把值转换为包含键和值的*Morsel* 对象。

若给出 *input* ，将会传给*load()* 方法。

class http.cookies.**SimpleCookie**([*input*])

该类派生自*BaseCookie*并重写了*value_decode()*和*value_encode()*。SimpleCookie 支持用字符串作为 cookie 值。在设置值时，SimpleCookie 会调用内置*str()*将值转换为字符串。从 HTTP 接收的值仍然保持为字符串。

参见

***http.cookiejar* 模块**
处理网络 客户端的 HTTP cookie。*http.cookiejar*和*http.cookies*模块相互没有依赖关系。

RFC 2109 - HTTP 状态管理机制
这是本模块实现的状态管理规范。

22.18.1 Cookie 对象

BaseCookie.**value_decode**(*val*)

由字符串返回元组 (real_value, coded_value)。real_value 可为任意类型。*BaseCookie*中的此方法未实现任何解码工作——只为能被子类重写。

BaseCookie.**value_encode**(*val*)

返回元组 (real_value, coded_value)。*val* 可为任意类型，coded_value 则会转换为字符串。*BaseCookie*中的此方法未实现任何编码工作——只为能被子类重写。

通常在 *value_decode* 的取值范围内，*value_encode()*和*value_decode()*应为可互逆操作。

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)

返回可作为 HTTP 标头信息发送的字符串表示。*attrs* 和 *header* 会发给每个*Morsel*的*output()*方法。*sep* 用来将标头连接在一起，默认为 '\r\n' (CRLF) 组合。

BaseCookie.**js_output**(*attrs=None*)

返回一段可供嵌入的 JavaScript 代码，若在支持 JavaScript 的浏览器上运行，其作用如同发送 HTTP 头部信息一样。

attrs 的含义与*output()*的相同。

BaseCookie.**load**(*rawdata*)

若 *rawdata* 为字符串，则会作为 HTTP_COOKIE 进行解析，并将找到的值添加为*Morsel*。如果是字典值，则等价于:

```
for k, v in rawdata.items():
    cookie[k] = v
```

22.18.2 Morsel 对象

class http.cookies.**Morsel**

对键/值对的抽象，带有 **RFC 2109** 的部分属性。

morsel 对象类似于字典，它的键是一组常量 --- 即有效的 **RFC 2109** 属性，包括:

expires

path

comment

domain

max-age

secure

version

httponly

samesite

httponly 属性指明了该 cookie 仅在 HTTP 请求中传输，且不能通过 JavaScript 访问。这是为了减轻某些跨站脚本攻击的危害。

samesite 属性指明了浏览器不得与跨站请求一起发送该 cookie。这有助于减轻 CSRF 攻击的危害。此属性的有效值为 “Strict” 和 “Lax”。

键不区分大小写，默认值为 ''。

在 3.5 版本发生变更: 现在 `__eq__()` 会同时考虑*key* 和*value*。

在 3.7 版本发生变更: Attributes *key*, *value* and *coded_value* are read-only. Use *set()* for setting them.

在 3.8 版本发生变更: 增加对*samesite* 属性的支持。

`Morsel.`**value**
Cookie 的值。

`Morsel.`**coded_value**
编码后的 cookie 值——也即要发送的内容。

`Morsel.`**key**
cookie 名称

`Morsel.`**set**(*key*, *value*, *coded_value*)
设置 *key*、*value* 和 *coded_value* 属性。

`Morsel.`**isReservedKey**(*K*)
判断 *K* 是否属于*Morsel* 的键。

`Morsel.`**output**(*attrs=None*, *header='Set-Cookie:'*)
返回 morsel 的字符串形式，适用于作为 HTTP 头部信息进行发送。默认包含所有属性，除非给出 *attrs* 属性列表。*header* 默认为 "Set-Cookie:"。

`Morsel.`**js_output**(*attrs=None*)
返回一段可供嵌入的 JavaScript 代码，若在支持 JavaScript 的浏览器上运行，其作用如同发送 HTTP 头部信息一样。

attrs 的含义与*output()* 的相同。

`Morsel.`**OutputString**(*attrs=None*)
返回 morsel 的字符串形式，不含 HTTP 或 JavaScript 数据。

attrs 的含义与*output()* 的相同。

`Morsel.`**update**(*values*)
用字典 *values* 中的值更新 morsel 字典中的值。若有 *values* 字典中的键不是有效的 **RFC 2109** 属性，则会触发错误。

在 3.5 版本发生变更: 无效键会触发错误。

`Morsel.`**copy**(*value*)
返回 morsel 对象的浅表复制副本。

在 3.5 版本发生变更: 返回一个 morsel 对象，而非字典。

`Morsel.`**setdefault**(*key*, *value=None*)
若 key 不是有效的 **RFC 2109** 属性则触发错误，否则与*dict.setdefault()* 相同。

22.18.3 示例

以下例子演示了*http.cookies* 模块的用法。

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # 生成 HTTP 标头
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # 同样的内容
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # 从字符串加载 (HTTP 标头)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # 等价于 C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

22.19 `http.cookiejar` --- HTTP 客户端的 Cookie 处理

源代码： Lib/http/cookiejar.py

---

http.cookiejar 模块定义了用于自动处理 HTTP cookie 的类。这对访问需要小段数据——*cookies* 的网站很有用，这些数据由 Web 服务器的 HTTP 响应在客户端计算机上设置，然后在以后的 HTTP 请求中返回给服务器。

常规的 Netscape cookie 协议和由 **RFC 2965** 定义的协议都可以被处理。RFC 2965 的处理默认是关闭的。**RFC 2109** cookie 被解析为 Netscape cookie，随后根据当前使用的" 策略"，被视为 Netscape 或 RFC 2965 cookie。*http.cookiejar* 试图遵循事实上的 Netscape cookie 协议（它与原始 Netscape 规范中的协议有很大不同），包括注意到 RFC 2965 中引入的 `max-age` 和 `port` cookie 属性。

备注

在 *Set-Cookie* 和 *Set-Cookie2* 头中找到的各种命名参数通常指 *attributes*。为了不与 Python 属性相混淆，模块文档使用 *cookie-attribute* 代替。

此模块定义了以下异常：

exception http.cookiejar.**LoadError**

FileCookieJar 实例在从文件加载 cookies 出错时抛出这个异常。*LoadError* 是 *OSError* 的一个子类。

在 3.3 版本发生变更: *LoadError* 曾经是 *IOError* 的子类型，现在它是 *OSError* 的一个别名。

提供了以下类：

class http.cookiejar.**CookieJar**(*policy=None*)

policy 是实现了 *CookiePolicy* 接口的一个对象。

CookieJar 类储存 HTTP cookies。它从 HTTP 请求提取 cookies，并在 HTTP 响应中返回它们。*CookieJar* 实例在必要时自动处理包含 cookie 的到期情况。子类还负责储存和从文件或数据库中查找 cookies。

class http.cookiejar.**FileCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

policy 是实现了 *CookiePolicy* 接口的一个对象。对于其他参数，参考相应属性的文档。

一个可以从硬盘中文件加载或保存 cookie 的 *CookieJar*。Cookies **不**会在 *load()* 或 *revert()* 方法调用前从命名的文件中加载。子类的文档位于段落 *FileCookieJar 的子类及其与 Web 浏览器的协同*。

此类不应被直接初始化——请改用它的下列子类。

在 3.8 版本发生变更: 文件名形参支持 *path-like object*。

class http.cookiejar.**CookiePolicy**

此类负责确定是否应从服务器接受每个 cookie 或将其返回给服务器。

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*, *secure_protocols=('https', 'wss')*)

构造参数只能以关键字参数传递，*blocked_domains* 是一个我们既不会接受也不会返回 cookie 的域名序列。*allowed_domains* 如果不是 *None*，则是仅有的我们会接受或返回的域名序列。*secure_protocols* 是可以添加安全 cookies 的协议序列。默认将 *https* 和 *wss*（安全 WebSocket）考虑为安全协议。对于其他参数，参考 *CookiePolicy* 和 *DefaultCookiePolicy* 对象的文档。

DefaultCookiePolicy 实现了 Netscape 和 **RFC 2965** cookies 的标准接受 / 拒绝规则。默认情况下，**RFC 2109** cookies（即在 *Set-Cookie* 头中收到的 cookie-attribute 版本为 1 的 cookies）将按照 RFC 2965 规则处理。然而，如果 RFC 2965 的处理被关闭，或者 *rfc2109_as_netscape* 为 `True`，*Cookie* 实例的 `version` 属性设置将被为 0，RFC 2109 cookies *CookieJar* 实例将" 降级" 为 Netscape cookies。*DefaultCookiePolicy* 也提供一些参数以允许一些策略微调。

class http.cookiejar.**Cookie**

这个类代表 Netscape、**RFC 2109** 和 **RFC 2965** 的 cookie 。我们不希望 *http.cookiejar* 的用户构建他们自己的 *Cookie* 实例。如果有必要，请在一个 *CookieJar* 实例上调用 `make_cookies()` 。

参见

模块*urllib.request*
URL 打开带有自动的 cookie 处理。

模块*http.cookies*
HTTP cookie 类，主要是对服务端代码有用。*http.cookiejar* 和*http.cookies* 模块不相互依赖。

https://curl.se/rfc/cookie_spec.html
原始 Netscape cookie 协议的规范。虽然这仍然是主流协议，但所有主要浏览器（以及*http.cookiejar*）实现的"Netscape cookie 协议" 与 `cookie_spec.html` 中描述的协议仅有几分相似之处。

RFC 2109 - HTTP 状态管理机制
被 **RFC 2965** 所取代。使用 *Set-Cookie* version=1 。

RFC 2965 - HTTP 状态管理机制
修正了错误的 Netscape 协议。使用 *Set-Cookie2* 来代替 *Set-Cookie* 。没有广泛被使用。

https://kristol.org/cookie/errata.html
未完成的 **RFC 2965** 勘误表。

RFC 2964 - HTTP 状态管理使用方法

22.19.1 CookieJar 和 FileCookieJar 对象

CookieJar 对象支持*iterator* 协议，用于迭代包含的*Cookie* 对象。

CookieJar 有以下方法：

`CookieJar.`**`add_cookie_header`**(*request*)

在 *request* 中添加正确的 *Cookie* 头。

如果策略允许（即 `rfc2965` 和 `hide_cookie2` 属性在*CookieJar* 的*CookiePolicy* 实例中分别为 True 和 False ），*Cookie2* 标头也会在适当时候添加。

如*urllib.request* 文档所言 *request* 对象（通常是*urllib.request.Request* 的实例）必须支持 `get_full_url()`，`has_header()`，`get_header()`，`header_items()`，`add_unredirected_header()` 等方法以及 `host`，`type`，`unverifiable` 和 `origin_req_host` 等属性。

在 3.3 版本发生变更：*request* 对象需要 `origin_req_host` 属性。对已废弃的方法 `get_origin_req_host()` 的依赖已被移除。

`CookieJar.`**`extract_cookies`**(*response*, *request*)

从 HTTP *response* 中提取 cookie，并在政策允许的情况下，将它们存储在*CookieJar* 中。

CookieJar 将在 *response* 参数中寻找允许的 *Set-Cookie* 和 *Set-Cookie2* 头信息，并适当地存储 cookies（须经*CookiePolicy.set_ok()* 方法批准）。

response 对象（通常是调用*urllib.request.urlopen()* 或类似方法的结果）应该支持 `info()` 方法，它返回*email.message.Message* 实例。

如*urllib.request* 文档所言 *request* 对象（通常是*urllib.request.Request* 的实例）必须支持 `get_full_url()` 方法以及 `host`，`unverifiable` 和 `origin_req_host` 等属性。该请求被用于设置 cookie-attributes 的默认值并检查 cookie 是否允许被设置。

在 3.3 版本发生变更：*request* 对象需要 `origin_req_host` 属性。对已废弃的方法 `get_origin_req_host()` 的依赖已被移除。

`CookieJar.`**`set_policy`**(*policy*)

设置要使用的*CookiePolicy* 实例。

`CookieJar.`**`make_cookies`**(*response*, *request*)

返回从 *response* 对象中提取的*Cookie* 对象的序列。

关于 *response* 和 *request* 参数所需的接口，请参见*extract_cookies()* 的文档。

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)

如果策略规定可以这样做，就设置一个*Cookie* 。

CookieJar.**set_cookie**(*cookie*)

设置一个*Cookie*，不需要检查策略是否应该被设置。

CookieJar.**clear**([*domain*[, *path*[, *name*]]])

清除一些 cookie。

如果调用时没有参数，则清除所有的 cookie。如果给定一个参数，只有属于该 *domain* 的 cookies 将被删除。如果给定两个参数，那么属于指定的 *domain* 和 URL *path* 的 cookie 将被删除。如果给定三个参数，那么属于指定的 *domain* 、*path* 和 *name* 的 cookie 将被删除

如果不存在匹配的 cookie，则会引发*KeyError*。

CookieJar.**clear_session_cookies**()

丢弃所有的会话 cookie。

丢弃所有 `discard` 属性为真值的已包含 cookie（通常是因为它们没有 `max-age` 或 `expires` cookie 属性，或者显式的 `discard` cookie 属性）。对于交互式浏览器，会话的结束通常对应于关闭浏览器窗口。

请注意 `save()` 方法并不会保存会话的 cookie，除非你通过传入一个真值给 *ignore_discard* 参数来提出明确的要求。

FileCookieJar 实现了下列附加方法:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

将 cookie 保存到文件。

基类会引发*NotImplementedError*。子类可以继续不实现该方法。

filename 为要用来保存 cookie 的文件名称。如果未指定 *filename*，则会使用 `self.filename` (该属性默认为传给构造器的值，如果有传入的话)；如果 `self.filename` 为*None*，则会引发*ValueError*。

ignore_discard: 即使设定了丢弃 cookie 仍然保存它们。*ignore_expires*: 即使 cookie 已超期仍然保存它们

文件如果已存在则会被覆盖，这将清除其所包含的全部 cookie。已保存的 cookie 可以使用*load()* 或*revert()* 方法来恢复。

FileCookieJar.**load**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

从文件加载 cookie。

旧的 cookie 将被保留，除非是被新加载的 cookie 所覆盖。

其参数与*save()* 的相同。

指定的文件必须为该类所能理解的格式，否则将引发*LoadError*。也可能会引发*OSError*，例如当文件不存在的时候。

在 3.3 版本发生变更: 过去触发的*IOError*，现在是*OSError* 的别名。

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

清除所有 cookie 并从保存的文件重新加载 cookie。

revert() 可以引发与*load()* 相同的异常。如果执行失败，对象的状态将不会被改变。

FileCookieJar 实例具有下列公有属性:

FileCookieJar.**filename**

默认的保存 cookie 的文件的文件名。该属性可以被赋值。

FileCookieJar.**delayload**

如为真值，则惰性地从磁盘加载 cookie。该属性不应当被赋值。这只是一个提示，因为它只会影响性能，而不会影响行为（除非磁盘中的 cookie 被改变了）。*CookieJar* 对象可能会忽略它。任何包括在标准库中的*FileCookieJar* 类都不会惰性地加载 cookie。

22.19.2 FileCookieJar 的子类及其与 Web 浏览器的协同

提供了以下*CookieJar* 子类用于读取和写入。

class http.cookiejar.**MozillaCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

一个能够以 Mozilla `cookies.txt` 文件格式（该格式也被 curl 和 Lynx 以及 Netscape 浏览器所使用）从硬盘加载和存储 cookie 的*FileCookieJar*。

> **备注**
>
> 这会丢失有关 **RFC 2965** cookie 的信息，以及有关较新或非标准的 cookie 属性例如 `port`。

> **警告**
>
> 在存储之前备份你的 cookie，如果你的 cookie 丢失/损坏会造成麻烦的话（有一些微妙的因素可能导致文件在加载/保存的往返过程中发生细微的变化）。

还要注意在 Mozilla 运行期间保存的 cookie 将可能被 Mozilla 清除。

class http.cookiejar.**LWPCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

一个能够以 libwww-perl 库的 `Set-Cookie3` 文件格式从磁盘加载和存储 cookie 的*FileCookieJar*。这适用于当你想以人类可读的文件来保存 cookie 的情况。

在 3.8 版本发生变更: 文件名形参支持*path-like object*。

22.19.3 CookiePolicy 对象

实现了*CookiePolicy* 接口的对象具有下列方法:

CookiePolicy.**set_ok**(*cookie*, *request*)

返回指明是否应当从服务器接受 cookie 的布尔值。

cookie 是一个*Cookie* 实例。*request* 是一个实现了由*CookieJar.extract_cookies()* 的文档所定义的接口的对象。

CookiePolicy.**return_ok**(*cookie*, *request*)

返回指明是否应当将 cookie 返回给服务器的布尔值。

cookie 是一个*Cookie* 实例。*request* 是一个实现了*CookieJar.add_cookie_header()* 的文档所定义的接口的对象。

CookiePolicy.**domain_return_ok**(*domain*, *request*)

对于给定的 cookie 域如果不应当返回 cookie 则返回 `False`。

此方法是一种优化操作。它消除了检查每个具有特定域的 cookie 的必要性（这可能会涉及读取许多文件）。从*domain_return_ok()* 和*path_return_ok()* 返回真值并将所有工作留给*return_ok()*。

如果*domain_return_ok()* 为 cookie 域返回真值，则会为 cookie 路径调用*path_return_ok()*。在其他情况下，则不会为该 cookie 域调用*path_return_ok()* 和*return_ok()*。如果*path_return_ok()* 返回真值，则会调用*return_ok()* 并附带*Cookie* 对象本身以进行全面检查。在其他情况下，都永远不会为该 cookie 路径调用*return_ok()*。

请注意*domain_return_ok()* 会针对每个 *cookie* 域被调用，而非只针对 *request* 域。例如，该函数会针对 `".example.com"` 和 `"www.example.com"` 被调用，如果 request 域为 `"www.example.com"` 的话。对于*path_return_ok()* 也是如此。

request 参数与*return_ok()* 的文档所说明的一致。

CookiePolicy.**path_return_ok**(*path*, *request*)

对于给定的 cookie 路径如果不应当返回 cookie 返回 False。

请参阅*domain_return_ok()* 的文档。

除了实现上述方法，*CookiePolicy* 接口的实现还必须提供下列属性，指明应当使用哪种协议以及如何使用。所有这些属性都可以被赋值。

CookiePolicy.**netscape**

实现 Netscape 协议。

CookiePolicy.**rfc2965**

实现 **RFC 2965** 协议。

CookiePolicy.**hide_cookie2**

不要向请求添加 *Cookie2* 标头（此标头是提示服务器请求方能识别 **RFC 2965** cookie）。

定义*CookiePolicy* 类的最适用方式是通过子类化*DefaultCookiePolicy* 并重写部分或全部上述的方法。*CookiePolicy* 本身可被用作' 空策略' 以允许设置和接收所有的 cookie（但这没有什么用处）。

22.19.4 DefaultCookiePolicy 对象

实现接收和返回 cookie 的标准规则。

RFC 2965 和 Netscape cookie 均被涵盖。RFC 2965 处理默认关闭。

提供自定义策略的最容易方式是重写此类并在你重写的实现中添加你自己的额外检查之前调用其方法:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

在实现*CookiePolicy* 接口所要求的特性之外，该类还允许你阻止和允许特定的域设置和接收 cookie。还有一些严格性开关允许你将相当宽松的 Netscape 协议规则收紧一点（代价是可能会阻止某些无害的 cookie）。

提供了域阻止名单和允许名单（默认都是关闭的）。只有不存在于阻止列表且存在于允许列表（如果允许名单被启用）的域才能参与 cookie 的设置与返回。请使用 *blocked_domains* 构造器参数，以及 blocked_domains() 和 set_blocked_domains() 方法（以及 and the corresponding argument and methods for *allowed_domains* 的相应参数和方法）。如果你设置了允许名单，你可以通过将其设为*None* 来关闭它。

阻止名单或允许名单中不以点号开头的域名必须与要匹配的 cookie 域完全相等。例如，"example.com" 将匹配阻止名单条目 "example.com"，但不匹配 "www.example.com"。以点号开头的域名也能与更明确的域相匹配。例如，"www.example.com" 和 "www.coyote.example.com" 将匹配 ".example.com" (但不匹配 "example.com" 本身)。IP 地址不在此例，而是必须完全匹配。例如，如果 blocked_domains 包含 "192.168.1.2" 和 ".168.1.2"，则会阻止 192.168.1.2，但不会阻止 193.168.1.2。

DefaultCookiePolicy 实现了下列附加方法:

DefaultCookiePolicy.**blocked_domains**()

返回被阻止域的序列（元组类型）。

DefaultCookiePolicy.**set_blocked_domains**(*blocked_domains*)

设置被阻止域的序列。

DefaultCookiePolicy.**is_blocked**(*domain*)

如果 *domain* 在设置或接受 cookie 的阻止列表中则返回 True。

DefaultCookiePolicy.**allowed_domains**()

返回*None*，或者被允许域的序列（元组类型）。

DefaultCookiePolicy.**set_allowed_domains**(*allowed_domains*)

设置被允许域的序列，或者为*None*。

DefaultCookiePolicy.**is_not_allowed**(*domain*)

如果 *domain* 不在设置或接受 cookie 的允许列表中则返回 `True`。

DefaultCookiePolicy 实例具有下列属性，它们都是基于同名的构造器参数来初始化的，并且都可以被赋值。

DefaultCookiePolicy.**rfc2109_as_netscape**

如为真值，则请求*CookieJar* 实例将 **RFC 2109** cookie (即在带有 version 值为 1 的 cookie 属性的 *Set-Cookie* 标头中接收到的 cookie) 降级为 Netscape cookie: 即将*Cookie* 实例的 version 属性设为 0。默认值为*None*，在此情况下 RFC 2109 cookie 仅在 s are downgraded if and only if **RFC 2965** 处理被关闭时才会被降级。因此，RFC 2109 cookie 默认会被降级。

通用严格性开关:

DefaultCookiePolicy.**strict_domain**

不允许网站设置带国家码顶级域的包含两部分的域名例如 `.co.uk`, `.gov.uk`, `.co.nz` 等。此开关尚未十分完善，并不保证有效!

RFC 2965 协议严格性开关:

DefaultCookiePolicy.**strict_rfc2965_unverifiable**

遵循针对不可验证事务的 **RFC 2965** 规则（不可验证事务通常是由重定向或请求发布在其它网站的图片导致的）。如果该属性为假值，则 永远不会基于可验证性而阻止 cookie。

Netscape 协议严格性开关:

DefaultCookiePolicy.**strict_ns_unverifiable**

即便是对 Netscape cookie 也要应用 **RFC 2965** 规则。

DefaultCookiePolicy.**strict_ns_domain**

指明针对 Netscape cookie 的域匹配规则的严格程度。可接受的值见下文。

DefaultCookiePolicy.**strict_ns_set_initial_dollar**

忽略 Set-Cookie 中的 cookie: 即名称前缀为 `'{TX-PL-LABEL}#x27;` 的标头。

DefaultCookiePolicy.**strict_ns_set_path**

不允许设置路径与请求 URL 路径不匹配的 cookie。

`strict_ns_domain` 是一组旗标。其值是通过或运算来构造的（例如，`DomainStrictNoDots|DomainStrictNonDomain` 表示同时设置两个旗标）。

DefaultCookiePolicy.**DomainStrictNoDots**

当设置 cookie 是，'host prefix' 不可包含点号（例如 `www.foo.bar.com` 不能为 `.bar.com` 设置 cookie，因为 `www.foo` 包含了一个点号）。

DefaultCookiePolicy.**DomainStrictNonDomain**

没有显式指明 Cookies that did not explicitly specify a `domain` cookie 属性的 cookie 只能被返回给与设置 cookie 的域相同的域（例如 `spam.example.com` 不会是来自 `example.com` 的返回 cookie，如果该域名没有 `domain` cookie 属性的话）。

DefaultCookiePolicy.**DomainRFC2965Match**

当设置 cookie 时，要求完整的 **RFC 2965** 域匹配。

下列属性是为方便使用而提供的，是上述旗标的几种最常用组合:

DefaultCookiePolicy.**DomainLiberal**

等价于 0 (即所有上述 Netscape 域严格性旗标均停用)。

DefaultCookiePolicy.**DomainStrict**

等价于 DomainStrictNoDots|DomainStrictNonDomain。

22.19.5 Cookie 对象

Cookie 实例具有与各种 cookie 标准中定义的标准 cookie 属性大致对应的 Python 属性。这并非一一对应，因为存在复杂的赋默认值的规则，因为 max-age 和 expires cookie 属性包含相同信息，也因为 **RFC 2109** cookie 可以被*http.cookiejar* 从第 1 版' 降级为' 第 0 版 (Netscape) cookie。

对这些属性的赋值在*CookiePolicy* 方法的极少数情况以外应该都是不必要的。该类不会强制内部一致性，因此如果这样做则你应当清楚自己在做什么。

Cookie.**version**

整数或*None*。Netscape cookie 的*version* 值为 0。**RFC 2965** 和 **RFC 2109** cookie 的 version cookie 属性值为 1。但是，请注意*http.cookiejar* 可以将 RFC 2109 cookie ' 降级' 为 Netscape cookie，在此情况下*version* 值也为 0。

Cookie.**name**

Cookie 名称（一个字符串）。

Cookie.**value**

Cookie 值（一个字符串），或为*None*。

Cookie.**port**

代表一个端口或一组端口（例如'80' 或'80,8080'）的字符串，或为*None*。

Cookie.**domain**

Cookie 域（一个字符串）。

Cookie.**path**

Cookie 路径 (字符串类型，例如 '/acme/rocket_launchers')。

Cookie.**secure**

如果 cookie 应当只能通过安全连接返回则为 True。

Cookie.**expires**

整数类型的过期时间，以距离 Unix 纪元的秒数表示，或者为*None*。另请参阅*is_expired()* 方法。

Cookie.**discard**

如果是会话 cookie 则为 True。

Cookie.**comment**

来自服务器的解释此 cookie 功能的字符串形式的注释，或者为*None*。

Cookie.**comment_url**

链接到来自服务器的解释此 cookie 功能的注释的 URL，或者为*None*。

Cookie.**rfc2109**

如果 cookie 是作为 **RFC 2109** cookie 被接收（即该 cookie 是在 *Set-Cookie* 标头中送达，且该标头的 Version cookie 属性的值为 1）则为 True。之所以要提供该属性是因为*http.cookiejar* 可能会从 RFC 2109 cookies ' 降级' 为 Netscape cookie，在此情况下*version* 值为 0。

Cookie.**port_specified**

如果服务器显式地指定了一个端口或一组端口（在 *Set-Cookie* / *Set-Cookie2* 标头中）则为 True。

Cookie.**domain_specified**

如果服务器显式地指定了一个域则为 True。

Cookie.**domain_initial_dot**

该属性为 True 表示服务器显式地指定了以一个点号 ('.') 打头的域。

Cookie 可能还有额外的非标准 cookie 属性。这些属性可以通过下列方法来访问:

Cookie.**has_nonstandard_attr**(*name*)

如果 cookie 具有相应名称的 cookie 属性则返回 True。

Cookie.**get_nonstandard_attr**(*name*, *default=None*)

如果 cookie 具有相应名称的 cookie 属性，则返回其值。否则，返回 *default*。

Cookie.**set_nonstandard_attr**(*name*, *value*)

设置指定名称的 cookie 属性的值。

Cookie 类还定义了下列方法:

Cookie.**is_expired**(*now=None*)

如果 cookie 传入了服务器请求其所应过期的时间则为 True。如果给出 *now* 值（距离 Unix 纪元的秒数），则返回在指定的时间 cookie 是否已过期。

22.19.6 例子

第一个例子显示了*http.cookiejar* 的最常见用法:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

这个例子演示了如何使用你的 Netscape, Mozilla 或 Lynx cookie 打开一个 URL (假定 cookie 文件位置采用 Unix/Netscape 惯例):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

下一个例子演示了*DefaultCookiePolicy* 的使用。启用 **RFC 2965** cookie，在设置和返回 Netscape cookie 时更严格地限制域，以及阻止某些域设置 cookie 或返回它们:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

22.20 xmlrpc --- XMLRPC 服务端与客户端模块

XML-RPC 是一种远程过程调用方法，它使用通过 HTTP 传递的 XML 作为载体。有了它，客户端可以在远程服务器上调用带参数的方法（服务器以 URI 命名）并获取结构化的数据。

xmlrpc 是一个集合了 XML-RPC 服务端与客户端实现模块的包。这些模块是:

- *xmlrpc.client*
- *xmlrpc.server*

22.21 xmlrpc.client --- XML-RPC 客户端访问

源代码: Lib/xmlrpc/client.py

XML-RPC 是一种远程过程调用方法，它以使用 HTTP(S) 传递的 XML 作为载体。通过它，客户端可以在远程服务器（服务器以 URI 指明）上调用带参数的方法并获取结构化的数据。本模块支持编写 XML-RPC 客户端代码；它会处理在通用 Python 对象和 XML 之间进行在线翻译的所有细节。

警告

xmlrpc.client 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

在 3.5 版本发生变更: 对于 HTTPS URI，现在*xmlrpc.client* 默认会执行所有必要的证书和主机名检查。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *headers=()*, *context=None*)

ServerProxy 实例是管理与远程 XML-RPC 服务器通信的对象。要求的第一个参数为 URI (统一资源定位符)，通常就是服务器的 URL。可选的第二个参数为传输工厂实例；在默认情况下对于 https: URL 是一个内部 `SafeTransport` 实例，在其他情况下则是一个内部 HTTP `Transport` 实例。可选的第三个参数为编码格式，默认为 UTF-8。可选的第四个参数为调试旗标。

下列形参控制所返回代理实例的使用。如果 *allow_none* 为真值，则 Python 常量 `None` 将被转写至 XML；默认行为是针对 `None` 引发*TypeError*。这是对 XML-RPC 规格的一个常用扩展，但并不被所有客户端和服务器所支持；请参阅 http://ontosys.com/xml-rpc/extensions.php 了解详情。*use_builtin_types* 旗标可被用来将日期/时间值表示为*datetime.datetime* 对象而将二进制数据表示为*bytes* 对象；此旗标默认为假值。*datetime.datetime*, *bytes* 和*bytearray* 对象可以被传给调用操作。*headers* 形参为可选的随每次请求发送的 HTTP 标头序列，其形式为包含代表标头名称和值的 2 元组序列。(例如 `[('Header-Name', 'value')]`)。如果提供了 HTTPS URL，则 *context* 可以为*ssl.SSLContext* 并配置底层 HTTPS 连接的 SSL 设置。已过时的 *use_datetime* 旗标与 *use_builtin_types* 类似但它只适用于日期/时间值。

在 3.3 版本发生变更: 增加了 *use_builtin_types* 旗标。

在 3.8 版本发生变更: 增加了 *headers* 形参。

HTTP 和 HTTPS 传输均支持用于 HTTP 基本身份验证的 URL 语法扩展: `http://user:pass@host:port/path`。`user:pass` 部分将以 base64 编码为 HTTP 'Authorization' 标头，并在唤起 XML-RPC 方法时作为连接过程的一部分发送给远程服务器。你只需要在远程服务器要求基本身份验证用户名和密码时使用此语法。

返回的实例是一个代理对象，具有可被用来在远程服务器上发起相应 RPC 调用的方法。如果远程服务器支持内省 API，则也可使用该代理对象在远程服务器上查询它所支持的方法（服务发现）并获取其他服务器相关的元数据

适用的类型（即可通过 XML 生成 marshall 对象），包括如下类型（除了已说明的例外，它们都会被反 marshall 为同样的 Python 类型）：

XML-RPC 类型	Python 类型
boolean	*bool*
int, i1, i2, i4, i8 或者 biginteger	*int* 的范围从 -2147483648 到 2147483647。值将获得 <int> 标志。
double 或 float	*float*。值将获得 <double> 标志。
string	*str*
array	*list* 或 *tuple* 包含整合元素。数组以 *lists* 形式返回。
struct	*dict*。键必须为字符串，值可以为任何适用的类型。可以传入用户自定义类的对象；只有其 __dict__ 属性会被传输。
dateTime.iso8601	*DateTime* 或 *datetime.datetime*。返回的类型取决于 *use_builtin_types* 和 *use_datetime* 标志的值。
base64	*Binary*, *bytes* 或 *bytearray*。返回的类型取决于 *use_builtin_types* 标志的值。
nil	None 常量。仅当 *allow_none* 为 true 时才允许传递。
bigdecimal	*decimal.Decimal*. 仅返回类型。

这是 This is the full set of data types supported by XML-RPC 所支持数据类型的完整集合。方法调用也可能引发一个特殊的 *Fault* 实例，用来提示 XML-RPC 服务器错误，或是用 *ProtocolError* 来提示 HTTP/HTTPS 传输层中的错误。*Fault* 和 *ProtocolError* 都派生自名为 Error 的基类。请注意 xmlrpc client 模块目前不可 marshal 内置类型的子类的实例。

当传入字符串时，XML 中的特殊字符如 <, > 和 & 将被自动转义。但是，调用方有责任确保字符串中没有 XML 中不允许的字符，例如 ASCII 值在 0 和 31 之间的控制字符（当然，制表、换行和回车除外）；不这样做将导致 XML-RPC 请求的 XML 格式不正确。如果你必须通过 XML-RPC 传入任意字节数据，请使用 *bytes* 或 *bytearray* 类或者下文描述的 *Binary* 包装器类。

Server 被保留作为 *ServerProxy* 的别名用于向下兼容。新的代码应当使用 *ServerProxy*。

在 3.5 版本发生变更: 增加了 *context* 参数。

在 3.6 版本发生变更: 增加了对带有前缀的类型标签的支持 (例如 ex:nil)。增加了对反 marshall 被 Apache XML-RPC 实现用于表示数值的附加类型的支持: i1, i2, i8, biginteger, float 和 bigdecimal。请参阅 https://ws.apache.org/xmlrpc/types.html 了解详情。

参见

XML-RPC HOWTO
以多种语言对 XML-RPC 操作和客户端软件进行了很好的说明。包含 XML-RPC 客户端开发者所需知道的几乎任何事情。

XML-RPC Introspection
描述了用于内省的 XML-RPC 协议扩展。

XML-RPC Specification
官方规范说明。

22.21.1 ServerProxy 对象

ServerProxy 实例有一个方法与 XML-RPC 服务器所接受的每个远程过程调用相对应。调用该方法会执行一个 RPC，通过名称和参数签名来调度（例如同一个方法名可通过多个参数签名来重载）。RPC 结束时返回一个值，它可以是适用类型的返回数据或是表示错误的 *Fault* 或 *ProtocolError* 对象。

支持 XML 内省 API 的服务器还支持一些以保留的 system 属性分组的通用方法:

ServerProxy.system.**listMethods**()
此方法返回一个字符串列表，每个字符串都各自对应 XML-RPC 服务器所支持的（非系统）方法。

ServerProxy.system.**methodSignature**(*name*)

此方法接受一个形参，即某个由 XML-RPC 服务器所实现的方法名称。它返回一个由此方法可能的签名组成的数组。一个签名就是一个类型数组。这些类型中的第一个是方法的的返回类型，其余的均为形参。

由于允许多个签名（即重载），此方法是返回一个签名列表而非一个单例。

签名本身被限制为一个方法所期望的最高层级形参。举例来说如果一个方法期望有一个结构体数组作为形参，并返回一个字符串，则其签名就是”string, array”。如果它期望有三个整数并返回一个字符串，则其签名是”string, int, int, int”。

如果方法没有定义任何签名，则将返回一个非数组值。在 Python 中这意味着返回值的类型为列表以外的类型。

ServerProxy.system.**methodHelp**(*name*)

此方法接受一个形参，即 XML-RPC 服务器所实现的某个方法的名称。它返回描述相应方法用法的文档字符串。如果没有可用的文档字符串，则返回空字符串。文档字符串可以包含 HTML 标记。

在 3.5 版本发生变更: *ServerProxy* 的实例支持*context manager* 协议用于关闭下层传输。

以下是一个可运行的示例。服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

22.21.2 DateTime 对象

class xmlrpc.client.**DateTime**

该类的初始化可以使用距离 Unix 纪元的秒数、时间元组、ISO 8601 时间/日期字符串或*datetime.datetime* 实例。它具有下列方法，主要是为 marshall 和反 marshall 代码的内部使用提供支持:

decode(*string*)

接受一个字符串作为实例的新时间值。

encode(*out*)

将此*DateTime* 条目的 XML-RPC 编码格式写入到 *out* 流对象。

它还通过 富比较和 __repr__() 方法来支持特定的 Python 内置运算符。

以下是一个可运行的示例。服务器端代码:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
```

(续下页)

(接上页)

```
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# 将 ISO8601 字符串转换为日期时间对象
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

22.21.3 Binary 对象

class xmlrpc.client.**Binary**

该类的初始化可以使用字节数据（可包括 NUL）。对*Binary* 对象的初始访问是由一个属性来提供的:

data

被*Binary* 实例封装的二进制数据。该数据以*bytes* 对象的形式提供。

Binary 对象具有下列方法，支持这些方法主要是供 marshall 和反 marshall 代码在内部使用:

decode(*bytes*)

接受一个 base64 *bytes* 对象并将其解码为实例的新数据。

encode(*out*)

将此二进制条目的 XML-RPC base 64 编码格式写入到 *out* 流对象。

被编码数据将依据 **RFC 2045 第 6.8 节** 每 76 个字符换行一次，这是撰写 XML-RPC 规范说明时 base64 规范的事实标准。

它还通过 `__eq__()` 和 `__ne__()` 方法来支持特定的 Python 内置运算符。

该二进制对象的示例用法。我们将通过 XMLRPC 来传输一张图片:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

客户端会获取图片并将其保存为一个文件:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
```

(续下页)

(接上页)

```
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

22.21.4 Fault 对象

class xmlrpc.client.**Fault**

Fault 对象封装了 XML-RPC fault 标签的内容。Fault 对象具有下列属性:

faultCode

一个指明 fault 类型的整数。

faultString

一个包含与 fault 相关联的诊断消息的字符串。

在接下来的示例中我们将通过返回一个复数类型的值来故意引发一个*Fault*。服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer

# 将发生 marshall 操作错误因为我们将返回一个复数
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

22.21.5 ProtocolError 对象

class xmlrpc.client.**ProtocolError**

ProtocolError 对象描述了下层传输层中的协议错误（例如当 URI 所指定的服务器不存在时的 404 'not found' 错误）。它具有下列属性:

url

触发错误的 URI 或 URL。

errcode

错误代码。

errmsg

错误消息或诊断字符串。

headers

一个包含触发错误的 HTTP/HTTPS 请求的标头的字典。

在接下来的示例中我们将通过提供一个无效的 URI 来故意引发一个*ProtocolError*:

```
import xmlrpc.client

# 创建一个 ServerProxy，所用 URI 不与 XMLRPC 请求对应
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

22.21.6 MultiCall 对象

MultiCall 对象提供了一种将对远程服务器的多个调用封装为一个单独请求的方式[1]。

class xmlrpc.client.**MultiCall**(*server*)

创建一个用于盒式方法调用的对象。*server* 是调用的最终目标。可以针对结果对象被唤起，但它们将立即返回 None，并只在*MultiCall* 对象中存储调用名称和形参。调用该对象本身会导致所有已存储的调用作为一个单独的 system.multicall 请求被发送。此调用的结果是一个*generator*；迭代这个生成器会产生各个结果。

以下是该类的用法示例。服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# 一个带有简单算术函数的简单服务器
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
```

(续下页)

[1] 此做法被首次提及是在 a discussion on xmlrpc.com。

(接上页)

```
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

22.21.7 便捷函数

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

请 *params* 转换为一个 XML-RPC 请求。或者当 *methodresponse* 为真值时则转换为一个请求。*params* 可以是一个参数元组或是一个*Fault* 异常类的实例。如果 *methodresponse* 为真值，只有单独的值可以被返回，这意味着 *params* 的长度必须为 1。如果提供了 *encoding*，则在生成的 XML 会使用该编码格式；默认的编码格式为 UTF-8。Python 的*None* 值不可在标准 XML-RPC 中使用；要通过扩展来允许使用它，请为 *allow_none* 提供一个真值。

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

将一个 XML-RPC 请求或响应转换为 Python 对象 `(params, methodname)`。*params* 是一个参数元组；*methodname* 是一个字符串，或者如果数据包没有提供方法名则为 `None`。如果 XML-RPC 数据包是代表一个故障条件，则此函数将引发一个*Fault* 异常。*use_builtin_types* 旗标可被用于将日期/时间值表示为*datetime.datetime* 对象并将二进制数据表示为*bytes* 对象；此旗标默认为假值。

已过时的 *use_datetime* 旗标与 *use_builtin_types* 类似但只作用于日期/时间值。

在 3.3 版本发生变更: 增加了 *use_builtin_types* 旗标。

22.21.8 客户端用法的示例

```
# 简单的测试程序（来自 XML-RPC 规范说明）
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # 本地服务器
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

要通过 HTTP 代理访问一个 XML-RPC 服务器，你必须自行定义一个传输。下面的例子演示了具体做法:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
```

(续下页)

(接上页)

```
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

22.21.9 客户端与服务器用法的示例

参见*SimpleXMLRPCServer 示例*。

备注

22.22 `xmlrpc.server` --- 基本 XML-RPC 服务器

源代码: Lib/xmlrpc/server.py

`xmlrpc.server` 模块为以 Python 编写 XML-RPC 服务器提供了一个基本服务器框架。服务器可以是独立的，使用`SimpleXMLRPCServer`，或是嵌入某个 CGI 环境中，使用`CGIXMLRPCRequestHandler`。

警告

`xmlrpc.server` 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

创建一个新的服务器实例。这个类提供了一些用来注册可以被 XML-RPC 协议所调用的函数的方法。*requestHandler* 形参应该是一个用于请求处理器实例的工厂函数；它默认为`SimpleXMLRPCRequestHandler`。*addr* 和 *requestHandler* 形参会被传给`socketserver.TCPServer` 构造器。如果 *logRequests* 为真值（默认），请求将被记录到日志；将此形参设为假值将关闭日志记录。*allow_none* 和 *encoding* 形参会被传给`xmlrpc.client` 并控制将从服务器返回的 XML-RPC 响应。*bind_and_activate* 形参控制 `server_bind()` 和 `server_activate()` 是否会被构造器立即调用；它默认为真值。将其设为假值将允许代码在地址被绑定之前操作 *allow_reuse_address* 类变量。*use_builtin_types* 形参会被传给`loads()` 函数并控制当收到日期/时间值或二进制数据时将处理哪些类型；它默认为假值。

在 3.3 版本发生变更: 增加了 *use_builtin_types* 旗标。

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

创建一个新的实例来处理 CGI 环境中的 XML-RPC 请求。*allow_none* 和 *encoding* 形参会被传递给`xmlrpc.client` 并控制将要从服务器返回的 XML-RPC 响应。*use_builtin_types* 形参会被传递给`loads()` 函数并控制当接收到日期/时间值或二进制数据时要处理哪种类型；该形参默认为假值。

在 3.3 版本发生变更: 增加了 *use_builtin_types* 旗标。

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

创建一个新的请求处理器实例。该请求处理器支持 `POST` 请求并会修改日志记录操作以便使用传递给`SimpleXMLRPCServer` 构造器形参的 *logRequests* 形参。

22.22.1 SimpleXMLRPCServer 对象

SimpleXMLRPCServer 类是基于*socketserver.TCPServer* 并提供了一个创建简单、独立的 XML-RPC 服务器的方式。

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)

注册一个可以响应 XML-RPC 请求的函数。如果给出了 *name*，它将成为与 *function* 相关联的方法名，否则将使用 `function.__name__`。*name* 是一个字符串，并可能包含不能用于 Python 标识符的字符，包括句点符等。

此方法也可用作装饰器。当被用作装饰器时，*name* 只能被指定为以 *name* 注册的 *function* 的一个关键字参数。如果没有指定 *name*，则将使用 `function.__name__`。

在 3.7 版本发生变更: *register_function()* 可被用作装饰器。

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

注册一个被用来公开未使用*register_function()* 注册的方法名的对象。如果 *instance* 包含 `_dispatch()` 方法，它将被调用并附带被请求的方法名和来自请求的形参。它的 API 是 `def _dispatch(self, method, params)`（请注意 *params* 并不表示变量参数列表）。如果它调用了一个下层函数来执行任务，该函数将以 `func(*params)` 的形式被调用，即扩展了形参列表。来自 `_dispatch()` 的返回值将作为结果被返回给客户端。如果 *instance* 不包含 `_dispatch()` 方法，则会在其中搜索与被请求的方法名相匹配的属性。

如果可选的 *allow_dotted_names* 参数为真值且该实例没有 `_dispatch()` 方法，则如果被请求的方法名包含句点符，会单独搜索该方法名的每个组成部分，其效果就是执行了简单的分层级搜索。通过搜索找到的值将随即附带来自请求的形参被调用，并且返回值会被回传给客户端。

> **警告**
>
> 启用 *allow_dotted_names* 选项将允许入侵者访问你的模块的全局变量并可能允许入侵者在你的机器上执行任意代码。仅可在安全、封闭的网络中使用此选项。

SimpleXMLRPCServer.**register_introspection_functions**()

注册 XML-RPC 内省函数 `system.listMethods`, `system.methodHelp` 和 `system.methodSignature`。

SimpleXMLRPCServer.**register_multicall_functions**()

注册 XML-RPC 多调用函数 system.multicall。

SimpleXMLRPCRequestHandler.**rpc_paths**

一个必须为元组类型的属性值，其中列出所接收 XML-RPC 请求的有效路径部分。发送到其他路径的请求将导致 404 "no such page" HTTP 错误。如果此元组为空，则所有路径都将被视为有效。默认值为 `('/', '/RPC2')`。

SimpleXMLRPCServer 示例

服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# 限制为特定的路径。
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# 创建服务器
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()
```

(续下页)

(接上页)

```
    # 注册 pow() 函数；这将使用 pow.__name__ 的值
    # 作为名称，即 'pow'。
    server.register_function(pow)

    # 以不同的名称注册一个函数
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # 注册一个实例；该实例的所有方法将发布为
    # XML-RPC 方法 (在本例中，即为 'mul')。
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # 运行服务器的主循环
    server.serve_forever()
```

以下客户端代码将调用上述服务器所提供的方法:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # 返回 2**3 = 8
print(s.add(2,3))  # 返回 5
print(s.mul(5,2))  # 返回 5*2 = 10

# 打印可用方法的列表
print(s.system.listMethods())
```

`register_function()` 也可被用作装饰器。上述服务器端示例可以通过装饰器方式来注册函数:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # 注册 pow() 函数；这将使用 pow.__name__ 的值
    # 作为名称，该值即为 'pow'.
    server.register_function(pow)

    # 以不同的名称注册一个函数，
    # 使用 register_function 作为装饰器。
    # *name* 只能作为关键字参数给出。
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # 以 function.__name__ 为名称注册一个函数。
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

以下包括在 Lib/xmlrpc/server.py 模块中的例子演示了一个允许带点号名称并注册有多调用函数的服务器。

> **警告**
>
> 启用 *allow_dotted_names* 选项将允许入侵者访问你的模块的全局变量并可能允许入侵者在你的机器上执行任意代码。仅可在安全、封闭的网络中使用此示例。

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

这个 ExampleService 演示程序可通过命令行来唤起:

```
python -m xmlrpc.server
```

可与上述服务器进行交互的客户端包括在 Lib/xmlrpc/client.py 中:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

这个可与示例 XMLRPC 服务器进行交互的客户端的启动方式如下:

```
python -m xmlrpc.client
```

22.22.2 CGIXMLRPCRequestHandler

CGIXMLRPCRequestHandler 类可被用来处理发送给 Python CGI 脚本的 XML-RPC 请求。

CGIXMLRPCRequestHandler.**register_function**(*function=None*, *name=None*)

注册一个可以响应 XML-RPC 请求的函数。如果给出了 *name*，它将成为与 *function* 相关联的方法名，否则将使用 `function.__name__`。*name* 是一个字符串，并可能包含不能用于 Python 标识符的字符，包括句点符等。

此方法也可用作装饰器。当被用作装饰器时，*name* 只能被指定为以 *name* 注册的 *function* 的一个关键字参数。如果没有指定 *name*，则将使用 `function.__name__`。

在 3.7 版本发生变更: *register_function()* 可被用作装饰器。

CGIXMLRPCRequestHandler.**register_instance**(*instance*)

注册一个对象用来公开未使用*register_function()* 进行注册的方法名。如果实例包含 `_dispatch()` 方法，它会附带所请求的方法名和来自请求的形参被调用；返回值会作为结果被返回给客户端。如果实例不包含 `_dispatch()` 方法，则在其中搜索与所请求方法名相匹配的属性；如果所请求方法名包含句点，则会分别搜索方法名的每个部分，其效果就是执行了简单的层级搜索。搜索找到的值将附带来自请求的形参被调用，其返回值会被返回给客户端。

CGIXMLRPCRequestHandler.**register_introspection_functions**()

注册 XML-RPC 内海函数 `system.listMethods`, `system.methodHelp` 和 `system.methodSignature`。

CGIXMLRPCRequestHandler.**register_multicall_functions**()

注册 XML-RPC 多调用函数 `system.multicall`。

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)

处理一个 XML-RPC 请求。如果给出了 *request_text*，它应当是 HTTP 服务器所提供的 POST 数据，否则将使用 stdin 的内容。

示例:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

22.22.3 文档 XMLRPC 服务器

这些类扩展了上面的类以发布响应 HTTP GET 请求的 HTML 文档。服务器可以是独立的，使用*DocXMLRPCServer*，或是嵌入某个 CGI 环境中，使用*DocCGIXMLRPCRequestHandler*。

class xmlrpc.server.**DocXMLRPCServer**(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)

创建一个新的服务器实例。所有形参的含义与*SimpleXMLRPCServer* 的相同；*requestHandler* 默认为*DocXMLRPCRequestHandler*。

在 3.3 版本发生变更: 增加了 *use_builtin_types* 旗标。

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**

创建一个新的实例来处理 CGI 环境中的 XML-RPC 请求。

class xmlrpc.server.**DocXMLRPCRequestHandler**

创建一个新的请求处理器实例。该请求处理器支持 XML-RPC POST 请求、文档 GET 请求并会修改日志记录操作以便使用传递给*DocXMLRPCServer* 构造器形参的 *logRequests* 形参。

22.22.4 DocXMLRPCServer 对象

DocXMLRPCServer 类派生自*SimpleXMLRPCServer* 并提供了一种创建自动记录文档的、独立的 XML-RPC 服务器的方式。HTTP POST 请求将作为 XML-RPC 方法调用来处理。HTTP GET 请求将通过生成 pydoc 风格的 HTML 文档来处理。这将允许服务器自己提供基于 Web 的文档。

DocXMLRPCServer.**set_server_title**(*server_title*)

设置所生成 HTML 文档要使用的标题。此标题将在 HTML ”title” 元素中使用。

DocXMLRPCServer.**set_server_name**(*server_name*)

设置所生成 HTML 文档要使用的名称。此名称将出现在所生成文档顶部的”h1” 元素中。

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

设置所生成 Set the description used in the generated HTML 文档要使用的描述。此描述将显示为文档中的一个段落，位于服务器名称之下。

22.22.5 DocCGIXMLRPCRequestHandler

DocCGIXMLRPCRequestHandler 类派生自*CGIXMLRPCRequestHandler* 并提供了一种创建自动记录文档的 XML-RPC CGI 脚本的方式。HTTP POST 请求将作为 XML-RPC 方法调用来处理。HTTP GET 请求将通过生成 pydoc 风格的 HTML 文档来处理。这将允许服务器自己提供基于 Web 的文档。

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)

设置所生成 HTML 文档要使用的标题。此标题将在 HTML ”title” 元素中使用。

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)

设置所生成 HTML 文档要使用的名称。此名称将出现在所生成文档顶部的”h1” 元素中。

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)

设置所生成 Set the description used in the generated HTML 文档要使用的描述。此描述将显示为文档中的一个段落，位于服务器名称之下。

22.23 ipaddress --- IPv4/IPv6 操作库

源代码： Lib/ipaddress.py

---

ipaddress 提供了创建、处理和操作 IPv4 和 IPv6 地址和网络的功能。

该模块中的函数和类可以直接处理与 IP 地址相关的各种任务，包括检查两个主机是否在同一个子网中，遍历某个子网中的所有主机，检查一个字符串是否是一个有效的 IP 地址或网络定义等等。

这是完整的模块 API 参考—若要查看概述，请见 ipaddress-howto.

Added in version 3.3.

22.23.1 方便的工厂函数

ipaddress 模块提供来工厂函数来方便地创建 IP 地址，网络和接口：

ipaddress.**ip_address**(*address*)

返回一个*IPv4Address* 或*IPv6Address* 对象，取决于作为参数传递的 IP 地址。可以提供 IPv4 或 IPv6 地址，小于 `2**32` 的整数默认被认为是 IPv4。如果 *address* 不是有效的 IPv4 或 IPv6 地址，则会抛出*ValueError*。

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address*, *strict=True*)

返回一个*IPv4Network* 或*IPv6Network* 对象，具体取决于作为参数传入的 IP 地址。*address* 是表示 IP 网址的字符串或整数。可以提供 IPv4 或 IPv6 网址；小于 `2**32` 的整数默认被视为 IPv4。*strict* 会被传给*IPv4Network* 或*IPv6Network* 构造器。如果 *address* 不表示有效的 IPv4 或 IPv6 网址，或者网络设置了 host 比特位，则会引发*ValueError*。

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)

返回一个*IPv4Interface* 或*IPv6Interface* 对象，取决于作为参数传递的 IP 地址。*address* 是代表 IP 地址的字符串或整数。可以提供 IPv4 或 IPv6 地址，小于 `2**32` 的整数默认认为是 IPv4。如果 *address* 不是有效的 IPv4 或 IPv6 地址，则会抛出一个*ValueError*。

这些方便的函数的一个缺点是需要同时处理 IPv4 和 IPv6 格式，这意味着提供的错误信息并不精准，因为函数不知道是打算采用 IPv4 还是 IPv6 格式。更详细的错误报告可以通过直接调用相应版本的类构造函数来获得。

22.23.2 IP 地址

地址对象

IPv4Address 和*IPv6Address* 对象有很多共同的属性。一些只对 IPv6 地址有意义的属性也在*IPv4Address* 对象实现，以便更容易编写正确处理两种 IP 版本的代码。地址对象是可哈希的*hashable*，所以它们可以作为字典中的键来使用。

class ipaddress.**IPv4Address**(*address*)

构造一个 IPv4 地址。如果 *address* 不是一个有效的 IPv4 地址，会抛出*AddressValueError* 。

以下是有效的 IPv4 地址：

1. 以十进制小数点表示的字符串，由四个十进制整数组成，范围为 0--255，用点隔开 (例如 `192.168.0.1`)。每个整数代表地址中的八位（一个字节）。不允许使用前导零，以免与八进制表示产生歧义。
2. 一个 32 位可容纳的整数。
3. 一个长度为 4 的封装在*bytes* 对象中的整数 (高位优先)。

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

在 3.8 版本发生变更: 前导零可被接受，即使是在可能与八进制表示混淆的情况下也会被接受。

在 3.9.5 版本发生变更: 前导零不再被接受，并且会被视作错误。IPv4 地址字符串现在严格按照 glibc 的*inet_pton()* 函数进行解析。

version

合适的版本号：IPv4 为 `4` ，IPv6 为 `6` 。

max_prefixlen

在该版本的地址表示中，比特数的总数：IPv4 为 `32` ；IPv6 为 `128` 。

前缀定义了地址中的前导位数量，通过比较来确定一个地址是否是网络的一部分。

compressed

exploded

以点符号分割十进制表示的字符串。表示中不包括前导零。

由于 IPv4 没有为八位数设为零的地址定义速记符号，这两个属性始终与 IPv4 地址的 `str(addr)` 相同。暴露这些属性使得编写能够处理 IPv4 和 IPv6 地址的显示代码变得更加容易。

packed

这个地址的二进制表示——一个适当长度的*bytes*对象（最高的八位在最前）。对于 IPv4 来说是 4 字节，对于 IPv6 来说是 16 字节。

reverse_pointer

IP 地址的反向 DNS PTR 记录的名称，例如:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

这是可用于执行 PTR 查询的名称，而不是已解析的主机名本身。

Added in version 3.5.

is_multicast

如果该地址被保留用作多播用途，返回 `True` 。关于多播地址，请参见 **RFC 3171** （IPv4）和 **RFC 2373** （IPv6）。

is_private

如果地址被 iana-ipv4-special-registry (针对 IPv4) 或 iana-ipv6-special-registry (针对 IPv6) 定义为非全局可用则为 `True`，例外情况如下：

- 对于共享地址空间 (`100.64.0.0/10`) `is_private` 为 `False`
- 对于映射 IPv4 的 IPv6 地址来说 `is_private` 值由下层 IPv4 地址的语义决定并且以下条件将保持成立 (见*IPv6Address.ipv4_mapped*):

  ```
  address.is_private == address.ipv4_mapped.is_private
  ```

`is_private` 具有与*is_global* 相反的值，除了对于共享的地址空间（即 `100.64.0.0/10` 范围）来说它们均为 `False`。

在 3.13 版本发生变更: 修复了一些错误的正值和错误的负值。

- `192.0.0.0/24` 被视为私有但 `192.0.0.9/32` 和 `192.0.0.10/32` 除外（之前：只有 `192.0.0.0/29` 子范围被视为私有）。
- `64:ff9b:1::/48` 被视为私有。
- `2002::/16` 被视为私有。
- 在 `2001::/23` 中存在例外（其余范围被视为私有）：`2001:1::1/128`, `2001:1::2/128`, `2001:3::/32`, `2001:4:112::/48`, `2001:20::/28`, `2001:30::/28`。这些例外不被视为私有。

is_global

如果地址被 iana-ipv4-special-registry (针对 IPv4) 或 iana-ipv6-special-registry (针对 IPv6) 定义为全局可用则为 `True`，例外情况如下：

对于映射 IPv4 的 IPv6 地址来说 `is_private` 值由下层 IPv4 地址的语义决定并且以下条件将保持成立 (见*IPv6Address.ipv4_mapped*):

```
address.is_global == address.ipv4_mapped.is_global
```

is_global 具有与*is_private* 相反的值，除了对于共享的地址空间（即 100.64.0.0/10 范围）来说它们均为 False。

Added in version 3.4.

在 3.13 版本发生变更: 修复了一些错误的正值和错误的负值，请参阅*is_private* 了解详情。

is_unspecified

当地址未指定时为 True 。参见 **RFC 5735** (IPv4) 或 **RFC 2373** (IPv6).

is_reserved

如果该地址属于互联网工程任务组（IETF）所规定的其他保留地址，返回 True 。

is_loopback

如果该地址为一个回环地址，返回 True 。关于回环地址，请见 **RFC 3330** （IPv4）和 **RFC 2373** （IPv6）。

is_link_local

如果该地址被保留用于本地链接则为 True。参见 **RFC 3927**。

ipv6_mapped

IPv4Address 对象代表已映射 IPv4 的 IPv6 地址。参见 **RFC 4291**。

Added in version 3.13.

IPv4Address.**__format__**(*fmt*)

返回一个 IP 地址的字符串表示，由一个明确的格式字符串控制。*fmt* 可以是以下之一: 's'，默认选项，相当于*str()*，'b' 用于零填充的二进制字符串，'X' 或者 'x' 用于大写或小写的十六进制表示，或者 'n' 相当于 'b' 用于 IPv4 地址和 'x' 用于 IPv6 地址。对于二进制和十六进制表示法，可以使用形式指定器 '#' 和分组选项 '_'。__format__ 被 format、str.format 和 f 字符串使用。

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

Added in version 3.9.

class ipaddress.**IPv6Address**(*address*)

构造一个 IPv6 地址。如果 *address* 不是一个有效的 IPv6 地址，会抛出*AddressValueError* 。

以下是有效的 IPv6 地址：

1. 一个由八组四个 16 进制数字组成的字符串，每组展示为 16 位. 以冒号分隔. 这可以描述为分解 (普通书写). 此字符可以被压缩 (速记书写). 详见 **RFC 4291** . 例如, "0000:0000:0000:0000:0000:0abc:0007:0def" 可以被精简为 "::abc:7:def".

   可选择的是，该字符串也可以有一个作用域 ID，用后缀 %scope_id 表示。如果存在，作用域 ID 必须是非空的，并且不能包含 %。详见 **RFC 4007**。例如，fe80::1234%1 可以识别节点第一条链路上的地址 fe80::1234

2. 适合 128 位的整数.

3. 一个打包在长度为 16 字节的大端序*bytes* 对象中的整数。

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

地址表示的简短形式，省略了组中的前导零，完全由零组成的最长的组序列被折叠成一个空组。

这也是 `str(addr)` 对 IPv6 地址返回的值。

exploded

地址的长形式表示，包括所有前导零和完全由零组成的组。

关于以下属性和方法，请参见*IPv4Address* 类的相应文档。

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

Added in version 3.4.

is_unspecified

is_reserved

is_loopback

is_link_local

is_site_local

如果地址被保留用于本地站点则为 `True`。请注意本地站点地址空间已被 **RFC 3879** 弃用。请使用*is_private* 来检测此地址是否位于 **RFC 4193** 所定义的本地唯一地址空间中。

ipv4_mapped

地址为映射的 IPv4 地址 (起始为 `::FFFF/96`), 此属性记录为嵌入 IPv4 地址. 其他的任何地址, 此属性为 `None`.

scope_id

对于 **RFC 4007** 定义的作用域地址，此属性以字符串的形式确定地址所属的作用域的特定区域。当没有指定作用域时，该属性将是 `None`。

sixtofour

对于看起来是 6to4 的地址（以 `2002::/16``开头），如:RFC:`3056` 所定义的，此属性将返回嵌入的 IPv4 地址。对于任何其他地址，该属性将是``None`。

teredo

对于看起来是 **RFC 4380** 定义的 Teredo 地址（以 `2001::/32` 开头）的地址，此属性将返回嵌入式 IP 地址对 `(server, client)`。对于任何其他地址，该属性将是 `None`。

`IPv6Address.`**`__format__`**(*fmt*)

请参考*IPv4Address* 中对应的方法文档。

Added in version 3.9.

转换字符串和整数

与网络模块互操作像套接字模块, 地址必须转换为字符串或整数. 这是使用*str()* 和*int()* 内置函数:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

请注意，IPv6 范围内的地址被转换为没有范围区域 ID 的整数。

运算符

地址对象支持一些运算符。除非另有说明，运算符只能在兼容对象之间应用（即 IPv4 与 IPv4，IPv6 与 IPv6）。

比较运算符

地址对象可以用通常的一组比较运算符进行比较。具有不同范围区域 ID 的相同 IPv6 地址是不平等的。一些例子:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

算术运算符

整数可以被添加到地址对象或从地址对象中减去。一些例子:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

22.23.3 IP 网络的定义

IPv4Network 和 *IPv6Network* 对象提供了一个定义和检查 IP 网络定义的机制。一个网络定义由一个 *掩码*和一个 *网络地址*组成，因此定义了一个 IP 地址的范围，当用掩码屏蔽（二进制 AND）时，等于网络地址。例如，一个带有掩码 `255.255.255.0` 和网络地址 `192.168.1.0` 的网络定义由包括 `192.168.1.0` 到 `192.168.1.255` 的 IP 地址组成。

前缀、网络掩码和主机掩码

有几种相等的方法来指定 IP 网络掩码。*前缀* `/<nbits>` 是一个符号，表示在网络掩码中设置了多少个高阶位。一个 *网络掩码*是一个设置了一定数量高阶位的 IP 地址。因此，*前缀* `/24` 等同于 IPv4 中的网络掩码 `255.255.255.0` 或 IPv6 中的网络掩码 `ffff:ff00::` 。此外，*主机掩码*是 *网络掩码*的逻辑取反，有时被用来表示网络掩码（例如在 Cisco 访问控制列表中）。在 IPv4 中，相当于主机掩码 `0.0.0.255` 的是 `/24` 。

网络对象

所有由地址对象实现的属性也由网络对象实现。此外，网络对象还实现了额外的属性。所有这些在*IPv4Network* 和*IPv6Network* 之间是共同的，所以为了避免重复，它们只在*IPv4Network* 中记录。网络对象是*hashable*，所以它们可以作为字典中的键使用。

class ipaddress.**IPv4Network**(*address*, *strict=True*)

构建一个 IPv4 网络定义。*address* 可以是以下之一：

1. 一个由 IP 地址和可选掩码组成的字符串，用斜线 (/) 分开。IP 地址是网络地址，掩码可以是一个单一的数字，这意味着它是一个 *前缀*，或者是一个 IPv4 地址的字符串表示。如果是后者，如果掩码以非零字段开始，则被解释为 *网络掩码*，如果以零字段开始，则被解释为 *主机掩码*，唯一的例外是全零的掩码被视为 *网络掩码*。如果没有提供掩码，它就被认为是 `/32`。

   例如，以下的 * 地址 * 描述是等同的: `192.168.1.0/24`, `192.168.1.0/255.255.255.0``和``192.168.1.0/0.0.0.255`

2. 一个可转化为 32 位的整数。这相当于一个单地址的网络，网络地址是 *address*，掩码是 `/32`。

3. 一个整数被打包成一个长度为 4 的大端序*bytes* 对象。其解释类似于一个整数 *address*。

4. 一个地址描述和一个网络掩码的双元组，其中地址描述是一个字符串，一个 32 位的整数，一个 4 字节的打包整数，或一个现有的 IPv4Address 对象；而网络掩码是一个代表前缀长度的整数 (例如 `24`) 或一个代表前缀掩码的字符串 (例如 `255.255.255.0`)。

如果 *address* 不是一个有效的 IPv4 地址则会引发*AddressValueError*。如果掩码不是有效的 IPv4 地址则会引发*NetmaskValueError*。

如果 *strict* 是 `True`，并且在提供的地址中设置了主机位，那么*ValueError* 将被触发。否则，主机位将被屏蔽掉，以确定适当的网络地址。

除非另有说明，如果参数的 IP 版本与 `self` 不兼容，所有接受其他网络/地址对象的网络方法都将引发*TypeError*。

在 3.5 版本发生变更: 为 *address* 构造函数参数添加了双元组形式。

version

max_prefixlen

请参考*IPv4Address* 中的相应属性文档。

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

如果这些属性对网络地址和广播地址都是 True，那么它们对整个网络来说就是 True。

network_address

网络的网络地址。网络地址和前缀长度一起唯一地定义了一个网络。

broadcast_address

网络的广播地址。发送到广播地址的数据包应该被网络上的每台主机所接收。

hostmask

主机掩码，作为一个*IPv4Address* 对象。

netmask

网络掩码，作为一个*IPv4Address* 对象。

with_prefixlen

compressed

exploded

网络的字符串表示，其中掩码为前缀符号。

`with_prefixlen` 和 `compressed` 总是与 `str(network)` 相同，`exploded` 使用分解形式的网络地址。

with_netmask

网络的字符串表示，掩码用网络掩码符号表示。

with_hostmask

网络的字符串表示，其中的掩码为主机掩码符号。

num_addresses

网络中的地址总数。

prefixlen

网络前缀的长度，以比特为单位。

hosts()

返回一个网络中可用主机的迭代器。可用的主机是属于该网络的所有 IP 地址，除了网络地址本身和网络广播地址。对于掩码长度为 31 的网络，网络地址和网络广播地址也包括在结果中。掩码为 32 的网络将返回一个包含单一主机地址的列表。

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)

如果这个网络部分或全部包含在 *other* 中，或者 *other* 全部包含在这个网络中，则为 `True`。

address_exclude(*network*)

计算从这个网络中移除给定的 *network* 后产生的网络定义。返回一个网络对象的迭代器。如果 *network* 不完全包含在这个网络中则会引发 *ValueError*。

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)

根据参数值，加入的子网构成当前的网络定义。*prefixlen_diff* 是我们的前缀长度应该增加的数量。*new_prefix* 是所需的子网的新前缀；它必须大于我们的前缀。必须设置 *prefixlen_diff* 和 *new_prefix* 中的一个，且只有一个。返回一个网络对象的迭代器。

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
```

(续下页)

(接上页)

```
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

包含这个网络定义的超级网，取决于参数值。*prefixlen_diff* 是我们的前缀长度应该减少的数量。*new_prefix* 是超级网的新前缀；它必须比我们的前缀小。必须设置 *prefixlen_diff* 和 *new_prefix* 中的一个，而且只有一个。返回一个单一的网络对象。

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

如果这个网络是 *other* 的子网，则返回 `True`。

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

Added in version 3.7.

supernet_of(*other*)

如果这个网络是 *other* 的超网，则返回 `True`。

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

Added in version 3.7.

compare_networks(*other*)

将这个网络与 *ohter* 网络进行比较。在这个比较中，只考虑网络地址；不考虑主机位。返回是 `-1` 、 `0`` 或`` `1`。

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

自 3.7 版本弃用: 它使用与"<"、"==" 和">" 相同的排序和比较算法。

class ipaddress.**IPv6Network**(*address*, *strict=True*)

构建一个 IPv6 网络定义。*address* 可以是以下之一：

1. 一个由 IP 地址和可选前缀长度组成的字符串，用斜线（`/`）分开。IP 地址是网络地址，前缀长度必须是一个数字，即 *prefix*。如果没有提供前缀长度，就认为是 `/128`。

   请注意，目前不支持扩展的网络掩码。这意味着 `2001:db00::0/24` 是一个有效的参数，而 `2001:db00::0/ffff:ff00::` 不是。

2. 一个适合 128 位的整数。这相当于一个单地址网络，网络地址是 *address*，掩码是 `/128`。
3. 一个整数被打包成一个长度为 16 的大端序*bytes* 对象。其解释类似于一个整数的 *address*。
4. 一个地址描述和一个网络掩码的双元组，其中地址描述是一个字符串，一个 128 位的整数，一个 16 字节的打包整数，或者一个现有的 IPv6Address 对象；而网络掩码是一个代表前缀长度的整数。

如果 *address* 不是一个有效的 IPv6 地址则会引发*AddressValueError*。如果掩码对 IPv6 地址无效则会引发*NetmaskValueError*。

如果 *strict* 是 `True`，并且在提供的地址中设置了主机位，那么*ValueError* 将被触发。否则，主机位将被屏蔽掉，以确定适当的网络地址。

在 3.5 版本发生变更: 为 *address* 构造函数参数添加了双元组形式。

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

返回一个网络中可用主机的迭代器。可用的主机是属于该网络的所有 IP 地址，除了 Subnet-Router 任播的地址。对于掩码长度为 127 的网络，子网-路由器任播地址也包括在结果中。掩码为 128 的网络将返回一个包含单一主机地址的列表。

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)

请参考*IPv4Network* 中的相应属性文档。

is_site_local

如果这些属性对网络地址和广播地址都是 True，那么对整个网络来说就是 True。

运算符

网络对象支持一些运算符。除非另有说明，运算符只能在兼容的对象之间应用（例如，IPv4 与 IPv4，IPv6 与 IPv6）。

逻辑运算符

网络对象可以用常规的逻辑运算符集进行比较。网络对象首先按网络地址排序，然后按网络掩码排序。

迭代

网络对象可以被迭代，以列出属于该网络的所有地址。对于迭代，所有主机都会被返回，包括不可用的主机（对于可用的主机，使用*hosts()* 方法）。一个例子:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

作为地址容器的网络

网络对象可以作为地址的容器。一些例子:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

22.23.4 接口对象

接口对象是*hashable* 的，所以它们可以作为字典中的键使用。

class ipaddress.**IPv4Interface**(*address*)

构建一个 IPv4 接口。*address* 的含义与*IPv4Network* 构造器中的一样，不同之处在于任意主机地址总是会被接受。

IPv4Interface 是*IPv4Address* 的一个子类，所以它继承了该类的所有属性。此外，还有以下属性可用：

ip

地址（*IPv4Address*）没有网络信息。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

该接口所属的网络（*IPv4Network*）。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

用前缀符号表示的接口与掩码的字符串。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

带有网络的接口的网络掩码字符串表示。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

带有网络的接口的主机掩码字符串表示。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)

构建一个 IPv6 接口。*address* 的含义与*IPv6Network* 构造器中的一样，不同之处在于任意主机地址总是会被接受。

IPv6Interface 是*IPv6Address* 的一个子类，所以它继承了该类的所有属性。此外，还有以下属性可用：

ip

network

with_prefixlen

with_netmask

with_hostmask

请参考*IPv4Interface* 中的相应属性文档。

运算符

接口对象支持一些运算符。除非另有说明，运算符只能在兼容的对象之间应用（即 IPv4 与 IPv4，IPv6 与 IPv6）。

逻辑运算符

接口对象可以用通常的逻辑运算符集进行比较。

对于相等比较（== 和 !=），IP 地址和网络都必须是相同的对象才会相等。一个接口不会与任何地址或网络对象相等。

对于排序 (<、> 等)，规则是不同的。具有相同 IP 版本的接口和地址对象可以被比较，而地址对象总是在接口对象之前排序。两个接口对象首先通过它们的网络进行比较，如果它们是相同的，则通过它们的 IP 地址进行比较。

22.23.5 其他模块级别函数

该模块还提供以下模块级函数：

ipaddress.**v4_int_to_packed**(*address*)

以网络（大端序）顺序将一个地址表示为 4 个打包的字节。*address* 是一个 IPv4 IP 地址的整数表示。如果整数是负数或太大而不满足 IPv4 IP 地址要求，会触发一个*ValueError*。

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)

以网络（大端序）顺序将一个地址表示为 4 个打包的字节。*address* 是一个 IPv6 IP 地址的整数表示。如果整数是负数或太大而不满足 IPv6 IP 地址要求，会触发一个*ValueError*。

ipaddress.**summarize_address_range**(*first*, *last*)

给出第一个和最后一个 IP 地址，返回总结的网络范围的迭代器。*first* 是范围内的第一个*IPv4Address* 或*IPv6Address*，*last* 是范围内的最后一个*IPv4Address* 或*IPv6Address*。如果 *first* 或 *last* 不是 IP 地址或不是同一版本则会引发*TypeError*。如果 *last* 不大于 *first*，或者 *first* 的地址版本不是 4 或 6 则会引发*ValueError*。

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.130/32
↪')]
```

ipaddress.**collapse_addresses**(*addresses*)

返回一个已展开的*IPv4Network* 或*IPv6Network* 对象的迭代器。*addresses* 是一个*IPv4Network* 或*IPv6Network* 对象的*iterable*。如果 *addresses* 包含混合版本的对象则会引发*TypeError*。

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

返回一个适合在网络和地址之间进行排序的键。地址和网络对象在默认情况下是不可排序的；它们在本质上是不同的，所以表达式：

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

是没有意义的。不过在某些时候，你可能还是希望让*ipaddress*对这些进行排序。如果你需要这样做，你可以使用这个函数作为*sorted()*的 *key* 参数。

obj 是一个网络或地址对象。

22.23.6 自定义异常

为了支持来自类构造函数的更具体的错误报告，模块定义了以下异常：

exception ipaddress.**AddressValueError**(*ValueError*)

与地址有关的任何数值错误。

exception ipaddress.**NetmaskValueError**(*ValueError*)

与网络掩码有关的任何数值错误。

CHAPTER 23

多媒体服务

本章描述的模块实现了主要用于多媒体应用的各种算法或接口。它们可在安装时自行决定。这是一个概述:

23.1 wave --- 读写 WAV 文件

源代码: Lib/wave.py

wave 模块提供了一个处理 Waveform Audio "WAVE" (或称"WAV") 文件格式的便利接口。仅支持未压缩的 PCM 编码波形文件。

在 3.12 版本发生变更: 增加了对 WAVE_FORMAT_EXTENSIBLE 标头的支持，要求扩展格式为 KSDATAFORMAT_SUBTYPE_PCM。

wave 模块定义了以下函数和异常:

wave.**open**(*file*, *mode=None*)

如果 *file* 是一个字符串，打开对应文件名的文件。否则就把它作为文件型对象来处理。*mode* 可以为以下值:

'rb'
只读模式。

'wb'
只写模式。

注意不支持同时读写 WAV 文件。

mode 设为 'rb' 时返回一个 *Wave_read* 对象，而 *mode* 设为 'wb' 时返回一个 *Wave_write* 对象。如果省略 *mode* 并指定 *file* 来传入一个文件型对象，则 file.mode 会被用作 *mode* 的默认值。

如果你传入一个文件型对象，当调用 wave 对象的 close() 方法时并不会真正关闭它；调用者需要负责关闭文件对象。

open() 函数可以在 with 语句中使用。当 with 代码块结束时，*Wave_read.close()* 或 *Wave_write.close()* 方法会被调用。

在 3.4 版本发生变更: 添加了对不可搜索文件的支持。

exception wave.**Error**

当不符合 WAV 格式或无法操作时引发的错误。

23.1.1 Wave_read 对象

class wave.**Wave_read**

读取一个 WAV 文件。

由*open()* 返回的 Wave_read 对象，有以下几种方法:

close()

关闭*wave* 打开的数据流并使对象不可用。当对象销毁时会自动调用。

getnchannels()

返回声道数量（1 为单声道，2 为立体声）

getsampwidth()

返回采样字节长度。

getframerate()

返回采样频率。

getnframes()

返回音频总帧数。

getcomptype()

返回压缩类型（只支持 'NONE' 类型）

getcompname()

getcomptype() 的通俗版本。使用 'not compressed' 代替 'NONE'。

getparams()

返回一个*namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname)，等价于 get*() 方法的输出。

readframes(*n*)

读取并返回以*bytes* 对象表示的最多 *n* 帧音频。

rewind()

重置文件指针至音频开头.

以下两个方法是为了和旧的 aifc 模块保持兼容而定义的，实际上不会做任何事情。

getmarkers()

返回 None。

Deprecated since version 3.13, will be removed in version 3.15: 此方法只是为了和已在 Python 3.13 中被移除的 aifc 模块保持兼容而存在的。

getmark(*id*)

引发错误异常。

Deprecated since version 3.13, will be removed in version 3.15: 此方法只是为了和已在 Python 3.13 中被移除的 aifc 模块保持兼容而存在的。

以下两个方法都使用指针，具体实现由其底层决定。

setpos(*pos*)

设置文件指针到指定位置。

tell()

返回当前文件指针位置。

23.1.2 Wave_write 对象

class wave.**Wave_write**

写入一个 WAV 文件。

Wave_write 对象，由*open()* 返回。

对于可查找的输出流，wave 头将自动更新以反映实际写入的帧数。对于不可查找的流，当写入第一帧时 *nframes* 值必须是准确的。要获取准确的 *nframes* 值可以通过调用*setnframes()* 或*setparams()* 并附带*close()* 被调用之前将要写入的帧数然后使用*writeframesraw()* 来写入帧数据，或者通过调用*writeframes()* 并附带所有要写入的帧。在后一种情况下*writeframes()* 将计算数据中的帧数并在写入帧数据之前相应地设置 *nframes*。

在 3.4 版本发生变更: 添加了对不可搜索文件的支持。

Wave_write 对象具有以下方法:

close()

确保 *nframes* 是正确的，并在文件被*wave* 打开时关闭它。此方法会在对象收集时被调用。如果输出流不可查找且 *nframes* 与实际写入的帧数不匹配时引发异常。

setnchannels(*n*)

设置声道数。

setsampwidth(*n*)

设置采样字节长度为 *n*。

setframerate(*n*)

设置采样频率为 *n*。

在 3.2 版本发生变更: 对此方法的非整数输入会被舍入到最接近的整数。

setnframes(*n*)

设置总帧数为 *n*。如果与之后实际写入的帧数不一致此值将会被更改（如果输出流不可查找则此更改尝试将引发错误）。

setcomptype(*type*, *name*)

设置压缩格式。目前只支持 NONE 即无压缩格式。

setparams(*tuple*)

tuple 应该是 (nchannels, sampwidth, framerate, nframes, comptype, compname)，每项的值可用于 set*() 方法。设置所有形参。

tell()

返回当前文件指针，其指针含义和*Wave_read.tell()* 以及*Wave_read.setpos()* 是一致的。

writeframesraw(*data*)

写入音频数据但不更新 *nframes*。

在 3.4 版本发生变更: 现在可接受任意*bytes-like object*。

writeframes(*data*)

写入音频帧并确保 *nframes* 是正确的。如果输出流不可查找且在 *data* 被写入之后写入的总帧数与之前设定的 *nframes* 值不匹配将会引发错误。

在 3.4 版本发生变更: 现在可接受任意*bytes-like object*。

注意在调用*writeframes()* 或*writeframesraw()* 之后再设置任何格式参数是无效的，而且任何这样的尝试将引发*wave.Error*。

23.2 colorsys --- 颜色系统间的转换

源代码： Lib/colorsys.py

colorsys 模块定义了计算机显示器所用的 RGB (Red Green Blue) 色彩空间与三种其他色彩坐标系统 YIQ, HLS (Hue Lightness Saturation) 和 HSV (Hue Saturation Value) 表示的颜色值之间的双向转换。所有这些色彩空间的坐标都使用浮点数值来表示。在 YIQ 空间中，Y 坐标取值为 0 和 1 之间，而 I 和 Q 坐标均可以为正数或负数。在所有其他空间中，坐标取值均为 0 和 1 之间。

参见

有关色彩空间的更多信息可访问 https://poynton.ca/ColorFAQ.html 和 https://www.cambridgeincolour.com/tutorials/color-spaces.htm。

colorsys 模块定义了如下函数：

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
把颜色从 RGB 值转为 YIQ 值。

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
把颜色从 YIQ 值转为 RGB 值。

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HLS 值。

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
把颜色从 HLS 值转为 RGB 值。

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HSV 值。

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
把颜色从 HSV 值转为 RGB 值。

示例：

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

CHAPTER 24

国际化

本章中介绍的模块通过提供选择要在程序信息中使用的语言的机制或通过定制输出以匹配本地约定来帮助你编写不依赖于语言和区域设置的软件。

本章中描述的模块列表是：

24.1 gettext --- 多语种国际化服务

源代码： Lib/gettext.py

gettext 模块为 Python 模块和应用程序提供国际化 (Internationalization, I18N) 和本地化 (Localization, L10N) 服务。它同时支持 GNU **gettext** 消息编目 API 和更高级的、基于类的 API，后者可能更适合于 Python 文件。下方描述的接口允许用户使用一种自然语言编写模块和应用程序消息，并提供翻译后的消息编目，以便在不同的自然语言下运行。

同时还给出一些本地化 Python 模块及应用程序的小技巧。

24.1.1 GNU gettext API

模块*gettext* 定义了下列 API，这与 **gettext** API 类似。如果你使用该 API，将会对整个应用程序产生全局的影响。如果你的应用程序支持多语种，而语言选择取决于用户的语言环境设置，这通常正是你所想要的。而如果你正在本地化某个 Python 模块，或者你的应用程序需要在运行时切换语言，相反你或许想用基于类的 API。

gettext.**bindtextdomain**(*domain*, *localedir=None*)

将 *domain* 绑定到本地目录 *localedir*。更具体地来说，模块*gettext* 将使用路径 (在 Unix 系统中): *localedir*/*language*/LC_MESSAGES/*domain*.mo 查找二进制 .mo 文件，此处对应地查找 *language* 的位置是环境变量 LANGUAGE, LC_ALL, LC_MESSAGES 和 LANG 中。

如果遗漏了 *localedir* 或者设置为 None，那么将返回当前 *domain* 所绑定的值[1]

gettext.**textdomain**(*domain=None*)

修改或查询当前的全局域。如果 *domain* 为 None，则返回当前的全局域，不为 None 则将全局域设置为 *domain*，并返回它。

[1] 默认的语言区域目录取决于具体系统；例如，在 Red Hat Linux 上为 /usr/share/locale，但在 Solaris 上则为 /usr/lib/locale。gettext 模块没有试图支持这些依赖于系统的默认值；而是默认设为 *sys.base_prefix*/share/locale (参见*sys.base_prefix*)。基于上述原因，最好每次都在程序启动时调用*bindtextdomain()* 并附带一个显式的绝对路径。

gettext.**gettext**(*message*)

返回 *message* 的本地化翻译，依据当前的全局域、语言和语言区域目录。本函数在局部命名空间中通常包含别名 `_()` (参见下面的示例)。

gettext.**dgettext**(*domain*, *message*)

与 *gettext()* 类似，但在指定的 *domain* 中查找 message。

gettext.**ngettext**(*singular*, *plural*, *n*)

与 *gettext()* 类似，但考虑了复数形式。如果找到了翻译，则将 *n* 代入复数公式，然后返回得出的消息（某些语言具有两种以上的复数形式）。如果未找到翻译，则 *n* 为 1 时返回 *singular*，为其他数时返回 *plural*。

复数公式取自编目头文件。它是 C 或 Python 表达式，有一个自变量 *n*，该表达式计算的是所需复数形式在编目中的索引号。关于在 `.po` 文件中使用的确切语法和各种语言的公式，请参阅 GNU gettext 文档 。

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)

与 *ngettext()* 类似，但在指定的 *domain* 中查找 message。

gettext.**pgettext**(*context*, *message*)

gettext.**dpgettext**(*domain*, *context*, *message*)

gettext.**npgettext**(*context*, *singular*, *plural*, *n*)

gettext.**dnpgettext**(*domain*, *context*, *singular*, *plural*, *n*)

与前缀中没有 `p` 的相应函数类似（即 *gettext()*, *dgettext()*, *ngettext()*, *dngettext()* ），但是仅翻译给定的 message *context*。

Added in version 3.8.

请注意 GNU **gettext** 还定义了一个 `dcgettext()` 方法，但它被认为并不实用因此目前尚未实现它。

这是该 API 的典型用法示例:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

24.1.2 基于类的 API

与 GNU **gettext** API 相比，*gettext* 模块的基于类的 API 提供了更多的灵活性和便利性。这是本地化 Python 应用程序和模块的推荐方式。`gettext` 定义了一个 *GNUTranslations* 类，它实现了对 GNU `.mo` 格式文件的解析，并且具有用于返回字符串的方法。本类的实例也可以将自身作为函数 `_()` 安装到内置命名空间中。

gettext.**find**(*domain*, *localedir=None*, *languages=None*, *all=False*)

本函数实现了标准的 `.mo` 文件搜索算法。它接受一个 *domain*，它与 *textdomain()* 接受的域相同。可选参数 *localedir* 与 *bindtextdomain()* 中的相同。可选参数 *languages* 是多条字符串的列表，其中每条字符串都是一种语言代码。

如果没有传入 *localedir*，则使用默认的系统语言环境目录。[2] 如果没有传入 *languages*，则搜索以下环境变量：`LANGUAGE`、`LC_ALL`、`LC_MESSAGES` 和 `LANG`。从这些变量返回的第一个非空值将用作 *languages* 变量。环境变量应包含一个语言列表，由冒号分隔，该列表会被按冒号拆分，以产生所需的语言代码字符串列表。

find() 将扩展并规范化 language，然后遍历它们，搜索由这些组件构建的现有文件:

localedir/*language*/LC_MESSAGES/*domain*.mo

[2] 参阅上方 *bindtextdomain()* 的脚注。

find() 返回找到类似的第一个文件名。如果找不到这样的文件，则返回 None。如果传入了 *all*，它将返回一个列表，包含所有文件名，并按它们在语言列表或环境变量中出现的顺序排列。

gettext.**translation**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*)

根据 *domain*, *localedir* 和 *languages* 返回一个 *Translations 实例，它们将首先被传给*find()* 以获取由所关联的 .mo 文件路径组成的列表。具有相同 .mo 文件名的实例会被缓存。如果提供了 *class_* 则它将是被实例化的类，否则将是*GNUTranslations*。该类的构造器必须接受一个*file object* 参数。

如果找到多个文件，后找到的文件将用作先前文件的替补。为了设置替补，将使用*copy.copy()* 从缓存中克隆每个 translation 对象。实际的实例数据仍在缓存中共享。

如果 .mo 文件未找到，且 *fallback* 为 false（默认值），则本函数引发*OSError* 异常，如果 *fallback* 为 true，则返回一个*NullTranslations* 实例。

在 3.3 版本发生变更: 过去触发的*IOError*，现在是*OSError* 的别名。

在 3.11 版本发生变更: *codeset* 形参已被移除。

gettext.**install**(*domain*, *localedir=None*, *, *names=None*)

这将在 Python 的内置命名空间中安装 _() 函数，基于传给*translation()* 函数的 *domain* 和 *localedir*。

names 参数的信息请参阅 translation 对象的*install()* 方法的描述。

如下所示，通常是将字符串包裹在对 _() 函数的调用中，以标记应用程序中待翻译的字符串，就像这样:

```
print(_('This string will be translated.'))
```

为了方便，可将 _() 函数安装在 Python 的内置命名空间中，这样就可以在应用程序的所有模块中轻松地访问它。

在 3.11 版本发生变更: *names* 现在是仅限关键字形参。

NullTranslations 类

translation 类实际实现的是，将原始源文件消息字符串转换为已翻译的消息字符串。所有 translation 类使用的基类为*NullTranslations*，它提供了基本的接口，可用于编写自己定制的 translation 类。以下是 NullTranslations 的方法:

class gettext.**NullTranslations**(*fp=None*)

接受一个可选参数文件对象 *fp*，该参数会被基类忽略。初始化由派生类设置的"protected"（受保护的）实例变量 *_info* 和 *_charset*，与 *_fallback* 类似，但它是通过*add_fallback()* 来设置的。如果 *fp* 不为 None，就会调用 self._parse(fp)。

_parse(*fp*)

在基类中没有操作，本方法接受文件对象 *fp*，从该文件读取数据，用来初始化消息编目。如果你手头的消息编目文件的格式不受支持，则应重写本方法来解析你的格式。

add_fallback(*fallback*)

添加 *fallback* 为当前 translation 对象的替补对象。如果 translation 对象无法为指定消息提供翻译，则应向替补查询。

gettext(*message*)

如果设置了替补，则转发 gettext() 给替补。否则返回 *message*。在派生类中被重写。

ngettext(*singular*, *plural*, *n*)

如果设置了替补，则转发 ngettext() 给替补。否则，*n* 为 1 时返回 *singular*，为其他时返回 *plural*。在派生类中被重写。

pgettext(*context*, *message*)

如果设置了替补，则转发*pgettext()* 给替补。否则返回已翻译的消息。在派生类中被重写。

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

如果设置了替补，则转发*npgettext()* 给替补。否则返回已翻译的消息。在派生类中被重写。

Added in version 3.8.

info()

返回一个包含在消息编目文件中找到的元数据的字典。

charset()

返回消息编目文件的编码。

install(*names=None*)

本方法将*gettext()* 安装至内建命名空间，并绑定为 `_`。

如果给出了 *names* 形参，则它必须是一个包含除 `_()` 外需要在内置命名空间中安装的函数的名称的序列。受支持的名称有 `'gettext'`, `'ngettext'`, `'pgettext'` 和 `'npgettext'`。

请注意这只是将 `_()` 函数提供给应用程序的一种方式，尽管也是最方便的方式。由于它会全局性地影响整个应用程序，特别是内置命名空间，因此本地化的模块绝不应安装 `_()`。作为替代，它们应使用以下代码使 `_()` 可用于它们的模块:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

这样只把 `_()` 放在模块的全局命名空间中所以只会影响该模块内的调用。

在 3.8 版本发生变更: 添加了 `'pgettext'` 和 `'npgettext'`。

GNUTranslations 类

`gettext` 模块提供了一个派生自*NullTranslations* 的附加类: *GNUTranslations*。该类重写了 `_parse()` 以同时支持以大端序和小端序格式读取 GNU **gettext** 格式的 `.mo` 文件。

GNUTranslations 会从翻译编目中解析可选的元数据。根据惯例 GNU **gettext** 会以空字符串翻译的形式包括元数据。该元数据使用 **RFC 822** 风格的 `key: value` 对，并且应当包含 `Project-Id-Version` 键。如果找到了 `Content-Type` 键，则将使用 `charset` 属性来初始化"protected" `_charset` 实例变量，如未找到则默认为 `None`。如果指定了 charset 编码格式，则从编目中读取的所有消息 ID 和消息字符串都将使用该编码格式转换为 Unicode，否则会设定使用 ASCII。

由于消息 ID 也是以 Unicode 字符串的形式读取的，因此所有 `*gettext()` 方法都会假定消息 ID 为 Unicode 字符串，而不是字节串。

整个键/值对集合将被放入一个字典并设置为"protected" `_info` 实例变量。

如果 `.mo` 文件的魔法值 (magic number) 无效，或遇到意外的主版本号，或在读取文件时发生其他问题，则实例化*GNUTranslations* 类会引发*OSError*。

class gettext.**GNUTranslations**

下列方法是根据基类实现重写的:

gettext(*message*)

在编目中查找 *message* ID，并以 Unicode 字符串形式返回相应的消息字符串。如果在编目中没有 *message* ID 条目，且配置了替补，则查找请求将被转发到替补的*gettext()* 方法。否则，返回 *message* ID。

ngettext(*singular*, *plural*, *n*)

查找消息 ID 的复数形式。*singular* 用作消息 ID，用于在编目中查找，同时 *n* 用于确定使用哪种复数形式。返回的消息字符串是 Unicode 字符串。

如果在编目中没有找到消息 ID，且配置了替补，则查找请求将被转发到替补的*ngettext()* 方法。否则，当 *n* 为 1 时返回 *singular*，其他情况返回 *plural*。

例如:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)

在编目中查找 *context* 和 *message* ID，并以 Unicode 字符串形式返回相应的消息字符串。如果在编目中没有 *message* ID 和 *context* 条目，且配置了替补，则查找请求将被转发到替补的*pgettext()*方法。否则，返回 *message* ID。

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

查找消息 ID 的复数形式。*singular* 用作消息 ID，用于在编目中查找，同时 *n* 用于确定使用哪种复数形式。

如果在编目中没有找到 *context* 对应的消息 ID，且配置了替补，则查找请求将被转发到替补的*npgettext()* 方法。否则，当 *n* 为 1 时返回 *singular*，其他情况返回 *plural*。

Added in version 3.8.

Solaris 消息编目支持

Solaris 操作系统定义了自己的二进制 `.mo` 文件格式，但由于找不到该格式的文档，因此目前不支持该格式。

编目构造器

GNOME 用的 *gettext* 模块是 James Henstridge 写的版本，但该版本的 API 略有不同。它文档中的用法是:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

为了与此模块的旧版本兼容，函数 `Catalog()` 是上述*translation()* 函数的别名。

本模块与 Henstridge 的模块有一个区别：他的编目对象支持通过映射 API 进行访问，但是该特性似乎从未使用过，因此目前不支持该特性。

24.1.3 国际化 (I18N) 你的程序和模块

国际化 (I18N) 是指使程序可切换多种语言的操作。本地化 (L10N) 是指程序的适配能力，一旦程序被国际化，就能适配当地的语言和文化习惯。为了向 Python 程序提供不同语言的消息，需要执行以下步骤:

1. 准备程序或模块，将可翻译的字符串特别标记起来
2. 在已标记的文件上运行一套工具，用来生成原始消息编目
3. 创建消息编目的不同语言的翻译
4. 使用*gettext* 模块，以便正确翻译消息字符串

为了准备代码以实现 I18N，你需要查看文件中的所有字符串。任何需要翻译的字符串都应在 `_('...')` 中包含它来进行标记 --- 即调用函数_。例如:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

在这个例子中，字符串 'writing a log message' 被标记为待翻译，而字符串 'mylog.txt' 和 'w' 没有被标记。

有一些工具可以将待翻译的字符串提取出来。原版的 GNU **gettext** 仅支持 C 或 C++ 源代码，但其扩展版 **xgettext** 可以扫描多种语言的代码，包括 Python 在内，来找出标记为可翻译的字符串。Babel 是一个包括了可用于提取并编译消息编目的 pybabel 脚本的 Python 国际化库。François Pinard 的 **xpot** 程序也能完成类似的工作并可在他的 po-utils 包 中获取。

（Python 还包括了这些程序的纯 Python 版本，称为 **pygettext.py** 和 **msgfmt.py**，某些 Python 发行版已经安装了它们。**pygettext.py** 类似于 **xgettext**，但只能理解 Python 源代码，无法处理诸如 C 或 C++ 的其他编程语言。**pygettext.py** 支持的命令行界面类似于 **xgettext**，查看其详细用法请运行 pygettext.py --help。**msgfmt.py** 与 GNU **msgfmt** 是二进制兼容的。有了这两个程序，可以不需要 GNU **gettext** 包来国际化 Python 应用程序。）

xgettext、**pygettext** 或类似工具生成的 .po 文件就是消息编目。它们是结构化的人类可读文件，包含源代码中所有被标记的字符串，以及这些字符串的翻译的占位符。

然后把这些 .po 文件的副本交给各个人工译者，他们为所支持的每种自然语言编写翻译。译者以 < 语言名称>.po 文件的形式发送回翻译完的某个语言的版本，将该文件用 **msgfmt** 程序编译为机器可读的 .mo 二进制编目文件。*gettext* 模块使用 .mo 文件在运行时进行实际的翻译处理。

如何在代码中使用*gettext* 模块取决于国际化单个模块还是整个应用程序。接下来的两节将讨论每种情况。

本地化你的模块

如果要本地化模块，则切忌进行全局性的更改，如更改内建命名空间。不应使用 GNU **gettext** API，而应使用基于类的 API。

假设你的模块叫做"spam"，并且该模块的各种自然语言翻译 .mo 文件存放于 /usr/share/locale，为 GNU **gettext** 格式。以下内容应放在模块顶部:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

本地化你的应用程序

如果你正在本地化你的应用程序，你可以将 _() 函数全局安装到内置命名空间中，通常位于应用程序的主驱动文件内。这样将让你的应用程序专属的所有文件都可以使用 _('...') 而无需在每个文件中显示安装它。

最简单的情况，就只需将以下代码添加到应用程序的主程序文件中:

```
import gettext
gettext.install('myapplication')
```

如果需要设置语言环境目录，可以将其传递给*install()* 函数:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

即时更改语言

如果程序需要同时支持多种语言，则可能需要创建多个翻译实例，然后在它们之间进行显式切换，如下所示:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])
```

(续下页)

(接上页)

```
# 从使用语言 1 开始
lang1.install()

# ... 过一段时间后，用户选择了语言 2
lang2.install()

# ... 再过一段时间后，用户选择了语言 3
lang3.install()
```

延迟翻译

在大多数代码中，字符串会在编写位置进行翻译。但偶尔需要将字符串标记为待翻译，实际翻译却推迟到后面。一个典型的例子是:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

此处希望将 `animals` 列表中的字符串标记为可翻译，但不希望在打印之前对它们进行翻译。

这是处理该情况的一种方式:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

这样做是因为 `_()` 的虚定义只是简单地原样返回字符串。并且这个虚定义将临时覆盖内置命名空间中任何的 `_()` 定义（直到 `del` 命令）。但是如果之前你在局部命名空间中已有 `_()` 的定义，则需要特别注意。

请注意在第二次使用 `_()` 时将不会认为 “a” 可以由 **gettext** 程序去翻译，因为该形参不是字符串字面值。

解决该问题的另一种方法是下面这个例子:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

在这种情况下，你用函数 `N_()` 来标记可翻译的字符串，它与 `_()` 的任何定义都不会冲突。不过，你需要让你的消息提取程序寻找用 `N_()` 标记的可翻译字符串。**xgettext**, **pygettext**, `pybabel extract` 和 **xpot** 都通过使用 `-k` 命令行开关来支持此功能。这里选择用 `N_()` 完全是任意的；它也可以简单地改为 `MarkThisStringForTranslation()`。

24.1.4 致谢

以下人员为创建此模块贡献了代码、反馈、设计建议、早期实现和宝贵的经验：

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

备注

24.2 `locale` --- 国际化服务

源代码： Lib/locale.py

---

locale 模块开通了 POSIX 本地化数据库和功能的访问。POSIX 本地化机制让程序员能够为应用程序处理某些本地化的问题，而不需要去了解运行软件的每个国家的全部语言习惯。

locale 模块是在 `_locale` 模块之上实现的，后者又会在可能的情况下使用 ANSI C 语言区域实现。

locale 模块定义了以下异常和函数：

exception locale.**Error**

当传给*setlocale()* 的 locale 无法识别时，会触发异常。

locale.**setlocale**(*category*, *locale=None*)

如果给定了 *locale* 而不是 `None`，*setlocale()* 会修改 *category* 的 locale 设置。可用的类别会在下面的数据描述中列出。*locale* 可以是一个字符串，也可以是两个字符串（语言代码和编码）组成的可迭代对象。若为可迭代对象，则会用地区别名引擎转换为一个地区名称。若为空字符串则指明采用用户的默认设置。如果 locale 设置修改失败，会触发*Error* 异常。如果成功则返回新的 locale 设置。

如果省略 *locale* 或为 `None`，将返回 *category* 但当前设置。

setlocale() 在大多数系统上都不是线程安全的。应用程序通常会如下调用：

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

这会把所有类别的 locale 都设为用户的默认设置（通常在 `LANG` 环境变量中指定）。如果后续 locale 没有改动，则使用多线程应该不会产生问题。

locale.**localeconv**()

以字典的形式返回本地约定的数据库。此字典具有以下字符串作为键：

类别	键	含意
LC_NUMERIC	'decimal_point'	小数点字符。
	'grouping'	数字列表，指定 'thousands_sep' 应该出现的位置。如果列表以*CHAR_MAX* 结束，则不会作分组。如果列表以 0 结束，则重复使用最后的分组大小。
	'thousands_sep'	组之间使用的字符。
LC_MONETARY	'int_curr_symbol'	国际货币符号。
	'currency_symbol'	当地货币符号。
	'p_cs_precedes/n_cs_precedes'	货币符号是否在值之前（对于正值或负值）。
	'p_sep_by_space/n_sep_by_space'	货币符号是否通过空格与值分隔（对于正值或负值）。
	'mon_decimal_point'	用于货币金额的小数点。
	'frac_digits'	货币值的本地格式中使用的小数位数。
	'int_frac_digits'	货币价值的国际格式中使用的小数位数。
	'mon_thousands_sep'	用于货币值的组分隔符。
	'mon_grouping'	相当于 'grouping'，用于货币价值。
	'positive_sign'	用于标注正货币价值的符号。
	'negative_sign'	用于注释负货币价值的符号。
	'p_sign_posn/n_sign_posn'	符号的位置（对于正值或负值），见下文。

可以将所有数值设置为*CHAR_MAX*，以指示本 locale 中未指定任何值。

下面给出了 'p_sign_posn' 和 'n_sign_posn' 的可能值。

值	说明
0	被括号括起来的货币和金额。
1	该标志应位于值和货币符号之前。
2	该标志应位于值和货币符号之后。
3	标志应该紧跟在值之前。
4	标志应该紧跟在值之后。
CHAR_MAX	该地区未指定内容。

此函数可临时性地将 LC_CTYPE 语言区域设为 LC_NUMERIC 语言区域或者如果语言区域不同且数字或货币字符串是非 ASCII 的则设为 LC_MONETARY 语言区域。这个临时性地改变会影响到其他线程。

在 3.7 版本发生变更: 现在此函数在某些情况下会临时性地将 LC_CTYPE 语言区域设为 LC_NUMERIC 语言区域。

locale.**nl_langinfo**(*option*)

以字符串形式返回一些地区相关的信息。本函数并非在所有系统都可用，而且可用的 option 在不同平台上也可能不同。可填的参数值为数值，在 locale 模块中提供了对应的符号常量。

nl_langinfo() 函数可接受以下值。大部分含义都取自 GNU C 库。

locale.**CODESET**

获取一个字符串，代表所选地区采用的字符编码名称。

locale.**D_T_FMT**

获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示日期和时间。

locale.**D_FMT**
: 获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示日期。

locale.**T_FMT**
: 获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示时间。

locale.**T_FMT_AMPM**
: 获取一个字符串，可用作*time.strftime()* 的格式串，以便以 am/pm 的格式表示时间。

locale.**DAY_1**

locale.**DAY_2**

locale.**DAY_3**

locale.**DAY_4**

locale.**DAY_5**

locale.**DAY_6**

locale.**DAY_7**
: 获取一周中第 n 天的名称。

> **备注**
>
> 这里遵循美国惯例，即*DAY_1* 是星期天，而不是国际惯例（ISO 8601），即星期一是一周的第一天。

locale.**ABDAY_1**

locale.**ABDAY_2**

locale.**ABDAY_3**

locale.**ABDAY_4**

locale.**ABDAY_5**

locale.**ABDAY_6**

locale.**ABDAY_7**
: 获取一周中第 n 天的缩写名称。

locale.**MON_1**

locale.**MON_2**

locale.**MON_3**

locale.**MON_4**

locale.**MON_5**

locale.**MON_6**

locale.**MON_7**

locale.**MON_8**

locale.**MON_9**

locale.**MON_10**

locale.**MON_11**

locale.**MON_12**
: 获取第 n 个月的名称。

locale.**ABMON_1**

locale.**ABMON_2**

locale.**ABMON_3**

locale.**ABMON_4**

locale.**ABMON_5**

locale.**ABMON_6**

locale.**ABMON_7**

locale.**ABMON_8**

locale.**ABMON_9**

locale.**ABMON_10**

locale.**ABMON_11**

locale.**ABMON_12**

获取第 n 个月的缩写名称。

locale.**RADIXCHAR**

获取小数点字符（小数点、小数逗号等）。

locale.**THOUSEP**

获取千位数（三位数一组）的分隔符。

locale.**YESEXPR**

获取一个可供 regex 函数使用的正则表达式，用于识别需要回答是或否的问题的肯定回答。

locale.**NOEXPR**

获取一个可供 `regex(3)` 函数使用的正则表达式以识别对于是/否型问题的否定回答。

> **备注**
>
> 针对*YESEXPR* 和*NOEXPR* 的正则表达式使用了适合来自 C 库的 `regex` 函数的语法，它与*re* 中使用的语法会有所不同。

locale.**CRNCYSTR**

获取货币符号，如果符号应位于数字之前，则在其前面加上“-”；如果符号应位于数字之后，则前面加“+”；如果符号应取代小数点字符，则前面加“.”。

locale.**ERA**

获取用来在某一语言区域中描述每个时代如何计算和显示年份的字符串。

大多数地区都没有定义该值。定义了该值的一个案例日本。日本传统的日期表示方法中，包含了当时天皇统治朝代的名称。

通常没有必要直接使用该值。在格式字符串中使用 `E` 说明符将会让*time.strftime()* 函数使用此信息。被返回字符串的格式是在 *The Open Group Base Specifications Issue 8*, paragraph 7.3.5.2 LC_TIME C-Language Access 中指明的。

locale.**ERA_D_T_FMT**

获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示带纪元的日期和时间。

locale.**ERA_D_FMT**

获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示带纪元的日期。

locale.**ERA_T_FMT**

获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示带纪元的时间。

locale.**ALT_DIGITS**

获取一个由至多 100 个以分号分隔的用于通过特定语言区域专属的方式表示数值 0 到 99 的符号名称组成的字符串。在大多数语言区域中这都是一个空字符串。

locale.**getdefaultlocale**([*envvars*])

尝试确定默认的地区设置，并以 `(language code, encoding)` 元组的形式返回。

根据 POSIX 的规范，未调用 `setlocale(LC_ALL, '')` 的程序采用可移植的 `'C'` 区域设置运行。调用 `setlocale(LC_ALL, '')` 则可采用 `LANG` 变量定义的默认区域。由于不想干扰当前的区域设置，因此就以上述方式进行了模拟。

为了维持与其他平台的兼容性，不仅需要检测 `LANG` 变量，还需要检测 envvars 参数给出的变量列表。首先发现的定义将被采用。*envvars* 默认为 GNU gettext 采用的搜索路径；必须包含 `'LANG'` 变量。GNU gettext 的搜索路径依次包含了 `'LC_ALL'`、`'LC_CTYPE'`、`'LANG'` 和 `'LANGUAGE'`。

除了 `'C'` 之外，语言代码对应 **RFC 1766** 标准。若 *语言代码*和 *编码*无法确定，则可为 `None`。

Deprecated since version 3.11, will be removed in version 3.15.

`locale.`**`getlocale`**(*category=LC_CTYPE*)

以包含 *语言代码*, *编码格式*的序列形式返回指定语言区域类别的当前设置。*category* 可以是某个 `LC_*` 值但不能是*LC_ALL*。默认值为*LC_CTYPE*。

除了 `'C'` 之外，语言代码对应 **RFC 1766** 标准。若 *语言代码*和 *编码*无法确定，则可为 `None`。

`locale.`**`getpreferredencoding`**(*do_setlocale=True*)

根据用户的偏好，返回用于文本数据的*locale encoding*。用户偏好在不同的系统上有不同的表达方式，而且在某些系统上可能无法以编程方式获取到，所以本函数只是返回猜测结果。

某些系统必须调用*setlocale()* 才能获取用户偏好，所以本函数不是线程安全的。如果不需要或不希望调用 setlocale，*do_setlocale* 应设为 `False`。

在 Android 上或者如果启用了*Python UTF-8 模式*，则将始终返回 `'utf-8'`，*locale encoding* 和 *do_setlocale* 参数将被忽略。

Python preinitialization 用于配置 LC_CTYPE 区域。还请参阅*filesystem encoding and error handler*。

在 3.7 版本发生变更: 目前在 Android 上或者如果启用了*Python UTF-8 模式* 此函数将总是返回 `"utf-8"`。

`locale.`**`getencoding`**()

获取当前的*locale encoding*:

- 在 Android 和 VxWorks 上，将返回 `"utf-8"`。
- 在 Unix 上，将返回当前, return the encoding of the current *LC_CTYPE* 语言区域的编码格式。如果 `nl_langinfo(CODESET)` 返回空字符串则将返回 `"utf-8"`: 举例来说，如果当前 LC_CTYPE 语言区域不受支持的时候。
- 在 Windows 上，返回 ANSI 代码页。

Python preinitialization 用于配置 LC_CTYPE 区域。还请参阅*filesystem encoding and error handler*。

此函数类似于*getpreferredencoding(False)*，区别是此函数会忽略*Python UTF-8 模式*。

Added in version 3.11.

`locale.`**`normalize`**(*localename*)

为给定的区域名称返回标准代码。返回的区域代码已经格式化，可供*setlocale()* 使用。如果标准化操作失败，则返回原名称。

如果给出的编码无法识别，则本函数默认采用区域代码的默认编码，这正类似于*setlocale()*。

`locale.`**`strcoll`**(*string1*, *string2*)

根据当前的*LC_COLLATE* 设置，对两个字符串进行比较。与其他比较函数一样，根据 *string1* 位于 *string2* 之前、之后或是相同，返回负值、、正值或者 `0`。

`locale.`**`strxfrm`**(*string*)

将字符串转换为可用于本地化比较的字符串。例如 `strxfrm(s1) < strxfrm(s2)` 相当于 `strcoll(s1, s2) < 0`。在重复比较同一个字符串时，可能会用到本函数，比如整理字符串列表时。

locale.**format_string**(*format*, *val*, *grouping=False*, *monetary=False*)

根据当前的*LC_NUMERIC* 设置对数字 *val* 进行格式化。此格式将遵循 `%` 运算符的约定。对于浮点数值，会根据具体情况修改小数点。如果 *grouping* 为 `True`，还会将分组纳入考虑。

若 *monetary* 为 True，则会用到货币千位分隔符和分组字符串。

格式化符的处理类似 `format % val`，但会考虑到当前的区域设置。

在 3.7 版本发生变更: 增加了关键字参数 *monetary* 。

locale.**currency**(*val*, *symbol=True*, *grouping=False*, *international=False*)

根据当前的*LC_MONETARY* 设置，对数字 *val* 进行格式化。

如果 *symbol* 为真值则返回的字符串将包括货币符号，该参数默认为真值。如果 *grouping* 为 `True` (非默认值)，则会对值进行分组。如果 *international* 为 `True` (非默认值)，则会使用国际货币符号。

> **备注**
>
> 此函数将不适用于'C' 语言区域，所以你必须先通过*setlocale()* 设置一个语言区域。

locale.**str**(*float*)

对浮点数进行格式化，格式要求与内置函数 `str(float)` 相同，但会考虑小数点。

locale.**delocalize**(*string*)

根据*LC_NUMERIC* 的设置，将字符串转换为标准化的数字字符串。

Added in version 3.5.

locale.**localize**(*string*, *grouping=False*, *monetary=False*)

根据*LC_NUMERIC* 的设置，将标准化的数字字符串转换为格式化的字符串。

Added in version 3.10.

locale.**atof**(*string*, *func=float*)

将一个字符串转换为数字，遵循*LC_NUMERIC* 设置，通过在 *string* 上调用*delocalize()* 的结果上调用 *func* 来实现。

locale.**atoi**(*string*)

按照*LC_NUMERIC* 的约定，将字符串转换为整数。

locale.**LC_CTYPE**

字符类型函数的语言区域类别。最重要的是，该类别定义了文件编码格式，即如何将字节解读为 Unicode 码位点。请参阅 **PEP 538** 和 **PEP 540** 了解如何将该变量自动强制转换为 `C.UTF-8` 以避免容器中的无效设置或通过远程 SSH 连接传递的不兼容设置所造成的问题。

Python 在内部并不使用来自 `ctype.h` 的依赖于语言区域的字符转换函数。相反，内部 `pyctype.h` 提供了不依赖于语言区域的等价物如 `Py_TOLOWER`。

locale.**LC_COLLATE**

字符串排序会用到的区域类别。将会影响*locale* 模块的*strcoll()* 和*strxfrm()* 函数。

locale.**LC_TIME**

格式化时间时会用到的区域类别。*time.strftime()* 函数会参考这些约定。

locale.**LC_MONETARY**

格式化货币值时会用到的区域类别。可用值可由*localeconv()* 函数获取。

locale.**LC_MESSAGES**

显示消息时用到的区域类别。目前 Python 不支持应用定制的本地化消息。由操作系统显示的消息，比如由*os.strerror()* 返回的消息可能会受到该类别的影响。

这个值在不符合 POSIX 标准的操作系统上可能不可用，最主要是指 Windows。

locale.**LC_NUMERIC**

用于格式化数字的语言区域类别。*locale* 模块的*format_string()*，*atoi()*，*atof()* 和*str()* 等函数会受到该类别的影响。其他所有数字格式化操作将不受影响。

locale.**LC_ALL**

混合所有的区域设置。如果在区域改动时使用该标志，将尝试设置所有类别的区域参数。只要有任何一个类别设置失败，就不会修改任何类别。在使用此标志获取区域设置时，会返回一个代表所有类别设置的字符串。之后可用此字符串恢复设置。

locale.**CHAR_MAX**

一个符号常量，*localeconv()* 返回多个值时将会用到。

示例：

```
>>> import locale
>>> loc = locale.getlocale()  # 获取当前语言区域
# 使用德语的语言区域；在具体平台上名称可能不同
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # 比较一个包含变音的字符串
>>> locale.setlocale(locale.LC_ALL, '')    # 使用用户首选的语言区域
>>> locale.setlocale(locale.LC_ALL, 'C')  # 使用默认的语言区域 (C)
>>> locale.setlocale(locale.LC_ALL, loc)  # 恢复已保存的语言区域
```

24.2.1 背景、细节、提示、技巧和注意事项

C 语言标准将区域定义为程序级别的属性，修改的代价可能相对较高。此外，有某些实现代码写得不好，频繁改变区域可能会导致内核崩溃。于是要想正确使用区域就变得有些痛苦。

当程序第一次启动时，无论用户偏好定义成什么，区域值都是 C。不过有一个例外，就是在启动时修改*LC_CTYPE* 类别，设置当前区域编码为用户偏好编码。程序必须调用 `setlocale(LC_ALL, '')` 明确表示用户偏好区域将设为其他类别。

若要从库程序中调用*setlocale()*，通常这不是个好主意，因为副作用是会影响整个程序。保存和恢复区域设置也几乎一样糟糕：不仅代价高昂，而且会影响到恢复之前运行的其他线程。

如果是要编写通用模块，需要有一种不受区域设置影响的操作方式（比如某些用到*time.strftime()* 的格式），将不得不寻找一种不用标准库的方案。更好的办法是说服自己，可以采纳区域设置。只有在万不得已的情况下，才能用文档标注出模块与非 C 区域设置不兼容。

根据语言区域执行数字运算的唯一方式就是使用本模块所定义的特殊函数：*atof()*，*atoi()*，*format_string()*，*str()*。

无法根据区域设置进行大小写转换和字符分类。对于（Unicode）文本字符串来说，这些操作都是根据字符值进行的；而对于字节符串来说，转换和分类则是根据字节的 ASCII 值进行的，高位被置位的字节（即非 ASCII 字节）永远不会被转换或被视作字母或空白符之类。

24.2.2 针对扩展程序编写人员和嵌入 Python 运行的程序

除了要查询当前区域，扩展模块不应去调用*setlocale()*。但由于返回值只能用于恢复设置，所以也没什么用（也许只能用于确认是否为 C）。

当 Python 代码使用*locale* 模块来修改语言区域时，这也会影响到嵌入的应用程序。如果嵌入的应用程序不希望发生这种情况，它应当从 `config.c` 文件的内置模块表中移除 `_locale` 扩展模块（所有工作都是由它完成的），以确保 `_locale` 模块不能作为共享库来访问。

24.2.3 访问消息目录

locale.**gettext**(*msg*)

locale.**dgettext**(*domain*, *msg*)

locale.**dcgettext**(*domain*, *msg*, *category*)

locale.**textdomain**(*domain*)

locale.**bindtextdomain**(*domain*, *dir*)

locale.**bind_textdomain_codeset**(*domain*, *codeset*)

locale 模块在提供了 C 库的 gettext 接口的系统上对外公开该接口。它由*gettext()*, *dgettext()*, *dcgettext()*, *textdomain()*, *bindtextdomain()* 和*bind_textdomain_codeset()* 等函数组成。它们与*gettext* 模块中的同名函数类似，但使用了 C 库的二进制格式来表示消息目录，并使用 C 库的搜索算法来查找消息目录。

Python 应用程序通常不需要唤起这些函数，而应当改用*gettext*。这条规则的一个已知例外是与附加 C 库相链接的应用程序，它们会在内部唤起 C 函数 gettext 或 dcgettext。对于这些应用程序，可能有必要绑定文本域，以便库能够正确地找到它们的消息目录。

CHAPTER 25

程序框架

本章中描述的模块是很大程度上决定程序结构的框架。目前，这里描述的模块都面向编写命令行接口。

本章描述的完整模块列表如下：

25.1 `turtle` --- 海龟绘图

源码： Lib/turtle.py

25.1.1 概述

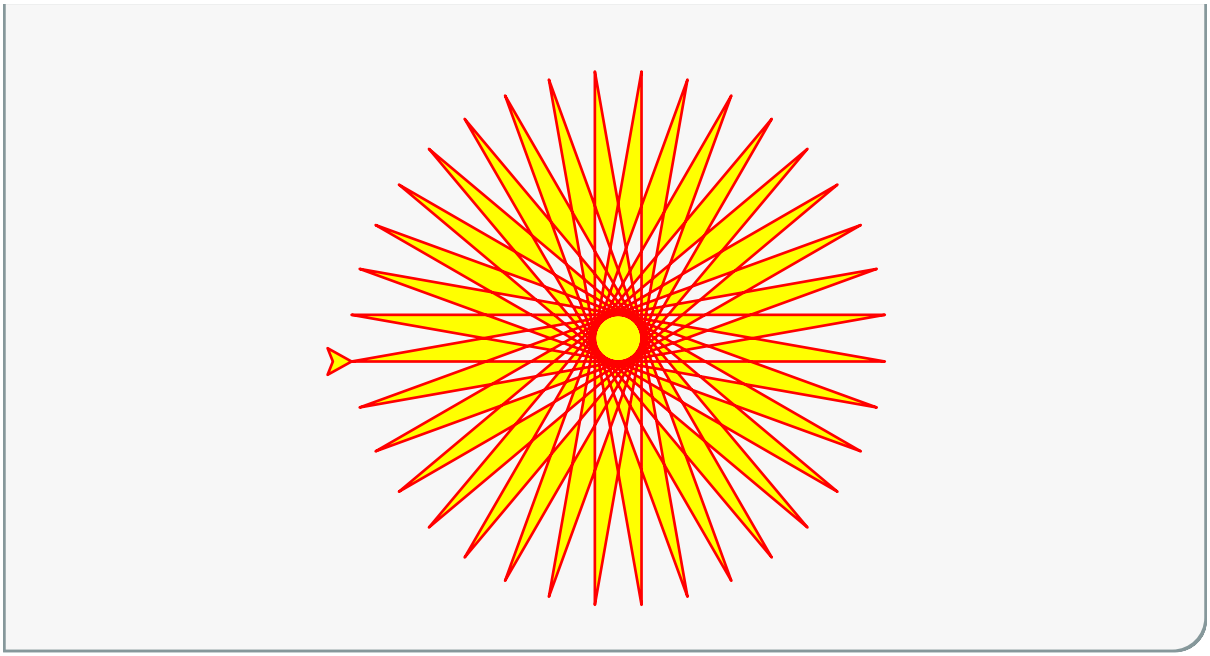
海龟绘图是对 最早在 Logo 中引入的受欢迎的几何绘图工具 的实现，它由 Wally Feurzeig, Seymour Papert 和 Cynthia Solomon 在 1967 年开发。

25.1.2 入门

请想象绘图区有一只机器海龟，起始位置在 x-y 平面的 (0, 0) 点。先执行 `import turtle`，再执行 `turtle.forward(15)`，它将 (在屏幕上) 朝所面对的 x 轴正方向前进 15 像素，随着它的移动画出一条线段。再执行 `turtle.right(25)`，它将原地右转 25 度。

Turtle star

使用海龟绘图可以编写重复执行简单动作的程序画出精细复杂的形状。



在 Python 中，海龟绘图提供了一个实体“海龟”形象（带有画笔的小机器动物），假定它在地板上平铺的纸张上画线。

对于学习者来说这是一种接触编程概念和与软件交互的高效且久经验证的方式，因为它能提供即时、可见的反馈。它还能提供方便直观的图形输出。

海龟绘图最初是作为一种教学工具被创建的，供教师在课堂上使用。对于需要生成一些图形输出的程序员来说这是一种无需在工作中引入更高复杂度或外部库的方式。

25.1.3 教程

新用户应当从这里开始。在本教程中我们将探索海龟绘图的一些基本知识。

启动海龟环境

在 Python shell 中，导入 `turtle` 模块的所有对象:

```
from turtle import *
```

如果你遇到了 `No module named '_tkinter'` 错误，则需要在你的系统中安装 *Tk 接口包*。

基本绘图

让海龟前进 100 步:

```
forward(100)
```

你应该会看到（最可能的情况，是在你的显示器的一个新窗口中）海龟画出一条线段，方向朝东。改变海龟的方向，让它向左转 120 度（逆时针）：

```
left(120)
```

让我们继续画一个三角形:

```
forward(100)
left(120)
forward(100)
```

注意以一个箭头表示的海龟是如何随着你的操纵指向不同方向的。

请继续尝试这些命令，还可以使用 `backward()` 和 `right()`。

画笔控制

试着改变颜色——例如，`color('blue')` 和线宽——例如，`width(3)` 然后再次绘制。

您也可以在不绘制线条的情况下移动海龟，即在移动前抬起画笔: `up()`。要重新开始绘制，请使用 `down()`。

海龟的位置

将海龟送回起点（这适用于海龟消失在屏幕之外的情况）：

```
home()
```

初始位置在海龟屏幕的中心。如果你需要知道具体数值，可以这样获取海龟的 x-y 坐标:

```
pos()
```

初始点在 `(0, 0)`。

过一段时间后，也许可以考虑清空窗口这样我们就可以重新开始:

```
clearscreen()
```

使用算法绘制图案

使用循环，可以构建出各种几何图案:

```
for steps in range(100):
    for c in ('blue', 'red', 'green'):
        color(c)
        forward(steps)
        right(30)
```

- 当然，这仅受限于你的想象力!

让我们绘制本页面顶部的星形。我们想要用红色线条，黄色填充:

```
color('red')
fillcolor('yellow')
```

就像用 `up()` 和 `down()` 决定是否画线一样，填充也可以打开或关闭:

```
begin_fill()
```

接下来我们将创建一个循环:

```
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
```

`abs(pos()) < 1` 是确定海龟何时回到初始点的好办法。

最后，完成填充:

```
end_fill()
```

（请注意只有在你给出 `end_fill()` 命令时才会实际进行填充。）

25.1.4 如何...

本节介绍一些典型的海龟使用案例和操作方式。

尽快地开始

海龟绘图形的乐趣之一在于通过简单的命令就能获得即时的视觉反馈——这是一种向儿童介绍编程理念的绝佳方式，而且开销最小（当然，不仅适用于儿童）。

海龟模块将其所有基本功能作为函数公开，并通过 `from turtle import *` 提供使这一切成为可能。*海龟绘图教程* 介绍了相关的步骤。

值得注意的是许多海乌命令还有更简洁的等价形式，例如 `fd()` 对应*forward()*。对于不擅长打字的学习者来说这尤其有用。

> 你需要在系统中安装*Tk 接口软件包*，才能使用海龟绘图。请注意这并不总是很容易做到的，所以如果你打算让学习者使用海龟绘图请事先检查这一点。

使用 turtle 模块命名空间

使用 `from turtle import *` 是很方便——但要注意它导入的对象集相当大，如果你还在做海龟绘图以外的事情就有发生名称冲突的风险（如果你在可能导入了其他模块的脚本中使用海龟绘图则可能会遇到更大的问题）。

解决办法是使用 `import turtle` ——`fd()` 将变成 `turtle.fd()`，`width()` 将变成 `turtle.width()` 等等。（如果反复输入“turtle”太过烦琐，还可改成 `import turtle as t` 等。）

在脚本中使用海龟绘图

建议使用上文所述的 `turtle` 模块命名空间，例如:

```
import turtle as t
from random import random

for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)
```

但还需要另一个步骤——因为一旦脚本结束，Python 将会同时关闭海龟的窗口。请添加:

```
t.mainloop()
```

到脚本的末尾。现在脚本将等待被关闭而不会自动退出直到被主动终止，例如海龟绘图窗口被关闭。

使用面向对象的海龟绘图

> **参见**
>
> *面向对象接口说明*

除了非常基本的入门目的，或是尽快尝试操作之外，使用面向对象的方式进行海龟绘图更为常见也更为强大。例如，这将允许屏幕上同时存在多只海龟。

在这种方式下，各种海龟命令都是对象（主要是 `Turtle` 对象）的方法。你可以在 shell 中使用面向对象的方法，但在 Python 脚本中使用是更为典型的做法。

这样上面的例子就将变成:

```
from turtle import Turtle
from random import random

t = Turtle()
for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)

t.screen.mainloop()
```

请注意最后一行。`t.screen` 是 Turtle 实例所在的*Screen* 的实例；它是与海龟一起自动创建的。

海龟的屏幕可以被自定义，例如:

```
t.screen.title('Object-oriented turtle demo')
t.screen.bgcolor("orange")
```

25.1.5 海龟绘图参考

备注

以下文档给出了函数的参数列表。对于方法来说当然还有额外的第一个参数 *self*，这里省略了。

Turtle 方法

海龟动作

移动和绘制

forward() | *fd()* 前进
backward() | *bk()* | *back()* 后退
right() | *rt()* 右转
left() | *lt()* 左转
goto() | *setpos()* | *setposition()* 前往/定位
teleport()
setx() 设置 x 坐标
sety() 设置 y 坐标
setheading() | *seth()* 设置朝向
home() 返回原点
circle() 画圆
dot() 画点
stamp() 印章
clearstamp() 清除印章
clearstamps() 清除多个印章
undo() 撤消
speed() 速度

获取海龟的状态

position() | *pos()* 位置
towards() 目标方向
xcor() x 坐标
ycor() y 坐标
heading() 朝向
distance() 距离

设置与度量单位

degrees() 角度
radians() 弧度

画笔控制

绘图状态

pendown() | *pd()* | *down()* 画笔落下
penup() | *pu()* | *up()* 画笔抬起
pensize() | *width()* 画笔粗细
pen() 画笔
isdown() 画笔是否落下

颜色控制

color() 颜色
pencolor() 画笔颜色
fillcolor() 填充颜色

填充

filling() 是否填充
begin_fill() 开始填充
end_fill() 结束填充

更多绘图控制

reset() 重置
clear() 清空
write() 书写

海龟状态

可见性

showturtle() | *st()* 显示海龟
hideturtle() | *ht()* 隐藏海龟
isvisible() 是否可见

外观

shape() 形状
resizemode() 大小调整模式
shapesize() | *turtlesize()* 形状大小
shearfactor() 剪切因子
tiltangle() 倾角
tilt() 倾斜
shapetransform() 变形
get_shapepoly() 获取形状多边形

使用事件

onclick() 当鼠标点击
onrelease() 当鼠标释放
ondrag() 当鼠标拖动

特殊海龟方法

begin_poly() 开始记录多边形
end_poly() 结束记录多边形
get_poly() 获取多边形
clone() 克隆

getturtle() | *getpen()* 获取海龟画笔
getscreen() 获取屏幕
setundobuffer() 设置撤消缓冲区
undobufferentries() 撤消缓冲区条目数

TurtleScreen/Screen 方法

窗口控制

bgcolor() 背景颜色
bgpic() 背景图片
clearscreen()
resetscreen()
screensize() 屏幕大小
setworldcoordinates() 设置世界坐标系

动画控制

delay() 延迟
tracer() 追踪
update() 更新

使用屏幕事件

listen() 监听
onkey() | *onkeyrelease()* 当键盘按下并释放
onkeypress() 当键盘按下
onclick() | *onscreenclick()* 当点击屏幕
ontimer() 当达到定时
mainloop() | *done()* 主循环

设置与特殊方法

mode()
colormode() 颜色模式
getcanvas() 获取画布
getshapes() 获取形状
register_shape() | *addshape()* 添加形状
turtles() 所有海龟
window_height() 窗口高度
window_width() 窗口宽度

输入方法

textinput() 文本输入
numinput() 数字输入

Screen 专有方法

bye() 退出
exitonclick() 当点击时退出
setup() 设置
title() 标题

25.1.6 RawTurtle/Turtle 方法和对应函数

本节中的大部分示例都使用 Turtle 类的一个实例，命名为 `turtle`。

海龟动作

turtle.**forward**(*distance*)

turtle.**fd**(*distance*)

参数
distance -- 一个数值 (整型或浮点型)

海龟前进 *distance* 指定的距离，方向为海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

turtle.**back**(*distance*)

turtle.**bk**(*distance*)

turtle.**backward**(*distance*)

参数
distance -- 一个数值

海龟后退 *distance* 指定的距离，方向与海龟的朝向相反。不改变海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.**right**(*angle*)

turtle.**rt**(*angle*)

参数
angle -- 一个数值 (整型或浮点型)

海龟右转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.**left**(*angle*)

turtle.**lt**(*angle*)

参数
angle -- 一个数值 (整型或浮点型)

海龟左转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)

turtle.**setpos**(*x*, *y=None*)

turtle.**setposition**(*x*, *y=None*)

参数

- **x** -- 一个数值或数值对/向量
- **y** -- 一个数值或 None

如果 *y* 为 None，*x* 应为一个表示坐标的数值对或 *Vec2D* 类对象 (例如 *pos()* 返回的对象).

海龟移动到一个绝对坐标。如果画笔已落下将会画线。不改变海龟的朝向。

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**teleport**(*x*, *y=None*, *, *fill_gap=False*)

参数

- **x** -- 一个数值或 None
- **y** -- 一个数值或 None
- **fill_gap** -- 布尔

将海龟移到某个绝对位置。不同于 goto(x, y)，这将不会画一条线段。海龟的方向不变。如果当前正在填充，离开后原位置上的多边形将被填充，在移位后将再次开始填充。这可以通过 fill_gap=True 来禁用，此设置将使在移位期间海龟的移动轨迹线像在 goto(x, y) 中一样被当作填充边缘。

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.teleport(60)
>>> turtle.pos()
(60.00,0.00)
>>> turtle.teleport(y=10)
>>> turtle.pos()
(60.00,10.00)
>>> turtle.teleport(20, 30)
>>> turtle.pos()
(20.00,30.00)
```

Added in version 3.12.

turtle.**setx**(*x*)

参数

x -- 一个数值 (整型或浮点型)

设置海龟的横坐标为 *x*，纵坐标保持不变。

```
>>> turtle.position()
(0.00,240.00)
```

(续下页)

(接上页)

```
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

参数
y -- 一个数值 (整型或浮点型)

设置海龟的纵坐标为 *y*，横坐标保持不变。

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

参数
to_angle -- 一个数值 (整型或浮点型)

设置海龟的朝向为 *to_angle*。以下是以角度表示的几个常用方向：

标准模式	logo 模式
0 - 东	0 - 北
90 - 北	90 - 东
180 - 西	180 - 南
270 - 南	270 - 西

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()
海龟移至初始坐标 (0,0)，并设置朝向为初始方向 (由海龟模式确定，参见*mode()*)。

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

参数

- **radius** -- 一个数值
- **extent** -- 一个数值 (或 None)
- **steps** -- 一个整型数 (或 None)

绘制一个 *radius* 指定半径的圆。圆心在海龟左边 *radius* 个单位；*extent* 为一个夹角，用来决定绘制圆的一部分。如未指定 *extent** 则绘制整个圆。如果 **extent* 不是完整圆周，则以当前画笔位置为一

个端点绘制圆弧。如果 *radius* 为正值则朝逆时针方向绘制圆弧，否则朝顺时针方向。最终海龟的朝向会依据 *extent* 的值而改变。

圆实际是以其内切正多边形来近似表示的，其边的数量由 *steps* 指定。如果未指定边数则会自动确定。此方法也可用来绘制正多边形。

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # 画一个半圆
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

参数

- **size** -- 一个整型数 >= 1 (如果指定)
- **color** -- 一个颜色字符串或颜色数值元组

绘制一个直径为 *size*，颜色为 *color* 的圆点。如果 *size* 未指定，则直径取 pensize+4 和 2*pensize 中的较大值。

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

在海龟当前位置印制一个海龟形状。返回该印章的 stamp_id，印章可以通过调用 `clearstamp(stamp_id)` 来删除。

```
>>> turtle.color("blue")
>>> stamp_id = turtle.stamp()
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

参数

stampid -- 一个整型数，必须是之前 *stamp()* 调用的返回值

删除 *stampid* 指定的印章。

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
```

(续下页)

(接上页)

```
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

参数
n -- 一个整型数 (或 None)

删除全部或前/后 *n* 个海龟印章。如果 *n* 为 None 则删除全部印章，如果 $n > 0$ 则删除前 *n* 个印章，否则如果 $n < 0$ 则删除后 *n* 个印章。

```
>>> for i in range(8):
...     unused_stamp_id = turtle.stamp()
...     turtle.fd(30)
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

撤消 (或连续撤消) 最近的一个 (或多个) 海龟动作。可撤消的次数由撤消缓冲区的大小决定。

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

参数
speed -- 一个 0..10 范围内的整型数或速度字符串 (见下)

设置海龟移动的速度为 0..10 表示的整型数值。如未指定参数则返回当前速度。

如果输入数值大于 10 或小于 0.5 则速度设为 0。速度字符串与速度值的对应关系如下:

- ”fastest”: 0 最快
- ”fast”: 10 快
- ”normal”: 6 正常
- ”slow”: 3 慢
- ”slowest”: 1 最慢

速度值从 1 到 10，画线和海龟转向的动画效果逐级加快。

注意: *speed* = 0 表示 没有动画效果。forward/back 将使海龟向前/向后跳跃，同样的 left/right 将使海龟立即改变朝向。

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

获取海龟的状态

turtle.**position**()

turtle.**pos**()

返回海龟当前的坐标 (x,y) (为*Vec2D* 矢量类对象)。

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

参数

- **x** -- 一个数值或数值对/矢量，或一个海龟实例
- **y** -- 一个数值——如果 *x* 是一个数值，否则为 None

返回从海龟位置到由 (x,y)、矢量或另一海龟所确定位置的连线的夹角。此数值依赖于海龟的初始朝向，这又取决于”standard”/”world” 或”logo” 模式设置。

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()

返回海龟的 x 坐标。

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()

返回海龟的 y 坐标。

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()

返回海龟当前的朝向 (数值依赖于海龟模式参见*mode()*)。

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

参数

- **x** -- 一个数值或数值对/矢量，或一个海龟实例
- **y** -- 一个数值——如果 *x* 是一个数值，否则为 None

返回从海龟位置到由 (x,y)，适量或另一海龟对应位置的单位距离。

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

度量单位设置

turtle.**degrees**(*fullcircle=360.0*)

参数
fullcircle -- 一个数值

设置角度的度量单位，即设置一个圆周为多少” 度”。默认值为 360 度。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

>>> # 将角度计量单位改为 grad (或称 gon, grade
>>> # 或 gradian, 等于直角的 1/100。)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

设置角度的度量单位为弧度。其值等于 `degrees(2*math.pi)`。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

画笔控制

绘图状态

turtle.**pendown**()

turtle.**pd**()

turtle.**down**()

画笔落下 -- 移动时将画线。

turtle.**penup**()

turtle.**pu**()

turtle.**up**()

画笔抬起 -- 移动时不画线。

turtle.**pensize**(*width=None*)

turtle.**width**(*width=None*)

参数
width -- 一个正数值

设置线条的粗细为 *width* 或返回该值。如果 resizemode 设为"auto" 并且 turtleshape 为多边形，该多边形也以同样组细的线条绘制。如未指定参数，则返回当前的 pensize。

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # 从这里开始，画出宽度为10的线
```

turtle.**pen**(*pen=None*, ***pendict*)

参数

- **pen** -- 一个包含部分或全部下列键的字典
- **pendict** -- 一个或多个以下列键为关键字的关键字参数

返回或设置画笔的属性，以一个包含以下键值对的" 画笔字典" 表示:

- "shown": True/False
- "pendown": True/False
- "pencolor": 颜色字符串或颜色元组
- "fillcolor": 颜色字符串或颜色元组
- "pensize": 正数值
- "speed": 0..10 范围内的数值
- "resizemode": "auto" 或"user" 或"noresize"
- "stretchfactor": (正数值, 正数值)
- "outline": 正数值
- "tilt": 数值

此字典可作为后续调用*pen()* 时的参数，以恢复之前的画笔状态。另外还可将这些属性作为关键词参数提交。使用此方式可以用一条语句设置画笔的多个属性。

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.**isdown**()

如果画笔落下返回 `True`，如果画笔抬起返回 `False`。

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
```

(续下页)

(接上页)

```
>>> turtle.isdown()
True
```

颜色控制

turtle.**pencolor**(**args*)

返回或设置画笔颜色。

允许以下四种输入格式:

pencolor()
返回以颜色描述字符串或元组(见示例)表示的当前画笔颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

pencolor(colorstring)
设置画笔颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 `"red"`、`"yellow"` 或 `"#33cc8c"`。

pencolor((r, g, b))
设置画笔颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

pencolor(r, g, b)
设置画笔颜色为以 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形轮廓也以新设置的画笔颜色绘制。

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

返回或设置填充颜色。

允许以下四种输入格式:

fillcolor()
返回以颜色描述字符串或元组(见示例)表示的当前填充颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

fillcolor(colorstring)
设置填充颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 `"red"`、`"yellow"` 或 `"#33cc8c"`。

fillcolor((r, g, b))
设置填充颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

fillcolor(r, g, b)
设置填充颜色为 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形内部也以新设置的填充颜色填充。

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # 整数，而非浮点数
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)

返回或设置画笔颜色和填充颜色。

允许多种输入格式。使用如下 0 至 3 个参数:

color()
返回以一对颜色描述字符串或元组表示的当前画笔颜色和填充颜色，两者可分别由*pencolor()* 和*fillcolor()* 返回。

color(colorstring), color((r,g,b)), color(r,g,b)
输入格式与*pencolor()* 相同，同时设置填充颜色和画笔颜色为指定的值。

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))
相当于 `pencolor(colorstring1)` 加 `fillcolor(colorstring2)`，使用其他输入格式的方法也与之类似。

如果 turtleshape 为多边形，该多边形轮廓与填充也使用新设置的颜色。

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

另参见: Screen 方法*colormode()*。

填充

turtle.**filling**()

返回填充状态 (填充为 `True`，否则为 `False`)。

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**begin_fill**()

在绘制要填充的形状之前调用。

turtle.**end_fill**()

填充上次调用*begin_fill()* 之后绘制的形状。

自相交多边形或多个形状间的重叠区域是否填充取决于操作系统的图形引擎、重叠的类型以及重叠的层数。例如上面的 Turtle 多芒星可能会全部填充为黄色，也可能会有一些白色区域。

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

更多绘图控制

turtle.**reset**()

从屏幕中删除海龟的绘图，海龟回到原点并设置所有变量为默认值。

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()

从屏幕中删除指定海龟的绘图。不移动海龟。海龟的状态和位置以及其他海龟的绘图不受影响。

turtle.**write**(*arg*, *move=False*, *align='left'*, *font=('Arial', 8, 'normal')*)

参数

- **arg** -- 要书写到 TurtleScreen 的对象
- **move** -- True/False
- **align** -- 字符串"left", "center" 或"right"
- **font** -- 一个三元组 (fontname, fontsize, fonttype)

基于 *align* ("left", "center" 或"right") 并使用给定的字体将文本——*arg* 的字符串表示形式——写到当前海龟位置。如果 *move* 为真值，画笔会移至文本的右下角。默认情况下 *move* 为 `False`。

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

海龟状态

可见性

turtle.**hideturtle**()

turtle.**ht**()

使海龟不可见。当你绘制复杂图形时这是个好主意，因为隐藏海龟可显著加快绘制速度。

```
>>> turtle.hideturtle()
```

turtle.**showturtle**()

turtle.**st**()

使海龟可见。

```
>>> turtle.showturtle()
```

turtle.**isvisible**()

如果海龟显示返回 `True`，如果海龟隐藏返回 `False`。

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

外观

turtle.**shape**(*name=None*)

参数
name -- 一个有效的形状名字符串

设置海龟形状为 *name* 指定的形状名，如未指定形状名则返回当前的形状名。*name* 指定的形状名应存在于 TurtleScreen 的 shape 字典中。多边形的形状初始时有以下几种: "arrow", "turtle", "circle", "square", "triangle", "classic"。要了解如何处理形状请参看 Screen 方法*register_shape()*。

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

参数
rmode -- 字符串"auto", "user", "noresize" 其中之一

设置大小调整模式为以下值之一: "auto", "user", "noresize"。如未指定 *rmode* 则返回当前的大小调整模式。不同的大小调整模式的效果如下:

- "auto": 根据画笔粗细值调整海龟的外观。
- "user": 根据拉伸因子和轮廓宽度 (outline) 值调整海龟的外观，两者是由*shapesize()* 设置的。
- "noresize": 不调整海龟的外观大小。

`resizemode("user")` 会由*shapesize()* 带参数使用时被调用。

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

参数

- **stretch_wid** -- 正数值
- **stretch_len** -- 正数值
- **outline** -- 正数值

返回或设置画笔的属性 x/y 拉伸因子和/或轮廓。设置大小调整模式为"user"。当且仅当大小调整模式为"user" 时，海龟会基于其拉伸因子调整外观: *stretch_wid* 为垂直于其朝向的宽度拉伸因子，*stretch_len* 为平行于其朝向的长度拉伸因子，*outline* 决定形状轮廓线的宽度。

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

参数
shear -- 数值 (可选)

设置或返回当前的剪切因子。根据 share 指定的剪切因子即剪切角度的切线来剪切海龟形状。不改变海龟的朝向 (移动方向)。如未指定 shear 参数: 返回当前的剪切因子即剪切角度的切线，与海龟朝向平行的线条将被剪切。

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

参数
angle -- 一个数值

海龟形状自其当前的倾角转动 *angle* 指定的角度，但 不改变海龟的朝向 (移动方向)。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**tiltangle**(*angle=None*)

参数
angle -- 一个数值 (可选)

设置或返回当前的倾角。如果指定 angle 则旋转海龟形状使其指向 angle 指定的方向，忽略其当前的倾角。不改变海龟的朝向 (移动方向)。如果未指定 angle: 返回当前的倾角，即海龟形状的方向和海龟朝向 (移动方向) 之间的夹角。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

参数

- **t11** -- 一个数值 (可选)
- **t12** -- 一个数值 (可选)
- **t21** -- 一个数值 (可选)
- **t12** -- 一个数值 (可选)

设置或返回海龟形状的当前变形矩阵。

如未指定任何矩阵元素，则返回以 4 元素元组表示的变形矩阵。否则就根据设置指定元素的矩阵来改变海龟形状，矩阵第一排的值为 t11, t12 而第二排的值为 t21, t22。行列式 t11 * t22 - t12 * t21 必须不为零，否则会引发错误。根据指定矩阵修改拉伸因子 stretchfactor, 剪切因子 shearfactor 和倾角 tiltangle。

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

返回以坐标值对元组表示的当前形状多边形。这可以用于定义一个新形状或一个复合形状的多个组成部分。

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

使用事件

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

参数

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- `True` 或 `False` -- 如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到鼠标点击此海龟事件。如果 *fun* 值为 `None`，则移除现有的绑定。以下为使用匿名海龟即过程式的示例:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # 现在点击海龟将使其转向。
>>> onclick(None)  # 事件绑定将被移除
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

参数

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- `True` 或 `False` -- 如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上释放鼠标按键事件。如果 *fun* 值为 `None`，则移除现有的绑定。

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # 点击turtle会将填充颜色设置为红色。
>>> turtle.onrelease(turtle.unglow) # 释放会使它变得透明
```

turtle.**ondrag**(*fun*, *btn=1*, *add=None*)

参数

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。

- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- `True` 或 `False` -- 如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上移动鼠标事件。如果 *fun* 值为 `None`，则移除现有的绑定。

注: 在海龟上移动鼠标事件之前应先发生在此海龟上点击鼠标事件。

```
>>> turtle.ondrag(turtle.goto)
```

在此之后点击并拖动海龟可在屏幕上手绘线条 (如果画笔为落下)。

特殊海龟方法

`turtle.`**`begin_poly`**`()`

开始记录多边形的顶点。当前海龟位置为多边形的第一个顶点。

`turtle.`**`end_poly`**`()`

停止记录多边形的顶点。当前海龟位置为多边形的最后一个顶点。它将连线到第一个顶点。

`turtle.`**`get_poly`**`()`

返回最新记录的多边形。

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

`turtle.`**`clone`**`()`

创建并返回海龟的克隆体，具有相同的位置、朝向和海龟属性。

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

`turtle.`**`getturtle`**`()`

`turtle.`**`getpen`**`()`

返回海龟对象自身。唯一合理的用法: 作为一个函数来返回” 匿名海龟”:

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

`turtle.`**`getscreen`**`()`

返回作为海龟绘图场所的*`TurtleScreen`* 类对象。该对象将可调用 TurtleScreen 方法。

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

`turtle.`**`setundobuffer`**`(`*`size`*`)`

参数

size -- 一个整型数值或 `None`

设置或禁用撤销缓冲区。如果 *size* 为整数，则开辟一个给定大小的空撤销缓冲区。*size* 给出了可以通过*undo()* 方法/函数撤销海龟动作的最大次数。如果 *size* 为 None，则禁用撤销缓冲区。

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()

返回撤销缓冲区里的条目数。

```
>>> while undobufferentries():
...     undo()
```

复合形状

要使用由多个不同颜色多边形构成的复合海龟形状，你必须明确地使用辅助类*Shape*，具体步骤如下:

1. 创建一个空 Shape 对象，类型为"compound"。
2. 可根据需要使用*addcomponent()* 方法向此对象添加多个组件。

   例如:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. 接下来将 Shape 对象添加到 Screen 对象的形状列表并使用它:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

备注

Shape 类在*register_shape()* 方法的内部以多种方式使用。应用程序编写者 只有在使用上述的复合形状时才需要处理 Shape 类。

25.1.7 TurtleScreen/Screen 方法及对应函数

本节中的大部分示例都使用 TurtleScreen 类的一个实例，命名为 screen。

窗口控制

turtle.**bgcolor**(**args*)

参数

args -- 一个颜色字符串或三个取值范围 0..colormode 内的数值或一个取值范围相同的数值 3 元组

设置或返回 TurtleScreen 的背景颜色。

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

turtle.**bgpic**(*picname=None*)

参数

picname -- 一个字符串, gif-文件名, `"nopic"`, 或 `None`

设置背景图片或返回当前背景图片名称。如果 *picname* 为一个文件名，则将相应图片设为背景。如果 *picname* 为 `"nopic"`，则删除当前背景图片。如果 *picname* 为 `None`，则返回当前背景图片文件名。:

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**`()`

备注

此 TurtleScreen 方法作为全局函数时只有一个名字 `clearscreen`。全局函数 `clear` 所对应的是 Turtle 方法 `clear`。

`turtle.`**`clearscreen`**`()`

从中删除所有海龟的全部绘图。将已清空的 TurtleScreen 重置为初始状态: 白色背景，无背景片，无事件绑定并启用追踪。

`turtle.`**`reset`**`()`

备注

此 TurtleScreen 方法作为全局函数时只有一个名字 `resetscreen`。全局函数 `reset` 所对应的是 Turtle 方法 `reset`。

`turtle.`**`resetscreen`**`()`

重置屏幕上的所有海龟为其初始状态。

`turtle.`**`screensize`**`(`*canvwidth=None*, *canvheight=None*, *bg=None*`)`

参数

- **canvwidth** -- 正整型数，以像素表示画布的新宽度值
- **canvheight** -- 正整型数，以像素表示画面的新高度值
- **bg** -- 颜色字符串或颜色元组，新的背景颜色

如未指定任何参数，则返回当前的 (canvaswidth, canvasheight)。否则改变作为海龟绘图场所的画布大小。不改变绘图窗口。要观察画布的隐藏区域，可以使用滚动条。通过此方法可以令之前绘制于画布之外的图形变为可见。

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

也可以用来寻找意外逃走的海龟;-)

`turtle.`**`setworldcoordinates`**`(`*llx*, *lly*, *urx*, *ury*`)`

参数

- **llx** -- 一个数值, 画布左下角的 x-坐标

- **lly** -- 一个数值, 画布左下角的 y-坐标
- **urx** -- 一个数值, 画面右上角的 x-坐标
- **ury** -- 一个数值, 画布右上角的 y-坐标

设置用户自定义坐标系并在必要时切换模式为"world"。这会执行一次 `screen.reset()`。如果"world"模式已激活，则所有图形将根据新的坐标系重绘。

注意: 在用户自定义坐标系中，角度可能显得扭曲。

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # 一个正八边形
```

动画控制

`turtle.`**`delay`**(*delay=None*)

参数

delay -- 正整型数

设置或返回以毫秒数表示的延迟值 *delay*。(这约等于连续两次画布刷新的间隔时间。) 绘图延迟越长，动画速度越慢。

可选参数:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

`turtle.`**`tracer`**(*n=None*, *delay=None*)

参数

- **n** -- 非负整型数
- **delay** -- 非负整型数

启用/禁用海龟动画并设置刷新图形的延迟时间。如果指定 *n* 值，则只有每第 n 次屏幕刷新会实际执行。(可被用来加速复杂图形的绘制。) 如果调用时不带参数，则返回当前保存的 n 值。第二个参数设置延迟值 (参见*delay()*)。

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

`turtle.`**`update`**()

执行一次 TurtleScreen 刷新。在禁用追踪时使用。

另参见 RawTurtle/Turtle 方法*speed()*。

使用屏幕事件

turtle.**listen**(*xdummy=None*, *ydummy=None*)

设置焦点到 TurtleScreen (以便接收按键事件)。使用两个 Dummy 参数以便能够传递 *listen()* 给 onclick 方法。

turtle.**onkey**(*fun*, *key*)

turtle.**onkeyrelease**(*fun*, *key*)

参数

- **fun** -- 一个无参数的函数或 None
- **key** -- 一个字符串: 键 (例如"a") 或键标 (例如"space")

绑定 *fun* 指定的函数到按键释放事件。如果 *fun* 值为 None，则移除事件绑定。注: 为了能够注册按键事件，TurtleScreen 必须得到焦点。(参见 method *listen()* 方法。)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

参数

- **fun** -- 一个无参数的函数或 None
- **key** -- 一个字符串: 键 (例如"a") 或键标 (例如"space")

绑定 *fun* 指定的函数到指定键的按下事件。如未指定键则绑定到任意键的按下事件。注: 为了能够注册按键事件，必须得到焦点。(参见 *listen()* 方法。)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

参数

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- True 或 False -- 如为 True 则将添加一个新绑定，否则将取代先前的绑定

绑定 *fun* 指定的函数到鼠标点击屏幕事件。如果 *fun* 值为 None，则移除现有的绑定。

以下示例使用一个 TurtleScreen 实例 screen 和一个 Turtle 实例 turtle:

```
>>> screen.onclick(turtle.goto) # 后续对 TurtleScreen 的点击
>>>                             # 将使海龟移至被点击的位置。
>>> screen.onclick(None)        # 再次移除事件绑定
```

备注

此 TurtleScreen 方法作为全局函数时只有一个名字 onscreenclick。全局函数 onclick 所对应的是 Turtle 方法 onclick。

turtle.**ontimer**(*fun*, *t=0*)

参数

- **fun** -- 一个无参数的函数
- **t** -- 一个数值 >= 0

安装一个计时器，在 *t* 毫秒后调用 *fun* 函数。

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### 让海龟随意前进
>>> running = False
```

turtle.**mainloop**()

turtle.**done**()

开始事件循环 - 调用 Tkinter 的 mainloop 函数。必须作为一个海龟绘图程序的结束语句。如果一个脚本是在以 -n 模式 (无子进程) 启动的 IDLE 中运行时 不可使用 - 用于实现海龟绘图的交互功能。:

```
>>> screen.mainloop()
```

输入方法

turtle.**textinput**(*title*, *prompt*)

参数

- **title** -- string
- **prompt** -- string

弹出一个对话框窗口用来输入一个字符串。形参 title 为对话框窗口的标题，prompt 为一条文本，通常用来提示要输入什么信息。返回输入的字符串。如果对话框被取消则返回 None。:

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

参数

- **title** -- string
- **prompt** -- string
- **default** -- 数值 (可选)
- **minval** -- 数值 (可选)
- **maxval** -- 数值 (可选)

弹出一个用于输入数值的对话框窗口。title 是对话框窗口的标题，prompt 是通常用来描述要输入的数字信息的文本。default: 默认值, minval: 可输入的最小值, maxval: 可输入的最大值。如果给出 minval .. maxval 则输入的数值必须在此范围以内。如未给出，则将发出提示并且让话框保持打开以便修正。返回输入的数值。如果对话框被取消，则返回 None。

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

设置与特殊方法

turtle.**mode**(*mode=None*)

参数
mode -- 字符串"standard", "logo" 或"world" 其中之一

设置海龟模式 ("standard", "logo" 或"world") 并执行重置。如未指定模式则返回当前的模式。

"standard" 模式与旧的*turtle* 兼容。"logo" 模式与大部分 Logo 海龟绘图兼容。"world" 模式使用用户自定义的" 世界坐标系"。**注意**: 在此模式下，如果 x/y 单位比率不等于 1 则角度会显得扭曲。

模式	初始海龟朝向	正数角度
"standard"	朝右 (东)	逆时针
"logo"	朝上 (北)	顺时针

```
>>> mode("logo")   # 将海龟重置为朝向北方
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

参数
cmode -- 数值 1.0 或 255 其中之一

返回 colormode 或将其设为 1.0 或 255。后续表示三原色的 *r*, *g*, *b* 值必须在 0..*cmode* 范围之内。

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

返回此 TurtleScreen 的 Canvas 对象。供了解 Tkinter 的 Canvas 对象内部机理的人士使用。

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

返回所有当前可用海龟形状的列表。

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)

turtle.**addshape**(*name*, *shape=None*)

调用此函数有三种不同方式:

(1) *name* 为一个 gif 文件的文件名，*shape* 为 None: 安装相应的图像形状。:

```
>>> screen.register_shape("turtle.gif")
```

> **备注**
>
> 当海龟转向时图像形状 不会转动，因此无法显示海龟的朝向!

(2) *name* 为指定的字符串，*shape* 为由坐标值对构成的元组: 安装相应的多边形形状。

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* 为任意字符串而 *shape* 为 (复合) *Shape* 对象: 安装相应的复合形状。Install the corresponding compound shape.

将一个海龟形状加入 TurtleScreen 的形状列表。只有这样注册过的形状才能通过执行 `shape(shapename)` 命令来使用。

`turtle.`**`turtles`**`()`

返回屏幕上的海龟列表。

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`

返回海龟窗口的高度。:

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`

返回海龟窗口的宽度。:

```
>>> screen.window_width()
640
```

Screen 专有方法, 而非继承自 TurtleScreen

`turtle.`**`bye`**`()`

关闭海龟绘图窗口。

`turtle.`**`exitonclick`**`()`

将 `bye()` 方法绑定到 Screen 上的鼠标点击事件。

如果配置字典中"using_IDLE" 的值为 `False` (默认值) 则同时进入主事件循环。注: 如果启动 IDLE 时使用了 `-n` 开关 (无子进程)，`turtle.cfg` 中此数值应设为 `True`。在此情况下 IDLE 本身的主事件循环同样会作用于客户脚本。

`turtle.`**`setup`**`(`*width=_CFG['width']*, *height=_CFG['height']*, *startx=_CFG['leftright']*, *starty=_CFG['topbottom']*`)`

设置主窗口的大小和位置。默认参数值保存在配置字典中，可通过 `turtle.cfg` 文件进行修改。

参数

- **`width`** -- 如为一个整型数值，表示大小为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 50%
- **`height`** -- 如为一个整型数值，表示高度为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 75%
- **`startx`** -- 如为正值，表示初始位置距离屏幕左边缘多少像素，负值表示距离右边缘，`None` 表示窗口水平居中
- **`starty`** -- 如为正值，表示初始位置距离屏幕上边缘多少像素，负值表示距离下边缘，`None` 表示窗口垂直居中

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # 设置窗口为 200x200 像素，位于屏幕左上角
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # 设置窗口宽度为屏幕的 75% 高度为屏幕的 50% 并居中
```

turtle.**title**(*titlestring*)

参数

titlestring -- 一个字符串，显示为海龟绘图窗口的标题栏文本

设置海龟窗口标题为 *titlestring* 指定的文本。

```
>>> screen.title("Welcome to the turtle zoo!")
```

25.1.8 公共类

class turtle.**RawTurtle**(*canvas*)

class turtle.**RawPen**(*canvas*)

参数

canvas -- 一个 tkinter.Canvas, *ScrolledCanvas* 或 *TurtleScreen*

创建一个海龟。海龟对象具有"Turtle/RawTurtle 方法" 一节所述的全部方法。

class turtle.**Turtle**

RawTurtle 的子类，具有相同的接口，但其绘图场所为默认的 *Screen* 类对象，在首次使用时自动创建。

class turtle.**TurtleScreen**(*cv*)

参数

cv -- 一个 tkinter.Canvas

提供面向屏幕的方法如 *bgcolor()* 等。说明见上文。

class turtle.**Screen**

TurtleScreen 的子类，*增加了四个方法*.

class turtle.**ScrolledCanvas**(*master*)

参数

master -- 可容纳 ScrolledCanvas 的 Tkinter 部件，即添加了滚动条的 Tkinter-canvas

由 Screen 类使用，使其能够自动提供一个 ScrolledCanvas 作为海龟的绘图场所。

class turtle.**Shape**(*type_*, *data*)

参数

type_ -- 字符串"polygon", "image", "compound" 其中之一

实现形状的数据结构。(type_, data) 必须遵循以下定义:

*type_*	*data*
"polygon"	一个多边形元组，即由坐标值对构成的元组
"image"	一个图片 (此形式仅限内部使用!)
"compound"	None (复合形状必须使用 *addcomponent()* 方法来构建)

addcomponent(*poly*, *fill*, *outline=None*)

参数

- **poly** -- 一个多边形，即由数值对构成的元组

- **fill** -- 一种颜色，将用来填充 *poly* 指定的多边形
- **outline** -- 一种颜色，用于多边形的轮廓 (如有指定)

示例:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... 添加更多组件，然后使用 register_shape()
```

参见复合形状。

class turtle.**Vec2D**(*x*, *y*)

一个二维矢量类，用来作为实现海龟绘图的辅助类。也可能在海龟绘图程序中使用。派生自元组，因此矢量也属于元组!

提供的运算 (*a*, *b* 为矢量, *k* 为数值):

- `a + b` 矢量加法
- `a - b` 矢量减法
- `a * b` 内积
- `k * a` 和 `a * k` 与标量相乘
- `abs(a)` a 的绝对值
- `a.rotate(angle)` 旋转

25.1.9 说明

海龟对象在屏幕对象上绘图，在 turtle 的面向对象接口中有许多关键的类可被用于创建它们并将它们相互关联。

Turtle 实例将自动创建一个*Screen* 实例，如果它还未创建的话。

`Turtle` 是*RawTurtle* 的子类，它 不会自动创建绘图区域——需要为其提供或创建一个 *canvas*。*canvas* 可以是一个 `tkinter.Canvas`, *ScrolledCanvas* 或*TurtleScreen*。

TurtleScreen 是基本的海龟绘图区域。*Screen* 是 `TurtleScreen` 的子类，并包括一些额外方法用来管理其外观（包括大小和标题）及行为。`TurtleScreen` 的构造器需要一个 `tkinter.Canvas` 或*ScrolledCanvas* 作为参数。

海龟绘图形的函数式接口使用 `Turtle` 和 `TurtleScreen`/`Screen` 的各种方法。在下层，每当从 `Screen` 方法派生的函数被调用时就会自动创建一个屏幕对象。同样地，每当从 Turtle 方法派生的函数被调用时也都会自动创建一个 Turtle 对象。

要在一个屏幕中使用多个海龟，就必须使用面向对象的接口。

25.1.10 帮助与配置

如何使用帮助

Screen 和 Turtle 类的公用方法以文档字符串提供了详细的文档。因此可以利用 Python 帮助工具获取这些在线帮助信息:

- 当使用 IDLE 时，输入函数/方法调用将弹出工具提示显示其签名和文档字符串的头几行。
- 对文法或函数调用*help()* 将显示其文档字符串:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.
```

(续下页)

(接上页)

```
    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- 方法对应函数的文档字符串的形式会有一些修改:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

这些修改版文档字符串是在导入时与方法对应函数的定义一起自动生成的。

文档字符串翻译为不同的语言

可使用工具创建一个字典，键为方法名，值为 Screen 和 Turtle 类公共方法的文档字符串。

turtle.**write_docstringdict**(*filename='turtle_docstringdict'*)

参数
filename -- 一个字符串，表示文件名

创建文档字符串字典并将其写入 filename 指定的 Python 脚本文件。此函数必须显示地调用 (海龟绘图类并不使用此函数)。文档字符串字典将被写入到 Python 脚本文件 *filename*.py。该文件可作为模板用来将文档字符串翻译为不同语言。

如果你 (或你的学生) 想使用本国语言版本的*turtle* 在线帮助，你必须翻译文档字符串并保存结果文件，例如 turtle_docstringdict_german.py.

如果你在 turtle.cfg 文件中加入了相应的条目，此字典将在导入模块时被读取并替代原有的英文版文档字符串。

在撰写本文档时已经有了德语和意大利语版的文档字符串字典。(更多需求请联系 glingl@aon.at)

如何配置 Screen 和 Turtle

内置的默认配置是模仿旧 turtle 模块的外观和行为，以便尽可能地与其保持兼容。

如果你想使用不同的配置，以便更好地反映此模块的特性或是更适合你的需求，例如在课堂中使用，你可以准备一个配置文件 turtle.cfg，该文件将在导入模块时被读取并根据其中的设定修改模块配置。

内置的配置对应了下面的 turtle.cfg:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

选定条目的简短说明:

- 开头的四行对应了*Screen.setup* 方法的参数。
- 第 5 和第 6 行对应于*Screen.screensize* 方法的参数。
- *shape* 可以是任何内置形状，即: arrow, turtle 等。更多信息可用 help(shape) 查看。
- 如果你想使用无填充色（即让海龟变透明），则你必须写 fillcolor = "" (但在 but all nonempty strings must not have quotes in the cfg 文件中所有非空字符串都不可加引号)。
- 如果你想令海龟反映其状态，你必须使用 resizemode = auto。
- 例如当你设置了 language = italian 则文档字符串字典 turtle_docstringdict_italian.py 将在导入时被加载（如果它存在于导入路径，即与*turtle* 相同的目录中）。

- *exampleturtle* 和 *examplescreen* 条目定义了相应对象在文档字符串中显示的名称。方法文档字符串转换为函数文档字符串时将从文档字符串中删去这些名称。
- *using_IDLE*: 如果你经常使用 IDLE 及其 `-n` 开关选项（”无子进程”）则将此项设为 `True`。这将阻止*exitonclick()* 进入主事件循环。

`turtle.cfg` 文件可以保存于*turtle* 所在目录，当前工作目录也可以有一个同名文件。后者会重载覆盖前者的设置。

`Lib/turtledemo` 目录中也有一个 `turtle.cfg` 文件。你可以将其作为示例进行研究，并在运行演示时查看其作用效果 (但最好不要在演示查看器中运行)。

25.1.11 `turtledemo` --- 演示脚本集

turtledemo 包汇集了一组演示脚本。这些脚本可以通过以下命令打开所提供的演示查看器运行和查看:

```
python -m turtledemo
```

此外，你也可以单独运行其中的演示脚本。例如，:

```
python -m turtledemo.bytedesign
```

turtledemo 包目录中的内容:

- 一个演示查看器 `__main__.py`，可用来查看脚本的源码并即时运行。
- 多个脚本文件，演示*turtle* 模块的不同特性。所有示例可通过 Examples 菜单打开。也可以单独运行每个脚本。
- 一个 `turtle.cfg` 文件，作为说明如何编写并使用模块配置文件的示例模板。

演示脚本清单如下:

名称	描述	相关特性
bytedesign	复杂的传统海龟绘图模式	*tracer()*, delay, *update()*
chaos	绘制 Verhulst 动态模型，演示通过计算机的运算可能会生成令人惊叹的结果	世界坐标系
clock	绘制模拟时钟显示本机的当前时间	海龟作为表针, ontimer
colormixer	试验 r, g, b 颜色模式	*ondrag()* 当鼠标拖动
forest	绘制 3 棵广度优先树	随机化
fractalcurves	绘制 Hilbert & Koch 曲线	递归
lindenmayer	文化数学 (印度装饰艺术)	L-系统
minimal_hanoi	汉诺塔	矩形海龟作为汉诺盘 (shape, shapesize)
nim	玩经典的“尼姆”游戏，开始时有三堆小棒，与电脑对战。	海龟作为小棒，事件驱动 (鼠标, 键盘)
paint	超极简主义绘画程序	*onclick()* 当鼠标点击
peace	初级技巧	海龟: 外观与动画
penrose	非周期性地使用风筝和飞镖形状铺满平面	*stamp()* 印章
planet_and_moon	模拟引力系统	复合开关, *Vec2D* 类
rosette	一个来自介绍海龟绘图的维基百科文章的图案	*clone()*, *undo()*
round_dance	两两相对并不断旋转舞蹈的海龟	复合形状, clone shapesize, tilt, get_shapepoly, update
sorting_animate	动态演示不同的排序方法	简单对齐, 随机化
tree	一棵 (图形化的) 广度优先树 (使用生成器)	*clone()* 克隆
two_canvases	简单设计	两块画布上的海龟
yinyang	另一个初级示例	*circle()* 画圆

祝你玩得开心!

25.1.12 Python 2.6 之后的变化

- *Turtle.tracer*, *Turtle.window_width* 和 *Turtle.window_height* 等方法已被去除。具有这些名称和功能的方法现在只限于作为 *Screen* 的方法。自这些方法派生的函数仍保持可用。(实际上在 Python 2.6 中这些方法就已经只是对应 *TurtleScreen*/*Screen* 方法的副本了。)
- `Turtle.fill()` 方法已被去除。*begin_fill()* 和 *end_fill()* 的行为则有细微改变：现在每个填充过程必须以一个 `end_fill()` 调用来结束。
- 增加了一个 *Turtle.filling* 方法。该方法返回一个布尔值：如果填充过程正在运行则为 `True`，否则为 `False`。此行为对应于 Python 2.6 中一个不带参数的 `fill()` 调用。

25.1.13 Python 3.0 之后的变化

- 增加了 *Turtle* 方法 *shearfactor()*, *shapetransform()* 和 *get_shapepoly()*。这样就可以使用所有的常规线性变换来改变海龟形状。*tiltangle()* 的功能已得到加强：现在它可以被用来获取或设置倾斜角度。
- 增加了 *Screen* 方法 *onkeypress()* 作为 *onkey()* 的补充。当后者将动作绑定到松开按键事件时，还将为它添加一个别名：*onkeyrelease()*。
- 增加了 *Screen.mainloop* 方法，这样在操作 *Screen* 和 *Turtle* 对象时就无需再使用单独的 *mainloop()* 函数。
- 增加了两个输入方法：*Screen.textinput* 和 *Screen.numinput*。这两个方法会弹出输入对话框接受输入并分别返回字符串和数字。These pop up input dialogs and return strings and numbers respectively.
- 两个新的示例脚本 `tdemo_nim.py` 和 `tdemo_round_dance.py` 被加入到 `Lib/turtledemo` 目录中。

25.2 cmd --- 对面向行的命令解释器的支持

源代码: Lib/cmd.py

Cmd 类提供简单框架用于编写面向行的命令解释器。这些通常对测试工具，管理工具和原型有用，这些工具随后将被包含在更复杂的接口中。

class cmd.**Cmd**(*completekey='tab'*, *stdin=None*, *stdout=None*)

一个 *Cmd* 实例或子类实例是面向行的解释器框架结构。实例化 *Cmd* 本身是没有充分理由的，它作为自定义解释器类的超类是非常有用的为了继承 *Cmd* 的方法并且封装动作方法。

可选参数 *completekey* 是完成键的 *readline* 名称；默认是 `Tab` 。如果 *completekey* 不是 *None* 并且 *readline* 是可用的，命令完成会自动完成。

默认值 `'tab'` 会被特殊对待，以使其在任何 *readline.backend* 上都指向 `Tab` 键。根据特殊规则，如果 *readline.backend* 是 `editline`，`Cmd` 将使用 `'^I'` 而不是 `'tab'`。请注意其它的值不会以这种方式来处理，并可能仅适用于特定的后端。

可选参数 *stdin* 和 *stdout* 指定了 Cmd 实例或子类实例将用于输入和输出的输入和输出文件对象。如果没有指定，他们将默认为 *sys.stdin* 和 *sys.stdout* 。

如果你想要使用一个给定的 *stdin* ，确保将实例的 *use_rawinput* 属性设置为 `False` ，否则 *stdin* 将被忽略。

在 3.13 版本发生变更: 对于 `editline` 来说 `completekey='tab'` 将被替换为 `'^I'`。

25.2.1 Cmd 对象

Cmd 实例有下列方法：

Cmd.**cmdloop**(*intro=None*)

反复发出提示，接受输入，从收到的输入中解析出一个初始前缀，并分派给操作方法，将其余的行作为参数传递给它们。

可选参数是在第一个提示之前发布的横幅或介绍字符串（这将覆盖*intro*类属性）。

如果*readline*继承模块被加载，输入将自动继承类似 **bash** 的历史列表编辑（例如，Control-P 滚动回到最后一个命令，Control-N 转到下一个命令，以 Control-F 非破坏性的方式向右 Control-B 移动光标，破坏性地等）。

输入的文件结束符被作为字符串传回 `'EOF'`。

当且仅当命令名称 foo 具有 do_foo() 方法时解释器实例才会识别它。存在一种特殊情况，以字符 `'?'` 开头的行将被分派给*do_help()*方法。还存在另一种特殊情况，以字符 `'!'` 开头的行将被分派给 do_shell() 方法（如果定义了该方法的话）。

当*postcmd()*方法返回真值时此方法将返回。*postcmd()*的 *stop* 参数是命令对应的 do_*() 方法的返回值。

如果启用了补全，则会自动完成命令的补全，命令参数的补全则是通过调用 complete_foo() 并附带参数 *text*, *line*, *begidx* 和 *endidx* 来完成的。*text* 是我们要尝试匹配的字符串前缀：所有被返回的匹配必须以它为开头。*line* 是去除了开头空白符的当前输入行，*begidx* 和 *endidx* 是前缀文本的开始和结束索引号，它们可被用来根据参数所在的位置提供不同的补全。

Cmd.**do_help**(*arg*)

所有*Cmd*的子类都继承了一个预定义的 do_help()。调用该方法时传入一个参数 `'bar'`，将唤起对应的方法 help_bar()，如果该方法不存在，则将打印 do_bar() 的文档字符串，如果有文档字符串的话。在没有参数的情况下，do_help() 将列出所有可用的帮助主题（即任何具有对应的 help_*() 方法的命令或具有文档字符串的命令），还会列出任何未写入文档的命令。

Cmd.**onecmd**(*str*)

解释该参数就好像它是为响应提示而键入的一样。此方法可以被重写，但通常不需要这样做；请参阅针对有用的执行钩子的*precmd()*和*postcmd()*方法。返回值是一个指明解释器对命令的解释是否应停止的旗标。如果对于命令 *str* 存在 do_*() 方法，则将返回该方法的返回值，否则将返回*default()*方法的返回值。

Cmd.**emptyline**()

在响应提示输入空行时调用的方法。如果此方法未被覆盖，则重复输入的最后一个非空命令。

Cmd.**default**(*line*)

当命令前缀不能被识别的时候在输入行调用的方法。如果此方法未被覆盖，它将输出一个错误信息并返回。

Cmd.**completedefault**(*text*, *line*, *begidx*, *endidx*)

当没有命令专属的 complete_*() 方法可供使用时将被调用以完成输入行的方法。在默认情况下，它将返回一个空列表。

Cmd.**columnize**(*list*, *displaywidth=80*)

调用以将一个字符串列表显示为紧凑的列集的方法。每列的宽度仅够显示其内容。各列之间以两个空格分隔以保证可读性。

Cmd.**precmd**(*line*)

钩方法在命令行 *line* 被解释之前执行，但是在输入提示被生成和发出后。这个方法是一个在*Cmd*中的存根；它的存在是为了被子类覆盖。返回值被用作*onecmd()*方法执行的命令；*precmd()*的实现或许会重写命令或者简单的返回 *line* 不变。

Cmd.**postcmd**(*stop*, *line*)

钩方法只在命令调度完成后执行。这个方法是一个在*Cmd*中的存根；它的存在是为了子类被覆盖。*line* 是被执行的命令行，*stop* 是一个表示在调用*postcmd()*之后是否终止执行的标志；这将作为*onecmd()*方法的返回值。这个方法的返回值被用作与 *stop* 相关联的内部标志的新值；返回 false 将导致解释继续。

Cmd.**preloop**()
: 钩方法当*cmdloop()* 被调用时执行一次。方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。

Cmd.**postloop**()
: 钩方法在*cmdloop()* 即将返回时执行一次。这个方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。

Instances of *Cmd* subclasses have some public instance variables:

Cmd.**prompt**
: 发出提示以请求输入。

Cmd.**identchars**
: 接受命令前缀的字符串。

Cmd.**lastcmd**
: 看到最后一个非空命令前缀。

Cmd.**cmdqueue**
: 排队的输入行列表。当需要新的输入时，在*cmdloop()* 中检查 cmdqueue 列表；如果它不是空的，它的元素将被按顺序处理，就像在提示符处输入一样。

Cmd.**intro**
: 要作为简介或横幅发出的字符串。可以通过给*cmdloop()* 方法一个参数来覆盖它。

Cmd.**doc_header**
: 如果帮助输出具有记录命令的段落，则发出头文件。

Cmd.**misc_header**
: 如果帮助输出包含一个杂项帮助主题小节时（也就是说，存在没有对应 `do_*()` 方法的 `help_*()` 方法）要发出的标题。

Cmd.**undoc_header**
: 如果帮助输出包含一个未写入文档的命令小节时（也就是说，存在没有对应 `help_*()` 方法的 `do_*()` 方法）要发出的标题。

Cmd.**ruler**
: 用于在帮助信息标题的下方绘制分隔符的字符，如果为空，则不绘制标尺线。这个字符默认是 `'='`。

Cmd.**use_rawinput**
: 一个旗标，默认为真值。如为真值，*cmdloop()* 将使用*input()* 显示一条提示并读取下一个命令；如为假值，则将使用*sys.stdout.write()* 和*sys.stdin.readline()*。（这意味着通过在受支持的系统上导入*readline*，解释器将自动支持类似 **Emacs** 的行编辑和命令历史按键操作。）

25.2.2 Cmd 例子

The *cmd* module is mainly useful for building custom shells that let a user work with a program interactively.

这部分提供了一个简单的例子来介绍如何使用一部分在*turtle* 模块中的命令构建一个 shell 。

基本 turtle 命令比如*forward()* 将被添加到一个具有名为 `do_forward()` 的方法的*Cmd* 子类。参数将被转换为数字并发送给 turtle 模块。文档字符串将被用于 shell 所提供的帮助工具。

该示例还包括一个通过*precmd()* 方法实现的基本录制和回放工具，这个方法负责将输入转换为小写形式并将命令写入到文件。`do_playback()` 方法将读取文件并将被录制的命令添加到*cmdqueue* 用于立即回放:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
```

(续下页)

(接上页)

```
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
    file = None

    # ----- 基本 turtle 命令 -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- 记录和回放 -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
        if self.file:
            self.file.close()
```

(续下页)

(接上页)

```
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

这是一个示例会话，其中 turtle shell 显示帮助功能，使用空行重复命令，以及简单的记录和回放功能：

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0
```

(续下页)

(接上页)

```
Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

25.3 shlex --- 简单词法分析

源代码： Lib/shlex.py

---

shlex 类可用于编写类似 Unix shell 的简单词法分析程序。通常可用于编写“迷你语言”（如 Python 应用程序的运行控制文件）或解析带引号的字符串。

shlex 模块中定义了以下函数：

shlex.**split**(*s*, *comments=False*, *posix=True*)

用类似 shell 的语法拆分字符串 *s*。如果 *comments* 为*False*(默认值)，则不会解析给定字符串中的注释 (*commenters* 属性的*shlex* 实例设为空字符串)。本函数默认工作于 POSIX 模式下，但若 *posix* 参数为 False，则采用非 POSIX 模式。

在 3.12 版本发生变更: 传入 None 作为 *s* 参数现在会引发异常，而不是读取*sys.stdin*。

shlex.**join**(*split_command*)

将列表 *split_command* 中的词法单元（token）串联起来，返回一个字符串。本函数是*split()* 的逆运算。

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

为防止注入漏洞，返回值是经过 shell 转义的（参见*quote()* ）。

Added in version 3.8.

shlex.**quote**(*s*)

返回经过 shell 转义的字符串 *s* 。返回值为字符串，可以安全地用作 shell 命令行中的词法单元，可用于不能使用列表的场合。

> **警告**
>
> shlex 模块 **仅适用于 Unix shell**。
>
> 在不兼容 POSIX 的 shell 或其他操作系统（如 Windows）的 shell 上，并不保证*quote()* 函数能够正常使用。在这种 shell 中执行用本模块包装过的命令，有可能会存在命令注入漏洞。
>
> 请考虑采用命令参数以列表形式给出的函数，比如带了 shell=False 参数的*subprocess.run()* 。

以下用法是不安全的：

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

用*quote()* 可以堵住这种安全漏洞：

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

这种包装方式兼容于 UNIX shell 和*split()*。

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Added in version 3.3.

shlex 模块中定义了以下类：

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

shlex 及其子类的实例是一种词义分析器对象。利用初始化参数可指定从哪里读取字符。初始化参数必须是具备*read()* 和*readline()* 方法的文件/流对象，或者是一个字符串。如果没有给出初始化参数，则会从 `sys.stdin` 获取输入。第二个可选参数是个文件名字符串，用于设置*infile* 属性的初始值。如果 *instream* 参数被省略或等于 `sys.stdin`，则第二个参数默认为"stdin"。*posix* 参数定义了操作的模式：若 *posix* 不为真值（默认），则*shlex* 实例将工作于兼容模式。若运行于 POSIX 模式下，则*shlex* 会尽可能地应用 POSIX shell 解析规则。*punctuation_chars* 参数提供了一种使行为更接近于真正的 shell 解析的方式。该参数可接受多种值：默认值、`False`、保持 Python 3.5 及更早版本的行为。如果设为 `True`，则会改变对字符 `();<>|&` 的解析方式：这些字符将作为独立的词法单元被返回（视作标点符号）。如果设为非空字符串，则这些字符将被用作标点符号。出现在 *punctuation_chars* 中的*wordchars* 属性中的任何字符都会从*wordchars* 中被删除。请参阅*改进的 shell 兼容性* 了解详情。*punctuation_chars* 只能在创建*shlex* 实例时设置，以后不能再作修改。

在 3.6 版本发生变更: 加入 *punctuation_chars* 参数。

参见

***configparser* 模块**
配置文件解析器，类似于 Windows 的 `.ini` 文件。

25.3.1 shlex 对象

shlex 实例具备以下方法：

shlex.**get_token**()

返回一个词法单元。如果所有单词已用*push_token()* 堆叠在一起了，则从堆栈中弹出一个词法单元。否则就从输入流中读取一个。如果读取时遇到文件结束符，则会返回 `eof`（在非 POSIX 模式下为空字符串 `''`，在 POSIX 模式下为 ``None`）。

shlex.**push_token**(*str*)

将参数值压入词法单元堆栈。

shlex.**read_token**()

读取一个原始词法单元。忽略堆栈，且不解释源请求。（通常没什么用，只是为了完整起见。）

shlex.**sourcehook**(*filename*)

当*shlex* 检测到源请求（见下面的*source*），以下词法单元可作为参数，并应返回一个由文件名和打开的文件对象组成的元组。

通常本方法会先移除参数中的引号。如果结果为绝对路径名，或者之前没有有效的源请求，或者之前的源请求是一个流对象（比如 `sys.stdin`），那么结果将不做处理。否则，如果结果是相对路径名，那么前面将会加上目录部分，目录名来自于源堆栈中前一个文件名（类似于 C 预处理器对 `#include "file.h"` 的处理方式）。

结果被视为一个文件名，并作为元组的第一部分返回，元组的第二部分以此为基础调用*open()* 获得。（注意：这与实例初始化过程中的参数顺序相反！）

此钩子函数是公开的，可用于实现路径搜索、添加文件扩展名或黑入其他命名空间。没有对应的“关闭”钩子函数，但 shlex 实例在返回 EOF 时会调用源输入流的*close()* 方法。

若要更明确地控制源堆栈，请采用*push_source()* 和*pop_source()* 方法。

shlex.**push_source**(*newstream*, *newfile=None*)

将输入源流压入输入堆栈。如果指定了文件名参数，以后错误信息中将会用到。*sourcehook()* 内部同样使用了本方法。

shlex.**pop_source**()

从输入堆栈中弹出最后一条输入源。当遇到输入流的 EOF 时，内部也使用同一方法。

shlex.**error_leader**(*infile=None*, *lineno=None*)

本方法生成一条错误信息的首部，以 Unix C 编译器错误标签的形式；格式为 `'"%s", line %d: '`，其中 `%s` 被替换为当前源文件的名称，`%d` 被替换为当前输入行号（可用可选参数覆盖）。

这是个快捷函数，旨在鼓励*shlex* 用户以标准的、可解析的格式生成错误信息，以便 Emacs 和其他 Unix 工具理解。

shlex 子类的实例有一些公共实例变量，这些变量可以控制词义分析，也可用于调试。

shlex.**commenters**

将被视为注释起始字符串。从注释起始字符串到行尾的所有字符都将被忽略。默认情况下只包括 `'#'`。

shlex.**wordchars**

可连成多字符词法单元的字符串。默认包含所有 ASCII 字母数字和下划线。在 POSIX 模式下，Latin-1 字符集的重音字符也被包括在内。如果*punctuation_chars* 不为空，则可出现在文件名规范和命令行参数中的 `~-./*?=` 字符也将包含在内，任何 `punctuation_chars` 中的字符将从 `wordchars` 中移除。如果*whitespace_split* 设为 `True`，则本规则无效。

shlex.**whitespace**

将被视为空白符并跳过的字符。空白符是词法单元的边界。默认包含空格、制表符、换行符和回车符。

shlex.**escape**

将视为转义字符。仅适用于 POSIX 模式，默认只包含 `'\'`。

shlex.**quotes**

将视为引号的字符。词法单元中的字符将会累至再次遇到同样的引号（因此，不同的引号会像在 shell 中一样相互包含。）默认包含 ASCII 单引号和双引号。

shlex.**escapedquotes**

quotes 中的字符将会解析*escape* 定义的转义字符。这只在 POSIX 模式下使用，默认只包含 `'"'`。

shlex.**whitespace_split**

若为 `True`，则只根据空白符拆分词法单元。这很有用，比如用*shlex* 解析命令行，用类似 shell 参数的方式读取各个词法单元。当与*punctuation_chars* 一起使用时，将根据空白符和这些字符拆分词法单元。

在 3.8 版本发生变更: *punctuation_chars* 属性已与*whitespace_split* 属性兼容。

shlex.**infile**

当前输入的文件名，可能是在类实例化时设置的，或者是由后来的源请求堆栈生成的。在构建错误信息时可能会用到本属性。

shlex.**instream**

shlex 实例正从中读取字符的输入流。

shlex.**source**

本属性默认值为 None。如果给定一个字符串，则会识别为包含请求，类似于各种 shell 中的 source 关键字。也就是说，紧随其后的词法单元将作为文件名打开，作为输入流，直至遇到 EOF 后调用流的*close()* 方法，然后原输入流仍变回输入源。Source 请求可以在词义堆栈中嵌套任意深度。

shlex.**debug**

如果本属性为大于 1 的数字，则*shlex* 实例会把动作进度详细地输出出来。若需用到本属性，可阅读源代码来了解细节。

shlex.**lineno**

源的行数（到目前为止读到的换行符数量加 1）。

shlex.**token**

词法单元的缓冲区。在捕获异常时可能会用到。

shlex.**eof**

用于确定文件结束的词法单元。在非 POSIX 模式下，将设为空字符串 ''，在 POSIX 模式下被设为 None。

shlex.**punctuation_chars**

只读属性。表示应视作标点符号的字符。标点符号将作为单个词法单元返回。然而，请注意不会进行语义有效性检查：比如“»>”可能会作为一个词法单元返回，虽然 shell 可能无法识别。

Added in version 3.6.

25.3.2 解析规则

在非 POSIX 模式下时，*shlex* 会试图遵守以下规则：

- 不识别单词中的引号（Do"Not"Separate 解析为一个单词 Do"Not"Separate）；
- 不识别转义字符；
- 引号包裹的字符保留字面意思；
- 成对的引号会将单词分离（"Do"Separate 解析为 "Do" 和 Separate）；
- 如果*whitespace_split* 为 False，则未声明为单词字符、空白或引号的字符将作为单字符的词法单元返回。若为 True，则*shlex* 只根据空白符拆分单词。
- EOF 用空字符串（''）表示；
- 空字符串无法解析，即便是加了引号。

在 POSIX 模式时，*shlex* 将尝试遵守以下解析规则：

- 引号会被剔除，且不会拆分单词（"Do"Not"Separate" 将解析为单个单词 DoNotSeparate）；
- 未加引号包裹的转义字符（如 '\' ）保留后一个字符的字面意思；
- 引号中的字符不属于*escapedquotes*（例如，"'"），则保留引号中所有字符的字面值；
- 若引号包裹的字符属于*escapedquotes*（例如 '"'），则保留引号中所有字符的字面意思，属于*escape* 中的字符除外。仅当后跟后半个引号或转义字符本身时，转义字符才保留其特殊含义。否则，转义字符将视作普通字符；
- EOF 用*None* 表示；
- 允许出现引号包裹的空字符串（''）。

25.3.3 改进的 shell 兼容性

Added in version 3.6.

shlex 类提供了与常见 Unix shell（如 bash、dash 和 sh）的解析兼容性。为了充分利用这种兼容性，请在构造函数中设定 punctuation_chars 参数。该参数默认为 False，维持 3.6 以下版本的行为。如果设为 True，则会改变对 ();<>|& 字符的解析方式：这些字符都将视为单个的词法单元返回。虽然不算是完整的 shell 解析程序（考虑到 shell 的多样性，超出了标准库的范围），但确实能比其他方式更容易进行命令行的处理。以下代码段演示了两者的差异：

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';',
'(', 'def', 'ghi', ')']
```

当然，返回的词法单元对 shell 无效，需要对返回的词法单元自行进行错误检查。

punctuation_chars 参数可以不传入 True ，而是传入包含特定字符的字符串，用于确定由哪些字符构成标点符号。例如：

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

备注

如果指定了 punctuation_chars，则*wordchars* 属性的参数会是 ~-./*?=。因为这些字符可以出现在文件名（包括通配符）和命令行参数中（如 --color=auto）。因此：

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

不过为了尽可能接近于 shell ，建议在使用*punctuation_chars* 时始终使用 posix 和*whitespace_split* ，这将完全否定*wordchars* 。

为了达到最佳效果，punctuation_chars 应与 posix=True 一起设置。（注意 posix=False 是*shlex* 的默认设置）。

CHAPTER 26

Tk 图形用户界面 (GUI)

Tk/Tcl 早已成为 Python 的一部分。它提供了一套健壮且独立于平台的窗口工具集，Python 程序员可通过`tkinter`包及其扩展`tkinter.ttk`模块来使用它。

`tkinter`包是使用面向对象方式对 Tcl/Tk 进行的一层薄包装。使用`tkinter`，你不需要写 Tcl 代码，但你将需要参阅 Tk 文档，有时还需要参阅 Tcl 文档。`tkinter`是一组包装器，它将 Tk 的可视化部件实现为相应的 Python 类。

`tkinter`的主要特点是速度很快，并且通常直接附带在 Python 中。虽然它的官方文档做得不好，但还是有许多可用的资源，包括：在线参考、教程、入门书等等。`tkinter`还有众所周知的较过时的外观界面，这在 Tk 8.5 中已得到很大改进。无论如何，你还可以考虑许多其他的 GUI 库。Python wiki 例出了一些替代性的 GUI 框架和工具。

26.1 `tkinter` --- Tcl/Tk 的 Python 接口

源代码: Lib/tkinter/__init__.py

`tkinter`包 ("Tk 接口") 是针对 Tcl/Tk GUI 工具包的标准 Python 接口。Tk 和`tkinter`在大多数 Unix 平台，包括 macOS，以及 Windows 系统上均可使用。

若在命令行执行`python -m tkinter`，应会弹出一个简单的 Tk 界面窗口，表明`tkinter`包已安装完成，还会显示当前安装的 Tcl/Tk 版本，以便阅读对应版本的 Tcl/Tk 文档。

Tkinter 支持众多的 Tcl/Tk 版本，带或不带多线程版本均可。官方的 Python 二进制版本捆绑了 Tcl/Tk 8.6 多线程版本。关于可支持版本的更多信息，请参阅`_tkinter`模块的源代码。

Tkinter 并不只是做了简单的封装，而是增加了相当多的代码逻辑，让使用体验更具 Python 风格（pythonic）。本文将集中介绍这些增加和变化部分，关于未改动部分的细节，请参考 Tcl/Tk 官方文档。

备注

Tcl/Tk 8.5 (2007) 引入了支持主题的现代风格用户界面组件集以及使用这些组件的新版 API。旧版和新版 API 都可以使用。你在网上所能找到的大多数文档仍然是使用旧版 API 因此也许已经相当过时。

参见

- **TkDocs**
  关于使用 Tkinter 创建用户界面的详细教程。讲解了关键概念，并介绍了使用现代 API 的推荐方式。
- **Tkinter 8.5 参考手册：一种 Python GUI**
  详细讲解可用的类、方法和选项的 Tkinter 8.5 参考文档。

Tcl/Tk 资源:

- **Tk 命令**
  有关 Tkinter 所使用的每个底层 Tcl/Tk 命令的完整参考文档。
- **Tcl/Tk 主页**
  额外的文档，以及 Tcl/Tk 核心开发相关链接。

书籍:

- **Modern Tkinter for Busy Python Developers**
  Mark Roseman 著。(ISBN 978-1999149567)
- **Python GUI programming with Tkinter**
  Alan D. Moore 著。(ISBN 978-1788835886)
- **Programming Python**
  Mark Lutz 著；对 Tkinter 进行了精彩的讲解。(ISBN 978-0596158101)
- **Tcl and the Tk Toolkit (2nd edition)**
  John Ousterhout , Tcl/Tk 的创造者, 与 Ken Jones 合著; 未涉及 Tkinter。(ISBN 978-0321336330)

26.1.1 架构

Tcl/Tk 不是只有单个库，而是由几个不同的模块组成的，每个模块都有各自的功能和各自的官方文档。Python 的二进制发行版还会再附加一个模块。

Tcl
Tcl 是一种动态解释型编程语言，正如 Python 一样。尽管它可作为一种通用的编程语言单独使用，但最常见的用法还是作为脚本引擎或 Tk 工具包的接口嵌入到 C 程序中。Tcl 库有一个 C 接口，用于创建和管理一个或多个 Tcl 解释器实例，并在这些实例中运行 Tcl 命令和脚本，添加用 Tcl 或 C 语言实现的自定义命令。每个解释器都拥有一个事件队列，某些部件可向解释器发送事件交由其处理。与 Python 不同，Tcl 的执行模型是围绕协同多任务而设计的，Tkinter 协调了两者的差别（详见*Threading model* ）。

Tk
Tk 是一个用 C 语言实现的 Tcl 包，它添加了用于创建和操纵 GUI 部件的自定义命令。每个`Tk`对象都嵌入了自己的 Tcl 解释器实例并将 Tk 加载到其中。Tk 的部件是高度可定制的，但其代价则是过时的外观。Tk 使用 Tcl 的事件队列来生成并处理 GUI 事件。

Ttk
带有主题的 Tk（Ttk）是较新加入的 Tk 部件，相比很多经典的 Tk 部件，在各平台提供的界面更加美观。自 Tk 8.5 版本开始，Ttk 作为 Tk 的成员进行发布。Python 则捆绑在一个单独的模块中，`tkinter.ttk`。

在内部，Tk 和 Ttk 使用下层操作系统的工具库，例如在 Unix/X11 上是 Xlib，在 macOS 上是 Cocoa，在 Windows 上是 GDI。

当你的 Python 应用程序使用 Tkinter 中的某个类，例如创建一个部件时，`tkinter`模块将首先生成一个 Tcl/Tk 命令字符串。它会把这个 Tcl 命令字符串传给内部的`_tkinter`二进制模块，后者将随后调用 Tcl 解释器来对其求值。Tcl 解释器随后将对 Tk 和/和 Ttk 包唤起，它们又将继续对 Xlib, Cocoa 或 GDI 发起调用。

26.1.2 Tkinter 模块

对 Tkinter 的支持分布在多个模块中。大多数应用程序将需要主模块*tkinter*，以及*tkinter.ttk* 模块，后者提供了带主题的现代部件集及相应的 API:

```
from tkinter import *
from tkinter import ttk
```

class tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

构造一个最高层级的 Tk 部件，这通常是一个应用程序的主窗口，并为这个部件初始化 Tcl 解释器。每个实例都有其各自所关联的 Tcl 解释器。

Tk 类通常全部使用默认值来初始化。不过，目前还可识别下列关键字参数:

screenName
当（作为字符串）给出时，设置 DISPLAY 环境变量。（仅限 X11）

baseName
预置文件的名称。在默认情况下，*baseName* 是来自于程序名称 (sys.argv[0])。

className
控件类的名称。会被用作预置文件同时也作为 Tcl 唤起的名称 (*interp* 中的 *argv0*)。

useTk
如果为 True，则初始化 Tk 子系统。*tkinter.Tcl()* 函数会将其设为 False。

sync
如果为 True，则同步执行所有 X 服务器命令，以便立即报告错误。可被用于调试。（仅限 X11）

use
指定嵌入应用程序的窗口 *id*，而不是将其创建为独立的顶层窗口。*id* 必须以与顶层控件的 -use 选项值相同的方式来指定（也就是说，它具有与 winfo_id() 的返回值相同的形式）。

请注意在某些平台上只有当 *id* 是指向一个启用了 -container 选项的 Tk 框架或顶层窗口时此参数才能正确生效。

Tk 读取并解释预置文件，其名称为 *.className*.tcl 和 *.baseName*.tcl，进入 Tcl 解释器并基于 *.className*.py 和 *.baseName*.py 的内容来调用*exec()*。预置文件的路径为 HOME 环境变量，或者如果它未被定义，则为*os.curdir*。

tk
通过实例化*Tk* 创建的 Tk 应用程序对象。这提供了对 Tcl 解释器的访问。每个被附加到相同*Tk* 实例的控件都具有相同的*tk* 属性值。

master
包含此控件的控件对象。对于*Tk*，*master* 将为*None* 因为它是主窗口。术语 *master* 和 *parent* 是类似的且有时作为参数名称被交替使用；但是，调用 winfo_parent() 将返回控件名称字符串而*master* 将返回控件对象。*parent*/*child* 反映了树型关系而 *master*/*slave* 反映了容器结构。

children
以*dict* 表示的此控件的直接下级其中的键为子控件名称而值为子实例对象。

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

Tcl() 函数是一个工厂函数，它创建的对象类似于*Tk* 类创建的，只是不会初始化 Tk 子系统。这在调动 Tcl 解释器时最为有用，这时不想创建多余的顶层窗口，或者无法创建（比如不带 X 服务的 Unix/Linux 系统）。由*Tcl()* 创建的对象可调用 loadtk() 方法创建一个顶层窗口（且会初始化 Tk 子系统）。

提供 Tk 支持的模块包括:

tkinter
主 Tkinter 模块。

tkinter.colorchooser
让用户选择颜色的对话框。

tkinter.commondialog
本文其他模块定义的对话框的基类。

tkinter.filedialog
允许用户指定文件的通用对话框，用于打开或保存文件。

tkinter.font
帮助操作字体的工具。

tkinter.messagebox
访问标准的 Tk 对话框。

tkinter.scrolledtext
内置纵向滚动条的文本组件。

tkinter.simpledialog
基础对话框和一些便捷功能。

tkinter.ttk
在 Tk 8.5 中引入的带主题的控件集，提供了对应于*tkinter* 模块中许多经典控件的现代替代。

附加模块:

_tkinter
一个包含低层级 Tcl/Tk 接口的二进制模块。它会被主*tkinter* 模块自动导入，且永远不应被应用程序员所直接使用。它通常是一个共享库（或 DLL），但在某些情况下可能被动态链接到 Python 解释器。

idlelib
Python 的集成开发与学习环境（IDLE）。基于*tkinter*。

tkinter.constants
当向 Tkinter 调用传入各种形参时可被用来代替字符串的符号常量。由主*tkinter* 模块自动导入。

tkinter.dnd
针对*tkinter* 的（实验性的）拖放支持。当以 Tk DND 代替时它将会被弃用。

turtle
Tk 窗口中的海龟绘图库。

26.1.3 Tkinter 拾遗

这一章节的设计目的不是要编写有关 Tk 或 Tkinter 的冗长教程。要获取教程，请参阅之前列出的外部资源之一。相反地，这一章节提供了对于 Tkinter 应用程序大致样貌的快速指导，列出了基本的 Tk 概念，并解释了 Tkinter 包装器的构造是什么样的。

这一章节的剩余部分将帮助你识别在你的 Tkinter 应用程序中需要的类、方法和选项，以及在哪里可以找到有关它们的更详细文档，包括官方 Tcl/Tk 参考手册等。

Hello World 程序

让我们先来看一个 Tkinter 的"Hello World" 应用程序。这并不是我们所能写出的最简短版本，但也足够说明你所需要了解的一些关键概念。

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
root.mainloop()
```

在导入语句之后，下一行语句创建了一个 `Tk` 类的实例，它会初始化 Tk 并创建与其关联的 Tcl 解释器。它还会创建一个顶层窗口，名为 root 窗口，它将被作为应用程序的主窗口。

下一行创建了一个框架控件，在本示例中它会包含我们即将创建的一个标签和一个按钮。框架被嵌在 root 窗口内部。

下一行创建了一个包含静态文本字符串的标签控件。`grid()` 方法被用来指明标签在包含它的框架控件中的相对布局（定位），作用类似于 HTML 中的表格。

接下来创建了一个按钮控件，并被放置到标签的右侧。当被按下时，它将调用 root 窗口的 `destroy()` 方法。

最后，`mainloop()` 方法将所有控件显示出来，并响应用户输入直到程序终结。

重要的 Tk 概念

即便是这样简单的程序也阐明了以下关键 Tk 概念:

控件
Tkinter 用户界面是由一个个 *控件* 组成的。每个控件都由相应的 Python 对象表示，由 `ttk.Frame`, `ttk.Label` 以及 `ttk.Button` 这样的类来实例化。

控件层级结构
控件按 *层级结构* 来组织。标签和按钮包含在框架中，框架又包含在根窗口中。当创建每个 子控件时，它的 父控件会作为控件构造器的第一个参数被传入。

配置选项
控件具有 *配置选项*，配置选项会改变控件的外观和行为，例如要在标签或按钮中显示的文本。不同的控件类会具有不同的选项集。

几何管理
小部件在创建时不会自动添加到用户界面。一个像 `grid` 的 *几何管理器* 控制这些小部件在用户界面的位置。

事件循环
只有主动运行一个 *事件循环*，Tkinter 才会对用户的输入做出反应，改变你的程序，以及刷新显示。如果你的程序没有运行事件循环，你的用户界面不会更新。

了解 Tkinter 如何封装 Tcl/Tk

当你的应用程序使用 Tkinter 的类和方法时，Tkinter 内部汇编代表 Tcl/Tk 命令的字符串，并在连接到你的应用程序的 `Tk` 实例的 Tcl 解释器中执行这些命令。

无论是试图浏览参考文档，或是试图找到正确的方法或选项，调整一些现有的代码，亦或是调试 Tkinter 应用程序，有时候理解底层 Tcl/Tk 命令是什么样子的会很有用。

为了说明这一点，下面是 Tcl/Tk 等价于上面 Tkinter 脚本的主要部分。

```
ttk::frame .frm -padding 10
grid .frm
grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row 0
grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -column 1 -row 0
```

Tcl 的语法类似于许多 shell 语言，其中第一个单词是要执行的命令，后面是该命令的参数，用空格分隔。不谈太多细节，请注意以下几点:

- 用于创建窗口小部件（如 `ttk::frame`）的命令对应于 Tkinter 中的 widget 类。
- Tcl 窗口控件选项（如 `-tex`）对应于 Tkinter 中的关键字参数。
- 在 Tcl 中，小部件是通过 *路径名* 引用的（例如 `.frm.btn`），而 Tkinter 不使用名称，而是使用对象引用。
- 控件在控件层次结构中的位置在其（层次结构）路径名中编码，该路径名使用一个 `.`（点）作为路径分隔符。根窗口的路径名是 `.`（点）。在 Tkinter 中，层次结构不是通过路径名定义的，而是通过在创建每个子控件时指定父控件来定义的。
- 在 Tcl 中以独立的 *命令* 实现的操作（比如 `grid` 和 `destroy` ）在 Tkinter 控件对象上以 *方法* 表示。稍后您将看到，在其他时候，Tcl 在控件对象调用的方法，在 Tkinter 也有对应的使用。

我该如何... ? 这个选项会做... ?

如果您不确定如何在 Tkinter 中做一些事情，并且您不能立即在您正在使用的教程或参考文档中找到它，这里有一些策略可以帮助您。

首先，请记住，在不同版本的 Tkinter 和 Tcl/Tk 中，各个控件如何工作的细节可能会有所不同。如果您正在搜索文档，请确保它与安装在系统上的 Python 和 Tcl/Tk 版本相对应。

在搜索如何使用 API 时，知道正在使用的类、选项或方法的确切名称会有所帮助。内省，无论是在交互式 Python shell 中，还是在*print()* 中，都可以帮助你确定你需要什么。

要找出控件上可用的配置选项，请调用其 `configure()` 方法，该方法返回一个字典，其中包含每个对象的各种信息，包括其默认值和当前值。使用 `keys()` 获取每个选项的名称。

```
btn = ttk.Button(frm, ...)
print(btn.configure().keys())
```

由于大多数控件都有许多共同的配置选项，因此找出特定于特定控件类的配置选项可能会很有用。将选项列表与更简单的控件（如框架）的列表进行比较是一种方法。

```
print(set(btn.configure().keys()) - set(frm.configure().keys()))
```

类似地，你可以使用标准函数*dir()* 来查找控件对象的可用方法。如果您尝试一下，您会发现有超过 200 种常见的控件方法，因此再次确认那些特定于控件类的方法是有帮助的。

```
print(dir(btn))
print(set(dir(btn)) - set(dir(frm)))
```

浏览 Tcl/Tk 参考手册

如上所述，官方的 Tk commands 参考手册（手册页）通常有对控件特定操作的最准确描述。即使您知道需要的选项或方法的名称，您可能仍然有一些地方可以查找。

虽然 Tkinter 中的所有操作都是通过对控件对象的方法调用来实现的，但您已经看到许多 Tcl/Tk 操作都是以命令的形式出现的，这些命令以小部件的路径名作为它的第一个参数，然后是可选参数，例如：

```
destroy .
grid .frm.btn -column 0 -row 0
```

但是，其他方法看起来更像在控件对象上调用的方法（实际上，当您在 Tcl/Tk 中创建小部件时，它会使用控件路径名创建 Tcl 命令，该命令的第一个参数是要调用的方法名）。

```
.frm.btn invoke
.frm.lbl configure -text "Goodbye"
```

在 Tcl/Tk 官方参考文档中，你会发现手册页上大多数操作看起来都像是特定控件的的方法调用（例如，你会在 ttk::button 手册页上找到 `invoke()` 方法），而以控件作为参数的函数通常有自己的手册页（例如，grid ）。

您将在 options 或 ttk::widget 手册页中找到许多常见的选项和方法，而其他的选项和方法可以在特定控件类的手册页中找到。

你还会发现许多 Tkinter 方法有复合名称，例如 `winfo_x()`，`winfo_height()`，`winfo_viewable()`。你可以在 winfo 页面找到这些文档。

备注

有些令人困惑的是，所有 Tkinter 小部件上还有一些方法实际上并不在控件上操作，而是在全局范围内操作，独立于任何控件。例如访问剪贴板或系统响铃的方法。(它们恰好被实现为所有 Tkinter 小部件都继承自的基类 `Widget` 中的方法)。

26.1.4 线程模型

Python 和 Tcl/Tk 的线程模型大不相同，而*tkinter* 则会试图进行调和。若要用到线程，可能需要注意这一点。

一个 Python 解释器可能会关联很多线程。在 Tcl 中，可以创建多个线程，但每个线程都关联了单独的 Tcl 解释器实例。线程也可以创建一个以上的解释器实例，尽管每个解释器实例只能由创建它的那个线程使用。

tkinter 创建的每个 `Tk` 对象均包含一个 Tcl 解释器。它还会记录解释器是由哪个线程创建的。对*tkinter* 的调用可从任何 Python 线程发起。如果唤起的线程不是创建 `Tk` 对象的线程，则会自动向解释器的事件队列中发送一条事件，在执行该事件时，会向发起调用的 Python 线程返回结果。

Tcl/Tk 应用程序通常是事件驱动的，这意味着在完成初始化以后，解释器会运行一个事件循环（即 `Tk.mainloop()`）并对事件做出响应。因为它是单线程的，所以事件处理程序必须快速响应，否则会阻塞其他事件的处理。为了避免阻塞，不应在事件处理程序中执行任何耗时很久的计算，而应利用计时器将任务分块，或者在其他线程中运行。而其他很多工具包的 GUI 是在一个完全独立的线程中运行的，独立于包括事件处理程序在内的所有代码。

如果 Tcl 解释器没有运行事件循环并处理解释器事件，则除运行 Tcl 解释器的线程外，任何其他线程发起的*tkinter* 调用都会失败。

存在一些特殊情况：

- Tcl/Tk 库可编译为不支持多线程的版本。这时*tkinter* 会从初始 Python 线程调用底层库，即便那不是创建 Tcl 解释器的线程。会有一个全局锁来确保每次只会发生一次调用。
- 虽然*tkinter* 允许创建一个以上的 `Tk` 实例（都带有自己的解释器），但所有属于同一线程的解释器均会共享同一个事件队列，这样很快就会一团糟。在实际编程时，一次创建的 `Tk` 实例不要超过一个。否则最好在不同的线程中创建，并确保运行的是支持多线程的 Tcl/Tk 版本。
- 为了防止 Tcl 解释器重新进入事件循环，阻塞事件处理程序并不是唯一的做法。甚至可以运行多个嵌套的事件循环，或者完全放弃事件循环。如果在处理事件或线程时碰到棘手的问题，请小心这些可能的事情。
- 有几个*tkinter* 函数，目前只在创建 Tcl 解释器的线程中调用才行。

26.1.5 快速参考

可选配置项

配置参数可以控制组件颜色和边框宽度等。可通过三种方式进行设置：

在对象创建时，使用关键字参数

```
fred = Button(self, fg="red", bg="blue")
```

在对象创建后，将参数名用作字典索引

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

利用 config() 方法修改对象的多个属性

```
fred.config(fg="red", bg="blue")
```

关于这些参数及其表现的完整解释，请参阅 Tk 手册中有关组件的 man 帮助页。

请注意，man 手册页列出了每个部件的“标准选项”和“组件特有选项”。前者是很多组件通用的选项列表，后者是该组件特有的选项。标准选项在 *options(3)* man 手册中有文档。

本文没有区分标准选项和部件特有选项。有些选项不适用于某类组件。组件是否对某选项做出响应，取决于组件的类别；按钮组件有一个 `command` 选项，而标签组件就没有。

组件支持的选项在其手册中有列出，也可在运行时调用 config() 方法（不带参数）查看，或者通过调用组件的 keys() 方法进行查询。这些调用的返回值为字典，字典的键是字符串格式的选项名（比如 'relief'），字典的值为五元组。

有些选项，比如 bg 是全名通用选项的同义词（bg 是“background”的简写）。向 config() 方法传入选项的简称将返回一个二元组，而不是五元组。传回的二元组将包含同义词的全名和“真正的”选项（比如 ('bg', 'background')）。

索引	含意	示例
0	选项名称	'relief'
1	数据库查找的选项名称	'relief'
2	数据库查找的选项类	'Relief'
3	默认值	'raised'
4	当前值	'groove'

示例:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

当然，输出的字典将包含所有可用选项及其值。这里只是举个例子。

包装器

包装器是 Tk 的形状管理机制之一。形状（geometry）管理器用于指定多个部件在容器（共同的 主组件）内的相对位置。与更为麻烦的 定位器相比（不太常用，这里不做介绍），包装器可接受定性的相对关系——上面、左边、填充等，并确定精确的位置坐标。

主部件的大小都由其内部的“从属部件”的大小决定。包装器用于控制从属部件在主部件中出现的位置。可以把部件包入框架，再把框架包入其他框架中，搭建出所需的布局。此外，只要完成了包装，组件的布局就会进行动态调整，以适应布局参数的变化。

请注意，只有用形状管理器指定几何形状后，部件才会显示出来。忘记设置形状参数是新手常犯的错误，惊讶于创建完部件却啥都没出现。部件只有在应用了类似于打包器的 pack() 方法之后才会显示在屏幕上。

调用 pack() 方法时可以给出由关键字/参数值组成的键值对，以便控制组件在其容器中出现的位置，以及主程序窗口大小变动时的行为。下面是一些例子:

```
fred.pack()                     # 默认为 side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

包装器的参数

关于包装器及其可接受的参数，更多信息请参阅 man 手册和 John Ousterhout 书中的第 183 页。

anchor
: anchor 类型。表示包装器要放置的每个从属组件的位置。

expand
: 布尔型，0 或 1 。

fill
: 合法值为: 'x' 、'y' 、'both' 、'none'。

ipadx 和 ipady
: 距离值，指定从属部件的内边距。

padx 和 pady
: 距离值，指定从属部件的外边距。

side
合法值为：'left'、'right'、'top'、'bottom'。

部件与变量的关联

通过一些特定参数，某些组件（如文本输入组件）的当前设置可直接与应用程序的变量关联。这些参数包括 variable、textvariable、onvalue、offvalue、value。这种关联是双向的：只要这些变量因任何原因发生变化，其关联的部件就会更新以反映新的参数值。

不幸的是，在目前*tkinter*的实现代码中，不可能通过 variable 或 textvariable 参数将任意 Python 变量移交给组件。变量只有是*tkinter*中定义的 Variable 类的子类，才能生效。

已经定义了很多有用的 Variable 子类：StringVar、IntVar、DoubleVar 和 BooleanVar。调用 get() 方法可以读取这些变量的当前值；调用 set() 方法则可改变变量值。只要遵循这种用法，组件就会保持跟踪变量的值，而不需要更多的干预。

例如：

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # 创建应用程序变量。
        self.contents = tk.StringVar()
        # 将其设为特定的值。
        self.contents.set("this is a variable")
        # 告诉输入控件监视此变量。
        self.entrythingy["textvariable"] = self.contents

        # 定义一个回调在用户按下回车时调用。
        # 它将打印变量的当前值。
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

窗口管理器

Tk 有个实用命令 wm，用于与窗口管理器进行交互。wm 命令的参数可用于控制标题、位置、图标之类的东西。在*tkinter*中，这些命令已被实现为 Wm 类的方法。顶层部件是 Wm 类的子类，所以可以直接调用 Wm 的这些方法。

要获得指定部件所在的顶层窗口，通常只要引用该部件的主窗口即可。当然，如果该部件是包装在框架内的，那么主窗口不代表就是顶层窗口。为了获得任意组件所在的顶层窗口，可以调用 _root() 方法。该方法以下划线开头，表明其为 Python 实现的代码，而非 Tk 提供的某个接口。

以下是一些典型用法：

```
import tkinter as tk

class App(tk.Frame):
```

(续下页)

(接上页)

```
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# 创建应用程序
myapp = App()

#
# 以下是对窗口管理器类的方法调用
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# 启动程序
myapp.mainloop()
```

Tk 参数的数据类型

anchor
合法值是罗盘的方位点："n"、"ne"、"e"、"se"、"s"、"sw"、"w"、"nw" 和 "center"。

bitmap
内置已命名的位图有八个：'error'、'gray25'、'gray50'、'hourglass'、'info'、'questhead'、'question'、'warning'。若要指定位图的文件名，请给出完整路径，前面加一个 @，比如 "@/usr/contrib/bitmap/gumby.bit"。

boolean
可以传入整数 0 或 1，或是字符串 "yes" 或 "no"。

callback -- 回调
指任何无需调用参数的 Python 函数。例如：

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color
可在 rgb.txt 文件中以颜色名的形式给出，或是 RGB 字符串的形式，4 位："#RGB"，8 位："#RRGGBB"，12 位："#RRRGGGBBB"，16 位："#RRRRGGGGBBBB"，其中 R、G、B 为合法的十六进制数值。详见 Ousterhout 书中的第 160 页。

cursor
可采用 cursorfont.h 中的标准光标名称，去掉 XC_ 前缀。比如要获取一个手形光标（XC_hand2），可以用字符串 "hand2"。也可以指定自己的位图和掩码文件作为光标。参见 Ousterhout 书中的第 179 页。

distance
屏幕距离可以用像素或绝对距离来指定。像素是数字，绝对距离是字符串，后面的字符表示单位：c 是厘米，i 是英寸，m 是毫米，p 则表示打印机的点数。例如，3.5 英寸可表示为 "3.5i"。

font
Tk 采用一串名称的格式表示字体，例如 {courier 10 bold}。正数的字体大小以点为单位，负数的大小以像素为单位。

geometry
这是一个 widthxheight 形式的字符串，其中宽度和高度对于大多数部件来说是以像素为单位的（对于显示文本的部件来说是以字符为单位的）。例如：fred["geometry"] = "200x100"。

justify
合法的值为字符串："left"、"center"、"right" 和 "fill"。

region
这是包含四个元素的字符串，以空格分隔，每个元素是表示一个合法的距离值（见上文）。例如：

"2 3 4 5"、"3i 2i 4.5i 2i" 和 "3c 2c 4c 10.43c" 都是合法的区域值。

relief
决定了组件的边框样式。合法值包括："raised"、"sunken"、"flat"、"groove" 和 "ridge"。

scrollcommand
这几乎就是带滚动条部件的 set() 方法，但也可是任一只有一个参数的部件方法。

wrap
只能是以下值之一："none"、"char"、"word"。

绑定和事件

部件命令中的 bind 方法可觉察某些事件，并在事件发生时触发一个回调函数。bind 方法的形式是：

```
def bind(self, sequence, func, add=''):
```

其中：

sequence
是一个表示事件的目标种类的字符串。（详情请看 *bind(3tk) 的手册页和 John Outsterhout 的书，:title-reference:`Tcl and the Tk Toolkit (2nd edition)`*，第 201 页。）

func
是带有一个参数的 Python 函数，发生事件时将会调用。传入的参数为一个 Event 实例。（以这种方式部署的函数通常称为 *回调函数*。）

add
可选项，'' 或 '+'。传入空字符串表示本次绑定将替换与此事件关联的其他所有绑定。传递 '+' 则意味着加入此事件类型已绑定函数的列表中。

例如：

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

请注意，在 turn_red() 回调函数中如何访问事件的 widget 字段。该字段包含了捕获 X 事件的控件。下表列出了事件可供访问的其他字段，及其在 Tk 中的表示方式，这在查看 Tk 手册时很有用处。

Tk	Tkinter 事件字段	Tk	Tkinter 事件字段
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

index 参数

很多控件都需要传入 index 参数。该参数用于指明 Text 控件中的位置，或指明 Entry 控件中的字符，或指明 Menu 控件中的菜单项。

Entry 控件的索引（index、view index 等）
Entry 控件带有索引属性，指向显示文本中的字符位置。这些*tkinter* 函数可用于访问文本控件中的这些特定位置：

Text 控件的索引
Text 控件的索引语法非常复杂，最好还是在 Tk 手册中查看。

Menu 索引（menu.invoke()、menu.entryconfig() 等）

菜单的某些属性和方法可以操纵特定的菜单项。只要属性或参数需要用到菜单索引，就可用以下方式传入：

- 一个整数，指的是菜单项的数字位置，从顶部开始计数，从 0 开始；
- 字符串 `"active"`，指的是当前光标所在的菜单；
- 字符串 `"last"`，指的是上一个菜单项；
- 带有 `@` 前缀的整数，比如 `@6`，这里的整数解释为菜单坐标系中的 y 像素坐标；
- 表示没有任何菜单条目的字符串 `"none"` 经常与 menu.activate() 一同被用来停用所有条目，以及——
- 与菜单项的文本标签进行模式匹配的文本串，从菜单顶部扫描到底部。请注意，此索引类型是在其他所有索引类型之后才会考虑的，这意味着文本标签为 `last`、`active` 或 `none` 的菜单项匹配成功后，可能会视为这些单词文字本身。

图片

通过 `tkinter.Image` 的各种子类可以创建相应格式的图片：

- `BitmapImage` 对应 XBM 格式的图片。
- `PhotoImage` 对应 PGM、PPM、GIF 和 PNG 格式的图片。后者自 Tk 8.6 开始支持。

这两种图片可通过 `file` 或 `data` 属性创建的（也可能由其他属性创建）。

在 3.13 版本发生变更: 添加了 `PhotoImage` 方法 `copy_replace()` 以将一个图像的某个区域拷贝到另一个图像，可能带有像素缩放和/或子采样。为 `PhotoImage` 方法 `copy()`, `zoom()` 和 `subsample()` 添加了 *from_coords* 形参。为 `PhotoImage` 方法 `copy()` 添加了 *zoom* 和 *subsample* 形参。

然后可在某些支持 `image` 属性的控件中（如标签、按钮、菜单）使用图片对象。这时，Tk 不会保留对图片对象的引用。当图片对象的最后一个 Python 引用被删除时，图片数据也会删除，并且 Tk 会在用到图片对象的地方显示一个空白框。

参见

Pillow 包增加了对 BMP, JPEG, TIFF 和 WebP 等多种格式的支持。

26.1.6 文件处理程序

Tk 允许为文件操作注册和注销一个回调函数，当对文件描述符进行 I/O 时，Tk 的主循环会调用该回调函数。每个文件描述符只能注册一个处理程序。示例代码如下：

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

在 Windows 系统中不可用。

由于不知道可读取多少字节，你可能不希望使用 *BufferedIOBase* 或 *TextIOBase* 的 *read()* 或 *readline()* 方法，因为这些方法必须读取预定数量的字节。对于套接字，可使用 *recv()* 或 *recvfrom()* 方法；对于其他文件，可使用原始读取方法或 `os.read(file.fileno(), maxbytecount)`。

`Widget.tk.`**`createfilehandler`**(*file*, *mask*, *func*)

注册文件处理程序的回调函数 *func*。*file* 参数可以是具备 *fileno()* 方法的对象（例如文件或套接字对象），也可以是整数文件描述符。*mask* 参数是下述三个常量的逻辑“或”组合。回调函数将用以下格式调用：

```
callback(file, mask)
```

Widget.tk.**deletefilehandler**(*file*)

注销文件处理函数。

_tkinter.**READABLE**

_tkinter.**WRITABLE**

_tkinter.**EXCEPTION**

Constants used in the *mask* arguments.

26.2 tkinter.colorchooser --- 颜色选择对话框

源代码: Lib/tkinter/colorchooser.py

---

tkinter.colorchooser 模块提供了*Chooser* 类作为原生颜色选择对话框的接口。Chooser 实现了一个模式颜色选择对话框窗口。Chooser 类继承自*Dialog* 类。

class tkinter.colorchooser.**Chooser**(*master=None*, ***options*)

tkinter.colorchooser.**askcolor**(*color=None*, ***options*)

创建一个颜色选择对话框。调用此方法将显示相应窗口，等待用户进行选择，并将选择的颜色 (或 None) 返回给调用者。

参见

模块*tkinter.commondialog*
Tkinter 标准对话框模块

26.3 tkinter.font --- Tkinter 字体包装器

源代码: Lib/tkinter/font.py

---

tkinter.font 模块提供用于创建和使用命名字体的*Font* 类。

不同的字体粗细和倾斜是：

tkinter.font.**NORMAL**

tkinter.font.**BOLD**

tkinter.font.**ITALIC**

tkinter.font.**ROMAN**

class tkinter.font.**Font**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)

Font 类表示命名字体。*Font* 实例具有唯一的名称，可以通过其族、大小和样式配置进行指定。命名字体是 Tk 将字体创建和标识为单个对象的方法，而不是通过每次出现时的属性来指定字体。

参数：

font - 字体指示符元组 (family, size, options)
name - 唯一的字体名
exists - 指向现有命名字体（如果有）

其他关键字选项（如果指定了 *font*，则忽略）：

family - 字体系列，例如 Courier，Times

size - 字体大小
如果 *size* 为正数，则解释为以磅为单位的大小。
如果 *size* 是负数，则将其绝对值
解释为以像素为单位的大小。
weight - 字体强调 (NORMAL, BOLD)（普通，加粗）
slant - ROMAN, ITALIC（正体，斜体）
underline - 字体下划线（0 - 无下划线，1 - 有下划线）
overstrike - 字体删除线（0 - 无删除线，1 - 有删除线）

actual(*option=None*, *displayof=None*)
返回字体的属性。

cget(*option*)
检索字体的某一个属性值。

config(***options*)
修改字体的某一个属性值。

copy()
返回当前字体的新实例。

measure(*text*, *displayof=None*)
返回以当前字体格式化时文本将在指定显示上占用的空间量。如果未指定显示，则假定为主应用程序窗口。

metrics(**options*, ***kw*)
返回特定字体的数据。选项包括：

***ascent* - 基线和最高点之间的距离**
（在该字体中的一个字符可以占用的空间中）

***descent* - 基线和最低点之间的距离**
（在该字体中的一个字符可以占用的空间中）

***linespace* - 所需最小垂直间距（在两个**
该字体的字符间，使得这两个字符在垂直方向上不重叠）。

fixed - 如果该字体宽度被固定则为 1，否则为 0。

tkinter.font.**families**(*root=None*, *displayof=None*)
返回不同的字体系列。

tkinter.font.**names**(*root=None*)
返回定义字体的名字。

tkinter.font.**nametofont**(*name*, *root=None*)
返回一个*Font* 类，代表一个 tk 命名的字体。

在 3.10 版本发生变更: 增加了 *root* 形参。

26.4 Tkinter 对话框

26.4.1 tkinter.simpledialog --- 标准 Tkinter 输入对话框

源码: Lib/tkinter/simpledialog.py

---

tkinter.simpledialog 模块包含了一些便捷类和函数，用于创建简单的模态对话框，从用户那里读取一个值。

tkinter.simpledialog.**askfloat**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askinteger**(*title*, *prompt*, ***kw*)

tkinter.simpledialog.**askstring**(*title*, *prompt*, ***kw*)

以上三个函数提供给用户输入期望值的类型的对话框.

class tkinter.simpledialog.**Dialog**(*parent*, *title=None*)

自定义对话框的基类.

body(*master*)

可以覆盖，用于构建对话框的界面，并返回初始焦点所在的部件。

buttonbox()

加入 OK 和 Cancel 按钮的默认行为. 重写自定义按钮布局.

26.4.2 tkinter.filedialog --- 文件选择对话框.

源码: Lib/tkinter/filedialog.py

---

tkinter.filedialog 模块提供了用于创建文件/目录选择窗口的类和工厂函数。

原生的载入/保存对话框.

以下类和函数提供了文件对话窗口，这些窗口带有原生外观，具备可定制行为的配置项。这些关键字参数适用于下列类和函数：

parent ——对话框下方的窗口

title ——窗口的标题

initialdir ——对话框的启动目录

initialfile ——打开对话框时选中的文件

filetypes ——（标签，匹配模式）元组构成的列表，允许使用“*”通配符

defaultextension ——默认的扩展名，用于加到文件名后面（保存对话框）。

multiple ——为 True 则允许多选

** 静态工厂函数 **

调用以下函数时，会创建一个模态的、原生外观的对话框，等待用户选取，然后将选中值或 None 返回给调用者。

tkinter.filedialog.**askopenfile**(*mode='r'*, ***options*)

tkinter.filedialog.**askopenfiles**(*mode='r'*, ***options*)

上述两个函数创建了 *Open* 对话框，并返回一个只读模式打开的文件对象。

tkinter.filedialog.**asksaveasfile**(*mode='w'*, ***options*)

创建 *SaveAs* 对话框并返回一个写入模式打开的文件对象。

tkinter.filedialog.**askopenfilename**(***options*)

tkinter.filedialog.**askopenfilenames**(***options*)

以上两个函数创建了 *Open* 对话框，并返回选中的文件名，对应着已存在的文件。

tkinter.filedialog.**asksaveasfilename**(***options*)

创建*SaveAs*对话框，并返回选中的文件名。

tkinter.filedialog.**askdirectory**(***options*)

提示用户选择一个目录.

其他关键字参数：

mustexist ——确定是否必须为已存在的目录。

class tkinter.filedialog.**Open**(*master=None*, ***options*)

class tkinter.filedialog.**SaveAs**(*master=None*, ***options*)

上述两个类提供了用于保存和加载文件的原生对话窗口。

** 便捷类 **

以下类用于从头开始创建文件/目录窗口。不会模仿当前系统的原生外观。

class tkinter.filedialog.**Directory**(*master=None*, ***options*)

创建对话框，提示用户选择一个目录。

备注

为了实现自定义的事件处理和行为，应继承 *FileDialog* 类。

class tkinter.filedialog.**FileDialog**(*master*, *title=None*)

创建一个简单的文件选择对话框。

cancel_command(*event=None*)

触发对话窗口的终止。

dirs_double_event(*event*)

目录双击事件的处理程序。

dirs_select_event(*event*)

目录单击事件的处理程序。

files_double_event(*event*)

文件双击事件的处理程序。

files_select_event(*event*)

文件单击事件的处理程序。

filter_command(*event=None*)

按目录筛选文件。

get_filter()

获取当前使用的文件筛选器。

get_selection()

获取当前选中项。

go(*dir_or_file=os.curdir*, *pattern='*'*, *default=''*, *key=None*)

显示对话框并启动事件循环。

ok_event(*event*)

退出对话框并返回当前选中项。

quit(*how=None*)

退出对话框并返回文件名。

set_filter(*dir*, *pat*)

设置文件筛选器。

set_selection(*file*)

将当前选中文件更新为 *file*。

class tkinter.filedialog.**LoadFileDialog**(*master*, *title=None*)

FileDialog 的一个子类，创建用于选取已有文件的对话窗口。

ok_command()

检测有否给出文件，以及选中的文件是否存在。

class tkinter.filedialog.**SaveFileDialog**(*master*, *title=None*)

FileDialog 的一个子类，创建用于选择目标文件的对话窗口。

ok_command()

检测选中文件是否为目录。如果选中了已存在文件，则需要用户进行确认。

26.4.3 tkinter.commondialog --- 对话窗口模板

源码: Lib/tkinter/commondialog.py

---

tkinter.commondialog 模块提供了 *Dialog* 类，是其他模块定义的对话框的基类。

class tkinter.commondialog.**Dialog**(*master=None*, ***options*)

show(*color=None*, ***options*)

显示对话窗口。

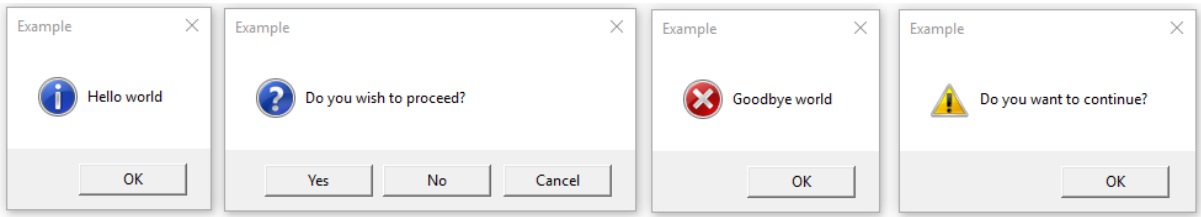
参见

模块 *tkinter.messagebox*, tut-files

26.5 tkinter.messagebox --- Tkinter 消息提示

源代码: Lib/tkinter/messagebox.py

The *tkinter.messagebox* 模块提供了一个模板基类以及多个常用配置的便捷方法。消息框为模式窗口并将基于用户的选择返回 (`True`, `False`, `None`, *OK*, *CANCEL*, *YES*, *NO*) 的一个子集。常用消息框风格和布局包括但不限于：



class tkinter.messagebox.**Message**(*master=None*, ***options*)

创建一个带有应用专属消息、图标和按钮组的消息窗口。消息窗口中的每个按钮均以唯一符号名称进行标识（参见 *type* 选项）。

支持以下选项：

command
指定当用户关闭对话框时要唤起的函数。用户关闭对话框所点击的按钮名称将作为参数传入。此选项仅在 macOS 上可用。

default
指定消息窗口默认按钮的符号名称 (*OK*, *CANCEL* 等等)。如果未指定此选项，则对话框中的第一个按钮将成为默认。

detail
为由 *message* 选项给出的主消息指定一条辅助消息。消息详情将在主消息之下展示，并且在操作系统支持的情况下，会使用次于主消息的字体。

icon
指定一个要显示的图标。如果未指定此选项，则将显示 *INFO* 图标。

message
指定要在此消息框中显示的消息。默认值为空字符串。

parent
将指定的窗口设为该消息框的逻辑上级。消息框将在其上级窗口之前显示。

title
指定要作为消息框标题的字符串。此选项在 macOS 上会被忽略，因为该平台的设计指导禁止在这种对话框中使用标题。

type
安排显示一个预定义的按钮集合。

show(***options*)
显示一个消息窗口并等待用户选择某一个按钮。然后返回所选择按钮的符号名称。关键字参数可以覆盖在构造器中指定的选项。

信息消息框

tkinter.messagebox.**showinfo**(*title=None*, *message=None*, ***options*)
创建并显示一个具有指定标题和消息的信息消息框。

警告消息框

tkinter.messagebox.**showwarning**(*title=None*, *message=None*, ***options*)
创建并显示一个具有指定标题和消息的警告消息框。

tkinter.messagebox.**showerror**(*title=None*, *message=None*, ***options*)
创建和显示一个具有指定标题和消息的错误消息框。

疑问消息框

tkinter.messagebox.**askquestion**(*title=None*, *message=None*, ***, *type=YESNO*, ***options*)
提出一个问题。在默认情况下显示 *YES* 和 *NO* 按钮。返回所选择按钮的符号名称。

tkinter.messagebox.**askokcancel**(*title=None*, *message=None*, ***options*)
询问操作是否要继续。显示 *OK* 和 *CANCEL* 按钮。如果选择确定将返回 `True` 否则返回 `False`。

tkinter.messagebox.**askretrycancel**(*title=None*, *message=None*, ***options*)
询问操作是否要重试。显示 *RETRY* 和 *CANCEL*。如果选择重试将返回 `True` 否则返回 `False`。

tkinter.messagebox.**askyesno**(*title=None*, *message=None*, ***options*)
提出一个问题。显示 *YES* 和 *NO* 按钮。如果选择是则返回 `True` 否则返回 `False`。

tkinter.messagebox.**askyesnocancel**(*title=None*, *message=None*, ***options*)
提出一个问题。显示 *YES*, *NO* 和 *CANCEL* 按钮。如果选择是则返回 `True`，取消则返回 `None`，否则返回 `False`。

按钮的符号名称:

tkinter.messagebox.**ABORT = 'abort'**

tkinter.messagebox.**RETRY = 'retry'**

tkinter.messagebox.**IGNORE = 'ignore'**

tkinter.messagebox.**OK = 'ok'**

tkinter.messagebox.**CANCEL = 'cancel'**

tkinter.messagebox.**YES = 'yes'**

tkinter.messagebox.**NO = 'no'**

预定义的按钮集合：

tkinter.messagebox.**ABORTRETRYIGNORE = 'abortretryignore'**
显示符号名称为*ABORT*, *RETRY* 和*IGNORE* 的三个按钮。

tkinter.messagebox.**OK = 'ok'**
显示符号名称为*OK* 的一个按钮。

tkinter.messagebox.**OKCANCEL = 'okcancel'**
显示符号名称为*OK* 和*CANCEL* 的两个按钮。

tkinter.messagebox.**RETRYCANCEL = 'retrycancel'**
显示符号名称为*RETRY* 和*CANCEL* 的两个按钮。

tkinter.messagebox.**YESNO = 'yesno'**
显示符号名称为*YES* 和*NO* 的两个按钮。

tkinter.messagebox.**YESNOCANCEL = 'yesnocancel'**
显示符号名称为*YES*, *NO* 和*CANCEL* 的三个按钮。

图标图像：

tkinter.messagebox.**ERROR = 'error'**

tkinter.messagebox.**INFO = 'info'**

tkinter.messagebox.**QUESTION = 'question'**

tkinter.messagebox.**WARNING = 'warning'**

26.6 tkinter.scrolledtext --- 流动文本控件

源代码： Lib/tkinter/scrolledtext.py

---

tkinter.scrolledtext 模块提供了一个同名的类，实现了带有垂直滚动条并被配置为可以“正常运作”的文本控件。使用*ScrolledText* 类会比直接配置一个文本控件附加滚动条要简单得多。

文本控件与滚动条打包在一个 `Frame` 中，`Grid` 方法和 `Pack` 方法的布局管理器从 `Frame` 对象中获得。这允许*ScrolledText* 控件可以直接用于实现大多数正常的布局管理行为。

如果需要更具体的控制，可以使用以下属性：

class tkinter.scrolledtext.**ScrolledText**(*master=None*, ***kw*)

frame
围绕文本和滚动条控件的框架。

vbar
滚动条控件。

26.7 tkinter.dnd --- 拖放操作支持

源代码: Lib/tkinter/dnd.py

备注

此模块是实验性的且在为 Tk DND 所替代后将被弃用。

tkinter.dnd 模块为单个应用内部的对象提供了在同一窗口中或多个窗口间的拖放操作支持。要将对象设为可拖放，你必须为其创建启动拖放进程的事件绑定。通常，你要将 ButtonPress 事件绑定到你所编写的回调函数 (参见绑定和事件)。该函数应当调用 *dnd_start()*，其中'source' 为要拖动的对象，而'event' 为被唤起的事件（你的回调函数的参数）。

目标对象的选择方式如下:

1. 从顶至底地在鼠标之下的区域中搜索目标控件
   - 目标控件应当具有一个指向可调用对象的 *dnd_accept* 属性
   - 如果 *dnd_accept* 不存在或者返回 `None`，则将转至父控件中搜索
   - 如果目标控件未找到，则目标对象为 `None`
2. 调用 *<old_target>.dnd_leave(source, event)*
3. 调用 *<new_target>.dnd_enter(source, event)*
4. 调用 *<target>.dnd_commit(source, event)* 来通知释放
5. 调用 *<source>.dnd_end(target, event)* 来表明拖放的结束

class tkinter.dnd.**DndHandler**(*source*, *event*)

DndHandler 类处理拖放事件，在事件控件的根对象上跟踪 Motion 和 ButtonRelease 事件。

cancel(*event=None*)

取消拖放进程。

finish(*event*, *commit=0*)

执行结束播放函数。

on_motion(*event*)

在执行拖动期间为目标对象检查鼠标之下的区域。

on_release(*event*)

当释放模式被触发时表明拖动的结束。

tkinter.dnd.**dnd_start**(*source*, *event*)

用于拖放进程的工厂函数。

参见

绑定和事件

26.8 tkinter.ttk --- Tk 带主题的控件

源代码: Lib/tkinter/ttk.py

tkinter.ttk 模块自 Tk 8.5 开始引入，它提供了对 Tk 风格的部件集的访问。它还带来了一些额外好处包括在 X11 下的反锯齿字体渲染和透明化窗口（需要有 X11 上的混合窗口管理器）。

tkinter.ttk 的基本设计思路，就是尽可能地把控件的行为代码与实现其外观的代码分离开来。

参见

Tk 控件风格
介绍 Tk 风格的文档

26.8.1 ttk 的用法

使用 ttk 之前，首先要导入模块：

```
from tkinter import ttk
```

为了覆盖基础的 Tk 控件，应该在 Tk 之后进行导入：

```
from tkinter import *
from tkinter.ttk import *
```

这段代码会让以下几个 `tkinter.ttk` 控件（`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` 和 `Scrollbar`）自动替换掉 Tk 的对应控件。

使用新控件的直接好处，是拥有更好的跨平台的外观，但新旧控件并不完全兼容。主要区别在于，Ttk 组件不再包含“fg”、“bg”等与样式相关的属性。而是用 `ttk.Style` 类来定义更美观的样式效果。

参见

将现有应用程序转换为使用 Tile 部件
此文介绍迁移为新控件时的常见差别（使用 Tcl ）。

26.8.2 ttk 控件

ttk 中有 18 种部件，其中 12 种在 tkinter 中已包含了：`Button` 、`Checkbutton` 、`Entry` 、`Frame` 、`Label`, `LabelFrame` 、`Menubutton` 、`PanedWindow` 、`Radiobutton` 、`Scale` 、`Scrollbar` 和*Spinbox*。另有 6 种是新增的：*Combobox* 、*Notebook* 、*Progressbar* 、`Separator` 、`Sizegrip` 和*Treeview*。这些控件全都是*Widget* 的子类。

ttk 控件可以改善应用程序的外观。如上所述，修改样式的代码与 tk 控件存在差异。

Tk 代码:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk 代码：

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

有关*TtkStyling* 的更多信息，请参阅*Style* 类文档。

26.8.3 控件

`ttk.Widget` 定义了 Tk 风格控件支持的标准属性和方法，不应直接对其进行实例化。

标准属性

所有 ttk 控件均接受以下选项:

属性	描述
class	指定窗口类。若要从参数库中查找窗口的其他属性，或确认窗口的默认绑定标签，或选择控件的默认布局和样式，会用到 class 属性。该属性只读，且只能在创建窗口时指定。
cursor	指定控件使用的鼠标光标。若设为空字符串（默认值），则会继承父控件的光标。
takefocus	决定了窗口是否可用键盘获得焦点。返回 0 、1 或空字符串。若返回 0，则表示在用键盘遍历时应该跳过该窗口。如果为 1，则表示只要窗口可见即应接收输入焦点。而空字符串则表示由遍历代码决定窗口是否接收焦点。
style	可用于指定自定义控件样式。

可滚动控件的属性

带滚动条的控件支持以下属性:

属性	描述
xscrollcommand	用于与水平滚动条通讯. 当窗口中的可见内容发生变化时，控件将根据 scrollcommand 生成 Tcl 命令。 通常该属性由一些滚动条的 Scrollbar.set() 方法组成。当窗口中的可见内容发生变化时，将会刷新滚动条的状态。
yscrollcommand	用于与垂直滚动条通讯，更多信息请参考上一条。

标签控件的属性

标签、按钮和类似按钮的控件支持以下属性。

属性	描述
text	指定显示在控件内的文本。
textvariable	指定一个变量名，其值将用于设置 text 属性。
underline	设置文本字符串中带下划线字符的索引（基于 0）。下划线字符用于激活快捷键。
image	指定一个用于显示的图片。这是一个由 1 个或多个元素组成的列表。第一个元素是默认的图片名称。列表的其余部分是由*Style.map()* 定义的“状态/值对”的序列，指定控件在某状态或状态组合时要采用的图片。列表中的所有图片应具备相同的尺寸。
compound	指定同时存在 text 和 image 属性时，应如何显示文本和对应的图片。合法的值包括: • text:：只显示文本 • image：只显示图片 • top、bottom、left、right：分别在文本的上、下、左、右显示图片。 • none：默认值。如果给出了图片则显示，否则显示文本。
width	如果值大于零，指定文本标签留下多少空间，单位是字符数；如果值小于零，则指定最小宽度。如果等于零或未指定，则使用文本标签本身的宽度。

兼容性属性

属性	描述
state	可以设为“normal”或“disabled”，以便控制“禁用”状态标志位。本属性只允许写入：用以改变控件的状态，但*Widget.state()* 方法不影响本属性。

控件状态

控件状态是多个相互独立的状态标志位的组合。

标志位	描述
active	鼠标光标经过控件并按下鼠标按钮，将引发动作。
disabled	控件处于禁用状态，而由程序控制。
focus	控件接受键盘焦点。
pressed	控件已被按下。
selected	勾选或单选框之类的控件，表示启用、选中状态。
background	Windows 和 Mac 系统的窗口具有“激活”或后台的概念。后台窗口的控件会设置 *foreground* 参数，而前台窗口的控件则会清除此状态。
readonly	控件不允许用户修改。
alternate	控件的备选显式格式。
invalid	控件的值是无效的

所谓的控件状态，就是一串状态名称的组合，可在某个名称前加上感叹号，表示该状态位是关闭的。

ttk.Widget

除了以下方法之外，`ttk.Widget` 还支持 `tkinter.Widget.cget()` 和 `tkinter.Widget.configure()` 方法。

class tkinter.ttk.**Widget**

identify(*x*, *y*)

返回位于 *x y* 的控件名称，如果该坐标点不属于任何控件，则返回空字符串。

x 和 *y* 是控件内的相对坐标，单位是像素。

instate(*statespec*, *callback=None*, **args*, ***kw*)

检测控件的状态。如果没有设置回调函数，那么当控件状态符合 *statespec* 时返回 `True`，否则返回 `False`。如果指定了回调函数，那么当控件状态匹配 *statespec* 时将会调用回调函数，且会带上后面的参数。

state(*statespec=None*)

修改或查询部件状态。如果指定了 *statespec*，则会用它来设置部件状态并返回一个新的 *statespec* 来指明哪些旗标做过改动。如果未指定 *statespec*，则返回当前启用的状态旗标。

statespec 通常是个列表或元组。

26.8.4 Combobox

`ttk.Combobox` 控件是文本框和下拉列表的组合体。该控件是 `Entry` 的子类。

除了从*Widget* 继承的 `Widget.cget()` 、`Widget.configure()` 、*Widget.identify()* 、*Widget.instate()* 和*Widget.state()* 方法，以及从 `Entry` 继承的 `Entry.bbox()` 、`Entry.delete()` 、`Entry.icursor()` 、`Entry.index()` 、`Entry.insert()` 、`Entry.selection()` 、`Entry.xview()` 方法，控件还自带了其他几个方法，在 `ttk.Combobox` 中都有介绍。

属性

控件可设置以下属性：

属性	描述
exportselection	布尔值，如果设为 True，则控件的选中文字将关联为窗口管理器的选中文字（可由 Misc.selection_get 返回）。
justify	指定文本在控件中的对齐方式。可为 left、center、right 之一。
height	设置下拉列表框的高度。
postcommand	在显示之前将被调用的代码（可用 Misc.register 进行注册）。可用于选择要显示的值。
state	normal 、readonly 或 disabled。在 readonly 状态下，数据不能直接编辑，用户只能从下拉列表中选取。在 normal 状态下，可直接编辑文本框。在 disabled 状态下，无法做任何交互。
textvariable	设置一个变量名，其值与控件的值关联。每当该变量对应的值发生变动时，控件值就会更新，反之亦然。参见 `tkinter.StringVar` 。
values	设置显示于下拉列表中的值。
width	设置为整数值，表示输入窗口的应有宽度，单位是字符单位（控件字体的平均字符宽度）。

虚拟事件

当用户从下拉列表中选择某个元素时，控件会生成一条 **«ComboboxSelected»** 虚拟事件。

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)

如果给出了 *newindex*，则把控件值设为 *newindex* 位置的元素值。否则，返回当前值的索引，当前值未在列表中则返回 -1。

get()

返回控件的当前值。

set(*value*)

设置控件的值为 *value* 。

26.8.5 Spinbox

`ttk.Spinbox` 控件是 `ttk.Entry` 的扩展，带有递增和递减箭头。可用于数字或字符串列表。这是 `Entry` 的子类。

除了从*Widget* 继承的 `Widget.cget()` 、`Widget.configure()` 、`Widget.identified()` 、*Widget.instate()* 和*Widget.state()* 方法，以及从 `Entry` 继承的 `Entry. bbox()` 、`Entry.delete()` 、`Entry.icursor()` 、`Entry.index()` 、`Entry.insert()` 、`Entry.xview()` 方法，控件还自带了其他一些方法，在 `ttk.Spinbox` 中都有介绍。

属性

控件可设置以下属性：

属性	描述
from	浮点值。如若给出，则为递减按钮能够到达的最小值。作为参数使用时必须写成 `from_`，因为 `from` 是 Python 关键字。
to	浮点值。如若给出，则为递增按钮能够到达的最大值。
increment	浮点值。指定递增/递减按钮每次的修改量。默认值为 1.0。
values	字符串或浮点值构成的序列。如若给出，则递增/递减会在此序列元素间循环，而不是增减数值。
wrap	布尔值。若为 `True`，则递增和递减按钮会由 `to` 值循环至 `from` 值，或由 `from` 值循环至 `to` 值。
format	字符串。指定递增/递减按钮的数字格式。必须以“%W.Pf”的格式给出，W 是填充的宽度，P 是小数精度，% 和 f 就是本身的含义。
command	Python 回调函数。只要递增或递减按钮按下之后，就会进行不带参数的调用。

虚拟事件

用户若按下 <Up>，则控件会生成 **«Increment»** 虚拟事件，若按下 <Down> 则会生成 **«Decrement»** 事件。

ttk.Spinbox

class tkinter.ttk.**Spinbox**

get()

返回控件的当前值。

set(*value*)

设置控件值为 *value*。

26.8.6 Notebook

Ttk Notebook 部件可管理由窗口组成的多项集并每次显示其中的某一个。每个子窗口都与一个选项卡相关联，用户可以选择该选项卡来改变当前所显示的窗口。

属性

控件可设置以下属性：

属性	描述
height	如若给出且大于 0，则指定面板的应有高度（不含内部 padding 或 tab）。否则会采用所有子窗口面板的最大高度。
padding	指定在控件外部添加的留白。padding 是最多包含四个值的列表，指定左顶右底的空间。如果给出的元素少于四个，底部值默认为顶部值，右侧值默认为左侧值，顶部值默认为左侧值。
width	若给出且大于 0，则设置面板的应有宽度（不含内部 padding）。否则将采用所有子窗口面板的最大宽度。

Tab 属性

Tab 特有属性如下：

属性	描述
state	可为 normal、disabled 或 disabled 之一。若为 disabled 则不能选中。若为 hidden 则不会显示。
sticky	指定子窗口在面板内的定位方式。应为包含零个或多个 n、s、e 、w 字符的字符串。每个字母表示子窗口应紧靠的方向（北、南、东或西），正如 `grid()` 位置管理器所述。
padding	指定控件和面板之间的留白空间。格式与本控件的 padding 属性相同。
text	指定显示在 tab 上的文本。
image	指定显示在 tab 上的图片。参见*`Widget`* 的 image 属性。
compound	当文本和图片同时存在时，指定图片相对于文本的显示位置。合法的属性值参见*Label Options* 。
underline	指定下划线在文本字符串中的索引（基于 0）。如果调用过了*`Notebook.enable_traversal()`*，带下划线的字符将用于激活快捷键。

Tab ID

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- 介于 0 和 tab 总数之间的整数值。
- 子窗口的名称。
- 以 “@x,y” 形式给出的位置，唯一标识了 tab 页。
- 字符串字面值”current”，它标识当前被选中的选项卡
- 字符串字面值”end”，它返回标签页的数量 (仅适用于*`Notebook.index()`*)

虚拟事件

当选中一个新 tab 页之后，控件会生成一条 **«NotebookTabChanged»** 虚拟事件。

ttk.Notebook

class tkinter.ttk.**Notebook**

add(*child*, ***kw*)

添加一个新 tab 页。

如果窗口是由 Notebook 管理但处于隐藏状态，则会恢复到之前的位置。

可用属性请参见*Tab Options* 。

forget(*tab_id*)

删除 *tab_id* 指定的 tab 页，对其关联的窗口不再作映射和管理。

hide(*tab_id*)

隐藏 *tab_id* 指定的 tab 页。

tab 页不会显示出来，但关联的窗口仍接受 Notebook 的管理，其配置属性会继续保留。隐藏的 tab 页可由*`add()`* 恢复。

identify(*x*, *y*)

返回 tab 页内位置为 *x*、*y* 的控件名称，若不存在则返回空字符串。

index(*tab_id*)

返回 *tab_id* 指定 tab 页的索引值，如果 *tab_id* 为 end 则返回 tab 页的总数。

insert(*pos*, *child*, ***kw*)

在指定位置插入一个 tab。

pos 可为字符串 “end”、整数索引值或子窗口名称。如果 *child* 已由 Notebook 管理，则将其移至指定位置。

可用属性请参见*Tab Options* 。

select(*tab_id=None*)

选中 *tab_id* 指定 tab。

关联的子窗口将被显示，而之前所选择的窗口（如果不同）将被取消映射关系。如果省略 *tab_id*，则返回当前被选中的面板的部件名称。

tab(*tab_id*, *option=None*, ***kw*)

查询或修改 *tab_id* 指定 tab 的属性。

如果未给出 *kw*，则返回由 tab 属性组成的字典。如果指定了 *option*，则返回其值。否则，设置属性值。

tabs()

返回 Notebook 管理的窗口列表。

enable_traversal()

为包含 Notebook 的顶层窗口启用键盘遍历。

这将为包含 Notebook 的顶层窗口增加如下键盘绑定关系：

- `Control-Tab`： 选中当前 tab 之后的页。
- `Shift-Control-Tab`： 选中当前 tab 之前的页。
- `Alt-K`： 这里 *K* 是任意 tab 页的快捷键（带下划线）字符，将会直接选中该 tab。

一个顶层窗口中可为多个 Notebook 启用键盘遍历，包括嵌套的 Notebook 。但仅当所有面板都将所在 Notebook 作为父控件时，键盘遍历才会生效。

26.8.7 Progressbar

`ttk.Progressbar` 控件可为长时间操作显示状态。可工作于两种模式：1）determinate 模式，显示相对完成进度；2）indeterminate 模式，显示动画让用户知道工作正在进行中。

属性

控件可设置以下属性：

属性	描述
orient	horizontal 或 vertical。指定进度条的显示方向。
length	指定进度条长轴的长度（横向为宽度，纵向则为高度）。
mode	determinate 或 indeterminate。
maximum	设定最大值。默认为 100。
value	进度条的当前值。在 determinate 模式下代表已完成的工作量。在 indeterminate 模式下，解释为 *maximum* 的模；也就是说，当本值增至 *maximum* 时，进度条完成了一个“周期”。
variable	与属性值关联的变量名。若给出，则当变量值变化时会自动设为进度条的值。
phase	只读属性。只要值大于 0 且在 determinate 模式下小于最大值，控件就会定期增大该属性值。当前主题可利用本属性提供额外的动画效果。

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)

开启自增模式：安排一个循环的定时器事件，每隔 *interval* 毫秒调用一次`Progressbar.step()`。*interval* 可省略，默认为 50 毫秒。

step(*amount=None*)

将进度条的值增加 *amount*。

amount 可省略，默认为 1.0。

stop()

停止自增模式：取消所有由*Progressbar.start()* 启动的循环定时器事件。

26.8.8 Separator

ttk.Separator 控件用于显示横向或纵向的分隔条。

除由 ttk.Widget 继承而来的方法外，没有定义其他方法。

属性

属性如下：

属性	描述
orient	horizontal 或 vertical。指定分隔条的方向。

26.8.9 Sizegrip

ttk.Sizegrip 控件允许用户通过按下并拖动控制柄来调整内部顶层窗口的大小。

除由 ttk.Widget 继承的之外，没有其他属性和方法。

与平台相关的注意事项

- 在 macOS 上，顶层窗口默认自动包括了一个内置的大小控制柄。再加一个 Sizegrip 也没什么坏处，因为内置的控制柄会盖住该控件。

Bug

- 假如内部的顶层窗口位置是相对于屏幕的右侧或底部进行设置的，那么 Sizegrip 控件将不会改变窗口的大小。
- Sizegrip 仅支持往“东南”方向的缩放。

26.8.10 Treeview

ttk.Treeview 控件可将多项内容分层级显示。每个数据项都带有一个文本标签、一个可选的图片和一个可选的数据值列表。这些数据值将在树标签后面分列显示。

数据值的显示顺序可用属性 displaycolumns 进行控制。树控件还可以显示列标题。数据列可通过数字或名称进行访问，各列的名称在属性 columns 中列出。参阅*Column Identifiers*。

每个数据项都由唯一名称进行标识。如果调用者未提供数据项的 ID，树控件会自动生成。根有且只有一个，名为 {}。根本身不会显示出来；其子项将显示在顶层。

每个数据项均带有一个 tag 列表，可用于绑定事件及控制外观。

Treeview 组件支持水平和垂直滚动，滚动时会依据*Scrollable Widget Options* 描述的属性和*Treeview.xview()* 和*Treeview.yview()* 方法。

属性

控件可设置以下属性：

属性	描述
columns	列标识的列表，定义了列的数量和名称。
displaycolumns	列标识的列表（索引可为符号或整数），指定要显示的数据列及显示顺序，或为字符串 “#all”。
height	指定可见的行数。注意：所需宽度由各列宽度之和决定。
padding	指定控件内部的留白。为不超过四个元素的长度列表。
selectmode	控制内部类如何进行选中项的管理。可为 extended、browse 或 none。若设为 extended（默认），则可选中多个项。若为 browse ，则每次只能选中一项。若为 none，则无法修改选中项。 请注意，代码和 tag 绑定可自由进行选中操作，不受本属性的限制。
show	由 0 个或下列值组成的列表，指定要显示树的哪些元素。 • tree： 在 #0 列显示树的文本标签。 • headings： 显示标题行。 默认为 “tree headings”，显示所有元素。 ** 注意 **： 第 #0 列一定是指 tree 列，即便未设置 show=”tree” 也一样。

数据项的属性

可在插入和数据项操作时设置以下属性。

属性	描述
text	用于显示的文本标签。
image	Tk 图片对象，显示在文本标签左侧。
values	关联的数据值列表。 每个数据项关联的数据数量应与 columns 属性相同。如果比 columns 属性的少，剩下的值将视为空。如果多于 columns 属性的，多余数据将被忽略。
open	`True` 或 `False`，表明是否显示数据项的子树。
tags	与该数据项关联的 tag 列表。

tag 属性

tag 可定义以下属性：

属性	描述
foreground	定义文本前景色。
background	定义单元格或数据项的背景色。
font	定义文本的字体。
image	定义数据项的图片，当 image 属性为空时使用。

列标识

列标识可用以下格式给出：

- 由 columns 属性给出的符号名。
- 整数值 n，指定第 n 列。
- #n 的字符串格式，n 是整数，指定第 n 个显示列。

注意：

- 数据项属性的显示顺序可能与存储顺序不一样。
- #0 列一定是指 tree 列，即便未指定 show=”tree” 也是一样。

数据列号是指属性值列表中的索引值，显示列号是指显示在树控件中的列号。树的文本标签将显示在 #0 列。如果未设置 displaycolumns 属性，则数据列 n 将显示在第 #n+1 列。再次强调一下，**#0 列一定是指 tree 列**。

虚拟事件

Treeview 控件会生成以下虚拟事件。

事件	描述
«TreeviewSelect»	当选中项发生变化时生成。
«TreeviewOpen»	当焦点所在项的 open= True 之前立即生成。
«TreeviewClose»	当焦点所在项的 open= True 之后立即生成。

Treeview.focus() 和*Treeview.selection()* 方法可用于确认涉及的数据项。

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)

返回某 *数据项*的边界（相对于控件窗口的坐标），形式为 (x, y, width, height) 。

若给出了 *column*，则返回该单元格的边界。若该 *数据项*不可见（即从属于已关闭项或滚动至屏幕外），则返回空字符串。

get_children(*item=None*)

返回 *item* 的下属数据项列表。

若未给出 *item* ，则返回根的下属数据。

set_children(*item*, **newchildren*)

用 *newchildren* 替换 *item* 的下属数据。

对于 *item* 中存在而 *newchildren* 中不存在的数据项，会从树中移除。*newchildren* 中的数据不能是 *item* 的上级。注意，未给出 *newchildren* 会导致 *item* 的子项被解除关联。

column(*column*, *option=None*, ***kw*)

查询或修改列 *column* 的属性。

如果未给出 *kw*，则返回属性值的字典。若指定了 *option*，则会返回该属性值。否则将设置属性值。

合法的属性/值可为：

id
返回列名。这是只读属性。

***anchor*: 某个标准 Tk 锚点值。**
指定该列的文本在单元格内的对齐方式。

***minwidth*: 宽度**
列的最小宽度，单位是像素。在缩放控件或用户拖动某一列时，Treeview 会保证列宽不小于此值。

***stretch*: `True`/`False`**
指明缩放控件时是否调整列宽。

***width*: 宽度**
列宽，单位为像素数。

若要设置 tree 列，请带上参数 column = "#0" 进行调用。

delete(**items*)

删除所有 *items* 及其下属。

根不能删除。

detach(**items*)

将所有 *items* 与树解除关联。

数据项及其下属依然存在，后续可以重新插入，目前只是不显示出来。

根不能解除关联。

exists(*item*)

如果给出的 *item* 位于树中，则返回 `True`。

focus(*item=None*)

如果给出 *item* 则设为当前焦点。否则返回当前焦点所在数据项，若无则返回”。

heading(*column*, *option=None*, ***kw*)

查询或修改某 *column* 的标题。

若未给出 *kw*，则返回列标题组成的列表。若给出了 *option* 则返回对应属性值。否则，设置属性值。

合法的属性/值可为：

***text*: 文本**
显示为列标题的文本。

***image*: 图片名称**
指定显示在列标题右侧的图片。

***anchor*: 锚点**
指定列标题文本的对齐方式。应为标准的 Tk 锚点值。

***command*: 回调**
点击列标题时执行的回调函数。

若要对 tree 列进行设置，请带上 column = ”#0” 进行调用。

identify(*component*, *x*, *y*)

返回 *x*、*y* 位置上 *component* 数据项的描述信息，如果此处没有该数据项，则返回空字符串。

identify_row(*y*)

返回 *y* 位置上的数据项 ID。

identify_column(*x*)

返回 *x* 位置上的单元格所在的数据列 ID。

tree 列的 ID 为 #0 。

identify_region(*x*, *y*)

返回以下值之一：

区域	含义
heading	树的标题栏区域。
separator	两个列标题之间的间隔区域。
tree	树区域。
cell	数据单元格。

可用性：Tk 8.6。

identify_element(*x*, *y*)

返回位于 *x* 、*y* 的数据项。

可用性：Tk 8.6。

index(*item*)

返回 *item* 在父项的子项列表中的整数索引。

insert(*parent*, *index*, *iid=None*, ***kw*)

新建一个数据项并返回其 ID。

parent 是父项的 ID，若要新建顶级项则为空字符串。*index* 是整数或 “end”，指明在父项的子项列表中的插入位置。如果 *index* 小于等于 0，则在开头插入新节点；如果 *index* 大于或等于当前子节点数，则将其插入末尾。如果给出了 *iid*，则将其用作数据项 ID；*iid* 不得存在于树中。否则会新生成一个唯一 ID。

可用的选项列表请参阅*Item Options*。

item(*item*, *option=None*, ***kw*)

查询或修改某 *item* 的属性。

如果未给出 option，则返回属性/值构成的字典。如果给出了 *option*，则返回该属性的值。否则，将属性设为 *kw* 给出的值。

move(*item*, *parent*, *index*)

将 *item* 移至指定位置，父项为 *parent* ，子项列表索引为 *index* 。

将数据项移入其子项之下是非法的。如果 *index* 小于等于 0，*item* 将被移到开头；如果大于等于子项的总数，则被移至最后。如果 *item* 已解除关联，则会被重新关联。

next(*item*)

返回 *item* 的下一个相邻项，如果 *item* 是父项的最后一个子项，则返回”。

parent(*item*)

返回 *item* 的父项 ID，如果 *item* 为顶级节点，则返回”。

prev(*item*)

返回 *item* 的前一个相邻项，若 *item* 为父项的第一个子项，则返回”。

reattach(*item*, *parent*, *index*)

`Treeview.move()` 的别名。

see(*item*)

确保 *item* 可见。

将 *item* 所有上级的 open 属性设为 `True`，必要时会滚动控件，让 *item* 处于树的可见部分。

selection()

返回由选中项构成的元组。

在 3.8 版本发生变更: `selection()` 不再接受参数了。若要改变选中的状态，请使用下面介绍的方法。

selection_set(**items*)

让 *items* 成为新的选中项。

在 3.6 版本发生变更: *items* 可作为多个单独的参数传递，而不只是作为一个元组。

selection_add(**items*)

将 *items* 加入选中项。

在 3.6 版本发生变更: *items* 可作为多个单独的参数传递，而不只是作为一个元组。

selection_remove(**items*)

从选中项中移除 *items* 。

在 3.6 版本发生变更: *items* 可作为多个单独的参数传递，而不只是作为一个元组。

selection_toggle(**items*)

切换 *items* 中各项的选中状态。

在 3.6 版本发生变更: *items* 可作为多个单独的参数传递，而不只是作为一个元组。

set(*item*, *column=None*, *value=None*)

若带一个参数，则返回 *item* 的列/值字典。若带两个参数，则返回 *column* 的当前值。若带三个参数，则将 *item* 的 *column* 设为 *value*。

tag_bind(*tagname*, *sequence=None*, *callback=None*)

为 tag 为 *tagname* 的数据项绑定事件 *sequence* 的回调函数。当事件分发给该数据项时，tag 参数为 *tagname* 的全部数据项的回调都会被调用到。

tag_configure(*tagname*, *option=None*, ***kw*)

查询或修改 *tagname* 指定项的属性。

如果给出了 *kw*，则返回 *tagname* 项的属性字典。如果给出了 *option*，则返回 *tagname* 项的 *option* 属性值。否则，设置 *tagname* 项的属性值。

tag_has(*tagname*, *item=None*)

如果给出了 *item* ，则依据 *item* 是否具备 *tagname* 而返回 1 或 0。否则，返回 tag 为 *tagname* 的所有数据项构成的列表。

可用性：Tk 8.6。

xview(**args*)

查询或修改 Treeview 的横向位置。

yview(**args*)

查询或修改 Treeview 的纵向位置。

26.8.11 Ttk 样式

`ttk` 的每种控件都赋有一个样式，指定了控件内的元素及其排列方式，以及元素属性的动态和默认设置。默认情况下，样式名与控件的类名相同，但可能会被控件的 style 属性覆盖。如果不知道控件的类名，可用 `Misc.winfo_class()` 方法获取（somewidget.winfo_class()）。

参见

Tcl'2004 会议报告
文章解释了主题引擎的工作原理。

class tkinter.ttk.**Style**

用于操控样式数据库的类。

configure(*style*, *query_opt=None*, ***kw*)

查询或设置 *style* 的默认属性值。

Each key in *kw* is an option and each value is a string identifying the value for that option.

例如，要将默认按钮改为扁平样式，并带有留白和各种背景色：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)

查询或设置 *style* 的指定属性的动态值。

kw 的每个键都是一个属性，每个值通常应为列表或元组，其中包含以元组、列表或其他形式组合而成的状态标识（statespec）。状态标识是由一个或多个状态组合，加上一个值组成。

举个例子能更清晰些：

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

请注意，要点是属性的（状态，值）序列的顺序，如果前景色属性的顺序改为 `[('active', 'blue'), ('pressed', 'red')]`，则控件处于激活或按下状态时的前景色将为蓝色。

lookup(*style*, *option*, *state=None*, *default=None*)

返回 *style* 中的 *option* 属性值。

如果给出了 *state*，则应是一个或多个状态组成的序列。如果设置了 *default* 参数，则在属性值缺失时会用作后备值。

若要检测按钮的默认字体，可以：

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout(*style*, *layoutspec=None*)

按照 *style* 定义控件布局。如果省略了 *layoutspec*，则返回该样式的布局属性。

若给出了 *layoutspec*，则应为一个列表或其他的序列类型（不包括字符串），其中的数据项应为元组类型，第一项是布局名称，第二项的格式应符合*Layouts* 的描述。

以下示例有助于理解这种格式（这里并没有实际意义）：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
```

(续下页)

(接上页)

```
mbtn.pack()
root.mainloop()
```

element_create(*elementname*, *etype*, **args*, ***kw*)

在当前主题中创建一个新元素，为给定的 *etype*，它应当是"image", "from" 或"vsapi"。后者权在 Windows 版 Tk 8.6 中可用。

如果用了 image，则 *args* 应包含默认的图片名，后面跟着状态标识/值（这里是 imagespec），*kw* 可带有以下属性：

border=padding
padding 是由不超过四个整数构成的列表，分别定义了左、顶、右、底的边界。

height=height
定义了元素的最小高度。如果小于零，则默认采用图片本身的高度。

padding=padding
定义了元素的内部留白。若未指定则默认采用 border 值。

sticky=spec
定义了图片的对齐方式。spec 包含零个或多个 n、s、w、e 字符。

width=width
定义了元素的最小宽度。如果小于零，则默认采用图片本身的宽度。

示例:

```
img1 = tkinter.PhotoImage(master=root, file='button.png')
img1 = tkinter.PhotoImage(master=root, file='button-pressed.png')
img1 = tkinter.PhotoImage(master=root, file='button-active.png')
style = ttk.Style(root)
style.element_create('Button.button', 'image',
                     img1, ('pressed', img2), ('active', img3),
                     border=(2, 4), sticky='we')
```

如果 *etype* 的值用了 from，则`element_create()` 将复制一个现有的元素。*args* 应包含主题名和可选的要复制的元素。若未给出要克隆的元素，则采用空元素。*kw* 参数将被丢弃。

示例:

```
style = ttk.Style(root)
style.element_create('plain.background', 'from', 'default')
```

如果使用"vsapi" 作为 *etype* 的值，`element_create()` 将在当前主题中创建一个新元素，其视觉外观将使用负责处理 Windows XP 和 Vista 上带主题风格的 Microsoft Visual Styles API 来绘制。*args* 应当包含 Microsoft 文档中给出的 Visual Styles 类和部件并带有由 ttk 状态及对应 Visual Styles API 状态值组成的元组的可选序列。*kw* 可能具有下列选项：

padding=padding
指定元素的内边距。*padding* 是由至多四个分别指定左、上、右、下边距值的整数组成的列表。如果指定的元素少于四个，则下边距默认等于上边距，右边距默认等于左边距。换句话说，由三个数字组成的列表将指定左边距、垂直边距和右边距；由两个数字组成的列表将指定水平边距和垂直边距；一个单独数字将为该部件的所有边指定相同的边距。此选项不可与其他任何选项混用。

margins=padding
指定元素的外边距。*padding* 是由至多四个分别指定左、上、右、下边距值的整数组成的列表。此选项不可与其他任何选项混用。

width=width
指定元素的宽度。如果设置了此选项则不会查询 Visual Styles API 来获取推荐的大小或部件。如果设置了此选项则还应当设置 *height*。*width* 和 *height* 选项不可与 *padding* 或 *margins* 选项混用。

height=height
指定元素的高度。参见 *width* 的注释。

示例:

```
style = ttk.Style(root)
style.element_create('pin', 'vsapi', 'EXPLORERBAR', 3, [
                     ('pressed', '!selected', 3),
                     ('active', '!selected', 2),
                     ('pressed', 'selected', 6),
                     ('active', 'selected', 5),
                     ('selected', 4),
                     ('', 1)])
style.layout('Explorer.Pin',
             [('Explorer.Pin.pin', {'sticky': 'news'})])
pin = ttk.Checkbutton(style='Explorer.Pin')
pin.pack(expand=True, fill='both')
```

在 3.13 版本发生变更: 增加了对"vsapi" 元素工厂的支持。

element_names()
返回当前主题已定义的元素列表。

element_options(*elementname*)
返回 *elementname* 元素的属性列表。

theme_create(*themename*, *parent=None*, *settings=None*)
新建一个主题。

如果 *themename* 已经存在，则会报错。如果给出了 *parent*，则新主题将从父主题继承样式、元素和布局。若给出了 *settings* ，则语法应与*theme_settings()* 的相同。

theme_settings(*themename*, *settings*)
将当前主题临时设为 *themename*，并应用 *settings*，然后恢复之前的主题。

settings 中的每个键都是一种样式而每个值可能包含'configure', 'map', 'layout' 和'element create' 等键并且它们被预期具有与分别由*Style.configure()*, *Style.map()*, *Style.layout()* 和*Style.element_create()* 方法所指定的相符的格式。

以下例子会对 Combobox 的默认主题稍作修改:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()

返回所有已知主题的列表。

theme_use(*themename=None*)

若未给出 *themename*，则返回正在使用的主题。否则，将当前主题设为 *themename*，刷新所有控件并引发 «ThemeChanged» 事件。

布局

布局可以为 None，如果未传入任何选项，或传入一个指明元素排列方式的字典的话。布局机制使用简化版本的打包位置管理器：给定一个初始容器，并为每个元素分配一个区块。

合法的属性/值可为：

***side*: 边缘**

指定元素置于容器的哪一侧；顶、右、底或左。如果省略，则该元素将占据整个容器。

***sticky*: 方向**

指定元素在已分配包装盒内的放置位置。

***unit*: 0 或 1**

如果设为 1，则将元素及其所有后代均视作单个元素以供*Widget.identify()* 等使用。它被用于滚动条之类带有控制柄的东西。

***children*: [子布局...]**

指定要放置于元素内的元素列表。每个元素都是一个元组（或其他序列类型），其中第一项是布局名称，另一项是个*Layout* 。

26.9 IDLE --- Python 编辑器和 shell

源代码： Lib/idlelib/

IDLE 是 Python 所内置的开发与学习环境。

IDLE 具有以下特性：

- 跨平台：在 Windows、Unix 和 macOS 上工作近似。
- 提供输入输出高亮和错误信息的 Python 命令行窗口（交互解释器）
- 提供多次撤销操作、Python 语法高亮、智能缩进、函数调用提示、自动补全等功能的多窗口文本编辑器
- 在多个窗口中检索，在编辑器中替换文本，以及在多个文件中检索（通过 grep）
- 提供持久保存的断点调试、单步调试、查看本地和全局命名空间功能的调试器
- 配置、浏览以及其它对话框

26.9.1 目录

IDEL 有两种主要的窗口类型：Shell 窗口和编辑器窗口。其中编辑器窗口可以同时打开多个。并且对于 Windows 和 Linux 平台，窗口顶部主菜单各不相同。以下每个菜单说明项，都标识了与之关联的平台类型。

导出窗口，例如使用编辑 => 在文件中查找是编辑器窗口的的一个子类型。它们目前有着相同的主菜单，但是默认标题和上下文菜单不同。

在 macOS 上，只有一个应用程序菜单。它会根据当前选择的窗口动态变化。它具有一个 IDLE 菜单，并且下面描述的某些条目已移动到符合 Apple 准则的位置。

文件菜单（命令行和编辑器）

新建文件
创建一个文件编辑器窗口。

打开...
使用打开窗口以打开一个已存在的文件。

打开模块...
打开一个已存在的模块（搜索 sys.path）

近期文件
打开一个近期文件列表，选取一个以打开它。

模块浏览器
于当前所编辑的文件中使用树形结构展示函数、类以及方法。在命令行中，首先打开一个模块。

路径浏览
在树状结构中展示 sys.path 目录、模块、函数、类和方法。

保存
如果文件已经存在，则将当前窗口保存至对应的文件。自打开或上次保存之后经过修改的文件的窗口标题栏首尾将出现星号 * 。如果没有对应的文件，则使用“另存为”代替。

保存为...
通过 Save As 对话框保存当前窗口。被保存的文件将成为关联到该窗口的新文件。（如果你的文件管理器被设为隐藏扩展名，则当前扩展名将在文件名文本框中被省略。如果新的文件名不带'.'，则将为 Python 和文本文件添加'.py' 和'.txt'，除非是在 macOS Aqua, 上，这时将为所有文件添加'.py'。）

另存为副本...
将当前窗口保存到不同的文件而不改变已关联的文件。（请参阅上文 Save As 中有关文件扩展名的说明。）

打印窗口
通过默认打印机打印当前窗口。

关闭窗口
关闭当前窗口（如果是未保存的编辑器窗口，则会提示保存；如果是未保存的 Shell 窗口，则会提示退出执行）。在 Shell 窗口中调用 `exit()` 或 `close()` 也会关闭 Shell 窗口。如果这是唯一的窗口，则还会退出 IDLE。

退出 IDLE
关闭所有窗口并退出 IDLE (将提示保存未保存的编辑窗口)。

编辑菜单（命令行和编辑器）

撤销操作
撤销当前窗口的最近一次操作。最高可以撤回 1000 条操作记录。

重做
重做当前窗口最近一次所撤销的操作。

全选
选择当前窗口的全部内容。

剪切
复制选区至系统剪贴板，然后删除选区。

复制
复制选区至系统剪贴板。

粘贴
插入系统剪贴板的内容至当前窗口。

剪贴板功能也可用于上下文目录。

查找...
打开一个提供多选项的查找窗口。

再次查找
重复上一次搜索（如果有的话）。

查找选区
查找当前选中的字符串，如果存在

在文件中查找...
打开文件查找对话框。将结果输出至新的输出窗口。

替换...
打开查找并替换对话框。

前往行
将光标移到所请求行的开头并使该行可见。对于超过文件尾的请求将会移到文件尾。清除所有选区并更新行列状态。

显示补全信息
打开一个可滚动列表以允许选择现有的名称。请参阅下面编辑与导航一节中的自动补全。

展开文本
展开键入的前缀以匹配同一窗口中的完整单词；重复以获得不同的扩展。

显示调用提示
在函数的右括号后，打开一个带有函数参数提示的小窗口。请参阅下面的“编辑和导航”部分中的调用提示。

显示周围括号
突出显示周围的括号。

格式菜单（仅 window 编辑器）

格式段落
在注释块或多行字符串或字符串中的选定行中，重新格式化当前以空行分隔的段落。段落中的所有行的格式都将少于 N 列，其中 N 默认为 72。

增加缩进
将选定的行右缩进（默认为 4 个空格）。

减少缩进
将选定的行左缩进（默认为 4 个空格）。

注释
在所选行的前面插入 ##。

取消注释
从所选行中删除开头的 # 或 ##。

制表符化
将 前导空格变成制表符。（注意：我们建议使用 4 个空格来缩进 Python 代码。）

取消制表符化
将 所有制表符转换为正确的空格数。

缩进方式切换
打开一个对话框，以在制表符和空格之间切换。

缩进宽度调整
打开一个对话框以更改缩进宽度。Python 社区接受的默认值为 4 个空格。

去除尾随空格
通过将 str.rstrip 应用于每行（包括多行字符串中的行），删除行尾非空白字符之后的尾随空格和其他空白字符。除 Shell 窗口外，在文件末尾删除多余的换行符。

运行菜单（仅 window 编辑器）

运行模块
执行检查模块 。如果没有错误，重新启动 shell 以清理环境，然后执行模块。输出显示在 shell 窗口中。请注意，输出需要使用“打印”或“写入”。执行完成后，Shell 将保留焦点并显示提示。此时，可以交互地探索执行的结果。这类似于在命令行执行带有 `python -i file` 的文件。

运行... 定制
与运行模块 相同，但使用自定义设置运行该模块。命令行参数扩展*sys.argv* ，就像在命令行上传递一样。该模块可以在命令行管理程序中运行，而无需重新启动。

检查模块
检查“编辑器”窗口中当前打开的模块的语法。如果尚未保存该模块，则 IDLE 会提示用户保存或自动保存，如在“空闲设置”对话框的“常规”选项卡中所选择的那样。如果存在语法错误，则会在“编辑器”窗口中指示大概位置。

Python Shell
打开或唤醒 Python Shell 窗口。

Shell 菜单（仅限 Shell 窗口）

查看最近重启
将 Shell 窗口滚动到上一次 Shell 重启。

重启 Shell
重启 shell 以清理环境，重置显示和异常处理。

上一条历史记录
循环浏览历史记录中与当前条目匹配的早期命令。

下一条历史记录
循环浏览历史记录中与当前条目匹配的后续命令。

中断执行
停止正在运行的程序。

调试菜单（仅限 Shell 窗口）

跳转到文件/行
查看当前行。以光标提示，且上一行为文件名和行号。如果找到目标，如果文件尚未打开则打开该文件，并显示目标行。使用此菜单项来查看异常回溯中引用的源代码行以及用文件中查找功能找到的行。也可在 Shell 窗口和 Output 窗口的上下文菜单中使用。

调试器（切换）
激活后，在 Shell 中输入的代码或从编辑器中运行的代码将在调试器下运行。在编辑器中，可以使用上下文菜单设置断点。此功能不完整，具有实验性。

堆栈查看器
在树状目录中显示最后一个异常的堆栈回溯，可以访问本地和全局。

自动打开堆栈查看器
在未处理的异常上切换自动打开堆栈查看器。

选项菜单（命令行和编辑器）

配置 IDLE
打开配置对话框并更改以下各项的首选项：字体、缩进、键绑定、文本颜色主题、启动窗口和大小、其他帮助源和扩展名。在 MacOS 上，通过在应用程序菜单中选择首选项来打开配置对话框。有关详细信息，请参阅：帮助和首选项下的首选项设置。

大多数配置选项适用于所有窗口或将来的所有窗口。以下选项仅适用于活动窗口。

显示/隐藏代码上下文（仅限编辑器窗口）
在编辑窗口顶部打开一个面板来显示在窗口顶部滚动的代码块上下文。请参阅下文“编辑与导航”章节中的代码上下文。

显示/隐藏行号（仅限 Editor 窗口）
在编辑窗口左侧打开一个显示代码文本行编号的列。默认为关闭显示，这可以在首选项中修改 (参见设置首选项)。

缩放/还原高度
在窗口的正常尺寸和最大高度之间进行切换。初始尺寸默认为 40 行每行 80 字符，除非在配置 IDLE 对话框的通用选项卡中做了修改。屏幕的最大高度由首次在屏幕上将缩小的窗口最大化的操作来确定。改变屏幕设置可能使保存的高度失效。此切换操作在窗口最大化状态下无效。

Window 菜单（命令行和编辑器）

列出所有打开的窗口的名称；选择一个将其带到前台（必要时对其进行去符号化）。

帮助菜单（命令行和编辑器）

关于 IDLE
显示版本，版权，许可证，荣誉等。

IDLE 帮助
显示此 IDLE 文档，详细介绍菜单选项，基本编辑和导航以及其他技巧。

Python 文档
访问本地 Python 文档（如果已安装），或启动 Web 浏览器并打开 docs.python.org 显示最新的 Python 文档。

海龟演示
使用示例 Python 代码运行 turtledemo 模块和海龟绘图

可以在“常规”选项卡下的“配置 IDLE”对话框中添加其他帮助源。有关“帮助”菜单选项的更多信息，请参见下面的帮助源 小节。

上下文菜单

通过在窗口中右击（在 macOS 上则为按住 Control 键点击）来打开一个上下文菜单。上下文菜单也具有编辑菜单中的标准剪贴板功能。

剪切
复制选区至系统剪贴板，然后删除选区。

复制
复制选区至系统剪贴板。

粘贴
插入系统剪贴板的内容至当前窗口。

编辑器窗口也具有断点功能。设置了断点的行会被特别标记。断点仅在启用调试器运行时有效。文件的断点会被保存在用户的 `.idlerc` 目录中。

设置断点
在当前行设置断点

清除断点
清除当前行断点

shell 和输出窗口还具有以下内容。

跳转到文件/行
与调试菜单相同。

Shell 窗口也有一个输出折叠功能，参见下文的 *Python Shell 窗口* 小节。

压缩
如果将光标位于输出行上，则会折叠在上方代码和下方提示直到'Squeezed text' 标签之间的所有输出。

26.9.2 编辑和导航

编辑窗口

IDLE 可以在启动时打开编辑器窗口，这取决于选项设置和你启动 IDLE 的方式。在此之后，请使用 File 菜单。对于给定的文件只能打开一个编辑器窗口。

标题栏包含文件名称、完整路径，以及运行该窗口的 Python 和 IDLE 版本。状态栏包含行号 ('Ln') 和列号 ('Col')。行号从 1 开始；列号则从 0 开始。

IDE 会定扩展名为.py* 的文件包含 Python 代码而其他文件不包含。可使用 Run 菜单来运行 Python 代码。

按键绑定

IDLE 插入光标是字符位置之间的一个细竖条。当输入字符时，插入光标及其右侧的所有内容将右移一个字符并使新字符输入到新空位中。

某些非字符类按键会移动光标并可能会删除字符。删除操作不会将文本放入剪贴板，但 IDLE 有一个撤销列表。当本文档讨论到按键时，'C' 是指 Windows 和 Unix 上的 `Control` 键和 macOS 上的 `Command` 键。（并且这样的讨论并假定这些按键没有被重新绑定到其他目标。）

- 方向键会使光标移动一个字符或一行。
- `C-LeftArrow` 和 `C-RightArrow` 会左移或右移一个单词。
- `Home` 和 `End` 会移至行的开头或末尾。
- `Page Up` 和 `Page Down` 会上移或下移一屏。
- `C-Home` 会 `C-End` 移至文档的开头或末尾。
- `Backspace` 和 `Del` (或 `C-d`) 会删除上一个或下一个字符。
- `C-Backspace` 和 `C-Del` 会向左或向右删除一个单词。
- `C-k` 会删除 (' 杀掉') 右侧的所有内容。

标准的键绑定（例如 `C-c` 复制和 `C-v` 粘贴）仍会有效。键绑定可在配置 IDLE 对话框中选择。

自动缩进

在一个代码块开头的语句之后，下一行会缩进 4 个空格符（在 Python Shell 窗口中是一个制表符）。在特定关键字之后（break, return 等），下一行将不再缩进。在开头的缩进中，按 `Backspace` 将会删除 4 个空格符。`Tab` 则会插入空格符（在 Python Shell 窗口中是一个制表符），具体数量取决于缩进宽度。目前，Tab 键按照 Tcl/Tk 的规定设置为四个空格符。

另请参阅*Format 菜单* 的缩进/取消缩进区的命令。

搜索和替换

任何选择都将成为搜索目标。但是，只有在同一行内的选择才有效因为搜索只针对行进行并会移除换行符。如果勾选了 `[x] Regular expression`，目标将按照 Python re 模块的规则来解读。

补全

当被请求并且可用时，将为模块名、类属性、函数或文件名提供补全。每次请求方法将显示包含现有名称的补全提示框。（例外情况参见下文的 Tab 补全。）对于任意提示框，要改变被补全的名称和提示框中被高亮的条目，可以通过输入和删除字符、按 `Up`, `Down`, `PageUp`, `PageDown`, `Home` 和 `End` 键；或是在提示框中单击。要关闭补全提示框可以通过 `Escape`, `Enter` 或按两次 `Tab` 键或是在提示框外单击。在提示框内双击则将执行选择并关闭。

有一种打开提示框的方式是输入一个关键字符并等待预设的一段间隔。此间隔默认为 2 秒；这可以在设置对话框中定制。（要防止自动弹出，可将时延设为一个很大的豪秒数值，例如 100000000。）对于导入的模块名或者类和函数属性，请输入'.'。对于根目录下的文件名，请在开头引号之后立即输入*os.sep* 或*os.altsep*。（在 Windows 下，可以先指定一个驱动器。）可通过输入目录名和分隔符来进入子目录。

除了等待，或是在提示框关闭之后，可以使用 Edit 菜单的 Show Completions 来立即打开一个补全提示框。默认的热键是 C-space。如果在打开提示框之前输入一某个名称的前缀，则将显示第一个匹配项或最接近的项。结果将与在提示框已显示之后输入前缀时相同。在一个引号之后执行 Show Completions 将会实例当前目录下的文件名而不是根目录下的。

在输入前缀后按 Tab 键的效果通常与 Show Completions 相同。（如果未输入前缀则为缩进。）但是，如果输入的前缀只有一个匹配项，则该匹配项会立即被添加到编辑器文本中而不打开补全提示框。

在字符串以外且开头不带'.' 地输入前缀并执行'Show Completions' 或按 Tab 键将打开一个包含关键字、内置名称和现有模块级名称的补全提示框。

当在编辑器（而非 Shell）中编辑代码时，可以通过运行你的代码并在此后不重启 Shell 来增加可用的模块级名称。这在文件顶部添加了导入语句之后会特别有用。这还会增加可用的属性补全。

补全提示框在初始时会排除以'_' 打头的名称，对于模块还会排除未包括在'__all__' 中的名称。这些隐藏名称可通过在'.' 之后输入'_' 来访问，这在提示框打开之前或之后都是有效的。

提示

当在一个 可用的函数名称之后输入（时将自动显示一个调用提示。函数名称表达式可以包括点号和方括号索引操作。调用提示将保持打开直到它被点击、光标移出参数区、或是输入了）。当光标位于某个定义的参数区时，可以在菜单中选择 Edit 的"Show Call Tip" 或是输入其快捷键来显示调用提示。

调用提示是由函数的签名和文档字符串到第一个空行或第五个非空行为止的内容组成的。（某些内置函数没有可访问的签名。）签名中的'/' 或'*' 指明其前面或后面的参数仅限以位置或名称（关键字）方式传入。具体细节可能会改变。

在 Shell 中，可访问的函数取决于有哪些模块已被导入用户进程，包括由 IDLE 本身导入的模块，以及哪些定义已被运行，以上均从最近的重启动开始算起。

例如，重启动 Shell 并输入 itertools.count(。将显示调用提示，因为 IDLE 出于自身需要已将 itertools 导入了用户进程。（此行为可能会改变。）输入 turtle.write(则不显示任何提示。因为 IDLE 本身不会导入 turtle。菜单项和快捷键同样不会有任何反应。输入 import turtle。则在此之后，turtle.write(将显示调用提示。

在编辑器中，import 语句在文件运行之前是没有效果的。在输入 import 语句之后、添加函数定义之后，或是打开一个现有文件之后可以先运行一下文件。

代码上下文

在一个包含 Python 代码的编辑器窗口内部，可以切换代码上下文以便显示或隐藏窗口顶部的面板。当显示时，此面板可以冻结代码块的开头行，例如以 class, def 或 if 关键字开头的行，这样的行在不显示时面板时可能离开视野。此面板的大小将根据需要扩展和收缩以显示当前层级的全部上下文，直至达到配置 IDLE 对话框中所定义的最大行数（默认为 15）。如果如果没有当前上下文行而此功能被启用，则将显示一个空行。点击上下文面板中的某一行将把该行移至编辑器顶部。

上下文面板的文本和背景颜色可在配置 IDLE 对话框的 Highlights 选项卡中进行配置。

Shell 窗口

在 IDLE 的 Shell 中，输入、编辑和重新执行完整的语句。（多数控制台和终端每次只能操作一个物理行）。

在光标位于行内任意位置的情况下按 Return 键来将单行语句提交执行。如果带有强制换行的反斜杠 (\)，则光标必须位于最后一个物理行。对于包含多行的复合语句要通过在语句之后额外输入一个空行来提交执行。

当将代码粘贴到 Shell 中时，它并不会被编译并执行除非如上所述地再按下 Return 键。可以先对粘贴的代码进行编辑。如果将多条语句粘贴到 Shell 中，则多条语句被当作一条语句来编译并引发 *SyntaxError*。

包含 RESTART 的行表明用户执行进程已被重启。这种情况会在用户执行进程崩溃、在 Shell 菜单中选择重启，或在编辑器窗口中运行代码时发生。

之前小节中介绍的编辑功能在交互式地输入代码时也适用。IDLE 的 Shell 窗口还会响应以下按键：

- C-c 会尝试中断语句执行（但可能会失败）。

- C-d 如果是在 >>> 提示符后按下会关闭窗口。
- Alt-p 和 Alt-n (在 macOS 上为 C-p 和 C-n) 将在当前提示符下恢复与已输入内容匹配的上一条或下一条之前输入的语句。
- 当光标位于任何之前的语句上时按下 Return 将把该语句添加到在提示符下已输入的内容之后。

文本颜色

IDLE 文本默认为白底黑字，但有特殊含义的文本将以彩色显示。对于 Shell 来说包括 Shell 输出，Shell 错误，用户输出和用户错误。对于 Shell 提示符下或编辑器中的 Python 代码来说则包括关键字，内置类和函数名称，class 和 def 之后的名称，字符串和注释等。对于任意文本窗口来说则包括光标（如果存在）、找到的文本（如果可能）和选定的文本。

IDLE 还会高亮显示模式匹配语句中的 软关键字 match, case 和 _。但是，这种高亮显示并不完美，在某些极端情况下还会出现错误，包括 _ 在 case 模式中出现的时候。

广西着色是在背景上完成的，因此有时会看到非着色的文本。要改变颜色方案，请使用配置 IDLE 对话框的高亮选项卡。编辑器中的调试器断点行标记和弹出面板和对话框中的文本则是用户不可配置的。

26.9.3 启动和代码执行

在附带 -s 选项启用的情况下，IDLE 将会执行环境变量 IDLESTARTUP 或 PYTHONSTARTUP 所引用的文件。IDLE 会先检查 IDLESTARTUP；如果 IDLESTARTUP 存在则会运行被引用的文件。如果 IDLESTARTUP 不存在，则 IDLE 会检查 PYTHONSTARTUP。这些环境变量所引用的文件是存放经常被 IDLE Shell 所使用的函数，或者执行导入常用模块的 import 语句的便捷场所。

此外，Tk 也会在存在启动文件时加载它。请注意 Tk 文件会被无条件地加载。此附加文件名为 .Idle.py 并且会在用户的家目录下查找。此文件中的语句将在 Tk 的命名空间中执行，所以此文件不适用于导入要在 IDLE 的 Python Shell 中使用的函数。

命令行用法

IDLE 可从命令行附带多种选项来唤起。基本语法为：

```
python -m idlelib [options] [file ...]
```

有以下几种选项：

-c <command>
: 在 shell 窗口中运行指定的 Python 命令。例如，传入 -c "print('Hello, World!')"。在 Windows 上，外面的引号如上面所显示的那样必须为双引号。

-d
: 启用调试器并打开 shell 窗口。

-e
: 打开一个 editor 窗口。

-h
: 打印一条带有一组合法选项的帮助消息框并退出。

-i
: 打开一个 shell 窗口。

-r <file>
: 在 shell 窗口中运行指定的文件。

-s
: 在打开 shell 窗口之前运行启动文件（由环境变量 IDLESTARTUP 或 PYTHONSTARTUP 定义）。

-t <title>
: 设置 shell 窗口的标题。

-

读取并执行 shell 窗口中的标准输入。此选项必须是位于其他参数之前的最后一项。

如果提供了参数：

- 如果使用了 `-`, `-c` 或 `-r`，则所有参数将放在 `sys.argv[1:]` 中，而 `sys.argv[0]` 将分别被设为 `''`, `'-c'` 或 `'-r'`。不会打开任何 editor 窗口，即使这是在 *Options* 对象框中的默认设置。
- 在其他情况下，参数将作为要被打开编辑的文件来处理，而 `sys.argv` 反映的则是传给 IDLE 本身的参数。

启动失败

IDLE 使用一个套接字在 IDLE GUI 进程和用户代码执行进程之间通信。当 Shell 启动或重启动时必须建立一个连接。(重启动会以一个内容为'RESTART' 的分隔行来标示)。如果用户进程无法连接到 GUI 进程，它通常会显示一个包含'cannot connect' 消息的 `Tk` 错误提示框来引导用户。随后将会退出程序。

有一个 Unix 系统专属的连接失败是由系统网络设置中错误配置的掩码规则导致的。当从一个终端启动 IDLE 时，用户将看到一条以 `** Invalid host:` 开头的消息。有效的值为 `127.0.0.1 (idlelib.rpc.LOCALHOST)`。用户可以在一个终端窗口输入 `tcpconnect -irv 127.0.0.1 6543` 并在另一个终端窗口中输入 `tcplisten <same args>` 来进行诊断。

导致连接失败的一个常见原因是用户创建的文件与标准库模块同名，例如 *random.py* 和 *tkinter.py*。当这样的文件与要运行的文件位于同一目录中时，IDLE 将无法导入标准库模块。可用的解决办法是重命名用户文件。

虽然现在已不太常见，但杀毒软件或防火墙程序也有可能会阻止连接。如果无法将此类程序设为允许连接，那么为了运行 IDLE 就必须将其关闭。允许这样的内部连接是安全的，因为数据在外部端口上不可见。一个类似的问题是错误的网络配置阻止了连接。

Python 的安装问题有时会使 IDLE 退出：存在多个版本时可能导致程序崩溃，或者单独安装时可能需要管理员权限。如果想要避免程序崩溃，或是不想以管理员身份运行，最简单的做法是完全卸载 Python 并重新安装。

有时会出现 pythonw.exe 僵尸进程问题。在 Windows 上，可以使用任务管理员来检查并停止该进程。有时由程序崩溃或键盘中断（control-C）所发起的重启动可能会出现连接失败。关闭错误提示框或使用 Shell 菜单中的 Restart Shell 可能会修复此类临时性错误。

当 IDLE 首次启动时，它会尝试读取 `~/.idlerc/` 中的用户配置文件（~ 是用户的家目录）。如果配置有问题，则应当显示一条错误消息。除随机磁盘错误之外，此类错误均可通过不手动编辑这些文件来避免。请使用 Options 菜单来打开配置对话框。一旦用户配置文件出现错误，最好的解决办法就是删除它并使用配置对话框重新设置。

如果 IDLE 退出时没有发出任何错误消息，并且它不是通过控制台启动的，请尝试通过控制台或终端 (`python -m idlelib`) 来启动它以查看是否会出现错误消息。

在基于 Unix 的系统上使用 tcl/tk 低于 `8.6.11` 的版本 (查看 `About IDLE`) 时特定字体的特定字符可能导致终端提示 tk 错误消息。这可能发生在启动 IDLE 编辑包此种字符的文件或是在之后输入此种字符的时候。如果无法升级 tcl/tk，可以重新配置 IDLE 来使用其他的字体。

运行用户代码

除了少量例外，使用 IDLE 执行 Python 代码的结果应当与使用默认方法，即在文本模式的系统控制台或终端窗口中直接通过 Python 解释器执行同样的代码相同。但是，不同的界面和操作有时会影响显示的结果。例如，`sys.modules` 初始时具有更多条目，而 `threading.active_count()` 将返回 2 而不是 1。

在默认情况下，IDLE 会在单独的 OS 进程中运行用户代码而不是在运行 Shell 和编辑器的用户界面进程中运行。在执行进程中，它会将 `sys.stdin`, `sys.stdout` 和 `sys.stderr` 替换为从 Shell 窗口获取输入并向其发送输出的对象。保存在 `sys.__stdin__`, `sys.__stdout__` 和 `sys.__stderr__` 中的原始值不会被改变，但可能会为 `None`。

将打印输出从一个进程发送到另一个进程中的文本部件要比打印到同一个进程中的系统终端慢。这在打印多个参数时将有更明显的影响，因为每个参数、每个分隔符和换行符对应的字符串都要单独发送。在开发中，这通常不算是问题，但如果希望能在 IDLE 中更快地打印，可以将想要显示的所有内容先格式化并拼接到一起然后打印单个字符串。格式字符串和*str.join()* 都可以被用于合并字段和文本行。

IDLE 的标准流替换不会被执行进程中创建的子进程所继承，不论它是由用户代码直接创建还是由 multiprocessing 之类的模块创建的。如果这样的子进程使用了 `input` 从 sys.stdin 输入或者使用了 `print` 或 `write` 向 sys.stdout 或 sys.stderr 输出，则应当在命令行窗口中启动 IDLE。(在 Windows 中，要使用 `python` 或 `py` 而不是 `pythonw` 或 `pyw`。) 这样二级子进程将会被附加到该窗口进行输入和输出。

如果 `sys` 被用户代码重置，例如使用了 `importlib.reload(sys)`，则 IDLE 的修改将丢失，来自键盘的输入和向屏幕的输出将无法正确运作。

当 Shell 获得焦点时，它将控制键盘与屏幕。这通常会保持透明，但一些直接访问键盘和屏幕的函数将会不起作用。这也包括那些确定是否有键被按下以及是哪个键被按下的系统专属函数。

在执行进程中运行的 IDLE 代码会向调用栈添加在其他情况下不存在的帧。IDLE 包装了 `sys.getrecursionlimit` 和 `sys.setrecursionlimit` 以减少额外栈帧的影响。

当用户代码直接或者通过调用 sys.exit 引发 SystemExit 时，IDLE 将返回 Shell 提示符而非完全退出。

Shell 中的用户输出

当一个程序输出文本时，结果将由相应的输出设备来确定。当 IDLE 执行用户代码时，`sys.stdout` 和 `sys.stderr` 会被连接到 IDLE Shell 的显示区。它的某些特性是从底层的 Tk Text 部件继承而来。其他特性则是程序所添加的。总之，Shell 被设计用于开发环境而非生产环境运行。

例如，Shell 绝不会丢弃输出。一个向 Shell 发送无限输出的程序将最终占满内存，导致内存错误。作为对比，某些系统文本模式窗口只会保留输出的最后 n 行。例如，Windows 控制台可由用户设置保留 1 至 9999 行，默认为 300 行。

IDLE 的 Shell 所使用的 Tk Text 部件可显示的字符 (码位) 为 Unicode 的 BMP (基本多语言平面) 子集。具体哪些字符会显示为正确的字形而哪些会以方框代替取决于操作系统和所安装的字体。制表符会使之后的文本从下一个制表位开始。(每 8 个' 字符' 对应一个制表位)。换行符会使之后的文本开始新的一行。其他控制符会被忽略或是显示为空格、方框或其他形式，具体取决于操作系统和字体。(使用方向键在这样的输出中移动文本光标有可能让显示间距发生怪异的变化。)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # 输入 22 个字符。
>>> len(s)
14
>>> s  # 显示 repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # 原样显示 s。
# 结果会因 OS 和字体而不同。 请试一下。
```

`repr` 函数会被用于表达式值的交互式回显。它将返回输入字符串的一个修改版本，其中的控制代码、部分 BMP 码位以及所有非 BMP 码位都将被替换为转义代码。如上面所演示的，它使用户可以辨识字符串中的字符，无论它们会如何显示。

普通的与错误的输出通常会在与代码输入和彼此之间保持区分 (显示于不同的行)。它们也会分别使用不同的高亮颜色。

对于 SyntaxError 回溯信息，表示检测到错误位置的正常'^' 标记被替换为带有代表错误的文本颜色高亮。当从文件运行的代码导致了其他异常时，用户可以右击回溯信息行在 IDLE 编辑器中跳转到相应的行。如有必要将打开相应的文件。

Shell 具有将输出行折叠为一个'Squeezed text' 标签的特殊功能。此功能将自动应用于超过 N 行的输出 (默认 N = 50)。N 可以在设置对话框中 General 页的 PyShell 区域中修改。行数更少的输出也可通过在输出上右击来折叠。此功能适用于过多的输出行数导致滚动操作变慢的情况。

已折叠的输出可通过双击该标签来原地展开。也可通过右击该标签将其发送到剪贴板或单独的查看窗口。

开发 tkinter 应用程序

IDLE 有意与标准 Python 保持区别以方便 tkinter 程序的开发。在标准 Python 中输入 `import tkinter as tk; root = tk.Tk()` 不会显示任何东西。在 IDLE 中输入同样的代码则会显示一个 tk 窗口。在标准 Python 中，还必须输入 `root.update()` 才会将窗口显示出来。IDLE 会在幕后执行同样的方法，每秒大约 20 次，即每隔大约 50 毫秒。下面输入 `b = tk.Button(root, text='button'); b.pack()`。在标准 Python 中仍然不会有任何可见的变化，直到输入 `root.update()`。

大多数 tkinter 程序都会运行 `root.mainloop()`，它通常直到 tk 应用被销毁时才会返回。如果程序是通过 `python -i` 或 IDLE 编辑器运行的，则 >>> Shell 提示符将直到 `mainloop()` 返回时才会出现，这时将结束程序的交互。

当通过 IDLE 编辑器运行 tkinter 程序时，可以注释掉 mainloop 调用。这样将立即回到 Shell 提示符并可与正在运行的应用程序交互。请记得当在标准 Python 中运行时重新启用 mainloop 调用。

在没有子进程的情况下运行

在默认情况下，IDLE 是通过一个套接字在单独的子进程中执行用户代码，它将使用内部的环回接口。这个连接在外部不可见并且不会在互联网上发送或接收数据。如果防火墙仍然会报警，你完全可以忽略。

如果创建套接字连接的尝试失败，IDLE 将会通知你。这样的失败可能是暂时性的，但是如果持续存在，问题可能是防火墙阻止了连接或某个系统配置错误。在问题得到解决之前，可以使用 -n 命令行开关来运行 IDLE。

如果 IDLE 启动时使用了 -n 命令行开关则它将在单个进程中运行并且将不再创建运行 RPC Python 执行服务器的子进程。这适用于 Python 无法在你的系统平台上创建子进程或 RPC 套接字接口的情况。但是，在这种模式下用户代码没有与 IDLE 本身相隔离。而且，当选择 Run/Run Module (F5) 时运行环境也不会重启。如果你的代码已被修改，你必须为受影响的模块执行 reload() 并重新导入特定的条目 (例如 from foo import baz) 才能让修改生效。出于这些原因，在可能的情况下最好还是使用默认的子进程来运行 IDLE。

自 3.4 版本弃用.

26.9.4 帮助和首选项 Help and Preferences

帮助源

Help 菜单项"IDLE Help" 会显示标准库参考中 IDLE 一章的带格式 HTML 版本。这些内容放在只读的 tkinter 文本窗口中，与在浏览器中看到的内容类似。可使用鼠标滚轮、滚动条或上下方向键来浏览文本。或是点击 TOC (Table of Contents) 按钮并在打开的选项框中选择一个节标题。

Help 菜单项"Python Docs" 会打开更丰富的帮助源，包括教程，通过 `docs.python.org/x.y` 来访问，其中'x.y' 是当前运行的 Python 版本。如果你的系统有此文档的离线副本 (这可能是一个安装选项)，则将打开这个副本。

选定的 URL 可以使用配置 IDLE 对话框的 General 选项卡随时在帮助菜单中添加或移除。

首选项设置

字体首选项、高亮、按键和通用首选项可通过 Option 菜单的配置 IDLE 项来修改。非默认的用户设置将保存在用户家目录下的 `.idlerc` 目录中。用户配置文件错误导致的问题可通过编辑或删除 `.idlerc` 中的一个或多个文件来解决。

在 Font 选项卡中，可以查看使用多种语言的多个字符的示例文本来了解字体或字号效果。可以编辑示例文本来添加想要的其他字符。请使用示例文本选择等宽字体。如果某些字符在 Shell 或编辑器中的显示有问题，可以将它们添加到示例文本的开头并尝试改变字号和字体。

在 Highlights 和 Keys 选项卡中，可以选择内置或自定义的颜色主题和按键集合。要将更新的内置颜色主题或按键集合与旧版 IDLE 一起使用，可以将其保存为新的自定义主题或按键集合就将可在旧版 IDLE 中使用。

macOS 上的 IDLE

在 System Preferences: Dock 中，可以将"Prefer tabs when opening documents" 设为"Always"。但是该设置不能兼容 IDLE 所使用的 tk/tkinter GUI 框架，并会使得部分 IDLE 特性失效。

扩展

IDLE 可以包含扩展插件。扩展插件的首选项可通过首选项对话框的 Extensions 选项卡来修改。请查看 idlelib 目录下 config-extensions.def 的开头来了解详情。目前唯一的扩展插件是 zzdummy，它也是一个测试用的示例。

26.9.5 idlelib --- IDLE 应用程序的实现

源代码: Lib/idlelib

Lib/idlelib 包实现了 IDLE 应用程序。请查看本页面的其余的内容来了解如何使用 IDLE。

有关 idlelib 中文件的描述见 idlelib/README.txt。可以在 idlelib 中或者在 IDLE 中点击 Help => About IDLE 来查看它。此文件还将 IDLE 菜单条目映射到了实现这些条目的代码。除了在'Startup' 中列出的文件以外，idlelib 代码都是' 私有' 的意即特性的修改可以被向下移植 (参见 **PEP 434**)。

CHAPTER 27

开发工具

本章中介绍的模块可帮助你编写软件。例如，*pydoc* 模块接受一个模块并根据该模块的内容来生成文档。*doctest* 和*unittest* 模块包含用于编写自动执行代码并验证是否产生预期的输出的单元测试的框架。

本章中描述的模块列表是：

27.1 typing ——对类型提示的支持

Added in version 3.5.

源代码： Lib/typing.py

> **备注**
>
> Python 运行时不强制要求函数与变量类型标注。它们可被类型检查器、IDE、语法检查器等第三方工具使用。

本模块提供了对类型提示的运行时支持。

考虑下面的函数：

```
def surface_area_of_cube(edge_length: float) -> str:
    return f"The surface area of the cube is {6 * edge_length ** 2}."
```

函数 surface_area_of_cube 接受一个预期为*float* 实例的参数，如*type hint* edge_length: float 所指明的。该函数预期返回一个*str* 实例，如 -> str 提示所指明的。

类型提示可以是简单的类比如*float* 或*str*，它们也可以更为复杂。*typing* 模块提供了一套用于更高级类型提示的词汇。

新特性被频繁添加到 typing 模块中。typing_extensions 包提供了这些新特性针对较旧版本 Python 的向下移植。

> **参见**

" 类型系统备忘单"
关于类型提示的概览（发布于 mypy 文档站点）

mypy 文档 的"Type System Reference" 章节
Python 类型系统是通过 PEP 来标准化的，因此该参考应当广泛适用于大多数 Python 类型检查器。（但某些部分仍然是 mypy 专属的。）

"Static Typing with Python"
由社区编写的不限定具体类型检查器的文档，详细讲解了类型系统特性，有用的类型相关工具以及类型的最佳实践。

27.1.1 有关 Python 类型系统的规范说明

Python 类型系统最新的标准规范说明可在 "Specification for the Python type system" 查看。

27.1.2 类型别名

类型别名是使用 `type` 语句来定义的，它将创建一个 *TypeAliasType* 的实例。在这个示例中，`Vector` 和 `list[float]` 将被静态类型检查器等同处理:

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# 通过类型检查；浮点数列表是合格的 Vector。
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

类型别名适用于简化复杂的类型签名。例如:

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# 静态类型检查器会认为上面的类型签名
# 完全等价于下面这个写法。
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]
) -> None:
    ...
```

`type` 语句是在 Python 3.12 中新增加的。为了向下兼容，类型别名也可以通过简单的赋值来创建:

```
Vector = list[float]
```

或者用 *TypeAlias* 标记来显式说明这是一个类型别名，而非一般的变量赋值:

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

27.1.3 NewType

用*NewType*助手创建与原类型不同的类型：

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

静态类型检查器把新类型当作原始类型的子类，这种方式适用于捕捉逻辑错误：

```
def get_user_name(user_id: UserId) -> str:
    ...

# 通过类型检查
user_a = get_user_name(UserId(42351))

# 未通过类型检查；整数不能作为 UserId
user_b = get_user_name(-1)
```

UserId 类型的变量可执行所有 int 操作，但返回结果都是 int 类型。这种方式允许在预期 int 时传入 UserId，还能防止意外创建无效的 UserId：

```
# 'output' 的类型为 'int' 而非 'UserId'
output = UserId(23413) + UserId(54341)
```

注意，这些检查只由静态类型检查器强制执行。在运行时，语句 Derived = NewType('Derived', Base) 将产生一个 Derived 可调用对象，该对象立即返回你传递给它的任何参数。这意味着语句 Derived(some_value) 不会创建一个新的类，也不会引入超出常规函数调用的很多开销。

更确切地说，在运行时，some_value is Derived(some_value) 表达式总为 True。

创建 Derived 的子类型是无效的:

```
from typing import NewType

UserId = NewType('UserId', int)

# 将在运行时失败且无法通过类型检查
class AdminUserId(UserId): pass
```

然而，我们可以在" 派生的" NewType 的基础上创建一个*NewType*。

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

同时，ProUserId 的类型检查也可以按预期执行。

详见 **PEP 484**。

备注

请记住使用类型别名将声明两个类型是相互 等价的。使用 type Alias = Original 将使静态类型检查器在任何情况下都把 Alias 视为与 Original 完全等价。这在你想要简化复杂的类型签名时会很有用处。

反之，NewType 声明把一种类型当作另一种类型的 子类型。Derived = NewType('Derived', Original) 时，静态类型检查器把 Derived 当作 Original 的 子类，即，Original 类型的值不能用在预期 Derived 类型的位置。这种方式适用于以最小运行时成本防止逻辑错误。

Added in version 3.5.2.

在 3.10 版本发生变更: NewType 现在是一个类而不是一个函数。因此，当调用 NewType 而非常规函数时会有一些额外的运行时开销。

在 3.11 版本发生变更: 调用 NewType 的性能已恢复到 Python 3.9 时的水平。

27.1.4 标注可调用对象

函数 -- 或是其他*callable* 对象 -- 可以使用*collections.abc.Callable* 或已被弃用的*typing.Callable* 来标注。Callable[[int], str] 表示一个接受*int* 类型的单个形参并返回一个*str* 的函数。

例如:

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # 函数体

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    ...  # 函数体

async def on_update(value: str) -> None:
    ...  # 函数体

callback: Callable[[str], Awaitable[None]] = on_update
```

下标语法总是要刚好使用两个值：参数列表和返回类型。参数列表必须是一个由类型组成的列表、*ParamSpec*、*Concatenate* 或省略号。返回类型必须是单一类型。

如果将一个省略号字面值 ... 作为参数列表，则表示可以接受包含任意形参列表的可调用对象:

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # 可以
x = concat  # 同样可以
```

Callable 无法表达复杂的签名如接受可变数量参数的函数，*重载的函数*，或具有仅限关键字形参的函数。但是，这些签名可通过自定义具有 __call__() 方法的*Protocol* 类来表达:

```
from collections.abc import Iterable
from typing import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None = None) -> list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) -> list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) -> list[bytes]:
    ...

batch_proc([], good_cb)  # 可以
batch_proc([], bad_cb)   # 错误！参数 2 的类型不兼容
                         # 因为在回调中有不同的名称和类别
```

以其他可调用对象为参数的可调用对象可以使用*ParamSpec* 来表明其参数类型是相互依赖的。此外，如果该可调用对象增加或删除了其他可调用对象的参数，可以使用*Concatenate* 操作符。它们分别采

取 `Callable[ParamSpecVariable, ReturnType]` 和 `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` 的形式。

在 3.10 版本发生变更: `Callable` 现在支持*ParamSpec* 和*Concatenate*。详情见 **PEP 612**。

参见

ParamSpec 和*Concatenate* 的文档提供了在 `Callable` 中使用的例子。

27.1.5 泛型（Generic）

由于无法以通用方式静态地推断容器中保存的对象的类型信息，标准库中的许多容器类都支持下标操作来以表示容器元素的预期类型。

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] 表明该序列中的所有元素
# 都必须是 "Employee" 的实例。
# Mapping[str, str] 表明该映射中的所有键和所有值
# 都必须是字符串。
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

泛型函数和类可以通过使用 类型形参语法来实现参数化:

```
from collections.abc import Sequence

def first[T](l: Sequence[T]) -> T:  # 函数是 TypeVar "T" 泛型
    return l[0]
```

或直接使用*TypeVar* 工厂:

```
from collections.abc import Sequence
from typing import TypeVar

U = TypeVar('U')                  # 声明类型变量 "U"

def second(l: Sequence[U]) -> U:  # 函数是 TypeVar "U" 泛型
    return l[1]
```

在 3.12 版本发生变更: 对泛型的语法支持是在 Python 3.12 中新增的。

27.1.6 标注元组

对于 Python 中的大多数容器，类型系统会假定容器中的所有元素都是相同类型的。例如:

```
from collections.abc import Mapping

# 类型检查器将推断 ``x`` 中的所有元素均为整数
x: list[int] = []

# 类型检查器错误：``list`` 只接受单个类型参数：
y: list[int, str] = [1, 'foo']

# 类型检查器将推断 ``z`` 中的所有键均为字符串，
# 并且 ``z`` 中的所有值均为字符串或整数
z: Mapping[str, str | int] = {}
```

list 只接受一个类型参数，因此类型检查器将在上述代码中对 `y` 赋值时报告错误。同样，*Mapping* 只接受两个类型参数：第一个给出键的类型，第二个则给出值的类型。

然而，与大多数其它 Python 容器不同的是，在常见的 Python 代码中，元组中元素的类型并不相同。因此，在 Python 的类型系统中，元组是特殊情况。*tuple* 可以接受 *任意数量的类型参数*：

```
# 可以：``x`` 被赋值为长度为 1 的元组，其中的唯一元素是个整数
x: tuple[int] = (5,)

# 可以：``y`` 被赋值为长度为 2 的元素；
# 第 1 个元素是个整数，第 2 个元素是个字符串
y: tuple[int, str] = (5, "foo")

# 错误：类型标注表明是长度为 1 的元组，
# 但 ``z`` 却被赋值为长度为 3 的元组
z: tuple[int] = (1, 2, 3)
```

要表示一个可以是 *任意长度的元组*，并且其中的所有元素都是相同类型的 `T`，请使用 `tuple[T, ...]`。要表示空元组，请使用 `tuple[()]`。只使用 `tuple` 作为注解等效于使用 “tuple[Any, ...]“：

```
x: tuple[int, ...] = (1, 2)
# 这些赋值是可以的 OK：``tuple[int, ...]`` 表明 x 可以为任意长度
x = (1, 2, 3)
x = ()
# 这个赋值是错误的：``x`` 中的所有元素都必须为整数
x = ("foo", "bar")

# ``y`` 只能被赋值为一个空元组
y: tuple[()] = ()

z: tuple = ("foo", "bar")
# 这些重新赋值是可以的 OK：简单的 ``tuple`` 等价于 ``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

27.1.7 类对象的类型

带有 `C` 标注的变量可接受 `C` 类型的值。反之，带有 `type[C]` (或已被弃用的*typing.Type[C]*) 标注的变量则可接受本身是类的值 -- 准确地说，它将接受 `C` 的 *类对象*。例如：

```
a = 3         # 为 ``int`` 类型
b = int       # 为 ``type[int]`` 类型
c = type(a)   # 同样为 ``type[int]`` 类型
```

注意，`type[C]` 是协变的：

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...

def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)      # 可以
make_new_user(ProUser)   # 同样可以：``type[ProUser]`` 是 ``type[User]`` 的子类型
make_new_user(TeamUser)  # 仍然可以
make_new_user(User())    # 错误：预期为 ``type[User]`` 但得到 ``User``
make_new_user(int)       # 错误：``type[int]`` 不是 ``type[User]`` 的子类型
```

type 的合法形参只有类, *Any*, *类型变量* 以及前面这些类型的并集。例如：

```
def new_non_team_user(user_class: type[BasicUser | ProUser]): ...

new_non_team_user(BasicUser)  # 可以
new_non_team_user(ProUser)    # 可以
new_non_team_user(TeamUser)   # 错误：``type[TeamUser]`` 不是
                              # ``type[BasicUser | ProUser]`` 的子类型
new_non_team_user(User)       # 同样错误
```

`type[Any]` 等价于*type*，它是 Python 的 元类层级结构的根对象。

27.1.8 标注生成器和协程

生成器可以使用泛型类型*Generator[YieldType, SendType, ReturnType]* 来标。例如:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

请注意与标准库里的许多其他泛型类不同，*Generator* 的 SendType 采用逆变行为，而不是协变或不变行为。

SendType 和 ReturnType 形参默认为 None：

```
def infinite_stream(start: int) -> Generator[int]:
    while True:
        yield start
        start += 1
```

也可以显式设置这些类型：

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

仅产生值的简单生成器可以被标注为具有*Iterable[YieldType]* 或*Iterator[YieldType]* 类型的返回值:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

异步生成器的处理方式类似，但不要指望有 ReturnType 类型参数 (*AsyncGenerator[YieldType, SendType]*)。SendType 参数默认为 None，因此以下定义是等价的:

```
async def infinite_stream(start: int) -> AsyncGenerator[int]:
    while True:
        yield start
        start = await increment(start)

async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

与同步情况一样，*AsyncIterable[YieldType]* 和*AsyncIterator[YieldType]* 也可用:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

协程可使用 *[YieldType, SendType, ReturnType]* 进行注释。泛型参数对应于 *Generator* 的参数，例如:

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # 在其他地方定义的协程
x = c.send('hi')                   # 推断 'x' 的类型为 list[str]
async def bar() -> None:
    y = await c                    # 推断 'y' 的类型为 int
```

27.1.9 用户定义的泛型类型

用户定义的类可以定义为泛型类。

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

这种语法表示类 `LoggedVar` 是围绕单个类型变量 `T` 实现参数化的。这也使得 `T` 成为类体内部有效的类型。

泛型类隐式继承自 *Generic*。为了与 Python 3.11 及更低版本兼容，也允许显式地从 *Generic* 继承以表示泛型类:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class LoggedVar(Generic[T]):
    ...
```

泛型类具有 `__class_getitem__()` 方法，这意味着泛型类可在运行时进行参数化（例如下面的 `LoggedVar[int]`）:

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

一个泛型可以有任何数量的类型变量。所有种类的 *TypeVar* 都可以作为泛型的参数:

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...

OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes], covariant=True)
OldS = TypeVar('OldS', int, str)

class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

Generic 类型变量的参数应各不相同。下列代码就是无效的：

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # 无效
    ...
```

泛型类也可以从其他类继承：

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

从泛型类继承时，某些类型参数可被固定：

```
from collections.abc import Mapping

class MyDict[T](Mapping[str, T]):
    ...
```

在这个例子中，`MyDict` 就只有一个参数 `T`。

未指定泛型类的类型参数时，会假定每个位置的类型都为*Any*。在下面的例子中，`MyIterable` 不是泛型，但却隐式继承了 `Iterable[Any]`：

```
from collections.abc import Iterable

class MyIterable(Iterable): # 与 Iterable[Any] 相同
    ...
```

用户定义的泛型类型别名也同样受到支持。例如：

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# 这里的返回类型与 Iterable[str] | int 相同
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]
```

(续下页)

(接上页)

```
def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: # 与 Iterable[tuple[T, T]] 相同
    return sum(x*y for x, y in v)
```

出于向后兼容性的考虑，也允许使用简单的赋值来创建泛型类型别名：

```
from collections.abc import Iterable
from typing import TypeVar

S = TypeVar("S")
Response = Iterable[S] | int
```

在 3.7 版本发生变更: *Generic* 不再支持自定义元类。

在 3.12 版本发生变更: 3.12 版本新增了对泛型和类型别名的语法支持。在之前的版本中，泛型类必须显式继承自*Generic*，或者在其基类之一中包含有类型变量。

用户定义的参数表达式的泛型也受到支持，可以采用 `[**P]` 形式的参数规格变量来表示。该行为与上面描述的类型变量一致，因为参数规格变量被 typing 模块视为专门的类型变量。这方面的一个例外是，类型的列表可用于替代*ParamSpec*：

```
>>> class Z[T, **P]: ...  # T 为 TypeVar；P 为 ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

带有*ParamSpec* 的泛型类也可以使用从*Generic* 显式继承的方式来创建。在这种情况下，不需要使用 `**`：

```
from typing import ParamSpec, Generic

P = ParamSpec('P')

class Z(Generic[P]):
    ...
```

TypeVar 与*ParamSpec* 的另一个区别在于只有单个参数规格变量的泛型会接受形如 `X[[Type1, Type2, ...]]` 的参数列表，同时为了美观，也接受 `X[Type1, Type2, ...]` 这样的形式。在内部，后者被转换为前者，所以下面的内容是等价的：

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

请注意：在某些情况下，具有*ParamSpec* 的泛型在替换后可能不具有正确的 `__parameters__`，因为参数规格主要用于静态类型检查。

在 3.10 版本发生变更: *Generic* 现在可以通过参数表达式进行参数化。参见*ParamSpec* 和 **PEP 612** 以了解更多细节。

用户定义的泛型类可以将 ABC 作为基类而不会导致元类冲突。参数化泛型的输出结果会被缓存，且 typing 模块中的大多数类型都是*hashable* 并且支持相等性比较。

27.1.10 Any 类型

Any 是一种特殊的类型。静态类型检查器认为所有类型均与*Any* 兼容，同样，*Any* 也与所有类型兼容。

也就是说，可对*Any* 类型的值执行任何操作或方法调用，并赋值给任意变量：

```
from typing import Any

a: Any = None
a = []          # 可以
a = 2           # 可以

s: str = ''
s = a           # 可以

def foo(item: Any) -> int:
    # 通过类型检查；'item' 可以为任意类型，
    # 并且其类型会具有 'bar' 方法
    item.bar()
    ...
```

注意，*Any* 类型的值赋给更精确的类型时，不执行类型检查。例如，把 a 赋给 s，在运行时，即便 s 已声明为*str* 类型，但接收*int* 值时，静态类型检查器也不会报错。

此外，未指定返回值与参数类型的函数，都隐式地默认使用*Any*：

```
def legacy_parser(text):
    ...
    return data

# 静态类型检查器将认为上面的函数
# 具有与下面的函数相同的签名：
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

需要混用动态与静态类型代码时，此操作把*Any* 当作 *应急出口*。

Any 和*object* 的区别。与*Any* 相似，所有类型都是*object* 的子类型。然而，与*Any* 不同，object 不可逆：*object* 不是其它类型的子类型。

就是说，值的类型是*object* 时，类型检查器几乎会拒绝所有对它的操作，并且，把它赋给更精确的类型变量（或返回值）属于类型错误。例如：

```
def hash_a(item: object) -> int:
    # 不能通过类型检查；对象没有 'magic' 方法。
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # 通过类型检查
    item.magic()
    ...

# 通过类型检查，因为整数和字符串都是 object 的子类
hash_a(42)
hash_a("foo")

# 通过类型检查，因为 Any 可以兼容所有类型
hash_b(42)
hash_b("foo")
```

使用*object*，说明值能以类型安全的方式转为任何类型。使用*Any*，说明值是动态类型。

27.1.11 名义子类型 vs 结构子类型

最初 PEP 484 将 Python 静态类型系统定义为使用 *名义子类型*。这意味着当且仅当类 `A` 是 `B` 的子类时，才满足有类 `B` 预期时使用类 `A` 。

此项要求以前也适用于抽象基类，例如，*Iterable* 。这种方式的问题在于，定义类时必须显式说明，既不 Pythonic，也不是动态类型式 Python 代码的惯用写法。例如，下列代码就遵从了 PEP 484 的规范：

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 允许用户在类定义时不显式说明基类，从而解决了这一问题，静态类型检查器隐式认为 `Bucket` 既是 `Sized` 的子类型，又是 `Iterable[int]` 的子类型。这就是 *结构子类型*（又称为静态鸭子类型）：

```
from collections.abc import Iterator, Iterable

class Bucket:  # 注意：没有基类
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # 通过类型检查
```

此外，结构子类型的优势在于，通过继承特殊类*Protocol* ，用户可以定义新的自定义协议（见下文中的例子）。

27.1.12 模块内容

`typing` 模块定义了下列类、函数和装饰器。

特殊类型原语

特殊类型

这些类型可用于在注解中表示类型，但不支持下标用法（`[]`）。

`typing.`**`Any`**

特殊类型，表示没有约束的类型。

- 所有类型都与*Any* 兼容。
- *Any* 与所有类型都兼容。

在 3.11 版本发生变更: *Any* 现在可以用作基类。这有助于避免类型检查器在高度动态或可通过鸭子类型使用的类上报错。

`typing.`**`AnyStr`**

受约束的类型变量。

定义：

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

`AnyStr` 用于可接受*str* 或*bytes* 参数但不允许两者混用的函数。

例如：

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat("foo", "bar")    # 可以，输出为 'str' 类型
concat(b"foo", b"bar")  # 可以，输出为 'bytes' 类型
concat("foo", b"bar")   # 错误，不可混用 str 和 bytes
```

请注意：尽管名为 `AnyStr`，但它与*Any* 类型毫无关系，也不是指 “任何字符串”。而且，`AnyStr` 更是和 `str | bytes` 彼此互不相同，各有各的使用场景：

```
# AnyStr 的无效使用：
# 类型变量在函数签名中仅使用一次，
# 因此无法通过类型检查器 “解决”
def greet_bad(cond: bool) -> AnyStr:
    return "hi there!" if cond else b"greetings!"

# 注释此函数的更好方法：
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

Deprecated since version 3.13, will be removed in version 3.18: 已被弃用而应改用新的 类型形参语法。使用 `class A[T: (str, bytes)]: ...` 而不是导入 `AnyStr`。详情参见 **PEP 695**。

在 Python 3.16 中，`AnyStr` 将从 `typing.__all__` 中被移除，在运行时当它被访问或从 `typing` 导入时将发出弃用警告。在 Python 3.18 中 `AnyStr` 将从 `typing` 中被移除。

typing.**LiteralString**

只包括字符串字面值的的特殊类型。

任何字符串字面值或其他 `LiteralString` 都与 `LiteralString` 兼容。但 `str` 类型的对象不与其兼容。组合 `LiteralString` 类型的对象产生的字符串也被认为是 `LiteralString`。

示例:

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string: LiteralString) -> None:
    run_query("SELECT * FROM students")  # 可以
    run_query(literal_string)  # 可以
    run_query("SELECT * FROM " + literal_string)  # 可以
    run_query(arbitrary_string)  # 类型检查器错误
    run_query(  # 类型检查器错误
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

`LiteralString` 对于会因用户可输入任意字符串而导致问题的敏感 API 很有用。例如，上述两处导致类型检查器报错的代码可能容易被 SQL 注入攻击。

请参阅 **PEP 675** 了解详情。

Added in version 3.11.

typing.**Never**

typing.**NoReturn**

`Never` 和 `NoReturn` 代表 底类型，一种没有成员的类型。

它们可被用于指明一个函数绝不会返回，例如*sys.exit()*:

```
from typing import Never  # 或 NoReturn

def stop() -> Never:
    raise RuntimeError('no way')
```

或者用于定义一个绝不应被调用的函数，因为不存在有效的参数，例如*assert_never()*：

```
from typing import Never  # 或 NoReturn

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
    never_call_me(arg)  # 类型检查器错误
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never (or NoReturn)
```

`Never` 和 `NoReturn` 在类型系统中具有相同的含义并且静态类型检查器会以相同的方式对待这两者。

Added in version 3.6.2: 增加了*NoReturn*。

Added in version 3.11: 增加了*Never*。

typing.**Self**

特殊类型，表示当前闭包内的类。

例如：

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # 揭示的类型为 "Foo"
reveal_type(SubclassOfFoo().return_self())  # 揭示的类型为 "SubclassOfFoo"
```

此注解在语法上等价于以下代码，但形式更为简洁：

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

通常来说，如果某些内容返回 `self`，如上面的示例所示，您应该使用 `Self` 作为返回值注解。如果 `Foo.return_self` 被注解为返回 `"Foo"`，那么类型检查器将推断从 `SubclassOfFoo.return_self` 返回的对象是 `Foo` 类型，而不是 `SubclassOfFoo`。

其它常见用例包括：

- 被用作替代构造器的*classmethod*，它将返回 `cls` 形参的实例。
- 标注一个返回自身的 `__enter__()` 方法。

如果不能保证在子类中方法会返回子类的实例（而非父类的实例），则不应使用 `Self` 作为返回值注解：

```
class Eggs:
    # 在这里 self 是一个不正确的返回注释，
    # 因为返回的对象始终是 Eggs 的一个实例，
    # 即使在子类中
    def returns_eggs(self) -> "Eggs":
        return Eggs()
```

更多细节请参见 **PEP 673**。

Added in version 3.11.

typing.**TypeAlias**

特殊注解，用于显式声明类型别名.

例如：

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

在较早的 Python 版本上，`TypeAlias` 对注解使用前向引用的别名时特别有用，因为类型检查器可能很难将这些别名与正常的变量赋值区分开来：

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" 还不存在，
# 因此我们必须在 Python <3.12 版本中使用引号进行前向引用。
# 使用 ``TypeAlias`` 告诉类型检查器这是一个类型别名声明，
# 而不是对字符串的变量赋值。
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

请参阅 **PEP 613** 了解详情。

Added in version 3.10.

自 3.12 版本弃用：*TypeAlias* 被弃用，请使用 `type` 语句，后者创建*TypeAliasType* 的实例，并且天然支持正向引用。请注意，虽然*TypeAlias* 和*TypeAliasType* 具有相似的用途和名称，但它们是不同的，后者并不是前者的类型。目前还没有移除*TypeAlias* 的计划，但鼓励用户迁移到 `type` 语句。

特殊形式

这些内容在注解中可以视为类型，且都支持下标用法（`[]`），但每个都有唯一的语法。

typing.**Union**

联合类型；`Union[X, Y]` 等价于 `X | Y`，意味着满足 X 或 Y 之一。

要定义一个联合类型，可以使用类似 `Union[int, str]` 或简写 `int | str`。建议使用这种简写。细节：

- 参数必须是某种类型，且至少有一个。
- 联合类型之联合类型会被展平，例如：

  ```
  Union[Union[int, str], float] == Union[int, str, float]
  ```

- 单参数之联合类型就是该参数自身，例如：

```
Union[int] == int  # 该构造器确实返回 int
```

- 冗余的参数会被跳过，例如：

```
Union[int, str, int] == Union[int, str] == int | str
```

- 比较联合类型，不涉及参数顺序，例如：

```
Union[int, str] == Union[str, int]
```

- 不可创建 Union 的子类或实例。
- 没有 Union[X][Y] 这种写法。

在 3.7 版本发生变更: 在运行时，不要移除联合类型中的显式子类。

在 3.10 版本发生变更: 联合类型现在可以写成 X | Y。参见联合类型表达式。

typing.**Optional**

Optional[X] 等价于 X | None （或 Union[X, None] ）。

注意，可选类型与含默认值的可选参数不同。含默认值的可选参数不需要在类型注解上添加 Optional 限定符，因为它仅是可选的。例如：

```
def foo(arg: int = 0) -> None:
    ...
```

另一方面，显式应用 None 值时，不管该参数是否可选，Optional 都适用。例如：

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

在 3.10 版本发生变更: 可选参数现在可以写成 X | None。参见联合类型表达式。

typing.**Concatenate**

特殊形式，用于注解高阶函数。

Concatenate 可用于与*Callable* 和*ParamSpec* 连用来注解高阶可调用对象，该可象可以添加、移除或转换另一个可调用对象的形参。使用形式为 Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]。Concatenate 目前仅可用作传给*Callable* 的第一个参数。传给 Concatenate 的最后一个形参必须是*ParamSpec* 或省略号（...）。

例如，为了注释一个装饰器 with_lock，它为被装饰的函数提供了*threading.Lock*，Concatenate 可以用来表示 with_lock 期望一个可调用对象，该对象接收一个 Lock 作为第一个参数，并返回一个具有不同类型签名的可调用对象。在这种情况下，*ParamSpec* 表示返回的可调用对象的参数类型取决于被传入的可调用程序的参数类型：

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# 使用此锁来确保在任何时候只有一个线程正在执行某个函数。
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R]) -> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
```

(续下页)

(接上页)

```
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# 由于装饰器的存在，我们不需要自己传递锁。
sum_threadsafe([1.1, 2.2, 3.3])
```

Added in version 3.10.

参见

- **PEP 612** -- 参数规范变量（引入 ParamSpec 和 Concatenate 的 PEP）
- *ParamSpec*
- 标注可调用对象

typing.**Literal**

特殊类型注解形式，用于定义“字面值类型”。

Literal 可以用来向类型检查器说明被注解的对象具有与所提供的字面量之一相同的值。

例如：

```
def validate_simple(data: Any) -> Literal[True]:  # 总是返回 True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')      # 通过类型检查
open_helper('/other/path', 'typo')  # 类型检查错误
```

Literal[...] 不能创建子类。在运行时，任意值均可作为 Literal[...] 的类型参数，但类型检查器可以对此加以限制。字面量类型详见 **PEP 586** 。

Added in version 3.8.

在 3.9.1 版本发生变更: Literal 现在能去除形参的重复。Literal 对象的相等性比较不再依赖顺序。现在如果有某个参数不为*hashable*，Literal 对象在相等性比较期间将引发*TypeError*。

typing.**ClassVar**

特殊类型注解构造，用于标注类变量。

如 **PEP 526** 所述，打包在 ClassVar 内的变量注解是指，给定属性应当用作类变量，而不应设置在类实例上。用法如下：

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # 类变量
    damage: int = 10                     # 实例变量
```

ClassVar 仅接受类型，也不能使用下标。

ClassVar 本身不是类，不应用于*isinstance()* 或*issubclass()*。*ClassVar* 不改变 Python 运行时行为，但可以用于第三方类型检查器。例如，类型检查器会认为以下代码有错：

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # 错误，在实例上设置类变量
Starship.stats = {}     # 这是可以的
```

Added in version 3.5.3.

在 3.13 版本发生变更: 现在*ClassVar* 可以被嵌套在*Final* 中，反之亦然。

typing.**Final**

特殊类型注解构造，用于向类型检查器表示最终名称。

不能在任何作用域中重新分配最终名称。类作用域中声明的最终名称不能在子类中重写。

例如:

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # 类型检查器将报告错误

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # 类型检查器将报告错误
```

这些属性没有运行时检查。详见 **PEP 591**。

Added in version 3.8.

在 3.13 版本发生变更: 现在*Final* 可以被嵌套在*ClassVar* 中，反之亦然。

typing.**Required**

特殊类型注解构造，用于标记*TypedDict* 键为必填项。

这主要用于 `total=False` 的 TypedDict。有关更多详细信息，请参阅*TypedDict* 和 **PEP 655** 。

Added in version 3.11.

typing.**NotRequired**

特殊类型注解构造，用于标记*TypedDict* 键为可能不存在的键。

详情参见*TypedDict* 和 **PEP 655**。

Added in version 3.11.

typing.**ReadOnly**

一个特殊的类型标注构造，用于将*TypedDict* 的项标记为只读。

例如:

```
class Movie(TypedDict):
   title: ReadOnly[str]
   year: int

def mutate_movie(m: Movie) -> None:
   m["year"] = 1999  # allowed
   m["title"] = "The Matrix"  # 类型检查错误
```

这个属性没有运行时检查。

详见*TypedDict* 和 **PEP 705**。

Added in version 3.13.

typing.**Annotated**

特殊类型注解形式，用于向注解添加特定于上下文的元数据。

使用注解 `Annotated[T, x]` 将元数据 `x` 添加到给定类型 `T` 。使用 `Annotated` 添加的元数据可以被静态分析工具使用，也可以在运行时使用。在运行时使用的情况下，元数据存储在 `__metadata__` 属性中。

如果库或工具遇到注解 `Annotated[T, x]`，并且没有针对这一元数据的特殊处理逻辑，则应该忽略该元数据，简单地将注解视为 `T`。因此，`Annotated` 对于希望将注解用于 Python 的静态类型注解系统之外的目的的代码很有用。

使用 `Annotated[T, x]` 作为注解仍然允许对 `T` 进行静态类型检查，因为类型检查器将简单地忽略元数据 `x`。因此，`Annotated` 不同于*@no_type_check* 装饰器，后者虽然也可以用于在类型注解系统范围之外添加注解，但是会完全禁用对函数或类的类型检查。

具体解释元数据的方式由遇到 `Annotated` 注解的工具或库来负责。遇到 `Annotated` 类型的工具或库可以扫描元数据的各个元素以确定其是否有意处理（比如使用*isinstance()*）。

Annotated[<type>, <metadata>]

以下示例演示在进行区间范围分析时使用 `Annotated` 将元数据添加到类型注解的方法：

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

传给 `Annotated` 的第一个参数必须是合法的类型。可以将多个元数据元素作为支持可变参数的 `Annotated` 来提供。元数据元素的顺序将被保留并会影响相等性检测：

```
@dataclass
class ctype:
     kind: str

a1 = Annotated[int, ValueRange(3, 10), ctype("char")]
a2 = Annotated[int, ctype("char"), ValueRange(3, 10)]

assert a1 != a2  # 顺序会有影响
```

由处理注解的工具决定是否允许向一个注解中添加多个元数据元素，以及如何合并这些注解。

嵌套的 `Annotated` 类型会被展平。元数据元素从最内层的注解开始依次展开：

```
assert Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] == Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

元数据中的重复元素不会被移除：

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

`Annotated` 可以与嵌套别名和泛型别名一起使用：

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# 当在类型注释中使用时，类型检查器会将“V”视为
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

`Annotated` 不能与已解包的*TypeVarTuple* 一起使用：

```
type Variadic[*Ts] = Annotated[*Ts, Ann1] = Annotated[T1, T2, T3, ..., Ann1]  # 不合法
```

其中 T1, T2 ... 都是*TypeVars*。这是不合法的因为只能传递一种类型给 Annotated。

默认情况下，*get_type_hints()* 会去除注解中的元数据。传入 include_extras=True 可以保留元数据：

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None: pass
...
>>> get_type_hints(func)
{'x': <class 'int'>, 'return': <class 'NoneType'>}
>>> get_type_hints(func, include_extras=True)
{'x': typing.Annotated[int, 'metadata'], 'return': <class 'NoneType'>}
```

在运行时，与特定 Annotated 类型相关联的元数据可通过 __metadata__ 属性来获取：

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

如果你想要获取由 Annotated 包装的原始类型，请使用 __origin__ 属性：

```
>>> from typing import Annotated, get_origin
>>> Password = Annotated[str, "secret"]
>>> Password.__origin__
<class 'str'>
```

请注意使用*get_origin()* 将返回 Annotated 本身：

```
>>> get_origin(Password)
typing.Annotated
```

参见

PEP 593 - 灵活的函数与变量标注
该 PEP 将 Annotated 引入到标准库中。

Added in version 3.9.

typing.**TypeIs**

特殊类型注解构造，用于标记用户定义的谓词函数。

TypeIs 能用来注解用户定义的谓词函数的返回值类型。TypeIs 接受单个类型参数。如此标注的函数在运行时应当有至少一个位置参数，并且返回一个布尔值。

TypeIs 旨在方便 类型收窄 -- 一个被静态类型检查器使用，用来更精准地决定程序代码流中表达式类型的技巧。通常类型收窄通过分析有条件的代码流并对代码块执行类型收窄实现。此处的条件表达式有时也被称为“类型谓词”。

```
def is_str(val: str | float):
    # "isinstance" 类型谓词
    if isinstance(val, str):
        #  ``val`` 的类型缩小为 ``str``
        ...
    else:
```

(续下页)

(接上页)

```
        # 否则，``val`` 的类型缩小为 ``float``。
        ...
```

使用一个用户定义的布尔函数作为类型谓词有时很方便。这样的函数应当将 `TypeIs[...]` 或*TypeGuard* 作为它的返回类型，以向静态类型检查器传达这个意图。`TypeIs` 的行为通常比 `TypeGuard` 更直观，但在函数的输入与输出类型不兼容 (例如从 `list[object]` 到 `list[int]`) 或函数不会对所有收窄后类型的实例返回 `True` 时不能使用。

使用 `-> TypeIs[NarrowedType]` 告诉静态类型检查器对于给定的函数：

1. 返回一个布尔值。
2. 如果返回值是 `True`，那么其参数的类型收窄到参数本身类型与 `NarrowedType` 的并。
3. 如果返回值是 `False`，那么其参数的类型收窄到排除 `NarrowedType`。

例如：

```
from typing import assert_type, final, TypeIs

class Parent: pass
class Child(Parent): pass
@final
class Unrelated: pass

def is_parent(val: object) -> TypeIs[Parent]:
    return isinstance(val, Parent)

def run(arg: Child | Unrelated):
    if is_parent(arg):
        # ``arg`` 的类型缩小为 ``Parent`` 和 ```Child`` 的交集，
        # 相当于 ``Child``.
        assert_type(arg, Child)
    else:
        # ``arg`` 的类型缩小为 ``Parent`` 除外，所以仅剩 ``Unrelated``。
        assert_type(arg, Unrelated)
```

`TypeIs` 内的类型必须与函数参数类型契合，否则静态类型检查器会引发错误。编写不正确的 `TypeIs` 可能导致类型系统中出现不健全行为，以类型安全的方式编写这些函数是用户的责任。

如果 `TypeIs` 函数是一个类或实例方法，那么 `TypeIs` 中的类型将映射到（在 `cls` 或 `self` 之后）第二个形参的类型。

简单来说，`def foo(arg: TypeA) -> TypeIs[TypeB]: ...` 意味着如果 `foo(arg)` 返回 `True`，那么 `arg` 就是 `TypeB` 的实例，如果返回 `False`，它就不是 `TypeB`. 的实例。

`TypeIs` 同样可作用于类型变量，详见 **PEP 742** (使用 `TypeIs` 收窄类型) 。

Added in version 3.13.

typing.**TypeGuard**

特殊类型注解构造，用于标记用户定义的谓词函数。

类型谓词函数是由用户定义的函数，它的返回值指示参数是否为某个特定类型的实例。`TypeGuard` 和*TypeIs* 用法相近，但是对类型检查行为有不同的影响（如下）。

`-> TypeGuard` 告诉静态类型检查器，某函数：

1. 返回一个布尔值。
2. 如果返回值是 `True`，那么其参数的类型是 `TypeGuard` 内的类型。

`TypeGuard` 也适用于类型变量。详情参见 **PEP 647**。

例如：

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # ``val`` 的类型缩小为 ``list[str]``.
        print(" ".join(val))
    else:
        # ``val`` 的类型仍为 ``list[object]``.
        print("Not a list of strings!")
```

TypeIs 和 TypeGuard 有以下不同：

- TypeIs 要求收窄的类型是输入类型的子类型，但 TypeGuard 不要求。这主要是为了允许将 list[object] 缩小为 list[str]，即使后者不是前者的子类型，因为 list 是不变的。
- 当 TypeGuard 函数返回 True 时，类型检查器会将变量的类型精确地收窄到 TypeGuard 类型。当 TypeIs 函数返回 True 时，类型检查程序可以结合先前已知的变量类型和 TypeIs 类型推断出更精确的类型。（从技术上讲，这叫做交类型。）
- 当 TypeGuard 函数返回 False 时，类型检查器不会收窄变量的类型范围。当 TypeIs 函数返回 False 时，类型检查器可以收窄变量的类型范围至排除 TypeIs 类型。

Added in version 3.10.

typing.**Unpack**

在概念上将对象标记为已解包的类型运算符。

例如，在一个*类型变量元组* 上使用解包运算符 * 就等价于使用 Unpack 来将该类型变量元组标记为已被解包：

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# 实际所做的：
tup: tuple[Unpack[Ts]]
```

实际上，Unpack 在*typing.TypeVarTuple* 和*builtins.tuple* 类型的上下文中可以和 * 互换使用。你可能会看到 Unpack 在较旧版本的 Python 中被显式地使用，这时 * 在特定场合则是无法使用的：

```
# 在旧版本的 Python 中，TypeVarTuple 和 Unpack 位于
# `typing_extensions` 反向移植包中。
from typing_extensions import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]          # Python <= 3.10 时的语法错误！
tup: tuple[Unpack[Ts]]   # 语义等效且向后兼容
```

Unpack 也可以与*typing.TypedDict* 一起使用以便在函数签名中对 **kwargs 进行类型标注：

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# 此函数需要两个关键字参数 -
# 类型为 `str` 的 `name` 和类型为 `int` 的 `year`。
def foo(**kwargs: Unpack[Movie]): ...
```

请参阅 **PEP 692** 了解将 Unpack 用于 **kwargs 类型标注的更多细节。

Added in version 3.11.

构造泛型类型与类型别名

下列类不应被直接用作标注。它们的设计目标是作为创建泛型类型和类型别名的构件。

这些对象可通过特殊语法 (类型形参列表和 `type` 语句) 来创建。为了与 Python 3.11 及更早版本的兼容性，它们也可不用专门的语法来创建，如下文所述。

class typing.**Generic**

用于泛型类型的抽象基类。

泛型类型通常是通过在类名后添加一个类型形参列表来声明的:

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
        ...
        # 其他
```

这样的类将隐式地继承自 `Generic`。对于该语法的运行语义的讨论参见 语言参考。

该类的用法如下:

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

此处函数名之后的圆括号是表示 泛型函数。

为了保持向下兼容性，泛型类也可通过显式地继承自 `Generic` 来声明。在此情况下，类型形参必须单独声明:

```
KT = TypeVar('KT')
VT = TypeVar('VT')

class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # 其他
```

class typing.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*, *default=typing.NoDefault*)

类型变量。

构造类型变量的推荐方式是使用针对 泛型函数, 泛型类和 泛型类型别名的专门语法:

```
class Sequence[T]:  # T 是一个 TypeVar
    ...
```

此语法也可被用于创建绑定和约束类型变量:

```
class StrSequence[S: str]:  # S 是具有 `str` 上方绑定的 TypeVar;
    ...                     # 我们可以说 S 是 "被 `str` 绑定"


class StrOrBytesSequence[A: (str, bytes)]:  # A 是约束为 str 或 bytes 的 TypeVar
    ...
```

不过，如有需要，也可通过手动方式来构造可重用的类型变量，就像这样:

```
T = TypeVar('T')  # 可以是任意类型
S = TypeVar('S', bound=str)  # 可以是任意 str 的子类型
A = TypeVar('A', str, bytes)  # 必须是 str 或 bytes
```

类型变量的主要用处是为静态类型检查器提供支持。它们可作为泛型类型以及泛型函数和类型别名定义的形参。请参阅*Generic* 了解有关泛型类型的更多信息。泛型函数的作用方式如下:

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

请注意类型变量可以为 *已绑定*, *已约束*，或两者都不是，但不能同时为已绑定 *并且* 已约束。

类型变量的种类是在其通过 类型形参语法创建时或是在传入 `infer_variance=True` 时由类型检查器推断得到的。手动创建的类型变量可通过传入 `covariant=True` 或 `contravariant=True` 被显式地标记为协变或逆变。在默认情况下，手动创建的类型变量为不变。请参阅 **PEP 484** 和 **PEP 695** 了解更多细节。

已绑定类型变量和已约束类型变量在一些重要的方面具有不同的语义。使用 *已绑定* 类型变量意味着 `TypeVar` 将尽可能使用最专属的类型来解析:

```
x = print_capitalized('a string')
reveal_type(x)  # 显示的类型是 str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # 显示的类型是 StringSubclass

z = print_capitalized(45)  # 错误: int 不是 str 的子类型
```

类型变量的上层绑定可以是一个具体类型、抽象类型（ABC 或 Protocol），甚至是多个类型的联合:

```
# 可以是任何具有 __abs__ 方法的内容
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # 可以是联合 str|bytes 的任何子类型
V = TypeVar('V', bound=SupportsAbs)  # 可以是任何具有 __abs__ 方法的内容
```

但是，如果使用 *约束* 类型变量，则意味着 `TypeVar` 只能被解析为恰好是给定的约束之一:

```
a = concatenate('one', 'two')
reveal_type(a)  # 揭示的类型为 str

b = concatenate(StringSubclass('one'), StringSubclass('two'))
reveal_type(b)  # 揭示的类型为 str, 虽然传入的是 StringSubclass

c = concatenate('one', b'two')  # 错误: 在一个函数调用中类型变量 'A' 可以为 str 或␣
↪bytes, 但不可同时使用
```

在运行时，`isinstance(x, T)` 将引发*TypeError*。

__name__

类型变量的名称。

__covariant__
类型变量是否已被显式地标记为 covariant。

__contravariant__
类型变量是否已被显式地标记为 contravariant。

__infer_variance__
类型变量的种类是否应由类型检查器来推断。

Added in version 3.12.

__bound__
类型变量的上层绑定，如果有的话。

在 3.12 版本发生变更: 对于通过 类型形参语法创建的类型变量，只有在属性被访问的时候才会对绑定求值，而不是在类型变量被创建的时候 (参见 lazy-evaluation)。

__constraints__
一个包含对类型变量的约束的元组，如果有的话。A tuple containing the constraints of the type variable, if any.

在 3.12 版本发生变更: 对于通过 类型形参语法创建的类型变量，只有在属性被访问的时候才会对约束求值，而不是在类型变量被创建的时候 (参见 lazy-evaluation)。

__default__
类型变量的默认值，如果没有默认值，则为*typing.NoDefault*。

Added in version 3.13.

has_default()
返回类型变量是否有默认值。它等价于检查*__default__* 是否为*typing.NoDefault* 单例，但它不要求对 惰性求值的默认值求值。

Added in version 3.13.

在 3.12 版本发生变更: 类型变量现在可以通过使用 **PEP 695** 引入的 类型形参语法来声明。增加了 `infer_variance` 形参。

在 3.13 版本发生变更: 增加了对默认值的支持。

class typing.**TypeVarTuple**(*name*, *, *default=typing.NoDefault*)
类型变量元组。一种启用了 *variadic* 泛型的专属类型变量 形式。

类型变量元组可以通过在 类型形参列表中使用名称前的单个星号 (`*`) 来声明:

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

或者通过显式地唤起 `TypeVarTuple` 构造器:

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

一个普通类型变量将启用单个类型的形参化。作为对比，一个类型变量元组通过将 任意数量的类型变量封包在一个元组中来允许 任意数量类型的形参化。例如:

```
# T 绑定到 int, Ts 绑定到 ()
# 返回值为 (1,), 其类型为 tuple[int]
move_first_element_to_last(tup=(1,))

# T 绑定到 int, Ts 绑定到 (str,)
# 返回值为 ('spam', 1), 其类型为 tuple[str, int]
```

(续下页)

(接上页)

```
move_first_element_to_last(tup=(1, 'spam'))

# T 绑定到 int, Ts 绑定到 (str, float)
# 返回值为 ('spam', 3.0, 1), 其类型为 tuple[str, float, int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# 这不能通过类型检查（并会在运行时执行失败）
# 因为 tuple[()] 与 tuple[T, *Ts] 不兼容
# （至少需要有一个元素）
move_first_element_to_last(tup=())
```

请注意解包运算符 `*` 在 `tuple[T, *Ts]` 中的使用。在概念上，你可以将 `Ts` 当作一个由类型变量组成的元组 `(T1, T2, ...)`。那么 `tuple[T, *Ts]` 就将变为 `tuple[T, *(T1, T2, ...)]`，这等价于 `tuple[T, T1, T2, ...]`。（请注意在旧版本 Python 中，你可能会看到改用 *Unpack* 的写法，如 `Unpack[Ts]`。）

类型变量元组 *总是要被解包*。这有助于区分类型变量元组和普通类型变量:

```
x: Ts          # 不可用
x: tuple[Ts]   # 不可用
x: tuple[*Ts]  # 正确的做法
```

类型变量元组可被用在与普通类型变量相同的上下文中。例如，在类定义、参数和返回类型中:

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

类型变量元组可以很好地与普通类型变量结合在一起:

```
class Array[DType, *Shape]:  # 这样可以
    pass

class Array2[*Shape, DType]:  # 这样也可以
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()      # 完全可以
int_array_2d: Array[int, Height, Width] = Array()  # 是的，同样可以
```

但是，请注意在一个类型参数或类型形参列表中最多只能有一个类型变量元组:

```
x: tuple[*Ts, *Ts]            # 不可用
class Array[*Shape, *Shape]:  # 不可用
    pass
```

最后，一个已解包的类型变量元组可以被用作 `*args` 的类型标注:

```
def call_soon[*Ts](
    callback: Callable[[*Ts], None],
    *args: *Ts
) -> None:
    ...
    callback(*args)
```

相比非解包的 `*args` 标注——例如 `*args: int`，它将指明 *所有参数均为* `int` ——`*args: *Ts` 启用了对 `*args` 中 *单个参数的类型的引用*。在此，这允许我们确保传入 `call_soon` 的 `*args` 的类型与 `callback` 的（位置）参数的类型相匹配。

关于类型变量元组的更多细节，请参见 **PEP 646**。

__name__

类型变量元组的名称。

__default__

类型变量元组的默认值，如果没有默认值，则为*typing.NoDefault*。

Added in version 3.13.

has_default()

返回类型变量元组是否有默认值。它等价于检查*__default__* 是否为*typing.NoDefault* 单例，但它不要求对 惰性求值的默认值求值。

Added in version 3.13.

Added in version 3.11.

在 3.12 版本发生变更: 类型变量元组现在可以使用 **PEP 695** 所引入的 类型形参语法来声明。

在 3.13 版本发生变更: 增加了对默认值的支持。

class typing.**ParamSpec**(*name*, *, *bound=None*, *covariant=False*, *contravariant=False*, *default=typing.NoDefault*)

形参专属变量。*类型变量* 的一个专用版本。

In 类型形参列表，形参规格可以使用两个星号 (`**`) 来声明:

```
type IntFunc[**P] = Callable[P, int]
```

为了保持与 Python 3.11 及更早版本的兼容性，`ParamSpec` 对象也可以这样创建:

```
P = ParamSpec('P')
```

参数规范变量的存在主要是为了使静态类型检查器受益。它们被用来将一个可调用对象的参数类型转发给另一个可调用对象的参数类型——这种模式通常出现在高阶函数和装饰器中。它们只有在 `Concatenate` 中使用时才有效，或者作为 `Callable` 的第一个参数，或者作为用户定义的泛型的参数。参见*Generic* 以了解更多关于泛型的信息。

例如，为了给一个函数添加基本的日志记录，我们可以创建一个装饰器 `add_logging` 来记录函数调用。参数规范变量告诉类型检查器，传入装饰器的可调用对象和由其返回的新可调用对象有相互依赖的类型参数:

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

如果没有 `ParamSpec`，之前标注这个的最简单方式是使用一个*TypeVar* 并附带上层绑定 `Callable[..., Any]`。不过这会导致两个问题:

1. 类型检查器不能对 `inner` 函数进行类型检查，因为 `*args` 和 `**kwargs` 的类型必须是*Any*。
2. *cast()* 在返回 `inner` 函数时，可能需要在 `add_logging` 装饰器的主体中进行，或者必须告诉静态类型检查器忽略 `return inner`。

args

kwargs

由于 ParamSpec 同时捕获了位置参数和关键字参数，P.args 和 P.kwargs 可以用来将 ParamSpec 分割成其组成部分。P.args 代表给定调用中的位置参数的元组，只能用于注释 *args。P.kwargs 代表给定调用中的关键字参数到其值的映射，只能用于注释 **kwargs。在运行时，P.args 和 P.kwargs 分别是*ParamSpecArgs* 和*ParamSpecKwargs* 的实例。

__name__

形参规格的名称。

__default__

形参规格的默认值，如果没有默认值，则为*typing.NoDefault*。

Added in version 3.13.

has_default()

返回形参规格是否有默认值。它等价于检查*__default__* 是否为*typing.NoDefault* 单例，但它不要求对 惰性求值的默认值求值。

Added in version 3.13.

用 covariant=True 或 contravariant=True 创建的参数规范变量可以用来声明协变或逆变泛型类型。参数 bound 也被接受，类似于*TypeVar*。然而这些关键字的实际语义还有待决定。

Added in version 3.10.

在 3.12 版本发生变更: 形参说明现在可以使用 **PEP 695** 所引入的 类型形参语法来声明。

在 3.13 版本发生变更: 增加了对默认值的支持。

备注

只有在全局范围内定义的参数规范变量可以被 pickle。

参见

- **PEP 612** -- 参数规范变量（引入 ParamSpec 和 Concatenate 的 PEP）
- *Concatenate*
- 标注可调用对象

typing.**ParamSpecArgs**

typing.**ParamSpecKwargs**

ParamSpec`的参数和关键字参数属性。``ParamSpec` 的 P.args 属性是 ParamSpecArgs 的一个实例，P.kwargs 是 ParamSpecKwargs 的一个实例。它们的目的是用于运行时内部检查的，对静态类型检查器没有特殊意义。

在这些对象中的任何一个上调用*get_origin()* 将返回原始的 ParamSpec:

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

Added in version 3.10.

class typing.**TypeAliasType**(*name*, *value*, *, *type_params=()*)

通过 type 语句创建的类型别名的类型。

示例:

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

Added in version 3.12.

__name__

类型别名的名称:

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

__module__

类型别名定义所在的模块名称:

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__

类型别名的类型形参，或者如果别名不属于泛型则为一个空元组:

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__

类型别名的值。它将被 惰性求值，因此别名定义中使用的名称将直到 __value__ 属性被访问时才会被解析:

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

其他特殊指令

这些函数和类不应被直接用作标注。它们的设计目标是作为创建和声明类型的构件。

class typing.**NamedTuple**

collections.namedtuple() 的类型版本。

用法:

```
class Employee(NamedTuple):
    name: str
    id: int
```

这相当于：

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

为字段提供默认值，要在类体内赋值：

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

带默认值的字段必须在不带默认值的字段后面。

由此产生的类有一个额外的属性 `__annotations__`，给出一个 dict，将字段名映射到字段类型。(字段名在 `_fields` 属性中，默认值在 `_field_defaults` 属性中，这两者都是*namedtuple()* API 的一部分。)

`NamedTuple` 子类也支持文档字符串与方法：

```
class Employee(NamedTuple):
    """代表一位雇员。"""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` 子类也可以为泛型：

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

反向兼容用法：

```
# 在 Python 3.11 上创建通用 NamedTuple
T = TypeVar("T")

class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# 函数语法也支持
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

在 3.6 版本发生变更: 添加了对 **PEP 526** 中变量注解句法的支持。

在 3.6.1 版本发生变更: 添加了对默认值、方法、文档字符串的支持。

在 3.8 版本发生变更: `_field_types` 和 `__annotations__` 属性现已使用常规字典，不再使用 `OrderedDict` 实例。

在 3.9 版本发生变更: 移除了 `_field_types` 属性，改用具有相同信息，但更标准的 `__annotations__` 属性。

在 3.11 版本发生变更: 添加对泛型命名元组的支持。

Deprecated since version 3.13, will be removed in version 3.15: 创建命名元组 NamedTuple 的关键字参数语法 (`NT = NamedTuple("NT", x=int)`) 未被写入文档且已被弃用，它将在 3.15 中被禁止。使用基于类的语法或函数式语法作为替代。

Deprecated since version 3.13, will be removed in version 3.15: 使用函数式语法创建 NamedTuple 类时，不向'fields' 形参传值 (`NT = NamedTuple("NT")`) 或向'fields' 形参传递 `None` (`NT = NamedTuple("NT",`

None)) 的行为已被弃用，且在 Python 3.15 中都将被禁止。要创建一个无字段的 NamedTuple 类，请使用 `class NT(NamedTuple): pass` 或 `NT = NamedTuple("NT", [])`。

class typing.**NewType**(*name*, *tp*)

用于创建低开销的独有类型 的辅助类。

NewType 将被类型检查器视为一个独有类型。但是，在运行时，调用 NewType 将原样返回其参数。

用法：

```
UserId = NewType('UserId', int)  # 声明 NewType "UserId"
first_user = UserId(1)  # 在运行时 "UserId" 将原样返回参数
```

__module__

新类型定义所在的模块。

__name__

新类型的名称。

__supertype__

新类型所基于的类型。

Added in version 3.5.2.

在 3.10 版本发生变更: NewType 现在是一个类而不是函数。

class typing.**Protocol**(*Generic*)

协议类的基类。

协议类是这样定义的:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

这些类主要与静态类型检查器搭配使用，用来识别结构子类型（静态鸭子类型），例如：

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # 通过静态类型检查
```

请参阅 **PEP 544** 了解详情。使用*runtime_checkable()* 装饰的协议类（稍后介绍）将被用作只检查给定属性是否存在，而忽略其类型签名的简单运行时协议。没有此装饰器的协议类不可被用作传给*isinstance()* 或*issubclass()* 的第二个参数。

Protocol 类可以是泛型，例如：

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

在需要兼容 Python 3.11 或更早版本的代码中，可以这样编写泛型协议:

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Added in version 3.8.

@typing.**runtime_checkable**

用于把 Protocol 类标记为运行时协议。

这样的协议可以配合*isinstance()* 和*issubclass()* 使用。这允许简单的结构化检查，非常类似于*collections.abc* 中*Iterable* 这样特别擅长做一件事的对象。例如:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

当应用于非协议类时此装饰器将引发*TypeError*。

备注

runtime_checkable() 将只检查所需方法或属性是否存在，而不检查它们的类型签名或类型。例如，*ssl.SSLObject* 是一个类，因此它通过了针对*Callable* 的*issubclass()* 检查。但是，ssl.SSLObject.__init__ 方法的存在只是引发*TypeError* 并附带更具信息量的消息，因此它无法调用 (实例化) *ssl.SSLObject*。

备注

针对运行时可检查协议的*isinstance()* 检查相比针对非协议类的 isinstance() 检查可能会惊人的缓慢。请考虑在性能敏感的代码中使用替代性写法如*hasattr()* 调用进行结构检查。

Added in version 3.8.

在 3.12 版本发生变更: 现在*isinstance()* 的内部实现对于运行时可检查协议的检查会使用*inspect.getattr_static()* 来查找属性 (在之前版本中，会使用*hasattr()*)。因此，在 Python 3.12+ 上一些以前被认为是运行时可检查协议的实例的对象可能不再被认为是该协议的实例，反之亦反。大多数用户不太可能受到这一变化的影响。

在 3.12 版本发生变更: 一旦类被创建则运行时可检查协议的成员就会被视为在运行时 “已冻结” 。在运行时可检查协议上打上猴子补丁属性仍然有效，但不会影响将对象与协议进行比较的*isinstance()* 检查。请参阅 ”Python 3.12 有什么新变化了解更多细节。

class typing.**TypedDict**(*dict*)

把类型提示添加至字典的特殊构造器。在运行时，它是纯*dict*。

TypedDict 声明一个字典类型，该类型预期所有实例都具有一组键集，其中，每个键都与对应类型的值关联。运行时不检查此预期，而是由类型检查器强制执行。用法如下:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # 可以
b: Point2D = {'z': 3, 'label': 'bad'}           # 不能通过类型检查

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

另一种创建 TypedDict 的方法是使用函数调用语法。第二个参数必须是一个*dict* 字面值:

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

这种函数式语法允许定义不是合法 标识符的键，例如关键字或包含连字符，或者当键名不可被 并入像是常规私有名称等:

```
# 引发 SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' 是关键字
    x-y: int  # 名称中有连字符

class Definition(TypedDict):
    __schema: str  # 并入 `_Definition__schema`

# 可以，函数式语法
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
Definition = TypedDict('Definition', {'__schema': str})  # 未并入
```

默认情况下，所有的键都必须出现在一个 TypedDict 中。可以使用*NotRequired* 将单独的键标记为非必要的:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# 替代语法
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': NotRequired[str]})
```

这意味着一个 Point2D TypedDict 可以省略 label 键。

也可以通过全部指定 False 将所有键都标记为默认非必要的:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# 替代语法
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

这意味着一个 Point2D TypedDict 可以省略任何一个键。类型检查器只需要支持一个字面的 False 或 True 作为 total 参数的值。True 是默认的，它使类主体中定义的所有项目都是必需的。

一个 total=False TypedDict 中单独的键可以使用*Required* 标记为必要的:

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# 替代语法
Point2D = TypedDict('Point2D', {
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

一个 TypedDict 类型有可能使用基于类的语法从一个或多个其他 TypedDict 类型继承。用法:

```
class Point3D(Point2D):
    z: int
```

Point3D 有三个项目: x , y 和 z 。其等价于定义:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

TypedDict 不能从非 TypedDict 类继承，除了*Generic*。例如:

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # 非 TypedDict 类

class XY(X, Y): pass  # 可以

class XZ(X, Z): pass  # 引发 TypeError
```

TypedDict 也可以为泛型的:

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

要创建与 Python 3.11 或更低版本兼容的泛型 TypedDict，请显式地从*Generic* 继承:

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

TypedDict 可以通过注解字典（参见 annotations-howto 了解更多关于注解的最佳实践）、*__total__*、*__required_keys__* 和*__optional_keys__* 进行内省。

__total__

Point2D.__total__ 给出了 total 参数的值。例如:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

该属性只反映传给当前 TypedDict 类的 total 参数的值，而不反映这个类在语义上是否完整。例如，一个 __total__ 被设为 True 的 TypedDict 可能有用*NotRequired* 标记的键，或者它可能继承自另一个设置了 total=False 的 TypedDict。因此，使用*__required_keys__* 和*__optional_keys__* 进行内省通常会更好。

__required_keys__

Added in version 3.9.

__optional_keys__

Point2D.__required_keys__ 和 Point2D.__optional_keys__ 返回分别包含必要的和非必要的键的*frozenset* 对象。

标记为*Required*的键总是会出现在 `__required_keys__` 中而标记为*NotRequired*的键总是会出现在 `__optional_keys__` 中。

为了向下兼容 Python 3.10 及更老的版本，还可以使用继承机制在同一个 `TypedDict` 中同时声明必要和非必要的键。这是通过声明一个具有 `total` 参数值的 `TypedDict` 然后在另一个 `TypedDict` 中继承它并使用不同的 `total` 值来实现的：

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

Added in version 3.9.

> **备注**
>
> 如果使用了 `from __future__ import annotations` 或者如果以字符串形式给出标注，那么标注不会在定义 `TypedDict` 时被求值。因此，`__required_keys__` 和 `__optional_keys__` 所依赖的运行时内省可能无法正常工作，这些属性的值也可能不正确。

对*ReadOnly*的支持反映在下列属性中：

`__readonly_keys__`

一个包含所有只读键名称的*frozenset*。带有*ReadOnly*限定符的键被认为是只读的。

Added in version 3.13.

`__mutable_keys__`

一个包含所有可变键名称的*frozenset*。不带有*ReadOnly*限定符的键被认为是可变的。

Added in version 3.13.

更多示例与 `TypedDict` 的详细规则，详见 **PEP 589**。

Added in version 3.8.

在 3.11 版本发生变更: 增加了对将单独的键标记为*Required*或*NotRequired*的支持。参见 **PEP 655**。

在 3.11 版本发生变更: 添加对泛型 `TypedDict` 的支持。

在 3.13 版本发生变更: 移除了对使用关键字参数方法创建 `TypedDict` 的支持。

在 3.13 版本发生变更: 添加了对*ReadOnly*限定符的支持。

Deprecated since version 3.13, will be removed in version 3.15: 使用函数式语法创建 TypedDict 类时，不向'fields' 形参传值 (`TD = TypedDict("TD")`) 或向'fields' 形参传递 `None` (`TD = TypedDict ("TD", None)`) 的行为已被弃用，且在 Python 3.15 中都将被禁止。要创建一个无字段的 TypedDict 类，请使用 `class TD(TypedDict ): pass` 或 `TD = TypedDict ("TD", {})`。

协议

下列协议由 typing 模块提供并已全被装饰为*可在运行时检查的*。

class `typing.`**SupportsAbs**

一个抽象基类，含一个抽象方法 `__abs__`，该方法与其返回类型协变。

class typing.**SupportsBytes**

一个抽象基类，含一个抽象方法 __bytes__。

class typing.**SupportsComplex**

一个抽象基类，含一个抽象方法 __complex__。

class typing.**SupportsFloat**

一个抽象基类，含一个抽象方法 __float__。

class typing.**SupportsIndex**

一个抽象基类，含一个抽象方法 __index__。

Added in version 3.8.

class typing.**SupportsInt**

一个抽象基类，含一个抽象方法 __int__。

class typing.**SupportsRound**

一个抽象基类，含一个抽象方法 __round__，该方法与其返回类型协变。

与 IO 相关的抽象基类

class typing.**IO**

class typing.**TextIO**

class typing.**BinaryIO**

泛型 IO[AnyStr] 及其子类 TextIO(IO[str])、BinaryIO(IO[bytes]) 表示 I/O 流——例如*open()* 返回的对象——的类型。

函数与装饰器

typing.**cast**(*typ*, *val*)

把一个值转换为指定的类型。

这会把值原样返回。对类型检查器而言这代表了返回值具有指定的类型，在运行时我们故意没有设计任何检查（我们希望让这尽量快）。

typing.**assert_type**(*val*, *typ*, */*)

让静态类型检查器确认 *val* 具有推断为 *typ* 的类型。

在运行时这将不做任何事：它会原样返回第一个参数而没有任何检查或附带影响，无论参数的实际类型是什么。

当静态类型检查器遇到对 assert_type() 的调用时，如果该值不是指定的类型则会报错:

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, 推断 `name` 的类型是 `str`
    assert_type(name, int)  # 类型检查器错误
```

此函数适用于确保类型检查器对脚本的理解符合开发者的意图:

```
def complex_function(arg: object):
    # 执行某些复杂的类型细化逻辑，
    # 在此之后我们希望推断出的类型为 `int`
    ...
    # 测试类型检查器能否正确理解我们的函数
    assert_type(arg, int)
```

Added in version 3.11.

typing.**assert_never**(*arg*, */*)

让静态类型检查器确认一行代码是不可达的。

示例：

```
def int_or_str(arg: int | str) -> None:
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

在这里，标注允许类型检查器推断最后一种情况永远不会执行，因为 arg 要么是*int* 要么是*str*，而这两种选项都已被之前的情况覆盖了。

如果类型检查器发现对 assert_never() 的调用是可达的，它将报告一个错误。举例来说，如果 arg 的类型标注改为 int | str | float，则类型检查器将报告一个错误指出 unreachable 为*float* 类型。对于通过类型检查的 assert_never 调用，参数传入的推断类型必须为兜底类型*Never*，而不能为任何其他类型。

在运行时，如果调用此函数将抛出一个异常。

参见

Unreachable Code and Exhaustiveness Checking 提供了更多有关表达类型穷举检查的信息。

Added in version 3.11.

typing.**reveal_type**(*obj*, /)

让静态类型检查器显示推测的表达式类型。

当静态类型检查器遇到一个对此函数的调用时，它将发出带有所推测参数类型的诊断信息。例如:

```
x: int = 1
reveal_type(x)  # 揭示的类型为 "builtins.int"
```

这在你想要调试你的类型检查器如何处理一段特定代码时很有用处。

在运行时，此函数会将其参数类型打印到*sys.stderr* 并不加修改地返回该参数 (以允许该调用在表达式中使用):

```
x = reveal_type(1)  # 打印 "Runtime type is int"
print(x)  # 打印 "1"
```

请注意在运行时类型可能不同于类型静态检查器所推测的类型（明确程度可能更高也可能更低）。

大多数类型检查器都能在任何地方支持 reveal_type()，即使并未从 typing 导入该名称。不过，从 typing 导入该名称将允许你的代码在运行时不会出现运行时错误并能更清晰地传递意图。

Added in version 3.11.

@typing.**dataclass_transform**(*, *eq_default=True*, *order_default=False*, *kw_only_default=False*, *frozen_default=False*, *field_specifiers=()*, ***kwargs*)

将一个对象标记为提供类似*dataclass* 行为的装饰器。

dataclass_transform 可被用于装饰类、元类或本身为装饰器的函数。使用 @dataclass_transform() 将让静态类型检查器知道被装饰的对象会执行以类似*@dataclasses.dataclass* 的方式来转换类的运行时“魔法”。

装饰器函数使用方式的例子:

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls
```

(续下页)

(接上页)

```
@create_model
class CustomerModel:
    id: int
    name: str
```

在基类上:

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

在元类上:

```
@dataclass_transform()
class ModelMeta(type): ...

class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

上面定义的 `CustomerModel` 类将被类型检查器视为类似于使用*@dataclasses.dataclass* 创建的类。例如，类型检查器将假定这些类具有接受 `id` 和 `name` 的 `__init__` 方法。

被装饰的类、元类或函数可以接受以下布尔值参数，类型检查器将假定它们具有与*@dataclasses.dataclass* 装饰器相同的效果: `init`, `eq`, `order`, `unsafe_hash`, `frozen`, `match_args`, `kw_only` 和 `slots`。这些参数的值 (`True` 或 `False`) 必须可以被静态地求值。

传给 `dataclass_transform` 装饰器的参数可以被用来定制被装饰的类、元类或函数的默认行为:

参数

- **eq_default** (`bool`) -- 指明如果调用方省略了 `eq` 形参则应将其假定为 `True` 还是 `False`。默认为 `True`。
- **order_default** (`bool`) -- 指明如果调用方省略了 `order` 形参则应将其假定为 `True` 还是 `False`。默认为 `False`。
- **kw_only_default** (`bool`) -- 指明如果调用方省略了 `kw_only` 形参则应将其假定为 `True` 还是 `False`。默认为 `False`。
- **frozen_default** (`bool`) -- 指明如果调用方省略了 `frozen` 形参则应将其假定为 `True` 还是 `False`。默认为 `False`。.. versionadded:: 3.12
- **field_specifiers** (`tuple`*[Callable[..., Any], ...]*) -- 指定一个受支持的类或描述字段的函数的静态列表，类似于*dataclasses.field()*。默认为 `()`。
- ****kwargs** (*Any*) -- 接受任何其他关键字以便允许可能的未来扩展。

类型检查器能识别下列字段设定器的可选形参:

表 1: 字段设定器的可识别形参

形参名称	描述
init	指明字段是否应当被包括在合成的 __init__ 方法中。如果未指明，则 init 默认为 True。
default	为字段提供默认值。
default_factory	提供一个返回字段默认值的运行时回调。如果 default 或 default_factory 均未指定，则会假定字段没有默认值而在类被实例化时必须提供一个值。
factory	字段说明符上 default_factory 形参的别名。
kw_only	指明字段是否应被标记为仅限关键字的。如为 True，字段将是仅限关键字的。如为 False，它将不是仅限关键字的。如未指明，则将使用以 dataclass_transform 装饰的对象的 kw_only 形参的值，或者如果该值也未指明，则将使用 dataclass_transform 上 kw_only_default 的值。
alias	提供字段的替代名称。该替代名称会被用于合成的 __init__ 方法。

在运行时，该装饰器会将其参数记录到被装饰对象的 __dataclass_transform__ 属性。它没有其他的运行时影响。

更多细节请参见 **PEP 681**。

Added in version 3.11.

@typing.**overload**

用于创建重载函数和方法的装饰器。

@overload 装饰器允许描述支持多参数类型不同组合的函数和方法。一系列以 @overload 装饰的定义必须带上恰好一个非 @overload 装饰的定义（用于同一个函数/方法）。

以 @overload 装饰的定义仅对类型检查器有用，因为它们将被非 @overload 装饰的定义覆盖。与此同时，非 @overload 装饰的定义将在运行时使用但应被类型检查器忽略。在运行时，直接调用以 @overload 装饰的函数将引发*NotImplementedError*。

一个提供了比使用联合或类型变量更精确的类型的重载的示例：

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # 以下为真正的实现
```

请参阅 **PEP 484** 了解更多细节以及与其他类型语义的比较。

在 3.11 版本发生变更: 现在可以使用*get_overloads()* 在运行时内省有重载的函数。

typing.**get_overloads**(*func*)

为 *func* 返回以*@overload* 装饰的定义的序列。

func 是用于实现过载函数的函数对象。例如，根据文档中为*@overload* 给出的 process 定义，get_overloads(process) 将为所定义的三个过载函数返回由三个函数对象组成的序列。如果在不带过载的函数上调用，get_overloads() 将返回一个空序列。

get_overloads() 可被用来在运行时内省一个过载函数。

Added in version 3.11.

typing.**clear_overloads**()

清空内部注册表中所有已注册的重载。

这可用于回收注册表所使用的内存。

Added in version 3.11.

@typing.**final**

表示最终化方法和最终化类的装饰器。

以 @final 装饰一个方法将向类型检查器指明该方法不可在子类中被重载。以 @final 装饰一个类表示它不可被子类化。

例如:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # 类型检查器报告错误
        ...

@final
class Leaf:
    ...
class Other(Leaf):  # 类型检查器报告错误
    ...
```

这些属性没有运行时检查。详见 **PEP 591**。

Added in version 3.8.

在 3.11 版本发生变更: 该装饰器现在将尝试在被装饰的对象上设置 __final__ 属性为 True。这样，可以在运行时使用 if getattr(obj, "__final__", False) 这样的检查来确定对象 obj 是否已被标记为终结。如果被装饰的对象不支持设置属性，该装饰器将不加修改地返回对象而不会引发异常。

@typing.**no_type_check**

标明注解不是类型提示的装饰器。

此作用方式类似于类或函数的*decorator*。对于类，它将递归地应用到该类中定义的所有方法和类（但不包括在其超类或子类中定义的方法）。类型检查器将忽略带有此装饰器的函数或类的所有标注。

@no_type_check 将原地改变被装饰的对象。

@typing.**no_type_check_decorator**

让其他装饰器具有*no_type_check()* 效果的装饰器。

本装饰器用*no_type_check()* 里的装饰函数打包其他装饰器。

Deprecated since version 3.13, will be removed in version 3.15: @no_type_check_decorator 没有任何类型检查器支持过，所以它被弃用，并将在 Python 3.15 中被移除。

@typing.**override**

该装饰器指明子类中的某个方法是重载超类中的方法或属性。

如果一个以 @override 装饰的方法实际未重载任何东西则类型检查器应当报告错误。这有助于防止当基类发生修改而子类未进行相应修改而导致的问题。

例如:

```
class Base:
    def log_status(self) -> None:
        ...

class Sub(Base):
    @override
```

(续下页)

(接上页)

```
    def log_status(self) -> None:  # 可以：重写 Base.log_status
        ...

    @override
    def done(self) -> None:  # 类型检查器报告错误
        ...
```

没有对此特征属性的运行时检查。

该装饰器将尝试在被装饰的对象上设置 `__override__` 属性为 `True`。这样，可以在运行时使用 `if getattr(obj, "__override__", False)` 这样的检查来确定对象 `obj` 是否已被标记为重载。如果被装饰的对象不支持设置属性，该装饰器将不加修改地返回对象而不会引发异常。

更多细节参见 **PEP 698**。

Added in version 3.12.

`@typing.type_check_only`

将类或函数标记为在运行时不可用的装饰器。

在运行时，该装饰器本身不可用。实现返回的是私有类实例时，它主要是用于标记在类型存根文件中定义的类。

```
@type_check_only
class Response:  # 私有或在运行时不可用
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

注意，建议不要返回私有类实例，最好将之设为公共类。

内省辅助器

`typing.get_type_hints`(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)

返回函数、方法、模块、类对象的类型提示的字典。

该函数通常与 `obj.__annotations__` 相同，但会对注解字典进行以下更改：

- 以字符串字面形式或*ForwardRef* 对象编码的前向引用会在 *globalns*, *localns* 和 (如适用) *obj* 的类型形参命名空间中求值。如果没有传入 *globalns* 或 *localns*，则会从 *obj* 中推断出适当的命名空间字典。
- `None` 被替换为*types.NoneType*。
- 如果在 *obj* 上应用了*@no_type_check*，返回一个空字典。
- 如果 *obj* 是一个类 `C`，则函数返回一个合并了 `C` 的基类与 `C` 上的基类的字典。这是通过遍历 `C.__mro__` 并迭代地合并 `__annotations__` 字典来实现的。在*method resolution order* 中较早出现的类上的标注总是会优先于在方法解析顺序中较晚出现的类的标注。
- 除非 *include_extras* 设置为 `True`，否则函数会递归地将所有出现的 `Annotated[T, ...]` 替换为 `T` (详见*Annotated*)。

另请参阅*inspect.get_annotations()*，这是一个以更直接方式返回注解的低级函数。

> **备注**
>
> 如果 *obj* 注解中的任何前向引用不可解析或不是有效的 Python 代码，此函数将引发*NameError* 等异常。例如导入的*类型别名* 包含正向引用，或名称在*if TYPE_CHECKING* 下导入。

在 3.9 版本发生变更: 增加了作为 **PEP 593** 组成部分的 `include_extras` 形参。请参阅*Annotated* 文档了解详情。

在 3.11 版本发生变更: 在之前，如果设置了等于 None 的默认值则会为函数和方法标注添加 Optional[t]。现在标注将被不加修改地返回。

typing.**get_origin**(*tp*)

获取一个类型的不带下标的版本：对于 X[Y, Z, ...] 形式的类型对象将返回 X。

如果 X 是一个内置类型或*collections* 类在 typing 模块中的别名，它将被正规化为原始的类。如果 X 是*ParamSpecArgs* 或*ParamSpecKwargs* 的实例，则返回下层的*ParamSpec*。对于不受支持的对象将返回 None。

示例：

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
assert get_origin(Annotated[str, "metadata"]) is Annotated
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

Added in version 3.8.

typing.**get_args**(*tp*)

获取已执行所有下标的类型参数：对于 X[Y, Z, ...] 形式的类型对象将返回 (Y, Z, ...)。

如果 X 是一个并集或是包含在另一个泛型类型中的*Literal*，则 (Y, Z, ...) 的顺序可能因类型缓存而与原始参数 [Y, Z, ...] 存在差异。对于不受支持的对象将返回 ()。

示例：

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

Added in version 3.8.

typing.**get_protocol_members**(*tp*)

返回*Protocol* 中定义的成员构成的集合。

```
>>> from typing import Protocol, get_protocol_members
>>> class P(Protocol):
...     def a(self) -> str: ...
...     b: int
>>> get_protocol_members(P) == frozenset({'a', 'b'})
True
```

如果参数不是协议，引发*TypeError*。

Added in version 3.13.

typing.**is_protocol**(*tp*)

检查一个类型是否为*Protocol*。

例如：

```
class P(Protocol):
    def a(self) -> str: ...
    b: int

is_protocol(P)    # => True
is_protocol(int)  # => False
```

Added in version 3.13.

typing.**is_typeddict**(*tp*)

检查一个类型是否为*TypedDict*。

例如:

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict 是创建类型化字典的工厂,
# 不是类型化字典
assert not is_typeddict(TypedDict)
```

Added in version 3.10.

class typing.**ForwardRef**

用于字符串前向引用的内部类型表示的类。

例如，`List["SomeClass"]` 会被隐式转换为 `List[ForwardRef("SomeClass")]`。`ForwardRef` 不应由用户来实例化，但可以由内省工具使用。

> **备注**
>
> PEP 585 泛型类型例如 `list["SomeClass"]` 将不会被隐式地转换为 `list[ForwardRef("SomeClass")]` 因而将不会自动解析为 `list[SomeClass]`。

Added in version 3.7.4.

typing.**NoDefault**

一个用于指示类型形参没有默认值的哨兵对象。例如:

```
>>> T = TypeVar("T")
>>> T.__default__ is typing.NoDefault
True
>>> S = TypeVar("S", default=None)
>>> S.__default__ is None
True
```

Added in version 3.13.

常量

typing.**TYPE_CHECKING**

会被第 3 方静态类型检查器假定为 `True` 的特殊常量。在运行时将为 `False`。

用法:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

第一个类型注解必须用引号标注，才能把它当作“前向引用”，从而在解释器运行时中隐藏 `expensive_mod` 引用。局部变量的类型注释不会被评估，因此，第二个注解不需要用引号引起来。

> **备注**
>
> 若用了 `from __future__ import annotations`，函数定义时则不求值注解，直接把注解以字符串形式存在 `__annotations__` 里。这时毋需为注解打引号（见 **PEP 563**）。

Added in version 3.5.2.

一些已被弃用的别名

本模块给标准库中已有的类定义了许多别名，这些别名现已不再建议使用。起初 typing 模块包含这些别名是为了支持用 `[]` 来参数化泛型类。然而，在 Python 3.9 中，对应的已有的类也支持了 `[]` (参见 **PEP 585**)，因此这些别名了就成了多余的了。

这些多余的类型从 Python 3.9 起被弃用。然而，虽然它们可能会在某一时刻被移除，但目前还没有移除它们的计划。因此，解释器目前不会对这些别名发出弃用警告。

一旦确定了何时这些别名将被移除，解释器将比正式移除之时提前至少两个版本发出弃用警告 (deprecation warning)。但保证至少在 Python 3.14 之前，这些别名仍会留在 typing 模块中，并且不会引发弃用警告。

如果被类型检查器检查的程序旨在运行于 Python 3.9 或更高版本，则鼓励类型检查器标记出这些不建议使用的类型。

内置类型的别名

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)

dict 的已弃用的别名。

请注意，注释参数时，最好使用抽象的多项集类型，如*Mapping*，而不是使用*dict* 或 `typing.Dict`。

自 3.9 版本弃用: *builtins.dict* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**List**(*list, MutableSequence[T]*)

list 的已弃用的别名。

请注意，注释参数时，最好使用抽象的多项集类型，如*Sequence* 或*Iterable*，而不是使用*list* 或 `typing.List`。

自 3.9 版本弃用: *builtins.list* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Set**(*set, MutableSet[T]*)

builtins.set 的已弃用的别名。

请注意，注释参数时，最好使用抽象的多项集类型，如*collections.abc.Set*，而不是使用*set* 或*typing.Set*。

自 3.9 版本弃用: *builtins.set* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)

builtins.frozenset 的已弃用的别名。

自 3.9 版本弃用: *builtins.frozenset* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

typing.**Tuple**

tuple 的已弃用的别名。

tuple 和 `Tuple` 是类型系统中的特例；更多详细信息请参见标注元组。

自 3.9 版本弃用: *builtins.tuple* 现在支持下标操作（`[]`）。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Type**(*Generic[CT_co]*)

type 的已弃用的别名。

有关在类型注解中使用*type* 或 `typing.Type` 的详细信息，请参阅类对象的类型 。

Added in version 3.5.2.

自 3.9 版本弃用: *builtins.type* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

collections 中的类型的别名。

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)

collections.defaultdict 的已弃用的别名。

Added in version 3.5.2.

自 3.9 版本弃用: *collections.defaultdict* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)

collections.OrderedDict 的已弃用的别名。

Added in version 3.7.2.

自 3.9 版本弃用: *collections.OrderedDict* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)

collections.ChainMap 的已弃用的别名。

Added in version 3.6.1.

自 3.9 版本弃用: *collections.ChainMap* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Counter**(*collections.Counter, Dict[T, int]*)

collections.Counter 的已弃用的别名。

Added in version 3.6.1.

自 3.9 版本弃用: *collections.Counter* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Deque**(*deque, MutableSequence[T]*)

collections.deque 的已弃用的别名。

Added in version 3.6.1.

自 3.9 版本弃用: *collections.deque* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

其他具体类型的别名

class typing.**Pattern**

class typing.**Match**

re.compile() 和*re.match()* 的返回类型的已弃用的别名。

这些类型（与对应的函数）是*AnyStr* 上的泛型。`Pattern` 可以被特化为 `Pattern[str]` 或 `Pattern[bytes]`；`Match` 可以被特化为 `Match[str]` 或 `Match[bytes]`。

自 3.9 版本弃用: *re* 模块中的 `Pattern` 与 `Match` 类现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**Text**

str 的已弃用的别名。

`Text` 被用来为 Python 2 代码提供向上兼容的路径：在 Python 2 中，`Text` 是 `unicode` 的别名。

使用 `Text` 时，值中必须包含 unicode 字符串，以兼容 Python 2 和 Python 3：

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Added in version 3.5.2.

自 3.11 版本弃用: Python 2 已不再受支持，并且大部分类型检查器也都不再支持 Python 2 代码的类型检查。目前还没有计划移除该别名，但建议用户使用*str* 来代替 `Text`。

collections.abc 中容器 ABC 的别名

class typing.**AbstractSet**(*Collection[T_co]*)

collections.abc.Set 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.Set* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**ByteString**(*Sequence[int]*)

该类型代表了*bytes*、*bytearray*、*memoryview* 等字节序列类型。

Deprecated since version 3.9, will be removed in version 3.14: 首选*collections.abc.Buffer*，或是 `bytes | bytearray | memoryview` 这样的并集。

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)

collections.abc.Collection 的已弃用的别名。

Added in version 3.6.

自 3.9 版本弃用: *collections.abc.Collection* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**Container**(*Generic[T_co]*)

collections.abc.Container 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.Container* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**ItemsView**(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)

collections.abc.ItemsView 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.ItemsView* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**KeysView**(*MappingView, AbstractSet[KT_co]*)

collections.abc.KeysView 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.KeysView* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**Mapping**(*Collection[KT], Generic[KT, VT_co]*)

collections.abc.Mapping 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.Mapping* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**MappingView**(*Sized*)

collections.abc.MappingView 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.MappingView* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**MutableMapping**(*Mapping[KT, VT]*)

collections.abc.MutableMapping 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.MutableMapping* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**MutableSequence**(*Sequence[T]*)

collections.abc.MutableSequence 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.MutableSequence* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**MutableSet**(*AbstractSet[T]*)

collections.abc.MutableSet 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.MutableSet* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)

collections.abc.Sequence 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.Sequence* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**ValuesView**(*MappingView, Collection[_VT_co]*)

collections.abc.ValuesView 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.ValuesView* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

collections.abc 中异步 ABC 的别名

class typing.**Coroutine**(*Awaitable[ReturnType], Generic[YieldType, SendType, ReturnType]*)

collections.abc.Coroutine 的已弃用的别名。

有关在注解类型中使用*collections.abc.Coroutine* 和 `typing.Coroutine` 的详细信息，请参见标注生成器和协程。

Added in version 3.5.3.

自 3.9 版本弃用: *collections.abc.Coroutine* 现在支持下标操作（`[]`）。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**AsyncGenerator**(*AsyncIterator[YieldType], Generic[YieldType, SendType]*)

collections.abc.AsyncGenerator 的已弃用的别名。

有关在注解类型中使用*collections.abc.AsyncGenerator* 和 `typing.AsyncGenerator` 的详细信息，请参见标注生成器和协程。

Added in version 3.6.1.

自 3.9 版本弃用: *collections.abc.AsyncGenerator* 现在支持下标操作（`[]`）。参见 **PEP 585** 和*GenericAlias* 类型。

在 3.13 版本发生变更: `SendType` 形参现在有默认值。

class typing.**AsyncIterable**(*Generic[T_co]*)

collections.abc.AsyncIterable 的已弃用的别名。

Added in version 3.5.2.

自 3.9 版本弃用: *collections.abc.AsyncIterable* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)

collections.abc.AsyncIterator 的已弃用的别名。

Added in version 3.5.2.

自 3.9 版本弃用: *collections.abc.AsyncIterator* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**Awaitable**(*Generic[T_co]*)

collections.abc.Awaitable 的已弃用的别名。

Added in version 3.5.2.

自 3.9 版本弃用: *collections.abc.Awaitable* 现在支持下标操作 ([])。参见 **PEP 585** 和*GenericAlias* 类型。

collections.abc 中其他 ABC 的别名

class typing.**Iterable**(*Generic[T_co]*)

collections.abc.Iterable 的已弃用的别名

自 3.9 版本弃用: *collections.abc.Iterable* 现在支持下标操作 ([])。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**Iterator**(*Iterable[T_co]*)

collections.abc.Iterator 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.Iterator* 现在支持下标操作 ([])。参见 **PEP 585** 和*GenericAlias* 类型。

typing.**Callable**

collections.abc.Callable 的已弃用的别名。

有关如何在类型标注中使用*collections.abc.Callable* 和 typing.Callable 的详细信息请参阅标注可调用对象。

自 3.9 版本弃用: *collections.abc.Callable* 现在支持下标操作 ([])。参见 **PEP 585** 和*GenericAlias* 类型。

在 3.10 版本发生变更: Callable 现在支持*ParamSpec* 和*Concatenate*。详情见 **PEP 612**。

class typing.**Generator**(*Iterator[YieldType], Generic[YieldType, SendType, ReturnType]*)

collections.abc.Generator 的已弃用的别名。

有关在注解类型中使用*collections.abc.Generator* 和 typing.Generator 的详细信息，请参见标注生成器和协程。

自 3.9 版本弃用: *collections.abc.Generator* 现在支持下标操作 ([])。参见 **PEP 585** 和*GenericAlias* 类型。

在 3.13 版本发生变更: 添加了发送和返回类型的默认值。

class typing.**Hashable**

collections.abc.Hashable 的已弃用的别名。

自 3.12 版本弃用: 请改为直接使用*collections.abc.Hashable*。

class typing.**Reversible**(*Iterable[T_co]*)

collections.abc.Reversible 的已弃用的别名。

自 3.9 版本弃用: *collections.abc.Reversible* 现在支持下标操作 ([])。参见 **PEP 585** 和*GenericAlias* 类型。

class typing.**Sized**

collections.abc.Sized 的已弃用的别名。

自 3.12 版本弃用: 请改为直接使用*collections.abc.Sized*。

contextlib ABC 的别名

class typing.**ContextManager**(*Generic[T_co, ExitT_co]*)

contextlib.AbstractContextManager 的已弃用的别名。

第一个类型形参 `T_co` 表示 `__enter__()` 方法返回值的类型。可选的第二个类型形参 `ExitT_co` 默认为 `bool | None`，它表示 `__exit__()` 方法返回的类型。

Added in version 3.5.4.

自 3.9 版本弃用: *contextlib.AbstractContextManager* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

在 3.13 版本发生变更: 添加了可选的第二个类型形参，`ExitT_co`。

class typing.**AsyncContextManager**(*Generic[T_co, AExitT_co]*)

contextlib.AbstractAsyncContextManager 的已弃用的别名。

第一个类型形参 `T_co` 表示 `__aenter__()` 方法返回值的类型。可选的第二个类型形参 `AExitT_co` 默认为 `bool | None`，它表示 `__aexit__()` 方法返回的类型。

Added in version 3.6.2.

自 3.9 版本弃用: *contextlib.AbstractAsyncContextManager* 现在支持下标操作 (`[]`)。参见 **PEP 585** 和*GenericAlias 类型*。

在 3.13 版本发生变更: 添加了可选的第二个类型形参，`AExitT_co`。

27.1.13 主要特性的弃用时间线

`typing` 的某些特性被弃用，并且可能在将来的 Python 版本中被移除。下表总结了主要的弃用特性。该表可能会被更改，而且并没有列出所有的弃用特性。

特性	弃用于	计划移除	PEP/问题
标准容器的 `typing` 版本	3.9	未定（请参阅*一些已被弃用的别名* 了解详情）	**PEP 585**
typing.ByteString	3.9	3.14	gh-91896
typing.Text	3.11	未确定	gh-92332
typing.Hashable 和 *typing.Sized*	3.12	未确定	gh-94309
typing.TypeAlias	3.12	未确定	**PEP 695**
@typing.no_type_check_decorator	3.13	3.15	gh-106309
typing.AnyStr	3.13	3.18	gh-105578

27.2 pydoc --- 文档生成器和在线帮助系统

源代码: Lib/pydoc.py

`pydoc` 模块会根据 Python 模块自动生成文档。生成的文档可在控制台中显示为文本页面，提供给 Web 浏览器或者保存为 HTML 文件。

对于模块、类、函数和方法，显示的文档内容取自对象的文档字符串（即 the *__doc__* 属性），并会递归地从其带有文档的成员中获取。如果没有文档字符串，则 `pydoc` 会尝试从源文件中类、函数或方法的定义上方，或者模块最上方的注释行代码块获取描述文本（参见*inspect.getcomments()*。）

内置函数*help()* 会唤起交互式解释器的在线帮助系统，该系统使用 `pydoc` 在控制台上生成文本形式的文档内容。同样的文本文档也可以在 Python 解释器以外通过在操作系统的命令提示符中以脚本方式运行 **pydoc** 来查看。例如，运行

```
python -m pydoc sys
```

在终端提示符下将通过*sys* 模块显示文档内容，其样式类似于 Unix **man** 命令所显示的指南页面。**pydoc** 的参数可以为函数、模块、包，或带点号的对模块中的类、方法或函数以及包中的模块的引用。如果传给 **pydoc** 的参数像是一个路径（即包含所在操作系统的路径分隔符，例如 Unix 的正斜杠），并且其指向一个现有的 Python 源文件，则会为该文件生成文档内容。

备注

为了找到对象及其文档的内容，pydoc 会导入文档所属的模块。因而，在此情况下任何模块层级的代码都将被执行。请使用 `if __name__ == '__main__':` 来确保特定代码仅在文件是作为脚本被唤起而不是被导入时执行。

当将输出打印到控制台时，**pydoc** 会尝试对输出进行分页以更容易阅读。如果设置了 MANPAGER 或 PAGER 环境变量，**pydoc** 将使用其值作为分页程序。当两者均被设置时，将使用 MANPAGER。

在参数前指定 -w 旗标将把 HTML 文档写入到当前目录下的一个文件中，而不是在控制台中显示文本。

在参数前指定 -k 旗标将在全部可用模块的提要行中搜索参数所给定的关键字，具体方式同样类似于 Unix **man** 命令。模块的提要行就是其文档字符串的第一行。

你还可以使用 **pydoc** 在本机上启动一个 HTTP 服务器并向来访的 Web 浏览器展示文档。**python -m pydoc -p 1234** 将在 1234 端口上启动 HTTP 服务器，允许在你所用的 Web 浏览器上通过 http://localhost:1234/ 来浏览文档。指定 0 作为端口号将任意选择一个未使用的端口。

python -m pydoc -n <hostname> 将启动在给定主机名上监听的服务器。默认的主机名为'localhost' 但是如果你希望能从其他机器上搜索该服务器，你可能会想要改变服务器所响应的主机名。在开发阶段此特性会特别有用，如果你想要在一个容器中运行 pydoc 的话。

python -m pydoc -b 将启动服务器并额外打开一个 Web 浏览器访问模块索引页。所发布的每个页面顶端都带有导航栏，你可以点击 *Get* 来获取特定条目的帮助信息，点击 *Search* 在所有模块的摘要行中搜索某个关键词，或点击 *Module index*, *Topics* 和 *Keywords* 前往相应的页面。

当 **pydoc** 生成文档内容时，它会使用当前环境和路径来定位模块。因此，唤起 **pydoc spam** 得到的文档版本会与你启动 Python 解释器并输入 `import spam` 时得到的模块版本完全相同。

核心模块的模块文档应当位于 https://docs.python.org/X.Y/library/ 其中 X 和 Y 是 Python 解释器的主要和次要版本号。这可以通过将 PYTHONDOCS 环境变量设为不同的 URL 或包含标准库参考指南页面的本地目录来覆盖。

在 3.2 版本发生变更: 添加 -b 选项。

在 3.3 版本发生变更: 命令行选项 -g 已经移除。

在 3.4 版本发生变更: 现在 pydoc 会使用*inspect.signature()* 而不是*inspect.getfullargspec()* 来从可调用对象中提取签名信息。

在 3.7 版本发生变更: 添加 -n 选项。

27.3 Python 开发模式

Added in version 3.7.

开发模式下的 Python 加入了额外的运行时检查，由于开销太大，并非默认启用的。如果代码能够正确执行，默认的调试级别足矣，不应再提高了；仅当觉察到问题时再提升警告触发的级别。

使用 -X dev 命令行参数或将环境变量 PYTHONDEVMODE 置为 1 ，可以启用开发模式。

另请参考 Python debug build 。

27.3.1 Python 开发模式的效果

启用 Python 开发模式后的效果，与以下命令类似，不过还有下面的额外效果：

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python -W default -X faulthandler
```

Python 开发模式的效果：

- 加入 `default` *warning filter* 。下述警告信息将会显示出来：
  - *DeprecationWarning*
  - *ImportWarning*
  - *PendingDeprecationWarning*
  - *ResourceWarning*

  通常上述警告是由默认的*warning filters* 负责处理的。

  效果类似于采用了 `-W default` 命令行参数。

  使用命令行参数 `-W error` 或将环境变量 `PYTHONWARNINGS` 设为 `error`，可将警告视为错误。
- 在内存分配程序中安装调试钩子，用以查看：
  - 缓冲区下溢
  - 缓冲区上溢
  - 内存分配 API 冲突
  - 不安全的 GIL 调用

  参见 C 函数 `PyMem_SetupDebugHooks()` 。

  效果如同将环境变量 `PYTHONMALLOC` 设为 `debug`。

  若要启用 Python 开发模式，却又不要在内存分配程序中安装调试钩子，请将环境变量 `PYTHONMALLOC` 设为 `default`。
- 在 Python 启动时调用 *faulthandler.enable()* 来为 *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* 和 *SIGILL* 信号安装处理器以便在程序崩溃时转储 Python 回溯信息。

  其行为如同使用了 `-X faulthandler` 命令行选项或将 `PYTHONFAULTHANDLER` 环境变量设为 `1`。
- 启用*asyncio debug mode*。比如 *asyncio* 会检查没有等待的协程并记录下来。

  效果如同将环境变量 `PYTHONASYNCIODEBUG` 设为 `1`。
- 检查字符串编码和解码函数的 *encoding* 和 *errors* 参数。例如：*open()* 、*str.encode()* 和 *bytes.decode()*。

  为了获得最佳性能，默认只会在第一次编码/解码错误时才会检查 *errors* 参数，有时 *encoding* 参数为空字符串时还会被忽略。
- *io.IOBase* 的析构函数会记录 `close()` 触发的异常。
- 将 *sys.flags* 的 *dev_mode* 属性设为 `True`。

Python 开发模式下，默认不会启用 *tracemalloc* 模块，因为其性能和内存开销太大。启用 *tracemalloc* 模块后，能够提供有关错误来源的一些额外信息。例如，*ResourceWarning* 记录了资源分配的跟踪信息，而缓冲区溢出错误记录了内存块分配的跟踪信息。

Python 开发模式不会阻止命令行参数 `-O` 删除 `assert` 语句，也不会阻止将 *__debug__* 设为 `False`。

Python 开发模式只能在 Python 启动时启用。其参数值可从 *sys.flags.dev_mode* 读取。

在 3.8 版本发生变更: 现在，*io.IOBase* 的析构函数会记录 `close()` 触发的异常。

在 3.9 版本发生变更: 现在，字符串编码和解码操作时会检查 *encoding* 和 *errors* 参数。

27.3.2 ResourceWarning 示例

以下示例将统计由命令行指定的文本文件的行数：

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # 文件将隐式地关闭

if __name__ == "__main__":
    main()
```

上述代码没有显式关闭文件。默认情况下，Python 不会触发任何警告。下面用 README.txt 文件测试下，有 269 行：

```
$ python script.py README.txt
269
```

启用 Python 开发模式后，则会显示一条*ResourceWarning* 警告：

```
$ python -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst' mode='r'␣
↪encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

启用*tracemalloc* 后，则还会显示打开文件的那行代码：

```
$ python -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst' mode='r'␣
↪encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

修正方案就是显式关闭文件。下面用上下文管理器作为示例：

```
def main():
    # 在退出 with 语句块时显式地关闭文件
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

未能显式关闭资源，会让资源打开时长远超预期；在退出 Python 时可能会导致严重问题。这在 CPython 中比较糟糕，但在 PyPy 中会更糟。显式关闭资源能让应用程序更加稳定可靠。

27.3.3 文件描述符错误示例

显示自身的第一行代码：

```
import os

def main():
    fp = open(__file__)
```

(续下页)

(接上页)

```
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # 文件被隐式地关闭

main()
```

默认情况下，Python 不会触发任何警告：

```
$ python script.py
import os
```

在 Python 开发模式下，会在析构文件对象时显示*ResourceWarning* 并记录“Bad file descriptor”错误。

```
$ python -X dev script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='script.py' mode='r'␣
↪encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py' mode='r' encoding='UTF-8'>
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

`os.close(fp.fileno())` 会关闭文件描述符。当文件对象析构函数试图再次关闭文件描述符时会失败，并触发 `Bad file descriptor` 错误。每个文件描述符只允许关闭一次。在最坏的情况下，关闭两次会导致程序崩溃（示例可参见 bpo-18748 ）。

修正方案是删除 `os.close(fp.fileno())` 这一行，或者打开文件时带上 `closefd=False` 参数。

27.4 `doctest` --- 测试交互式的 Python 示例

源代码： Lib/doctest.py

---

doctest 模块寻找像 Python 交互式代码的文本，然后执行这些代码来确保它们的确就像展示的那样正确运行，有许多方法来使用 doctest：

- 通过验证所有交互式示例仍然按照记录的方式工作，以此来检查模块的文档字符串是否是最新的。
- 通过验证来自一个测试文件或一个测试对象的交互式示例按预期工作，来进行回归测试。
- 为一个包写指导性的文档，用输入输出的例子来说明。取决于是强调例子还是说明性的文字，这有一种” 文本测试” 或” 可执行文档” 的风格。

下面是一个小却完整的示例模块：

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.
```

(续下页)

(接上页)

```
    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # 捕获像 1e300 这样的值
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

如果你直接在命令行里运行 `example.py`，*doctest* 将发挥它的作用。

```
$ python example.py
$
```

没有输出！这很正常，这意味着所有的例子都成功了。把 `-v` 传给脚本，*doctest* 会打印出它所尝试的详细日志，并在最后打印出一个总结。

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
```

(续下页)

(接上页)

```
ok
```

以此类推，最终以：

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 test in __main__
   6 tests in __main__.factorial
7 tests in 2 items.
7 passed.
Test passed.
$
```

这就是关于高效使用*doctest* 你所需要知道的一切！开始上手吧。下面的小节提供了完整细节。请注意在标准 Python 测试套件和库中有许多 doctest 的例子。特别有用的例子可以在标准测试文件 `Lib/test/test_doctest/test_doctest.py` 中找到。

27.4.1 简单用法：检查 Docstrings 中的示例

开始使用 doctest 的最简单方式（但不一定是你今后沿用的方式）是这样结束每个模块 `M`:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

`doctest` 会随后检查模块 `M` 中的文档字符串。

以脚本形式运行该模块会使文档中的例子得到执行和验证:

```
python M.py
```

这不会显示任何东西，除非一个例子失败了，在这种情况下，失败的例子和失败的原因会被打印到 stdout，最后一行的输出是 `***Test Failed*** N failures.`，其中 *N* 是失败的例子的数量。

用 `-v` 来运行它来切换，而不是:

```
python M.py -v
```

并将所有尝试过的例子的详细报告打印到标准输出，最后还有各种总结。

你可以通过向*testmod()* 传递 `verbose=True` 来强制执行 verbose 模式，或者通过传递 `verbose=False` 来禁止它。在这两种情况下，`sys.argv` 都不会被*testmod()* 检查（所以传递 `-v` 或不传递都没有影响）。

另外还有一个运行*testmod()* 的命令行快捷方式，参见*命令行用法* 一节。

关于*testmod()* 的更多信息，请参见*基本 API* 部分。

27.4.2 简单的用法：检查文本文件中的例子

doctest 的另一个简单应用是测试文本文件中的交互式例子。这可以用*testfile()* 函数来完成:

```
import doctest
doctest.testfile("example.txt")
```

这个简短的脚本执行并验证文件 `example.txt` 中包含的任何交互式 Python 示例。该文件的内容被当作一个巨大的文档串来处理；该文件不需要包含一个 Python 程序！例如，也许 `example.txt` 包含以下内容:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

运行 `doctest.testfile("example.txt")`，然后发现这个文档中的错误:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

与*testmod()* 一样，*testfile()* 不会显示任何东西，除非一个例子失败。如果一个例子失败了，那么失败的例子和失败的原因将被打印到 stdout，使用的格式与*testmod()* 相同。

默认情况下，*testfile()* 在调用模块的目录中寻找文件。参见章节基本 *API*，了解可用于告诉它在其他位置寻找文件的可选参数的描述。

像*testmod()* 一样，*testfile()* 的详细程度可以通过命令行 `-v` 切换或可选的关键字参数 *verbose* 来设置。

另外还有一个运行*testfile()* 的命令行快捷方式，参见命令行用法 一节。

关于*testfile()* 的更多信息，请参见基本 *API* 一节。

27.4.3 命令行用法

doctest 模块可以在命令行下作为一个脚本来唤起:

```
python -m doctest [-v] [-o OPTION] [-f] file [file ...]
```

-v, --verbose

所有尝试过的例子的详细报告将被打印到标准输出，最后还会带上各种总结:

```
python -m doctest -v example.py
```

这会将 `example.py` 作为独立模块导入并对其运行*testmod()*。请注意如果文件是某个包的一部分并且从那个包导入了其他子模块则此命令可能无法正确工作。

如果文件名不是以 `.py` 结尾，`doctest` 将推断它必须改用*testfile()* 来运行:

```
python -m doctest -v example.txt
```

-o, --option <option>

控制 doctest 行为的各种不同方面，参见选项标记 一节。

Added in version 3.4.

-f, --fail-fast

这是 `-o FAIL_FAST` 的简写形式。

Added in version 3.4.

27.4.4 它是如何工作的

这一节详细研究了 doctest 的工作原理：它查看哪些文档串，它如何找到交互式的用例，它使用什么执行环境，它如何处理异常，以及如何用选项标志来控制其行为。这是你写 doctest 例子所需要知道的信息；关于在这些例子上实际运行 doctest 的信息，请看下面的章节。

哪些文件串被检查了？

模块的文档串以及所有函数、类和方法的文档串都将被搜索。导入模块的对象不被搜索。

此外，有时你会希望测试成为模块的一部分但不是帮助文本的一部分，这就要求测试不包括在文档字符串中。在此情况下 doctest 会查找一个名为 `__test__` 的模块级变量并用它来定位其他测试。如果 `M.__test__` 存在，则它必须是一个字典，其中每个条目都是将（字符串）名称映射到函数对象、类对象或字符串。从 `M.__test__` 找到的函数和类对象的文档字符串将会被搜索，字符串会被当作文档字符串来处理。在输出中，`M.__test__` 中的键 `K` 将作为名称 `M.__test__.K` 出现。

例如，将这段代码放在 `example.py` 的开头:

```
__test__ = {
    'numbers': """
>>> factorial(6)
720

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
"""
}
```

`example.__test__["numbers"]` 的值将被视为一个文档字符串并且其中的所有测试将被运行。重要的注意事项是该值可以被映射到一个函数、类对象或模块；如果是这样的话，`doctest` 会递归地搜索它们的文档字符串，然后扫描其中的测试。

任何发现的类都会以类似的方式进行递归搜索，以测试其包含的方法和嵌套类中的文档串。

文档串的例子是如何被识别的？

在大多数情况下，对交互式控制台会话的复制和粘贴功能工作得很好，但是 doctest 并不试图对任何特定的 Python shell 进行精确的模拟。

```
>>> # 注释将被忽略
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

任何预期的输出必须紧随包含代码的最后 `'>>> '` 或 `'... '` 行，预期的输出（如果有的话）延伸到下一 `'>>> '` 行或全空白行。

fine 输出：

- 预期输出不能包含一个全白的行，因为这样的行被认为是预期输出的结束信号。如果预期的输出包含一个空行，在你的测试例子中，在每一个预期有空行的地方加上 <BLANKLINE>。
- 所有硬制表符都被扩展为空格，使用 8 列的制表符。由测试代码生成的输出中的制表符不会被修改。因为样本输出中的任何硬制表符都会被扩展，这意味着如果代码输出包括硬制表符，文档测试通过的唯一方法是*NORMALIZE_WHITESPACE* 选项或者指令 是有效的。另外，测试可以被重写，以捕获输出并将其与预期值进行比较，作为测试的一部分。这种对源码中标签的处理是通过试错得出的，并被证明是最不容易出错的处理方式。通过编写一个自定义的*DocTestParser* 类，可以使用一个不同的算法来处理标签。
- 向 stdout 的输出被捕获，但不向 stderr 输出（异常回溯通过不同的方式被捕获）。
- 如果你在交互式会话中通过反斜线续行，或出于任何其他原因使用反斜线，你应该使用原始文件串，它将完全保留你输入的反斜线:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

  否则，反斜杠将被解释为字符串的一部分。例如，上面的 \n 会被解释为一个换行符。另外，你可以在 doctest 版本中把每个反斜杠加倍（而不使用原始字符串）：

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
...
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- 起始列并不重要:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

  并从预期的输出中剥离出与开始该例子的初始 '>>> ' 行中出现的同样多的前导空白字符。

什么是执行上下文？

默认情况下，每次*doctest* 找到要测试的文档字符串时，它都会使用 M 的全局变量的一个 浅拷贝，这样运行测试就不会改变模块真正的全局变量，并且 M 中的一个测试也不会留下任何痕迹从而意外地让另一个测试通过。这意味着示例可以自由地使用 M 中任何在最高层级上定义的名称，以及正在运行的文档字符串中之前定义的名称。示例将无法看到在其他文档字符串中定义的名称。

你可以通过将 globs=your_dict 传递给*testmod()* 或*testfile()* 来强制使用你自己的 dict 作为执行环境。

异常如何处理？

没问题，只要回溯是这个例子产生的唯一输出：只要粘贴回溯即可。[1] 由于回溯所包含的细节可能会迅速变化（例如，确切的文件路径和行号），这是 doctest 努力使其接受的内容具有灵活性的一种情况。

简单实例:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
```

(续下页)

[1] 不支持同时包含预期输出和异常的用例。试图猜测一个在哪里结束，另一个在哪里开始，太容易出错了，而且这也会使测试变得混乱。

(接上页)

```
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

如果*ValueError*被触发，该测试就会成功，`list.remove(x): x not in list`的细节如图所示。

异常的预期输出必须以回溯头开始，可以是以下两行中的任何一行，缩进程度与例子中的第一行相同:

```
Traceback (most recent call last):
Traceback (innermost last):
```

回溯头的后面是一个可选的回溯堆栈，其内容被 doctest 忽略。回溯堆栈通常是省略的，或者从交互式会话中逐字复制的。

回溯堆栈的后面是最有用的部分：包含异常类型和细节的一行（几行）。这通常是回溯的最后一行，但如果异常有多行细节，则可以延伸到多行:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

最后三行（以*ValueError*开头）将与异常的类型和细节进行比较，其余的被忽略。

最佳实践是省略回溯栈，除非它为这个例子增加了重要的文档价值。因此，最后一个例子可能更好，因为:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

请注意，回溯的处理方式非常特别。特别是，在重写的例子中，`...`的使用与 doctest 的*ELLIPSIS*选项无关。该例子中的省略号可以不写，也可以是三个（或三百个）逗号或数字，或者是一个缩进的 Monty Python 短剧的剧本。

有些细节你应该读一遍，但不需要记住：

- Doctest 不能猜测你的预期输出是来自异常回溯还是来自普通打印。因此，例如，一个期望`ValueError: 42 is prime`的用例将通过测试，无论*ValueError*是真的被触发，或者该用例只是打印了该回溯文本。在实践中，普通输出很少以回溯标题行开始，所以这不会产生真正的问题。
- 回溯堆栈的每一行（如果有的话）必须比例子的第一行缩进，或者以一个非字母数字的字符开始。回溯头之后的第一行缩进程度相同，并且以字母数字开始，被认为是异常细节的开始。当然，这对真正的回溯来说是正确的事情。
- 当*IGNORE_EXCEPTION_DETAIL* doctest 选项被指定时，最左边的冒号后面的所有内容以及异常名称中的任何模块信息都被忽略。
- 交互式 shell 省略了一些*SyntaxError*的回溯头行。但 doctest 使用回溯头行来区分异常和非异常。所以在罕见的情况下，如果你需要测试一个省略了回溯头的*SyntaxError*，你将需要手动添加回溯头行到你的测试用例中。
- 对于某些异常，Python 会使用 ^ 标记和波浪号来显示错误位置:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ~~^~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

由于显示错误位置的行在异常类型和细节之前，它们不被 doctest 检查。例如，下面的测试会通过，尽管它把 ^ 标记放在了错误的位置:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ^~~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

选项标记

一系列选项旗标控制着 doctest 的各方面行为。旗标的符号名称以模块常量的形式提供，可以一起 bitwise ORed 并传递给各种函数。这些名称也可以在*doctest directives* 中使用，并且可以通过 `-o` 选项传递给 doctest 命令行接口。

第一组选项定义了测试语义，控制 doctest 如何决定实际输出是否与用例的预期输出相匹配方面的问题。

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**

默认情况下，如果一个预期的输出块只包含 `1`，那么实际的输出块只包含 `1` 或只包含 `True` 就被认为是匹配的，同样，`0` 与 `False` 也是如此。当*DONT_ACCEPT_TRUE_FOR_1* 被指定时，两种替换都不允许。默认行为是为了适应 Python 将许多函数的返回类型从整数改为布尔值；期望" 小整数" 输出的测试在这些情况下仍然有效。这个选项可能会消失，但不会在几年内消失。

`doctest.`**`DONT_ACCEPT_BLANKLINE`**

默认情况下，如果一个预期输出块包含一个只包含字符串 `<BLANKLINE>` 的行，那么该行将与实际输出中的一个空行相匹配。因为一个真正的空行是对预期输出的限定，这是传达预期空行的唯一方法。当*DONT_ACCEPT_BLANKLINE* 被指定时，这种替换是不允许的。

`doctest.`**`NORMALIZE_WHITESPACE`**

当指定时，所有的空白序列（空白和换行）都被视为相等。预期输出中的任何空白序列将与实际输出中的任何空白序列匹配。默认情况下，空白必须完全匹配。*NORMALIZE_WHITESPACE* 在预期输出非常长的一行，而你想把它包在源代码的多行中时特别有用。

`doctest.`**`ELLIPSIS`**

当指定时，预期输出中的省略号（`...`）可以匹配实际输出中的任何子串。这包括跨行的子串和空子串，所以最好保持简单的用法。复杂的用法会导致与 `.*` 在正则表达式中容易出现的"oops, it matched too much!" 相同的意外情况。

`doctest.`**`IGNORE_EXCEPTION_DETAIL`**

当被指定时，只要有预期类型的异常被引发 doctests 就会预期异常测试通过，即使细节（消息和完整限定名称）并不匹配。

举例来说，一个预期 `ValueError: 42` 的用例在实际引发的异常为 `ValueError: 3*14` 将会通过，但是如果是引发*TypeError* 则将会失败。它也将忽略任何包括在异常类前面的完整限定名称，该名称在所使用的不同 Python 版本和代码/库中可能会不同。因此，以下三种形式对于指定的旗标均有效:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: message
```

请注意*ELLIPSIS* 也可以被用来忽略异常消息中的细节，但这样的测试仍然可能根据特定模块名称是否存在或是否完全匹配而失败。

在 3.2 版本发生变更: *IGNORE_EXCEPTION_DETAIL* 现在也忽略了与包含被测异常的模块有关的任何信息。

doctest.**SKIP**

当指定时，完全不运行这个用例。这在 doctest 用例既是文档又是测试案例的情况下很有用，一个例子应该包括在文档中，但不应该被检查。例如，这个例子的输出可能是随机的；或者这个例子可能依赖于测试驱动程序所不能使用的资源。

SKIP 标志也可用于临时" 注释" 用例。

doctest.**COMPARISON_FLAGS**

一个比特或运算将上述所有的比较标志放在一起。

第二组选项控制测试失败的报告方式：

doctest.**REPORT_UDIFF**

当指定时，涉及多行预期和实际输出的故障将使用统一的差异来显示。

doctest.**REPORT_CDIFF**

当指定时，涉及多行预期和实际输出的故障将使用上下文差异来显示。

doctest.**REPORT_NDIFF**

当指定时，差异由 `difflib.Differ` 来计算，使用与流行的:file:*ndiff.py*'工具相同的算法。这是唯一一种标记行内和行间差异的方法。例如，如果一行预期输出包含数字 *"1"*，而实际输出包含字母 `l`，那么就会插入一行，用圆点标记不匹配的列位置。

doctest.**REPORT_ONLY_FIRST_FAILURE**

当指定时，在每个 doctest 中显示第一个失败的用例，但隐藏所有其余用例的输出。这将防止 doctest 报告由于先前的失败而中断的正确用例；但也可能隐藏独立于第一个失败的不正确用例。当*REPORT_ONLY_FIRST_FAILURE* 被指定时，其余的用例仍然被运行，并且仍然计入报告的失败总数；只是输出被隐藏了。

doctest.**FAIL_FAST**

当指定时，在第一个失败的用例后退出，不尝试运行其余的用例。因此，报告的失败次数最多为 1。这个标志在调试时可能很有用，因为第一个失败后的用例甚至不会产生调试输出。

doctest.**REPORTING_FLAGS**

一个比特或操作将上述所有的报告标志组合在一起。

还有一种方法可以注册新的选项标志名称，不过这并不有用，除非你打算通过子类来扩展*doctest* 内部。

doctest.**register_optionflag**(*name*)

用给定的名称创建一个新的选项标志，并返回新标志的整数值。*register_optionflag()* 可以在继承*OutputChecker* 或*DocTestRunner* 时使用，以创建子类支持的新选项。*register_optionflag()* 应始终使用以下方式调用：

```
MY_FLAG = register_optionflag('MY_FLAG')
```

指令

Doctest 指令可以用来修改单个例子的*option flags* 。Doctest 指令是在一个用例的源代码后面的特殊 Python 注释。

```
directive             ::= "#" "doctest:" directive_options
directive_options     ::= directive_option ("," directive_option)*
directive_option      ::= on_or_off directive_option_name
on_or_off             ::= "+" | "-"
directive_option_name ::= "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...
```

+ 或 - 与指令选项名称之间不允许有空格。指令选项名称可以是上面解释的任何一个选项标志名称。

一个用例的 doctest 指令可以修改 doctest 对该用例的行为。使用 + 来启用指定的行为，或者使用 - 来禁用它。

例如，这个测试将会通过:

```
>>> print(list(range(20)))  # doctest: +NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

如果没有这个指令则它将失败，因为实际的输出在只有个位数的列表元素前没有两个空格，也因为实际的输出是单行的。这个测试也会通过，并且也需要一个指令来完成:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

在单个物理行中可以使用多条指令，以逗号分隔:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

如果在单个用例中使用了多条指令注释，则它们会被合并:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
...                         # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

正如前面的例子所示，你可以在你的用例中添加只包含指令的行 `...`。当一个用例太长以至于不能方便地放到同一行时这将会很有用:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

请注意，由于所有的选项都是默认禁用的，而指令只适用于它们出现的用例，所以启用选项 (通过指令中的 +) 通常是唯一有意义的选择。然而，选项标志也可以被传递给运行测试的函数，建立不同的默认值。在这种情况下，通过指令中的 - 来禁用一个选项可能是有用的。

警告

doctest 是严格地要求在预期输出中完全匹配。如果哪怕只有一个字符不匹配，测试就会失败。这可能会让你吃惊几次，在你确切地了解到 Python 对输出的保证和不保证之前。例如，当打印一个集合时，Python 不保证元素以任何特定的顺序被打印出来，所以像:

```
>>> foo()
{"spam", "eggs"}
```

是不可靠的！一个变通方法是用:

```
>>> foo() == {"spam", "eggs"}
True
```

来取代。另一个是使用

```
>>> d = sorted(foo())
>>> d
['eggs', 'spam']
```

还有其他的问题，但你会明白的。

另一个不好的做法是打印嵌入了对象地址的值，例如

```
>>> id(1.0)  # 确定有时会失败
7948648
>>> class C: pass
>>> C()  # 实例的默认 repr() 嵌入了一个地址
<C object at 0x00AC18F0>
```

ELLIPSIS 指令为之前的例子提供了一个不错的方案:

```
>>> C()  # doctest: +ELLIPSIS
<C object at 0x...>
```

浮点数在不同的平台上也会有小的输出变化，因为 Python 在浮点数的格式化上依赖于平台的 C 库，而 C 库在这个问题上的质量差异很大。:

```
>>> 1./7  # 危险
0.14285714285714285
>>> print(1./7) # 安全
0.142857142857
>>> print(round(1./7, 6)) # 更安全
0.142857
```

形式 `I/2.**J` 的数字在所有的平台上都是安全的，我经常设计一些测试的用例来产生该形式的数:

```
>>> 3./4  # 绝对安全
0.75
```

简单的分数也更容易让人理解，这也使得文件更加完善。

27.4.5 基本 API

函数*testmod()* 和*testfile()* 为 doctest 提供了一个简单的接口，应该足以满足大多数基本用途。关于这两个函数的不太正式的介绍，请参见简单用法：检查 *Docstrings* 中的示例 和简单的用法：检查文本文件中的例子 部分。

doctest.**testfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

除了 *filename*，所有的参数都是可选的，而且应该以关键字的形式指定。

测试名为 *filename* 的文件中的用例。返回 `(failure_count, test_count)`。

可选参数 *module_relative* 指定了文件名的解释方式。

- 如果 *module_relative* 是 `True` (默认)，那么 *filename* 指定一个独立于操作系统的模块相对路径。默认情况下，这个路径是相对于调用模块的目录的；但是如果指定了 *package* 参数，那么它就是相对于该包的。为了保证操作系统的独立性，*filename* 应该使用字符来分隔路径段，并且不能是一个绝对路径 (即不能以 `/` 开始)。
- 如果 *module_relative* 是 `False`，那么 *filename* 指定了一个操作系统特定的路径。路径可以是绝对的，也可以是相对的；相对路径是相对于当前工作目录而言的。

可选参数 *name* 给出了测试的名称；默认情况下，或者如果是 `None`，那么使用 `os.path.basename(filename)`。

可选参数 *package* 是一个 Python 包或一个 Python 包的名字，其目录应被用作模块相关文件名的基础目录。如果没有指定包，那么调用模块的目录将作为模块相关文件名的基础目录。如果 *module_relative* 是 `False`，那么指定 *package* 是错误的。

可选参数 *globs* 给出了一个在执行示例时用作全局变量的 dict。这个 dict 的一个新的浅层副本将为 doctest 创建，因此它的用例将从一个干净的地方开始。默认情况下，或者如果为 `None`，使用一个新的空 dict。

可选参数 *extraglobs* 给出了一个合并到用于执行用例全局变量中的 dict。这就像*dict.update()* 一样：如果 *globs* 和 *extraglobs* 有一个共同的键，那么 *extraglobs* 中的相关值会出现在合并的 dict 中。默

认情况下，或者为 None ，则不使用额外的全局变量。这是一个高级功能，允许对 doctest 进行参数化。例如，可以为一个基类写一个测试，使用该类的通用名称，然后通过传递一个 *extraglobs* dict，将通用名称映射到要测试的子类，从而重复用于测试任何数量的子类。

可选的参数 *verbose* 如果为真值会打印很多东西，如果为假值则只打印失败信息；默认情况下，或者为 None，只有当 '-v' 在 sys.argv 中时才为真值。

可选参数 *report* 为 True 时，在结尾处打印一个总结，否则在结尾处什么都不打印。在 verbose 模式下，总结是详细的，否则总结是非常简短的（事实上，如果所有的测试都通过了，总结就是空的）。

可选参数 *optionflags* （默认值为 0）是选项标志的 bitwise OR 。参见章节选项标记 。

可选参数 *raise_on_error* 默认为 False。如果是 True，在一个用例中第一次出现失败或意外的异常时，会触发一个异常。这允许对失败进行事后调试。默认行为是继续运行例子。

可选参数 *parser* 指定一个*DocTestParser* （或子类），它应该被用来从文件中提取测试。它默认为一个普通的解析器（即 DocTestParser()）。

可选参数 *encoding* 指定了一个编码，应该用来将文件转换为 unicode。

doctest.**testmod**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

所有的参数都是可选的，除了 *m* 之外，都应该以关键字的形式指定。

测试从模块 *m* （或模块*__main__* ，如果 *m* 没有被提供或为 None ）可达到的函数和类的文档串中的用例，从 m.__doc__ 开始。

还会测试从 dict m.__test__ 可达到的用例，如果存在的话。m.__test__ 将名称（字符串）映射到函数、类和字符串；将在函数和类的文档字符串中搜索用例；字符串将被直接搜索，就像它们是文档字符串一样。

只搜索附属于模块 *m* 中的对象的文档串。

返回 (failure_count, test_count) 。

可选参数 *name* 给出了模块的名称；默认情况下，或者如果为 None ，则为 m.__name__ 。

可选参数 *exclude_empty* 默认为假值。如果为真值，则未找到任何 doctests 的对象将被排除在考虑范围之外。默认情况下将做向下兼容处理，因而仍然使用*doctest.master.summarize* 来配合*testmod()* 的代码会继续得到没有测试的对象的输出。转给较新的*DocTestFinder* 构造器的 *exclude_empty* 参数默认为真值。

可选参数 *extraglobs* 、*verbose* 、*report* 、*optionflags* 、*raise_on_error* 和 *globs* 与上述函数*testfile()* 的参数相同，只是 *globs* 默认为 m.__dict__ 。

doctest.**run_docstring_examples**(*f*, *globs*, *verbose=False*, *name='NoName'*, *compileflags=None*, *optionflags=0*)

与对象 *f* 相关的测试用例；例如，*f* 可以是一个字符串、一个模块、一个函数或一个类对象。

dict 参数 *globs* 的浅层拷贝被用于执行环境。

可选参数 *name* 在失败信息中使用，默认为 "NoName" 。

如果可选参数 *verbose* 为真，即使没有失败也会产生输出。默认情况下，只有在用例失败的情况下才会产生输出。

可选参数 *compileflags* 给出了 Python 编译器在运行例子时应该使用的标志集。默认情况下，或者如果为 None ，标志是根据 *globs* 中发现的未来特征集推导出来的。

可选参数 *optionflags* 的作用与上述*testfile()* 函数中的相同。

27.4.6 Unittest API

随着你带有文档测试的模块不断增加，你会希望以系统性地方式运行所有的文档测试。*doctest* 提供了两个函数可用来根据模块和包含文档测试的文本文件创建*unittest* 测试套件。要与*unittest* 测试发现功能实现集成，请在你的测试模块中包括一个*load_tests* 函数:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

有两个主要函数用于从文本文件和带 doctest 的模块中创建*unittest.TestSuite* 实例。

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

将一个或多个文本文件中的 doctest 测试转换为一个*unittest.TestSuite* 。

返回的*unittest.TestSuite* 将由 unittest 框架运行并运行每个文件中的交互式示例。如果任何文件中的示例运行失败，则合成的单元测试就将失败，并引发一个*failureException* 异常显示包含该测试的文件名以及（有时为近似的）行号。如果某个文件中的示例被跳过，则合成的单元测试也会被标记为跳过。

传递一个或多个要检查的文本文件的路径（作为字符串）。

选项可以作为关键字参数提供：

可选参数 *module_relative* 指定了 *paths* 中的文件名应该如何解释。

- 如果 *module_relative* 是 `True` （默认值），那么 *paths* 中的每个文件名都指定了一个独立于操作系统的模块相对路径。默认情况下，这个路径是相对于调用模块的目录的；但是如果指定了 *package* 参数，那么它就是相对于该包的。为了保证操作系统的独立性，每个文件名都应该使用字符来分隔路径段，并且不能是绝对路径（即不能以 / 开始）。
- 如果 *module_relative* 是 `False` ，那么 *paths* 中的每个文件名都指定了一个操作系统特定的路径。路径可以是绝对的，也可以是相对的；相对路径是关于当前工作目录的解析。

可选参数 *package* 是一个 Python 包或一个 Python 包的名字，其目录应该被用作 *paths* 中模块相关文件名的基本目录。如果没有指定包，那么调用模块的目录将作为模块相关文件名的基础目录。如果 *module_relative* 是 `False` ，那么指定 *package* 是错误的。

可选的参数 *setUp* 为测试套件指定了一个设置函数。在运行每个文件中的测试之前，它被调用。*setUp* 函数将被传递给一个*DocTest* 对象。setUp 函数可以通过测试的 *globs* 属性访问测试的全局变量。

可选的参数 *tearDown* 指定了测试套件的卸载函数。在运行每个文件中的测试后，它会被调用。*tearDown* 函数将被传递一个*DocTest* 对象。setUp 函数可以通过测试的 *globs* 属性访问测试的全局变量。

可选参数 *globs* 是一个包含测试的初始全局变量的字典。这个字典的一个新副本为每个测试创建。默认情况下，*globs* 是一个新的空字典。

可选参数 *optionflags* 为测试指定默认的 doctest 选项，通过将各个选项的标志连接在一起创建。参见章节选项标记 。参见下面的函数*set_unittest_reportflags()* ，以了解设置报告选项的更好方法。

可选参数 *parser* 指定一个*DocTestParser* （或子类），它应该被用来从文件中提取测试。它默认为一个普通的解析器（即 `DocTestParser()` ）。

可选参数 *encoding* 指定了一个编码，应该用来将文件转换为 unicode。

该全局 `__file__` 被添加到提供给用*DocFileSuite()* 从文本文件加载的 doctest 的全局变量中。

doctest.**DocTestSuite**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *optionflags=0*, *checker=None*)

将一个模块的 doctest 测试转换为*unittest.TestSuite* 。

返回的*unittest.TestSuite* 将由 unittest 框架运行并运行模块中的每个文档测试。如果有任何文档测试运行失败，则合成的单元测试就将失败，并引发一个*failureException* 异常显示包含该测

试的文件名以及（有时为近似的）行号。如果文档字符串中的所有示例都被跳过，则合成的单元测试也会被标记为跳过。

可选参数 *module* 提供了要测试的模块。它可以是一个模块对象或一个（可能是带点的）模块名称。如果没有指定，则使用调用此函数的模块。

可选参数 *globs* 是一个包含测试的初始全局变量的字典。这个字典的一个新副本为每个测试创建。默认情况下，*globs* 是一个新的空字典。

可选参数 *extraglobs* 指定了一组额外的全局变量，这些变量被合并到 *globs* 中。默认情况下，不使用额外的全局变量。

可选参数 *test_finder* 是*DocTestFinder* 对象（或一个可替换的对象），用于从模块中提取测试。

可选参数 *setUp* 、*tearDown* 和 *optionflags* 与上述函数*DocFileSuite()* 相同。

这个函数使用与*testmod()* 相同的搜索技术。

在 3.5 版本发生变更: 如果 *module* 不包含任何文件串，则*DocTestSuite()* 返回一个空的*unittest.TestSuite*，而不是触发*ValueError*。

exception doctest.**failureException**

当已被*DocFileSuite()* 或*DocTestSuite()* 转换为单元测试的文档测试失败时，此异常将被引发并显示包含测试的文件名及行号（有时为估计值）。

在内部，*DocTestSuite()* 将根据 doctest.DocTestCase 实例创建*unittest.TestSuite*，而 DocTestCase 是*unittest.TestCase* 的子类。DocTestCase 没有记入本文档（它属于内部细节），但研究其代码可以回答有关*unittest* 集成的准确细节的问题。

类似地，*DocFileSuite()* 将根据 doctest.DocFileCase 实例创建*unittest.TestSuite*，而 DocFileCase 是 DocTestCase 的子类。

所以这两种创建*unittest.TestSuite* 的方式都会运行 DocTestCase 的实例。这一点很重要，因为有一个微妙的原因：当你自己运行*doctest* 函数时，你可以直接控制使用中的*doctest* 选项，具体是通过传递选项旗标给*doctest* 函数。然而，如果你正在编写一个*unittest* 框架，则最终要由*unittest* 来控制测试的运行时间和方式。框架作者通常希望控制*doctest* 的报告选项（例如，可能由命令行选项指定），但没有办法通过*unittest* 向*doctest* 测试运行方传递选项。

出于这个原因，*doctest* 也支持一个概念，即*doctest* 报告特定于*unittest* 支持的标志，通过这个函数:

doctest.**set_unittest_reportflags**(*flags*)

设置要使用的*doctest* 报告标志。

参数 *flags* 是选项标志的 bitwise OR 。参见章节选项标记 。只有” 报告标志” 可以被使用。

这是模块全局的设置，并会影响*unittest* 模块今后运行的所有文档测试: DocTestCase 的 runTest() 方法会查看 DocTestCase 实例构建时为测试用例指定的选项旗标。如果没有指定报告旗标志（这是典型和预期的情况），则 doctest 的*unittest* 报告旗标志将通过 按位或运算合并选项旗标志中，这样增强的选项标志将传递给为运行文档测试而创建的*DocTestRunner* 实例。如果在构建 DocTestCase 实例时指定了任何报告旗标志，则 doctest 的*unittest* 报告旗标将被忽略。

unittest 报告标志的值在调用该函数之前是有效的，由该函数返回。

27.4.7 高级 API

基本 API 是一个简单的封装，旨在使 doctest 易于使用。它相当灵活，应该能满足大多数用户的需求；但是，如果你需要对测试进行更精细的控制，或者希望扩展 doctest 的功能，那么你应该使用高级 API 。

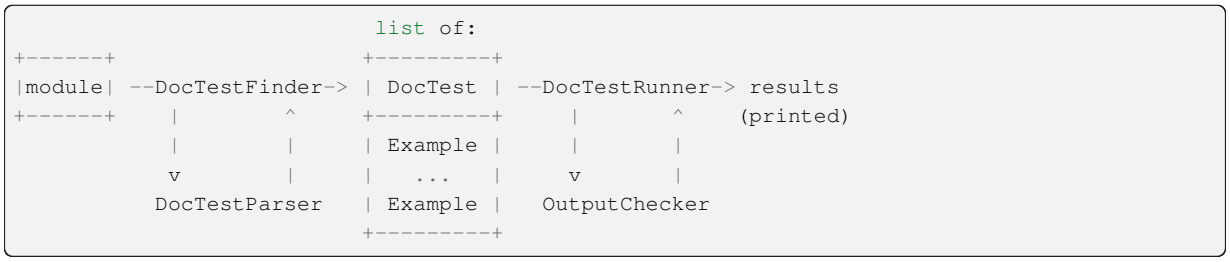
高级 API 围绕着两个容器类，用于存储从 doctest 案例中提取的交互式用例:

- *Example*: 一个单一的 Python *statement* ，与它的预期输出配对。
- *DocTest*: 一组*Example*s 的集合，通常从一个文档字符串或文本文件中提取。

定义了额外的处理类来寻找、解析和运行，并检查 doctest 的用例。

- *DocTestFinder*：查找给定模块中的所有文档串，并使用*DocTestParser* 从每个包含交互式用例的文档串中创建一个*DocTest* 。
- *DocTestParser*：从一个字符串（如一个对象的文档串）创建一个*DocTest* 对象。
- *DocTestRunner*：执行*DocTest* 中的用例，并使用*OutputChecker* 来验证其输出。
- *OutputChecker*：将一个测试用例的实际输出与预期输出进行比较，并决定它们是否匹配。

这些处理类之间的关系总结在下图中:



DocTest 对象

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

应该在单一命名空间中运行的 doctest 用例的集合。构造函数参数被用来初始化相同名称的属性。

DocTest 定义了以下属性。它们由构造函数初始化，不应该被直接修改。

examples

一个*Example* 对象的列表，它编码了应该由该测试运行的单个交互式 Python 用例。

globs

例子应该运行的命名空间（又称 globals ）。这是一个将名字映射到数值的字典。例子对名字空间的任何改变（比如绑定新的变量）将在测试运行后反映在*globs* 中。

name

识别*DocTest* 的字符串名称。通常情况下，这是从测试中提取的对象或文件的名称。

filename

这个*DocTest* 被提取的文件名；或者为 None，如果文件名未知，或者*DocTest* 没有从文件中提取。

lineno

filename 中的行号，这个*DocTest* 开始的地方，或者行号不可用时为 None。这个行号相对于文件的开头来说是零的。

docstring

从测试中提取的字符串，或者如果字符串不可用，或者为 None ，如果测试没有从字符串中提取。

Example 对象

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)

单个交互式用例，由一个 Python 语句及其预期输出组成。构造函数参数被用来初始化相同名称的属性。

Example 定义了以下属性。它们由构造函数初始化，不应该被直接修改。

source

一个包含该用例源码的字符串。源码由一个 Python 语句组成，并且总是以换行结束；构造函数在必要时添加一个换行。

want

运行这个用例的源码的预期输出（可以是 stdout，也可以是异常情况下的回溯）。*want* 以一个换行符结束，除非没有预期的输出，在这种情况下它是一个空字符串。构造函数在必要时添加一个换行。

exc_msg

用例产生的异常信息，如果这个例子被期望产生一个异常；或者为 None，如果它不被期望产生一个异常。这个异常信息与*traceback.format_exception_only()* 的返回值进行比较。*exc_msg* 以换行结束，除非是 None。

lineno

包含本例的字符串中的行号，即本例的开始。这个行号相对于包含字符串的开头来说是以零开始的。

indent

用例在包含字符串中的缩进，即在用例的第一个提示前有多少个空格字符。

options

一个将选项旗标映射到 True 或 False 的字典，用于覆盖这个例子的默认选项。任何不包含在这个字典中的选项旗标都将保持其默认值（由*DocTestRunner* 的*optionflags* 指定）。默认情况下，将不设置任何选项。

DocTestFinder 对象

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

一个处理类，用于从一个给定的对象的 docstring 和其包含的对象的 docstring 中提取与之相关的*DocTest*。*DocTest* 可以从模块、类、函数、方法、静态方法、类方法和属性中提取。

可选的参数 *verbose* 可以用来显示查找器搜索到的对象。它的默认值是 False（无输出）。

可选的参数 *parser* 指定了*DocTestParser* 对象（或一个可替换的对象），用于从文档串中提取 doctest。

如果可选的参数 *recurse* 是 False，那么*DocTestFinder.find()* 将只检查给定的对象，而不是任何包含的对象。

如果可选参数 *exclude_empty* 为 False，那么*DocTestFinder.find()* 将包括对文档字符串为空的对象的测试。

DocTestFinder 定义了以下方法：

find(*obj[, name][, module][, globs][, extraglobs]*)

返回*DocTest* 的列表，该列表由 *obj* 的文档串或其包含的任何对象的文档串定义。

可选参数 *name* 指定了对象的名称；这个名称将被用来为返回的*DocTest* 构建名称。如果没有指定 *name*，则使用 obj.__name__。

可选参数 *module* 是包含给定对象的模块。如果没有指定模块或者是 None，那么测试查找器将试图自动确定正确的模块。该对象被使用的模块：

- 作为一个默认的命名空间，如果没有指定 *globs*。
- 为了防止 DocTestFinder 从其他模块导入的对象中提取 DocTest。(包含有除 *module* 以外的模块的对象会被忽略)。
- 找到包含该对象的文件名。
- 找到该对象在其文件中的行号。

如果 *module* 是 False，将不会试图找到这个模块。这是不明确的，主要用于测试 doctest 本身：如果 *module* 是 False，或者是 None 但不能自动找到，那么所有对象都被认为属于（不存在的）模块，所以所有包含的对象将（递归地）被搜索到 doctest。

每个*DocTest* 的 globals 是由 *globs* 和 *extraglobs* 组合而成的（*extraglobs* 的绑定覆盖 *globs* 的绑定）。为每个*DocTest* 创建一个新的 globals 字典的浅层拷贝。如果没有指定 *globs*，那么它默

认为模块的 *__dict__*，如果指定了或者为 `{}`，则除外。如果没有指定 *extraglobs*，那么它默认为 `{}`。

DocTestParser 对象

class doctest.**DocTestParser**

一个处理类，用于从一个字符串中提取交互式的用例，并使用它们来创建一个*DocTest* 对象。

DocTestParser 定义了以下方法：

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

从给定的字符串中提取所有的测试用例，并将它们收集到一个*DocTest* 对象中。

globs 、*name* 、*filename* 和 *lineno* 是新的*DocTest* 对象的属性。更多信息请参见*DocTest* 的文档。

get_examples(*string*, *name='<string>'*)

从给定的字符串中提取所有的测试用例，并以*Example* 对象列表的形式返回。行数以 0 为基数。可选参数 *name* 用于识别这个字符串的名称，只用于错误信息。

parse(*string*, *name='<string>'*)

将给定的字符串分成用例和中间的文本，并以*Example* 和字符串交替的列表形式返回。*Example* 的行号以 0 为基数。可选参数 *name* 用于识别这个字符串的名称，只用于错误信息。

TestResults 对象

class doctest.**TestResults**(*failed*, *attempted*)

failed

失败的测试数量。

attempted

执行的测试数量。

skipped

跳过的测试数量。

Added in version 3.13.

DocTestRunner 对象

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

一个处理类，用于执行和验证*DocTest* 中的交互式用例。

预期输出和实际输出之间的比较是由一个*OutputChecker* 完成的。这种比较可以用一些选项标志来定制；更多信息请看选项标记 部分。如果选项标志不够，那么也可以通过向构造函数传递*OutputChecker* 的子类来定制比较。

测试运行器的显示输出可以通过两种方式来控制。首先，一个输出函数可以被传递给*run()*；这个函数将被调用并传入要显示的字符串。默认使用 `sys.stdout.write`。如果捕获输出是不够的，那么也可以通过子类化 DocTestRunner，并重写*report_start()*，*report_success()*，*report_unexpected_exception()* 和*report_failure()* 等方法来定制显示输出。

可选的关键字参数 *checker* 指定了*OutputChecker* 对象（或其相似替换），它应该被用来比较预期输出和 doctest 用例的实际输出。

可选的关键字参数 *verbose* 控制*DocTestRunner* 的详细程度。如果 *verbose* 是 `True`，那么每个用例的信息都会被打印出来，当它正在运行时。如果 *verbose* 是 `False`，则只打印失败的信息。当 *verbose* 没有指定，或者为 `None`，如果使用了命令行开关 `-v`，则使用 verbose 输出。

可选的关键字参数 *optionflags* 可以用来控制测试运行器如何比较预期输出和实际输出，以及如何显示失败。更多信息，请参见章节选项标记 。

测试运行将累积统计数据。已尝试、已失败和已跳过的示例的聚合计数也可通过*tries*, *failures* 和*skips* 属性来获取。*run()* 和*summarize()* 方法将返回一个*TestResults* 实例。

DocTestRunner 定义了以下方法：

report_start(*out*, *test*, *example*)

报告测试运行器即将处理给定的用例。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是即将被处理的用。*test* 是 包含用例的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

report_success(*out*, *test*, *example*, *got*)

报告给定的用例运行成功。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是即将被处理的用例。*got* 是这个例子的实际输出。*test* 是包含 *example* 的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

report_failure(*out*, *test*, *example*, *got*)

报告给定的用例运行失败。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是即将被处理的用例。*got* 是这个例子的实际输出。*test* 是包含 *example* 的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

报告给定的用例触发了一个异常。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是将要被处理的用例。*exc_info* 是一个元组，包含关于异常的信息（如由*sys.exc_info()* 返回）。*test* 是包含 *example* 的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

运行 *test* (一个*DocTest* 对象) 中的示例，并使用写入函数 *out* 显示结果。返回一个*TestResults* 实例。

这些用例都是在命名空间 `test.globs` 中运行的。如果 *clear_globs* 为 True （默认），那么这个命名空间将在测试运行后被清除，以帮助进行垃圾回收。如果你想在测试完成后检查命名空间，那么使用 *clear_globs=False* 。

compileflags 给出了 Python 编译器在运行例子时应该使用的标志集。如果没有指定，那么它将默认为适用于 *globs* 的 future-import 标志集。

每个例子的输出都使用 The output of each example is checked using the *DocTestRunner* 的输出检查器进行检查，并且结果将由 `DocTestRunner.report_*()` 方法来格式化。

summarize(*verbose=None*)

打印这个 DocTestRunner 运行过的所有测试用例的概要，并返回一个*TestResults* 实例。

可选的 *verbose* 参数控制摘要的详细程度。如果没有指定 verbose ，那么将使用*DocTestRunner* 的 verbose 。

DocTestParser 具有下列属性：

tries

尝试的示例数量。

failures

失败的示例数量。

skips

跳过的示例数量。

Added in version 3.13.

OutputChecker 对象

class doctest.**OutputChecker**

一个用于检查测试用例的实际输出是否与预期输出相匹配的类。*OutputChecker* 定义了两个方法：*check_output()*，比较给定的一对输出，如果它们匹配则返回 `True`；*output_difference()*，返回一个描述两个输出之间差异的字符串。

OutputChecker 定义了以下方法：

check_output(*want*, *got*, *optionflags*)

如果一个用例的实际输出（*got*）与预期输出（*want*）匹配，则返回 `True`。如果这些字符串是相同的，总是被认为是匹配的；但是根据测试运行器使用的选项标志，也可能有几种非精确的匹配类型。参见章节选项标记 了解更多关于选项标志的信息。

output_difference(*example*, *got*, *optionflags*)

返回一个字符串，描述给定用例（*example*）的预期输出和实际输出（*got*）之间的差异。*optionflags* 是用于比较 *want* 和 *got* 的选项标志集。

27.4.8 调试

Doctest 提供了几种调试 doctest 用例的机制：

- 有几个函数将测试转换为可执行的 Python 程序，这些程序可以在 Python 调试器，*pdb* 下运行。
- *DebugRunner* 类是*DocTestRunner* 的一个子类，它为第一个失败的用例触发一个异常，包含关于这个用例的信息。这些信息可以用来对这个用例进行事后调试。
- 由*DocTestSuite()* 生成的*unittest* 用例支持由*unittest.TestCase* 定义的*debug()* 方法，。
- 你可以在 doctest 的用例中加入对*pdb.set_trace()* 的调用，当这一行被执行时，你会进入 Python 调试器。然后你可以检查变量的当前值，等等。例如，假设 `a.py` 只包含这个模块 docstring

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

那么一个交互式 Python 会话可能是这样的:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
```

(续下页)

(接上页)

```
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

将测试转换为 Python 代码的函数，并可能在调试器下运行合成的代码:

doctest.**script_from_examples**(*s*)

将带有用例的文本转换为脚本。

参数 *s* 是一个包含测试用例的字符串。该字符串被转换为 Python 脚本，其中 *s* 中的 doctest 用例被转换为常规代码，其他的都被转换为 Python 注释。生成的脚本将以字符串的形式返回。例如，

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

显示:

```
# 将 x 和 y 设为 1 和 2。
x, y = 1, 2
#
# 打印两者之和:
print(x+y)
# 预期:
## 3
```

这个函数在内部被其他函数使用（见下文），但当你想把一个交互式 Python 会话转化为 Python 脚本时，也会很有用。

doctest.**testsource**(*module*, *name*)

将一个对象的 doctest 转换为一个脚本。

参数 *module* 是一个模块对象，或是一个带点号的模块名称，其中包含文档测试需要的对象。参数 *name* 是文档测试需要的对象（在模块中）的名称。结果是一个字符串，包含该对象的文档字符串转换成的 Python 脚本，如上面*script_from_examples()* 所描述的。举例来说，如果模块 `a.py` 包含一个最高层级的函数 `f()`，那么

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

打印函数 `f()` 的文档字符串的脚本版本，将文档测试转换为代码，而将其余内容放在注释中。

doctest.**debug**(*module*, *name*, *pm=False*)

对一个对象的 doctest 进行调试。

module 和 *name* 参数与上面函数*testsource()* 的参数相同。被命名对象的文本串的合成 Python 脚本被写入一个临时文件，然后该文件在 Python 调试器*pdb* 的控制下运行。

`module.__dict__` 的一个浅层拷贝被用于本地和全局的执行环境。

可选参数 *pm* 控制是否使用事后调试。如果 *pm* 为 True ，则直接运行脚本文件，只有当脚本通过引发一个未处理的异常而终止时，调试器才会介入。如果是这样，就会通过*pdb.post_mortem()* 调用事后调试，并传递未处理异常的跟踪对象。如果没有指定 *pm* ，或者是 False ，脚本将从一开始就在调试器下运行，通过传递一个适当的*exec()* 调用给*pdb.run()* 。

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

在一个字符串中调试 doctest 。

这就像上面的函数*debug()* ，只是通过 *src* 参数，直接指定一个包含测试用例的字符串。

可选参数 *pm* 的含义与上述函数*debug()* 的含义相同。

可选的参数 *globs* 给出了一个字典，作为本地和全局的执行环境。如果没有指定，或者为 None ，则使用一个空的字典。如果指定，则使用字典的浅层拷贝。

DebugRunner 类，以及它可能触发的特殊异常，是测试框架作者最感兴趣的，在此仅作简要介绍。请看源代码，特别是*DebugRunner* 的文档串（这是一个测试！）以了解更多细节。

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

DocTestRunner 的一个子类，一旦遇到失败，就会触发一个异常。如果一个意外的异常发生，就会引发一个*UnexpectedException* 异常，包含测试、用例和原始异常。如果输出不匹配，那么就会引发一个*DocTestFailure* 异常，包含测试、用例和实际输出。

关于构造函数参数和方法的信息，请参见*DocTestRunner* 部分的文档*高级 API* 。

DebugRunner 实例可能会触发两种异常。

exception doctest.**DocTestFailure**(*test*, *example*, *got*)

DocTestRunner 触发的异常，表示一个 doctest 用例的实际输出与预期输出不一致。构造函数参数被用来初始化相同名称的属性。

DocTestFailure 定义了以下属性:

DocTestFailure.**test**

当该用例失败时正在运行的*DocTest* 对象。

DocTestFailure.**example**

失败的*Example* 。

DocTestFailure.**got**

用例的实际输出。

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)

一个由*DocTestRunner* 触发的异常，以提示一个 doctest 用例引发了一个意外的异常。构造函数参数被用来初始化相同名称的属性。

UnexpectedException 定义了以下属性:

UnexpectedException.**test**

当该用例失败时正在运行的*DocTest* 对象。

UnexpectedException.**example**

失败的*Example* 。

UnexpectedException.**exc_info**

一个包含意外异常信息的元组，由 sys.ex_info() 返回。

27.4.9 肥皂盒

正如介绍中提到的，*doctest* 已经发展到有三个主要用途:

1. 检查 docstring 中的用例。
2. 回归测试。
3. 可执行的文档/文字测试。

这些用途有不同的要求，区分它们是很重要的。特别是，用晦涩难懂的测试用例来填充你的文档字符串会使文档变得很糟糕。

在编写文档串时，要谨慎地选择文档串的用例。这是一门需要学习的艺术——一开始可能并不自然。用例应该为文档增加真正的价值。一个好的用例往往可以抵得上许多文字。如果用心去做，这些用例对你的用户来说是非常有价值的，而且随着时间的推移和事情的变化，收集这些用例所花费的时间也会得到很多倍的回报。我仍然惊讶于我的*doctest*用例在一个“无害”的变化后停止工作。

Doctest 也是回归测试的一个很好的工具，特别是如果你不吝啬解释的文字。通过交错的文本和用例，可以更容易地跟踪真正的测试，以及为什么。当测试失败时，好的文本可以使你更容易弄清问题所在，以及如何解决。的确，你可以在基于代码的测试中写大量的注释，但很少有程序员这样做。许多人发现，使用 doctest 方法反而能使测试更加清晰。也许这只是因为 doctest 使写散文比写代码容易一些，而在代码中写注释则有点困难。我认为它比这更深入：当写一个基于 doctest 的测试时，自然的态度是你想解释你的软件的细微之处，并用例子来说明它们。这反过来又自然地导致了测试文件从最简单的功能开始，然后逻辑地发展到复杂和边缘案例。一个连贯的叙述是结果，而不是一个孤立的函数集合，似乎是随机的测试孤立的功能位。这是一种不同的态度，产生不同的结果，模糊了测试和解释之间的区别。

回归测试最好限制在专用对象或文件中。有几种组织测试的选择：

- 编写包含测试案例的文本文件作为交互式例子，并使用*testfile()*或*DocFileSuite()*来测试这些文件。建议这样做，尽管对于新的项目来说是最容易做到的，从一开始就设计成使用 doctest 。
- 定义命名为 `_regrtest_topic` 的函数，由单个文档串组成，包含命名主题的测试用例。这些函数可以包含在与模块相同的文件中，或分离出来成为一个单独的测试文件。
- 定义一个从回归测试主题到包含测试用例的文档串的 `__test__` 字典映射。

当你把你的测试放在一个模块中时，这个模块本身就可以成为测试运行器。当一个测试失败时，你可以安排你的测试运行器只重新运行失败的测试，同时调试问题。下面是这样一个测试运行器的最小例子:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print(f"{fail} failures out of {total} tests")
```

备注

27.5 `unittest` --- 单元测试框架

源代码： Lib/unittest/__init__.py

（如果你已经对测试的概念比较熟悉了，你可能想直接跳转到这一部分*断言方法*。）

unittest 单元测试框架是受到 JUnit 的启发，与其他语言中的主流单元测试框架有着相似的风格。其支持测试自动化，配置共享和关机代码测试。支持将测试样例聚合到测试集中，并将测试与报告框架独立。

为了实现这些，*unittest* 通过面向对象的方式支持了一些重要的概念。

测试脚手架
: *test fixture* 表示为了开展一项或多项测试所需要进行的准备工作，以及所有相关的清理操作。举个例子，这可能包含创建临时或代理的数据库、目录，再或者启动一个服务器进程。

测试用例
一个测试用例是一个独立的测试单元。它检查输入特定的数据时的响应。*unittest* 提供一个基类：*TestCase*，用于新建测试用例。

测试套件
test suite 是一系列的测试用例，或测试套件，或两者皆有。它用于归档需要一起执行的测试。

测试运行器（test runner）
test runner 是一个用于执行和输出测试结果的组件。这个运行器可能使用图形接口、文本接口，或返回一个特定的值表示运行测试的结果。

参见

***doctest* --- 文档测试模块**
另一个风格完全不同的测试模块。

简单 Smalltalk 测试：便用模式
Kent Beck 有关使用*unittest* 所共享的模式的测试框架的原创论文。

pytest
第三方单元测试框架，提供轻量化的语法来编写测试，例如：`assert func(10) == 42`。

Python 测试工具分类
一个 Python 测试工具的详细列表，包含测试框架和模拟对象库。

Python 中的测试邮件列表
一个讨论 Python 中的测试和测试工具的特别兴趣小组。

Python 源代码分发包中的脚本 `Tools/unittestgui/unittestgui.py` 是一个用于发现和执行测试的 GUI 工具。这主要是为了方便单元测试的新手使用。在生产环境中推荐通过持续集成系统如 Buildbot, Jenkins, GitHub Actions 或 AppVeyor 来驱动测试过程。

27.5.1 基本实例

unittest 模块提供了一系列创建和运行测试的工具。这一段落演示了这些工具的一小部分，但也足以满足大部分用户的需求。

这是一段简短的代码，来测试三种字符串方法:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # 检查当分隔符不为字符串时 s.split 是否失败
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

继承*unittest.TestCase* 就创建了一个测试样例。上述三个独立的测试是三个类的方法，这些方法的命名都以 `test` 开头。这个命名约定告诉测试运行者类的哪些方法表示测试。

每个测试的关键是：调用*assertEqual()* 来检查预期的输出；调用*assertTrue()* 或*assertFalse()* 来验证一个条件；调用*assertRaises()* 来验证抛出了一个特定的异常。使用这些方法而不是 assert 语句是为了让测试运行者能聚合所有的测试结果并产生结果报告。

通过*setUp()* 和*tearDown()* 方法，可以设置测试开始前与完成后需要执行的指令。在*组织你的测试代码* 中，对此有更为详细的描述。

最后的代码块中，演示了运行测试的一个简单的方法。*unittest.main()* 提供了一个测试脚本的命令行接口。当在命令行运行该测试脚本，上文的脚本生成如以下格式的输出:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

在调用测试脚本时添加 -v 参数使*unittest.main()* 显示更为详细的信息，生成如以下形式的输出:

```
test_isupper (__main__.TestStringMethods.test_isupper) ... ok
test_split (__main__.TestStringMethods.test_split) ... ok
test_upper (__main__.TestStringMethods.test_upper) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

以上例子演示了*unittest* 中最常用的、足够满足许多日常测试需求的特性。文档的剩余部分详述该框架的完整特性。

在 3.11 版本发生变更: 从测试方法返回一个值（而非返回默认的 None 值）现在已被弃用。

27.5.2 命令行接口

unittest 模块可以通过命令行运行模块、类和独立测试方法的测试:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

你可以传入模块名、类或方法名或他们的任意组合。

同样的，测试模块可以通过文件路径指定:

```
python -m unittest tests/test_something.py
```

这样就可以使用 shell 的文件名补全指定测试模块。所指定的文件仍需要可以被作为模块导入。路径通过去除'.py' 、把分隔符转换为'.' 转换为模块名。若你需要执行不能被作为模块导入的测试文件，你需要直接执行该测试文件。

在运行测试时，你可以通过添加 -v 参数获取更详细（更多的冗余）的信息。

```
python -m unittest -v test_module
```

当运行时不包含参数，开始*探索性测试*

```
python -m unittest
```

用于获取命令行选项列表:

```
python -m unittest -h
```

在 3.2 版本发生变更: 在早期版本中，只支持运行独立的测试方法，而不支持模块和类。

命令行选项

unittest supports these command-line options:

-b, --buffer

在测试运行时，标准输出流与标准错误流会被放入缓冲区。成功的测试的运行时输出会被丢弃；测试不通过时，测试运行中的输出会正常显示，错误会被加入到测试失败信息。

-c, --catch

当测试正在运行时，Control-C 会等待当前测试完成，并在完成后报告已执行的测试的结果。当再次按下 Control-C 时，引发平常的*KeyboardInterrupt* 异常。

See *Signal Handling* for the functions that provide this functionality.

-f, --failfast

当出现第一个错误或者失败时，停止运行测试。

-k

只运行匹配模式或子字符串的测试方法和类。此选项可以被多次使用，在此情况下将会包括匹配任何给定模式的所有测试用例。

包含通配符（*）的模式使用*fnmatch.fnmatchcase()* 对测试名称进行匹配。另外，该匹配是大小写敏感的。

模式对测试加载器导入的测试方法全名进行匹配。

例如，-k foo 可以匹配到 foo_tests.SomeTest.test_something 和 bar_tests.SomeTest.test_foo，但是不能匹配到 bar_tests.FooTest.test_something。

--locals

在回溯中显示局部变量。

--durations N

显示 N 个最慢的测试用例 (N=0 表示全部)。

Added in version 3.2: 添加命令行选项 -b, -c 和 -f。

Added in version 3.5: 命令行选项 --locals。

Added in version 3.7: 命令行选项 -k。

Added in version 3.12: 命令行选项 --durations。

命令行亦可用于探索性测试，以运行一个项目的所有测试或其子集。

27.5.3 探索性测试

Added in version 3.2.

unittest 支持简单的测试发现。为了与测试发现兼容，所有测试文件都必须是可从项目的最高层级目录导入的 模块或 包 (这意味着它们的文件名必须是有效的 标识符)。

探索性测试在*TestLoader.discover()* 中实现，但也可以通过命令行使用。它在命令行中的基本用法如下：

```
cd project_directory
python -m unittest discover
```

备注

方便起见，python -m unittest 与 python -m unittest discover 等价。如果你需要向探索性测试传入参数，必须显式地使用 discover 子命令。

discover 有以下选项：

-v, --verbose
更详细地输出结果。

-s, --start-directory directory
开始进行搜索的目录 (默认值为当前目录 .)。

-p, --pattern pattern
用于匹配测试文件的模式（默认为 test*.py)。

-t, --top-level-directory directory
指定项目的最上层目录（通常为开始时所在目录)。

-s ， *-p* 和*-t* 选项可以按顺序作为位置参数传入。以下两条命令是等价的:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

正如可以传入路径那样，传入一个包名作为起始目录也是可行的，如 myproject.subpackage.test 。你提供的包名会被导入，它在文件系统中的位置会被作为起始目录。

小心

探索性测试通过导入测试对测试进行加载。在找到所有你指定的开始目录下的所有测试文件后，它把路径转换为包名并进行导入。如 foo/bar/baz.py 会被导入为 foo.bar.baz 。

如果你有一个全局安装的包，并尝试对这个包的副本进行探索性测试，可能会从错误的地方开始导入。如果出现这种情况，测试会输出警告并退出。

如果你使用包名而不是路径作为开始目录，搜索时会假定它导入的是你想要的目录，所以你不会收到警告。

测试模块和包可以通过*load_tests protocol* 自定义测试的加载和搜索。

在 3.4 版本发生变更: 测试发现支持初始目录下的*命名空间包*。注意你也需要指定顶层目录（例如: python -m unittest discover -s root/namespace -t root)。

在 3.11 版本发生变更: 在 Python 3.11 中*unittest* 丢弃了*命名空间包* 支持。它自 Python 3.7 就已不可用。包含测试的起始目录和子目录都必须是具有 __init__.py 文件的常规包。

包含起始目录的目录仍然可以是命名空间包。在此情况下，你需要以带点号的包名称来显式地指明起始目录和目标目录。例如:

```
# proj/  <-- current directory
#   namespace/
#     mypkg/
#       __init__.py
#       test_mypkg.py

python -m unittest discover -s namespace.mypkg -t .
```

27.5.4 组织你的测试代码

单元测试的构建单位是 *test cases* ： 独立的、包含执行条件与正确性检查的方案。在*unittest* 中，测试用例表示为*unittest.TestCase* 的实例。通过编写*TestCase* 的子类或使用*FunctionTestCase* 编写你自己的测试用例。

一个*TestCase* 实例的测试代码必须是完全自含的，因此它可以独立运行，或与其它任意组合任意数量的测试用例一起运行。

TestCase 的最简单的子类需要实现一个测试方法（例如一个命名以 test 开头的方法）以执行特定的测试代码:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

请注意为了进行测试，我们使用了 *TestCase* 基类提供的某个 *assert* 方法*。如果测试不通过，将会引发一个异常并附带解释性的消息，并且 *unittest* 会将这个测试用例标记为 *failure*。任何其它异常都将被视为 *errors*。

可能同时存在多个前置操作相同的测试，我们可以把测试的前置操作从测试代码中拆解出来，并实现测试前置方法 *setUp()* 。在运行测试时，测试框架会自动地为每个单独测试调用前置方法。

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

备注

多个测试运行的顺序由内置字符串排序方法对测试名进行排序的结果决定。

在测试运行时，若 *setUp()* 方法引发异常，测试框架会认为测试发生了错误，因此测试方法不会被运行。

相似的，我们提供了一个 *tearDown()* 方法在测试方法运行后进行清理工作。

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

若 *setUp()* 成功运行，无论测试方法是否成功，都会运行 *tearDown()* 。

这样的一个测试代码运行的环境被称为 *test fixture* 。一个新的 TestCase 实例作为一个测试脚手架，用于运行各个独立的测试方法。在运行每个测试时，*setUp()* 、*tearDown()* 和 `__init__()` 会被调用一次。

推荐你根据用例所测试的功能将测试用 TestCase 分组。*unittest* 为此提供了 *test suite*：*unittest* 的 *TestSuite* 类是一个代表。通常情况下，调用 *unittest.main()* 就能正确地找到并执行这个模块下所有用 TestCase 分组的测试。

然而，如果你需要自定义你的测试套件的话，你可以参考以下方法组织你的测试：

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
```

(续下页)

(接上页)

```
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

你可以把测试用例和测试套件放在与被测试代码相同的模块中（比如 `widget.py`），但将测试代码放在单独的模块中（比如 `test_widget.py`）有几个优势。

- 测试模块可以从命令行被独立调用。
- 更容易在分发的代码中剥离测试代码。
- 降低没有好理由的情况下修改测试代码以通过测试的诱惑。
- 测试代码应比被测试代码更少地被修改。
- 被测试代码可以更容易地被重构。
- 对用 C 语言写成的模块无论如何都得单独写成一个模块，为什么不保持一致呢?
- 如果测试策略发生了改变，没有必要修改源代码。

27.5.5 复用已有的测试代码

一些用户希望直接使用*unittest* 运行已有的测试代码，而不需要把已有的每个测试函数转化为一个*TestCase* 的子类。

因此，*unittest* 提供*FunctionTestCase* 类。这个*TestCase* 的子类可用于打包已有的测试函数，并支持设置前置与后置函数。

假定有一个测试函数:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

可以创建等价的测试用例如下，其中前置和后置方法是可选的。

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

备注

用*FunctionTestCase* 可以快速将现有的测试转换成基于*unittest* 的测试，但不推荐你这样做。花点时间继承*TestCase* 会让以后重构测试无比轻松。

在某些情况下，现有的测试可能是用*doctest* 模块编写的。如果是这样，*doctest* 提供了一个 `DocTestSuite` 类，可以从现有基于*doctest* 的测试中自动构建*unittest.TestSuite* 用例。

27.5.6 跳过测试与预计的失败

Added in version 3.1.

Unittest 支持跳过单个或整组的测试用例。它还支持把测试标注成“预期失败”的测试。这些坏测试会失败，但不会算进*TestResult* 的失败里。

要跳过测试只需使用*skip() decorator* 或其附带条件的版本，在*setUp()* 内部使用*TestCase.skipTest()*，或是直接引发*SkipTest*。

跳过测试的基本用法如下:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # 测试其是否仅适用于特定的库版本。
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # Windows 专属的测试代码
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # 依赖于外部资源的测试代码
        pass
```

在🄴嗦模式下运行以上测试例子时，程序输出如下：

```
test_format (__main__.MyTestCase.test_format) ... skipped 'not supported in this library␣
↪version'
test_nothing (__main__.MyTestCase.test_nothing) ... skipped 'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase.test_maybe_skipped) ... skipped 'external resource␣
↪not available'
test_windows_support (__main__.MyTestCase.test_windows_support) ... skipped 'requires Windows
↪'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

跳过测试类的写法跟跳过测试方法的写法相似：

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() 也可以跳过测试。可以用于所需资源不可用的情况下跳过接下来的测试。

使用*expectedFailure()* 装饰器表明这个测试预计失败。：

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

你可以很容易地编写在测试时调用*skip()* 的装饰器作为自定义的跳过测试装饰器。下面这个装饰器会跳过测试，除非所传入的对象具有特定的属性：

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

以下的装饰器和异常实现了跳过测试和预期失败两种功能：

@unittest.**skip**(*reason*)

跳过被此装饰器装饰的测试。*reason* 为测试被跳过的原因。

@unittest.**skipIf**(*condition*, *reason*)

当 *condition* 为真时，跳过被装饰的测试。

@unittest.**skipUnless**(*condition*, *reason*)

跳过被装饰的测试，除非 *condition* 为真。

@unittest.**expectedFailure**

将测试标记为预期的失败或错误。如果测试失败或在测试函数自身（而非在某个 *test fixture* 方法）中出现错误则将认为是测试成功。如果测试通过，则将认为是测试失败。

exception unittest.**SkipTest**(*reason*)

引发此异常以跳过一个测试。

通常来说，你可以使用 *TestCase.skipTest()* 或其中一个跳过测试的装饰器实现跳过测试的功能，而不是直接引发此异常。

被跳过的测试的 *setUp()* 和 *tearDown()* 不会被运行。被跳过的类的 *setUpClass()* 和 *tearDownClass()* 不会被运行。被跳过的模组的 `setUpModule()` 和 `tearDownModule()` 不会被运行。

27.5.7 使用子测试区分测试迭代

Added in version 3.4.

当你的几个测试之间的差异非常小，例如只有某些形参不同时，unittest 允许你使用 *subTest()* 上下文管理器在一个测试方法体的内部区分它们。

例如，以下测试:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

可以得到以下输出:

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=1)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=3)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0
```

(续下页)

(接上页)

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=5)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0
```

如果不使用子测试，程序遇到第一次错误之后就会停止。而且因为 `i` 的值不显示，错误也更难找。

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

27.5.8 类与函数

本节深入介绍了*unittest* 的 API。

测试用例

class unittest.**TestCase**(*methodName='runTest'*)

TestCase 类的实例代表了*unittest* 宇宙中的逻辑测试单元。该类旨在被当作基类使用，特定的测试将由其实体子类来实现。该类实现了测试运行器所需的接口以允许它驱动测试，并实现了可被测试代码用来检测和报告各种类型的失败的方法。

每个*TestCase* 实例将运行一个单位的基础方法：即名为 *methodName* 的方法。在使用*TestCase* 的大多数场景中，你都不需要修改 *methodName* 或重新实现默认的 `runTest()` 方法。

在 3.2 版本发生变更: *TestCase* 不需要提供 *methodName* 即可成功实例化。这使得从交互式解释器试验*TestCase* 更为容易。

TestCase 的实例提供了三组方法：一组用来运行测试，另一组被测试实现用来检查条件和报告失败，还有一些查询方法用来收集有关测试本身的信息。

第一组（用于运行测试的）方法是：

setUp()

为测试预备而调用的方法。此方法会在调用测试方法之前被调用；除了*AssertionError* 或*SkipTest*，此方法所引发的任何异常都将被视为错误而非测试失败。默认的实现将不做任何事情。

tearDown()

在测试方法被调用并记录结果之后立即被调用的方法。此方法即使在测试方法引发异常时仍会被调用，因此子类中的实现将需要特别注意检查内部状态。除*AssertionError* 或*SkipTest* 外，此方法所引发的任何异常都将被视为额外的错误而非测试失败（因而会增加总计错误报告数）。此方法将只在*setUp()* 成功执行时被调用，无论测试方法的结果如何。默认的实现将不做任何事情。

setUpClass()

在一个单独类中的测试运行之前被调用的类方法。`setUpClass` 会被作为唯一的参数在类上调用且必须使用*classmethod()* 装饰器:

```
@classmethod
def setUpClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

Added in version 3.2.

tearDownClass()

在一个单独类的测试完成运行之后被调用的类方法。tearDownClass 会被作为唯一的参数在类上调用且必须使用*classmethod()* 装饰器:

```
@classmethod
def tearDownClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

Added in version 3.2.

run(*result=None*)

运行测试，将结果收集至作为 *result* 传入的*TestResult*。如果 *result* 被省略或为 None，则会创建一个临时的结果对象（通过调用*defaultTestResult()* 方法）并使用它。结果对象会被返回给*run()* 的调用方。

同样的效果也可通过简单地调用*TestCase* 实例来达成。

在 3.3 版本发生变更: 之前版本的 run 不会返回结果。也不会对实例执行调用。

skipTest(*reason*)

在测试方法或*setUp()* 执行期间调用此方法将跳过当前测试。详情参见跳过测试与预计的失败。

Added in version 3.1.

subTest(*msg=None*, ***params*)

返回一个上下文管理器以将其中的代码块作为子测试来执行。可选的 *msg* 和 *params* 是将在子测试失败时显示的任意值，以便让你能清楚地标识它们。

一个测试用例可以包含任意数量的子测试声明，并且它们可以任意地嵌套。

查看使用子测试区分测试迭代 获取更详细的信息。

Added in version 3.4.

debug()

运行测试而不收集结果。这允许测试所引发的异常被传递给调用方，并可被用于支持在调试器中运行测试。

TestCase 类提供了一些断言方法用于检查并报告失败。下表列出了最常用的方法（请查看下文的其他表来了解更多的断言方法）：

方法	检查对象	引入版本
assertEqual(a, b)	a == b	
assertNotEqual(a, b)	a != b	
assertTrue(x)	bool(x) is True	
assertFalse(x)	bool(x) is False	
assertIs(a, b)	a is b	3.1
assertIsNot(a, b)	a is not b	3.1
assertIsNone(x)	x is None	3.1
assertIsNotNone(x)	x is not None	3.1
assertIn(a, b)	a in b	3.1
assertNotIn(a, b)	a not in b	3.1
assertIsInstance(a, b)	isinstance(a, b)	3.2
assertNotIsInstance(a, b)	not isinstance(a, b)	3.2

这些断言方法都支持 *msg* 参数，如果指定了该参数，它将被用作测试失败时的错误消息 (另请参阅*longMessage*)。请注意将 *msg* 关键字参数传给*assertRaises()*, *assertRaisesRegex()*, *assertWarns()*, *assertWarnsRegex()* 的前提是它们必须被用作上下文管理器。

assertEqual(*first*, *second*, *msg=None*)

测试 *first* 和 *second* 是否相等。如果两个值的比较结果是不相等，则测试将失败。

此外，如果 *first* 和 *second* 的类型完全相同且属于 list, tuple, dict, set, frozenset 或 str 或者属于通过*addTypeEqualityFunc()* 注册子类的类型则将会调用类型专属的相等判断函数以便生成更有用的默认错误消息 (另请参阅类型专属方法列表)。

在 3.1 版本发生变更: 增加了对类型专属的相等判断函数的自动调用。

在 3.2 版本发生变更: 增加了*assertMultiLineEqual()* 作为用于比较字符串的默认类型相等判断函数。

assertNotEqual(*first*, *second*, *msg=None*)

测试 *first* 和 *second* 是否不等。如果两个值的比较结果是相等，则测试将失败。

assertTrue(*expr*, *msg=None*)

assertFalse(*expr*, *msg=None*)

测试 *expr* 是否为真值（或假值）。

请注意这等价于 `bool(expr) is True` 而不等价于 `expr is True` (后者要使用 `assertIs(expr, True)`)。当存在更专门的方法时也应避免使用此方法 (例如应使用 `assertEqual(a, b)` 而不是 `assertTrue(a == b)`)，因为它们在测试失败时会提供更有用的错误消息。

assertIs(*first*, *second*, *msg=None*)

assertIsNot(*first*, *second*, *msg=None*)

测试 *first* 和 *second* 是 (或不是) 同一个对象。

Added in version 3.1.

assertIsNone(*expr*, *msg=None*)

assertIsNotNone(*expr*, *msg=None*)

测试 *expr* 是 (或不是) `None`。

Added in version 3.1.

assertIn(*member*, *container*, *msg=None*)

assertNotIn(*member*, *container*, *msg=None*)

测试 *member* 是 (或不是) *container* 的成员。

Added in version 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)

assertNotIsInstance(*obj*, *cls*, *msg=None*)

测试 *obj* 是 (或不是) *cls* (此参数可以为一个类或包含类的元组，即*isinstance()* 所接受的参数) 的实例。要检测是否为指定类型，请使用*assertIs(type(obj), cls)*。

Added in version 3.2.

还可以使用下列方法来检查异常、警告和日志消息的产生:

方法	检查对象	引入版本
*assertRaises(exc, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发了 *exc*	
*assertRaisesRegex(exc, r, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发了 *exc* 并且消息可与正则表达式 *r* 相匹配	3.1
*assertWarns(warn, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发了 *warn*	3.2
*assertWarnsRegex(warn, r, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发了 *warn* 并且消息可与正则表达式 *r* 相匹配	3.2
assertLogs(logger, level)	`with` 代码块在 *logger* 上使用了最小的 *level* 级别写入日志	3.4
assertNoLogs(logger, level)	**`with` 代码块没有在** *logger* 上使用最小的 *level* 级别写入日志	3.10

assertRaises(*exception*, *callable*, **args*, ***kwds*)

assertRaises(*exception*, ***, *msg=None*)

测试当 *callable* 附带任何同时被传给*assertRaises()* 的位置或关键字参数被调用时是否引发了异常。如果引发了 *exception* 则测试通过，如果引发了另一个异常则报错，或者如果未引发任何异常则测试失败。要捕获一组异常中的任何一个，可以将包含多个异常类的元组作为 *exception* 传入。

如果只给出了 *exception* 和可能的 *msg* 参数，则返回一个上下文管理器以便被测试的代码可以被写成内联形式而不是被写成函数:

```
with self.assertRaises(SomeException):
    do_something()
```

当被作为上下文管理器使用时，*assertRaises()* 接受额外的关键字参数 *msg*。

上下文管理器将把捕获的异常对象存入在其 `exception` 属性中。这适用于需要对所引发异常执行额外检查的场合:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

在 3.1 版本发生变更: 添加了将*assertRaises()* 用作上下文管理器的功能。

在 3.2 版本发生变更: 增加了 `exception` 属性。

在 3.3 版本发生变更: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)

assertRaisesRegex(*exception*, *regex*, ***, *msg=None*)

与*assertRaises()* 类似但还会测试 *regex* 是否匹配被引发异常的字符串表示形式。*regex* 可以是一个正则表达式对象或包含正则表达式的字符串以提供给*re.search()* 使用。例如:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

或者:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

Added in version 3.1: 以方法名 assertRaisesRegexp 添加。

在 3.2 版本发生变更: 重命名为*assertRaisesRegex()*。

在 3.3 版本发生变更: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertWarns(*warning*, *callable*, **args*, ***kwds*)

assertWarns(*warning*, ***, *msg=None*)

测试当 *callable* 附带任何同时被传给*assertWarns()* 的位置或关键字参数被调用时是否触发了警告。如果触发了 *warning* 则测试通过，否则测试失败。引发任何异常则报错。要捕获一组警告中的任何一个，可将包含多个警告类的元组作为 *warnings* 传入。

如果只给出了 *warning* 和可能的 *msg* 参数，则返回一个上下文管理器以便被测试的代码可以被写成内联形式而不是被写成函数:

```
with self.assertWarns(SomeWarning):
    do_something()
```

当被作为上下文管理器使用时，*assertWarns()* 接受额外的关键字参数 *msg*。

上下文管理器将把捕获的警告对象保存在其 warning 属性中，并把触发警告的源代码行保存在 filename 和 lineno 属性中。这适用于需要对捕获的警告执行额外检查的场合:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

无论被调用时警告过滤器是否就位此方法均可工作。

Added in version 3.2.

在 3.3 版本发生变更: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)

assertWarnsRegex(*warning*, *regex*, ***, *msg=None*)

与*assertWarns()* 类似但还会测试 *regex* 是否匹配被触发警告的消息文本。*regex* 可以是一个正则表达式对象或包含正则表达式的字符串以提供给*re.search()* 使用。例如:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

或者:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Added in version 3.2.

在 3.3 版本发生变更: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertLogs(*logger=None*, *level=None*)

一个上下文管理器，它测试在 *logger* 或其子对象上是否至少记录了一条至少为指定 *level* 以上级别的消息。

如果给出了 *logger* 则它应为一个*logging.Logger* 对象或为一个指定日志记录器名称的*str*。默认为根日志记录器，它将捕获未被非传播型后继日志记录器所拦阻的所有消息。

如果给出了 *level*，它应为一个用数字表示的日志记录级别或其字符串等价形式 (例如为 "ERROR" 或*logging.ERROR*)。默认为*logging.INFO*。

如果在 with 代码块内部发出了至少一条与 *logger* 和 *level* 条件相匹配的消息则测试通过，否则测试失败。

上下文管理器返回的对象是一个记录辅助器，它会记录所匹配的日志消息。它有两个属性:

records

所匹配的日志消息*logging.LogRecord* 对象组成的列表。

output

由*str* 对象组成的列表，内容为所匹配消息经格式化后的输出。

示例:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

Added in version 3.4.

assertNoLogs(*logger=None*, *level=None*)

一个上下文管理器，它测试在 *logger* 或其子对象上是否未记录任何至少为指定 *level* 以上级别的消息。

如果给出了 *logger* 则它应为一个*logging.Logger* 对象或为一个指定日志记录器名称的*str*。默认为根日志记录器，它将捕获所有消息。

如果给出了 *level*，它应为一个用数字表示的日志记录级别或其字符串等价形式 (例如为 "ERROR" 或*logging.ERROR*)。默认为*logging.INFO*。

与*assertLogs()* 不同，上下文管理器将不返回任何对象。

Added in version 3.10.

还有其他一些方法可用于执行更专门的检查，例如:

方法	检查对象	引入版本
assertAlmostEqual(a, b)	round(a-b, 7) == 0	
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0	
assertGreater(a, b)	a > b	3.1
assertGreaterEqual(a, b)	a >= b	3.1
assertLess(a, b)	a < b	3.1
assertLessEqual(a, b)	a <= b	3.1
assertRegex(s, r)	r.search(s)	3.1
assertNotRegex(s, r)	not r.search(s)	3.2
assertCountEqual(a, b)	*a* 和 *b* 具有同样数量的相同元素，无论其顺序如何。	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

测试 *first* 与 *second* 是否几乎相等，比较的标准是计算差值并舍入到 *places* 所指定的十进制位数 (默认为 7 位)，再与零相比较。请注意此方法是将结果值舍入到指定的 十进制位数 (即相当于*round()* 函数) 而非 有效位数。

如果提供了 *delta* 而非 *places* 则 *first* 和 *second* 之间的差值必须小于等于 (或大于) *delta*。

同时提供 *delta* 和 *places* 将引发*TypeError*。

在 3.2 版本发生变更: *assertAlmostEqual()* 会自动将几乎相等的对象视为相等。而如果对象相等则*assertNotAlmostEqual()* 会自动测试失败。增加了 *delta* 关键字参数。

assertGreater(*first*, *second*, *msg=None*)

assertGreaterEqual(*first*, *second*, *msg=None*)

assertLess(*first*, *second*, *msg=None*)

assertLessEqual(*first*, *second*, *msg=None*)

根据方法名分别测试 *first* 是否 >, >=, < 或 <= *second*。如果不是，则测试失败:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

Added in version 3.1.

assertRegex(*text*, *regex*, *msg=None*)

assertNotRegex(*text*, *regex*, *msg=None*)

测试一个 *regex* 搜索匹配（或不匹配）文本。如果不匹配，错误信息中将包含匹配模式和 文本 * (或部分匹配失败的 * 文本)。*regex* 可以是正则表达式对象或能够用于*re.search()* 的包含正则表达式的字符串。

Added in version 3.1: 以方法名 `assertRegexpMatches` 添加。

在 3.2 版本发生变更: 方法 `assertRegexpMatches()` 已被改名为*assertRegex()*。

Added in version 3.2: *assertNotRegex()*

assertCountEqual(*first*, *second*, *msg=None*)

测试序列 *first* 与 *second* 是否包含同样的元素，无论其顺序如何。当存在差异时，将生成一条错误消息来列出两个序列之间的差异。

重复的元素 不会在 *first* 和 *second* 的比较中被忽略。它会检查每个元素在两个序列中的出现次数是否相同。等价于: `assertEqual(Counter(list(first)), Counter(list(second)))` 但还适用于包含不可哈希对象的序列。

Added in version 3.2.

assertEqual() 方法会将相同类型对象的相等性检查分派给不同的类型专属方法。这些方法已被大多数内置类型所实现，但也可以使用*addTypeEqualityFunc()* 来注册新的方法:

addTypeEqualityFunc(*typeobj*, *function*)

注册一个由*assertEqual()* 调用的特定类型专属方法来检查恰好为相同 *typeobj* (而非子类) 的两个对象是否相等。*function* 必须接受两个位置参数和第三个 msg=None 关键字参数，就像*assertEqual()* 那样。当检测到前两个形参之间不相等时它必须引发*self.failureException(msg)* -- 可能还会提供有用的信息并在错误消息中详细解释不相等的原因。

Added in version 3.1.

以下是*assertEqual()* 自动选用的不同类型的比较方法。一般情况下不需要直接在测试中调用这些方法。

方法	用作比较	引入版本
assertMultiLineEqual(a, b)	字符串	3.1
assertSequenceEqual(a, b)	序列	3.1
assertListEqual(a, b)	列表	3.1
assertTupleEqual(a, b)	元组	3.1
assertSetEqual(a, b)	集合	3.1
assertDictEqual(a, b)	字典	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

测试多行字符串 *first* 是否与字符串 *second* 相等。当不相等时将在错误消息中包括两个字符串之间差异的高亮显示。此方法会在通过*assertEqual()* 进行字符串比较时默认被使用。

Added in version 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

测试两个序列是否相等。如果提供了 *seq_type*，则 *first* 和 *second* 都必须为 *seq_type* 的实例否则将引发失败。如果两个序列不相等则会构造一个错误消息来显示两者之间的差异。

此方法不会被*assertEqual()* 直接调用，但它会被用于实现*assertListEqual()* 和*assertTupleEqual()*。

Added in version 3.1.

assertListEqual(*first*, *second*, *msg=None*)

assertTupleEqual(*first*, *second*, *msg=None*)

测试两个列表或元组是否相等。如果不相等，则会构造一个错误消息来显示两者之间的差异。如果某个形参的类型不正确也会引发错误。这些方法会在通过*assertEqual()* 进行列表或元组比较时默认被使用。

Added in version 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

测试两个集合是否相等。如果不相等，则会构造一个错误消息来列出两者之间的差异。此方法会在通过*assertEqual()* 进行集合或冻结集合比较时默认被使用。

如果 *first* 或 *second* 没有 `set.difference()` 方法则测试失败。

Added in version 3.1.

assertDictEqual(*first*, *second*, *msg=None*)

测试两个字典是否相等。如果不相等，则会构造一个错误消息来显示两个字典的差异。此方法会在对*assertEqual()* 的调用中默认被用来进行字典的比较。

Added in version 3.1.

最后*TestCase* 还提供了以下的方法和属性:

fail(*msg=None*)

无条件地发出测试失败消息，附带错误消息 *msg* 或 `None`。

failureException

这个类属性给出测试方法所引发的异常。如果某个测试框架需要使用专门的异常，并可能附带额外的信息，则必须子类化该类以便与框架“正常互动”。这个属性的初始值为*AssertionError*。

longMessage

这个类属性决定当将一个自定义失败消息作为 msg 参数传给一个失败的 assertXYY 调用时会发生什么。默认值为 `True`。在此情况下，自定义消息会被添加到标准失败消息的末尾。当设为 `False` 时，自定义消息会替换标准消息。

类设置可以通过在调用断言方法之前将一个实例属性 self.longMessage 赋值为 `True` 或 `False` 在单个测试方法中进行重载。

类设置会在每个测试调用之前被重置。

Added in version 3.1.

maxDiff

这个属性控制来自在测试失败时报告 diffs 的断言方法的 diffs 输出的最大长度。它默认为 80*8 个字符。这个属性所影响的断言方法有*assertSequenceEqual()* (包括所有委托给它的序列比较方法), *assertDictEqual()* 以及*assertMultiLineEqual()*。

将 `maxDiff` 设为 `None` 表示不限制 diffs 的最大长度。

Added in version 3.2.

测试框架可使用下列方法来收集测试的有关信息:

countTestCases()

返回此测试对象所提供的测试数量。对于*TestCase* 实例，该数量将总是为 `1`。

defaultTestResult()

返回此测试类所要使用的测试结果类的实例（如果未向*run()* 方法提供其他结果实例）。

对于*TestCase* 实例，该返回值将总是为*TestResult* 的实例；*TestCase* 的子类应当在有必要时重写此方法。

id()

返回一个标识指定测试用例的字符串。该返回值通常为测试方法的完整名称，包括模块名和类名。

shortDescription()

返回测试的描述，如果未提供描述则返回 None。此方法的默认实现将在可用的情况下返回测试方法的文档字符串的第一行，或者返回 None。

在 3.1 版本发生变更: 在 3.1 中已修改此方法将测试名称添加到简短描述中，即使存在文档字符串。这导致了与单元测试扩展的兼容性问题因而在 Python 3.2 中将添加测试名称操作改到*TextTestResult* 中。

addCleanup(*function*, /, **args*, ***kwargs*)

在*tearDown()* 之后添加了一个要调用的函数来清理测试期间所使用的资源。函数将按它们被添加的相反顺序被调用 (LIFO)。它们在调用时将附带它们被添加时传给*addCleanup()* 的任何参数和关键字参数。

如果*setUp()* 失败，即意味着*tearDown()* 未被调用，则已添加的任何清理函数仍将被调用。

Added in version 3.1.

enterContext(*cm*)

进入所提供的*context manager*。如果成功，还会将其 __exit__() 方法作为使用*addCleanup()* 的清理函数并返回 __enter__() 方法的结果。

Added in version 3.11.

doCleanups()

此方法会在*tearDown()* 之后，或者如果*setUp()* 引发了异常则会在*setUp()* 之后被调用。

它将负责调用由*addCleanup()* 添加的所有清理函数。如果你需要在*tearDown()* 之前调用清理函数则可以自行调用*doCleanups()*。

doCleanups() 每次会弹出清理函数栈中的一个方法，因此它可以在任何时候被调用。

Added in version 3.1.

classmethod addClassCleanup(*function*, /, **args*, ***kwargs*)

在 Add a function to be called after *tearDownClass()* 之后添加了一个要调用的函数来清理测试类运行期间所使用的资源。函数将按它们被添加的相反顺序被调用 (LIFO)。它们在调用时将附带它们被添加时传给*addClassCleanup()* 的任何参数和关键字参数。

如果*setUpClass()* 失败，即意味着*tearDownClass()* 未被调用，则已添加的任何清理函数仍将被调用。

Added in version 3.8.

classmethod enterClassContext(*cm*)

进入所提供的*context manager*。如果成功，还会将其 __exit__() 方法作为使用*addClassCleanup()* 的清理函数并返回 __enter__() 方法的结果。

Added in version 3.11.

classmethod doClassCleanups()

此方法会在*tearDownClass()* 之后无条件地被调用，或者如果*setUpClass()* 引发了异常则会在*setUpClass()* 之后被调用。

它将负责访问由*addClassCleanup()* 添加的所有清理函数。如果你需要在*tearDownClass()* 之前调用清理函数则可以自行调用*doClassCleanups()*。

doClassCleanups() 每次会弹出清理函数栈中的一个方法，因此它在任何时候被调用。

Added in version 3.8.

class unittest.**IsolatedAsyncioTestCase**(*methodName='runTest'*)

这个类提供了与*TestCase* 类似的 API 并也接受协程作为测试函数。

Added in version 3.8.

loop_factory

传给*asyncio.Runner* 的 *loop_factory*。通过*asyncio.EventLoop* 重写子类以避免使用 asyncio 策略系统。

Added in version 3.13.

async asyncSetUp()

为测试预备而调用的方法。此方法会在 setUp() 之后被调用。此方法将在调用测试方法之前立即被调用；除了*AssertionError* 或*SkipTest*，此方法所引发的任何异常都将被视为错误而非测试失败。默认的实现将不做任何事情。

async asyncTearDown()

在测试方法被调用并记录结果之后立即被调用的方法。此方法会在 tearDown() 之前被调用。此方法即使在测试方法引发异常时仍会被调用，因此子类中的实现将需要特别注意检查内部状态。除*AssertionError* 或*SkipTest* 外，此方法所引发的任何异常都将被视为额外的错误而非测试失败（因而会增加总计错误报告数）。此方法将只在*asyncSetUp()* 成功执行时被调用，无论测试方法的结果如何。默认的实现将不做任何事情。

addAsyncCleanup(*function*, */*, **args*, ***kwargs*)

此方法接受一个可被用作清理函数的协程。

async enterAsyncContext(*cm*)

进入所提供的*asynchronous context manager*。如果成功，还会将其 __aexit__() 方法作为使用*addAsyncCleanup()* 的清理函数并返回 __aenter__() 方法的结果。

Added in version 3.11.

run(*result=None*)

设置一个新的事件循环来运行测试，将结果收集至作为 *result* 传入的*TestResult*。如果 *result* 被省略或为 None，则会创建一个临时的结果对象（通过调用 defaultTestResult() 方法）并使用它。结果对象会被返回给*run()* 的调用方。在测试结束时事件循环中的所有任务都将被取消。

一个显示先后顺序的例子:

```
from unittest import IsolatedAsyncioTestCase

events = []


class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)
```

(续下页)

(接上页)

```
    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")

if __name__ == "__main__":
    unittest.main()
```

在运行测试之后，events 将会包含 ["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]。

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

这个类实现了*TestCase* 的部分接口，允许测试运行方驱动测试，但不提供可被测试代码用来检查和报告错误的方法。这个类被用于创建使用传统测试代码的测试用例，允许它被集成到基于*unittest* 的测试框架中。

分组测试

class unittest.**TestSuite**(*tests=()*)

这个类代表对单独测试用例和测试套件的聚合。这个类提供给测试运行方所需的接口以允许其像任何其他测试用例一样运行。运行一个*TestSuite* 实例与对套件执行迭代来逐一运行每个测试的效果相同。

如果给出了 *tests*，则它必须是一个包含单独测试用例的可迭代对象或是将被用于初始构建测试套件的其他测试套件。还有一些附加的方法会被提供用来在随后向测试集添加测试用例和测试套件。

TestSuite 对象的行为与*TestCase* 对象很相似，区别在于它们并不会真正实现一个测试。它们会被用来将测试聚合为多个要同时运行的测试分组。还有一些附加的方法会被用来向*TestSuite* 实例添加测试:

addTest(*test*)

向测试套件添加*TestCase* 或*TestSuite*。

addTests(*tests*)

将来自包含*TestCase* 和*TestSuite* 实例的可迭代对象的所有测试添加到这个测试套件。

这等价于对 *tests* 进行迭代，并为其中的每个元素调用*addTest()*。

TestSuite 与*TestCase* 共享下列方法:

run(*result*)

运行与这个套件相关联的测试，将结果收集到作为 *result* 传入的测试结果对象中。请注意与*TestCase.run()* 的区别，*TestSuite.run()* 必须传入结果对象。

debug()

运行与这个套件相关联的测试而不收集结果。这允许测试所引发的异常被传递给调用方并可被用于支持在调试器中运行测试。

countTestCases()

返回此测试对象所提供的测试数量，包括单独的测试和子套件。

__iter__()

由*TestSuite* 分组的测试总是可以通过迭代来访问。其子类可以通过重载 __iter__() 来惰性地提供测试。请注意此方法可在单个套件上多次被调用（例如在计数或相等性比较时），为此在*TestSuite.run()* 之前重复迭代所返回的测试对于每次调用迭代都必须相同。在*TestSuite.run()* 之后，调用方不应继续访问此方法所返回的测试，除非调用方使用重载了 TestSuite._removeTestAtIndex() 的子类来保留对测试的引用。

在 3.2 版本发生变更: 在较早的版本中 *TestSuite* 会直接访问测试而不是通过迭代，因此只重载 `__iter__()` 并不足以提供所有测试。

在 3.4 版本发生变更: 在较早的版本中 *TestSuite* 会在 *TestSuite.run()* 之后保留对每个 *TestCase* 的引用。其子类可以通过重载 `TestSuite._removeTestAtIndex()` 来恢复此行为。

在 *TestSuite* 对象的典型应用中，*run()* 方法是由 `TestRunner` 唤起而不是由最终用户测试来控制。

加载和运行测试

class unittest.**TestLoader**

TestLoader 类可被用来基于类和模块创建测试套件。通常，没有必要创建该类的实例；*unittest* 模块提供了一个可作为 *unittest.defaultTestLoader* 共享的实例。但是，使用子类或实例允许对某些配置属性进行定制。

TestLoader 对象具有下列属性:

errors

由在加载测试期间遇到的非致命错误组成的列表。在任何时候都不会被加载方重围。致命错误是通过相关方法引发一个异常来向调用方发出信号的。非致命错误也是由一个将在运行时引发原始错误的合成测试来提示的。

Added in version 3.5.

TestLoader 对象具有下列方法:

loadTestsFromTestCase(*testCaseClass*)

返回一个包含在 *TestCase* 所派生的 `testCaseClass` 中的所有测试用例的测试套件。

会为每个由 *getTestCaseNames()* 指明的方法创建一个测试用例实例。在默认情况下这些都是以 `test` 开头的方法名称。如果 *getTestCaseNames()* 不返回任何方法，但 `runTest()` 方法已被实现，则会为该方法创建一个单独的测试用例。

loadTestsFromModule(*module*, *, *pattern=None*)

返回包含在给定模块中的所有测试用例的测试套件。此方法会在 *module* 中搜索从派生自 *TestCase* 的类并为该类定义的每个测试方法创建一个类实例。

> **备注**
>
> 虽然使用 *TestCase* 所派生的类的层级结构可以方便地共享配置和辅助函数，但在不打算直接实例化的基类上定义测试方法并不能很好地配合此方法使用。不过，当配置有差异并且定义在子类当中时这样做还是有用处的。

如果一个模块提供了 `load_tests` 函数则它将被调用以加载测试。这允许模块自行定制测试加载过程。这就称为 *load_tests protocol*。*pattern* 参数会被作为传给 `load_tests` 的第三个参数。

在 3.2 版本发生变更: 添加了对 `load_tests` 的支持。

在 3.5 版本发生变更: 增加了对仅限关键字参数 *pattern* 的支持。

在 3.12 版本发生变更: 未写入文档的非正式 *use_load_tests* 形参已被移除。

loadTestsFromName(*name*, *module=None*)

返回由给出了字符串形式规格描述的所有测试用例组成的测试套件。

描述名称 *name* 是一个“带点号的名称”，它可以被解析为一个模块、一个测试用例类、一个测试用例类内部的测试方法、一个 *TestSuite* 实例，或者一个返回 *TestCase* 或 *TestSuite* 实例的可调用对象。这些检查将按在此列出的顺序执行；也就是说，一个可能的测试用例类上的方法将作为“一个测试用例内部的测试方法”而非作为“一个可调用对象”被选定。

举例来说，如果你有一个模块 `SampleTests`，其中包含一个派生自 *TestCase* 的类 `SampleTestCase`，其中包含三个测试方法 (`test_one()`, `test_two()` 和 `test_three()`)。则

描述名称 'SampleTests.SampleTestCase' 将使此方法返回一个测试套件，它将运行全部三个测试方法。使用描述名称 'SampleTests.SampleTestCase.test_two' 将使它返回一个测试套件，它将仅运行 test_two() 测试方法。描述名称可以指向尚未被导入的模块和包；它们将作为附带影响被导入。

本模块可以选择相对于给定的 *module* 来解析 *name*。

在 3.5 版本发生变更: 如果在遍历 *name* 时发生了 *ImportError* 或 *AttributeError* 则在运行时引发该错误的合成测试将被返回。这些错误被包括在由 self.errors 所积累的错误中。

loadTestsFromNames(*names*, *module=None*)

类似于 *loadTestsFromName()*，但是接受一个名称序列而不是单个名称。返回值是一个测试套件，它支持为每个名称所定义的所有测试。

getTestCaseNames(*testCaseClass*)

返回由 *testCaseClass* 中找到的方法名称组成的已排序的序列；这应当是 *TestCase* 的一个子类。

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

通过从指定的开始目录向其子目录递归来找出所有测试模块，并返回一个包含该结果的 TestSuite 对象。只有与 *pattern* 匹配的测试文件才会被加载。（使用 shell 风格的模式匹配。）只有可导入的模块名称（即有效的 Python 标识符）将会被加载。

所有测试模块都必须可以从项目的最高层级上导入。如果起始目录不是最高层级则必须单独指明 *top_level_dir*。

如果导入某个模块失败，比如因为存在语法错误，则会将其记录为单独的错误并将继续查找模块。如果导入失败是因为引发了 *SkipTest*，则会将其记录为跳过而不是错误。

如果找到了一个包(即包含名为 __init__.py 的文件的目录)，则将在包中查找 load_tests 函数。如果存在此函数则将对其执行调用 package.load_tests(loader, tests, pattern)。测试发现操作会确保在执行期间仅检查测试一次，即使 load_tests 函数本身调用了 loader.discover 也是如此。.

如果 load_tests 存在则发现操作 不会对包执行递归处理，load_tests 将负责加载包中的所有测试。is responsible for loading all tests in the package.

该模式故意不被保存为加载器属性以使得包可以继续发自其自身。

top_level_dir 是在内部保存的，并被用作任何对 discover() 的嵌套调用的默认值。也就是说，如果一个包的 load_tests 调用了 loader.discover()，则无需传递此参数。

start_dir 可以是一个带点号的名称或是一个目录。

Added in version 3.2.

在 3.4 版本发生变更: 在导入时引发 *SkipTest* 的模块会被记录为跳过，而不是错误。

在 3.4 版本发生变更: *start_dir* 可以是一个命名空间包。

在 3.4 版本发生变更: 路径在被导入之前会先被排序以使得执行顺序保持一致，即使下层文件系统的顺序不是取决于文件名的。

在 3.5 版本发生变更: 现在 load_tests 会检查已找到的包，无论它们的路径是否与 *pattern* 匹配，因为包名称是无法与默认的模式匹配的。

在 3.11 版本发生变更: *start_dir* 不可以为命名空间包。它自 Python 3.7 开始已不可用而 Python 3.11 正式将其移除。

在 3.13 版本发生变更: *top_level_dir* 仅会在 *discover* 调用期间被保存。

TestLoader 的下列属性可通过子类化或在实例上赋值来配置:

testMethodPrefix

给出将被解读为测试方法的方法名称的前缀的字符串。默认值为 'test'。

这会影响 *getTestCaseNames()* 以及所有 loadTestsFrom* 方法。

sortTestMethodsUsing

将被用来在*getTestCaseNames()* 以及所有 loadTestsFrom* 方法中比较方法名称以便对它们进行排序。

suiteClass

根据一个测试列表来构造测试套件的可调用对象。不需要结果对象上的任何方法。默认值为*TestSuite* 类。

这会影响所有 loadTestsFrom* 方法。

testNamePatterns

由 Unix shell 风格通配符的测试名称模式组成的列表，供测试方法进行匹配以包括在测试套件中（参见 -k 选项）。

如果该属性不为 None (默认值)，则将要包括在测试套件中的所有测试方法都必须匹配该列表中的某个模式。请注意匹配总是使用*fnmatch.fnmatchcase()*，因此不同于传给 -k 选项的模式，简单的子字符串模式将必须使用 * 通配符来进行转换。

这会影响所有 loadTestsFrom* 方法。

Added in version 3.7.

class unittest.**TestResult**

这个类被用于编译有关哪些测试执行成功而哪些失败的信息。

存放一组测试的结果的*TestResult* 对象。*TestCase* 和*TestSuite* 类将确保结果被正确地记录；测试创建者无须担心如何记录测试的结果。

建立在*unittest* 之上的测试框架可能会想要访问通过运行一组测试所产生的*TestResult* 对象用来报告信息；TestRunner.run() 方法是出于这个目的而返回*TestResult* 实例的。

TestResult 实例具有下列属性，在检查运行一组测试的结果的时候很有用处。

errors

一个包含*TestCase* 实例和保存了格式化回溯信息的字符串 2 元组的列表。每个元组代表一个引发了非预期的异常的测试。

failures

一个包含*TestCase* 实例和保存了格式化回溯信息的字符串的 2 元组的列表。每个元素代表一个使用*assert* 方法* 显式地发出失败信号的测试。

skipped

一个包含 2-tuples of *TestCase* 实例和保存了跳过测试原因的字符串 2 元组的列表。

Added in version 3.1.

expectedFailures

一个包含*TestCase* 实例和保存了格式化回溯信息的 2 元组的列表。每个元组代表测试用例的一个已预期的失败或错误。

unexpectedSuccesses

一个包含被标记为已预期失败，但却测试成功的*TestCase* 实例的列表。

collectedDurations

一个包含测试用例名称和代表所运行的每个测试所用时间的浮点数 2 元组的列表。

Added in version 3.12.

shouldStop

当测试的执行应当被*stop()* 停止时则设为 True。

testsRun

目前已运行的测试的总数量。

buffer

如果设为真值，sys.stdout 和 sys.stderr 将在*startTest()* 和*stopTest()* 被调用之间被缓冲。被收集的输出将仅在测试失败或发生错误时才会被回显到真正的 sys.stdout 和 sys.stderr。任何输出还会被附加到失败/错误消息中。

Added in version 3.2.

failfast

如果设为真值则*stop()* 将在首次失败或错误时被调用，停止测试运行。

Added in version 3.2.

tb_locals

如果设为真值则局部变量将被显示在回溯信息中。

Added in version 3.5.

wasSuccessful()

如果当前所有测试都已通过则返回 True，否则返回 False。

在 3.4 版本发生变更: 如果有任何来自测试的*unexpectedSuccesses* 被*expectedFailure()* 装饰器所标记则返回 False。

stop()

此方法可被调用以提示正在运行的测试集要将*shouldStop* 属性设为 True 来表示其应当被中止。TestRunner 对象应当认同此旗标并返回而不再运行任何额外的测试。

例如，该特性会被*TextTestRunner* 类用来在当用户从键盘发出一个中断信号时停止测试框架。提供了 TestRunner 实现的交互式工具也可通过类似方式来使用该特性。

TestResult 类的下列方法被用于维护内部数据结构，并可在子类中被扩展以支持额外的报告需求。这特别适用于构建支持在运行测试时提供交互式报告的工具。

startTest(*test*)

当测试用例 *test* 即将运行时被调用。

stopTest(*test*)

在测试用例 *test* 已经执行后被调用，无论其结果如何。

startTestRun()

在任何测试被执行之前被调用一次。

Added in version 3.1.

stopTestRun()

在所有测试被执行之后被调用一次。

Added in version 3.1.

addError(*test*, *err*)

当测试用例 *test* 引发了非预期的异常时将被调用。*err* 是一个元组，其形式与*sys.exc_info()* 的返回值相同: (type, value, traceback)。

默认实现会将一个元组 (test, formatted_err) 添加到实例的*errors* 属性，其中 *formatted_err* 是派生自 *err* 的已格式化回溯信息。

addFailure(*test*, *err*)

当测试用例 *test* 发出了失败信号时将被调用。*err* 是一个元组，其形式与*sys.exc_info()* 的返回值相同: (type, value, traceback)。

默认实现会将一个元组 (test, formatted_err) 添加到实例的*failures* 属性，其中 *formatted_err* 是派生自 *err* 的已格式化回溯信息。

addSuccess(*test*)

当测试用例 *test* 成功时被调用。

默认实现将不做任何操作。

addSkip(*test*, *reason*)

当测试用例 *test* 被跳过时将被调用。*reason* 是给出的跳过测试的理由。

默认实现会将一个元组 `(test, reason)` 添加到实例的*skipped* 属性。

addExpectedFailure(*test*, *err*)

当测试用例 *test* 失败或发生错误，但是使用了*expectedFailure()* 装饰器来标记时将被调用。

默认实现会将一个元组 `(test, formatted_err)` 添加到实例的*expectedFailures* 属性，其中 *formatted_err* 是派生自 *err* 的已格式化回溯信息。

addUnexpectedSuccess(*test*)

当测试用例 *test* 使用了 was marked with the *expectedFailure()* 装饰器来标记，但是却执行成功时将被调用。

默认实现会将该测试添加到实例的*unexpectedSuccesses* 属性。

addSubTest(*test*, *subtest*, *outcome*)

当一个子测试结束时将被调用。*test* 是对应于该测试方法的测试用例。*subtest* 是一个描述该子测试的*TestCase* 实例。

如果 *outcome* 为*None*，则该子测试执行成功。否则，它将失败并引发一个异常，*outcome* 是一个元组，其形式与*sys.exc_info()* 的返回值相同: `(type, value, traceback)`。

默认实现在测试结果为成功时将不做任何事，并会将子测试的失败记录为普通的失败。

Added in version 3.4.

addDuration(*test*, *elapsed*)

在测试用例结束时被调用。*elapsed* 是以秒数表示的时间，并且它包括执行清理函数的时间。

Added in version 3.12.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*, *, *durations=None*)

供*TextTestRunner* 使用的*TestResult* 的具体实现。子类应当接受 `**kwargs` 以确保在接口改变时的兼容性。

Added in version 3.2.

在 3.12 版本发生变更: 增加了 *durations* 关键字形参。

unittest.**defaultTestLoader**

用于分享的*TestLoader* 类实例。如果不需要自制*TestLoader*，则可以使用该实例而不必重复创建新的实例。

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, *, *tb_locals=False*, *durations=None*)

一个将结果输出到流的基本测试运行器。如果 *stream* 为默认的 `None`，则会使用*sys.stderr* 作为输出流。这个类具有一些配置形参，但实际上都非常简单。运行测试套件的图形化应用程序应当提供替代实现。这样的实现应当在添加新特性到 unittest 时接受 `**kwargs` 作为修改构造运行器的接口。

在默认情况下该运行器将显示*DeprecationWarning*，*PendingDeprecationWarning*，*ResourceWarning* 和*ImportWarning* 即使它们*默认会被忽略*。此行为可使用 Python 的 `-Wd` 或 `-Wa` 选项并将 *warnings* 保持为 `None` 来覆盖 (参见 警告控制)。

在 3.2 版本发生变更: 增加了 *warnings* 形参。

在 3.2 版本发生变更: 默认流会在实例化而不是在导入时被设为*sys.stderr*。

在 3.5 版本发生变更: 增加了 *tb_locals* 形参。

在 3.12 版本发生变更: 增加了 *durations* 形参。

_makeResult()

此方法将返回由*run()* 使用的 TestResult 实例。它不应当被直接调用，但可在子类中被重载以提供自定义的 TestResult。

_makeResult() 会实例化传给 TextTestRunner 构造器的 resultclass 参数所指定的类或可迭代对象。如果没有提供 resultclass 则默认为*TextTestResult*。结果类会使用以下参数来实例化:

```
stream, descriptions, verbosity
```

run(*test*)

此方法是 TextTestRunner 的主要公共接口。此方法接受一个*TestSuite* 或*TestCase* 实例。通过调用*_makeResult()* 创建*TestResult* 来运行测试并将结果打印到标准输出。

unittest.**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

从 *module* 加载一组测试并运行它们的命令行程序；这主要是为了让测试模块能方便地执行。此函数的最简单用法是在测试脚本末尾包括下列行:

```
if __name__ == '__main__':
    unittest.main()
```

你可以通过传入冗余参数运行测试以获得更详细的信息:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

defaultTest 参数是要运行的单个测试名称，或者如果未通过 *argv* 指定任何测试名称则是包含多个测试名称的可迭代对象。如果未指定或为 None 且未通过 *argv* 指定任何测试名称，则会运行在 *module* 中找到的所有测试。

argv 参数可以是传给程序的选项列表，其中第一个元素是程序名。如未指定或为 None，则会使用*sys.argv* 的值。

testRunner 参数可以是一个测试运行器类或是其已创建的实例。在默认情况下 main 会调用*sys.exit()* 并附带一个退出码来指明测试运行是成功 (0) 还是失败 (1)。退出码为 5 表示没有运行或跳过任何测试。

testLoader 参数必须是一个*TestLoader* 实例，其默认值为*defaultTestLoader*。

main 支持通过传入 exit=False 参数以便在交互式解释器中使用。这将在标准输出中显示结果而不调用*sys.exit()*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

failfast, *catchbreak* 和 *buffer* 形参的效果与同名的*command-line options* 一致。

warnings 参数指定在运行测试时所应使用的警告过滤器。如果未指定，则默认的 None 会在将 -W 选项传给 **python** 命令时被保留 (参见 警告控制)，而在其他情况下将被设为 'default'。

调用 main 将返回一个带有 result 属性的对象，该属性包含*unittest.TestResult* 形式的测试运行结果。

在 3.1 版本发生变更: 增加了 *exit* 形参。

在 3.2 版本发生变更: 增加了 *verbosity*, *failfast*, *catchbreak*, *buffer* 和 *warnings* 形参。

在 3.4 版本发生变更: *defaultTest* 形参被修改为也接受一个由测试名称组成的迭代器。

load_tests 协议

Added in version 3.2.

模块或包可以通过实现一个名为 `load_tests` 的函数来定制在正常测试运行或测试发现期间要如何从中加载测试。

如果一个测试模块定义了 `load_tests` 则它将被*TestLoader.loadTestsFromModule()* 调用并传入下列参数:

```
load_tests(loader, standard_tests, pattern)
```

其中 *pattern* 会通过 `loadTestsFromModule` 传入。它的默认值为 `None`。

它应当返回一个*TestSuite*。

loader 是执行载入操作的*TestLoader* 实例。*standard_tests* 是默认要从该模块载入的测试。测试模块通常只需从标准测试集中添加或移除测试。第三个参数是在作为测试发现的一部分载入包时使用的。

一个从指定*TestCase* 类集合中载入测试的 `load_tests` 函数看起来可能是这样的:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

如果发现操作是在一个包含包的目录中开始的，不论是通过命令行还是通过调用*TestLoader.discover()*，则将在包 `__init__.py` 中检查 `load_tests`。如果不存在此函数，则发现将在包内部执行递归，就像它是另一个目录一样。在其他情况下，包中测试的发现操作将留给 `load_tests` 执行，它将附带下列参数被调用:

```
load_tests(loader, standard_tests, pattern)
```

这应当返回代表包中所有测试的*TestSuite*。(`standard_tests` 将只包含从 `__init__.py` 获取的测试。)

因为模式已被传入 `load_tests` 所以包可以自由地继续（还可能修改）测试发现操作。针对一个测试包的' 无操作' `load_tests` 函数看起来是这样的:

```
def load_tests(loader, standard_tests, pattern):
    # 在加载器实例上缓存的最高层级目录
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

在 3.5 版本发生变更: 发现操作不会再检查包名称是否匹配 *pattern*，因为包名称不可能匹配默认的模式。

27.5.9 类与模块设定

类与模块设定是在*TestSuite* 中实现的。当测试套件遇到来自新类的测试时则来自之前的类（如果存在）的 `tearDownClass()` 会被调用，然后再调用来自新类的 `setUpClass()`。

类似地如果测试是来自之前的测试的另一个模块则来自之前模块的 `tearDownModule` 将被运行，然后再运行来自新模块的 `setUpModule`。

在所有测试运行完毕后最终的 `tearDownClass` 和 `tearDownModule` 将被运行。

请注意共享设定不适用于一些 [潜在的] 特性例如测试并行化并且它们会破坏测试隔离。它们应当被谨慎地使用。

由 unittest 测试加载器创建的测试的默认顺序是将所有来自相同模块和类的测试归入相同分组。这将导致 setUpClass / setUpModule (等) 对于每个类和模块都恰好被调用一次。如果你将顺序随机化，以便使得来自不同模块和类的测试彼此相邻，那么这些共享的设定函数就可能会在一次测试运行中被多次调用。

共享的设定不适用与非标准顺序的套件。对于不想支持共享设定的框架来说 BaseTestSuite 仍然可用。

如果在共享的设定函数中引发了任何异常则测试将被报告错误。因为没有对应的测试实例，所以会创建一个 _ErrorHolder 对象（它具有与 *TestCase* 相同的接口）来代表该错误。如果你只是使用标准 unittest 测试运行器那么这个细节并不重要，但是如果你是一个框架开发者那么这可能会有关系。

setUpClass 和 tearDownClass

这些必须被实现为类方法:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

如果你希望在基类上的 setUpClass 和 tearDownClass 被调用则你必须自己去调用它们。在 *TestCase* 中的实现是空的。

如果在 setUpClass 中引发了异常则类中的测试将不会被运行并且 tearDownClass 也不会被运行。跳过的类中的 setUpClass 或 tearDownClass 将不会被运行。如果引发的异常是 *SkipTest* 异常则类将被报告为已跳过而非发生错误。

setUpModule 和 tearDownModule

这些应当被实现为函数:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

如果在 setUpModule 中引发了异常则模块中的任何测试都将不会被运行并且 tearDownModule 也不会被运行。如果引发的异常是 *SkipTest* 异常则模块将被报告为已跳过而非发生错误。

要添加即使在发生异常时也必须运行的清理代码，请使用 addModuleCleanup:

unittest.**addModuleCleanup**(*function*, */*, **args*, ***kwargs*)

在 tearDownModule() 之后添加一个要调用的函数来清理测试类运行期间所使用的资源。函数将按它们被添加的相反顺序被调用 (LIFO)。它们在调用时将附带它们被添加时传给 *addModuleCleanup()* 的任何参数和关键字参数。

如果 setUpModule() 失败，即意味着 tearDownModule() 未被调用，则已添加的任何清理函数仍将被调用。

Added in version 3.8.

classmethod unittest.**enterModuleContext**(*cm*)

进入所提供的 *context manager*。如果成功，还会将其 __exit__() 方法作为使用 *addModuleCleanup()* 的清理函数并返回 __enter__() 方法的结果。

Added in version 3.11.

unittest.**doModuleCleanups**()

此函数会在 tearDownModule() 之后无条件地被调用，或者如果 setUpModule() 引发了异常则会在 setUpModule() 之后被调用。

它将负责调用由 It is responsible for calling all the cleanup functions added by *addModuleCleanup()* 添加的所有清理函数。如果你需要在 tearDownModule() 之前调用清理函数则可以自行调用*doModuleCleanups()*。

doModuleCleanups() 每次会弹出清理函数栈中的一个方法，因此它可以在任何时候被调用。

Added in version 3.8.

27.5.10 信号处理

Added in version 3.2.

用于 unittest 的*-c/--catch* 命令行选项，加上传给*unittest.main()* 的 catchbreak 形参，提供了在测试运行期间处理 Ctrl-C 的更友好方式。在捕获中断行为被启用时 Ctrl-C 将允许当前运行的测试能够完成，而测试运行将随后结束并报告已有的全部结果。第二个 Ctrl-C 将会正常地引发*KeyboardInterrupt*。

处理 control-C 信号的处理器会尝试与安装了自定义*signal.SIGINT* 处理器的测试代码保持兼容。如果是 unittest 处理器而 不是已安装的*signal.SIGINT* 处理器被调用，即它被系统在测试的下层替换并委托处理，则它会调用默认的处理器。这通常会是替换了已安装处理器并委托处理的代码所预期的行为。对于需要禁用 unittest control-C 处理的单个测试则可以使用*removeHandler()* 装饰器。

还有一些工具函数让框架开发者可以在测试框架内部启用 control-C 处理功能。

unittest.**installHandler**()

安装 control-C 处理器。当接收到*signal.SIGINT* 时（通常是响应用户按下 control-C）所有已注册的结果都会执行*stop()* 调用。

unittest.**registerResult**(*result*)

注册一个*TestResult* 对象用于 control-C 的处理。注册一个结果将保存指向它的弱引用，因此这并不能防止结果被作为垃圾回收。

如果 control-C 未被启用则注册*TestResult* 对象将没有任何附带影响，因此不论是否启用了该项处理测试框架都可以无条件地注册他们独立创建的所有结果。

unittest.**removeResult**(*result*)

移除一个已注册的结果。一旦结果被移除则*stop()* 将不再会作为针对 control-C 的响应在结果对象上被调用。

unittest.**removeHandler**(*function=None*)

当不附带任何参数被调用时此函数将移除已被安装的 control-C 处理器。此函数还可被用作测试装饰器以在测试被执行时临时性地移除处理器:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

27.6 unittest.mock --- 模拟对象库

Added in version 3.3.

源代码： Lib/unittest/mock.py

---

unittest.mock 是一个用于测试的 Python 库。它允许使用模拟对象来替换受测系统的一些部分，并对这些部分如何被使用进行断言判断。

unittest.mock 提供的*Mock* 类，能在整个测试套件中模拟大量的方法。创建后，就可以断言调用了哪些方法/属性及其参数。还可以以常规方式指定返回值并设置所需的属性。

此外，mock 还提供了用于修补测试范围内模块和类级别属性的*patch()* 装饰器，和用于创建独特对象的*sentinel* 。阅读*quick guide* 中的案例了解如何使用*Mock* ，*MagicMock* 和*patch()* 。

mock 被设计为配合*unittest* 使用且它是基于'action -> assertion' 模式而非许多模拟框架所使用的'record -> replay' 模式。

对于较早版本的 Python 有一个反向移植的*unittest.mock*，即在 PyPI 上可用的 mock。

27.6.1 快速上手

当您访问对象时，*Mock* 和*MagicMock* 将创建所有属性和方法，并保存他们在使用时的细节。你可以通过配置，指定返回值或者限制可访问属性，然后断言他们如何被调用：

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

side_effect 允许你执行附带影响，包括在 mock 被调用时引发一个异常：

```
>>> from unittest.mock import Mock
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock 还可以通过其他方法配置和控制其行为。例如 mock 可以通过设置 *spec* 参数来从一个对象中获取其规格 (specification)。如果访问 mock 的属性或方法不在 spec 中，会报*AttributeError* 错误。

使用*patch()* 装饰去/上下文管理器，可以更方便地测试一个模块下的类或对象。你指定的对象会在测试过程中替换成 mock （或者其他对象），测试结束后恢复。

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

备注

当你嵌套 patch 装饰器时，mock 将以执行顺序传递给装饰器函数（*Python* 装饰器正常顺序）。由于从下至上，因此在上面的示例中，首先 mock 传入的 `module.ClassName1` 。

在查找对象的名称空间中修补对象使用*patch()* 。使用起来很简单，阅读*在哪里打补丁* 来快速上手。

patch() 也可以 with 语句中使用上下文管理。

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

还有一个*patch.dict()* 用于在一定范围内设置字典中的值，并在测试结束时将字典恢复为其原始状态：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock 支持 Python *魔术方法* 。使用模式方法最简单的方式是使用*MagicMock* class. 。它可以做如下事情：

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock 能指定函数（或其他 Mock 实例）为魔术方法，它们将被适当地调用。*MagicMock* 是预先创建了所有魔术方法（所有有用的方法）的 Mock 。

下面是一个使用了普通 Mock 类的魔术方法的例子

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

使用*auto-speccing* 可以保证测试中的模拟对象与要替换的对象具有相同的 api 。在 patch 中可以通过 *autospec* 参数实现自动推断，或者使用*create_autospec()* 函数。自动推断会创建一个与要替换对象相同的属相和方法的模拟对象，并且任何函数和方法（包括构造函数）都具有与真实对象相同的调用签名。

这么做是为了因确保不当地使用 mock 导致与生产代码相同的失败：

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: missing a required argument: 'b'
```

在类中使用*create_autospec()* 时，会复制 `__init__` 方法的签名，另外在可调用对象上使用时，会复制 `__call__` 方法的签名。

27.6.2 Mock 类

Mock 是一个可以灵活的替换存根 (stubs) 的对象，可以测试所有代码。Mock 是可调用的，在访问其属性时创建一个新的 mock[1] 。访问相同的属性只会返回相同的 mock 。Mock 会保存调用记录，可以通过断言获悉代码的调用。

MagicMock 是*Mock* 的子类，它有所有预创建且可使用的魔术方法。在需要模拟不可调用对象时，可以使用*NonCallableMock* 和*NonCallableMagicMock*

patch() 装饰器使得用*Mock* 对象临时替换特定模块中的类非常方便。默认情况下*patch()* 将为你创建一个*MagicMock*。你可以使用*patch()* 的 *new_callable* 参数指定替代*Mock* 的类。

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

创建一个新的*Mock* 对象。通过可选参数指定*Mock* 对象的行为：

- *spec*: 可以是要给字符串列表，也可以是充当模拟对象规范的现有对象（类或实例）。如果传入一个对象，则通过在该对象上调用 dir 来生成字符串列表（不支持的魔法属性和方法除外）。访问不在此列表中的任何属性都将触发*AttributeError* 。

  如果 *spec* 是一个对象（而不是字符串列表）则 `__class__` 将返回 spec 对象的类。这允许 mock 通过*isinstance()* 测试。

- *spec_set*： *spec* 的更严格的变体。如果使用了该属性，尝试模拟 *set* 或 *get* 的属性不在 *spec_set* 所包含的对象中时，会抛出*AttributeError* 。

- *side_effect*： 每当调用 Mock 时都会调用的函数。参见*side_effect* 属性。对于引发异常或动态更改返回值很有用。该函数使用与 mock 函数相同的参数调用，并且除非返回*DEFAULT* ，否则该函数的返回值将用作返回值。

  另外，*side_effect* 可以是异常类或实例。此时，调用模拟程序时将引发异常。

  如果 *side_effect* 是可迭代对象，则每次调用 mock 都将返回可迭代对象的下一个值。

  设置 *side_effect* 为 `None` 即可清空。

- *return_value*： 调用 mock 的返回值。默认情况下，是一个新的 Mock（在首次访问时创建）。参见*return_value* 属性。

- *unsafe*: 在默认情况下，访问任何名字以 *assert*, *assret*, *asert*, *aseert* 或 *assrt* 开头的属性都将引发*AttributeError*。传入 `unsafe=True` 将允许访问这些属性。

  Added in version 3.5.

- *wraps*： 要包装的 mock 对象。如果 *wraps* 不是 `None` ，那么调用 Mock 会将调用传递给 *wraps* 的对象（返回实际结果）。对模拟的属性访问将返回一个 Mock 对象，该对象包装了 *wraps* 对象的相应属性（因此，尝试访问不存在的属性将引发*AttributeError* ）。

  如果明确指定 *return_value* ，则调用时不会返回包装对象，而是返回 *return_value* 。

- *name*： mock 的名称。在调试时很有用。名称会传递到子 mock 。

还可以使用任意关键字参数来调用 mock 。创建模拟后，将使用这些属性来设置 mock 的属性。有关详细信息，请参见*configure_mock()* 方法。

assert_called()

断言 mock 已被调用至少一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Added in version 3.6.

[1] 仅有的例外是魔术方法和属性（其名称前后都带有双下划线）。Mock 不会创建它们而是将引发*AttributeError*。这是因为解释器将会经常隐式地请求这些方法，并且在它准备接受一个魔术方法却得到一个新的 Mock 对象时会 相当困惑。如果你需要魔术方法支持请参阅魔术方法。

assert_called_once()

断言 mock 已被调用恰好一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
Calls: [call(), call()].
```

Added in version 3.6.

assert_called_with(**args*, ***kwargs*)

此方法是断言上次调用已以特定方式进行的一种便捷方法：

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)

断言 mock 已被调用恰好一次，并且向该调用传入了指定的参数。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
Calls: [call('foo', bar='baz'), call('other', bar='values')].
```

assert_any_call(**args*, ***kwargs*)

断言 mock 已被调用并附带了指定的参数。

如果 mock 曾经被调用过则断言通过，不同于*assert_called_with()*和*assert_called_once_with()*那样只有在调用是最近的一次时才会通过，而对于*assert_called_once_with()*则它还必须是唯一的一次调用。

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)

断言 mock 已附带指定的参数被调用。将针对这些调用检查*mock_calls*列表。

如果 *any_order* 为假值则调用必须是顺序进行的。在指定的调用之前或之后还可以有额外的调用。

如果 *any_order* 为真值则调用可以是任意顺序的，但它们都必须在*mock_calls*中出现。

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
```

(续下页)

(接上页)

```
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

断言 mock 从未被调用过。

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
Calls: [call()].
```

Added in version 3.5.

reset_mock(*, *return_value=False*, *side_effect=False*)

reset_mock 方法将在 mock 对象上重围所有的调用属性:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

这在你希望执行重用相同对象的一系列断言的场合下将会很有用处。

return_value 形参在设为 `True` 时将重置*return_value*:

```
>>> mock = Mock(return_value=5)
>>> mock('hello')
5
>>> mock.reset_mock(return_value=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

side_effect 形参在设为 `True` 将重置*side_effect*:

```
>>> mock = Mock(side_effect=ValueError)
>>> mock('hello')
Traceback (most recent call last):
  ...
ValueError
>>> mock.reset_mock(side_effect=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

请注意*reset_mock()* 在默认情况下 不会清除*return_value*, *side_effect* 或任何你使用普通赋值设置的子属性。

子 mock 也会被重置。

在 3.6 版本发生变更: 向 reset_mock 函数添加了两个仅限关键字参数。

mock_add_spec(*spec*, *spec_set=False*)

为 mock 添加描述。*spec* 可以是一个对象或字符串列表。只有 *spec* 上的属性可以作为来自 mock 的属性被获取。

如果 *spec_set* 为真值则只有 spec 上的属性可以被设置。

attach_mock(*mock*, *attribute*)

附加一个 mock 作为这一个的属性，替换它的名称和上级。对附加 mock 的调用将记录在这一个的*method_calls* 和*mock_calls* 属性中。

configure_mock(**kwargs)

通过关键字参数在 mock 上设置属性。

属性加返回值和附带影响可以使用标准点号标记在子 mock 上设置并在方法调用中解包一个字典:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

同样的操作可在对 mock 的构造器调用中达成:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() 的存在是使得 mock 被创建之后的配置更为容易。

__dir__()

Mock 对象会将 `dir(some_mock)` 的结果限制为有用结果。对于带有 *spec* 的 mock 这还包括 mock 的所有被允许的属性。

请查看*FILTER_DIR* 了解此过滤做了什么，以及如何停用它。

_get_child_mock(**kw)

创建针对属性和返回值的子 mock。默认情况下子 mock 将为与其上级相同的类型。Mock 的子类可能需要重载它来定制子 mock 的创建方式。

对于非可调用的 mock 将会使用可调用的变化形式（而非不是任意的自定义子类）。

called

一个表示 mock 对象是否已被调用的布尔值:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

一个告诉你 mock 对象已被调用多少次的整数值:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

设置这个来配置通过调用该 mock 所返回的值:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

默认的返回值是一个 mock 对象并且你可以通过正常方式来配置它:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value 也可以在构造器中设置:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

这可以是当该 This can either be a function to be called when the mock 被调用时将被调用的一个函数，可调用对象或者要被引发的异常（类或实例）。

如果你传入一个函数则它将附带与该 mock 相同的参数被调用并且除了该函数返回*DEFAULT*单例的情况以外对该 mock 的调用都将随后返回该函数所返回的任何东西。如果该函数返回*DEFAULT* 则该 mock 将返回其正常值 (来自*return_value*)。

如果你传入一个可迭代对象，它会被用来获取一个在每次调用时必须产生一个值的迭代器。这个值可以是一个要被引发的异常实例，或是一个要从该调用返回给 mock 的值 (*DEFAULT* 处理与函数的情况一致)。

一个引发异常（来测试 API 的异常处理）的 mock 的示例:

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

使用*side_effect* 来返回包含多个值的序列:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

使用一个可调用对象:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect 可以在构造器中设置。下面是在 mock 被调用时增加一个该属性值并返回它的例子:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

将*side_effect* 设为 None 可以清除它:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

这可以是 None (如果 mock 没有被调用)，或是 mock 最近一次被调用时附带的参数。这将采用元组的形式：第一个成员也可以通过 args 特征属性来访问，它是 mock 被调用时所附带的任何位置参数 (或为空元组)，而第二个成员也可以通过 kwargs 特征属性来访问，它则是任何关键字参数 (或为空字典)。

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

call_args, 以及列表*call_args_list*, *method_calls* 和*mock_calls* 的成员都是*call* 对象。这些对象属性元组，因此它们可被解包以获得单独的参数并创建更复杂的断言。参见*作为元*

组的 *call*。

在 3.8 版本发生变更: 增加了 `args` 和 `kwargs` 特征属性。properties.

call_args_list

这是一个已排序的对 mock 对象的所有调用的列表（因此该列表的长度就是它已被调用的次数）。在执行任何调用之前它将是一个空列表。*call* 对象可以被用来方便地构造调用列表以与*call_args_list* 相比较。

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

call_args_list 的成员均为*call* 对象。它们可作为元组被解包以获得单个参数。参见*作为元组的 call*。

method_calls

除了会追踪对其自身的调用，mock 还会追踪对方法和属性，以及 它们的方法和属性的访问:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

method_calls 的成员均为*call* 对象。它们可以作为元组被解包以获得单个参数。参见*作为元组的 call*。

mock_calls

mock_calls 会记录 所有对 mock 对象、它的方法、魔术方法的调用 以及返回值的 mock。

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

mock_calls 的成员均为*call* 对象。它们可以作为元组被解包以获得单个参数。参见*作为元组的 call*。

备注

mock_calls 的记录方式意味着在进行嵌套调用时，之前调用的形参不会被记录因而这样的比较将总是相等:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

通常一个对象的 __class__ 属性将返回其类型。对于具有 spec 的 mock 对象，__class__ 将改为返回 spec 类。这允许 mock 为它们所替换 / 屏蔽的对象跳过*isinstance()* 测试：

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ 是可以被赋值的，这允许 mock 跳过*isinstance()* 检测而不强制要求你使用 spec：

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

不可调用的*Mock* 版本。其构造器的形参具有与*Mock* 相同的含义，区别在于 *return_value* 和 *side_effect* 在不可调用的 mock 上没有意义。

使用一个类或实例作为 spec 或 spec_set 的对象能够跳过*isinstance()* 测试：

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

Mock 类具有对 mock 操作魔术方法的支持。请参阅魔术方法 了解完整细节。

mock 操作类和*patch()* 装饰器都接受任意关键字参数用于配置。对于*patch()* 装饰器来说关键字参数会被传给所创建 mock 的构造器。这些关键字被用于配置 mock 的属性：

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

子 mock 的返回值和附带效果也可使用带点号的标记通过相同的方式来设置。由于你无法直接在调用中使用带点号的名称因此你需要创建一个字典并使用 ** 来解包它：

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

使用 *spec* (或 *spec_set*) 创建的可调用 mock 将在匹配调用与 mock 时内省规格说明对象的签名。因此，它可以匹配实际调用的参数而不必关心它们是按位置还是按名称传入的:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

这 适 用 于*assert_called_with()*， *assert_called_once_with()*， *assert_has_calls()* 和*assert_any_call()*。当执行*自动 spec* 时，它还将应用于 mock 对象的方法调用。

在 3.4 版本发生变更: 添加了在附带规格说明和自动规格说明的 mock 对象上的签名内省

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

旨在作为类的*property* 或其他*descriptor* 的 mock。*PropertyMock* 提供了 `__get__()` 和 `__set__()` 方法以便你可以在它被提取时指定一个返回值。

当从一个对象提取*PropertyMock* 实例时将不附带任何参数地调用该 mock。如果设置它则调用该 mock 时将附带被设置的值。

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

由于 mock 属性的存储方式你无法直接将*PropertyMock* 附加到一个 mock 对象。但是你可以将它附加到 mock 类型对象:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

小心

如果由*PropertyMock* 引发了*AttributeError*，它将被解读为缺少描述器并将在父 mock 上调用 `__getattr__()`:

```
>>> m = MagicMock()
>>> no_attribute = PropertyMock(side_effect=AttributeError)
>>> type(m).my_property = no_attribute
>>> m.my_property
<MagicMock name='mock.my_property' id='140165240345424'>
```

请参阅 __getattr__() 了解详情。

class unittest.mock.**AsyncMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

MagicMock 的异步版本。*AsyncMock* 对象的行为方式将使该对象被识别为异步函数，其调用的结果将为可等待对象。

```
>>> mock = AsyncMock()
>>> asyncio.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

调用 mock() 的结果是一个异步函数，它在被等待之后将具有 side_effect 或 return_value 的结果:

- 如果 side_effect 是一个函数，则异步函数将返回该函数的结果,
- 如果 side_effect 是一个异常，则异步函数将引发该异常,
- 如果 side_effect 是一个可迭代对象，则异步函数将返回该可迭代对象的下一个值，但是，如果结果序列被耗尽，则会立即引发 StopAsyncIteration,
- 如果 side_effect 未被定义，则异步函数将返回 is not defined, the async function will return the value defined by return_value 所定义的值，因而，在默认情况下，异步函数会返回一个新的*AsyncMock* 对象。

将*Mock* 或*MagicMock* 的 *spec* 设为异步函数将导致在调用后返回一个协程对象。

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

将*Mock*, *MagicMock* 或*AsyncMock* 的 *spec* 设为带有异步和同步函数的类将自动删除其中的同步函数并将它们设为*MagicMock* (如果上级 mock 是*AsyncMock* 或*MagicMock*) 或者*Mock* (如果上级 mock 是*Mock*)。所有异步函数都将为*AsyncMock*。

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

Added in version 3.8.

assert_awaited()

断言 mock 已被等待至少一次。请注意这是从被调用的对象中分离出来的，必须要使用 `await` 关键字:

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

断言 mock 已被等待恰好一次。

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

断言上一次等待传入了指定的参数。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected await not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

断言 mock 已被等待恰好一次并且附带了指定的参数。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
```

(续下页)

(接上页)

```
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(**args*, ***kwargs*)

断言 mock 已附带了指定的参数被等待。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls*, *any_order=False*)

断言 mock 已附带了指定的调用被等待。将针对这些等待检查*await_args_list* 列表。

如果 *any_order* 为假值则等待必须是顺序进行的。在指定的等待之前或之后还可以有额外的调用。

如果 *any_order* 为真值则等待可以是任意顺序的，但它们都必须在*await_args_list* 中出现。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()

断言 mock 从未被等待过。

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args*, ***kwargs*)

参见*Mock.reset_mock()*。还会将*await_count* 设为 0，将*await_args* 设为 None，并清空*await_args_list*。

await_count

一个追踪 mock 对象已被等待多少次的整数值。

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
```

(续下页)

(接上页)

```
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args

这可能为 None (如果 mock 从未被等待)，或为该 mock 上一次被等待所附带的参数。其功能与*Mock.call_args* 相同。

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list

这是由对 mock 对象按顺序执行的所有等待组成的列表（因此该列表的长度即它被等待的次数）。在有任何等待被执行之前它将为一个空列表。

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

class unittest.mock.**ThreadingMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, *, *timeout=UNSET*, ***kwargs*)

针对多线程测试的*MagicMock* 版本。*ThreadingMock* 对象提供了额外的方法用来等待调用被唤起，而不是立即对其执行断言。

默认的超时值由 timeout 参数指定，或者由*ThreadingMock.DEFAULT_TIMEOUT* 属性来重置，该属性默认为阻塞型 (None)。

你可以通过设置*ThreadingMock.DEFAULT_TIMEOUT* 来配置全局默认超时。

wait_until_called(*, *timeout=UNSET*)

等待直到 mock 被调用。

如果在创建 mock 时传入了 timeout 值或者如果向该函数传入了 timeout 参数，那么当调用未在时限内执行完毕则会引发*AssertionError*。

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock)
>>> thread.start()
>>> mock.wait_until_called(timeout=1)
>>> thread.join()
```

wait_until_any_call_with(*args, **kwargs)

等待直到该 mock 附带指定参数被调用。

如果在创建该 mock 时传入了 timeout 值则当调用未在时限内执行完成则会引发*AssertionError*。

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock, args=("arg1", "arg2",), kwargs={"arg":
↪"thing"})
>>> thread.start()
>>> mock.wait_until_any_call_with("arg1", "arg2", arg="thing")
>>> thread.join()
```

DEFAULT_TIMEOUT

创建*ThreadingMock*实例的全局默认超时秒数。

Added in version 3.13.

调用

Mock 对象是可调用对象。调用将把值集合作为*return_value*属性返回。默认的返回值是一个新的 Mock 对象；它会在对返回值的首次访问（不论是显式访问还是通过调用 Mock）时被创建——但它会被保存并且每次都返回相同的对象。

对该对象的调用将被记录在*call_args*和*call_args_list*等属性中。

如果设置了*side_effect*则它将在调用被记录之后被调用，因此如果 side_effect 引发了异常该调用仍然会被记录。

让一个 mock 在被调用时引发异常的最简单方式是将*side_effect*设为一个异常类或实例:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

如果*side_effect*是一个函数则该函数所返回的对象就是调用该 mock 所返回的对象。side_effect 函数在被调用时将附带与该 mock 相同的参数。这允许你根据输入动态地改变返回值:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

如果你想让该 mock 仍然返回默认的返回值（一个新的 mock 对象），或是任何设定的返回值，那么有两种方式可以做到这一点。从*side_effect*内部返回*return_value*，或者返回*DEFAULT*:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

要移除一个*side_effect*，并返回到默认行为，请将 side_effect 设为 None:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

side_effect 也可以是任意可迭代对象。对该 mock 的重复调用将返回来自该可迭代对象的值（直到该可迭代对象被耗尽并导致*StopIteration* 被引发）：

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

如果该可迭代对象有任何成员属于异常则它们将被引发而不是被返回:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

删除属性

Mock 对象会根据需要创建属性。这允许它们可以假装成任意类型的对象。

你可能想要一个 mock 对象在调用*hasattr()* 时返回 False，或者在获取某个属性时引发*AttributeError*。你可以通过提供一个对象作为 mock 的 spec 来做到这点，但这并不总是很方

便。

你可以通过删除属性来“屏蔽”它们。属性一旦被删除，访问它将引发*AttributeError*。

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock 的名称与 name 属性

由于”name” 是*Mock* 构造器的参数之一，如果你想让你的 mock 对象具有”name” 属性你不可以在创建时传入该参数。有两个替代方式。一个选项是使用*configure_mock()*:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

一个更简单的选项是在 mock 创建之后简单地设置”name” 属性:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

附加 Mock 作为属性

当你附加一个 mock 作为另一个 mock 的属性（或作为返回值）时它会成为该 mock 的” 子对象”。对子对象的调用会被记录在父对象的*method_calls* 和*mock_calls* 属性中。这适用于配置子 mock 然后将它们附加到父对象，或是将 mock 附加到将记录所有对子对象的调用的父对象上并允许你创建有关 mock 之间的调用顺序的断言:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

这里有一个例外情况是如果 mock 设置了名称。这允许你在出于某些理由不希望其发生时避免” 父对象” 的影响。

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

通过*patch()* 创建的 mock 会被自动赋予名称。要将具有名称的 mock 附加到父对象上你应当使用*attach_mock()* 方法:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

27.6.3 patch 装饰器

patch 装饰器仅被用于在它们所装饰的函数作用域内部为对象添加补丁。它们会自动为你执行去除补丁的处理，即使是在引发了异常的情况下。所有这些函数都还可在 with 语句中使用或是作为类装饰器。

patch

备注

问题的关键是要在正确的命名空间中打补丁。参见*where to patch* 一节。

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch() 可以作为函数装饰器、类装饰器或上下文管理器。在函数或 with 语句的内部，*target* 会打上一个 *new* 对象补丁。当函数/with 语句退出时补丁将被撤销。

如果 *new* 被省略，那么如果被打补丁的对象是一个异步函数则 target 将被替换为*AsyncMock* 否则替换为*MagicMock*。如果*patch()* 被用作装饰器并且 *new* 被省略，那么已创建的 mock 将作为一个附加参数传入被装饰的函数。如果*patch()* 被用作上下文管理器那么已创建的 mock 将被该上下文管理器所返回。

target 应当为 `'package.module.ClassName'` 形式的字符串。*target* 将被导入并且该指定对象会被替换为 *new* 对象，因此 *target* 必须是可以从你调用*patch()* 的环境中导入的。target 会在被装饰的函数被执行的时候被导入，而非在装饰的时候。

spec 和 *spec_set* 关键字参数会被传递给*MagicMock*，如果 patch 为你创建了此对象的话。

此外你还可以传入 `spec=True` 或 `spec_set=True`，这将使 patch 将被模拟的对象作为 spec/spec_set 对象传入。

new_callable 允许你指定一个不同的类，或者可调用对象，它将被调用以创建 *新的*对象。在默认情况下将指定*AsyncMock* 用于异步函数，*MagicMock* 用于其他函数。

另一种更强形式的 *spec* 是 *autospec*。如果你设置了 `autospec=True` 则将以来自被替换对象的 spec 来创建 mock。mock 的所有属性也将具有被替换对象相应属性的 spec。被模拟的方法和函数将检查它们的参数并且如果使用了错误的签名调用它们则将引发*TypeError*。对于替换了一个类的 mock，它们的返回值（即‘实例’）将具有与该类相同的 spec。请参阅*create_autospec()* 函数以及*自动 spec*。

除了 `autospec=True` 你还可以传入 `autospec=some_object` 以使用任意对象而不是被替换的对象作为 spec。

在默认情况下*patch()* 将无法替换不存在的属性。如果你传入 `create=True`，且该属性并不存在，则 patch 将在调用被打补丁的函数时为你创建该属性，并在退出被打补丁的函数时再次删除它。这适用于编写针对生产代码在运行时创建的属性的测试。它默认是被关闭的因为这具有危险性。当它被开启时你将能够针对实际上并不存在的 API 编写通过测试！

> **备注**
>
> 在 3.5 版本发生变更: 如果你要给某个模块的内置函数打补丁则不必传入 `create=True`，它默认就会被添加。

Patch 可以被用作*TestCase* 类装饰器。它是通过装饰类中的每个测试方法来发挥作用的。当你的测试方法共享同一个补丁集时这将减少模板代码。*patch()* 会通过查找以 `patch.TEST_PREFIX` 打头的名称来找到测试。其默认值为 `'test'`，这与*unittest* 找到测试的方式一致。你可以通过设置 `patch.TEST_PREFIX` 来指定其他的前缀。

Patch 可以通过 with 语句作为上下文管理器使用。这时补丁将应用于 with 语句的缩进代码块。如果你使用了"as" 则打补丁的对象将被绑定到"as" 之后的名称；这非常适用于当*patch()* 为你创建 mock 对象的情况。

patch() 可接受任意关键字参数。如果打补丁的对象是异步的则这些参数将被传给*AsyncMock*，否则传给*MagicMock*，或者是指定的 *new_callable*。

`patch.dict(...)`, `patch.multiple(...)` 和 `patch.object(...)` 可用于其他使用场景。

patch() 作为函数装饰器，为你创建 mock 并将其传入被装饰的函数:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

为类打补丁将把该类替换为*MagicMock* 的 *实例*。如果该类是在受测试的代码中被实例化的则它将为所要使用的 mock 的*return_value*。

如果该类被多次实例化则你可以使用*side_effect* 来每次返回一个新 mock。或者你也可以将 *return_value* 设为你希望的任何对象。

要在被打补丁的类的 *实例*的方法上配置返回值你必须在*return_value* 进行操作。例如:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

如果你使用 *spec* 或 *spec_set* 并且*patch()* 替换的是 *class*，那么所创建的 mock 的返回值将具有同样的 spec。

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

new_callable 参数适用于当你想要使用其他类来替代所创建的 mock 默认的*MagicMock* 的场合。例如，如果你想要使用*NonCallableMock*:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
```

(续下页)

(接上页)

```
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

另一个使用场景是用*io.StringIO*实例来替换某个对象:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

当*patch()*为你创建 mock 时，通常你需要做的第一件事就是配置该 mock。某些配置可以在对 patch 的调用中完成。你在调用时传入的任何关键字参数都将被用来在所创建的 mock 上设置属性:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

除了所创建的 mock 的属性上的属性，例如*return_value*和*side_effect*，还可以配置子 mock 的属性。将这些属性直接作为关键字参数传入在语义上是无效的，但是仍然能够使用 `**` 将以这些属性为键的字典扩展至一个*patch()*调用中

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

在默认情况下，尝试给某个模块中并不存在的函数（或者某个类中的方法或属性）打补丁将会失败并引发*AttributeError*:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing_attribute
↪'
```

但在对*patch()*的调用中添加 `create=True` 将使之前示例的效果符合预期:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
```

(续下页)

(接上页)

```
...
>>> test()
```

在 3.8 版本发生变更: 如果目标为异步函数那么*patch()* 现在将返回一个*AsyncMock*。

patch.object

patch.**object**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

用一个 mock 对象为对象 (*target*) 中指定名称的成员 (*attribute*) 打补丁。

patch.object() 可以被用作装饰器、类装饰器或上下文管理器。*new*, *spec*, *create*, *spec_set*, *autospec* 和 *new_callable* 等参数的含义与*patch()* 的相同。与*patch()* 类似，*patch.object()* 接受任意关键字参数用于配置它所创建的 mock 对象。

当用作类装饰器时*patch.object()* 将认可 patch.TEST_PREFIX 作为选择所要包装方法的标准。

你可以附带三个参数或两个参数来调用*patch.object()*。三个参数的形式将接受要打补丁的对象、属性的名称以及将要替换该属性的对象。

将附带两个参数的形式调用时你将省略替换对象，还会为你创建一个 mock 并作为附加参数传入被装饰的函数:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

传给*patch.object()* 的 *spec*, *create* 和其他参数的含义与*patch()* 的同名参数相同。

patch.dict

patch.**dict**(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

为一个字典或字典类对象打补丁，并在测试之后将该目录恢复到其初始状态。

in_dict 可以是一个字典或映射类容器。如果它是一个映射则它必须至少支持获取、设置和删除条目以及对键执行迭代。

in_dict 也可以是一个指定字典名称的字符串，然后将通过导入操作来获取该字典。

values 可以是一个要在字典中设置的值的字典。*values* 也可以是一个包含 (key, value) 对的可迭代对象。

如果 *clear* 为真值则该字典将在设置新值之前先被清空。

patch.dict() 也可以附带任意关键字参数调用以设置字典中的值。

在 3.8 版本发生变更: 现在当*patch.dict()* 被用作上下文管理器时将返回被打补丁的字典。now returns the patched dictionary when used as a context manager.

patch.dict() 可被用作上下文管理器、装饰器或类装饰器:

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
...
>>> test()
>>> assert foo == {}
```

当被用作类装饰器时*patch.dict()* 将认可 patch.TEST_PREFIX (默认值为 'test') 作为选择所在包装方法的标准:

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

如果你在为你的测试使用不同的前缀，你可以通过设置 `patch.TEST_PREFIX` 来将不同的前缀告知打补丁方。有关如何修改该值的详情请参阅*TEST_PREFIX*。

patch.dict() 可被用来向一个字典添加成员，或者简单地让测试修改一个字典，并确保当测试结束时恢复该字典。

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same dict)
...     patched_foo['spam'] = 'eggs'
...
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

可以在*patch.dict()* 调用中使用关键字来设置字典的值:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() 可被用于实际上不是字典的字典类对象。它们至少必须支持条目获取、设置、删除以及迭代或成员检测两者之一。这对应于魔术方法 `__getitem__()`, `__setitem__()`, `__delitem__()` 以及*__iter__()* 或 `__contains__()` 两者之一。

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
```

(续下页)

(接上页)

```
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

在单个调用中执行多重补丁。它接受要打补丁的对象（一个对象或一个通过导入来获取对象的字符串）以及用于补丁的关键字参数:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

如果你希望*patch.multiple()* 为你创建 mock 则要使用*DEFAULT* 作为值。在此情况下所创建的 mock 会通过关键字参数传入被装饰的函数，而当*patch.multiple()* 被用作上下文管理器时则将返回一个字典。

patch.multiple() 可以被用作装饰器、类装饰器或上下文管理器。*spec*, *spec_set*, *create*, *autospec* 和 *new_callable* 等参数的含义与*patch()* 的相同。这些参数将被应用到*patch.multiple()* 所打的 所有补丁。

当被用作类装饰器时*patch.multiple()* 将认可 patch.TEST_PREFIX 作为选择所要包装方法的标准。

如果你希望*patch.multiple()* 为你创建 mock，那么你可以使用*DEFAULT* 作为值。如果你使用*patch.multiple()* 作为装饰器则所创建的 mock 会作为关键字参数传入被装饰的函数。

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() 可以与其他 patch 装饰器嵌套使用，但要将作为关键字传入的参数要放在*patch()* 所创建的标准参数 之后:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

如果*patch.multiple()* 被用作上下文管理器，则上下文管理器的返回值将是一个以所创建的 mock 的名称为键的字典:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

补丁方法: start 和 stop

所有 patcher 对象都具有 `start()` 和 `stop()` 方法。使用这些方法可以更简单地在 `setUp` 方法上打补丁，还可以让你不必嵌套使用装饰器或 with 语句就能打多重补丁。

要使用这些方法请按正常方式调用*patch()*, *patch.object()* 或*patch.dict()* 并保留一个指向所返回 `patcher` 对象的引用。你可以随后调用 `start()` 来原地打补丁并调用 `stop()` 来恢复它。

如果你使用*patch()* 来创建自己的 mock 那么将可通过调用 `patcher.start` 来返回它。

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

此操作的一个典型应用场景是在一个*TestCase* 的 `setUp` 方法中执行多重补丁:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

小心

如果你要使用这个技巧则你必须通过调用 `stop` 来确保补丁被“恢复”。这可能要比你想像的更麻烦，因为如果在 `setUp` 中引发了异常那么 `tearDown` 将不会被调用。*unittest.TestCase.addCleanup()* 可以简化此操作:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

一项额外的好处是你不再需要保留指向 `patcher` 对象的引用。

还可以通过使用*patch.stopall()* 来停止已启动的所有补丁。

`patch.`**`stopall`**`()`

停止所有激活的补丁。仅会停止通过 `start` 启动的补丁。

为内置函数打补丁

你可以为一个模块中的任何内置函数打补丁。以以示例是为内置函数*ord()* 打补丁:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

所有补丁都可被用作类装饰器。当以这种方式使用时它们将会包装类中的每个测试方法。补丁会将以名字以 'test' 开头的方法识别为测试方法。这与*unittest.TestLoader* 查找测试方法的默认方式相同。

你可能会想要为你的测试使用不同的前缀。你可以通过设置 patch.TEST_PREFIX 来告知打补丁方不同的前缀:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

嵌套补丁装饰器

如果你想要应用多重补丁那么你可以简单地堆叠多个装饰器。

你可以使用以下模式来堆叠多个补丁装饰器:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

请注意装饰器是从下往上被应用的。这是 Python 应用装饰器的标准方式。被创建并传入你的测试函数的 mock 的顺序也将匹配这个顺序。

补丁的位置

patch() 通过（临时性地）修改某一个对象的 *名称* 指向另一个对象来发挥作用。可以有多个名称指向任意单独对象，因此要让补丁起作用你必须确保已为被测试的系统所使用的名称打上补丁。

基本原则是你要在对象 *被查找* 的地方打补丁，这不一定就是它被定义的地方。一组示例将有助于厘清这一点。

想像我们有一个想要测试的具有如下结构的项目:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

现在我们想要测试 `some_function` 但我们想使用*patch()* 来模拟 `SomeClass`。问题在于当我们导入模块 b 时，我们必须让它从模块 a 导入 `SomeClass`。如果我们使用*patch()* 来模拟 `a.SomeClass` 那么它将不会对我们的测试造成影响；模块 b 已经拥有对 *真正的* `SomeClass` 的血此看起来我们的补丁不会有任何影响。

关键在于对 `SomeClass` 打补丁操作是在它被使用（或它被查找）的地方。在此情况下实际上 `some_function` 将在模块 b 中查找 `SomeClass`，而我们已经在那里导入了它。补丁看上去应该是这样:

```
@patch('b.SomeClass')
```

但是，再考虑另一个场景，其中不是 `from a import SomeClass` 而是模块 b 执行了 `import a` 并且 `some_function` 使用了 `a.SomeClass`。这两个导入形式都很常见。在这种情况下我们要打补丁的类将在该模块中被查找因而我们必须改为对 `a.SomeClass` 打补丁:

```
@patch('a.SomeClass')
```

对描述器和代理对象打补丁

patch 和*patch.object* 都能正确地对描述器打补丁并恢复：包含类方法、静态方法和特征属性。你应当在 *类* 而不是实例上为它们打补丁。它们也适用于代理属性访问的 *部分对象*，例如 django settings object。

27.6.4 MagicMock 与魔术方法支持

模拟魔术方法

Mock 支持模拟 Python 协议方法，或称"*魔术方法*"。这允许 mock 对象替代容器或其他实现了 Python 协议的对象。

因为查找魔术方法的方式不同于普通方法[2]，这种支持采用了特别的实现。这意味着只有特定的魔术方法受到支持。受支持的是 *几乎所有* 魔术方法。如果有你需要但被遗漏的请告知我们。

你可以通过在某个函数或 mock 实例中设置魔术方法来模拟它们。如果你是使用函数则它 *必须* 接受 `self` 作为第一个参数[3]。

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

[2] 魔术方法 *应当* 是在类中而不是在实例中查找。不同的 Python 版本对这个规则的应用并不一致。受支持的协议方法应当适用于所有受支持的 Python 版本。

[3] 该函数基本上是与类挂钩的，但每个 `Mock` 实例都会与其他实例保持隔离。

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

一个这样的应用场景是在 `with` 语句中模拟作为上下文管理器的对象:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

对魔术方法的调用不会在*method_calls* 中出现，但它们会被记录在*mock_calls* 中。

备注

如果你使用 *spec* 关键字参数来创建 mock 那么尝试设置不包含在 spec 中的魔术方法将引发*AttributeError*。

受支持魔术方法的完整列表如下:

- `__hash__`, `__sizeof__`, `__repr__` 和 `__str__`
- `__dir__`, `__format__` 和 `__subclasses__`
- `__round__`, `__floor__`, `__trunc__` 和 `__ceil__`
- 比较运算: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` 和 `__ne__`
- 容器方法: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` 和 `__missing__`
- 上下文管理器: `__enter__`, `__exit__`, `__aenter__` 和 `__aexit__`
- 单目数值运算方法: `__neg__`, `__pos__` 和 `__invert__`
- 数值运算方法（包括右侧和原地两种形式）: `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__` 和 `__pow__`
- 数值转换方法: `__complex__`, `__int__`, `__float__` 和 `__index__`
- 描述器方法: `__get__`, `__set__` and `__delete__`
- 封存方法: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` 和 `__setstate__`
- 文件系统路径表示: `__fspath__`
- 异步迭代方法: `__aiter__` and `__anext__`

在 3.8 版本发生变更: 增加了对*os.PathLike.__fspath__()* 的支持。

在 3.8 版本发生变更: 增加了对 `__aenter__`, `__aexit__`, `__aiter__` 和 `__anext__` 的支持。

下列方法均存在但是 不受支持，因为它们或者被 mock 所使用，或者无法动态设置，或者可能导致问题:

- __getattr__, __setattr__, __init__ 和 __new__
- __prepare__, __instancecheck__, __subclasscheck__, __del__

MagicMock

存在两个版本的 MagicMock: *MagicMock* 和*NonCallableMagicMock*.

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock 是具有大部分*魔术方法* 的默认实现的*Mock* 的子类。你可以使用 MagicMock 而无法自行配置这些魔术方法。

构造器形参的含义与*Mock* 的相同。

如果你使用了 *spec* 或 *spec_set* 参数则将 只有存在于 spec 中的魔术方法会被创建。

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

MagicMock 的不可调用版本。

其构造器的形参具有与*MagicMock* 相同的含义，区别在于 *return_value* 和 *side_effect* 在不可调用的 mock 上没有意义。

魔术方法是通过*MagicMock* 对象来设置的，因此你可以用通常的方式来配置它们并使用它们:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

在默认情况下许多协议方法都需要返回特定类型的对象。这些方法都预先配置了默认的返回值，以便它们在你对返回值不感兴趣时可以不做任何事就能被使用。如果你想要修改默认值则你仍然可以手动 设置返回值。

方法及其默认返回值:

- __lt__: *NotImplemented*
- __gt__: NotImplemented
- __le__: NotImplemented
- __ge__: NotImplemented
- __int__: 1
- __contains__: False
- __len__: 0
- __iter__: iter([])
- __exit__: False
- __aexit__: False
- __complex__: 1j
- __float__: 1.0
- __bool__: True
- __index__: 1
- __hash__: mock 的默认 hash
- __str__: mock 的默认 str
- __sizeof__: mock 的默认 sizeof

例如:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

两个相等性方法 `__eq__()` 和 `__ne__()` 是特殊的。它们会基于标识号进行默认的相等性比较，使用*side_effect* 属性，除非你修改它们的返回值以返回其他内容:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

`MagicMock.__iter__()` 的返回值可以是任意可迭代对象而不要求必须是迭代器:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

如果返回值 是迭代器，则对其执行一次迭代就会将它耗尽因而后续执行的迭代将会输出空列表:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` 已配置了所有受支持的魔术方法，只有某些晦涩和过时的魔术方法是例外。如果你需要仍然可以设置它们。

在 `MagicMock` 中受到支持但默认未被设置的魔术方法有:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` 和 `__delete__`
- `__reversed__` 和 `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` 和 `__setstate__`
- `__getformat__`

27.6.5 辅助对象

sentinel

unittest.mock.**sentinel**

sentinel 对象提供了一种为你的测试提供独特对象的便捷方式。

属性是在你通过名称访问它们时按需创建的。访问相同的属性将始终返回相同的对象。返回的对象会有一个合理的 repr 以使测试失败消息易于理解。

在 3.7 版本发生变更: 现在 sentinel 属性会在它们被*copy* 或*pickle* 时保存其标识。

在测试时你可能需要测试是否有一个特定的对象作为参数被传给了另一个方法，或是被其返回。通常的做法是创建一个指定名称的 sentinel 对象来执行这种测试。*sentinel* 提供了一种创建和测试此类对象的标识的便捷方式。

在这个示例中我们为 method 打上便捷补丁以返回 sentinel.some_object:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> result
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**

DEFAULT 对象是一个预先创建的 sentinel (实际为 sentinel.DEFAULT)。它可被*side_effect* 函数用来指明其应当使用正常的返回值。

call

unittest.mock.**call**(**args*, ***kwargs*)

call() 是一个可创建更简单断言的辅助对象，用于同*call_args*, *call_args_list*, *mock_calls* 和*method_calls* 进行比较。*call()* 也可配合*assert_has_calls()* 使用。

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()

对于代表多个调用的 call 对象，*call_list()* 将返回一个包含所有中间调用以及最终调用的列表。

call_list 特别适用于创建针对“链式调用”的断言。链式调用是指在一行代码中执行的多个调用。这将使得一个 mock 的中存在多个条目*mock_calls*。手动构造调用的序列将会很烦琐。

call_list() 可以根据同一个链式调用构造包含多个调用的序列:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

根据构造方式的不同，call 对象可以是一个 (位置参数, 关键字参数) 或 (名称, 位置参数, 关键字参数) 元组。当你自行构造它们时这没有什么关系，但是*Mock.call_args*, *Mock.call_args_list* 和*Mock.mock_calls* 属性当中的 call 对象可以被反查以获取它们所包含的单个参数。

Mock.call_args 和*Mock.call_args_list* 中的 call 对象是 (位置参数, 关键字参数) 二元组而*Mock.mock_calls* 中以及你自行构造的 call 对象则是 (名称, 位置参数, 关键字参数) 三元组。

你可以使用它们作为“元组”的特性来为更复杂的内省和断言功能获取单个参数。位置参数是一个元组（如无位置参数则为空元组）而关键字参数是一个字典:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

使用另一对象作为 spec 来创建 mock 对象。mock 的属性将使用 *spec* 对象上的对应属性作为其 spec。

被模拟的函数或方法的参数将会被检查以确保它们是附带了正确的签名被调用的。

如果 *spec_set* 为 True 则尝试设置不存在于 spec 对象中的属性将引发*AttributeError*。

如果将类用作 spec 则 mock（该类的实例）的返回值将为这个 spec。你可以通过传入 instance=True 来将某个类用作一个实例对象的 spec。被返回的 mock 将仅在该 mock 的实例为可调用对象时才会是可调用对象。

create_autospec() 也接受被传入所创建 mock 的构造器的任意关键字参数。

请参阅*自动 spec* 来了解如何通过*create_autospec()* 以及*patch()* 的 *autospec* 参数来自动设置 spec。

在 3.8 版本发生变更: 如果目标为异步函数那么*create_autospec()* 现在将返回一个*AsyncMock*。

ANY

unittest.mock.**ANY**

有时你可能需要设置关于要模拟的调用中的某些参数的断言，但是又不想理会其他参数或者想要从*call_args* 中单独拿出它们并针对它们设置更复杂的断言。

为了忽略某些参数你可以传入与任意对象相等的对象。这样再调用*assert_called_with()* 和*assert_called_once_with()* 时无论传入什么参数都将执行成功。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY 也可被用在与*mock_calls* 这样的调用列表相比较的场合:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

ANY 并不仅限于同调用对象的比较而是也可被用于测试断言:

```
class TestStringMethods(unittest.TestCase):

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', ANY])
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR 是一个控制 mock 对象响应*dir()* 的方式的模块组变量。默认值为 `True`，这将使用下文所描述的过滤操作，以便只显示有用的成员。如果你不喜欢这样的过滤，或是出于诊断目的需要将其关闭，则应设置 `mock.FILTER_DIR = False`。

当启用过滤时，`dir(some_mock)` 将只显示有用的属性并将包括正常情况下不会被显示的任何动态创建的属性。如果 mock 是使用 *spec* (当然也可以是 *autospec*) 创建的则原有的所有属性都将被显示，即使它们还未被访问过:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

许多不太有用的（是*Mock* 的而不是被模拟对象的私有成员）开头带有下划线和双下划线的属性已从在*Mock* 上对*dir()* 的调用的结果中被过滤。如果你不喜欢此行为你可以通过设置模块级开关*FILTER_DIR* 来将其关闭:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

或者你也可以直接使用 `vars(my_mock)` (实例成员) 和 `dir(type(my_mock))` (类型成员) 来绕过不考虑*FILTER_DIR* 的过滤。

mock_open

unittest.mock.**mock_open**(*mock=None*, *read_data=None*)

创建 mock 来代替使用*open()* 的辅助函数。它对于*open()* 被直接调用或被用作上下文管理器的情况都适用。

mock 参数是要配置的 mock 对象。如为 `None` (默认值) 则将为你创建一个*MagicMock*，其 API 会被限制为可用于标准文件处理的方法或属性。

read_data 是供文件句柄的*read()*, *readline()* 和*readlines()* 方法返回的字符串。调用这些方法将会从 *read_data* 获取数据直到它被耗尽。对这些方法的模拟是相当简化的：每次 *mock* 被调用时，*read_data* 都将从头开始。如果你需要对你提供给测试代码的数据有更多控制那么你将需要自行定制这个 mock。如果这还不够用，那么通过一些 PyPI 上的内存文件系统包可以提供更真实的测试用文件系统。

在 3.4 版本发生变更: 增加了对*readline()* 和*readlines()* 的支持。对*read()* 的模拟被改为消耗 *read_data* 而不是每次调用时返回它。

在 3.5 版本发生变更: *read_data* 现在会在每次 *mock* 的调用时重置。

在 3.8 版本发生变更: 为实现增加了*__iter__()* 以便迭代操作（例如在 for 循环中）能正确地使用 *read_data*。

将*open()* 用作上下文管理器一确保你的文件处理被正确关闭的最佳方式因而十分常用:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

这里的问题在于即使你模拟了对*open()* 的调用但被用作上下文管理器（并调用其 `__enter__()` 和 `__exit__()` 方法）的却是 被返回的对象。

通过*MagicMock* 来模拟上下文管理器是十分常见且十分麻烦的因此需要使用一个辅助函数。

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

以及针对读取文件:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

自动 spec

自动 spec 是基于 mock 现有的 `spec` 特性。它将 mock 的 api 限制为原始对象（spec）的 api，但它是递归（惰性实现）的因而 mock 的属性只有与 spec 的属性相同的 api。除此之外被模拟的函数 / 方法具有与原对应物相同的调用签名因此如果它们被不正确地调用时会引发*TypeError*。

在我开始解释自动 spec 如何运作之前，先说明一下它的必要性。

Mock 是个非常强大和灵活的对象，但对 mock 操作来说有一个普遍存在的缺陷。如果你重构了你的部分代码，如修改成员的名称等，则针对仍然使用 旧 *API* 但其使用的是 mock 而非真实对象的代码的测试仍将通过。这意味着即使你的代码已被破坏你的测试却仍可能全部通过。

在 3.5 版本发生变更: 在 3.5 之前，在词断言中带有拼写错误应当引发错误的测试将会静默地通过。你仍然可通过向 Mock 传入 `unsafe=True` 来实现此行为。

请注意这是为什么你在单元测试之外还需要集成测试的原因之一。孤立地测试每个部分时全都正常而顺滑，但是如果你没有测试你的各个单元“联成一体”的情况如何那么就仍然存在测试可以发现大量错误的空间。

unittest.mock 已经提供了一个对此有帮助的特性，称为 spec 控制。如果你使用类或实例作为一个 mock 的 `spec` 那么你将仅能访问 mock 中只存在于实际的类中的属性:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

这个 spec 仅会应用于 mock 本身，因此对于 mock 中的任意方法我们仍然面临相同的问题:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()  # 故意的拼写错误!
```

自动 spec 解决了这个问题。你可以将 `autospec=True` 传给*patch()* / *patch.object()* 或是使用*create_autospec()* 函数来创建带有 spec 的 mock。如果你是将 `autospec=True` 参数传给*patch()* 那么被替代的那个对象将被用作 spec 对象。因为 spec 控制是“惰性地”执行的（spec 在 mock 中的属性被访问时才会被创建）所以即使是非常复杂或深度嵌套的对象（例如需要导入本身已导入了多个模块的模块）你也可以使用它而不会有太大的性能损失。

以下是一个实际应用的示例:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

你可以看到 `request.Request` 有一个 spec。`request.Request` 构造器接受两个参数 (其中一个是 *self*)。如果我们尝试不正确地调用它就会是这样的结果:

```
>>> req = request.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

该 spec 也将应用于被实例化的类（即附带 i.e. the return value of spec 的 mock 的返回值）:

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

Request 对象不是可调用对象，因此实例化我们的被模拟 request.Request 的返回值是一个不可调用的 mock。有了这个 spec 我们的断言中出现的任何拼写问题都将引发正确的报错:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # 故意的拼写错误!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

在许多情况下你将只需将 autospec=True 添加到你现有的*patch()* 调用中即可防止拼写错误和 api 变化所导致的问题。

除了通过*patch()* 来使用 *autospec* 还有一个*create_autospec()* 可以直接创建带有自动 spec 的 mock:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

不过这并非没有缺点和限制，这也就是为什么它不是默认行为。为了知道在 spec 对象上有哪些属性是可用的，autospec 必须对 spec 进行自省（访问其属性）。当你遍历 mock 上的属性时在原始对象上的对应遍历也将在底层进行。如果你的任何带 spec 的对象具有可触发代码执行的特征属性或描述器则你可能会无法使用 autospec。在另一方面更好的的做法是将你的对象设计为可以安全地执行自省[4]。

一个更严重的问题在于实例属性通常是在 __init__() 方法中被创建而在类中完全不存在。*autospec* 无法获取动态创建的属性而使得 api 被限制于可见的属性。

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

要解决这个问题有几种不同的方式。最容易但多少有些烦扰的方式是简单地在 mock 创建完成后再设置所需的属性。Just because *autospec* 只是不允许你获取不存在于 spec 上的属性但并不会阻止你设置它们:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

spec 和 *autospec* 都有更严格的版本 确实能阻止你设置不存在的属性。这在你希望确保你的代码只能 设置有效的属性时也很有用，但显然它会阻止下面这个特定的应用场景:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'a'
```

[4] 这仅适用于类或已实例化的对象。调用一个被模拟的类来创建一个 mock 实例 不会创建真的实例。只有属性查找——以及对*dir()* 的调用——会被执行。

解决此问题的最好方式可能是添加类属性作为在 `__init__()` 中初始化的实例属性的默认值。请注意如果你只在. Note that if you are only setting default attributes in `__init__()` 中设置默认属性那么通过类属性来提供它们（当然会在实例之间共享）也将有更快的速度。例如

```
class Something:
    a = 33
```

这带来了另一个问题。为今后将变为不同类型对象的那些成员提供默认值 `None` 是比较常见的做法。`None` 作为 spec 是没有用处的，因为它会使你无法访问 *any* 任何属性或方法。由于 `None` 作为 spec 将 永远不会有任何用处，并且有可能要指定某个通常为其他类型的成员，因此 autospec 不会为被设为 `None` 的成员使用 spec。它们将为普通的 mock (嗯——应为 MagicMocks):

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

如果你不喜欢修改你的生产类来添加默认值那么还有其他的选项。其中之一是简单地使用一个实例而非类作为 spec。另一选项则是创建一个生产类的子类并向该子类添加默认值而不影响到生产类。这两个选项都需要你使用一个替代对象作为 spec。值得庆幸的是*patch()* 支持这样做——你可以简单地传入替代对象作为 *autospec* 参数:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

将 mock 封包

unittest.mock.**seal**(*mock*)

封包将在访问被封包的 mock 的属性或其任何已经被递归地模拟的属性时禁止自动创建 mock。

如果一个带有名称或 spec 的 mock 实例被分配给一个属性则将不会在封包链中处理它。这可以防止人们对 mock 对象的固定部分执行封包。

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # 这将引发 AttributeError。
>>> mock.submock.attribute2  # 这将引发 AttributeError。
>>> mock.not_submock.attribute2  # 这不会引发异常。
```

Added in version 3.7.

27.6.6 `side_effect`, `return_value` 和 *wraps* 的优先顺序

它们的优先顺序是:

1. *side_effect*
2. *return_value*
3. *wraps*

如果三个均已设置，模拟对象将返回来自*side_effect*的值，完全忽略*return_value*和被包装的对象。如果设置任意两个，则具有更高优先级的那个将返回值。无论设置的顺序是哪个在前，优先级顺序将保持不变。

```
>>> from unittest.mock import Mock
>>> class Order:
...     @staticmethod
...     def get_value():
...         return "third"
...
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first"]
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'first'
```

由于 None 是*side_effect*的默认值，如果你将其值重新赋为 None，则优先级顺序将在*return_value*和被包装的对象之间进行检查，并忽略*side_effect*。

```
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
```

如果*side_effect*所返回的值为*DEFAULT*，它将被忽略并且优先级顺序将移至后继者来获取要返回的值。

```
>>> from unittest.mock import DEFAULT
>>> order_mock.get_value.side_effect = [DEFAULT]
>>> order_mock.get_value()
'second'
```

当*Mock*包装一个对象时，*return_value*的默认值将为*DEFAULT*。

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.return_value
sentinel.DEFAULT
>>> order_mock.get_value.return_value
sentinel.DEFAULT
```

优先级顺序将忽略该值并且它将移至末尾的后继者即被包装的对象。

由于真正调用的是被包装的对象，创建该模拟对象的实例将返回真正的该类实例。被包装的对象所需要的任何位置参数都必须被传入。

```
>>> order_mock_instance = order_mock()
>>> isinstance(order_mock_instance, Order)
True
>>> order_mock_instance.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'second'
```

但是如果你将其赋值为 None，由于它是一个显式赋值所以不会被忽略。因此，优先级顺序将不会移至被包装的对象。

```
>>> order_mock.get_value.return_value = None
>>> order_mock.get_value() is None
True
```

即使你在初始化模拟对象时立即立即全部设置这三者，优先级顺序仍会保持原样：

```
>>> order_mock = Mock(spec=Order, wraps=Order,
...                   **{"get_value.side_effect": ["first"],
...                      "get_value.return_value": "second"}
...                   )
...
>>> order_mock.get_value()
'first'
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

如果*side_effect*已耗尽，优先级顺序将不会导致从后续者获取值。而是会引发 StopIteration 异常。

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first side effect value",
...                                     "another side effect value"]
>>> order_mock.get_value.return_value = "second"
```

```
>>> order_mock.get_value()
'first side effect value'
>>> order_mock.get_value()
'another side effect value'
```

```
>>> order_mock.get_value()
Traceback (most recent call last):
 ...
StopIteration
```

27.7 unittest.mock --- 新手入门

Added in version 3.3.

27.7.1 使用 mock

模拟方法调用

使用*Mock*的常见场景：

- 模拟函数调用
- 记录在对象上的方法调用

你可能需要替换一个对象上的方法，用于确认此方法被系统中的其他部分调用过，并且调用时使用了正确的参数。

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

使用了 mock (本例中的 real.method) 之后，它有方法和属性可以让你针对它是被如何使用的下断言。

备注

在多数示例中，*Mock* 与*MagicMock* 两个类可以相互替换，而 MagicMock 是一个更适用的类，通常情况下，使用它就可以了。

如果 mock 被调用，它的*called* 属性就会变成 True，更重要的是，我们可以使用*assert_called_with()* 或者*assert_called_once_with()* 方法来确认它在被调用时使用了正确的参数。

在如下的测试示例中，验证对于 ProductionClass().method 的调用会导致 something 的调用。

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

对象上的方法调用的 mock

上一个例子中我们直接在对象上给方法打补丁以检查它是否被正确地调用。另一个常见的用例是将一个对象传给一个方法（或被测试系统的某个部分）然后检查它是否以正确的方式被使用。

下面这个简单的 ProductionClass 具有一个 closer 方法。如果它附带一个对象被调用那么它就会调用其中的 close。

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

所以为了测试它我们需要传入一个带有 close 方法的对象并检查它是否被正确地调用。

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

我们不需要做任何事来在我们的 mock 上提供'close' 方法。访问 close 的操作就会创建它。因此，如果'close' 还未被调用那么在测试时访问它就将创建它，但是*assert_called_with()* 则会引发一个失败的异常。

模拟类

一个常见的用例是模拟被测试的代码所实例化的类。当你给一个类打上补丁，该类就会被替换为一个 mock。实例是通过 调用该类来创建的。这意味着你要通过查看被模拟类的返回值来访问 “mock 实例”。

在下面的例子中我们有一个函数 some_function 实例化了 Foo 并调用该实例中的一个方法。对*patch()* 的调用会将类 Foo 替换为一个 mock。Foo 实例是调用该 mock 的结果，所以它是通过修改*return_value* 来配置的。

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
```

(续下页)

(接上页)

```
...     result = some_function()
...     assert result == 'the result'
```

命名你的 mock

给你的 mock 起个名字可能会很有用。名字会显示在 mock 的 repr 中并在 mock 出现于测试失败消息中时可以帮助理解。这个名字也会被传播给 mock 的属性或方法:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

追踪所有的调用

通常你会想要追踪对某个方法的多次调用。*mock_calls* 属性记录了所有对 mock 的子属性的调用——并且还包括对它们的子属性的调用。

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

如果你做了一个有关 mock_calls 的断言并且有任何非预期的方法被调用，则断言将失败。这很有用处，因为除了断言你所预期的调用已被执行，你还会检查它们是否以正确的顺序被执行并且没有额外的调用:

你使用*call* 对象来构造列表以便与 mock_calls 进行比较:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

然而，返回 mock 的调用的形参不会被记录，这意味着不可能追踪附带了重要形参的创建上级对象的嵌套调用:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

设置返回值和属性

在 mock 对象上设置返回值是非常容易的:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

当然你也可以对 mock 上的方法做同样的操作:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

返回值也可以在构造器中设置:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

如果你需要在你的 mock 上设置一个属性，只需这样做:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

有时你会想要模拟更复杂的情况，例如这个例子 `mock.connection.cursor().execute("SELECT 1")`。如果我们希望这个调用返回一个列表，那么我们还必须配置嵌套调用的结果。

我们可以像这样使用*call* 在一个“链式调用”中构造调用集合以便随后方便地设置断言:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

对 `.call_list()` 的调用会将我们的调用对象转成一个代表链式调用的调用列表。

通过 mock 引发异常

一个很有用的属性是*side_effect*。如果你将该属性设为一个异常类或者实例那么当 mock 被调用时该异常将会被引发。

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

附带影响函数和可迭代对象

`side_effect` 也可以被设为一个函数或可迭代对象。`side_effect` 作为可迭代对象的应用场景适用于你的 mock 将要被多次调用，并且你希望每次调用都返回不同的值的情况。当你将 `side_effect` 设为一个可迭代对象时每次对 mock 的调用将返回可迭代对象的下一个值。

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

对于更高级的用例，例如根据 mock 调用时附带的参数动态改变返回值，`side_effect` 可以指定一个函数。该函数将附带与 mock 相同的参数被调用。该函数所返回的就是调用所返回的对象:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
```

(续下页)

(接上页)

```
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

模拟异步迭代器

从 Python 3.8 起，`AsyncMock` 和 `MagicMock` 支持通过 `__aiter__` 来模拟 async-iterators。`__aiter__` 的*return_value* 属性可以被用来设置要用于迭代的返回值。

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

模拟异步上下文管理器

从 Python 3.8 起，`AsyncMock` 和 `MagicMock` 支持通过 `__aenter__` 和 `__aexit__` 来模拟 async-context-managers。在默认情况下，`__aenter__` 和 `__aexit__` 将为返回异步函数的 `AsyncMock` 实例。

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager())  # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

基于现有对象创建模拟对象

使用模拟操作的一个问题是它会将你的测试与你的 mock 实现相关联而不是与你的真实代码相关联。假设你有一个实现了 `some_method` 的类。在对另一个类的测试中，你提供了一个 同 样提供了 `some_method` 的模拟该对象的 mock 对象。如果后来你重构了第一个类，使得它不再具有 `some_method` ——那么你的测试将继续保持通过，尽管现在你的代码已经被破坏了！

Mock 允许你使用 allows you to provide an object as a specification for the mock, using the *spec* 关键字参数来提供一个对象作为 mock 的规格说明。在 mock 上访问不存在于你的规格说明对象中的方法 / 属性将立即引发一个属性错误。如果你修改你的规格说明的实现，，那么使用了该类的测试将立即开始失败而不需要你在这些测试中实例化该类。

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: Mock object has no attribute 'old_method'. Did you mean: 'class_method'?
```

使用规格说明还可以启用对 mock 的调用的更聪明的匹配操作，无论是否有将某些形参作为位置或关键字参数传入:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

如果你想要让这些更聪明的匹配操作也适用于 mock 上的方法调用，你可以使用*auto-speccing*。

如果你想要更强形式的规格说明以防止设置任意属性并获取它们那么你可以使用 *spec_set* 来代替 *spec*。

使用 side_effect 返回每个文件的内容

mock_open() 被用来为*open()* 方法打补丁。*side_effect* 可被用来在每次调用中返回一个新的 Mock 对象。这可被用来返回存储在字典中的每个文件的不同内容:

```
DEFAULT = "default"
data_dict = {"file1": "data1",
             "file2": "data2"}

def open_side_effect(name):
    return mock_open(read_data=data_dict.get(name, DEFAULT))()

with patch("builtins.open", side_effect=open_side_effect):
    with open("file1") as file1:
        assert file1.read() == "data1"

    with open("file2") as file2:
        assert file2.read() == "data2"

    with open("file3") as file2:
        assert file2.read() == "default"
```

27.7.2 补丁装饰器

备注

在查找对象的名称空间中修补对象使用*patch()* 。使用起来很简单，阅读*补丁的位置* 来快速上手。

测试中的一个常见需求是为类属性或模块属性打补丁，例如修补内置对象或修补某个模块中的类来测试其是否被实例化。模块和类都可算是全局对象，因此对它们打补丁的操作必须在测试完成之后被还原否则补丁将持续影响其他测试并导致难以诊断的问题。

为此 mock 提供了三个便捷的装饰器: *patch()*, *patch.object()* 和*patch.dict()*。patch 接受单个字符串，其形式 package.module.Class.attribute 指明你要修补的属性。它还可选择接受一个值用来替换指定的属性（或者类对象等等）。'patch.object' 接受一个对象和你想要修补的属性名称，并可选择接受要用作补丁的值。

patch.object:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
```

(续下页)

(接上页)

```
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

如果你要给一个模块 (包括*builtins*) 打补丁则可使用*patch()* 来代替*patch.object()*:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

如有必要模块名可以是“带点号”的，其形式如 package.module:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

一个良好的模式是实际地装饰测试方法本身:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

如果你想要通过 Mock 来打补丁，你可以只附带一个参数使用*patch()* (或附带两个参数使用*patch.object()*)。这将为你创建 mock 并传递给测试函数 / 方法:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

你可以使用以下模式来堆叠多个补丁装饰器:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

当你嵌套 patch 装饰器时将以它们被应用的相同顺序（即 *Python* 应用装饰器的正常顺序）将 mock 传入被装饰的函数。也就是说从下往上，因此在上面的示例中 test_module.ClassName2 的 mock 会被最先传入。

还有一个*patch.dict()* 用于在一定范围内设置字典中的值，并在测试结束时将字典恢复为其原始状态:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

patch, patch.object 和 patch.dict 都可被用作上下文管理器。

在你使用*patch()* 为你创建 mock 时，你可以使用 with 语句的"as" 形式来获得对 mock 的引用:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

作为替代 patch, patch.object 和 patch.dict 可以被用作类装饰器。当以此方式使用时其效果与将装饰器单独应用到每个以"test" 打头的方法上相同。

27.7.3 更多示例

下面是一些针对更为高级应用场景的补充示例。

模拟链式调用

实际上一旦你理解了*return_value* 属性那么使用 mock 模拟链式调用就会相当直观。当一个 mock 首次被调用，或者当你在它被调用前获取其 return_value 时，将会创建一个新的*Mock*。

这意味着你可以通过检视 return_value mock 来了解从调用被模拟对象返回的对象是如何被使用的:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

从这里开始只需一个步骤即可配置并创建有关链式调用的断言。当然还有另一种选择是首先以更易于测试的方式来编写你的代码...

因此，如果我们有这样一些代码:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs').start_
↪call()
...         # more code
```

假定 BackendProvider 已经过良好测试，我们要如何测试 method()？特别地，我们希望测试代码段 # more code 是否以正确的方式使用了响应对象。

由于这个链式调用来自一个实例属性我们可以对 backend 属性在 Something 实例上进行猴子式修补。在这个特定情况下我们只对最后调用 start_call 的返回值感兴趣所以我们不需要进行太多的配置。让我们假定它返回的是“文件类”对象，因此我们将确保我们的响应对象使用内置的*open()* 作为其 spec。

为了做到这一点我们创建一个 mock 实例作为我们的 mock 后端并为它创建一个 mock 响应对象。要将该响应对象设为最后的 start_call 的返回值我们可以这样做:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_value =␣
↪mock_response
```

我们可以通过更好一些的方式做到这一点，即使用*configure_mock()* 方法直接为我们设置返回值:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_value':␣
↪mock_response}
>>> mock_backend.configure_mock(**config)
```

有了这些我们就能准备好给“mock 后端”打上猴子补丁并可以执行真正的调用:

```
>>> something.backend = mock_backend
>>> something.method()
```

使用*mock_calls* 我们可以通过一个断言来检查链式调用。一个链式调用就是在一行代码中连续执行多个调用，所以在 mock_calls 中将会有多个条目。我们可以使用*call.call_list()* 来为我们创建这个调用列表:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

部分模拟

在一些测试中，我想把对*datetime.date.today()* 的调用模拟为返回一个已知日期的调用，但又不想阻止测试中的代码创建新的日期对象。然而*datetime.date* 是用 C 语言编写的，因此我不能简单地给静态的*datetime.date.today()* 方法打上猴子补丁。

我找到了实现这一点的简单方式即通过一个 mock 来实际包装日期类，但通过对构造器的调用传递给真实的类（并返回真实的实例）。

这里使用*patch 装饰器* 来模拟被测试模块中的 date 类。然后将模拟 date 类的*side_effect* 属性设为一个返回真实日期的 lambda 函数。当模拟 date 类被调用时，将通过 side_effect 构造并返回一个真实日期。

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

请注意我们没有在全局范围上修补*datetime.date*，我们只是在 使用它的模块中给 date 打补丁。参见*补丁的位置*。

当 date.today() 被调用时将返回一个已知的日期，但对 date(...) 构造器的调用仍会返回普通的日期。如果不是这样你会发现你必须使用与被测试的代码完全相同的算法来计算出预期的结果，这是测试工作中的一个经典的反模式。

对 date 构造器的调用会被记录在 mock_date 属性中 (call_count 等)，它们也可能对你的测试有用处。

有关处理模块日期或其他内置类的一种替代方式的讨论请参见 这篇博客文章。

模拟生成器方法

Python 生成器是指在被迭代时使用 yield 语句来返回一系列值的函数或方法[1]。

[1] 此外还有生成器表达式和更多的生成器 进阶用法，但在这里我们不去关心它们。有关生成器及其强大功能的一个很好的介绍请参阅: 针对系统程序员的生成器妙招。

调用生成器方法 / 函数将返回生成器对象。生成器对象随后会被迭代。迭代操作对应的协议方法是*__iter__()*，因此我们可以使用*MagicMock* 来模拟它。

以下是一个使用"iter" 方法模拟为生成器的示例类:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

我们要如何模拟这个类，特别是它的"iter" 方法呢?

为了配置从迭代操作（隐含在对*list* 的调用中）返回的值，我们需要配置调用 `foo.iter()` 所返回的对象。

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

对每个测试方法应用相同的补丁

如果你想要为多个测试方法准备好多个补丁那么最简单的方式就是将 patch 装饰器应用到每个方法上。这在感觉上像上不必要的重复。对此，你可以使用*patch()* (包括基各种不同形式) 作为类装饰器。这将把补丁应用于类上的所有测试方法。测试方法是通过以 `test` 打头的名称来标识的:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

另一种管理补丁的方式是使用*补丁方法: start 和 stop*。它允许你将打补丁操作移至你的 `setUp` 和 `tearDown` 方法中。

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

如果你要使用这个技巧则你必须通过调用 stop 来确保补丁被“恢复”。这可能要比你想像的更麻烦，因为如果在 setUp 中引发了异常那么 tearDown 将不会被调用。*unittest.TestCase.addCleanup()* 可以做到更方便:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

模拟未绑定方法

当前在编写测试时我需要修补一个 *未绑定方法* (在类上而不是在实例上为方法打补丁)。我需要将 self 作为第一个参数传入因为我想对哪些对象在调用这个特定方法进行断言。问题是这里你不能用 mock 来打补丁，因为如果你用 mock 来替换一个未绑定方法那么当从实例中获取时它就不会成为一个已绑定方法，因而它不会获得传入的 self。绕过此问题的办法是改用一个真正的函数来修补未绑定方法。*patch()* 装饰器让使用 mock 来给方法打补丁变得如此简单以至于创建一个真正的函数成为一件麻烦事。

如果将 autospec=True 传给 patch 那么它就会用一个 *真正的* 函数对象来打补丁。这个函数对象具有与它所替换的函数相同的签名，但会在内部将操作委托给一个 mock。你仍然可以通过与以前完全相同的方式来自动创建你的 mock。但是这将意味着一件事，就是如果你用它来修补一个类上的非绑定方法那么如果它是从一个实例中获取则被模拟的函数将被转为已绑定方法。传给它的第一个参数将为 self，而这真是我想要的:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

如果我们不使用 autospec=True 那么这个未绑定方法会改为通过一个 Mock 补丁来修补，而不是附带 self 来调用。

通过 mock 检查多次调用

mock 有一个很好的 API 用于针对你的 mock 对象如何被使用来下断言。

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

如果你的 mock 只会被调用一次那么你可以使用*assert_called_once_with()* 方法，该方法也会断言*call_count* 的值为一。

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
```

(续下页)

(接上页)

```
AssertionError: Expected 'foo_bar' to be called once. Called 2 times.
Calls: [call('baz', spam='eggs'), call()].
```

assert_called_with 和 assert_called_once_with 都是有关 最近调用的断言。如果你的 mock 将被多次调用，并且你想要针对 所有这些调用下断言你可以使用*call_args_list*:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

使用*call* 辅助对象可以方便地针对这些调用下断言。你可以创建一个预期调用的列表并将其与 call_args_list 比较。这看起来与 call_args_list 的 repr 非常相似:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

处理可变参数

另一种很少见，但可能给你带来麻烦的情况会在你的 mock 附带可变参数被调用的时候发生。call_args 和 call_args_list 将保存对这些参数的 *引用*。如果这些参数被受测试的代码所改变那么你将无法再针对当该 mock 被调用时附带的参数值下断言。

下面是一些演示此问题的示例代码。设想在'mymodule' 中定义了下列函数:

```
def frob(val):
    pass

def grob(val):
    "先执行 frob 然后清空 val"
    frob(val)
    val.clear()
```

当我们想要测试 grob 调用 frob 并附带了正确的参数时将可看到发生了什么:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

对于 mock 的一个可能性是复制你传入的参数。如果你创建依赖于对象标识号相等性的断言那么这可能会在后面导致问题。

下面是一个使用*side_effect* 功能的解决方案。如果你为 mock 提供了 side_effect 函数那么 side_effect 将附带与该 mock 相同的参数被调用。这样我们就有机会拷贝这些参数并将其保存起来用于之后执行断言。在本例中我使用了 *另一个* mock 来保存参数以便可以使用该 mock 的方法来执行断言。在这里辅助函数再次为我设置好了这一切。

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
```

(续下页)

(接上页)

```
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

调用 `copy_call_args` 时会传入将被调用的 mock。它将返回一个新的 mock 供我们进行断言。`side_effect` 函数会拷贝这些参数并附带该副本来调用我们的 `new_mock`。

备注

如果你的 mock 只会被使用一次那么有更容易的方式可以在它们被调用时检查参数。你可以简单地在 `side_effect` 函数中执行检查。

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

一个替代方式是创建一个*Mock* 或*MagicMock* 的子类来拷贝 (使用*copy.deepcopy()*) 参数。下面是一个示例实现:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock({1})
Actual: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

当你子类化 Mock 或 MagicMock 时所有动态创建的属性以及 return_value 都将自动使用你的子类。这意味着 CopyingMock 的所有子类也都将为 CopyingMock 类型。

嵌套补丁

使用 patch 作为上下文管理器很不错，但是如果你要执行多个补丁你将不断嵌套 with 语句使得代码越来越深地向右缩进:

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

使用 unittest cleanup 函数和*补丁方法: start 和 stop* 我们可以达成同样的效果而无须嵌套缩进。一个简单的辅助方法 create_patch 会为我们执行打补丁操作并返回所创建的 mock:

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

使用 MagicMock 模拟字典

你可能会想要模拟一个字典或其他容器对象，记录所有对它的访问并让它的行为仍然像是一个字典。

要做到这点我们可以用*MagicMock*，它的行为类似于字典，并会使用*side_effect* 将字典访问委托给下层的在我们控制之下的一个真正的字典。

当我们的 MagicMock 的 __getitem__() 和 __setitem__() 方法被调用（即正常的字典访问操作）时 side_effect 将附带相应的键（对于 __setitem__ 还将附带值）被调用。我们还可以控制返回的内容。

在 MagicMock 被使用之后我们可以使用*call_args_list* 等属性来针对该字典是如何被使用的下断言。

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
```

(续下页)

(接上页)

```
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

备注

MagicMock 的一个可用替代是使用 Mock 并 仅仅提供你明确需要的魔术方法:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

第三个选项是使用 MagicMock 但传入 dict 作为 *spec* (或 *spec_set*) 参数以使得所创建的 MagicMock 只有字典魔术方法是可用的:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

通过提供这些附带影响函数，mock 的行为将类似于普通字典但又会记录所有访问。如果你尝试访问一个不存在的键它甚至会引发*KeyError*。

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

在它被使用之后你可以使用普通的 mock 方法和属性进行有关访问操作的断言:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

模拟子类及其属性

你可能出于各种原因想要子类化*Mock*。其中一个可能的原因是为了添加辅助方法。下面是一个笨兮兮的示例:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
```

(续下页)

(接上页)

```
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` 实例的标准行为是属性和返回值 mock 具有与它们所访问的 mock 相同的类型。这将确保 `Mock` 的属性均为 `Mocks` 而 `MagicMock` 的属性均为 `MagicMocks`[2]。因此如果你通过子类化来添加辅助方法那么它们也将在你的子类的实例的属性和返回值 mock 上可用。

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

有时这很不方便。例如，一位用户 子类化了 mock 来创建一个 Twisted 适配器。将它也应用于属性实际上会导致出错。

`Mock`(它的所有形式) 使用一个名为 `_get_child_mock` 的方法来创建这些用于属性和返回值的“子 mock”。你可以通过重写此方法来防止你的子类被用于属性。其签名被设为接受任意关键字参数 (`**kwargs`) 并且它们会被传递给 mock 构造器:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

通过 patch.dict 模拟导入

有一种会令模拟变困难的情况是当你在函数内部有局部导入。这更难模拟的原因是它们不是使用来自我们能打补丁的模拟命名空间中的对象。

一般来说局部导入是应当避免的。局部导入有时是为了防止循环依赖，而这个问题 通常都有更好的解决办法（重构代码）或者通过延迟导入来防止“前期成本”。这也可以通过比无条件地局部导入更好的方式来解决（将模块保存为一个类或模块属性并且只在首次使用时执行导入）。

除此之外还有一个办法可以使用 `mock` 来影响导入的结果。导入操作会从*`sys.modules`* 字典提取一个 对象。请注意是提取一个 对象，它不是必须为模块。首次导入一个模块将使一个模块对象被放入 `sys.modules`，因此通常当你执行导入时你将得到一个模块。但是并非必然如此。

这意味着你可以使用*`patch.dict()`* 来 临时性地将一个 mock 放入*`sys.modules`*。在补丁激活期间的任何导入操作都将得到该 mock。当补丁完成时（被装饰的函数退出，with 语句代码块结束或者 `patcher.stop()` 被调用）则之前存在的任何东西都将被安全地恢复。

下面是一个模拟’fooble’ 模拟的示例。

```
>>> import sys
>>> mock = Mock()
```

(续下页)

[2] 此规则的一个例外涉及不可调用 mock。属性会使用可调用对象版本是因为如非如此则不可调用 mock 将无法拥有可调用的方法。

(接上页)

```
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

你可以看到 `import fooble` 成功执行，而当退出时*sys.modules* 中将不再有'fooble'。

这同样适用于 `from module import name` 形式:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

稍微多做一点工作你还可以模拟包的导入:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

追踪调用顺序和不太冗长的调用断言

Mock 类允许你通过*method_calls* 属性来追踪在你的 mock 对象上的方法调用的 *顺序*。这并不允许你追踪单独 mock 对象之间的调用顺序，但是我们可以使用*mock_calls* 来达到同样的效果。

因为 mock 会追踪 `mock_calls` 中对子 mock 的调用，并且访问 mock 的任意属性都会创建一个子 mock，所以我们可以基于父 mock 创建单独的子 mock。随后对这些子 mock 的调用将按顺序被记录在父 mock 的 `mock_calls` 中:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

我们可以随后通过与管理器 mock 上的 `mock_calls` 属性进行比较来进行有关这些调用，包括调用顺序的断言:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

如果 `patch` 创建并准备好了你的 mock 那么你可以使用*attach_mock()* 方法将它们附加到管理器 mock 上。在附加之后所有调用都将被记录在管理器的 `mock_calls` 中。

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

如果已经进行了许多调用，但是你只对它们的一个特定序列感兴趣则有一种替代方式是使用*assert_has_calls()* 方法。这需要一个调用的列表（使用*call* 对象来构建）。如果该调用序列在*mock_calls* 中则断言将成功。

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

即使链式调用 `m.one().two().three()` 不是对 mock 的唯一调用，该断言仍将成功。

有时可能会对一个 mock 进行多次调用，而你只对断言其中的某些调用感兴趣。你甚至可能对顺序也不关心。在这种情况下你可以将 `any_order=True` 传给 `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

更复杂的参数匹配

使用与*ANY* 一样的基本概念我们可以实现匹配器以便在用作 mock 的参数的对象上执行更复杂的断言。

假设我们准备将某个对象传给一个在默认情况下基于对象标识相等（这是 Python 中用户自定义类的默认行为）的 mock。要使用*assert_called_with()* 我们就将必须传入完全相同的对象。如果我们只对该对象的某些属性感兴趣那么我们可以创建一个能为我们检查这些属性的匹配器。

在这个示例中你可以看到为何执行对 `assert_called_with` 的‘标准’调用并不足够:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock(<__main__.Foo object at 0x...>)
Actual: mock(<__main__.Foo object at 0x...>)
```

一个针对我们的 `Foo` 类的比较函数看上去会是这样的:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

而一个可以使用这样的比较函数进行相等性比较运算的匹配器对象看上去会是这样的:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

将所有这些放在一起:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

`Matcher` 是用我们的比较函数和我们想要比较的 `Foo` 对象来实例化的。在 `assert_called_with` 中将会调用 `Matcher` 的相等性方法，它会将调用 mock 时附带的对象与我们创建我们的匹配器时附带的对象进行比较。如果它们匹配则 `assert_called_with` 通过，而如果不匹配则会引发*AssertionError*:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

通过一些调整你可以让比较函数直接引发*AssertionError* 并提供更有用的失败消息。

从 1.5 版开始，Python 测试库 PyHamcrest 提供了类似的功能，在这里可能会很有用，它采用的形式是相等性匹配器 (hamcrest.library.integration.match_equality)。

27.8 `test` --- Python 回归测试包

> **备注**
>
> *test* 包只供 Python 内部使用。它的记录是为了让 Python 的核心开发者受益。我们不鼓励在 Python 标准库之外使用这个包，因为这里提到的代码在 Python 的不同版本之间可能会改变或被删除而不另行通知。

---

test 包包含了 Python 的所有回归测试，以及*test.support* 和*test.regrtest* 模块。*test.support* 用于增强你的测试，而*test.regrtest* 驱动测试套件。

`test`包中每个名字以 ``test_`` 开头的模块都是一个特定模块或功能的测试套件。所有新的测试应该使用*unittest* 或*doctest* 模块编写。一些旧的测试是使用“传统”的测试风格编写的，即比较打印出来的输出到 `sys.stdout`；这种测试风格被认为是过时的。

参见

模块*unittest*
编写 PyUnit 回归测试.

***doctest* --- 文档测试模块**
嵌入到文档字符串的测试。

27.8.1 为 test 包编写单元测试

使用*unittest* 模块的测试最好是遵循一些准则。其中一条是测试模块的名称要以 `test_` 打头并以被测试模块的名称结尾。测试模块中的测试方法应当以 `test_` 打头并以该方法所测试的内容的说明结尾。这很有必要因为这样测试驱动程序就会将这些方法识别为测试方法。此外，该方法不应当包括任何文档字符串。应当使用注释 (例如 `# Tests function returns only True or False`) 来为测试方法提供文档说明。这样做是因为文档字符串如果存在则会被打印出来因此无法指明正在运行哪个测试。

有一个基本模板经常会被使用:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # 仅在需要时使用 setUp() 和 tearDown()

    def setUp(self):
        ... 为准备测试而执行的代码 ...

    def tearDown(self):
        ... 为测试后的清理而执行的代码 ...

    def test_feature_one(self):
        # 测试特性一。
        ... 测试代码 ...

    def test_feature_two(self):
        # 测试特性二。
        ... 测试代码 ...

    ... 更多的测试方法 ...

class MyTestCase2(unittest.TestCase):
    ... 与 MyTestCase1 的结构相同 ...

... 更多的测试类 ...

if __name__ == '__main__':
    unittest.main()
```

这种代码模式允许测试套件由*test.regrtest* 运行，作为支持*unittest* CLI 的脚本单独运行，或者通过 `python -m unittest` CLI 来运行。

回归测试的目标是尝试破坏代码。这引出了一些需要遵循的准则:

- 测试套件应当测试所有的类、函数和常量。这不仅包括要向外界展示的外部 API 也包括“私有”的代码。
- 白盒测试（在编写测试时检查被测试的代码）是最推荐的。黑盒测试（只测试已发布的用户接口）因不够完整而不能确保所有边界和边缘情况都被测试到。
- 确保所有可能的值包括无效的值都被测试到。这能确保不仅全部的有效值都可被接受而且不适当的值也能被正确地处理。

- 消耗尽可能多的代码路径。测试发生分支的地方从而调整输入以确保通过代码采取尽可能多的不同路径。
- 为受测试的代码所发现的任何代码缺陷添加明确的测试。这将确保如果代码在将来被改变错误也不会再次出现。
- 确保在你的测试完成后执行清理（例如关闭并删除所有临时文件）。
- 如果某个测试依赖于操作系统上的特定条件那么要在尝试测试之前先验证该条件是否已存在。
- 尽可能少地导入模块并尽可能快地完成操作。这可以最大限度地减少测试的外部依赖性并且还可以最大限度地减少导入模块带来的附带影响所导致的异常行为。
- 尝试最大限度地重用代码。在某些情况下，测试结果会因使用不同类型的输入这样的小细节而变化。可通过一个指定输入的类来子类化一个基本测试类来最大限度地减少重复代码:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

当使用这种模式时，请记住所有继承自*unittest.TestCase* 的类都会作为测试来运行。上面例子中的 TestFuncAcceptsSequencesMixin 类没有任何数据所以其本身是无法运行的，因此它不是继承自*unittest.TestCase*。

参见

测试驱动的开发
Kent Beck 所著的阐述在实现代码之前编写驱动的书。

27.8.2 使用命令行界面运行测试

通过使用 -m 选项*test* 包可以作为脚本运行以驱动 Python 的回归测试套件: **python -m test**。在内部，它使用*test.regrtest*；之前 Python 版本所使用的 **python -m test.regrtest** 调用仍然有效。运行该脚本自身会自动开始运行*test* 包中的所有回归测试。它通过在包中查找所有名称以 test_ 打头的模块，导入它们，并在有 test_main() 函数时执行它或是在没有 test_main 时通过 unittest.TestLoader.loadTestsFromModule 载入测试。要执行的测试的名称也可以被传递给脚本。指定一个单独的回归测试 (**python -m test test_spam**) 将使输出最小化并且只打印测试通过或失败的消息。

直接运行*test* 将允许设置哪些资源可供测试使用。你可以通过使用 -u 命令行选项来做到这一点。指定 all 作为 -u 选项的值将启用所有可能的资源: **python -m test -uall**。如果只需要一项资源（这是更为常见的情况），可以在 all 之后加一个以逗号分隔的列表来指明不需要的资源。命令 **python -m test -uall,-audio,-largefile** 将运行*test* 并使用除 audio 和 largefile 资源之外的所有资源。要查看所有资源的列表和更多的命令行选项，请运行 **python -m test -h**。

另外一些执行回归测试的方式依赖于执行测试所在的系统平台。在 Unix 上，你可以在构建 Python 的最高层级目录中运行 **make test**。在 Windows 上，在你的 PCbuild 目录中执行 **rt.bat** 将运行所有的回归测试。

27.9 test.support --- 针对 Python 测试套件的工具

test.support 模块提供了对 Python 的回归测试套件的支持。

> **备注**
>
> *test.support* 不是一个公用模块。这篇文档是为了帮助 Python 开发者编写测试。此模块的 API 可能被改变而不顾及发行版本之间的向下兼容性问题。

此模块定义了以下异常:

exception test.support.**TestFailed**

当一个测试失败时将被引发的异常。此异常已被弃用而应改用基于*unittest* 的测试以及*unittest.TestCase* 的断言方法。

exception test.support.**ResourceDenied**

unittest.SkipTest 的子类。当一个资源（例如网络连接）不可用时将被引发。由*requires()* 函数所引发。

test.support 模块定义了以下常量:

test.support.**verbose**

当启用详细输出时为 `True`。当需要有关运行中的测试的更详细信息时应当被选择。*verbose* 是由*test.regrtest* 来设置的。

test.support.**is_jython**

如果所运行的解释器是 Jython 时为 `True`。

test.support.**is_android**

如果系统是 Android 时为 `True`。

test.support.**unix_shell**

如果系统不是 Windows 时则为 shell 的路径；否则为 `None`。

test.support.**LOOPBACK_TIMEOUT**

使用网络服务器监听网络本地环回接口如 `127.0.0.1` 的测试的以秒为单位的超时值。

该超时长到足以防止测试失败：它要考虑客户端和服务器可能会在不同线程甚至不同进程中运行。

该超时应当对于*socket.socket* 的*connect()*, *recv()* 和*send()* 方法都足够长。

其默认值为 5 秒。

参见*INTERNET_TIMEOUT*。

test.support.**INTERNET_TIMEOUT**

发往互联网的网络请求的以秒为单位的超时值。

该超时短到足以避免测试在互联网请求因任何原因被阻止时等待太久。

通常使用*INTERNET_TIMEOUT* 的超时不应该将测试标记为失败，而是跳过测试：参见*transient_internet()*。

其默认值是 1 分钟。

参见*LOOPBACK_TIMEOUT*。

test.support.**SHORT_TIMEOUT**

如果测试耗时“太长”而要将测试标记为失败的以秒为单位的超时值。

该超时值取决于 regrtest `--timeout` 命令行选项。

如果一个使用*SHORT_TIMEOUT* 的测试在慢速 buildbots 上开始随机失败，请使用*LONG_TIMEOUT* 来代替。

其默认值为 30 秒。

test.support.**LONG_TIMEOUT**

用于检测测试何时挂起的以秒为单位的超时值。

它的长度足够在最慢的 Python buildbot 上降低测试失败的风险。如果测试耗时“过长”也不应当用它将该测试标记为失败。此超时值依赖于 regrtest `--timeout` 命令行选项。

其默认值为 5 分钟。

另请参见*LOOPBACK_TIMEOUT*, *INTERNET_TIMEOUT* 和*SHORT_TIMEOUT*。

test.support.**PGO**

当测试对 PGO 没有用处时设置是否要跳过测试。

test.support.**PIPE_MAX_SIZE**

一个通常大于下层 OS 管道缓冲区大小的常量，以产生写入阻塞。

test.support.**Py_DEBUG**

如果 Python 编译时定义了 `Py_DEBUG` 宏则为 `True`，也就是说，当 Python 是 以调试模式编译的时候。

Added in version 3.12.

test.support.**SOCK_MAX_SIZE**

一个通常大于下层 OS 套接字缓冲区大小的常量，以产生写入阻塞。

test.support.**TEST_SUPPORT_DIR**

设为包含*test.support* 的最高层级目录。

test.support.**TEST_HOME_DIR**

设为 test 包的最高层级目录。

test.support.**TEST_DATA_DIR**

设为 test 包中的 `data` 目录。

test.support.**MAX_Py_ssize_t**

设为大内存测试的*sys.maxsize*。

test.support.**max_memuse**

通过*set_memlimit()* 设为针对大内存测试的内存限制。受*MAX_Py_ssize_t* 的限制。

test.support.**real_max_memuse**

通过*set_memlimit()* 设为针对大内存测试的内存限制。不受*MAX_Py_ssize_t* 的限制。

test.support.**MISSING_C_DOCSTRINGS**

如果 Python 编译时不带文档字符串（即未定义 `WITH_DOC_STRINGS` 宏）则设为 `True`。参见 `configure --without-doc-strings` 选项。

另请参阅*HAVE_DOCSTRINGS* 变量。

test.support.**HAVE_DOCSTRINGS**

如果函数带有文档字符串则设为 `True`。参见 `python -OO` 选项，该选项会去除在 Python 中实现的函数的文档字符串。

另请参阅*MISSING_C_DOCSTRINGS* 变量。

test.support.**TEST_HTTP_URL**

定义用于网络测试的韧性 HTTP 服务器的 URL。

test.support.**ALWAYS_EQ**

等于任何对象的对象。用于测试混合类型比较。

test.support.**NEVER_EQ**

不等于任何对象的对象 (即使是*ALWAYS_EQ*)。用于测试混合类型比较。

test.support.**LARGEST**
大于任何对象的对象（除了其自身）。用于测试混合类型比较。

test.support.**SMALLEST**
小于任何对象的对象（除了其自身）。用于测试混合类型比较。Used to test mixed type comparison.

test.support 模块定义了以下函数:

test.support.**busy_retry**(*timeout*, *err_msg=None*, */*, ***, *error=True*)
运行循环体直到以 break 停止循环。

在 *timeout* 秒后，如果 *error* 为真值则引发*AssertionError*，或者如果 *error* 为假值则只停止循环。

示例:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

error=False 的用法示例:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**sleeping_retry**(*timeout*, *err_msg=None*, */*, ***, *init_delay=0.010*, *max_delay=1.0*, *error=True*)
应用指数回退的等待策略。

运行循环体直到以 break 停止循环。在每次循环迭代时休眠，但第一次迭代时除外。每次迭代的休眠延时都将加倍（至多 *max_delay* 秒）。

请参阅*busy_retry()* 文档了解相关形参的用法。

在 SHORT_TIMEOUT 秒后引发异常的示例:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

error=False 的用法示例:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**is_resource_enabled**(*resource*)
如果 *resource* 已启用并可用则返回 True。可用资源列表只有当*test.regrtest* 正在执行测试时才会被设置。

test.support.**python_is_optimized**()
如果 Python 编译未使用 -O0 或 -Og 则返回 True。

test.support.**with_pymalloc**()
返回 _testcapi.WITH_PYMALLOC。

test.support.**requires**(*resource*, *msg=None*)
如果 *resource* 不可用则引发*ResourceDenied*。如果该异常被引发则 *msg* 为传给*ResourceDenied* 的参数。如果被 __name__ 为 '__main__' 的函数调用则总是返回 True。在测试由*test.regrtest* 执行时使用。

test.support.**sortdict**(*dict*)

返回 *dict* 按键排序的 repr。

test.support.**findfile**(*filename*, *subdir=None*)

返回名为 *filename* 的文件的路径。如果未找到匹配结果则返回 *filename*。这并不等于失败因为它也算是该文件的路径。

设置 *subdir* 指明要用来查找文件的相对路径而不是直接在路径目录中查找。

test.support.**get_pagesize**()

获取以字节表示的分页大小。

Added in version 3.12.

test.support.**setswitchinterval**(*interval*)

将*sys.setswitchinterval()* 设为给定的 *interval*。请为 Android 系统定义一个最小间隔以防止系统挂起。

test.support.**check_impl_detail**(***guards*)

使用此检测来保护 CPython 实现专属的测试或者仅在有这些参数保护的实现上运行它们。此函数将根据主机系统平台的不同返回 `True` 或 `False`。用法示例:

```
check_impl_detail()               # 仅限 CPython (默认)。
check_impl_detail(jython=True)    # 仅限 Jython。
check_impl_detail(cpython=False)  # 除 CPython 以外的任何地方。
```

test.support.**set_memlimit**(*limit*)

针对大内存测试设置*max_memuse* 和*real_max_memuse* 的值。

test.support.**record_original_stdout**(*stdout*)

存放来自 *stdout* 的值。它会在回归测试开始时处理 stdout。

test.support.**get_original_stdout**()

返回*record_original_stdout()* 所设置的原始 stdout 或者如果未设置则为 `sys.stdout`。

test.support.**args_from_interpreter_flags**()

返回在 `sys.flags` 和 `sys.warnoptions` 中重新产生当前设置的命令行参数列表。

test.support.**optim_args_from_interpreter_flags**()

返回在 `sys.flags` 中重新产生当前优化设置的命令行参数列表。

test.support.**captured_stdin**()

test.support.**captured_stdout**()

test.support.**captured_stderr**()

使用*io.StringIO* 对象临时替换指定流的上下文管理器。

使用输出流的示例:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

使用输入流的示例:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # 调用接受 sys.stdin 的代码
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**disable_faulthandler**()

临时禁用*faulthandler* 的上下文管理器。

test.support.**gc_collect**()

强制收集尽可能多的对象。这是有必要的因为垃圾回收器并不能保证及时回收资源。这意味着 __del__ 方法的调用可能会晚于预期而弱引用的存活长于预期。

test.support.**disable_gc**()

在进入时禁用垃圾回收器的上下文管理器。在退出时，垃圾回收器将恢复到先前状态。

test.support.**swap_attr**(*obj*, *attr*, *new_val*)

上下文管理器用一个新对象来交换一个属性。

用法：

```
with swap_attr(obj, "attr", 5):
    ...
```

这将把 obj.attr 设为 5 并在 with 语句块内保持，在语句块结束时恢复旧值。如果 attr 不存在于 obj 中，它将被创建并在语句块结束时被删除。

旧值 (或者如果不存在旧值则为 None) 将被赋给"as" 子句的目标，如果存在子句的话。

test.support.**swap_item**(*obj*, *attr*, *new_val*)

上下文件管理器用一个新对象来交换一个条目。

用法：

```
with swap_item(obj, "item", 5):
    ...
```

这将把 obj["item"] 设为 5 并在 with 语句块内保持，在语句块结束时恢复旧值。如果 item 不存在于 obj 中，它将被创建并在语句块结束时被删除。

旧值 (或者如果不存在旧值则为 None) 将被赋给"as" 子句的目标，如果存在子句的话。

test.support.**flush_std_streams**()

在*sys.stdout* 然后又在*sys.stderr* 上调用 flush() 方法。它可被用来确保日志顺序在写入到 stderr 之前的一致性。

Added in version 3.11.

test.support.**print_warning**(*msg*)

打印一个警告到*sys.__stderr__*。将消息格式化为: f"Warning -- {msg}"。如果 *msg* 包含多行，则为每行添加 "Warning -- " 前缀。

Added in version 3.9.

test.support.**wait_process**(*pid*, *, *exitcode*, *timeout=None*)

等待直到进程 *pid* 结束并检查进程退出代码是否为 *exitcode*。

如果进程退出代码不等于 *exitcode* 则引发*AssertionError*。

如果进程运行时长超过 *timeout* 秒 (默认为*SHORT_TIMEOUT*)，则杀死进程并引发*AssertionError*。超时特性在 Windows 上不可用。

Added in version 3.9.

test.support.**calcobjsize**(*fmt*)

返回 PyObject 的大小，其结构成员由 *fmt* 定义。返回的值包括 Python 对象头的大小和对齐方式。

test.support.**calcvobjsize**(*fmt*)

返回 PyVarObject 的大小，其结构成员由 *fmt* 定义。返回的值包括 Python 对象头的大小和对齐方式。

test.support.**checksizeof**(*test*, *o*, *size*)

对于测试用例 *test*，断言 *o* 的 sys.getsizeof 加 GC 头的大小等于 *size*。

@test.support.**anticipate_failure**(*condition*)

一个有条件地用*unittest.expectedFailure()* 来标记测试的装饰器。任何对此装饰器的使用都应当具有标识相应追踪事项的有关联注释。

test.support.**system_must_validate_cert**(*f*)

一个在 TLS 证书验证失败时跳过被装饰测试的装饰器。

@test.support.**run_with_locale**(*catstr*, **locales*)

一个在不同语言区域下运行函数的装饰器，并在其结束后正确地重置语言区域。*catstr* 是字符串形式的语言区域类别 (例如 "LC_ALL")。传入的 *locales* 将依次被尝试，并将使用第一个有效的语言区域。

@test.support.**run_with_tz**(*tz*)

一个在指定时区下运行函数的装饰器，并在其结束后正确地重置时区。

@test.support.**requires_freebsd_version**(**min_version*)

当在 FreeBSD 上运行测试时指定最低版本的装饰器。如果 FreeBSD 版本号低于指定值，测试将被跳过。

@test.support.**requires_linux_version**(**min_version*)

当在 Linux 上运行测试时指定最低版本的装饰器。如果 Linux 版本号低于指定值，测试将被跳过。

@test.support.**requires_mac_version**(**min_version*)

当在 macOS 上运行测试时指定最低版本的装饰器。如果 macOS 版本号低于指定值，测试将被跳过。

@test.support.**requires_gil_enabled**

在自由线程编译版上跳过测试的装饰器。如果禁用了*GIL*，测试将被跳过。

@test.support.**requires_IEEE_754**

用于在非 non-IEEE 754 平台上跳过测试的装饰器。

@test.support.**requires_zlib**

用于当*zlib* 不存在时跳过测试的装饰器。

@test.support.**requires_gzip**

用于当*gzip* 不存在时跳过测试的装饰器。

@test.support.**requires_bz2**

用于当*bz2* 不存在时跳过测试的装饰器。

@test.support.**requires_lzma**

用于当*lzma* 不存在时跳过测试的装饰器。

@test.support.**requires_resource**(*resource*)

用于当 *resource* 不可用时跳过测试的装饰器。

@test.support.**requires_docstrings**

用于仅当*HAVE_DOCSTRINGS* 时才运行测试的装饰器。

@test.support.**requires_limited_api**

设置仅在 受限 C API 可用时运行测试的装饰器。

@test.support.**cpython_only**

表示仅适用于 CPython 的测试的装饰器。

@test.support.**impl_detail**(*msg=None*, ***guards*)

用于在 *guards* 上唤起*check_impl_detail()* 的装饰器。如果调用返回 False，则使用 *msg* 作为跳过测试的原因。

@test.support.**no_tracing**

用于在测试期间临时关闭追踪的装饰器。

@test.support.**refcount_test**

用于涉及引用计数的测试的装饰器。如果测试不是由 CPython 运行则该装饰器不会运行测试。在测试期间会取消设置任何追踪函数以由追踪函数导致的意外引用计数。

@test.support.**bigmemtest**(*size*, *memuse*, *dry_run=True*)

用于大内存测试的装饰器。

size 是测试所请求的大小（以任意的，由测试解读的单位。）*memuse* 是测试的每单元字节数，或是对它的良好估计。例如，一个需要两个字节缓冲区，每个缓冲区 4 GiB，则可以用 `@bigmemtest(size=_4G, memuse=2)` 来装饰。

size 参数通常作为额外参数传递给被测试的方法。如果 *dry_run* 为 `True`，则传给测试方法的值可能少于所请求的值。如果 *dry_run* 为 `False`，则意味着当当未指定 `-M` 时测试将不支持虚拟运行。

@test.support.**bigaddrspacetest**

用于填充地址空间的测试的装饰器。

test.support.**check_syntax_error**(*testcase*, *statement*, *errtext=''*, ***, *lineno=None*, *offset=None*)

用于通过尝试编译 *statement* 来测试 *statement* 中的语法错误。*testcase* 是测试的 *unittest* 实例。*errtext* 是应当匹配所引发的 *SyntaxError* 的字符串表示形式的正则表达式。如果 *lineno* 不为 `None`，则与异常所在的行进行比较。如果 *offset* 不为 `None`，则与异常的偏移量进行比较。

test.support.**open_urlresource**(*url*, **args*, ***kw*)

打开 *url*。如果打开失败，则引发 *TestFailed*。

test.support.**reap_children**()

只要有子进程启动就在 `test_main` 的末尾使用此函数。这将有助于确保没有多余的子进程（僵尸）存在占用资源并在查找引用泄漏时造成问题。

test.support.**get_attribute**(*obj*, *name*)

获取一个属性，如果引发了 *AttributeError* 则会引发 *unittest.SkipTest*。

test.support.**catch_unraisable_exception**()

使用 *sys.unraisablehook()* 来捕获不可引发的异常的上下文管理器。

存储异常值 (`cm.unraisable.exc_value`) 会创建一个引用循环。引用循环将在上下文管理器退出时被显式地打破。

存储对象 (`cm.unraisable.object`) 如果被设置为一个正在最终化的对象则可以恢复它。退出上下文管理器将清除已存在对象。

用法：

```
with support.catch_unraisable_exception() as cm:
    # 创建一个“不可引发的异常”的代码
    ...

    # 检测这个不可引发的异常：使用 cm.unraisable
    ...

# 此时 cm.unraisable 属性已不存在
# （以打破循环引用）
```

Added in version 3.8.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

在测试包中使用的 *unittest* `load_tests` 协议的通用实现。*pkg_dir* 是包的根目录；*loader*, *standard_tests* 和 *pattern* 是 `load_tests` 所期望的参数。在简单的情况下，测试包的 `__init__.py` 可以是下面这样的:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)

返回未在 *other_api* 中找到的 *ref_api* 的属性、函数或方法的集合，除去在 *ignore* 中指明的要在这个检查中忽略的已定义条目列表。

在默认情况下这将跳过以'_' 打头的私有属性但包括所有魔术方法，即以'__' 打头和结尾的方法。

Added in version 3.5.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)

用 *new_value* 重载 *object_to_patch.attr_name*。并向 *test_instance* 添加清理过程以便为 *attr_name* 恢复 *object_to_patch*。*attr_name* 应当是 *object_to_patch* 的一个有效属性。

test.support.**run_in_subinterp**(*code*)

在子解释器中运行 *code*。如果启用了*tracemalloc* 则会引发*unittest.SkipTest*。

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)

断言 *cls* 的实例在迭代后被释放。

test.support.**missing_compiler_executable**(*cmd_names=[]*)

检查在 *cmd_names* 中列出名称的或者当 *cmd_names* 为空时所有的编译器可执行文件是否存在并返回第一个丢失的可执行文件或者如果未发现任何丢失则返回 None。

test.support.**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *not_exported=()*)

断言 *module* 的 __all__ 变量包含全部公共名称。

模块的公共名称（它的 API）是根据它们是否符合公共名称惯例并在 *module* 中被定义来自动检测的。

name_of_module 参数可以（用字符串或元组的形式）指定一个 API 可以被定义为什么模块以便被检测为一个公共 API。一种这样的情况会在 *module* 从其他模块，可能是一个 C 后端 (如 csv 和它的 _csv) 导入其公共 API 的某一组成部分时发生。

extra 参数可以是一个在其他情况下不会被自动检测为"public" 的名称的集合，例如没有适当的*__module__* 属性的对象。如果提供该参数，它将被添加到被自动检测的对象中。

not_exported 参数可以是一个不可被当作公共 API 的一部分的名称集合，即使其名称没有显式指明这一点。

用法示例:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        not_exported = {'baz'}  # 未写入文档的名称。
        # bar 从 _bar 导入其 API 的一部分。
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, not_exported=not_exported)
```

Added in version 3.6.

test.support.**skip_if_broken_multiprocessing_synchronize**()

如果没有 multiprocessing.synchronize 模块，没有可用的 semaphore 实现，或者如果创建一个锁会引发*OSError* 则跳过测试。

Added in version 3.10.

test.support.**check_disallow_instantiation**(*test_case*, *tp*, **args*, ***kwds*)

断言类型 *tp* 不能使用 *args* 和 *kwds* 来实例化。

Added in version 3.10.

test.support.**adjust_int_max_str_digits**(*max_digits*)

此函数返回一个将在上下文生效期间改变全局*sys.set_int_max_str_digits()* 设置的上下文管理器以便允许执行当在整数和字符串之间进行转换时需要对位数有不同限制的测试代码。

Added in version 3.11.

test.support 模块定义了以下的类:

class test.support.**SuppressCrashReport**

一个用于在预期会使子进程崩溃的测试时尽量防止弹出崩溃对话框的上下文管理器。

在 Windows 上，它会使用 SetErrorMode 来禁用 Windows 错误报告对话框。

在 UNIX 上，会使用*resource.setrlimit()* 来将*resource.RLIMIT_CORE* 的软限制设为 0 以防止创建核心转储文件。

在这两个平台上，旧值都可通过 __exit__() 恢复。

class test.support.**SaveSignals**

用于保存和恢复由 Python 句柄的所注册的信号处理器。

save(*self*)

将信号处理器保存到一个将信号编号映射到当前信号处理器的字典。

restore(*self*)

将来自*save()* 字典的信号编号设置到已保存的处理器上。

class test.support.**Matcher**

matches(*self*, *d*, ***kwargs*)

尝试对单个字典与所提供的参数进行匹配。

match_value(*self*, *k*, *dv*, *v*)

尝试对单个已存储值 (*dv*) 与所提供的值 (*v*) 进行匹配。

27.10 test.support.socket_helper --- 用于套接字测试的工具

test.support.socket_helper 模块提供了对套接字测试的支持。

Added in version 3.9.

test.support.socket_helper.**IPV6_ENABLED**

设置为 True 如果主机打开 IPv6, 否则 False .

test.support.socket_helper.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)

返回一个应当适合绑定的未使用端口。这是通过创建一个与 sock 形参相同协议族和类型的临时套接字来达成的 (默认为*AF_INET*, *SOCK_STREAM*)，并将其绑定到指定的主机地址 (默认为 0.0.0.0) 并将端口设为 0，以从 OS 引出一个未使用的瞬时端口。这个临时套接字随后将被关闭并删除，然后返回该瞬时端口。

这个方法或*bind_port()* 应当被用于任何在测试期间需要绑定到特定端口的测试。具体使用哪个取决于调用方代码是否会创建 Python 套接字，或者是否需要在构造器中提供或向外部程序提供未

使用的端口（例如传给 openssl 的 s_server 模式的 `-accept` 参数）。在可能的情况下将总是优先使用*bind_port()* 而非*find_unused_port()*。不建议使用硬编码的端口因为将使测试的多个实例无法同时运行，这对 buildbot 来说是个问题。

test.support.socket_helper.**bind_port**(*sock*, *host=HOST*)

将套接字绑定到一个空闲端口并返回端口号。这依赖于瞬时端口以确保我们能使用一个未绑定端口。这很重要因为可能会有许多测试同时运行，特别是在 buildbot 环境中。如果 `sock.family` 为*AF_INET* 而 `sock.type` 为*SOCK_STREAM*，并且套接字上设置了 `SO_REUSEADDR` 或 `SO_REUSEPORT` 则此方法将引发异常。测试绝不应该为 TCP/IP 套接字设置这些套接字选项。唯一需要设置这些选项的情况是通过多个 UDP 套接字来测试组播。

此外，如果 `SO_EXCLUSIVEADDRUSE` 套接字选项是可用的（例如在 Windows 上），它将在套接字上被设置。这将阻止其他任何人在测试期间绑定到我们的主机/端口。

test.support.socket_helper.**bind_unix_socket**(*sock*, *addr*)

绑定一个 Unix 套接字，如果*PermissionError* 被引发则会引发*unittest.SkipTest*。

@test.support.socket_helper.**skip_unless_bind_unix_socket**

一个用于运行需要 Unix 套接字 `bind()` 功能的测试的装饰器。

test.support.socket_helper.**transient_internet**(*resource_name*, ***, *timeout=30.0*, *errnos=()*)

一个在互联网连接的各种问题以异常的形式表现出来时会引发*ResourceDenied* 的上下文管理器。

27.11 test.support.script_helper --- 用于 Python 执行测试工具

test.support.script_helper 模块提供了对 Python 的脚本执行测试的支持。

test.support.script_helper.**interpreter_requires_environment**()

如果 `sys.executable interpreter` 需要环境变量才能运行则返回 `True`。

这被设计用来配合 `@unittest.skipIf()` 以便标注需要使用 to annotate tests that need to use an `assert_python*()` 函数来启动隔离模式 (`-I`) 或无环境模式 (`-E`) 子解释器的测试。

正常的编译和测试运行不会进入这种状况但它在尝试从一个使用 Python 的当前家目录查找逻辑找不到明确的家目录的解释器运行标准库测试套件时有可能发生。

设置 `PYTHONHOME` 是一种能让大多数测试套件在这种情况下运行的办法。`PYTHONPATH` 或 `PYTHONUSERSITE` 是另外两个可影响解释器是否能启动的常见环境变量。

test.support.script_helper.**run_python_until_end**(**args*, ***env_vars*)

基于 *env_vars* 设置环境以便在子进程中运行解释器。它的值可以包括 `__isolated`, `__cleanenv`, `__cwd`, and `TERM`。

在 3.9 版本发生变更: 此函数不会再从 *stderr* 去除空格符。

test.support.script_helper.**assert_python_ok**(**args*, ***env_vars*)

断言附带 *args* 和可选的环境变量 *env_vars* 运行解释器会成功 (`rc == 0`) 并返回一个 `(return code, stdout, stderr)` 元组。

如果设置了 *__cleanenv* 仅限关键字形参，*env_vars* 会被用作一个全新的环境。

Python 是以隔离模式 (命令行选项 `-I`) 启动的，除非 *__isolated* 仅限关键字形参被设为 `False`。

在 3.9 版本发生变更: 此函数不会再从 *stderr* 去除空格符。

test.support.script_helper.**assert_python_failure**(**args*, ***env_vars*)

断言附带 *args* 和可选的环境变量 *env_vars* 运行解释器会失败 (`rc != 0`) 并返回一个 `(return code, stdout, stderr)` 元组。

更多选项请参阅*assert_python_ok()*。

在 3.9 版本发生变更: 此函数不会再从 *stderr* 去除空格符。

test.support.script_helper.**spawn_python**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)

使用给定的参数运行一个 Python 子进程。

kw 是要传给*subprocess.Popen()* 的额外关键字参数。返回一个*subprocess.Popen* 对象。

test.support.script_helper.**kill_python**(*p*)

运行给定的*subprocess.Popen* 进程直至完成并返回 stdout。

test.support.script_helper.**make_script**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)

在路径 *script_dir* 和 *script_basename* 中创建包含 *source* 的脚本。如果 *omit_suffix* 为 `False`，则为名称添加 `.py`。返回完整的脚本路径。

test.support.script_helper.**make_zip_script**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)

使用 *zip_dir* 和 *zip_basename* 创建扩展名为 `zip` 的 zip 文件，其中包含 *script_name* 中的文件。*name_in_zip* 为归档名。返回一个包含 `(full path, full path of archive name)` 的元组。

test.support.script_helper.**make_pkg**(*pkg_dir*, *init_source=''*)

创建一个名为 *pkg_dir* 的目录，其中包含一个 `__init__` 文件并以 *init_source* 作为其内容。

test.support.script_helper.**make_zip_pkg**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)

使用 *zip_dir* 和 *zip_basename* 创建一个 zip 包目录，其中包含一个空的 `__init__` 文件和一个包含 *source* 的文件 *script_basename*。如果 *compiled* 为 `True`，则两个源文件将被编译并添加到 zip 包中。返回一个以完整 zip 路径和 zip 文件归档名为元素的元组。

27.12 test.support.bytecode_helper --- 用于测试正确字节码生成的支持工具

test.support.bytecode_helper 模块提供了对测试和检查字节码生成的支持。

Added in version 3.9.

The module defines the following class:

class test.support.bytecode_helper.**BytecodeTestCase**(*unittest.TestCase*)

这个类具有用于检查字节码的自定义断言。

BytecodeTestCase.**get_disassembly_as_string**(*co*)

以字符串形式返回 *co* 的汇编码。

BytecodeTestCase.**assertInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

如果找到 *opname* 则返回 instr，否则抛出*AssertionError*。

BytecodeTestCase.**assertNotInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)

如果找到 *opname* 则抛出*AssertionError*。

27.13 test.support.threading_helper --- 用于线程测试的工具

test.support.threading_helper 模块提供了对线程测试的支持。

Added in version 3.10.

test.support.threading_helper.**join_thread**(*thread*, *timeout=None*)

在 *timeout* 秒之内合并一个 *thread*。如果线程在 *timeout* 秒后仍然存活则引发*AssertionError*。

@test.support.threading_helper.**reap_threads**

用于确保即使测试失败线程仍然会被清理的装饰器。

test.support.threading_helper.**start_threads**(*threads*, *unlock=None*)

启动 *threads* 的上下文管理器，该参数为一个线程序列。*unlock* 是一个在所有线程启动之后被调用的函数，即使引发了异常也会执行；一个例子是*threading.Event.set()*。start_threads 将在退出时尝试合并已启动的线程。

test.support.threading_helper.**threading_cleanup**(**original_values*)

清理未在 *original_values* 中指定的线程。被设计为如果有一个测试在后台离开正在运行的线程时会发出警告。

test.support.threading_helper.**threading_setup**()

返回当前线程计数和悬空线程的副本。

test.support.threading_helper.**wait_threads_exit**(*timeout=None*)

等待直到 with 语句中所有已创建线程退出的上下文管理器。

test.support.threading_helper.**catch_threading_exception**()

使用*threading.excepthook()* 来捕获*threading.Thread* 异常的上下文管理器。

当异常被捕获时要设置的属性:

- exc_type
- exc_value
- exc_traceback
- thread

参见*threading.excepthook()* 文档。

这些属性在上下文管理器退出时将被删除。

用法:

```
with threading_helper.catch_threading_exception() as cm:
    # 生成一个引发异常的线程的代码
    ...

    # 检测这个线程异常，使用 cm 的属性:
    # exc_type, exc_value, exc_traceback, thread
    ...

# 此时 cm 的 exc_type, exc_value, exc_traceback, thread 属性
# 已不存在
# （以避免循环引用）
```

Added in version 3.8.

27.14 test.support.os_helper --- 用于操作系统测试的工具

test.support.os_helper 模块提供了对操作系统测试的支持。

Added in version 3.10.

test.support.os_helper.**FS_NONASCII**

一个可通过*os.fsencode()* 编码的非 ASCII 字符。

test.support.os_helper.**SAVEDCWD**

设置为*os.getcwd()*。

test.support.os_helper.**TESTFN**

设置为一个可以安全地用作临时文件名的名称。任何被创建的临时文件都应当被关闭和撤销链接（移除）。

test.support.os_helper.**TESTFN_NONASCII**

如果存在的话，设置为一个包含*FS_NONASCII* 字符的文件名。这会确保当文件名存在时，它可使用默认文件系统编码格式来编码和解码。这允许需要非 ASCII 文件名的测试在其不可用的平台上被方便地跳过。

test.support.os_helper.**TESTFN_UNENCODABLE**

设置为一个应当在严格模式下不可使用文件系统编码格式来编码的文件名（str 类型）。如果无法生成这样的文件名则可以为 None。

test.support.os_helper.**TESTFN_UNDECODABLE**

设置为一个应当在严格模式下不可使用文件系统编码格式来编码的文件名（bytes 类型）。如果无法生成这样的文件名则可以为 None。

test.support.os_helper.**TESTFN_UNICODE**

设置为用于临时文件的非 ASCII 名称。

class test.support.os_helper.**EnvironmentVarGuard**

用于临时性地设置或取消设置环境变量的类。其实例可被用作上下文管理器并具有完整的字典接口用来查询/修改下层的 os.environ。在从上下文管理器退出之后所有通过此实例对环境变量进行的修改都将被回滚。

在 3.1 版本发生变更: 增加了字典接口。

class test.support.os_helper.**FakePath**(*path*)

简单的*path-like object*。它实现了返回 *path* 参数的*__fspath__()* 方法。如果 *path* 是一个异常，它将在 __fspath__() 中被引发。

EnvironmentVarGuard.**set**(*envvar*, *value*)

临时性地将环境变量 envvar 的值设为 value。

EnvironmentVarGuard.**unset**(*envvar*)

临时性地取消设置环境变量 envvar。

test.support.os_helper.**can_symlink**()

如果操作系统支持符号链接则返回 True，否则返回 False。

test.support.os_helper.**can_xattr**()

如果操作系统支持 xattr 支返回 True，否则返回 False。

test.support.os_helper.**change_cwd**(*path*, *quiet=False*)

一个临时性地将当前工作目录改为 *path* 并输出该目录的上下文管理器。

如果 *quiet* 为 False，此上下文管理器将在发生错误时引发一个异常。在其他情况下，它将只发出一个警告并将当前工作目录保持原状。

test.support.os_helper.**create_empty_file**(*filename*)

创建一个名为 *filename* 的空文件。如果文件已存在，则清空其内容。

test.support.os_helper.**fd_count**()

统计打开的文件描述符数量。

test.support.os_helper.**fs_is_case_insensitive**(*directory*)

如果 *directory* 的文件系统对大小写敏感则返回 True。

test.support.os_helper.**make_bad_fd**()

通过打开并关闭临时文件来创建一个无效的文件描述符，并返回其描述器。

test.support.os_helper.**rmdir**(*filename*)

在 *filename* 上调用*os.rmdir()*。在 Windows 平台上，这将使用一个检测文件是否存在的等待循环来包装，需要这样做是因为反病毒程序会保持文件打开并阻止其被删除。

test.support.os_helper.**rmtree**(*path*)

在 *path* 上调用 *shutil.rmtree()* 或者调用 *os.lstat()* 和 *os.rmdir()* 来移除一个路径及其内容。与 *rmdir()* 一样，在 Windows 平台上这将使用一个检测文件是否存在的等待循环来包装。

@test.support.os_helper.**skip_unless_symlink**

一个用于运行需要符号链接支持的测试的装饰器。

@test.support.os_helper.**skip_unless_xattr**

一个用于运行需要 xattr 支持的测试的装饰器。

test.support.os_helper.**temp_cwd**(*name='tempcwd'*, *quiet=False*)

一个临时性地创建新目录并改变当前工作目录（CWD）的上下文管理器。

临时性地改变当前工作目录之前此上下文管理器会在当前目录下创建一个名为 *name* 的临时目录。如果 *name* 为 None，则会使用 *tempfile.mkdtemp()* 创建临时目录。

如果 *quiet* 为 False 并且无法创建或修改 CWD，则会引发一个错误。在其他情况下，只会引发一个警告并使用原始 CWD。

test.support.os_helper.**temp_dir**(*path=None*, *quiet=False*)

一个在 *path* 上创建临时目录并输出该目录的上下文管理器。

如果 *path* 为 None，则会使用 *tempfile.mkdtemp()* 来创建临时目录。如果 *quiet* 为 False，则该上下文管理器在发生错误时会引发一个异常。在其他情况下，如果 *path* 已被指定并且无法创建，则只会发出一个警告。

test.support.os_helper.**temp_umask**(*umask*)

一个临时性地设置进程掩码的上下文管理器。

test.support.os_helper.**unlink**(*filename*)

在 *filename* 上调用 *os.unlink()*。与 *rmdir()* 一样，在 Windows 平台上这将使用一个检测文本是否存在的等待循环来包装。

27.15 test.support.import_helper --- 用于导入测试的工具

test.support.import_helper 模块提供了对导入测试的支持。

Added in version 3.10.

test.support.import_helper.**forget**(*module_name*)

从 sys.modules 移除名为 *module_name* 的模块并删除该模块的已编译字节码文件。

test.support.import_helper.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)

此函数会在执行导入之前通过从 sys.modules 移除指定模块来导入并返回指定 Python 模块的新副本。请注意这不同于 reload()，原来的模块不会受到此操作的影响。

fresh 是包含在执行导入之前还要从 sys.modules 缓存中移除的附加模块名称的可迭代对象。

blocked 是包含模块名称的可迭代对象，导入期间在模块缓存中它会被替换为 None 以确保尝试导入将引发 *ImportError*。

指定名称的模块以及任何在 *fresh* 和 *blocked* 形参中指明的模块会在开始导入之前被保存并在全新导入完成时被重新插入到 sys.modules 中。

如果 *deprecated* 为 True 则在此导入操作期间模块和包的弃用消息会被屏蔽。

如果指定名称的模块无法被导入则此函数将引发 *ImportError*。

用法示例:

```
# 获取 warnings 模块的副本用于测试而会不影响
# 测试套件的其他部分所使用的版本。 一个副本
# 使用 C 实现，另一个被强制使用纯 Python 的
# 回退实现
```

(续下页)

(接上页)

```
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

Added in version 3.1.

test.support.import_helper.**import_module**(*name*, *deprecated=False*, *, *required_on=()*)

此函数会导入并返回指定名称的模块。不同于正常的导入，如果模块无法被导入则此函数将引发*unittest.SkipTest*。

如果 *deprecated* 为 `True` 则在此导入操作期间模块和包的弃用消息会被屏蔽。如果某个模块在特定平台上是必需的而在其他平台上是可选的，请为包含平台前缀的可迭代对象设置 *required_on*，此对象将与*sys.platform* 进行比对。

Added in version 3.1.

test.support.import_helper.**modules_setup**()

返回*sys.modules* 的副本。

test.support.import_helper.**modules_cleanup**(*oldmodules*)

移除 *oldmodules* 和 `encodings` 以外的模块以保留内部缓冲区。

test.support.import_helper.**unload**(*name*)

从 `sys.modules` 中删除 *name*。

test.support.import_helper.**make_legacy_pyc**(*source*)

将 **PEP 3147/PEP 488** pyc 文件移至旧版 pyc 位置并返回该旧版 pyc 文件的文件系统路径。*source* 的值是源文件的文件系统路径。它不必真实存在，但是 PEP 3147/488 pyc 文件必须存在。

class test.support.import_helper.**CleanImport**(**module_names*)

强制导入以返回一个新的模块引用的上下文管理器。这适用于测试模块层级的行为，例如在导入时发出*DeprecationWarning*。示例用法:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # 新引用
```

class test.support.import_helper.**DirsOnSysPath**(**paths*)

一个临时性地向*sys.path* 添加目录的上下文管理器。

这将创建*sys.path* 的一个副本，添加作为位置参数传入的任何目录，然后在上下文结束时将*sys.path* 还原到副本的设置。

请注意该上下文管理器代码块中 *所有*对*sys.path* 的修改，包括对象的替换，都将在代码块结束时被还原。

27.16 test.support.warnings_helper --- 用于警告测试的工具

test.support.warnings_helper 模块提供了对警告测试的支持。

Added in version 3.10.

test.support.warnings_helper.**ignore_warnings**(*, *category*)

抑制作为 *category* 实例的警告，它必须为*Warning* 或其子类。大致等价于*warnings.catch_warnings()* 设置*warnings.simplefilter('ignore', category=category)*。例如:

```
@warning_helper.ignore_warnings(category=DeprecationWarning)
def test_suppress_warning():
    # 做些什么
```

Added in version 3.8.

test.support.warnings_helper.**check_no_resource_warning**(*testcase*)

检测是否没有任何*ResourceWarning* 被引发的上下文管理器。你必须在该上下文管理器结束之前移除可能发出*ResourceWarning* 的对象。

test.support.warnings_helper.**check_syntax_warning**(*testcase*, *statement*, *errtext=''*, ***, *lineno=1*, *offset=None*)

用于通过尝试编译 *statement* 来测试 *statement* 中的语法警告。还会测试*SyntaxWarning* 是否只发出了一次，以及它在转成错误时是否将被转换为*SyntaxError*。*testcase* 是用于测试的*unittest* 实例。*errtext* 是应当匹配所发出的*SyntaxWarning* 以及所引发的*SyntaxError* 的字符串表示形式的正则表达式。如果 *lineno* 不为 `None`，则与警告和异常所在的行进行比较。如果 *offset* 不为 `None`，则与异常的偏移量进行比较。

Added in version 3.8.

test.support.warnings_helper.**check_warnings**(**filters*, *quiet=True*)

一个用于*warnings.catch_warnings()* 以更容易地测试特定警告是否被正确引发的便捷包装器。它大致等价于调用 `warnings.catch_warnings(record=True)` 并将*warnings.simplefilter()* 设为 `always` 并附带自动验证已记录结果的选项。

`check_warnings` 接受 `("message regexp", WarningCategory)` 形式的 2 元组作为位置参数。如果提供了一个或多个 *filters*，或者如果可选的关键字参数 *quiet* 为 `False`，则它会检查确认警告是符合预期的：每个已指定的过滤器必须匹配至少一个被包围的代码或测试失败时引发的警告，并且如果有任何未能匹配已指定过滤器的警告被引发则测试将失败。要禁用这些检查中的第一项，请将 *quiet* 设为 `True`。

如果未指定任何参数，则默认为:

```
check_warnings(("", Warning), quiet=True)
```

在此情况下所有警告都会被捕获而不会引发任何错误。

在进入该上下文管理器时，将返回一个 `WarningRecorder` 实例。来自*catch_warnings()* 的下层警告列表可通过该记录器对象的*warnings* 属性来访问。作为一个便捷方式，该对象中代表最近的警告的属性也可通过该记录器对象来直接访问（参见以下示例）。如果未引发任何警告，则在其他情况下预期代表一个警告的任何对象属性都将返回 `None`。

该记录器对象还有一个 `reset()` 方法，该方法会清空警告列表。

该上下文管理器被设计为像这样来使用:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

在此情况下如果两个警告都未被引发，或是引发了其他的警告，则*check_warnings()* 将会引发一个错误。

当一个测试需要更深入地查看这些警告，而不是仅仅检查它们是否发生时，可以使用这样的代码:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

在这里所有的警告都将被捕获，而测试代码会直接测试被捕获的警告。

在 3.2 版本发生变更: 新增可选参数 *filters* 和 *quiet*。

class test.support.warnings_helper.**WarningsRecorder**

用于为单元测试记录警告的类。请参阅以上*check_warnings()* 的文档来了解详情。

CHAPTER 28

调试和分析

这些库可以帮助你进行 Python 开发：调试器使你能够逐步执行代码，分析堆栈帧并设置中断点等等，性能分析器可以运行代码并为你提供执行时间的详细数据，使你能够找出你的程序中的瓶颈。审计事件提供运行时行为的可见性，如果没有此工具则需要进行侵入式调试或修补。

28.1 审计事件表

下表包含了在整个 CPython 运行时和标准库中由 *sys.audit()* 或 PySys_Audit() 调用所引发的全部事件。这些调用是在 3.8 或更高版本中添加的 (参见 **PEP 578**)。

请参阅 *sys.addaudithook()* 和 PySys_AddAuditHook() 了解有关处理这些事件的详细信息。

此表是根据 CPython 文档生成的，可能无法表示其他实现所引发的事件。请参阅你的运行时专属的文档了解实际引发的事件。

Audit event	Arguments
_thread.start_new_thread	function, args, kwargs
array.__new__	typecode, initializer
builtins.breakpoint	breakpointhook
builtins.id	id
builtins.input	prompt
builtins.input/result	result
code.__new__	code, filename, name, argcount, posonlyargcount, kwonlyargcount, nlocals, st
compile	source, filename
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.run_command	command
cpython.run_file	filename
cpython.run_interactivehook	hook
cpython.run_module	module-name
cpython.run_startup	filename
cpython.run_stdin	
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address

表 1 – 接上页

Audit event	Arguments
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.set_errno	errno
ctypes.set_exception	code
ctypes.set_last_error	error
ctypes.string_at	ptr, size
ctypes.wstring_at	ptr, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
glob.glob/2	pathname, recursive, root_dir, dir_fd
http.client.connect	self, host, port
http.client.send	self, data
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
marshal.dumps	value, version
marshal.load	
marshal.loads	bytes
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
object.__delattr__	obj, name
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	path, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	
os.fwalk	top, topdown, onerror, follow_symlinks, dir_fd
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig

表 1 – 接上页

Audit event	Arguments
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listdrives	
os.listmounts	volume
os.listvolumes	
os.listxattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.startfile/2	path, operation, arguments, cwd, show_cmd
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
os.walk	top, topdown, onerror, followlinks
pathlib.Path.glob	self, pattern
pathlib.Path.rglob	self, pattern
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path, dir_fd
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data
socket.__new__	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address
socket.getaddrinfo	host, port, family, type, protocol
socket.gethostbyaddr	ip_address
socket.gethostbyname	hostname
socket.gethostname	
socket.getnameinfo	sockaddr

表 1 – 接上页

Audit event	Arguments
socket.getservbyname	`servicename`, `protocolname`
socket.getservbyport	`port`, `protocolname`
socket.sendmsg	`self`, `address`
socket.sendto	`self`, `address`
socket.sethostname	`name`
sqlite3.connect	`database`
sqlite3.connect/handle	`connection_handle`
sqlite3.enable_load_extension	`connection`, `enabled`
sqlite3.load_extension	`connection`, `path`
subprocess.Popen	`executable`, `args`, `cwd`, `env`
sys._current_exceptions	
sys._current_frames	
sys._getframe	`frame`
sys._getframemodulename	`depth`
sys.addaudithook	
sys.excepthook	`hook`, `type`, `value`, `traceback`
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	`hook`, `unraisable`
syslog.closelog	
syslog.openlog	`ident`, `logoption`, `facility`
syslog.setlogmask	`maskpri`
syslog.syslog	`priority`, `message`
tempfile.mkdtemp	`fullpath`
tempfile.mkstemp	`fullpath`
time.sleep	`secs`
urllib.Request	`fullurl`, `data`, `headers`, `method`
webbrowser.open	`url`
winreg.ConnectRegistry	`computer_name`, `key`
winreg.CreateKey	`key`, `sub_key`, `access`
winreg.DeleteKey	`key`, `sub_key`, `access`
winreg.DeleteValue	`key`, `value`
winreg.DisableReflectionKey	`key`
winreg.EnableReflectionKey	`key`
winreg.EnumKey	`key`, `index`
winreg.EnumValue	`key`, `index`
winreg.ExpandEnvironmentStrings	`str`
winreg.LoadKey	`key`, `sub_key`, `file_name`
winreg.OpenKey	`key`, `sub_key`, `access`
winreg.OpenKey/result	`key`
winreg.PyHKEY.Detach	`key`
winreg.QueryInfoKey	`key`
winreg.QueryReflectionKey	`key`
winreg.QueryValue	`key`, `sub_key`, `value_name`
winreg.SaveKey	`key`, `file_name`
winreg.SetValue	`key`, `sub_key`, `type`, `value`

下列事件只在内部被引发，而不会回应任何 CPython 公共 API:

审计事件	参数
_winapi.CreateFile	file_name, desired_access, share_mode, creation_disposition, flags_and_attributes
_winapi.CreateJunction	src_path, dst_path
_winapi.CreateNamedP	name, open_mode, pipe_mode
_winapi.CreatePipe	
_winapi.CreateProcess	application_name, command_line, current_directory
_winapi.OpenProcess	process_id, desired_access
_winapi.TerminateProc	handle, exit_code
ctypes.PyObj_FromPtr	obj

28.2 bdb --- 调试器框架

源代码: Lib/bdb.py

---

bdb 模块处理基本的调试器函数，例如设置中断点或通过调试器来管理执行。

定义了以下异常:

exception bdb.**BdbQuit**

由*Bdb* 类引发用于退出调试器的异常。

bdb 模块还定义了两个类:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=False*, *cond=None*, *funcname=None*)

这个类实现了临时性中断点、忽略计数、禁用与（重新）启用，以及条件设置等。

中断点通过一个名为*bpbynumber* 的列表基于数字并通过*bplist* 基于 (file, line) 对进行索引。前者指向一个*Breakpoint* 类的单独实例。后者指向一个由这种实例组成的列表，因为在每一行中可能存在多个中断点。

当创建一个中断点时，它所关联的文件名 应当为规范形式。如果定义了*funcname*，则当该函数的第一行被执行时将增加一次中断点命中 次数。有条件的 中断点将总是会计入命中 次数。

Breakpoint 的实例具有下列方法:

deleteMe()

从关联到文件/行的列表中删除此中断点。如果它是该位置上的最后一个中断点，还将删除相应的文件/行条目。

enable()

将此中断点标记为启用。

disable()

将此中断点标记为禁用。

bpformat()

返回一个带有关于此中断点的所有信息的，格式良好的字符串:

- 中断点编号。
- 临时状态（删除或保留）。
- 文件/行位置。
- 中断条件
- 要忽略的次数。
- 命中的次数。

Added in version 3.2.

bpprint(*out=None*)

将*bpformat()* 的输出打印到文件 *out*，或者如果为 None 则打印到标准输出。, to standard output.

Breakpoint 实例具有以下属性:

file

Breakpoint 的文件名。

line

Breakpoint 在*file* 中的行号。

temporary

如果位于 (file, line) 的*Breakpoint* 是临时性的则返回 True。

cond

在 (file, line) 上对*Breakpoint* 求值的条件。

funcname

用于定义在进入函数时一个*Breakpoint* 是否命中的函数的名称。

enabled

如果*Breakpoint* 被启用则返回 True。

bpbynumber

一个*Breakpoint* 单独实例的数字索引。

bplist

以 (*file*, *line*) 元组作为索引的*Breakpoint* 实例的字典。

ignore

忽略一个*Breakpoint* 的次数。

hits

命中一个*Breakpoint* 的次数统计。

class bdb.**Bdb**(*skip=None*)

Bdb 类是作为通用的 Python 调试器基类。

这个类负责追踪工具的细节；所派生的类应当实现用户交互。标准调试器类 (*pdb.Pdb*) 就是一个例子。

如果给出了 *skip* 参数，它必须是一个包含 glob 风格的模块名称模式的可迭代对象。调试器将不会步进到来自与这些模式相匹配的模块的帧。一个帧是否会被视为来自特定的模块是由帧的 __name__ 全局变量来确定的。

在 3.1 版本发生变更: 增加了 *skip* 形参。

Bdb 的以下方法通常不需要被重写。

canonic(*filename*)

返回 *filename* 的规范形式。

对于真实的文件名称，此规范形式是一个依赖于具体操作系统的，大小写规范的 绝对路径。在交互模式下生成的带有尖括号的 *filename*，如 "<stdin>"，会被不加修改地返回。

reset()

将 botframe, stopframe, returnframe 和*quitting* 属性设为准备开始调试的值。

trace_dispatch(*frame*, *event*, *arg*)

此函数被安装为被调试帧的追踪函数。它的返回值是新的追踪函数（在大多数情况下就是它自身）。

默认实现会决定如何分派帧，这取决于即将被执行的事件的类型（作为字符串传入）。*event* 可以是下列值之一:

- "line": 一个新的代码行即将被执行。

- "call": 一个函数即将被调用，或者进入了另一个代码块。
- "return": 一个函数或其他代码块即将返回。
- "exception": 一个异常已发生。
- "c_call": 一个 C 函数即将被调用。
- "c_return": 一个 C 函数已返回。
- "c_exception": 一个 C 函数引发了异常。

对于 Python 事件，调用了专门的函数（见下文）。对于 C 事件，不执行任何操作。

arg 形参取决于之前的事件。

请参阅*sys.settrace()* 的文档了解追踪函数的更多信息。对于代码和帧对象的详情，请参考 types。

dispatch_line(*frame*)

如果调试器应该在当前行上停止，则唤起*user_line()* 方法（该方法应当在子类中被重写）。如果设置了*quitting* 旗标（可通过*user_line()* 来设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

dispatch_call(*frame*, *arg*)

如果调试器应该在此函数调用上停止，则唤起*user_call()* 方法（该方法应当在子类中被重写）。如果设置了*quitting* 旗标（可通过*user_call()* 来设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

dispatch_return(*frame*, *arg*)

如果调试器应当在此函数调用上停止，则唤起*user_return()* 方法（该方法应当在子类中被重写）。如果设置了*quitting* 旗标（可通过*user_return()* 来设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

dispatch_exception(*frame*, *arg*)

如果调试器应当在此异常上停止，则唤起*user_exception()* 方法（该方法应当在子类中被重写）。如果设置了*quitting* 旗标（可通过*user_exception()* 设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

通常情况下派生的类不会重写下列方法，但是如果想要重新定义停止和中断点的定义则可能会重写它们。

is_skipped_line(*module_name*)

如果 *module_name* 匹配到任何跳过模式则返回 True。

stop_here(*frame*)

如果 *frame* 在栈的起始帧之下则返回 True。

break_here(*frame*)

如果该行有生效的中断点则返回 True。

检测某行或某函数是否存在中断点且处于生效状态。基于来自*effective()* 的信息删除临时中断点。

break_anywhere(*frame*)

如果存在任何针对 *frame* 的文件名的中断点则返回 True。

派生的类应当重写这些方法以获取调试器操作的控制权。

user_call(*frame*, *argument_list*)

如果中断可能在被调用的函数内停止则会从*dispatch_call()* 来调用。

argument_list 已不再使用并将始终为 None。该参数被保留用于向下兼容。

user_line(*frame*)

当*stop_here()* 或*break_here()* 返回 True 时则会从*dispatch_line()* 来调用。

user_return(*frame*, *return_value*)

当*stop_here()* 返回 True 时则会从*dispatch_return()* 来调用。

user_exception(*frame*, *exc_info*)

当*stop_here()* 返回 True 时则会从*dispatch_exception()* 来调用。

do_clear(*arg*)

处理当一个中断点属于临时性中断点时是否必须要移除它。

此方法必须由派生类来实现。

派生类与客户端可以调用以下方法来影响步进状态。

set_step()

在一行代码之后停止。

set_next(*frame*)

在给定的帧以内或以下的下一行停止。

set_return(*frame*)

当从给定的帧返回时停止。

set_until(*frame*, *lineno=None*)

在 *lineno* 行大于当前所到达的行或者在从当前帧返回时停止。

set_trace([*frame*])

从 *frame* 开始调试。如果未指定 *frame*，则从调用者的帧开始调试。

在 3.13 版本发生变更: *set_trace()* 将立即进入调试器，而不是在下一行要执行的代码上进入。

set_continue()

仅在中断点上或是当结束时停止。如果不存在中断点，则将系统追踪函数设为 None。

set_quit()

将 quitting 属性设为 True。这将在对某个 dispatch_*() 方法的下一次调用中引发*BdbQuit*。

派生的类和客户端可以调用下列方法来操纵中断点。如果出现错误则这些方法将返回一个包含错误消息的字符串，或者如果一切正常则返回 None。

set_break(*filename*, *lineno*, *temporary=False*, *cond=None*, *funcname=None*)

设置一个新的中断点。如果对于作为参数传入的 *filename* 不存在 *lineno*，则返回一条错误消息。*filename* 应为规范的形式，如在*canonic()* 方法中描述的。

clear_break(*filename*, *lineno*)

删除位于 *filename* 和 *lineno* 的中断点。如果未设置过中断点，则返回一条错误消息。

clear_bpbynumber(*arg*)

删除*Breakpoint.bpbynumber* 中索引号为 *arg* 的中断点。如果 *arg* 不是数字或超出范围，则返回一条错误消息。

clear_all_file_breaks(*filename*)

删除位于 *filename* 的所有中断点。如果未设置过中断点，则返回一条错误消息。

clear_all_breaks()

删除所有现存的中断点。如果未设置过中断点，则返回一条错误消息。

get_bpbynumber(*arg*)

返回由指定数字所指明的中断点。如果 *arg* 是一个字符串，它将被转换为一个数字。如果 *arg* 不是一个表示数字的字符串，如果给定的中断点不存在或者已被删除，则会引发*ValueError*。

Added in version 3.2.

get_break(*filename*, *lineno*)

如果存在针对 *filename* 中 *lineno* 的中断点则返回 `True`。

get_breaks(*filename*, *lineno*)

返回 *filename* 中在 *lineno* 上的所有中断点，或者如果未设置任何中断点则返回一个空列表。

get_file_breaks(*filename*)

返回 *filename* 中的所有中断点，或者如果未设置任何中断点则返回一个空列表。

get_all_breaks()

返回已设置的所有中断点。

派生类与客户端可以调用以下方法来获取一个代表栈回溯信息的数组结构。

get_stack(*f*, *t*)

返回一个栈回溯中 (frame, lineno) 元组的列表，及一个大小值。

最近调用的帧将排在列表的末尾。大小值即调试器被唤起所在帧之下的帧数量。

format_stack_entry(*frame_lineno*, *lprefix=': '*)

返回一个字符串，其内容为有关以 `(frame, lineno)` 元组表示的特定栈条目的信息。返回的字符串包含:

- 包含该帧的规范文件名。
- 函数名称或 `"<lambda>"`。
- 输入参数。
- 返回值。
- 代码的行（如果存在）。

以下两个方法可由客户端调用以使用一个调试器来调试一条以字符串形式给出的*statement*。

run(*cmd*, *globals=None*, *locals=None*)

调试一条通过 *exec()* 函数执行的语句。*globals* 默认为 `__main__.__dict__`，*locals* 默认为 *globals*。

runeval(*expr*, *globals=None*, *locals=None*)

调试一条通过 *eval()* 函数执行的表达式。*globals* 和 *locals* 的含义与在 *run()* 中的相同。

runctx(*cmd*, *globals*, *locals*)

为了保证向下兼容性。调用 *run()* 方法。

runcall(*func*, */*, **args*, ***kwds*)

调试一个单独的函数调用，并返回其结果。

最后，这个模块定义了以下函数:

bdb.**checkfuncname**(*b*, *frame*)

如果应当在此中断则返回 `True`，具体取决于 *Breakpoint* *b* 的设置方式。

如果是通过行号设置的，它将检查 *b.line* 是否与 *frame* 中的行一致。如果中断点是通过函数名称设置的，则必须检查是否位于正确的 帧 (正确的函数) 以及是否位于其中第一个可执行的行。

bdb.**effective**(*file*, *line*, *frame*)

返回 `(active breakpoint, delete temporary flag)` 或 `(None, None)` 作为目标中断点。

*激活的中断点*是 *bplist* 中对应 (*file*, *line*) (它必须存在) 的第一个*已启用的* 条目，对应的 *checkfuncname()* 为真值，并且没有为假值的 *cond* 或为正值的 *ignore* 计数。*flag* 表示临时中断点应当被删除，它仅在 *cond* 无法被求值时 (在此情况下，*ignore* 计数将被忽略) 才会为 `False`。

如果不存在这样的条目，则返回 `(None, None)`。

bdb.**set_trace**()

使用一个来自调用方的帧的 *Bdb* 实例开始调试。

28.3 faulthandler --- 转储 Python 回溯信息

Added in version 3.3.

本模块包含当发生故障、超时或收到用户信号时可转储 Python 回溯信息的函数。调用 *faulthandler.enable()* 可安装针对 *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* 和 *SIGILL* 信号的故障处理器。你还可以在启动时通过设置 PYTHONFAULTHANDLER 环境变量或使用 -X faulthandler 命令行选项来启用它们。

故障处理器可兼容系统故障处理器如 Apport 或 Windows 故障处理器。本模块会在 sigaltstack() 函数可用时为信号处理器使用备用栈。这允许它即使在栈溢出的情况下也能转储回溯信息。

故障处理程序将在灾难性场合调用，因此只能使用信号安全的函数（比如不能在堆上分配内存）。由于这一限制，与正常的 Python 跟踪相比，转储量是最小的。

- 只支持 ASCII 码。编码时会用到 backslashreplace 错误处理程序。
- 每个字符串限制在 500 个字符以内。
- 只会显式文件名、函数名和行号。（不显示源代码）
- 上限是 100 页内存帧和 100 个线程。
- 反序排列：最近的调用最先显示。

默认情况下，Python 的跟踪信息会写入 *sys.stderr*。为了能看到跟踪信息，应用程序必须运行于终端中。日志文件也可以传给 *faulthandler.enable()*。

本模块是用 C 语言实现的，所以才能在崩溃或 Python 死锁时转储跟踪信息。

在 Python 启动时，*Python 开发模式* 会调用 *faulthandler.enable()*。

参见

模块 *pdb*
用于 Python 程序的交互式源代码调试器。

模块 *traceback*
提取、格式化和打印 Python 程序的栈回溯信息的标准接口。

28.3.1 转储跟踪信息

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)

将所有线程的跟踪数据转储到 *file* 中。如果 *all_threads* 为 False，则只转储当前线程。

参见

traceback.print_tb()，可被用于打印回溯对象。

在 3.5 版本发生变更: 增加了向本函数传入文件描述符的支持。

28.3.2 故障处理程序的状态

faulthandler.**enable**(*file=sys.stderr*, *all_threads=True*)

启用默认的处理器：为 *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* 和 *SIGILL* 信号安装处理器来转储 Python 回溯信息。如果 *all_threads* 为 True，则会为每个运行中的线程产生回溯信息。在其他情况下，将只转储当前线程。

file 必须保持打开状态，直至停用故障处理程序为止：参见 *文件描述符相关话题*。

在 3.5 版本发生变更: 增加了向本函数传入文件描述符的支持。

在 3.6 版本发生变更: 在 Windows 系统中，同时会安装一个 Windows 异常处理程序。

在 3.10 版本发生变更: 现在如果 *all_threads* 为 True，则转储信息会包含垃圾收集器是否正在运行。

faulthandler.**disable**()

停用故障处理程序：卸载由*enable()* 安装的信号处理程序。

faulthandler.**is_enabled**()

检查故障处理程序是否被启用。

28.3.3 一定时间后转储跟踪数据。

faulthandler.**dump_traceback_later**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)

在 *timeout* 秒超时后，转储所有线程的回溯信息，或者如果 *repeat* 为 True 则每隔 *timeout* 秒执行一次转储。如果 *exit* 为 True，则在转储回溯信息后调用 _exit() 并设置 status=1。（请注意 _exit() 会立即关闭进程，这意味着不会做任何清理工作，如刷新文件缓冲区等。）如果函数被调用两次，则新的调用将替代之前的形参并重置超时。计时器的精度为亚秒级。

file 必须保持打开状态，直至跟踪信息转储完毕，或调用了*cancel_dump_traceback_later()*：参见文件描述符相关话题。

本函数用一个看门狗线程实现。

在 3.5 版本发生变更: 增加了向本函数传入文件描述符的支持。

在 3.7 版本发生变更: 该函数现在总是可用。

faulthandler.**cancel_dump_traceback_later**()

取消*dump_traceback_later()* 的最后一次调用。

28.3.4 转储用户信号的跟踪信息。

faulthandler.**register**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

注册一个用户信号：为 *signum* 信号安装一个处理程序，将所有线程或当前线程（*all_threads* 为 False 时）的跟踪信息转储到 *file* 中。如果 chain 为 True，则调用上一层处理程序。

file 必须保持打开状态，直至该信号被*unregister()* 注销：参见文件描述符相关话题。

Windows 中不可用。

在 3.5 版本发生变更: 增加了向本函数传入文件描述符的支持。

faulthandler.**unregister**(*signum*)

注销一个用户信号：卸载由*register()* 安装的 *signum* 信号处理程序。如果信号已注册，返回 True，否则返回 False。

Windows 中不可用。

28.3.5 文件描述符相关话题

enable()、*dump_traceback_later()* 和*register()* 保留其 *file* 参数给出的文件描述符。如果文件关闭，文件描述符将被一个新文件重新使用；或者用*os.dup2()* 替换了文件描述符，则跟踪信息将被写入另一个文件。每次文件被替换时，都会再次调用这些函数。

28.3.6 示例

在 Linux 中启用和停用内存段故障的默认处理程序：

```
$ python -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python -q -X faulthandler
>>> import ctypes
```

(续下页)

(接上页)

```
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

28.4 pdb --- Python 的调试器

源代码： Lib/pdb.py

---

pdb 模块定义了一个交互式源代码调试器，用于 Python 程序。它支持在源码行间设置（有条件的）断点和单步执行，检视堆栈帧，列出源码列表，以及在任何堆栈帧的上下文中运行任意 Python 代码。它还支持事后调试，可以在程序控制下调用。

调试器是可扩展的——调试器实际被定义为*Pdb* 类。该类目前没有文档，但通过阅读源码很容易理解它。扩展接口使用了*bdb* 和*cmd* 模块。

参见

模块*faulthandler*
用于在发生错误、超时或用户信号时显式地转储 Python 回溯信息。

模块*traceback*
提取、格式化和打印 Python 程序的栈回溯信息的标准接口。

中断进入调试器的典型用法是插入:

```
import pdb; pdb.set_trace()
```

或者:

```
breakpoint()
```

到你想进入调试器的位置，再运行程序。然后你可以单步执行这条语句之后的代码，并使用*continue* 命令来关闭调试器继续运行。

在 3.7 版本发生变更: 内置函数*breakpoint()*，当以默认方式调用时，可被用来代替 `import pdb; pdb.set_trace()`。

```
def double(x):
   breakpoint()
   return x * 2
val = 3
print(f"{val} * 2 is {double(val)}")
```

调试器的提示符为 `(Pdb)`，这指明你正处于调试模式下:

```
> ...(2)double()
-> breakpoint()
(Pdb) p x
3
(Pdb) continue
3 * 2 is 6
```

在 3.3 版本发生变更: 由 *readline* 模块实现的 Tab 补全可用于补全本模块的命令和命令的参数，例如，Tab 补全会提供当前的全局变量和局部变量，用作 p 命令的参数。

你还可以从命令行唤起 *pdb* 来调试其他脚本。例如:

```
python -m pdb [-c command] (-m module | pyfile) [args ...]
```

当作为模块被唤起时，如果被调试的程序异常退出则 pdb 将自动进入事后调试。在事后调试之后（或程序正常退出之后），pdb 将重启程序。自动重启会保留 pdb 的状态（如断点）并且在大多数情况下这比在退出程序的同时退出调试器更实用。

-c, --command <command>

要以在 .pdbrc 文件中所给出形式来执行命令；请参阅调试器命令。

在 3.2 版本发生变更: 增加了 -c 选项。

-m <module>

要以类似于 python -m 的方式来执行模块。就像一个脚本那样，调试器将在模块的第一行之前暂停执行。

在 3.7 版本发生变更: 增加了 -m 选项。

在调试器控制下执行一条语句的典型用法如下:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
>>> pdb.run("f(2)")
> <string>(1)<module>()
(Pdb) continue
0.5
>>>
```

检查已崩溃程序的典型用法是:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
...
>>> f(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
ZeroDivisionError: division by zero
>>> pdb.pm()
> <stdin>(2)f()
(Pdb) p x
0
(Pdb)
```

在 3.13 版本发生变更: **PEP 667** 的实现意味着通过 pdb 进行的变量名赋值将立刻影响当前作用域，即使是在*optimized scope* 中运行的时候。

本模块定义了下列函数，每个函数进入调试器的方式略有不同:

pdb.**run**(*statement*, *globals=None*, *locals=None*)

在调试器控制范围内执行 *statement*（以字符串或代码对象的形式提供）。调试器提示符会在执行代码前出现，你可以设置断点并键入 *continue*，也可以使用 *step* 或 *next* 逐步执行语句（上述所有命令在后文有说明）。可选参数 *globals* 和 *locals* 指定代码执行环境，默认时使用 *__main__* 模块的字典。（请参阅内置函数 *exec()* 或 *eval()* 的说明。）

pdb.**runeval**(*expression*, *globals=None*, *locals=None*)

在调试器控制下对 *expression* (以字符串或代码对象的形式给出) 求值。当 *runeval()* 返回时，它将返回 *expression* 的值。在其他方面此函数与 *run()* 类似。

pdb.**runcall**(*function*, **args*, ***kwds*)

使用给定的参数调用 *function*（以函数或方法对象的形式提供，不能是字符串）。*runcall()* 返回的是所调用函数的返回值。调试器提示符将在进入函数后立即出现。

pdb.**set_trace**(*, *header=None*)

在调用本函数的堆栈帧处进入调试器。用于硬编码一个断点到程序中的固定点处，即使该代码不在调试状态（如断言失败时）。如果传入 *header*，它将在调试开始前被打印到控制台。

在 3.7 版本发生变更: 仅关键字参数 *header*。

在 3.13 版本发生变更: *set_trace()* 将立即进入调试器，而不是在下一行要执行的代码上进入。

pdb.**post_mortem**(*t=None*)

进入指定异常或 Enter post-mortem debugging of the given exception or 回溯对象的事后调试。如未指定值，它将使用当前正在处理的异常，或者在找不到时则会引发 ValueError。

在 3.13 版本发生变更: 增加了对异常对象的支持。

pdb.**pm**()

进入在*sys.last_exc* 中找到的异常的事后调试。

run* 函数和*set_trace()* 都是别名，用于实例化*Pdb* 类和调用同名方法。如果要使用其他功能，则必须自己执行以下操作:

class pdb.**Pdb**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*)

Pdb 是调试器类。

completekey、*stdin* 和 *stdout* 参数都会传递给底层的*cmd.Cmd* 类，请参考相应的描述。

如果给出 *skip* 参数，则它必须是一个迭代器，可以迭代出 glob-style 样式的模块名称。如果遇到匹配上述样式的模块，调试器将不会进入来自该模块的堆栈帧。[1]

默认情况下，当发出*continue* 命令时，Pdb 将为 SIGINT 信号（信号当用户在控制台按 Ctrl-C 时发出的）设置一个处理器。这使用户可以通过按 Ctrl-C 再次进入调试器。如果你希望 Pdb 不要改变 SIGINT 处理器，请将 *nosigint* 设为真值。to true.

readrc 参数默认为 true，它控制 Pdb 是否从文件系统加载.pdbrc 文件。

启用跟踪且带有 *skip* 参数的调用示范:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

引发一个不带参数的审计事件 pdb.Pdb。

在 3.1 版本发生变更: 增加了 *skip* 形参。

在 3.2 版本发生变更: 增加了 *nosigint* 形参。在之前版本中，pdb 绝不会设置 SIGINT 处理器。

在 3.6 版本发生变更: *readrc* 参数。

run(*statement*, *globals=None*, *locals=None*)

runeval(*expression*, *globals=None*, *locals=None*)

runcall(*function*, **args*, ***kwds*)

set_trace()

请参阅上文解释同名函数的文档。

28.4.1 调试器命令

下方列出的是调试器可接受的命令。如下所示，大多数命令可以缩写为一个或两个字母。如 h(elp) 表示可以输入 h 或 help 来输入帮助命令（但不能输入 he 或 hel，也不能是 H 或 Help 或 HELP）。命令的参数必须用空格（空格符或制表符）分隔。在命令语法中，可选参数括在方括号 ([]) 中，使用时请勿输入方括号。命令语法中的选择项由竖线 (|) 分隔。

[1] 一个帧是否会被认为源自特定模块是由帧全局变量 __name__ 来决定的。

输入一个空白行将重复最后输入的命令。例外：如果最后一个命令是*list* 命令，则会列出接下来的 11 行。

调试器无法识别的命令将被认为是 Python 语句，并在正在调试的程序的上下文中执行。Python 语句也可以用感叹号 (!) 作为前缀。这是检查正在调试的程序的强大方法，甚至可以修改变量或调用函数。当此类语句发生异常，将打印异常名称，但调试器的状态不会改变。

在 3.13 版本发生变更: 前缀为 pdb 命令的表达式/语句现在会被正确地标识并执行。

调试器支持别名。别名可以有参数，使得调试器对被检查的上下文有一定程度的适应性。

在一行中可以输入多条命令，以 `;;` 分隔。（不能使用单个 `;`，因为它已被用作传给 Python 解析器的一行中的多条命令的分隔符。）命令切分所用的方式没有任何智能可言；输入总是会在第一个 `;;` 对上被切分，即使它位于带引号的字符串中。对于带有双分号的字符串可以使用隐式字符串拼接 `';'';'` 或 `";"";"` 来变通处理。

要设置临时全局变量，请使用 *快捷变量*。*快捷变量*是名称以 `$` 打头的变量。例如，`$foo = 1` 将设置一个全局变量 `$foo` 供你在调试器会话中使用。*快捷变量*会在程序恢复执行时被清空因此它不大可能像使用普通变量如 `foo = 1` 那样影响到你的程序。

存在三个预设的 *快捷变量*:

- `$_frame`: 你正在调试的当前帧
- `$_retval`: 当帧返回时的返回值
- `$_exception`: 当帧引发异常时的异常值

Added in version 3.12: 增加了 *便捷变量*特性。

如果文件 `.pdbrc` 存在于用户主目录或当前目录中，则它将以 `'utf-8'` 编码格式被读入并执行，就像是在调试器提示符下被键入一样，不同之处在于空行和以 `#` 开头的行会被忽略。这对于别名特别有用。如果两个文件都存在，则会先读取主目录中的文件并且在那里定义的别名可以被本地文件所覆盖。

在 3.2 版本发生变更: `.pdbrc` 现在可以包含继续调试的命令，如*continue* 或*next*。文件中的这些命令以前是无效的。

在 3.11 版本发生变更: `.pdbrc` 现在将以 `'utf-8'` 编码格式来读取。在之前版本中，它是以系统语言区域编码格式来读取的。

h(elp) `[command]`

不带参数时，显示可用的命令列表。参数为 *command* 时，打印有关该命令的帮助。`help pdb` 显示完整文档（即*pdb* 模块的文档字符串）。由于 *command* 参数必须是标识符，因此要获取 `!` 的帮助必须输入 `help exec`。

w(here)

打印栈回溯，最新的帧位于底部。有一个箭头 (>) 指明当前帧，该帧决定了大多数命令的上下文。

d(own) `[count]`

在堆栈回溯中，将当前帧向下移动 *count* 级（默认为 1 级，移向更新的帧）。

u(p) `[count]`

在堆栈回溯中，将当前帧向上移动 *count* 级（默认为 1 级，移向更老的帧）。

b(reak) `[([filename:]lineno | function) [, condition]]`

传入 *lineno* 参数，在当前文件内的第 *lineno* 行设置中断。行号数值开头可以带有 *filename* 加一个冒号，以在另一个文件内指定中断点（可能是尚未载入的文件）。文件将根据*sys.path* 来搜索。可接受的 *filename* 形式有 `/abspath/to/file.py`, `relpath/file.py`, `module` 和 `package.module`。

传入 *function* 参数，在该函数内的第一条可执行语句上设置中断。*function* 可以是会在当前命名空间中被求值为一个函数的任意表达式。

如果第二个参数存在，它应该是一个表达式，且它的计算值为 true 时断点才起作用。

如果不带参数执行，将列出所有中断，包括每个断点、命中该断点的次数、当前的忽略次数以及关联的条件（如果有）。

每个中断点将被分配一个数值供所有其他中断点命令引用。

tbreak [([filename:]lineno | function) [, condition]]

临时断点，在第一次命中时会自动删除。它的参数与*break* 相同。

cl(ear) [filename:lineno | bpnumber ...]

如果参数是 *filename:lineno*，则清除此行上的所有断点。如果参数是空格分隔的断点编号列表，则清除这些断点。如果不带参数，则清除所有断点（但会先提示确认）。

disable bpnumber [bpnumber ...]

禁用断点，断点以空格分隔的断点编号列表给出。禁用断点表示它不会导致程序停止执行，但是与清除断点不同，禁用的断点将保留在断点列表中并且可以（重新）启用。

enable bpnumber [bpnumber ...]

启用指定的断点。

ignore bpnumber [count]

为指定的断点编号设置忽略次数。如果省略 *count*，则忽略次数将设置为 0。当忽略次数为零时断点将变为活动状态。如果为非零值，则在每次到达断点且断点未禁用且关联条件取真值时 *count* 就像递减。

condition bpnumber [condition]

为断点设置一个新 *condition*，它是一个表达式，且它的计算值为 true 时断点才起作用。如果没有给出 *condition*，则删除现有条件，也就是将断点设为无条件。

commands [bpnumber]

为编号是 *bpnumber* 的断点指定一系列命令。命令内容将显示在后续的几行中。输入仅包含 end 的行来结束命令列表。举个例子:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

要删除断点上的所有命令，请输入 commands 并立即以 end 结尾，也就是不指定任何命令。

如果不带 *bpnumber* 参数，commands 作用于最后一个被设置的断点。

可以为断点指定命令来重新启动程序。只需使用*continue* 或*step* 命令或其他可以继续运行程序的命令。

如果指定了某个继续运行程序的命令（目前包括*continue*, *step*, *next*, *return*, *jump*, *quit* 及它们的缩写）将终止命令列表（就像该命令后紧跟着 end）。因为在任何时候继续运行下去（即使是简单的 next 或 step），都可能会遇到另一个断点，该断点可能具有自己的命令列表，这导致要执行的列表含糊不清。

如果在命令列表中使用 silent 命令，那么在断点处停下时就不会打印常规信息。这正是要打印特定消息然后继续运行的断点所想要的。如果没有其他命令来打印任何消息，则你将不会看到已到达断点的迹象。

s(tep)

运行当前行，在第一个可以停止的位置（在被调用的函数内部或在当前函数的下一行）停下。

n(ext)

继续运行，直到运行到当前函数的下一行，或当前函数返回为止。（*next* 和*step* 之间的区别在于，*step* 进入被调用函数内部并停止，而*next* （几乎）全速运行被调用函数，仅在当前函数的下一行停止。）

unt(il) [lineno]

如果不带参数，则继续运行，直到行号比当前行大时停止。

如果带有 *lineno*，则继续执行直至行号大于或等于 *lineno*。在这两种情况下，在当前帧返回时也将停止。

在 3.2 版本发生变更: 允许明确给定行号。

r(eturn)
继续运行，直到当前函数返回。

c(ont(inue))
继续运行，仅在遇到断点时停止。

j(ump) lineno
设置即将运行的下一行。仅可用于堆栈最底部的帧。它可以往回跳来再次运行代码，也可以往前跳来跳过不想运行的代码。

需要注意的是，不是所有的跳转都是允许的 -- 例如，不能跳转到 `for` 循环的中间或跳出 `finally` 子句。

l(ist) [first[, last]]
列出当前文件的源代码。如果不带参数，则列出当前行周围的 11 行，或延续前一次列出。如果用 . 作为参数，则列出当前行周围的 11 行。如果带有一个参数，则列出那一行周围的 11 行。如果带有两个参数，则列出所给的范围中的代码；如果第二个参数小于第一个参数，则将其解释为列出行数的计数。

当前帧中的当前行用 -> 标记。如果正在调试异常，且最早抛出或传递该异常的行不是当前行，则那一行用 >> 标记。

在 3.2 版本发生变更: 增加了 >> 标记。

ll | longlist
列出当前函数或帧的所有源代码。相关行的标记与*list* 相同。

Added in version 3.2.

a(rgs)
打印当前函数的参数及其当前的值。

p expression
在当前上下文中对 *expression* 求值并打印该值。

> **备注**
> `print()` 也可以使用，但它不是一个调试器命令 --- 它执行 Python *print()* 函数。

pp expression
与*p* 命令类似，但 *expression* 的值将使用*pprint* 模块美观地打印。

whatis expression
打印 *expression* 的类型。

source expression
尝试获取 *expression* 的源代码并显示它。

Added in version 3.2.

display [expression]
如果 *expression* 的值发生变化则显示它的值，每次都会停止执行当前帧。

如果不带 *expression*，则列出当前帧的所有显示表达式。

> **备注**
> 显示 *expression* 的值并与 *expression* 之前的求值结果进行比较，因此当结果可变时，显示可能无法体现变化。

示例：

```
lst = []
breakpoint()
pass
lst.append(1)
print(lst)
```

显示将不会发现 `lst` 已被改变因为求值结果在执行比较之前已被 `lst.append(1)` 原地修改了:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst
display lst: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
(Pdb)
```

你可以通过拷贝机制巧妙地实现此功能:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst[:]
display lst[:]: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
display lst[:]: [1]  [old: []]
(Pdb)
```

Added in version 3.2.

undisplay [expression]

不再显示当前帧中的 *expression*。如果不带 *expression*，则清除当前帧的所有显示表达式。

Added in version 3.2.

interact

在根据当前作用域的局部和全局命名空间初始化的新建全局命名空间中启动一个交互式解释器 (使用 *code* 模块)。使用 `exit()` 或 `quit()` 退出解释器并返回调试器。

> **备注**
>
> 由于 `interact` 为代码执行创建了一个新的专用命名空间，对变量的赋值将不会影响原始命名空间。不过，对任何被引用的可变对象的修改都将照常在原始命名空间中反映出来。

Added in version 3.2.

在 3.13 版本发生变更: 可以使用 `exit()` 和 `quit()` 退出 *interact* 命令。

在 3.13 版本发生变更: *interact* 会将其输出引向调试器的输出通道而不是 *sys.stderr*。

alias [name [command]]

创建一个名为 *name* 的别名用来执行 *command*。*command* 的两边 不可用引号括起来。可替换形参可以通过 `%1`, `%2` ... 和 `%9` 等来指明，而 `%*` 将被所有这些形参替换。如果省略 *command*，则将显示 *name* 的当前别名。如未给出任何参数，则将列出所有别名。

别名允许嵌套并可包含能在 pdb 提示符下合法输入的任何内容。请注意内部 pdb 命令 可以被别名所覆盖。这样的命令将被隐藏直到别名被移除。别名会递归地应用到命令行的第一个单词；行内的其他单词不会受影响。

作为示例，这里列出了两个有用的别名（特别适合放在 .pdbrc 文件中）：

```
# 打印实例变量 (用法 "pi classInst")
alias pi for k in %1.__dict__.keys(): print(f"%1.{k} = {%1.__dict__[k]}")
# 打印 self 中的实例变量
alias ps pi self
```

unalias name

删除指定的别名 *name*。

! statement

在当前栈帧的上下文中执行 (单行的) *statement*。感叹号可以被省略，除非第一个语句的第一个单词与某个调试器命名重名，例如:

```
(Pdb) ! n=42
(Pdb)
```

要设置全局变量，你可以在同一行上在赋值命令前添加 global 语句，例如:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]

restart [args ...]

重启被调试的 Python 程序。如果提供了 *args*，它会用*shlex* 来拆分且拆分结果将被用作新的*sys.argv*。历史、中断点、动作和调试器选项将被保留。*restart* 是*run* 的一个别名。

q(uit)

退出调试器。被执行的程序将被中止。

debug code

进入一个对 *code* 执行步进的递归调试器（该参数是在当前环境中执行的任意表达式或语句）。

retval

打印当前函数最后一次返回的返回值。

exceptions [excnumber]

列出串连的异常或在其间跳转。

当将 pdb.pm() 或 Pdb.post_mortem(...) 用于串连的异常而不是回溯数据时，它允许用户使用 exceptions 命令在串连的异常间移动以列出异常，并使用 exceptions <number> 来切换到该异常。

示例：

```
def out():
    try:
        middle()
    except Exception as e:
        raise ValueError("reraise middle() error") from e

def middle():
    try:
        return inner(0)
    except Exception as e:
        raise ValueError("Middle fail")

def inner(x):
```

(续下页)

(接上页)

```
    1 / x

out()
```

调用 pdb.pm() 将允许在异常之间移动:

```
> example.py(5)out()
-> raise ValueError("reraise middle() error") from e

(Pdb) exceptions
  0 ZeroDivisionError('division by zero')
  1 ValueError('Middle fail')
> 2 ValueError('reraise middle() error')

(Pdb) exceptions 0
> example.py(16)inner()
-> 1 / x

(Pdb) up
> example.py(10)middle()
-> return inner(0)
```

Added in version 3.13.

备注

28.5 Python 性能分析器

源代码： Lib/profile.py 和 Lib/pstats.py

28.5.1 性能分析器简介

cProfile 和*profile* 提供了 Python 程序的 *确定性性能分析*。*profile* 是一组统计数据，描述程序的各个部分执行的频率和时间。这些统计数据可以通过*pstats* 模块格式化为报表。

Python 标准库提供了同一分析接口的两种不同实现：

1. 对于大多数用户，建议使用*cProfile*；这是一个 C 扩展插件，因为其合理的运行开销，所以适合于分析长时间运行的程序。该插件基于 lsprof，由 Brett Rosen 和 Ted Chaotter 贡献。
2. *profile* 是一个纯 Python 模块（*cProfile* 就是模拟其接口的 C 语言实现），但它会显著增加配置程序的开销。如果你正在尝试以某种方式扩展分析器，则使用此模块可能会更容易完成任务。该模块最初由 Jim Roskind 设计和编写。

备注

profiler 分析器模块被设计为给指定的程序提供执行概要文件，而不是用于基准测试目的（*timeit* 才是用于此目标的，它能获得合理准确的结果）。这特别适用于将 Python 代码与 C 代码进行基准测试：分析器为 Python 代码引入开销，但不会为 C 级别的函数引入开销，因此 C 代码似乎比任何 Python 代码都更快。

28.5.2 实时用户手册

本节是为“不想阅读手册”的用户提供的。它提供了非常简短的概述，并允许用户快速对现有应用程序执行评测。

要分析采用单个参数的函数，可以执行以下操作：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

（如果*cProfile*在您的系统上不可用，请使用*profile*。）

上述操作将运行*re.compile()*并打印分析结果，如下所示：

```
      214 function calls (207 primitive calls) in 0.002 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 __init__.py:250(compile)
     1    0.000    0.000    0.001    0.001 __init__.py:289(_compile)
     1    0.000    0.000    0.000    0.000 _compiler.py:759(compile)
     1    0.000    0.000    0.000    0.000 _parser.py:937(parse)
     1    0.000    0.000    0.000    0.000 _compiler.py:598(_code)
     1    0.000    0.000    0.000    0.000 _parser.py:435(_parse_sub)
```

第一行显示有 214 个调用被监控。在这些调用中，207 为 *primitive*，表示这些调用不是通过递归引起的。下一行: `Ordered by: cumulative time` 表示输出是按 `cumtime` 值排序的。列标题包括：

ncalls
: 调用次数

tottime
: 在指定函数中消耗的总时间（不包括调用子函数的时间）

percall
: 是 `tottime` 除以 `ncalls` 的商

cumtime
: 指定的函数及其所有子函数（从调用到退出）消耗的累积时间。这个数字对于递归函数来说是准确的。

percall
: 是 `cumtime` 除以原始调用（次数）的商（即：函数运行一次的平均时间）

filename:lineno(function)
: 提供相应数据的每个函数

如果第一列中有两个数字（例如 3/1），则表示函数递归。第二个值是原始调用次数，第一个是调用的总次数。请注意，当函数不递归时，这两个值是相同的，并且只打印单个数字。

profile 运行结束时，打印输出不是必须的。也可以通过为 `run()` 函数指定文件名，将结果保存到文件中：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

*pstats.Stats*类从文件中读取 profile 结果，并以各种方式对其进行格式化。

*cProfile*和*profile*文件也可以作为脚本调用，以分析另一个脚本。例如：

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

-o <output_file>
: 将性能分析结果写入到文件而不是标准输出。

-s <sort_order>
: 指定某个*sort_stats()*排序值以对输出进行排序。这仅适用于未提供*-o*的情况。

-m <module>

指定要分析的是一个模块而不是脚本。

Added in version 3.7: *cProfile* 添加 -m 选项

Added in version 3.8: *profile* 添加 -m 选项

pstats 模块的*Stats* 类具有各种方法用来操纵和打印保存到性能分析结果文件的数据。

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

strip_dirs() 方法移除了所有模块名称中的多余路径。*sort_stats()* 方法按照打印出来的标准模块/行/名称对所有条目进行排序。*print_stats()* 方法打印出所有的统计数据。你可以尝试下列排序调用:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

第一个调用实际上将按函数名称对列表进行排序，而第二个调用将打印出统计数据。下面是一些可以尝试的有趣调用:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

这将按一个函数中的累计时间对性能分析数据进行排序，然后只打印出最重要的十行。如果你了解哪些算法在耗费时间，上面这一行就是你应该使用的。

如果你想要看看哪些函数的循环次数多，且耗费时间长，你应当这样做:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

以按照每个函数耗费的时间进行排序，然后打印前十个函数的统计数据。

你也可以尝试:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

这将按照文件名对所有统计数据进行排序，然后只打印出类初始化方法的统计数据 (因为它们的名称中都有 __init__)。作为最后一个例子，你可以尝试:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

这一行以时间为主键，并以累计时间为次键进行排序，然后打印出部分统计数据。具体来说，该列表首先被缩减至原始大小的 50% (即: .5)，然后只保留包含 init 的行，并打印该子列表。

如果你想知道有哪些函数调用了上述函数，你现在就可以做 (p 仍然会按照最后一个标准进行排序):

```
p.print_callers(.5, 'init')
```

这样你将得到每个被列出的函数的调用方列表。

如果你想要更多的功能，你就必须阅读手册，或者自行猜测下列函数的作用:

```
p.print_callees()
p.add('restats')
```

作为脚本被唤起，*pstats* 模块是一个用于读取和性能分析转储文件的统计数据浏览器。它有一个简单的面向行的界面（使用*cmd* 实现）和交互式的帮助。

28.5.3 profile 和 cProfile 模块参考

profile 和*cProfile* 模块都提供下列函数:

profile.**run**(*command*, *filename=None*, *sort=-1*)

此函数接受一个可被传递给*exec()* 函数的单独参数，以及一个可选的文件名。在所有情况下这个例程都会执行:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

并收集执行过程中的性能分析统计数据。如果未提供文件名，则此函数会自动创建一个*Stats* 实例并打印一个简单的性能分析报告。如果指定了 sort 值，则它会被传递给这个*Stats* 实例以控制结果的排序方式。

profile.**runctx**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

此函数类似于*run()*，带有为 *command* 字符串提供 globals 和 locals 映射对象的附加参数。这个例程会执行:

```
exec(command, globals, locals)
```

并像在上述的*run()* 函数中一样收集性能分析数据。

class profile.**Profile**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

这个类通常只在需要比 `cProfile.run()` 函数所能提供的更精确的性能分析控制时被使用。

可以通过 *timer* 参数提供一个自定义计时器来测量代码运行花费了多长时间。它必须是一个返回代表当前时间的单个数字的函数。如果该数字为整数，则 *timeunit* 指定一个表示每个时间单位持续时间的乘数。例如，如果定时器返回以千秒为计量单位的时间值，则时间单位将为 `.001`。

直接使用*Profile* 类将允许格式化性能分析结果而无需将性能分析数据写入到文件:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# 做些事情……
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

Profile 类也可作为上下文管理器使用 (仅在*cProfile* 模块中支持。参见上下文管理器类型):

```
import cProfile

with cProfile.Profile() as pr:
    # 做些事情……

    pr.print_stats()
```

在 3.8 版本发生变更: 添加了上下文管理器支持。

enable()

开始收集分析数据。仅在*cProfile* 可用。

disable()

停止收集分析数据。仅在*cProfile* 可用。

create_stats()

停止收集分析数据，并在内部将结果记录为当前 profile。

print_stats(*sort=-1*)

根据当前性能分析数据创建一个*Stats* 对象并将结果打印到 stdout。

sort 形参指定所显示统计信息的排序。它接受单个键或由键组成的元组以启用多级排序，就像在*Stats.sort_stats* 中那样。

Added in version 3.13: 现在*print_stats()* 可接受一个由键组成的元组。

dump_stats(*filename*)

将当前 profile 的结果写入 *filename* 。

run(*cmd*)

通过*exec()* 对该命令进行性能分析。

runctx(*cmd*, *globals*, *locals*)

通过*exec()* 并附带指定的全局和局部环境对该命令进行性能分析。

runcall(*func*, */*, **args*, ***kwargs*)

对 `func(*args, **kwargs)` 进行性能分析

请注意性能分析只有在被调用的命令/函数确实能返回时才可用。如果解释器被终结（例如在被调用的命令/函数执行期间通过*sys.exit()* 调用）则将不会打印性能分析结果。

28.5.4 Stats 类

性能数据的分析是使用*Stats* 类来完成的。

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)

这个类构造器会基于 *filename* (或文件名列表) 或者 `Profile` 实例创建一个“统计对象”。输出将被打印到由 *stream* 所指定的流。

上述构造器所选择的文件必须由相应版本的*profile* 或*cProfile* 来创建。具体来说，不会保证文件与此性能分析器的未来版本兼容，也不会保证与其他性能分析器，或运行于不同操作系统的同一性能分析器所产生的文件兼容。如果提供了几个文件，则相同函数的所有统计数据将被聚合在一起，这样就可以在单个报告中同时考虑几个进程的总体情况。如果额外的文件需要与现有*Stats* 对象中的数据相结合，则可以使用*add()* 方法。

作为从一个文件读取性能分析数据的替代，可以使用 `cProfile.Profile` 或*profile.Profile* 对象作为性能分析数据源。

Stats 对象有以下方法:

strip_dirs()

这个用于*Stats* 类的方法会从文件名中去除所有前导路径信息。它对于减少打印输出的大小以适应（接近）80 列限制。这个方法会修改对象，被去除的信息将会丢失。在执行去除操作后，可以认为对象拥有的条目将使用“随机”顺序，就像它刚在对象初始化并加载之后一样。如果*strip_dirs()* 导致两个函数名变得无法区分（它们位于相同文件名的相同行，并且具有相同的函数名），那么这两个条目的统计数据将被累积到单个条目中。

add(**filenames*)

Stats 类的这个方法会将额外的性能分析信息累积到当前的性能分析对象中。它的参数应当指向由相应版本的*profile.run()* 或 `cProfile.run()` 所创建的文件名。相同名称（包括 file, line, name）函数的统计信息会自动累积到单个函数的统计信息。

dump_stats(*filename*)

将加载至*Stats* 对象内的数据保存到名为 *filename* 的文件。该文件如果不存在则将被创建，如果已存在则将被覆盖。这等价于*profile.Profile* 和 `cProfile.Profile` 类上的同名方法。

sort_stats(**keys*)

此方法通过根据所提供的准则修改*Stats* 对象的排序。其参数可以是一个字符串或标识排序准则的 SortKey 枚举 (例如: `'time'`, `'name'`, `SortKey.TIME` 或 `SortKey.NAME`)。SortKey 枚举参数优于字符串参数因为它更为健壮且更不容易出错。

当提供一个以上的键时，额外的键将在之前选择的所有键的值相等时被用作次级准则。例如，`sort_stats(SortKey.NAME, SortKey.FILE)` 将根据其函数名对所有条目排序，并通过按文件名排序来处理所有平局（即函数名相同）。

对于字符串参数，可以对任何键名使用缩写形式，只要缩写是无歧义的。

以下是有效的字符串和 SortKey:

有效字符串参数	有效枚举参数	含意
`'calls'`	SortKey.CALLS	调用次数
`'cumulative'`	SortKey.CUMULATIVE	累积时间
`'cumtime'`	N/A	累积时间
`'file'`	N/A	文件名
`'filename'`	SortKey.FILENAME	文件名
`'module'`	N/A	文件名
`'ncalls'`	N/A	调用次数
`'pcalls'`	SortKey.PCALLS	原始调用计数
`'line'`	SortKey.LINE	行号
`'name'`	SortKey.NAME	函数名称
`'nfl'`	SortKey.NFL	名称/文件/行
`'stdname'`	SortKey.STDNAME	标准名称
`'time'`	SortKey.TIME	内部时间
`'tottime'`	N/A	内部时间

请注意对统计信息的所有排序都是降序的（将最耗时的条目放在最前面），其中名称、文件和行号搜索则是升序的（字母顺序）。`SortKey.NFL` 和 `SortKey.STDNAME` 之间的细微区别在于标准名称是按打印形式来排序名称的，这意味着嵌入的行号将以一种怪异的方式进行比较。例如，第 3, 20 和 40 行将会按字符串顺序 20, 3 和 40 显示（如果文件名相同的话）。相反地，`SortKey.NFL` 则会对行号进行数值比较。实际上，`sort_stats(SortKey.NFL)` 就等同于 `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`。

出于向下兼容的理由，数值参数 `-1`, `0`, `1` 和 `2` 也是被允许的。它们将被分别解读为 `'stdname'`, `'calls'`, `'time'` 和 `'cumulative'`。如果使用这种老旧格式（数值），则将只使用一个排序键（数字键），额外的参数将被静默地忽略。

Added in version 3.7: 增加了 SortKey 枚举。

reverse_order()

这个用于 *Stats* 类的方法将会反转对象内基本列表的顺序。请注意在默认情况下升序和降序排列将基于所选定的排序键来进行适当的选择。

print_stats(**restrictions*)

这个用于 *Stats* 类的方法将打印出在 *profile.run()* 定义中描述的报告。

打印的顺序是基于在对象上执行的最后一次 *sort_stats()* 操作（需要注意 *add()* 和 *strip_dirs()* 规则）。

所提供的参数（如果存在）可被用来将列表限制为重要的条目。在初始状态下，列表将为加入性能分析的函数的完整集合。每条限制可以是一个整数（用来选择行数），或是一个 0.0 至 1.0 范围内左开右闭的十进制小数（用来选择行数百分比），或是一个将被解读为正则表达式的字符串（用来匹配要打印的标准名称的模式）。如果提供了多条限制，则它们将逐个被应用。例如:

```
print_stats(.1, 'foo:')
```

将首先限制为打印列表的前 10%，然后再限制为仅打印在名为 `.*foo:` 的文件内的函数。作为对比，以下命令:

```
print_stats('foo:', .1)
```

将列表限制为名为 `.*foo:` 的文件内的所有函数，然后再限制为仅打印它们当中的前 10%。

print_callers(**restrictions*)

这个用于*Stats*类的方法将打印调用了加入性能分析数据库的每个函数的所有函数的列表。打印顺序与*print_stats()*所提供的相同，受限参数的定义也是相同的。每个调用方将在单独的行中报告。具体格式根据产生统计数据的性能分析器的不同而有所差异。

- 使用*profile*时，将在每个调用方之后的圆括号内显示一个数字来指明相应的调用执行了多少次。为了方便起见，右侧还有第二个不带圆括号的数字来重复显示该函数累计耗费的时间。
- 使用*cProfile*时，每个调用方前面将有三个数字：这个调用的执行次数，以及当前函数在被这个调用方唤起期间共计和累计耗费的时间。

print_callees(**restrictions*)

这个用于*Stats*类的方法将打印被指定的函数所调用的所有函数的列表。除了调用方向是逆序的（对应：被调用和被调用方），其参数和顺序与*print_callers()*方法相同。

get_stats_profile()

此方法返回一个 StatsProfile 的实例，它包含从函数名称到 FunctionProfile 实例的映射。每个 FunctionProfile 实例保存了相应函数性能分析的有关信息如函数运行耗费了多长时间，它被调用了多少次等等……

Added in version 3.9: 添加了以下数据类：StatsProfile, FunctionProfile。添加了以下函数：get_stats_profile。

28.5.5 什么是确定性性能分析？

*确定性性能分析*旨在反映这样一个事实：即所有*函数调用*、*函数返回*和*异常*事件都被监控，并且对这些事件之间的间隔（在此期间用户的代码正在执行）进行精确计时。相反，统计分析（不是由该模块完成）随机采样有效指令指针，并推断时间花费在哪里。后一种技术传统上涉及较少的开销（因为代码不需要检测），但只提供了时间花在哪里的相对指示。

在 Python 中，由于在执行过程中总有一个活动的解释器，因此执行确定性评测不需要插入指令的代码。Python 自动为每个事件提供一个*钩子*（可选回调）。此外，Python 的解释特性往往会给执行增加太多开销，以至于在典型的应用程序中，确定性分析往往只会增加很小的处理开销。结果是，确定性分析并没有那么代价高昂，但是它提供了有关 Python 程序执行的大量运行时统计信息。

调用计数统计信息可用于识别代码中的错误（意外计数），并识别可能的内联扩展点（高频调用）。内部时间统计可用于识别应仔细优化的" 热循环" 。累积时间统计可用于识别算法选择上的高级别错误。请注意，该分析器中对累积时间的异常处理，允许直接比较算法的递归实现与迭代实现的统计信息。

28.5.6 局限性

一个限制是关于时间信息的准确性。确定性性能分析存在一个涉及精度的基本问题。最明显的限制是，底层的" 时钟" 周期大约为 0.001 秒（通常）。因此，没有什么测量会比底层时钟更精确。如果进行了足够的测量，那么" 误差" 将趋于平均。不幸的是，删除第一个错误会导致第二个错误来源。

第二个问题是，从调度事件到分析器调用获取时间函数实际*获取*时钟状态，这需要" 一段时间" 。类似地，从获取时钟值（然后保存）开始，直到再次执行用户代码为止，退出分析器事件句柄时也存在一定的延迟。因此，多次调用单个函数或调用多个函数通常会累积此错误。尽管这种方式的误差通常小于时钟的精度（小于一个时钟周期），但它*可以*累积并变得非常可观。

与开销较低的*cProfile*相比，*profile*的问题更为严重。出于这个原因，*profile*提供了一种针对指定平台的自我校准方法，以便可以在很大程度上（平均地）消除此误差。校准后，结果将更准确（在最小二乘意义上），但它有时会产生负数（当调用计数异常低，且概率之神对您不利时：-）。因此*不要*对产生的负数感到惊慌。它们应该只在你手工校准分析器的情况下才会出现，实际上结果比没有校准的情况要好。

28.5.7 准确估量

*profile*模块的 profiler 会从每个事件处理时间中减去一个常量，以补偿调用 time 函数和存储结果的开销。默认情况下，常数为 0。对于特定的平台，可用以下程序获得更好修正常数（*局限性*）。

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

此方法将执行由参数所给定次数的 Python 调用，在性能分析器之下直接和再次地执行，并对两次执行计时。它将随后计算每个性能分析器事件的隐藏开销，并将其以浮点数的形式返回。例如，在一台运行 macOS 的 1.8Ghz Intel Core i5 上，使用 Python 的 time.process_time() 作为计时器，魔数大约为 4.04e-6。

此操作的目标是获得一个相当稳定的结果。如果你的计算机 非常快速，或者你的计时器函数的分辨率很差，你可能必须传入 100000，甚至 1000000，才能得到稳定的结果。

当你有一个一致的答案时，有三种方法可以使用：

```
import profile

# 1. 将计算出的偏差应用于此后创建的所有 Profile 实例。
profile.Profile.bias = your_computed_bias

# 2 将计算出的偏差应用于特定的 Profile 实例。
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. 在实例构造函数中指定计算出的偏差。
pr = profile.Profile(bias=your_computed_bias)
```

如果你可以选择，那么选择更小的常量会更好，这样你的结果将“更不容易”在性能分析统计中显示负值。

28.5.8 使用自定义计时器

如果你想要改变当前时间的确定方式（例如，强制使用时钟时间或进程持续时间），请向 `Profile` 类构造器传入你想要的计时函数：

```
pr = profile.Profile(your_time_func)
```

结果性能分析器将随后调用 `your_time_func`。根据你使用的是*profile.Profile* 还是 `cProfile.Profile`，`your_time_func` 的返回值将有不同的解读方式：

profile.Profile

`your_time_func` 应当返回一个数字，或一个总和为当前时间的数字列表（如同*os.times()* 所返回的内容）。如果该函数返回一个数字，或所返回的数字列表长度为 2，则你将得到一个特别快速的调度例程版本。

请注意你应当为你选择的计时器函数校准性能分析器类 (参见*准确估量*)。对于大多数机器来说，一个返回长整数值的计时器在性能分析期间将提供在低开销方面的最佳结果。(*os.times()* 是 相当糟糕的，因为它返回一个浮点数值的元组)。如果你想以最干净的方式替换一个更好的计时器，请派生一个类并硬连线一个能最佳地处理计时器调用的替换调度方法，并使用适当的校准常量。

cProfile.Profile

`your_time_func` 应当返回一个数字。如果它返回整数，你还可以通过第二个参数指定一个单位时间的实际持续长度来唤起类构造器。举例来说，如果 `your_integer_time_func` 返回以千秒为单位的时间，则你应当以如下方式构造 `Profile` 实例：

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

由于 `cProfile.Profile` 类无法被校准，因此自定义计时器函数应当要小心地使用并应当尽可能地快速。为了使自定义计时器获得最佳结果，可能需要在内部 `_lsprof` 模块的 C 源代码中对其进行硬编码。

Python 3.3 在*time* 中添加了几个可被用来精确测量进程或时钟时间的新函数。例如，参见*time.perf_counter()*。

28.6 timeit --- 测量小代码片段的执行时间

源码： Lib/timeit.py

此模块提供了一种简单的方法来计算一小段 Python 代码的耗时。它有命令行接口 以及一个可调用 方法。它避免了许多测量时间的常见陷阱。另见 Tim Peter 在 O'Reilly 出版的 *Python Cookbook* 第二版中 "算法" 章节的概述。

28.6.1 基本示例

以下示例显示了如何使用命令行接口 来比较三个不同的表达式：

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
$ python -m timeit "'-'.join([str(n) for n in range(100)])"
10000 loops, best of 5: 27.5 usec per loop
$ python -m timeit "'-'.join(map(str, range(100)))"
10000 loops, best of 5: 23.2 usec per loop
```

这可以通过*Python 接口* 实现

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

从*Python 接口* 还可以传出一个可调用对象：

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

但请注意*timeit()* 仅在使用命令行界面时会自动确定重复次数。在例子 一节你可以找到更多的进阶示例。

28.6.2 Python 接口

该模块定义了三个便利函数和一个公共类：

timeit.**timeit**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)

使用给定语句、*setup* 代码和 *timer* 函数创建一个*Timer* 实例，并执行 *number* 次其*timeit()* 方法。可选的 *globals* 参数指定用于执行代码的命名空间。

在 3.5 版本发生变更: 添加可选参数 *globals* 。

timeit.**repeat**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)

使用给定语句、*setup* 代码和 *timer* 函数创建一个*Timer* 实例，并使用给定的 *repeat* 计数和 *number* 执行运行其*repeat()* 方法。可选的 *globals* 参数指定用于执行代码的命名空间。

在 3.5 版本发生变更: 添加可选参数 *globals* 。

在 3.7 版本发生变更: *repeat* 的默认值由 3 更改为 5 。

timeit.**default_timer**()

默认计时器始终为 time.perf_counter()，它返回浮点形式的秒数。另一个选择是 time.perf_counter_ns，它返回整数形式的纳秒数。

在 3.3 版本发生变更: *time.perf_counter()* 现在是默认计时器。

class timeit.**Timer**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

用于小代码片段的计数执行速度的类。

构造函数接受一个将计时的语句、一个用于设置的附加语句和一个定时器函数。两个语句都默认为 `'pass'`；计时器函数与平台有关（请参阅模块文档字符串）。*stmt* 和 *setup* 也可能包含多个以 ; 或换行符分隔的语句，只要它们不包含多行字符串文字即可。该语句默认在 timeit 的命名空间内执行；可以通过将命名空间传递给 *globals* 来控制此行为。

要测量第一个语句的执行时间，请使用`timeit()` 方法。`repeat()` 和`autorange()` 方法是方便的方法来调用`timeit()` 多次。

setup 的执行时间从总体计时执行中排除。

stmt 和 *setup* 参数也可以使用不带参数的可调用对象。这将在一个计时器函数中嵌入对它们的调用，然后由`timeit()` 执行。请注意，由于额外的函数调用，在这种情况下，计时开销会略大一些。

在 3.5 版本发生变更: 添加可选参数 *globals* 。

timeit(*number=1000000*)

主语句执行次数 *number*。这会执行一次设置语句，然后返回执行主语句若干次所需的时间。默认计时器以浮点形式返回秒数。参数是循环的次数，默认为一百万次。主语句、设置语句和要使用的定时器函数都将被传递给构造器。

> **备注**
>
> 默认情况下，`timeit()` 暂时关闭*garbage collection* 。这种方法的优点在于它使独立时序更具可比性。缺点是 GC 可能是所测量功能性能的重要组成部分。如果是这样，可以在 *setup* 字符串中的第一个语句重新启用 GC。例如:
>
> ```
> timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
> ```

autorange(*callback=None*)

自动决定调用多少次`timeit()` 。

这是一个便利函数，它反复调用`timeit()` 以使总耗时 >= 0.2 秒，返回最终结果（循环次数、循环的总耗时）。它调用`timeit()` 的次数以序列 1, 2, 5, 10, 20, 50, ... 递增，直到所用时间至少为 0.2 秒。

如果给出 *callback* 并且不是 `None`，则在每次试验后将使用两个参数调用它: `callback(number, time_taken)` 。

Added in version 3.6.

repeat(*repeat=5*, *number=1000000*)

调用`timeit()` 几次。

这是一个方便的函数，它反复调用`timeit()` ，返回结果列表。第一个参数指定调用`timeit()` 的次数。第二个参数指定`timeit()` 的 *number* 参数。

> **备注**
>
> 从结果向量计算并报告平均值和标准差这些是很诱人的。但是，这不是很有用。在典型情况下，最低值给出了机器运行给定代码段的速度的下限；结果向量中较高的值通常不是由 Python 的速度变化引起的，而是由于其他过程干扰你的计时准确性。所以结果的`min()` 可能是你应该感兴趣的唯一数字。之后，你应该看看整个向量并应用常识而不是统计。

在 3.7 版本发生变更: *repeat* 的默认值由 3 更改为 5 。

print_exc(*file=None*)

帮助程序从计时代码中打印回溯。

典型使用:

```
t = Timer(...)       # 在 try/except 之外
try:
    t.timeit(...)    # 或 t.repeat(...)
except Exception:
    t.print_exc()
```

与标准回溯相比，优势在于将显示已编译模板中的源行。可选的 *file* 参数指向发送回溯的位置；它默认为*sys.stderr* 。

28.6.3 命令行接口

从命令行调用程序时，使用以下表单:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-p] [-v] [-h] [statement ...]
```

如果了解以下选项:

-n N, **--number**=N
: 执行' 语句' 多少次

-r N, **--repeat**=N
: 重复计时器的次数（默认为 5）

-s S, **--setup**=S
: 最初要执行一次的语句（默认为 `pass` ）

-p, **--process**
: 测量进程时间，而不是 wallclock 时间，使用*time.process_time()* 而不是*time.perf_counter()* ，这是默认值

  Added in version 3.3.

-u, **--unit**=U
: 为定时器输出指定一个时间单位；可以选择 `nsec`, `usec`, `msec` 或 `sec`

  Added in version 3.5.

-v, **--verbose**
: 打印原始计时结果；重复更多位数精度

-h, **--help**
: 打印一条简短的使用信息并退出

可以通过将每一行指定为单独的语句参数来给出多行语句；通过在引号中包含参数并使用前导空格可以缩进行。多个*-s* 选项的处理方式相似。

如果未给出*-n*，则会通过尝试按序列 1, 2, 5, 10, 20, 50, ... 递增的数值来计算合适的循环次数，直到总计时间至少为 0.2 秒。

default_timer() 测量可能受到在同一台机器上运行的其他程序的影响，因此在需要精确计时时最好的做法是重复几次计时并使用最佳时间。*-r* 选项对此有利；在大多数情况下，默认的 5 次重复可能就足够了。你可以使用*time.process_time()* 来测量 CPU 时间。

备注

执行 pass 语句会产生一定的基线开销。这里的代码不会试图隐藏它，但你应该知道它。可以通过不带参数调用程序来测量基线开销，并且 Python 版本之间可能会有所不同。

28.6.4 例子

可以提供一个在开头只执行一次的 setup 语句：

```
$ python -m timeit -s "text = 'sample string'; char = 'g'" "char in text"
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s "text = 'sample string'; char = 'g'" "text.find(char)"
1000000 loops, best of 5: 0.342 usec per loop
```

在输出信息中，共有三个字段。首先是 loop count，它告诉你每个计时循环重复运行了多少次语句体。然后是 repetition count ('best of 5')，它告诉你计时循环重复了多少次，最后是语句体在计时循环重复中最好的平均耗时。即最快一次重复的耗时除以循环计数。

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

使用 *Timer* 类及其方法可以完成同样的操作：

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668, 0.
↪37866875250654886]
```

以下示例显示如何计算包含多行的表达式。在这里我们对比使用 *hasattr()* 与 `try`/`except` 的开销来测试缺失与提供对象属性：

```
$ python -m timeit "try:" "  str.__bool__" "except AttributeError:" "  pass"
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit "if hasattr(str, '__bool__'): pass"
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit "try:" "  int.__bool__" "except AttributeError:" "  pass"
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit "if hasattr(int, '__bool__'): pass"
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # 属性缺失
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
```

(续下页)

(接上页)

```
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

要让*timeit* 模块访问你定义的函数，你可以传递一个包含 import 语句的 *setup* 参数:

```
def test():
    """愚笨的测试函数"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

另一种选择是将*globals()* 传递给 *globals* 参数，这将导致代码在当前的全局命名空间中执行。这比单独指定 import 更方便

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

28.7 trace --- 跟踪或记录 Python 语句的执行

源代码: Lib/trace.py

模块*trace* 模块用于跟踪程序的执行过程，可生成带注释的语句覆盖率列表，打印调用/被调用关系，列出程序运行期间执行过的函数。该模块可在其他程序或命令行中使用。

参见

Coverage.py
流行的第三方代码覆盖工具，可输出 HTML ，并提供分支覆盖等高级功能。

28.7.1 命令行用法

trace 模块可由命令行调用。用法如此简单:

```
python -m trace --count -C . somefile.py ...
```

上述命令将执行 `somefile.py` ，并在当前目录生成执行期间所有已导入 Python 模块的带注解列表。

--help
显示用法并退出。

--version
显示模块版本并退出。

Added in version 3.8: 加入了 `--module` 选项，允许运行可执行模块。

主要的可选参数

在调用*trace*时，至少须指定以下可选参数之一。*-listfuncs*与*-trace*、*-count*相互排斥。如果给出*--listfuncs*，就再不会接受*--count*和*--trace*，反之亦然。

-c, --count
在程序完成时生成一组带有注解的报表文件，显示每个语句被执行的次数。参见下面的*-coverdir*、*-file*和*-no-report*。

-t, --trace
执行时显示每一行。

-l, --listfuncs
显示程序运行时执行到的函数。

-r, --report
由之前用了*--count*和*--file*运行的程序产生一个带有注解的报表。不会执行代码。

-T, --trackcalls
显示程序运行时暴露出来的调用关系。

修饰器

-f, --file=<file>
用于累计多次跟踪运行计数的文件名。应与*--count*一起使用。

-C, --coverdir=<dir>
报表文件的所在目录。package.module 的覆盖率报表将被写入文件 *dir/package/module*.cover。

-m, --missing
生成带注解的报表时，用 >>>>>> 标记未执行的行。

-s, --summary
在用到*--count*或*--report*时，将每个文件的简短摘要输出到 stdout。

-R, --no-report
不生成带注解的报表。如果打算用*--count*执行多次运行，然后在最后产生一组带注解的报表，该选项就很有用。

-g, --timing
在每一行前面加上时间，自程序运行算起。只在跟踪时有用。

过滤器

以下参数可重复多次。

--ignore-module=<mod>
忽略给出的模块名及其子模块（若为包）。参数可为逗号分隔的名称列表。

--ignore-dir=<dir>
忽略指定目录及其子目录下的所有模块和包。参数可为*os.pathsep*分隔的目录列表。

28.7.2 编程接口

class trace.**Trace**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)

创建一个对象来跟踪单个语句或表达式的执行。所有参数均为选填。*count* 可对行号计数。*trace* 启用单行执行跟踪。*countfuncs* 可列出运行过程中调用的函数。*countcallers* 可跟踪调用关系。*ignoremods* 是要忽略的模块或包的列表。*ignoredirs* 是要忽略的模块或包的目录列表。*infile* 是个文件名，从该文件中读取存储的计数信息。*outfile* 是用来写入最新计数信息的文件名。*timing* 可以显示相对于跟踪开始时间的时间戳。

run(*cmd*)

执行命令，并根据当前跟踪参数从执行过程中收集统计数据。*cmd* 必须为字符串或 code 对象，可供传入*exec()*。

runctx(*cmd*, *globals=None*, *locals=None*)

在定义的全局和局部环境中，执行命令并收集当前跟踪参数下的执行统计数据。若没有定义 *globals* 和 *locals*，则默认为空字典。

runfunc(*func*, */*, **args*, ***kwds*)

在*Trace*对象的控制下，用给定的参数调用 *func*，并采用当前的跟踪参数。

results()

返回一个*CoverageResults*对象，包含之前对指定*Trace*实例调用 `run`、`runctx` 和 `runfunc` 的累积结果。累积的跟踪结果不会重置。

class trace.**CoverageResults**

存放代码覆盖结果的容器，由*Trace.results()*创建。用户不应直接去创建。

update(*other*)

从另一个*CoverageResults*对象中合并代码覆盖数据。

write_results(*show_missing=True*, *summary=False*, *coverdir=None*, ***, *ignore_missing_files=False*)

写入代码覆盖结果。设置 *show_missing* 可显示未命中的行。设置 *summary 可在输出中包含每个模块的覆盖率摘要信息。*coverdir* 可指定覆盖率结果文件的输出目录，为 `None` 则结果将置于源文件所在目录中。

如果 *ignore_missing_files* 为 `True`，则对于已不存在文件的覆盖计数将被静默地忽略。在其他情况下，文件不存在将引发*FileNotFoundError*。

在 3.13 版本发生变更: 增加了 *ignore_missing_files* 形参。

以下例子简单演示了编程接口的用法:

```
import sys
import trace

# 创建一个 Trace 对象，告诉它要忽略什么，
# 及是否执行跟踪或行计数或者两者均执行。
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# 使用给定的 tracer 运行新命令
tracer.run('main()')

# 生成报告，将输出放入当前目录
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

28.8 tracemalloc --- 跟踪内存分配

Added in version 3.4.

源代码： Lib/tracemalloc.py

---

tracemalloc 模块是一个用于对 Python 已申请的内存块进行调试的工具。它能提供以下信息:

- 回溯对象分配内存的位置
- 按文件、按行统计 Python 的内存块分配情况: 内存块总大小、数量以及块平均大小。

- 对比两个内存快照的差异，以便排查内存泄漏

要追踪 Python 所分配的大部分内存块，模块应当通过将 `PYTHONTRACEMALLOC` 环境变量设置为 `1`，或是通过使用 `-X tracemalloc` 命令行选项来尽可能早地启动。可以在运行时调用*tracemalloc.start()* 函数来启动追踪 Python 内存分配。

在默认情况下，一个已分配内存块的追踪将只储存最新的帧 (1 帧)。要要启动时储存 25 帧：将 `PYTHONTRACEMALLOC` 环境变量设为 `25`，或使用 `-X tracemalloc=25` 命令行选项。

28.8.1 例子

显示前 10 项

显示内存分配最多的 10 个文件：

```
import tracemalloc

tracemalloc.start()

# ... 运行你的应用程序 ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Python 测试套件的输出示例：

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

我们可以看到 Python 从模块载入了 `4855 KiB` 数据（字节码和常量）并且*collections* 模块分配了 `244 KiB` 来构建*namedtuple* 类型。

更多选项，请参见*Snapshot.statistics()*

计算差异

获取两个快照并显示差异：

```
import tracemalloc
tracemalloc.start()
# ... 启动你的应用程序 ...

snapshot1 = tracemalloc.take_snapshot()
# ... 调用函数泄漏内存 ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
```

(续下页)

(接上页)

```
for stat in top_stats[:10]:
    print(stat)
```

运行 Python 测试套件的部分测试之前/之后的输出样例:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),␣
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),␣
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), average=139␣
↪B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), average=86␣
↪B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), average=96.0␣
↪KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), average=784␣
↪B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),␣
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969),␣
↪average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
↪average=546 B
```

我们可以看到 Python 已载入了 `8173 KiB` 模块数据（字节码和常量），并且这比测试之前，即保存前一个快照时载入的数据多出了 `4428 KiB`。类似地，*linecache* 模块已缓存 `940 KiB` 的 Python 源代码至格式回溯中，即从前一个快照开始的所有数据。

如果系统空闲内存太少，可以使用*Snapshot.dump()* 方法将快照写入磁盘来离线分析快照。然后使用*Snapshot.load()* 方法重载快照。

获取一个内存块的溯源

找出程序中最大内存块的溯源的代码:

```
import tracemalloc

# 存储 25 帧
tracemalloc.start(25)

# ... 运行你的应用程序 ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# 挑出最大的内存块
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

一段 Python 单元测试的输出案例 (限制最大 25 层堆栈)

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
```

(续下页)

(接上页)

```
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

我们可以看到大部分内存都被分配到*importlib*模块中以便从模块中加载数据（字节码和常量）：`870.1 KiB`。回溯位置是*importlib*最近加载数据的地方：在*doctest*模块的 `import pdb` 行。如果加载了新模块则回溯可能发生改变。line of the. The traceback may change if a new module is loaded.

美化的 top

使用美化输出显示分配最多内存的 10 行的代码，忽略 `<frozen importlib._bootstrap>` 和 `<unknown>` 文件:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
```

(续下页)

(接上页)

```
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... 运行你的应用程序 ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Python 测试套件的输出示例：

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

更多选项，请参见*Snapshot.statistics()*

记录所有被追踪内存块的当前和峰值大小

以下代码通过创建一个包含数字的列表来低效率地计算总计值如 `0 + 1 + 2 + ...`。该列表会临时消耗大量内存。我们可以使用*get_traced_memory()* 和*reset_peak()* 来观察计算总计值之后的内存使用减少以及计算过程中的内存使用峰值:

```
import tracemalloc

tracemalloc.start()

# 示例代码: 对一个大临时列表计算总计值
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# 示例代码: 对一个小临时列表计算总计值
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()
```

(续下页)

(接上页)

```
print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

输出:

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

使用*reset_peak()* 将确保我们能够准确地记录 `small_sum` 计算期间的峰值,即使它远小于从*start()* 调用以来内存块的总体峰值大小。如果没有对*reset_peak()* 的调用，`second_peak` 将仍为计算 `large_sum` 时的峰值 (也就是说，等于 `first_peak`)。在这种情况下，两个峰值都将比最终的内存使用量高得多，这表明我们可以进行优化 (通过移除不必要的对*list* 的调用，并改写为 `sum(range(...))`)。

28.8.2 API

函数

`tracemalloc.`**`clear_traces`**`()`

清空 Python 所分配的内存块的追踪数据。

另见*stop()*.

`tracemalloc.`**`get_object_traceback`**`(`*obj*`)`

获取 Python 对象 *obj* 被分配位置的回溯。返回一个*Traceback* 实例，或者如果*tracemalloc* 模块未追踪任何内存分配或未追踪该对象的分配则返回 `None`。

另请参阅*gc.get_referrers()* 和*sys.getsizeof()* 函数。

`tracemalloc.`**`get_traceback_limit`**`()`

获取保存在一个追踪的回溯中的最大帧数。

tracemalloc 模块必须正在追踪内存分配才能获得该限制值，否则将引发异常。

该限制是由*start()* 函数设置的。

`tracemalloc.`**`get_traced_memory`**`()`

获取一个元组形式的由*tracemalloc* 模块所追踪的内存块的当前大小和峰值大小: `(current: int, peak: int)`。

`tracemalloc.`**`reset_peak`**`()`

将由*tracemalloc* 模块所追踪的内存块的峰值大小设置为当前大小。

如果*tracemalloc* 模块未在追踪内存分配则不做任何事。

此函数只修改已记录的峰值大小，而不会修改或清空任何追踪，这不同于*clear_traces()*。在调用*reset_peak()* 之前使用*take_snapshot()* 保存的快照可以与调用之后保存的快照进行有意义的比较。

另请参阅*get_traced_memory()*。

Added in version 3.9.

`tracemalloc.`**`get_tracemalloc_memory`**`()`

获取*tracemalloc* 模块用于保存内存块追踪所使用的内存字节数。返回一个*int*。

`tracemalloc.`**`is_tracing`**`()`

如果*tracemalloc* 模块正在追踪 Python 内存分配则返回 `True`，否则返回 `False`。

另请参阅*start()* 和*stop()* 函数。

`tracemalloc.`**`start`**`(`*nframe: int = 1*`)`

开始追踪 Python 内存分配：在 Python 内存分配器上安装钩子。收集的追踪回溯将被限制为 *nframe* 个帧。在默认情况下，一个内存块的追踪将只保存最近的帧：即限制为 `1`。*nframe* 必须大于等于 `1`。

你仍然可以通过访问*Traceback.total_nframe* 属性来读取组成回溯的原始总帧数。

保存 1 帧以上仅适用于计算由 'traceback' 分组的统计数据或计算累积的统计数据：请参阅*Snapshot.compare_to()* 和*Snapshot.statistics()* 方法。

保存更多帧会增加*tracemalloc* 模块的内存和 CPU 开销。请使用*get_tracemalloc_memory()* 函数来检测*tracemalloc* 模块消耗了多少内存。

PYTHONTRACEMALLOC 环境变量 (PYTHONTRACEMALLOC=NFRAME) 和 -X tracemalloc=NFRAME 命令行选项可被用来在启动时开始追踪。

另请参阅*stop()*, *is_tracing()* 和*get_traceback_limit()* 等函数。

tracemalloc.**stop**()

停止追踪 Python 内存分配：卸载 Python 内存分配器上的钩子。并清空之前收集的所有由 Python 分配的内存块的追踪。

调用*take_snapshot()* 函数在清空追踪之前保存它们的快照。

另请参阅*start()*, *is_tracing()* 和*clear_traces()* 等函数。

tracemalloc.**take_snapshot**()

保存一个由 Python 分配的内存块的追踪的快照。返回一个新的*Snapshot* 实例。

该快照不包括在*tracemalloc* 模块开始追踪内存分配之前分配的内存块。

追踪的回溯被限制为*get_traceback_limit()* 个帧。可使用*start()* 函数的 *nframe* 形参来保存更多的帧。

tracemalloc 模块必须正在追踪内存分配才能保存快照，参见*start()* 函数。

另请参阅*get_object_traceback()* 函数。

域过滤器

class tracemalloc.**DomainFilter**(*inclusive:* bool, *domain:* int)

按地址空间（域）来过滤内存块的追踪。

Added in version 3.6.

inclusive

如果 *inclusive* 为 True (包括)，则匹配分配于地址空间*domain* 中的内存块。

如果 *inclusive* 为 False (排除)，则匹配不是分配于地址空间*domain* 中的内存块。

domain

内存块的地址空间 (int)。只读的特征属性。Read-only property.

过滤器

class tracemalloc.**Filter**(*inclusive:* bool, *filename_pattern:* str, *lineno:* int = *None*, *all_frames:* bool = *False*, *domain:* int = *None*)

对内存块的跟踪进行筛选。

请参阅*fnmatch.fnmatch()* 函数来了解 *filename_pattern* 的语法。'.pyc' 文件扩展名以 '.py' 替换。

示例：

- Filter(True, subprocess.__file__) 只包括*subprocess* 模块的追踪数据
- Filter(False, tracemalloc.__file__) 排除了*tracemalloc* 模块的追踪数据
- Filter(False, "<unknown>") 排除了空的回溯信息

在 3.5 版本发生变更: '.pyo' 文件扩展名不会再被替换为 '.py'。

在 3.6 版本发生变更: 增加了*domain* 属性。

domain

内存块的地址空间 (`int` 或 `None`)。

tracemalloc 使用 `0` 号域来追踪 Python 的内存分配操作。C 扩展可以使用其他域来追踪其他资源。extensions can use other domains to trace other resources.

inclusive

如果 *inclusive* 为 `True` (包括), 则只匹配名称与*filename_pattern* 匹配的文件在行号为*lineno* 的位置上分配的内存块。

如果 *inclusive* 为 `False` (排除), 则忽略名称与*filename_pattern* 匹配的文件在行号为*lineno* 的位置上分配的内存块。

lineno

过滤器的行号 (`int`)。如果 *lineno* 为 `None`，则该过滤器将匹配任意行号。

filename_pattern

过滤器的文件名模型 (`str`)。只读的特征属性。

all_frames

如果 *all_frames* 为 `True`，则回溯的所有帧都会被检查。如果 *all_frames* 为 `False`，则只有最近的帧会被检查。

如果回溯限制为 `1` 则该属性将没有效果。参见*get_traceback_limit()* 函数和*Snapshot.traceback_limit* 属性。

帧

class tracemalloc.**Frame**

回溯的帧。

Traceback 类是一个*Frame* 实例的序列。instances.

filename

文件名（字符串）

lineno

行号（整形）

快照

class tracemalloc.**Snapshot**

由 Python 分配的内存块的追踪的快照。

take_snapshot() 函数创建一个快照实例。

compare_to(*old_snapshot:* Snapshot, *key_type:* str, *cumulative:* bool = *False*)

计算与某个旧快照的差异。获取按 *key_type* 分组的*StatisticDiff* 实例的已排序列表形式的统计信息。

请参阅*Snapshot.statistics()* 方法了解 *key_type* 和 *cumulative* 形参。

结果将按以下值从大到小排序: *StatisticDiff.size_diff* 的绝对值, *StatisticDiff.size*, *StatisticDiff.count_diff* 的绝对值, *Statistic.count* 然后是*StatisticDiff.traceback*。

dump(*filename*)

将快照写入文件

使用*load()* 重载快照。

filter_traces(*filters*)

使用已过滤的*traces*序列新建一个*Snapshot*实例，*filters*是*DomainFilter*和*Filter*实例的列表。如果*filters*为空列表，则返回一个包含追踪的副本的新的*Snapshot*实例。

包括的所有过滤器将同时被应用，一个追踪如果没有任何包括的过滤器与其匹配则会被忽略。一个追踪如果有至少一个排除的过滤器与其匹配将会被忽略。

在 3.6 版本发生变更: *DomainFilter*实例现在同样被*filters*所接受。

classmethod load(*filename*)

从文件载入快照。

另见*dump()*.

statistics(*key_type:* str, *cumulative:* bool = *False*)

获取*Statistic*信息列表，按*key_type*分组排序：

key_type	description
`'filename'`	文件名
`'lineno'`	文件名和行号
`'traceback'`	回溯

如果*cumulative*为`True`，则累积一个追踪的回溯的所有帧的内存块的大小和数量，而不只是最近的帧。累积模式只能在*key_type*等于`'filename'`和`'lineno'`的情况下使用。

结果将按以下值从大到小排序: *Statistic.size*, *Statistic.count*然后是*Statistic.traceback*。

traceback_limit

保存在*traces*的回溯中的帧的最大数量：当快照被保存时*get_traceback_limit()*的结果。

traces

由 Python 分配的所有内存块的追踪: *Trace*实例的序列。

该序列的顺序是未定义的。请使用*Snapshot.statistics()*方法来获取统计信息的已排序列表。

统计

class tracemalloc.**Statistic**

统计内存分配

*Snapshot.statistics()*返回*Statistic*实例的列表。.

参见*StatisticDiff*类。

count

内存块数（整形）。

size

以字节数表示的内存块总计大小(`int`)。

traceback

内存块分配位置的回溯，*Traceback*实例。

StatisticDiff

class tracemalloc.**StatisticDiff**

在旧的和新的*Snapshot*实例之间内存分配上的统计差异。

*Snapshot.compare_to()*返回一个*StatisticDiff*实例的列表。另请参看*Statistic*类。

count

新快照中的内存块数量 (int): 如果在新快照中内存块已被释放则为 0。

count_diff

在旧的新的快照之间内存块数量之差 (int): 如果在新快照中内存块已被分配则为 0。

size

新快速中以字节数表示的内存块总计大小 (int): 如果在新快照中内存块已被释放则为 0。

size_diff

在旧的新的快照之间以字节数表示的内存块总计大小之差 (int): 如果在新快照中内存块已被分配则为 0。

traceback

内存块分配位置的回溯，*Traceback* 实例。

跟踪

class tracemalloc.**Trace**

一个内存块的跟踪信息。

Snapshot.traces 属性是一个*Trace* 实例的序列。

在 3.6 版本发生变更: 增加了*domain* 属性。

domain

内存块的地址空间 (int)。只读的特征属性。Read-only property.

tracemalloc 使用 0 号域来追踪 Python 的内存分配操作。C 扩展可以使用其他域来追踪其他资源。extensions can use other domains to trace other resources.

size

以字节数表示的内存块大小 (int)。

traceback

内存块分配位置的回溯，*Traceback* 实例。

回溯

class tracemalloc.**Traceback**

Frame 实例的序列将按从最旧的帧到最新的帧排序。

一个回溯包含至少 1 个帧。如果 tracemalloc 模块无法获取帧，则会使用文件名 "<unknown>" 和行号 0。

当保存一个快照时，追踪的回溯被限制为*get_traceback_limit()* 个帧。参见*take_snapshot()* 函数。回溯的原始帧数存放在*Traceback.total_nframe* 属性中。这可以让人了解一个回溯是否因回溯限制而被截断。

Trace.traceback 属性是一个*Traceback* 对象的实例。

在 3.7 版本发生变更: 现在帧的排序是从最旧到最新，而不是从最新到最旧。

total_nframe

在截断之前组成回溯的总帧数。如果此信息不可用则该属性可被设为 None。

在 3.9 版本发生变更: 增加了*Traceback.total_nframe* 属性。

format(*limit=None*, *most_recent_first=False*)

将回溯格式化为由行组成的列表。使用*linecache* 模块从源代码提取行。如果设置了 *limit*，则当 *limit* 为正值时将格式化 *limit* 个最新的帧。在其他情况下，则格式化 abs(limit) 个最新的帧。如果 *most_recent_first* 为 True，则将反转已格式化帧的顺序，首先返回最新的帧而不是最旧的。

类似于*traceback.format_tb()* 函数，不同之处是*format()* 不包括换行符。

示例:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

输出:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPTER 29

软件打包和分发

这些库可帮助你发布和安装 Python 软件。虽然这些模块设计为与 Python 包索引 结合使用，但它们也可以与本地索引服务器一起使用，或者根本不使用任何索引服务器。

29.1 ensurepip --- 初始设置 pip 安装器

Added in version 3.4.

源代码: Lib/ensurepip

---

ensurepip 包为在已有的 Python 安装实例或虚拟环境中引导 pip 安装器提供了支持。需要使用引导才能使用 pip 的这一事实也正好反映了 pip 是一个独立的项目，有其自己的发布周期，其最新版本随 CPython 解释器的维护版本和新特性版本一同捆绑。

在大多数情况下，Python 的终端使用者不需要直接调用这个模块（pip 默认应该已被引导），不过，如果在安装 Python（或创建虚拟环境）之时跳过了安装 pip 步骤，或者日后特意卸载了 pip ，则需要使用这个模块。

备注

这个模块 无需访问互联网。引导启动 pip 所需的全部组件均包含在包的内部。

参见

installing-index
安装 Python 包的终端使用者教程

PEP 453: 在 Python 安装实例中显式引导启动 pip
这个模块的原始缘由以及规范文档

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或*WebAssembly 平台* 上不受支持。

29.1.1 命令行界面

使用解释器的 `-m` 参数调用命令行接口。

最简单的调用方式为:

```
python -m ensurepip
```

该调用会在当前未安装 `pip` 的情况下安装 `pip`，如已安装则无事发生。如要确保安装的 `pip` 版本至少为 `ensurepip` 所支援的最新版本，传入 `--upgrade` 参数:

```
python -m ensurepip --upgrade
```

在默认情况下，`pip` 会被安装到当前虚拟环境（如果激活了虚拟环境）或系统的包目录（如果未激活虚拟环境）。安装位置可通过两个额外的命令行选项来控制:

--root <dir>

相对于给定的根目录而不是当前已激活虚拟环境（如果存在）的根目录或当前 Python 安装版的默认根目录来安装 `pip`。

--user

将 `pip` 安装到用户的站点包目录而不是针对当前 Python 安装版的全局安装（此选项在激活的虚拟环境内部是不被允许的）。

在默认情况下，脚本 `pipX` 和 `pipX.Y` 将被安装（其中 X.Y 表示被用来唤起 `ensurepip` 的 Python 的版本）。所安装的脚本可通过两个额外的命令行选项来控制:

--altinstall

如果请求了一个替代安装版，则 `pipX` 脚本将 不会被安装。

--default-pip

如果请求了一个" 默认的 pip" 安装版，则将在两个常规脚本的基础上额外安装 `pip` 脚本。

同时提供这两个脚本选择选项将会触发异常。

29.1.2 模块 API

ensurepip 暴露了两个函数用于编程:

ensurepip.**version**()

返回一个指明在初始创建环境时将被安装的可用 pip 版本的字符串。

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)

初始创建 `pip` 到当前的或指定的环境中。

root 指明要作为相对安装路径的替代根目录。如果 *root* 为 `None`，则安装会使用当前环境的默认安装位置。

upgrade 指明是否要将一个现有的较早版本的 `pip` 的安装版升级到可用的新版本。

user 指明是否使用针对用户的安装方案而不是全局安装。

在默认情况下，将会安装 `pipX` 和 `pipX.Y` 脚本（其中 X.Y 表示 Python 的当前版本）。

如果设置了 *altinstall*，则 `pipX` 将 不会被安装。

如果设置了 *default_pip*，则除了两个常规脚本外还将安装 `pip`。

同时设置 *altinstall* 和 *default_pip* 将触发 *ValueError*。

verbosity 控制初始创建操作对 *sys.stdout* 的输出信息级别。

引发一个审计事件 `ensurepip.bootstrap` 并附带参数 `root`。

> **备注**
>
> 创建创建过程对于 `sys.path` 和 `os.environ` 都会有附带影响。改为在子进程中唤起命令行接口可以避免这些附带影响。

> **备注**
>
> 初始创建过程可能会安装 `pip` 所需的额外模块，但其他软件不应假定这些依赖将总是会默认存在（因为这些依赖可能会在未来的 `pip` 版本中被移除）。

29.2 venv --- 虚拟环境的创建

Added in version 3.3.

源码： Lib/venv/

`venv` 模块支持创建轻量的“虚拟环境”，每个虚拟环境将拥有它们自己独立的安装在其`site` 目录中的 Python 软件包集合。虚拟环境是在现有的 Python 安装版基础之上创建的，这被称为虚拟环境的“基础” Python，并且还可选择与基础环境中的软件包隔离开来，这样只有在虚拟环境中显式安装的软件包才是可用的。

当在虚拟环境中使用时，常见安装工具如 pip 将把 Python 软件包安装到虚拟环境而无需显式地指明这一点。

虚拟环境是（主要的特性）：

- 用来包含支持一个项目（库或应用程序）所需的特定 Python 解释器、软件库和二进制文件。它们在默认情况下与其他虚拟环境中的软件以及操作系统中安装的 Python 解释器和库保持隔离。
- 包含在一个目录中，根据惯例被命名为项目目录下的 `.venv` 或 `venv`，或是有许多虚拟环境的容器目录下，如 `~/.virtualenvs`。
- 不可签入 Git 等源代码控制系统。
- 被认为是可丢弃的 -- 它应当能被简单地删除并从头开始重建。你不应在此环境中放置任何项目代码。
- 不被视为是可移动或可复制的——你只能在目标位置重建相同的环境。

请参阅 **PEP 405** 了解有关 Python 虚拟环境的更多背景信息。

> **参见**
>
> Python Packaging User Guide: Creating and using virtual environments

Availability: not Android, not iOS, not WASI.

此模块在*移动平台* 或*WebAssembly 平台* 上不受支持。

29.2.1 创建虚拟环境

虚拟环境 是通过执行 `venv` 模块来创建的：

```
python -m venv /path/to/new/virtual/environment
```

此命令会创建目标目录（必要时还包括父目录）并在其中放置一个 `pyvenv.cfg` 文件，文件带有一个指向运行此命令的 Python 安装版的 `home` 键。它还会创建一个 `bin` 子目录 (在 Windows 上为 `Scripts`)，其中包含 Python 可执行文件的副本或符号链接 (具体由创建环境时所使用的平台或参数而定)。它还会创建一个 `lib/pythonX.Y/site-packages` 子目录 (在 Windows 上则为 `Libsite-packages`)。如果指定了现有的目录，则该目录将被重用。

在 3.5 版本发生变更: 现在推荐使用 `venv` 来创建虚拟环境。

Deprecated since version 3.6, removed in version 3.8: **pyvenv** 是针对 Python 3.3 和 3.4 创建虚拟环境的推荐工具，并在 3.5 中被直接执行 `venv` 的方式所取代。

在 Windows，像下面这样唤起 `venv` 命令：

```
PS> python -m venv C:\path\to\new\virtual\environment
```

本命令如果以 `-h` 参数运行，将显示可用的选项:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps]
            [--without-scm-ignore-files]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

options:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when
                        symlinks are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear               Delete the contents of the environment directory
                        if it already exists, before environment creation.
  --upgrade             Upgrade the environment directory to use this
                        version of Python, assuming Python has been
                        upgraded in-place.
  --without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)
  --prompt PROMPT       Provides an alternative prompt prefix for this
                        environment.
  --upgrade-deps        Upgrade core dependencies (pip) to the latest
                        version in PyPI
  --without-scm-ignore-files
                        Skips adding SCM ignore files to the environment
                        directory (Git is supported by default).

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

在 3.4 版本发生变更: 默认安装 pip，添加了 `--without-pip` 和 `--copies` 选项。

在 3.4 版本发生变更: 在早期版本中，如果目标目录已存在，将引发错误，除非使用了 `--clear` 或 `--upgrade` 选项。

在 3.9 版本发生变更: 添加 `--upgrade-deps` 选项以将 pip + setuptools 升级为 PyPI 上的最新版本。

在 3.12 版本发生变更: `setuptools` 不再是核心的 venv 依赖项。

在 3.13 版本发生变更: 增加了 `--without-scm-ignore-files` 选项。

在 3.13 版本发生变更: 现在 `venv` 默认会创建为 Git 创建一个 `.gitignore` 文件。

备注

虽然 Windows 支持符号链接，但不推荐使用它们。特别注意，在文件资源管理器中双击 `python.exe` 将立即解析符号链接，并忽略虚拟环境。

备注

在 Microsoft Windows 上，为了启用 `Activate.ps1` 脚本，可能需要修改用户的执行策略。可以运行以下 PowerShell 命令来执行此操作：

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

参阅 About Execution Policies 以获取更多信息。

创建的 `pyvenv.cfg` 文件还包括 `include-system-site-packages` 键，如果运行 `venv` 时带有 `--system-site-packages` 选项则将会其设为 `true`，否则为 `false`。

除非采用 `--without-pip` 选项，否则将会调用*ensurepip* 将 `pip` 引导到虚拟环境中。

可以向 `venv` 传入多个路径，此时将根据给定的选项，在所给的每个路径上创建相同的虚拟环境。

29.2.2 虚拟环境是如何实现的

当运行虚拟环境中的 Python 解释器时，*sys.prefix* 和*sys.exec_prefix* 将指向该虚拟环境的相应目录，而*sys.base_prefix* 和*sys.base_exec_prefix* 将指向用于创建该虚拟环境的基础 Python 的相应目录。只需检测 `sys.prefix != sys.base_prefix` 就足以确定当前解释器是否运行于虚拟环境中。

一个虚拟环境可以通过位于其二进制文件目录目录 (在 POSIX 上为 `bin`；在 Windows 上为 `Scripts`) 中的脚本来“激活”。这会将该目录添加到你的 `PATH`，这样运行 **python** 时就会唤起虚拟环境的 Python 解释器，从而可以运行该目录中安装的脚本而不必使用其完整路径。激活脚本的唤起方式是平台专属的 (*<venv>* 必须用包含虚拟环境目录的路径来替换):

平台	Shell	用于激活虚拟环境的命令
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	pwsh	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Added in version 3.4: **fish** 和 **csh** 激活脚本。

Added in version 3.8: 在 POSIX 上安装 PowerShell 激活脚本，以支持 PowerShell Core。

激活一个虚拟环境的操作 *不是必需的*，因为你完全可以在唤起 Python 时指明特定虚拟环境的 Python 解释器的完整路径。更进一步地说，安装在虚拟环境中的所有脚本也都可以在不激活该虚拟环境的情况下运行。

为了达成此目的，安装到虚拟环境中的脚本将包含一个以“井号叹号”打头的行用来指定虚拟环境的 Python 解释器，`#!/<path-to-venv>/bin/python`。这意味着无论 `PATH` 的值是什么该脚本都将使用指定的解释器运行。在 Windows 上，对处理“井号叹号”行的支持将在你安装了 launcher 的情况下获得。这样，在 Windows 资源管理器窗口中双击一个已安装的脚本应当会使用正确的解释器运行它而无需激活相应虚拟环境或设置 `PATH`。

当一个虚拟环境已被激活时，`VIRTUAL_ENV` 环境变量会被设为该虚拟环境的路径。由于使用虚拟环境并不需要显式地激活它，因此 `VIRTUAL_ENV` 并不能被用来可靠地确定是否正在使用虚拟环境。

> **警告**
>
> 因为安装在虚拟环境中的脚本不应要求必须激活该虚拟环境，所以它们的“井号叹号”行会包含虚拟环境的绝对路径。因为这一点，所以虚拟环境在通常情况下都是不可移植的。你应当保证提供重建一个虚拟环境的简便方式（举例来说，如果你准备了需求文件 `requirements.txt`，则可以使用虚拟环境的 `pip` 执行 `pip install -r requirements.txt` 来安装虚拟环境所需的所有软件包）。如果出于某种原因你需要将虚拟环境移动到一个新的位置，则你应当在目标位置上重建它并删除旧位置上的虚拟环境。如果出于某种原因你移动了一个虚拟环境的上级目录，你也应当在新位置上重建该虚拟环境。否则，安装到该虚拟环境的软件包可能无法正常工作。

你可以通过在 shell 中输入 `deactivate` 来取消激活一个虚拟环境。取消激活的实现机制依赖于具体平台并且属于内部实现细节（通常，将会使用一个脚本或者 shell 函数）。

29.2.3 API

上述的高级方法使用了一个简单的 API，该 API 提供了一种机制，第三方虚拟环境创建者可以根据其需求自定义环境创建过程，该 API 为*EnvBuilder* 类。

class venv.**EnvBuilder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, ***, *scm_ignore_files=frozenset()*)

EnvBuilder 类在实例化时接受以下关键字参数：

- *system_site_packages* -- 一个指明系统 Python 的 site-packages 是否要对该环境可用的布尔值 (默认为 `False`)。
- *clear* -- 一个布尔值，当为真时，将在创建环境前删除任何现有目标目录的内容。
- *symlinks* -- 一个指明是否要尝试创建 Python 二进制文件的符号链接而非复制的布尔值。
- *upgrade* -- 一个布尔值，当为真时，将使用当前运行的 Python 升级现有环境——用于 Python 已原地更新的情况 (默认为 `False`)。
- *with_pip* -- 一个布尔值，当为真时，将确保在虚拟环境中已安装 pip。这将执行*ensurepip* 并使用 `--default-pip` 选项。
- *prompt* -- 一个要在虚拟环境被激活后显示的字符串（默认值 `None` 表示将使用环境的目录名）。如果提供特殊的字符串 `"."`，则将使用当前目录的基本名作为提示符。
- *upgrade_deps* -- 将 venv 的基础模块更新为 PyPI 上的最新版本
- *scm_ignore_files* -- 基于在可迭代对象中指定的源代码控制管理器（SCM）创建忽略文件。这项支持是通过名为 `create_{scm}_ignore_file` 的方法来定义的。默认支持的值只有通过*create_git_ignore_file()* 定义的 `"git"`。

在 3.4 版本发生变更: 添加 `with_pip` 参数

在 3.6 版本发生变更: 添加 `prompt` 参数

在 3.9 版本发生变更: 添加 `upgrade_deps` 参数

在 3.13 版本发生变更: 增加了 `scm_ignore_files` 形参

EnvBuilder 可以被用作基类。

create(*env_dir*)

指定要建立虚拟环境的目标目录（绝对路径或相对于当前路径）来创建虚拟环境。`create` 方法将在指定目录中创建环境，或者引发对应的异常。

EnvBuilder 类的 `create` 方法定义了可用于定制子类的钩子:

```
def create(self, env_dir):
    """
    在一个目录中创建虚拟的 Python 环境。
    env_dir 是用于创建环境的目标目录。
```

(续下页)

(接上页)

```
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

每个方法*ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()* 和*post_setup()* 都可以被重写。

ensure_directories(*env_dir*)

创建虚拟环境目录及尚不存在的所有必要的子目录，并返回一个上下文对象。这个上下文对象被用于存放供其他方法使用的属性（如路径等）。如果*EnvBuilder* 是附带参数 `clear=True` 创建的，则虚拟环境的内容将被清除并将重新创建所有必要的子目录。

返回的上下文对象是一个具有以下属性的*types.SimpleNamespace*:

- `env_dir` - 虚拟环境的位置。将被用作激活脚本中的 `__VENV_DIR__` (参见*install_scripts()*)。
- `env_name` - 虚拟环境的名称。将被用作激活脚本中的 `__VENV_NAME__` (参见*install_scripts()*)。
- `prompt` - 激活脚本要使用的提示符。将被用作激活脚本中的 `__VENV_PROMPT__` (参见*install_scripts()*)。
- `executable` - 虚拟环境所使用的下层 Python 可执行文件。这会将基于另一个虚拟环境创建虚拟环境的情况也纳入考虑。
- `inc_path` - 虚拟环境的 include 路径。
- `lib_path` - 虚拟环境的 purelib 路径。
- `bin_path` - 虚拟环境的 script 路径。
- `bin_name` - 相对于虚拟环境位置的 script 路径名称。用于激活脚本中的 `__VENV_BIN_NAME__` (参见*install_scripts()*)。
- `env_exe` - 虚拟环境中 Python 解释器的名称。用于激活脚本中的 `__VENV_PYTHON__` (参见*install_scripts()*)。
- `env_exec_cmd` - Python 解释器的名称，会将文件系统重定向也纳入考虑。这可被用于在虚拟环境中运行 Python。

在 3.11 版本发生变更: *venv sysconfig installation scheme* 被用于构造所创建目录的路径。

在 3.12 版本发生变更: 将属性 `lib_path` 添加到上下文中，并将上下文对象写入文档。

create_configuration(*context*)

在环境中创建 `pyvenv.cfg` 配置文件。

setup_python(*context*)

在环境中创建 Python 可执行文件的拷贝或符号链接。在 POSIX 系统上，如果给定了可执行文件 `python3.x`，将创建指向该可执行文件的 `python` 和 `python3` 符号链接，除非相同名称的文件已经存在。

setup_scripts(*context*)

将适用于平台的激活脚本安装到虚拟环境中。

upgrade_dependencies(*context*)

升级环境中的核心 venv 依赖包 (目前为 pip)。这是通过将 shell 命令传出到环境中的 `pip` 可执行文件来完成的。

Added in version 3.9.

在 3.12 版本发生变更: setuptools 不再属于核心 venv 依赖项。

post_setup(*context*)

占位方法，可以在第三方实现中重写，用于在虚拟环境中预安装软件包，或是其他创建后要执行的步骤。

install_scripts(*context*, *path*)

此方法可以从子类的*setup_scripts()* 或*post_setup()* 调用以将自定义脚本安装到虚拟环境中。

path 是一个目录的路径，该目录应包含子目录 `common`, `posix`, `nt`；每个子目录中包含指向环境中 `bin` 目录的脚本。在一些占位符文本替换完毕后将会拷贝 `common` 和对应于*os.name* 的子目录的内容：

- `__VENV_DIR__` 会被替换为环境目录的绝对路径。
- `__VENV_NAME__` 会被替换为环境名称（环境目录的最后一个字段）。
- `__VENV_PROMPT__` 会被替换为提示符（用括号括起来的环境名称紧跟着一个空格）。
- `__VENV_BIN_NAME__` 会被替换为 bin 目录的名称（`bin` 或 `Scripts`）。
- `__VENV_PYTHON__` 会被替换为环境可执行文件的绝对路径。

允许目录已存在（用于升级现有环境时）。

create_git_ignore_file(*context*)

在虚拟环境中创建 `.gitignore` 文件将使整个目录被 Git 源代码控制管理器所忽略。

Added in version 3.13.

在 3.7.2 版本发生变更: Windows 现在为 `python[w].exe` 使用重定向脚本，而不是复制实际的二进制文件。仅在 3.7.2 中，除非运行的是源码树中的构建，否则*setup_python()* 不会执行任何操作。

在 3.7.3 版本发生变更: Windows 将重定向脚本复制为*setup_python()* 的一部分而非*setup_scripts()*。在 3.7.2 中不是这种情况。使用符号链接时，将链接至原始可执行文件。

有一个方便实用的模块级别的函数:

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, ***, *scm_ignore_files=frozenset()*)

通过关键词参数来创建一个*EnvBuilder*，并且使用 *env_dir* 参数来调用它的*create()* 方法。

Added in version 3.3.

在 3.4 版本发生变更: 增加了 *with_pip* 形参

在 3.6 版本发生变更: 增加了 *prompt* 形参

在 3.9 版本发生变更: 增加了 *upgrade_deps* 形参

在 3.13 版本发生变更: 增加了 *scm_ignore_files* 形参

29.2.4 一个扩展 EnvBuilder 的例子

下面的脚本展示了如何通过实现一个子类来扩展*EnvBuilder*。这个子类会安装 setuptools 和 pip 到被创建的虚拟环境中。

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv
```

(续下页)

(接上页)

```
class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
                     The context argument can have one of three values:
                     'main', indicating that it is called from virtualize()
                     itself, and 'stdout' and 'stderr', which are obtained
                     by reading lines from the output streams of a subprocess
                     which is used to install the app.

                     If a callable is not specified, default progress
                     information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # 没有 setuptools 无法安装 pip
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
```

(续下页)

(接上页)

```
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # 将脚本下载到虚拟环境的可执行文件目录
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # 安装虚拟环境
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # 清理 ——— 不再需要
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = "https://bootstrap.pypa.io/ez_setup.py"
        self.install_script(context, 'setuptools', url)
        # 清理已下载的 setuptools 归档
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)

    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
```

(续下页)

(接上页)

```
        """
        url = 'https://bootstrap.pypa.io/get-pip.py'
        self.install_script(context, 'pip', url)


def main(args=None):
    import argparse

    parser = argparse.ArgumentParser(prog=__name__,
                                     description='Creates virtual Python '
                                                 'environments in one or '
                                                 'more target '
                                                 'directories.')
    parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                        help='A directory in which to create the '
                             'virtual environment.')
    parser.add_argument('--no-setuptools', default=False,
                        action='store_true', dest='nodist',
                        help="Don't install setuptools or pip in the "
                             "virtual environment.")
    parser.add_argument('--no-pip', default=False,
                        action='store_true', dest='nopip',
                        help="Don't install pip in the virtual "
                             "environment.")
    parser.add_argument('--system-site-packages', default=False,
                        action='store_true', dest='system_site',
                        help='Give the virtual environment access to the '
                             'system site-packages dir.')
    if os.name == 'nt':
        use_symlinks = False
    else:
        use_symlinks = True
    parser.add_argument('--symlinks', default=use_symlinks,
                        action='store_true', dest='symlinks',
                        help='Try to use symlinks rather than copies, '
                             'when symlinks are not the default for '
                             'the platform.')
    parser.add_argument('--clear', default=False, action='store_true',
                        dest='clear', help='Delete the contents of the '
                                           'virtual environment '
                                           'directory if it already '
                                           'exists, before virtual '
                                           'environment creation.')
    parser.add_argument('--upgrade', default=False, action='store_true',
                        dest='upgrade', help='Upgrade the virtual '
                                             'environment directory to '
                                             'use this version of '
                                             'Python, assuming Python '
                                             'has been upgraded '
                                             'in-place.')
    parser.add_argument('--verbose', default=False, action='store_true',
                        dest='verbose', help='Display the output '
                                             'from the scripts which '
                                             'install setuptools and pip.')
    options = parser.parse_args(args)
    if options.upgrade and options.clear:
        raise ValueError('you cannot supply --upgrade and --clear together.')
    builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                 clear=options.clear,
                                 symlinks=options.symlinks,
                                 upgrade=options.upgrade,
```

(续下页)

(接上页)

```
                                    nodist=options.nodist,
                                    nopip=options.nopip,
                                    verbose=options.verbose)
    for d in options.dirs:
        builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

这个脚本同样可以 在线下载。

29.3 zipapp --- 管理可执行的 Python zip 归档文件

Added in version 3.5.

源代码： Lib/zipapp.py

本模块提供了一套管理工具，用于创建包含 Python 代码的压缩文件，这些文件可以 直接由 Python 解释器执行。本模块提供*命令行接口* 和*Python API*。

29.3.1 简单示例

下述例子展示了用*命令行接口* 根据含有 Python 代码的目录创建一个可执行的打包文件。运行后该打包文件时，将会执行 `myapp` 模块中的 `main` 函数。

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

29.3.2 命令行接口

若要从命令行调用，则采用以下形式：

```
$ python -m zipapp source [options]
```

如果 *source* 是个目录，将根据 *source* 的内容创建一个打包文件。如果 *source* 是个文件，则应为一个打包文件，将会复制到目标打包文件中（如果指定了 -info 选项，将会显示 shebang 行的内容）。

可以接受以下参数：

-o <output>, **--output**=<output>

将程序的输出写入名为 *output* 的文件中。若未指定此参数，输出的文件名将与输入的 *source* 相同，并添加扩展名 `.pyz`。如果显式给出了文件名，将会原样使用（因此必要时应包含扩展名 `.pyz`）。

如果 *source* 是个打包文件，必须指定一个输出文件名（这时 *output* 必须与 *source* 不同）。

-p <interpreter>, **--python**=<interpreter>

给打包文件加入 `#!` 行，以便指定 解释器作为运行的命令行。另外，还让打包文件在 POSIX 平台上可执行。默认不会写入 `#!` 行，也不让文件可执行。

-m <mainfn>, **--main**=<mainfn>

在打包文件中写入一个 `__main__.py` 文件，用于执行 *mainfn*。*mainfn* 参数的形式应为 “pkg.mod:fn”，其中 “pkg.mod” 是打包文件中的某个包/模块，“fn” 是该模块中的一个可调用对象。`__main__.py` 文件将会执行该可调用对象。

在复制打包文件时，不能设置 *--main* 参数。

-c, --compress

利用 deflate 方法压缩文件，减少输出文件的大小。默认情况下，打包文件中的文件是不压缩的。

在复制打包文件时，*--compress* 无效。

Added in version 3.7.

--info

显示嵌入在打包文件中的解释器程序，以便诊断问题。这时会忽略其他所有参数，SOURCE 必须是个打包文件，而不是目录。

-h, --help

打印简短的用法信息并退出。

29.3.3 Python API

该模块定义了两个快捷函数：

zipapp.**create_archive**(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)

由 *source* 创建一个应用程序打包文件。source 可以是以下形式之一：

- 一个目录名，或指向目录的*path-like object*，这时将根据目录内容新建一个应用程序打包文件。
- 一个已存在的应用程序打包文件名，或指向这类文件的*path-like object*，这时会将该文件复制为目标文件（会稍作修改以反映出 *interpreter* 参数的值）。必要时文件名中应包括 `.pyz` 扩展名。
- 一个以字节串模式打开的文件对象。该文件的内容应为应用程序打包文件，且假定文件对象定位于打包文件的初始位置。

target 参数定义了打包文件的写入位置：

- 若是个文件名，或是*path-like object*，打包文件将写入该文件中。
- 若是个打开的文件对象，打包文件将写入该对象，该文件对象必须在字节串写入模式下打开。
- 如果省略了 target （或为 `None`），则 source 必须为一个目录，target 将是与 source 同名的文件，并加上 `.pyz` 扩展名。

参数 *interpreter* 指定了 Python 解释器程序名，用于执行打包文件。这将以 “释伴（shebang）” 行的形式写入打包文件的头部。在 POSIX 平台上，操作系统会进行解释，而在 Windows 平台则会由 Python 启动器进行处理。省略 *interpreter* 参数则不会写入释伴行。如果指定了解释器，且目标为文件名，则会设置目标文件的可执行属性位。

参数 *main* 指定某个可调用程序的名称，用作打包文件的主程序。仅当 source 为目录且不含 `__main__.py` 文件时，才能指定该参数。*main* 参数应采用 “pkg.module:callable” 的形式，通过导入 “pkg.module” 并不带参数地执行给出的可调用对象，即可执行打包文件。如果 source 是目录且不含 `__main__.py` 文件，省略 *main* 将会出错，生成的打包文件将无法执行。

可选参数 *filter* 指定了回调函数，将传给代表被添加文件路径的 Path 对象（相对于源目录）。如若文件需要加入打包文件，则回调函数应返回 `True`。

可选参数 *compressed* 指定是否要压缩打包文件。若设为 `True`，则打包中的文件将用 deflate 方法进行压缩；否则就不会压缩。本参数在复制现有打包文件时无效。

若 *source* 或 *target* 指定的是文件对象，则调用者有责任在调用 create_archive 之后关闭这些文件对象。

当复制已有的打包文件时，提供的文件对象只需 `read` 和 `readline` 方法，或 `write` 方法。当由目录创建打包文件时，若目标为文件对象，将会将其传给类，且必须提供 `zipfile.ZipFile` 类所需的方法。

在 3.7 版本发生变更: 增加了 *filter* 和 *compressed* 形参。

zipapp.**get_interpreter**(*archive*)

返回打包文件开头的行指定的解释器程序。如果没有 `#!` 行，则返回*None*。参数 *archive* 可为文件名或在字节串模式下打开以供读取的文件型对象。`#!` 行假定是在打包文件的开头。

29.3.4 例子

将目录打包成一个文件并运行它。

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

同样还可用*create_archive()* 函数完成：

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

要让应用程序能在 POSIX 平台上直接执行，需要指定所用的解释器。

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

若要替换已有打包文件中的释伴行，请用*create_archive()* 函数另建一个修改好的打包文件：

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

若要原地更新打包文件，可用*BytesIO* 对象在内存中进行替换，然后再覆盖源文件。请注意原地覆盖文件存在发生错误时丢失原始文件的风险。这段代码没有考虑发生错误的情况，但生产性代码应该要考虑。另外，此方法将仅在内存能容纳打包文件时才适用：

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

29.3.5 指定解释器程序

注意，如果指定了解释器程序再发布应用程序打包文件，需要确保所用到的解释器是可移植的。Windows 的 Python 启动器支持大多数常见的 POSIX `#!` 行，但还需要考虑一些其他问题。

- 如果采用“/usr/bin/env python”（或其他格式的 python 调用命令，比如“/usr/bin/python”），需要考虑默认版本既可能是 Python 2 又可能是 Python 3，应让代码在两个版本下均能正常运行。
- 如果用到的 Python 版本明确，如“/usr/bin/env python3”，则没有该版本的用户将无法运行应用程序。（如果代码不兼容 Python 2，可能正该如此）。
- 因为无法指定“python X.Y 以上版本”，所以应小心“/usr/bin/env python3.4”这种精确版本的指定方式，因为对于 Python 3.5 的用户就得修改释伴行，比如：

通常应该用“/usr/bin/env python2”或“/usr/bin/env python3”的格式，具体根据代码适用于 Python 2 还是 3 而定。

29.3.6 用 zipapp 创建独立运行的应用程序

利用*zipapp* 模块可以创建独立运行的 Python 程序，以便向最终用户发布，仅需在系统中装有合适版本的 Python 即可运行。操作的关键就是把应用程序代码和所有依赖项一起放入打包文件中。

创建独立运行打包文件的步骤如下：

1. 照常在某个目录中创建应用程序，于是会有一个 myapp 目录，里面有个 __main__.py 文件，以及所有支持性代码。
2. 用 pip 将应用程序的所有依赖项装入 myapp 目录。

```
$ python -m pip install -r requirements.txt --target myapp
```

（这里假定在 requirements.txt 文件中列出了项目所需的依赖项，也可以在 pip 命令行中列出依赖项）。

3. 用以下命令打包：

```
$ python -m zipapp -p "interpreter" myapp
```

这会生成一个独立的可执行文件，可在任何装有合适解释器的机器上运行。详情参见指定解释器程序。可以单个文件的形式分发给用户。

在 Unix 系统中，myapp.pyz 文件将以原有文件名执行。如果喜欢“普通”的命令名，可以重命名该文件，去掉扩展名 .pyz 。在 Windows 系统中，myapp.pyz[w] 是可执行文件，因为 Python 解释器在安装时注册了扩展名 .pyz 和 .pyzw 。

注意事项

如果应用程序依赖某个带有 C 扩展的包，则此程序包无法由打包文件运行（这是操作系统的限制，因为可执行代码必须存在于文件系统中，操作系统才能加载）。这时可去除打包文件中的依赖关系，然后要求用户事先安装好该程序包，或者与打包文件一起发布并在 __main__.py 中增加代码，将未打包模块的目录加入 sys.path 中。采用增加代码方式时，一定要为目标架构提供合适的二进制文件（可能还需在运行时根据用户的机器选择正确的版本加入 sys.path）。

29.3.7 Python 打包应用程序的格式

自 2.6 版开始，Python 即能够执行包含文件的打包文件了。为了能被 Python 执行，应用程序的打包文件必须为包含 __main__.py 文件的标准 zip 文件，__main__.py 文件将作为应用程序的入口运行。类似于常规的 Python 脚本，父级（这里指打包文件）将放入*sys.path*，因此可从打包文件中导入更多的模块。

zip 文件格式允许在文件中预置任意数据。利用这种能力，zip 应用程序格式在文件中预置了一个标准的 POSIX “释伴”行（#!/path/to/interpreter）。

因此，Python zip 应用程序的格式会如下所示：

1. 可选的 shebang 行，包含字符 b'#!' 后面跟一个解释器名，再带一个换行符 (b'\n')。解释器名可以是 OS ”shebang” 处理所能接受的任何名称，或为 Windows 上的 Python 启动器。解释器名在 Windows 上应当使用 UTF-8 编码，而在 POSIX 上则使用*sys.getfilesystemencoding()*。
2. 标准的打包文件由*zipfile* 模块生成。其中 必须包含一个名为 __main__.py 的文件（必须位于打包文件的“根”目录——不能位于某个子目录中）。打包文件中的数据可以是压缩或未压缩的。

如果应用程序的打包文件带有释伴行，则在 POSIX 系统中可能需要启用可执行属性，以允许直接执行。

不一定非要用本模块中的工具创建应用程序打包文件，本模块只是提供了便捷方案，上述格式的打包文件可用任何方式创建，均可被 Python 接受。

CHAPTER 30

Python 运行时服务

本章描述的模块广泛服务于 Python 解释器及其与其环境的交互:

30.1 sys --- 系统相关的形参和函数

该模块提供了一些由解释器使用或维护的变量以及与解释器高强度交互的函数。它将始终可用。除非显式地说明例外情况，所有变量都是只读的。

sys.**abiflags**

在 POSIX 系统上，以标准的 configure 脚本构建的 Python 中，这个变量会包含 **PEP 3149** 中定义的 ABI 标签。

Added in version 3.2.

在 3.8 版本发生变更: 默认的 flags 变为了空字符串（用于 pymalloc 的 m 旗标已经移除）

Availability: Unix.

sys.**addaudithook**(*hook*)

将可调用的对象 *hook* 附加到当前（子）解释器的活动的审计钩子列表中。

当通过*sys.audit()* 函数引发审计事件时，每个钩子将按照其被加入的先后顺序被调用，调用时会传入事件名称和参数元组。由 PySys_AddAuditHook() 添加的原生钩子会先被调用，然后是当前（子）解释器中添加的钩子。接下来这些钩子会记录事件，引发异常来中止操作，或是完全终止进程。

请注意审计钩子主要是用于收集有关内部或在其他情况下不可观察操作的信息，可能是通过 Python 或者用 Python 编写的库。它们不适合用于实现“沙盒”。特别重要的一点是，恶意代码可以轻易地禁用或绕过使用此函数添加的钩子。至少，在初始化运行时之前必须使用 C API PySys_AddAuditHook() 来添加任何安全敏感的钩子，并且应当完全删除或密切监视任何允许任意修改内存的模块 (如*ctypes*)。

调用*sys.addaudithook()* 时它自身将引发一个名为 sys.addaudithook 的审计事件且不附带参数。如果任何现有的钩子引发了派生自*RuntimeError* 的异常，则新的钩子不会被添加并且该异常会被抑制。其结果就是，调用者无法确保他们的钩子已经被添加，除非他们控制了全部现有的钩子。

请参阅审计事件表 以获取由 CPython 引发的所有事件，并参阅 **PEP 578** 了解最初的设计讨论。

Added in version 3.8.

在 3.8.1 版本发生变更: 派生自*Exception*（而非*RuntimeError*）的异常不会被抑制。

启用跟踪时（参阅*settrace()*），仅当可调用对象（钩子）的 __cantrace__ 成员设置为 true 时，才会跟踪该钩子。否则，跟踪功能将跳过该钩子。

sys.**argv**

一个列表，其中包含了被传递给 Python 脚本的命令行参数。argv[0] 为脚本的名称（是否是完整的路径名取决于操作系统）。如果是通过 Python 解释器的命令行参数 -c 来执行的，argv[0] 会被设置成字符串 '-c'。如果没有脚本名被传递给 Python 解释器，argv[0] 为空字符串。

为了遍历标准输入，或者通过命令行传递的文件列表，参照*fileinput* 模块

另请参阅*sys.orig_argv*。

备注

在 Unix 上，系统传递的命令行参数是字节类型的。Python 使用文件系统编码和"surrogateescape"错误处理方案对它们进行解码。当需要原始字节时，可以通过 [os.fsencode(arg) for arg in sys.argv] 来获取。

sys.**audit**(*event*, **args*)

引发一个审计事件并触发任何激活的审计钩子。*event* 是一个用于标识事件的字符串，*args* 会包含有关事件的更多信息的可选参数。特定事件的参数的数量和类型会被视为是公有的稳定 API 且不应当在版本之间进行修改。

例如，有一个审计事件的名称为 os.chdir。此事件具有一个名为 *path* 的参数，该参数将包含所请求的新工作目录。

sys.audit() 将调用现有的审计钩子，传入事件名称和参数，并将重新引发来自任何钩子的第一个异常。通常来说，如果有一个异常被引发，则它不应当被处理且其进程应当被尽可能快地终止。这将允许钩子实现来决定对特定事件要如何反应：它们可以只是将事件写入日志或是通过引发异常来中止操作。

钩子程序由*sys.addaudithook()* 或 PySys_AddAuditHook() 函数添加。

与本函数相等效的原生函数是 PySys_Audit()，应尽量使用原生函数。

参阅审计事件表 以获取 CPython 定义的所有审计事件。

Added in version 3.8.

sys.**base_exec_prefix**

在 site.py 运行之前，Python 启动的时候被设置为跟*exec_prefix* 同样的值。如果不是运行在虚拟环境 中，两个值会保持相同；如果 site.py 发现处于一个虚拟环境中，*prefix* 和*exec_prefix* 将会指向虚拟环境。然而*base_prefix* 和*base_exec_prefix* 将仍然会指向基础的 Python 环境（用来创建虚拟环境的 Python 环境）

Added in version 3.3.

sys.**base_prefix**

在 site.py 运行之前，Python 启动的时候被设置为跟*prefix* 同样的值。如果不是运行在虚拟环境中，两个值会保持相同；如果 site.py 发现处于一个虚拟环境中，*prefix* 和*exec_prefix* 将会指向虚拟环境。然而*base_prefix* 和*base_exec_prefix* 将仍然会指向基础的 Python 环境（用来创建虚拟环境的 Python 环境）

Added in version 3.3.

sys.**byteorder**

本地字节顺序的指示符。在大端序（最高有效位优先）操作系统上值为 'big'，在小端序（最低有效位优先）操作系统上为 'little'。

sys.**builtin_module_names**

一个包含所有被编译进 Python 解释器的模块的名称的字符串元组。(此信息无法通过任何其他办法获取 --- `modules.keys()` 仅会列出导入的模块。)

另请参阅*sys.stdlib_module_names* 列表。

sys.**call_tracing**(*func*, *args*)

当启用跟踪时，调用 `func(*args)`。跟踪状态将被保存，并在以后恢复。这被设计为由调试器从某个检查点执行调用，以便递归地调试或分析某些其他代码。

在调用由*settrace()* 或*setprofile()* 设置的跟踪函数时跟踪将暂停以避免无限递归。`call_tracing()` 会启用跟踪函数的显式递归。

sys.**copyright**

一个字符串，包含了 Python 解释器有关的版权信息

sys.**_clear_type_cache**()

清除内部的类型缓存。类型缓存是为了加速查找方法和属性的。在调试引用泄漏的时候调用这个函数 只会清除不必要的引用。

这个函数应该只在内部为了一些特定的目的使用。

自 3.13 版本弃用: 改用更一般化的*_clear_internal_caches()* 函数。

sys.**_clear_internal_caches**()

清空所有内部性能相关的缓存。此函数的使用 仅限于释放不再需要的引用和寻找泄漏的内存块时。

Added in version 3.13.

sys.**_current_frames**()

返回一个字典，存放着每个线程的标识符与（调用本函数时）该线程栈顶的帧（当前活动的帧）之间的映射。注意*traceback* 模块中的函数可以在给定某一帧的情况下构建调用堆栈。

这对于调试死锁最有用：本函数不需要死锁线程的配合，并且只要这些线程的调用栈保持死锁，它们就是冻结的。在调用本代码来检查栈顶的帧的那一刻，非死锁线程返回的帧可能与该线程当前活动的帧没有任何关系。

这个函数应该只在内部为了一些特定的目的使用。

引发一个不带参数的审计事件 `sys._current_frames`。

sys.**_current_exceptions**()

返回一个字典，存放着每个线程的标识与调用此函数时该线程当前活动帧的栈顶异常之间的映射。如果某个线程当前未在处理异常，它将不被包括在结果字典中。

这对于静态性能分析来说最为有用。

这个函数应该只在内部为了一些特定的目的使用。

引发一个不带参数的审计事件 `sys._current_exceptions`。

在 3.12 版本发生变更: 现在字典中的每个值都是单独的异常实例，而不是如 `sys.exc_info()` 所返回的 3 元组。

sys.**breakpointhook**()

本钩子函数由内建函数*breakpoint()* 调用。默认情况下，它将进入*pdb* 调试器，但可以将其改为任何其他函数，以选择使用哪个调试器。

该函数的特征取决于其调用的函数。例如，默认绑定（即 `pdb.set_trace()`）不要求提供参数，但可以将绑定换成要求提供附加参数（位置参数/关键字参数）的函数。内建函数 `breakpoint()` 直接将其 `*args` 和 `**kws` 传入。`breakpointhooks()` 返回的所有内容都会从 `breakpoint()` 返回。

默认的实现首先会查询环境变量 `PYTHONBREAKPOINT`。如果将该变量设置为 `"0"`，则本函数立即返回，表示在断点处无操作。如果未设置该环境变量或将其设置为空字符串，则调用 `pdb.set_trace()`。否则，此变量应指定要运行的函数，指定函数时应使用 Python 的点导入命名法，如 `package.subpackage.module.function`。这种情况下将导入 `package.subpackage.module`，且

导入的模块必须有一个名为 function() 的可调用对象。该可调用对象会运行，*args 和 **kws 会传入，且无论 function() 返回什么，sys.breakpointhook() 都将返回到内建函数 breakpoint()。

请注意，如果在导入 PYTHONBREAKPOINT 指定的可调用对象时出错，则将报告一个 RuntimeWarning 并忽略断点。

另请注意，如果以编程方式覆盖 sys.breakpointhook()，则 不会查询 PYTHONBREAKPOINT。

Added in version 3.7.

sys._debugmallocstats()

将有关 CPython 内存分配器状态的底层的信息打印至 stderr。

如果 Python 是 以调试模式编译的 (使用 --with-pydebug 配置选项)，它还会执行某些高开销的内部一致性检查。

Added in version 3.3.

本函数仅限 CPython。此处没有定义确切的输出格式，且可能会更改。

sys.dllhandle

指向 Python DLL 句柄的整数。

Availability: Windows.

sys.displayhook(*value*)

如果 *value* 不是 None，则本函数会将 repr(value) 打印至 sys.stdout，并将 *value* 保存在 builtins._ 中。如果 repr(value) 无法用 sys.stdout.errors 错误处理方案（可能为 'strict'）编码为 sys.stdout.encoding，则用 'backslashreplace' 错误处理方案将其编码为 sys.stdout.encoding。

在交互式 Python 会话中运行 *expression* 产生结果后，将在结果上调用 sys.displayhook。若要自定义这些 value 的显示，可以将 sys.displayhook 指定为另一个单参数函数。

伪代码:

```
def displayhook(value):
    if value is None:
        return
    # 将 '_' 设为 None 以避免继续递归
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

在 3.2 版本发生变更: 在发生 *UnicodeEncodeError* 时使用 'backslashreplace' 错误处理方案。

sys.dont_write_bytecode

如果该值为 true，则 Python 在导入源码模块时将不会尝试写入 .pyc 文件。该值会被初始化为 True 或 False，依据是 -B 命令行选项和 PYTHONDONTWRITEBYTECODE 环境变量，可以自行设置该值，来控制是否生成字节码文件。

sys._emscripten_info

这个 *named tuple* 保存了 *wasm32-emscripten* 平台中环境的相关信息。该命名元组处于暂定状态并可能在将来被更改。

_emscripten_info.**emscripten_version**

以整数元组 (major, minor, micro) 表示的 Emscripten 版本，例如 `(3, 1, 8)`。

_emscripten_info.**runtime**

运行时字符串，例如 browser user agent, `'Node.js v14.18.2'` 或 `'UNKNOWN'`。

_emscripten_info.**pthreads**

如果 Python 编译附带了 Emscripten pthreads 支持则为 `True`。

_emscripten_info.**shared_memory**

如果 Python 编译附带了共享内存支持则为 `True`。

Availability: Emscripten.

Added in version 3.11.

sys.**pycache_prefix**

如果设置了该值 (不能为 `None`)，Python 会将字节码缓存文件 `.pyc` 写入到以该值指定的目录为根的并行目录树中（并从中读取），而不是在源代码树的 `__pycache__` 目录下读写。源代码树中所有的 `__pycache__` 目录都将被忽略并且新的 `.pyc` 文件将被写入到 pycache 前缀指定的位置。因此如果你使用*compileall* 作为预编译步骤，你必须确保使用与在运行时相同的 pycache 前缀（如果有的话）来运行它。

相对路径将解释为相对于当前工作目录。

该值的初值设置，依据 `-X pycache_prefix=PATH` 命令行选项或 `PYTHONPYCACHEPREFIX` 环境变量的值（命令行优先）。如果两者均未设置，则为 `None`。

Added in version 3.8.

sys.**excepthook**(*type*, *value*, *traceback*)

本函数会将所给的回溯和异常输出到 `sys.stderr` 中。

当有*SystemExit* 以外的异常被引发且未被捕获时，解释器会调用 `sys.excepthook` 并附带三个参数：异常类、异常实例和回溯对象。在交互会话中这将发生在控制返回提示符之前；在 Python 程序中这将发生在程序退出之前。这种最高层级异常的处理可以通过为 `sys.excepthook` 指定另一个三参数函数来实现自定义。

当发生未捕获的异常时，引发一个审计事件 `sys.excepthook`，附带参数 `hook`, `type`, `value`, `traceback`。如果没有设置钩子，`hook` 可能为 `None`。如果某个钩子抛出了派生自*RuntimeError* 的异常，则将禁止对该钩子的调用。否则，审计钩子的异常将被报告为无法抛出，并将调用 `sys.excepthook`。

参见

sys.unraisablehook() 函数处理无法抛出的异常，*threading.excepthook()* 函数处理*threading.Thread.run()* 抛出的异常。

sys.**__breakpointhook__**

sys.**__displayhook__**

sys.**__excepthook__**

sys.**__unraisablehook__**

程序开始时，这些对象存有 `breakpointhook`、`displayhook`、`excepthook` 和 `unraisablehook` 的初始值。保存它们是为了可以在 `breakpointhook`、`displayhook` 和 `excepthook`、`unraisablehook` 被破坏或被替换时恢复它们。

Added in version 3.7: __breakpointhook__

Added in version 3.8: __unraisablehook__

sys.**exception**()

当此函数在某个异常处理器执行过程中（如 except 或 except* 子句）被调用时，将返回被该处理器所捕获的异常实例。当有多个异常处理器彼此嵌套时，只有最内层处理器所处理的异常可以被访问到。

如果没有任何异常处理器在执行，此函数将返回 None。

Added in version 3.11.

sys.**exc_info**()

此函数返回被处理异常的旧式表示形式。如果异常 e 当前已被处理 (因此*exception()* 将会返回 e)，则*exc_info()* 将返回元组 (type(e), e, e.__traceback__)。也就是说，一个包含了该异常类型 (*BaseException* 的子类)，异常本身，以及通常封装了异常最后发生位置上调用栈的 回溯 对象的元组。

如果堆栈上的任何地方都没有处理异常，则此函数将返回一个包含三个 None 的元组。

在 3.11 版本发生变更: type 和 traceback 字段现在是派生自 value (异常实例)，因此当一个异常在处理期间被修改时，其变化会在后续对*exc_info()* 的调用结果中反映出来。

sys.**exec_prefix**

一个字符串，提供特定域的目录前缀，该目录中安装了与平台相关的 Python 文件，默认也是 '/usr/local'。该目录前缀可以在构建时使用 **configure** 脚本的 --exec-prefix 参数进行设置。具体而言，所有配置文件（如 pyconfig.h 头文件）都安装在目录 *exec_prefix*/lib/python*X.Y*/config 中，共享库模块安装在 *exec_prefix*/lib/python*X.Y*/lib-dynload 中，其中 *X.Y* 是 Python 的版本号，如 3.2。

备注

如果在一个虚拟环境 中，那么该值将在 site.py 中被修改，指向虚拟环境。Python 安装位置仍然可以用*base_exec_prefix* 来获取。

sys.**executable**

一个字符串，提供 Python 解释器的可执行二进制文件的绝对路径，仅在部分系统中此值有意义。如果 Python 无法获取其可执行文件的真实路径，则*sys.executable* 将为空字符串或 None。

sys.**exit**([*arg*])

引发一个*SystemExit* 异常，表示打算退出解释器。

可选参数 *arg* 可以是表示退出状态的整数（默认为 0），也可以是其他类型的对象。如果它是整数，则 shell 等将 0 视为 “成功终止”，非零值视为 “异常终止”。大多数系统要求该值的范围是 0--127，否则会产生不确定的结果。某些系统为退出代码约定了特定的含义，但通常尚不完善；Unix 程序通常用 2 表示命令行语法错误，用 1 表示所有其他类型的错误。传入其他类型的对象，如果传入 None 等同于传入 0，如果传入其他对象则将其打印至*stderr*，且退出代码为 1。特别地，sys.exit("some error message") 可以在发生错误时快速退出程序。

由于*exit()* 最终” 只” 引发了一个异常，它只在从主线程调用时退出进程，而异常不会被拦截。try 语句的 finally 子句所指定的清理动作会被遵守，并且有可能在外层拦截退出的尝试。

在 3.6 版本发生变更: 在 Python 解释器捕获*SystemExit* 后，如果在清理中发生错误（如清除标准流中的缓冲数据时出错），则退出状态码将变为 120。

sys.**flags**

具名元组 *flags* 含有命令行标志的状态。这些属性是只读的。

flags.**debug**	-d
flags.**inspect**	-i
flags.**interactive**	-i
flags.**isolated**	-I
flags.**optimize**	-O 或 -OO
flags.**dont_write_bytecode**	-B
flags.**no_user_site**	-s
flags.**no_site**	-S
flags.**ignore_environment**	-E
flags.**verbose**	-v
flags.**bytes_warning**	-b
flags.**quiet**	-q
flags.**hash_randomization**	-R
flags.**dev_mode**	-X dev (*Python 开发模式*)
flags.**utf8_mode**	-X utf8
flags.**safe_path**	-P
flags.**int_max_str_digits**	-X int_max_str_digits (*integer string conversion length limitation*)
flags.**warn_default_encoding**	-X warn_default_encoding

在 3.2 版本发生变更: 为新的 -q 标志添加了 quiet 属性

Added in version 3.2.3: hash_randomization 属性

在 3.3 版本发生变更: 删除了过时的 `division_warning` 属性

在 3.4 版本发生变更: 为 `-I isolated` 标志添加了 `isolated` 属性。

在 3.7 版本发生变更: 为新的*Python 开发模式* 添加了 `dev_mode` 属性，为新的 `-X utf8` 标志添加了 `utf8_mode` 属性。

在 3.10 版本发生变更: 为 `-X warn_default_encoding` 旗标添加了 `warn_default_encoding` 属性。

在 3.11 版本发生变更: 添加了用于 `-P` 选项的 `safe_path` 属性。

在 3.11 版本发生变更: 增加了 `int_max_str_digits` 属性。

`sys.`**`float_info`**

一个具名元组，存有浮点型的相关信息。它包含的是关于精度和内部表示的底层信息。这些值与标准头文件 `float.h` 中为 C 语言定义的各种浮点常量对应，详情请参阅 1999 ISO/IEC C 标准 [C99] 的 5.2.4.2.2 节，'Characteristics of floating types（浮点型的特性）'。

表 1: float_info named tuple 的属性

attribute -- 属性	float.h 宏	说明
float_info.**epsilon**	DBL_EPSILON	1.0 与可表示为浮点数的大于 1.0 的最小值之间的差。 另请参阅*math.ulp()*。
float_info.**dig**	DBL_DIG	浮点数可以真实表示的十进制数的最大位数；见下文。
float_info.**mant_dig**	DBL_MANT_DIG	浮点数精度：以 radix 为基数浮点数的有效位数。
float_info.**max**	DBL_MAX	可表示的最大正有限浮点数。
float_info.**max_exp**	DBL_MAX_EXP	使得 radix**(e-1) 是可表示的有限浮点数的最大整数 *e*。
float_info.**max_10_exp**	DBL_MAX_10_EXP	使得 10**e 在可表示的有限浮点数范围内的最大整数 *e*。
float_info.**min**	DBL_MIN	可表示的最小正 规范化浮点数。 使用*math.ulp(0.0)* 获取可表示的最小正 非规格化浮点数
float_info.**min_exp**	DBL_MIN_EXP	使得 radix**(e-1) 是规范化浮点数的最小整数 *e*。
float_info.**min_10_exp**	DBL_MIN_10_EXP	使得 10**e 是归范化浮点数的最小整数 *e*。
float_info.**radix**	FLT_RADIX	指数表示法中采用的基数。
float_info.**rounds**	FLT_ROUNDS	一个代表浮点运算舍入模式的整数。它反映了解释器启动时系统 FLT_ROUNDS 宏的值： • -1: 不确定 • 0: 向零值 • 1: 向最近值 • 2: 向正无穷 • 3: 向负无穷 FLT_ROUNDS 的所有其他值被用于代表具体实现所定义的舍入行为。

属性*sys.float_info.dig* 需要进一步的解释。如果 s 是表示十进制数的字符串，且最多有 sys.float_info.dig 位有效数字，那么将 s 转换为浮点数再转换回来将恢复为一个表示相同十进制值的字符串:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # 有 15 个有效位的十进制小数字节串
>>> format(float(s), '.15g')  # 转换为浮点数再转换回来 -> 相同的值
'3.14159265358979'
```

但是对于超过*sys.float_info.dig* 位有效数字的字符串，转换前后并非总是相同:

```
>>> s = '9876543211234567'    # 16 个有效位就太多了!
>>> format(float(s), '.16g')  # 转换将改变原值
'9876543211234568'
```

sys.**float_repr_style**

一个字符串，反映*repr()* 函数在浮点数上的行为。如果该字符串是 'short'，那么对于（非无穷的）浮点数 x，repr(x) 将会生成一个短字符串，满足 float(repr(x)) == x 的特性。这是 Python 3.1 及更高版本中的常见行为。否则 float_repr_style 的值将是 'legacy'，此时 repr(x) 的行为方式将与 Python 3.1 之前的版本相同。

Added in version 3.1.

sys.**getallocatedblocks**()

返回解释器当前已分配的内存块数，无论它们的大小如何。此函数主要用于跟踪和调试内存泄漏。因为解释器有内部缓存，所以不同调用的结果会有变化；你可能需要调用*_clear_internal_caches()* 和*gc.collect()* 来获得更可预测的结果。

如果一个 Python 构建或实现无法合理地计算此信息，则允许*getallocatedblocks()* 返回 0。

Added in version 3.4.

sys.**getunicodeinternedsize**()

返回已被处置的 unicode 对象数量。

Added in version 3.12.

sys.**getandroidapilevel**()

以一个整数的形式返回 Android 的构建时级别。这代表此 Python 构建版可运行的最小 Android 版本。对于运行时版本信息，请查看*platform.android_ver()*。

Availability: Android.

Added in version 3.7.

sys.**getdefaultencoding**()

返回 'utf-8'。这是默认字符编码格式的名称，被用于*str.encode()* 等方法。

sys.**getdlopenflags**()

返回用于 dlopen() 调用的旗标的当前值。旗标值的符号名称可在*os* 模块中找到 (RTLD_*xxx* 常量，例如*os.RTLD_LAZY*)。

Availability: Unix.

sys.**getfilesystemencoding**()

获取*文件系统编码格式*: 该编码格式与*文件系统错误处理器* 一起使用以便在 Unicode 文件名和字节文件名之间进行转换。文件系统错误处理器是从*getfilesystemencodeerrors()* 返回的。

为获得最佳兼容性，在任何时候都应使用 str 来表示文件名，尽管使用 bytes 来表示文件名也是受支持的。接受还返回文件名的函数应当支持 str 或 bytes 并在内部将其转换为系统首选的表示形式。

应使用*os.fsencode()* 和*os.fsdecode()* 来保证所采用的编码和错误处理方案都是正确的。

filesystem encoding and error handler 是在 Python 启动时通过 PyConfig_Read() 函数来配置的：请参阅 PyConfig 的 filesystem_encoding 和 filesystem_errors 等成员。

在 3.2 版本发生变更: *getfilesystemencoding()* 的结果将不再有可能是 None。

在 3.6 版本发生变更: Windows 不再保证会返回 'mbcs'。详情请参阅 **PEP 529** 和*_enablelegacywindowsfsencoding()*。

在 3.7 版本发生变更: 返回 'utf-8'，如果启用了*Python UTF-8 模式* 的话。

sys.**getfilesystemencodeerrors**()

获取文件系统错误处理器: 该错误处理器与文件系统编码格式一起使用以便在 Unicode 文件名和字节文件名之间进程转换。文件系统编码格式是由*getfilesystemencoding()* 来返回的。

应使用*os.fsencode()* 和*os.fsdecode()* 来保证所采用的编码和错误处理方案都是正确的。

filesystem encoding and error handler 是在 Python 启动时通过 `PyConfig_Read()` 函数来配置的：请参阅 `PyConfig` 的 `filesystem_encoding` 和 `filesystem_errors` 等成员。

Added in version 3.6.

sys.**get_int_max_str_digits**()

返回整数字符串转换长度限制 的当前值。另请参阅*set_int_max_str_digits()*。

Added in version 3.11.

sys.**getrefcount**(*object*)

返回 *object* 的引用计数。返回的计数通常比预期的多一，因为它包括了作为*getrefcount()* 参数的这一次（临时）引用。

请注意返回的值可能并不真正反映实际持有的对象引用数。例如，有些对象属于*immortal* 对象并具有并不反映实际引用数的非常高的 refcount 值。因此，除了 0 或 1 这两个值，不要依赖返回值的准确性。

在 3.12 版本发生变更: 永生对象具有与对象的实际引用次数不相符的非常大的引用计数。

sys.**getrecursionlimit**()

返回当前的递归限制值，即 Python 解释器堆栈的最大深度。此限制可防止无限递归导致的 C 堆栈溢出和 Python 崩溃。该值可以通过*setrecursionlimit()* 设置。

sys.**getsizeof**(*object*[, *default*])

返回对象的大小（以字节为单位）。该对象可以是任何类型。所有内建对象返回的结果都是正确的，但对于第三方扩展不一定正确，因为这与具体实现有关。

只计算直接分配给对象的内存消耗，不计算它所引用的对象的内存消耗。

对象不提供计算大小的方法时，如果传入过 *default* 则返回它，否则抛出*TypeError* 异常。

如果对象由垃圾回收器管理，则*getsizeof()* 将调用对象的 `__sizeof__` 方法，并在上层添加额外的垃圾回收器。

请参阅 recursive sizeof recipe 获取一个递归地使用*getsizeof()* 来找出各个容器及其全部内容大小的示例。

sys.**getswitchinterval**()

返回解释器的以秒为单位的“线程切换间隔时间”；参见*setswitchinterval()*。

Added in version 3.2.

sys.**_getframe**([*depth*])

返回来自调用栈的一个帧对象。如果传入可选整数 *depth*，则返回从栈顶往下相应调用层数的帧对象。如果该数比调用栈更深，则抛出*ValueError*。*depth* 的默认值是 0，返回调用栈顶部的帧。

引发一个审计事件 `sys._getframe` 并附带参数 `frame`。

这个函数应该只在内部为了一些特定的目的使用。不保证它在所有 Python 实现中都存在。

sys.**_getframemodulename**([*depth*])

从调用栈返回一个模块的名称。如果给出了可选的整数 *depth*，则返回从栈顶往下相应调用层数的模块。如果该数值比调用栈更深，或者如果该模块不可被标识，则返回 `None`。*depth* 的默认值为零，即返回位于调用栈顶端的模块。

引发一个审计事件 `sys._getframemodulename` 并附带参数 `depth`。

这个函数应该只在内部为了一些特定的目的使用。不保证它在所有 Python 实现中都存在。

sys.**getobjects**(*limit*[, *type*])

此函数仅当 CPython 使用专门的配置选项 `--with-trace-refs` 构建时才存在。它仅针对调试垃圾回收问题而设计。

返回由至多 *limit* 个动态分配的 Python 对象组成的列表。如果给定了 *type*，则仅会包括该特定类型（不包括子类型）的对象。

使用来自该列表的对象并不保证安全。具体来说，该结果将包括来自共享对象分配状态的所有解释器的对象 (即创建时是将 `PyInterpreterConfig.use_main_obmalloc` 设为 1 或是使用 `Py_NewInterpreter()`，以及 主解释器)。混合来自不同解释器的对象可能导致程序崩溃或其他非预期的行为。

此函数应当仅用于一些特定的目的。并不保证它在所有 Python 实现中都存在。

在 3.13.1 版本发生变更: 结果可能包括来自其他解释器的对象。

sys.**getprofile**()

返回由*setprofile()* 设置的性能分析函数。

sys.**gettrace**()

返回由*settrace()* 设置的跟踪函数。

gettrace() 函数仅用于实现调试器，性能分析器，打包工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此并非在所有 Python 实现中都可用。

sys.**getwindowsversion**()

返回一个具名元组，描述当前正在运行的 Windows 版本。元素名称包括 *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* 和 *platform_version*。*service_pack* 包含一个字符串，*platform_version* 包含一个三元组，其他所有值都是整数。元素也可以通过名称来访问，所以 `sys.getwindowsversion()[0]` 与 `sys.getwindowsversion().major` 是等效的。为保持与旧版本的兼容性，只有前 5 个元素可以用索引检索。

platform 将为 `2` (VER_PLATFORM_WIN32_NT)。

product_type 可能是以下值之一：

常量	含意
`1` (VER_NT_WORKSTATION)	系统是工作站。
`2` (VER_NT_DOMAIN_CONTROLLER)	系统是域控制器。
`3` (VER_NT_SERVER)	系统是服务器，但不是域控制器。

该函数包装了 Win32 `GetVersionEx()` 函数；有关这些字段的更多信息请参阅 `OSVERSIONINFOEX()` 的 Microsoft 文档。

platform_version 返回当前操作系统的主要版本、次要版本和编译版本号，而不是为该进程所模拟的版本。它旨在用于日志记录而非特性检测。

备注

platform_version 会从 kernel32.dll 获取版本号，这个版本可能与 OS 版本不同。请使用*platform* 模块来获取准确的 OS 版本号。

Availability: Windows.

在 3.2 版本发生变更: 更改为具名元组，添加 *service_pack_minor*, *service_pack_major*, *suite_mask* 和 *product_type*。

在 3.6 版本发生变更: 添加了 *platform_version*

sys.**get_asyncgen_hooks**()

返回一个 *asyncgen_hooks* 对象，该对象类似于 `(firstiter, finalizer)` 形式的*namedtuple*，其中 *firstiter* 和 *finalizer* 应为 `None` 或是一个接受*asynchronous generator iterator* 作为参数的函数，并被用来在事件循环中调度异步生成器的最终化。

Added in version 3.6: 详情请参阅 **PEP 525**。

> **备注**
>
> 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。

sys.**get_coroutine_origin_tracking_depth**()

获取由*set_coroutine_origin_tracking_depth()* 设置的协程来源的追踪深度。

Added in version 3.7.

> **备注**
>
> 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。仅将其用于调试目的。

sys.**hash_info**

一个具名元组，给出数字类型的哈希的实现参数。关于数字类型的哈希的详情请参阅数字类型的哈希运算。

hash_info.**width**
: 用于哈希值的位宽度

hash_info.**modulus**
: 用于数字哈案方案的质数模数 P

hash_info.**inf**
: 为正无穷大返回的哈希值

hash_info.**nan**
: （该属性已不再被使用）

hash_info.**imag**
: 用于复数虚部的乘数

hash_info.**algorithm**
: 对字符串、字节串和内存视图进行哈希的算法名称

hash_info.**hash_bits**
: 哈希算法的内部输出大小

hash_info.**seed_bits**
: 哈希算法种子密钥的大小

Added in version 3.2.

在 3.4 版本发生变更: 添加了 *algorithm*, *hash_bits* 和 *seed_bits*

sys.**hexversion**

编码为单个整数的版本号。该整数会确保每个版本都自增，其中适当包括了未发布版本。举例来说，要测试 Python 解释器的版本不低于 1.5.2，请使用:

```
if sys.hexversion >= 0x010502F0:
    # 使用某些高级特性
    ...
```

(续下页)

(接上页)

```
else:
    # 使用替代实现或警告用户
    ...
```

之所以称它为 hexversion，是因为只有将它传入内置函数 *hex()* 后，其结果才看起来有意义。也可以使用具名元组 *sys.version_info*，它对相同信息有着更人性化的编码。

关于 hexversion 的更多信息可以在 apiabiversion 中找到。

sys.**implementation**

一个对象，该对象包含当前运行的 Python 解释器的实现信息。所有 Python 实现中都必须存在下列属性。

name 是当前实现的标识符，如 'cpython'。实际的字符串由 Python 实现定义，但保证是小写字母。

version 是一个具名元组，格式与 *sys.version_info* 相同。它表示 Python 实现的版本。另一个（由 sys.version_info 表示）是当前解释器遵循的相应 Python 语言的版本，两者具有不同的含义。例如，对于 PyPy 1.8，sys.implementation.version 可能是 sys.version_info(1, 8, 0, 'final', 0)，而 sys.version_info 则是 sys.version_info(2, 7, 2, 'final', 0)。对于 CPython 而言两个值是相同的，因为它是参考实现。

hexversion 是十六进制的实现版本，类似于 *sys.hexversion*。

cache_tag 是导入机制使用的标记，用于已缓存模块的文件名。按照惯例，它将由实现的名称和版本组成，如 'cpython-33'。但如果合适，Python 实现可以使用其他值。如果 cache_tag 被置为 None，表示模块缓存已禁用。

sys.implementation 可能包含相应 Python 实现的其他属性。这些非标准属性必须以下划线开头，此处不详细阐述。无论其内容如何，*sys.implementation* 在解释器运行期间或不同实现版本之间都不会更改。（但是不同 Python 语言版本间可能会不同。）详情请参阅 **PEP 421**。

Added in version 3.3.

备注

新的必要属性的添加必须经过常规的 PEP 过程。详情请参阅 **PEP 421**。

sys.**int_info**

一个具名元组，包含 Python 内部整数表示形式的信息。这些属性是只读的。

int_info.**bits_per_digit**

每个数位占用的比特位数。Python 整数在内部以 2**int_info.bits_per_digit 为基数存储。

int_info.**sizeof_digit**

用于表示一个数位的 C 类型的以字节为单位的大小。

int_info.**default_max_str_digits**

sys.get_int_max_str_digits() 在未被显式配置时所使用的默认值。

int_info.**str_digits_check_threshold**

sys.set_int_max_str_digits()，PYTHONINTMAXSTRDIGITS 或 -X int_max_str_digits 的最小非零值。

Added in version 3.1.

在 3.11 版本发生变更: 添加了 *default_max_str_digits* 和 *str_digits_check_threshold*。

sys.**__interactivehook__**

当本属性存在，则以 交互模式启动解释器时，将自动（不带参数地）调用本属性的值。该过程是在读取 PYTHONSTARTUP 文件之后完成的，所以可以在该文件中设置这一钩子。*site* 模块设置了这一属性。

如果在启动时调用了钩子，则引发一个审计事件 cpython.run_interactivehook，附带参数为 hook 对象。

Added in version 3.4.

sys.**intern**(*string*)

将 *string* 插入"interned"（驻留）字符串表，返回被插入的字符串 -- 它是 *string* 本身或副本。驻留字符串对提高字典查找的性能很有用 -- 如果字典中的键已驻留，且所查找的键也已驻留，则键（取散列后）的比较可以用指针代替字符串来比较。通常，Python 程序使用到的名称会被自动驻留，且用于保存模块、类或实例属性的字典的键也已驻留。

驻留字符串不属于*immortal* 对象；你必须保留一个对*intern()* 返回值的引用才能发挥其优势。

sys.**_is_gil_enabled**()

如果*GIL* 已启用则返回*True* 而如果已禁用则返回*False*。

Added in version 3.13.

不保证存在于所有的 Python 实现。

sys.**is_finalizing**()

如果主 Python 解释器正在关闭 则返回*True*。在其他情况下返回*False*。

另请参阅*PythonFinalizationError* 异常。

Added in version 3.5.

sys.**last_exc**

该变量并非总是会被定义；当有未处理的异常时它将被设为相应的异常实例并且解释器将打印异常消息和栈回溯。它的预期用途是允许交互用户导入调试器模块并进行事后调试而不必重新运行导致了错误的命令。（典型用法是执行 import pdb; pdb.pm() 来进入事后调试器；请参阅*pdb* 了解详情。）

Added in version 3.12.

sys.**_is_interned**(*string*)

如果给定的字符串为“驻留字符串”则返回*True*，在其他情况下返回*False*。

Added in version 3.13.

不保证存在于所有的 Python 实现。

sys.**last_type**

sys.**last_value**

sys.**last_traceback**

这三个变量已被弃用；请改用*sys.last_exc*。它们将保存 sys.last_exc 的旧表示形式，如上面*exc_info()* 所返回的。

sys.**maxsize**

一个整数，表示 Py_ssize_t 类型的变量可以取到的最大值。在 32 位平台上通常为 2**31 - 1，在 64 位平台上通常为 2**63 - 1。

sys.**maxunicode**

一个整数，表示最大的 Unicode 码点值，如 1114111 （十六进制为 0x10FFFF ）。

在 3.3 版本发生变更: 在 **PEP 393** 之前，sys.maxunicode 曾是 0xFFFF 或 0x10FFFF，具体取决于配置选项，该选项指定将 Unicode 字符存储为 UCS-2 还是 UCS-4。

sys.**meta_path**

一个由*meta path finder* 对象组成的列表，这些对象的*find_spec()* 方法将会被调用以确定其中的某个对象能否找到要导入的模块。在默认情况下，它将存放实现了 Python 默认导入语法的条目。调用*find_spec()* 方法至少要附带待导入模块的绝对名称。如果待导入模块包含在一个包中，则父包的 __path__ 属性将作为第二个参数被传入。此方法将返回一个*module spec*，或者如果找不到模块则返回 None。

> **参见**
>
> ***importlib.abc.MetaPathFinder***
> 抽象基类，定义了*meta_path* 内的查找器对象的接口。
>
> ***importlib.machinery.ModuleSpec***
> *find_spec()* 返回的实例所对应的具体类。

在 3.4 版本发生变更: 模块规格说明 是在 Python 3.4 中根据 **PEP 451** 引入的。

在 3.12 版本发生变更: 移除了当*meta_path* 条目没有*find_spec()* 方法时查找 `find_module()` 方法的回调。

`sys.`**`modules`**

这是一个字典，它将模块名称映射到已经被加载的模块。这可以被操纵来强制重新加载模块和其他技巧。然而，替换这个字典不一定会像预期的那样工作，从字典中删除重要的项目可能会导致 Python 出错。如果你想对这个全局字典进行迭代，一定要使用 `sys.modules.copy()` 或 `tuple(sys.modules)` 来避免异常，因为它的大小在迭代过程中可能会因为其他线程中的代码或活动的副作用而改变。

`sys.`**`orig_argv`**

传给 Python 可执行文件的原始命令行参数列表。

sys.orig_argv 中的元素是传给 Python 解释器的参数，而*sys.argv* 中的元素则是传给用户程序的参数。解释器本身所使用的参数将出现在*sys.orig_argv* 中而不会出现在*sys.argv* 中。

Added in version 3.10.

`sys.`**`path`**

一个由字符串组成的列表，用于指定模块的搜索路径。初始化自环境变量 `PYTHONPATH`，再加上一条与安装有关的默认路径。

在默认情况下，如在程序启动时被初始化的时候，会有潜在的不安全路径被添加到*sys.path* 的开头 (在作为的 `PYTHONPATH` 结果被插入的条目 之前位置):

- `python -m module` 命令行：添加当前工作目录。
- `python script.py` 命令行：添加脚本的目录。如果是一个符号链接，则会解析符号链接。
- `python -c code` 和 `python` (REPL) 命令行：添加一个空字符串，这表示当前工作目录。

如果不想添加这个具有潜在不安全性的路径，请使用 `-P` 命令行选项或 `PYTHONSAFEPATH` 环境变量。

程序可以出于自己的目的随意修改此列表。应当只将字符串添加到*sys.path* 中；所有其他数据类型都将在导入期间被忽略。

> **参见**
>
> - *site* 模块，该模块描述了如何使用.pth 文件来扩展*sys.path*。

`sys.`**`path_hooks`**

一个由可调用对象组成的列表，这些对象接受一个路径作为参数，并尝试为该路径创建一个查找器。如果成功创建查找器，则可调用对象将返回它，否则将引发*ImportError* 异常。

本特性最早在 **PEP 302** 中被提及。

`sys.`**`path_importer_cache`**

一个字典，作为查找器 对象的缓存。key 是传入*sys.path_hooks* 的路径，value 是相应已找到的查找器。如果路径是有效的文件系统路径，但在*sys.path_hooks* 中未找到查找器，则存入 `None`。

本特性最早在 **PEP 302** 中被提及。

sys.**platform**

一个包含平台标识的字符串。已知的值有：

系统	平台值
AIX	`'aix'`
Android	`'android'`
Emscripten	`'emscripten'`
iOS	`'ios'`
Linux	`'linux'`
macOS	`'darwin'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
WASI	`'wasi'`

对于未在表中列出的 Unix 系统，该值是 `uname -s` 所返回的小写形式 OS 名称，并附加 `uname -r` 所返回的版本号的第一部分，例如 `'sunos5'` 或 `'freebsd8'`，对应 *Python* 被构建的时间。除非你想要检测特定的系统版本，否则建议使用以下惯例：

```
if sys.platform.startswith('freebsd'):
    # 在此添加 FreeBSD 专属的代码...
```

在 3.3 版本发生变更: 在 Linux 上，*sys.platform* 将不再包含主版本号。它将始终为 `'linux'`，而不是 `'linux2'` 或 `'linux3'`。

在 3.8 版本发生变更: 在 AIX 上，*sys.platform* 将不再包含主版本号。它将始终为 `'aix'`，而不是 `'aix5'` 或 `'aix7'`。

在 3.13 版本发生变更: 在 Android 上，*sys.platform* 现在将返回 `'android'` 而不是 `'linux'`。

参见

os.name 具有更粗的粒度。*os.uname()* 将给出依赖于具体系统的版本信息。

platform 模块对系统的标识有更详细的检查。

sys.**platlibdir**

平台专用库目录。用于构建标准库的路径和已安装扩展模块的路径。

在大多数平台上，它等同于 `"lib"`。在 Fedora 和 SuSE 上，它等同于给出了以下 `sys.path` 路径的 64 位平台上的 `"lib64"`（其中 `X.Y` 是 Python 的 `major.minor` 版本）。

- `/usr/lib64/pythonX.Y/`：标准库（如*os* 模块的 `os.py`）
- `/usr/lib64/pythonX.Y/lib-dynload/`：标准库的 C 扩展模块（如*errno* 模块，确切的文件名取决于平台）
- `/usr/lib/pythonX.Y/site-packages/` (请使用 `lib`, 而非*sys.platlibdir*): 第三方模块
- `/usr/lib64/pythonX.Y/site-packages/`: 第三方包的 C 扩展模块

Added in version 3.9.

sys.**prefix**

一个指定用于安装与平台无关的 Python 文件的站点专属目录前缀的字符串；在 Unix 上，默认为 `/usr/local`。这可以在构建时通过将 `--prefix` 参数传入 **configure** 脚本来设置。请参阅安装路径 了解衍生的路径。

> **备注**
>
> 如果在一个虚拟环境 中，那么该值将在 site.py 中被修改，指向虚拟环境。Python 安装位置仍然可以用*base_prefix* 来获取。

sys.**ps1**

sys.**ps2**

字符串，指定解释器的首要和次要提示符。仅当解释器处于交互模式时，它们才有定义。这种情况下，它们的初值为 '>>> ' 和 '... '。如果赋给其中某个变量的是非字符串对象，则每次解释器准备读取新的交互式命令时，都会重新运行该对象的*str()*，这可以用来实现动态的提示符。

sys.**setdlopenflags**(*n*)

设置解释器在调用 dlopen() 时使用的旗标，例如当解释器加载扩展模块的时候。首先，如果以 sys.setdlopenflags(0) 的形式调用的话这将在导入模块时启用符号的惰性求值。要在扩展模块之间共享符号，请以 sys.setdlopenflags(os.RTLD_GLOBAL) 的形式调用。旗标志值的符号名称可以在*os* 模块中找到 (RTLD_*xxx* 常量，例如*os.RTLD_LAZY*)。

Availability: Unix.

sys.**set_int_max_str_digits**(*maxdigits*)

设置解释器所使用的整数字符串转换长度限制。另请参阅*get_int_max_str_digits()*。

Added in version 3.11.

sys.**setprofile**(*profilefunc*)

设置系统的性能分析函数，该函数使得在 Python 中能够实现一个 Python 源代码性能分析器。关于 Python Profiler 的更多信息请参阅*Python 性能分析器* 章节。性能分析函数的调用方式类似于系统的跟踪函数（参阅*settrace()*），但它是通过不同的事件调用的，例如，不是每执行一行代码就调用它一次（仅在调用某函数和从某函数返回时才会调用性能分析函数，但即使某函数发生异常也会算作返回事件）。该函数是特定于单个线程的，但是性能分析器无法得知线程之间的上下文切换，因此在存在多个线程的情况下使用它是没有意义的。另外，因为它的返回值不会被用到，所以可以简单地返回 None。性能分析函数中的错误将导致其自身被解除设置。

> **备注**
>
> setprofile() 使用与*settrace()* 相同的跟踪机制。要在跟踪函数内部使用 setprofile() 来跟踪调用（例如在调试器断点内），请参阅*call_tracing()*。

性能分析函数应接收三个参数：*frame*、*event* 和 *arg*。*frame* 是当前的堆栈帧。*event* 是一个字符串：'call'、'return'、'c_call'、'c_return' 或 'c_exception'。*arg* 取决于事件类型。

这些事件具有以下含义：

'call'
表示调用了某个函数（或进入了其他的代码块）。性能分析函数将被调用，*arg* 为 None。

'return'
表示某个函数（或别的代码块）即将返回。性能分析函数将被调用，*arg* 是即将返回的值，如果此次返回事件是由于抛出异常，*arg* 为 None。

'c_call'
表示即将调用某个 C 函数。它可能是扩展函数或是内建函数。*arg* 是 C 函数对象。

'c_return'
表示返回了某个 C 函数。*arg* 是 C 函数对象。

'c_exception'
表示某个 C 函数抛出了异常。*arg* 是 C 函数对象。

引发一个不带参数的审计事件 sys.setprofile。

sys.**setrecursionlimit**(*limit*)

将 Python 解释器堆栈的最大深度设置为 *limit*。此限制可防止无限递归导致的 C 堆栈溢出和 Python 崩溃。

不同平台所允许的最高限值不同。当用户有需要深度递归的程序且平台支持更高的限值，可能就需要调高限值。进行该操作需要谨慎，因为过高的限值可能会导致崩溃。

如果新的限值低于当前的递归深度，将抛出*RecursionError* 异常。

在 3.5.1 版本发生变更: 如果新的限值低于当前的递归深度，现在将抛出*RecursionError* 异常。

sys.**setswitchinterval**(*interval*)

设置解释器的线程切换间隔时间（单位为秒）。该浮点数决定了“时间片”的理想持续时间，时间片将分配给同时运行的 Python 线程。请注意，实际值可能更高，尤其是使用了运行时间长的内部函数或方法时。同时，在时间间隔末尾调度哪个线程是操作系统的决定。解释器没有自己的调度程序。

Added in version 3.2.

sys.**settrace**(*tracefunc*)

设置系统的跟踪函数，使得用户在 Python 中就可以实现 Python 源代码调试器。该函数是特定于单个线程的，所以要让调试器支持多线程，必须为正在调试的每个线程都用*settrace()* 注册一个跟踪函数，或使用*threading.settrace()*。

跟踪函数应接收三个参数: *frame*、*event* 和 *arg*。*frame* 是当前的堆栈帧。*event* 是一个字符串: `'call'`、`'line'`、`'return'`、`'exception'` 或 `'opcode'`。*arg* 取决于事件类型。

每次进入 trace 函数的新的局部作用范围，都会调用 trace 函数（*event* 会被设置为 `'call'`），它应该返回一个引用，指向即将用在新作用范围上的局部跟踪函数；如果不需要跟踪当前的作用范围，则返回 `None`。

本地跟踪函数应返回对自身的引用，或对另一个函数的引用然后将其用作本作用域的局部跟踪函数。

如果跟踪函数出错，则该跟踪函数将被取消设置，类似于调用 `settrace(None)`。

> **备注**
>
> 在调用跟踪函数（例如由 `settrace()` 设置的函数）时将禁用跟踪。有关递归跟踪请参阅*call_tracing()*。

这些事件具有以下含义:

'call'
表示调用了某个函数（或进入了其他的代码块）。全局跟踪函数将被调用，*arg* 为 `None`。返回值将指定局部跟踪函数。

'line'
解释器即将执行一个新的代码行或重新执行一个循环的条件。局部跟踪函数将被调用；*arg* 为 `None`；其返回值将指定新的局部跟踪函数。请参阅 `Objects/lnotab_notes.txt` 查看有关其工作原理的详细说明。可以通过在某个 帧上把 `f_trace_lines` 设为*False* 来禁用相应帧的每行触发事件。

'return'
表示某个函数（或别的代码块）即将返回。局部跟踪函数将被调用，*arg* 是即将返回的值，如果此次返回事件是由于抛出异常，*arg* 为 `None`。跟踪函数的返回值将被忽略。

'exception'
表示发生了某个异常。局部跟踪函数将被调用，*arg* 是一个 `(exception, value, traceback)` 元组，返回值将指定新的局部跟踪函数。

'opcode'
解释器即将执行一个新的操作码（请参阅*dis* 了解有关操作码的详情）。局部跟踪函数将被调

用；*arg* 为 None；共返回值将指定新的局部跟踪函数。在默认情况下不会发出每个操作码触发事件：必须通过在某个 帧上把 f_trace_opcodes 设为 *True* 来显式地发出请求。

注意，由于异常是在链式调用中传播的，所以每一级都会产生一个 'exception' 事件。

更细微的用法是，可以显式地通过赋值 frame.f_trace = tracefunc 来设置跟踪函数，而不是用现有跟踪函数的返回值去间接设置它。当前帧上的跟踪函数必须激活，而 *settrace()* 还没有做这件事。注意，为了使上述设置起效，必须使用 *settrace()* 来安装全局跟踪函数才能启用运行时跟踪机制，但是它不必与上述是同一个跟踪函数（它可以是一个开销很低的跟踪函数，只返回 None，即在各个帧上立即将其自身禁用）。

关于代码对象和帧对象的更多信息请参考 types。

引发一个不带参数的审计事件 sys.settrace。

settrace() 函数仅用于实现调试器，性能分析器，打包工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此并非在所有 Python 实现中都可用。

在 3.7 版本发生变更：添加了 'opcode' 事件类型；为帧添加了 f_trace_lines 和 f_trace_opcodes 属性

sys.**set_asyncgen_hooks**(*[firstiter] [, finalizer]*)

接受两个可选的关键字参数，要求它们是可调用对象，且接受一个异步生成器迭代器 作为参数。*firstiter* 对象将在异步生成器第一次迭代时调用。*finalizer* 将在异步生成器即将被销毁时调用。

引发一个不带参数的审计事件 sys.set_asyncgen_hooks_firstiter。

引发一个不带参数的审计事件 sys.set_asyncgen_hooks_finalizer。

之所以会引发两个审计事件，是因为底层的 API 由两个调用组成，每个调用都须要引发自己的事件。

Added in version 3.6: 更多详情请参阅 **PEP 525**，*finalizer* 方法的参考示例可参阅 Lib/asyncio/base_events.py 中 asyncio.Loop.shutdown_asyncgens 的实现。

> **备注**
>
> 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。

sys.**set_coroutine_origin_tracking_depth**(*depth*)

允许启用或禁用协程溯源。当启用时，协程对象上的 cr_origin 属性将包含一个由 (文件名, 行号, 函数名) 元组组成的元组，它描述了协程对象自创建以来的回溯信息，最近的调用在最上面。当禁用时，cr_origin 将为 None。

要启用，请向 *depth* 传递一个大于零的值，它指定了有多少帧将被捕获信息。要禁用，请将 *depth* 置为零。

该设置是特定于单个线程的。

Added in version 3.7.

> **备注**
>
> 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。仅将其用于调试目的。

sys.**activate_stack_trampoline**(*backend*, */*)

激活栈性能分析器 trampoline *backend*。唯一受支持的后端是 "perf"。

Availability: Linux.

Added in version 3.12.

参见

- perf_profiling
- https://perf.wiki.kernel.org

sys.**deactivate_stack_trampoline**()

取消激活当前的栈性能分析器 trampoline 后端。

如果没有激活的栈性能分析器，此函数将没有任何效果。

Availability: Linux.

Added in version 3.12.

sys.**is_stack_trampoline_active**()

如果激活了栈性能分析器 trampoline 则返回 `True`。

Availability: Linux.

Added in version 3.12.

sys.**_enablelegacywindowsfsencoding**()

将*filesystem encoding and error handler* 分别修改为'mbcs' 和'replace'，以便与 3.6 之前版本的 Python 保持一致。

这等同于在启动 Python 前先定义好 `PYTHONLEGACYWINDOWSFSENCODING` 环境变量。

另请参阅*sys.getfilesystemencoding()* 和*sys.getfilesystemencodeerrors()*。

Availability: Windows.

备注

在 Python 启动后改变文件系统编码格式是有风险的因为旧的文件系统编码格式或由旧的文件系统编码格式所编码的路径可能已被缓存。请改用 `PYTHONLEGACYWINDOWSFSENCODING`。

Added in version 3.6: 更多详情请参阅 **PEP 529**。

Deprecated since version 3.13, will be removed in version 3.16: 应改用 `PYTHONLEGACYWINDOWSFSENCODING`。

sys.**stdin**

sys.**stdout**

sys.**stderr**

解释器用于标准输入、标准输出和标准错误的文件对象：

- `stdin` 用于所有交互式输入（包括对*input()* 的调用）；
- `stdout` 用于*print()* 和*expression* 语句的输出，以及用于*input()* 的提示符；
- 解释器自身的提示符和它的错误消息都发往 `stderr`。

这些流都是常规文本文件，与*open()* 函数返回的对象一致。它们的参数选择如下：

- 编码格式和错误处理器是由 `PyConfig.stdio_encoding` 和 `PyConfig.stdio_errors` 来初始化的。

  在 Windows 上，控制台设备使用 UTF-8。非字符设备如磁盘文件和管道使用系统语言区域编码格式（例如 ANSI 代码页）。非控制台字符设备如 NUL（例如当 `isatty()` 返回 `True` 时）会在启动时分别让 stdin 和 stdout/stderr 使用控制台输入和输出代码页。如果进程初始化时没有被附加到控制台则会使用默认的系统*locale encoding*。

要重写控制台的特殊行为，可以在启动 Python 前设置 PYTHONLEGACYWINDOWSSTDIO 环境变量。此时，控制台代码页将用于其他字符设备。

在所有平台上，都可以通过在 Python 启动前设置 `PYTHONIOENCODING` 环境变量来重写字符编码，或通过新的 `-X utf8` 命令行选项和 `PYTHONUTF8` 环境变量来设置。但是，对 Windows 控制台来说，上述方法仅在设置了 `PYTHONLEGACYWINDOWSSTDIO` 后才起效。

- 交互模式下，`stdout` 流是行缓冲的。其他情况下，它像常规文本文件一样是块缓冲的。两种情况下的 `stderr` 流都是行缓冲的。要使得两个流都变成无缓冲，可以传入 `-u` 命令行选项或设置 `PYTHONUNBUFFERED` 环境变量。

在 3.9 版本发生变更: 非交互模式下，`stderr` 现在是行缓冲的，而不是全缓冲的。

> **备注**
>
> 要从标准流写入或读取二进制数据，请使用底层二进制*buffer* 对象。例如，要将字节写入*stdout*，请使用 `sys.stdout.buffer.write(b'abc')`。
>
> 但是，如果你正在编写一个库（并且不能控制其代码执行所在的上下文），请注意标准流可能会被不支持 `buffer` 属性的文件型对象如*io.StringIO* 所取代。

`sys.__stdin__`

`sys.__stdout__`

`sys.__stderr__`

程序开始时，这些对象存有 `stdin`、`stderr` 和 `stdout` 的初始值。它们在程序结束前都可以使用，且在需要向实际的标准流打印内容时很有用，无论 `sys.std*` 对象是否已重定向。

如果实际文件已经被覆盖成一个损坏的对象了，那它也可用于将实际文件还原成能正常工作的文件对象。但是，本过程的最佳方法应该是，在原来的流被替换之前就显式地保存它，并使用这一保存的对象来还原。

> **备注**
>
> 某些情况下的 `stdin`、`stdout` 和 `stderr` 以及初始值 `__stdin__`、`__stdout__` 和 `__stderr__` 可以是 `None`。通常发生在未连接到控制台的 Windows GUI app 中，以及在用 **pythonw** 启动的 Python app 中。

`sys.stdlib_module_names`

一个包含标准库模组名称字符串的冻结集合。

它在所有平台上都保持一致。在某些平台上不可用的模块和在 Python 编译时被禁用的模块也会被列出。所有种类的模块都会被列出：纯 Python 模块、内置模块、冻结模块和扩展模块等。测试模块则会被排除掉。

对于包来说，仅会列出主包：子包和子模块不会被列出。例如，`email` 包会被列出，但 `email.mime` 子包和 `email.message` 子模块不会被列出。

另请参阅*sys.builtin_module_names* 列表。

Added in version 3.10.

`sys.thread_info`

一个*具名元组*，包含线程实现的信息。

`thread_info.name`

线程实现的名称：

- `"nt"`: Windows 线程
- `"pthread"`: POSIX 线程
- `"pthread-stubs"`: 转存 POSIX 线程（在不支持线程的 WebAssembly 平台上）

- "solaris": Solaris 线程

thread_info.**lock**

锁实现的名称：

- "semaphore": 锁使用一个寄存器
- "mutex+cond": 锁使用互斥和条件变量
- None 如果此信息未知

thread_info.**version**

线程库的名称和版本。它是一个字符串，如果此信息未知则为 None。

Added in version 3.3.

sys.**tracebacklimit**

当该变量值设置为整数，在发生未处理的异常时，它将决定打印的回溯信息的最大层级数。默认为 1000。当将其设置为 0 或小于 0，将关闭所有回溯信息，并且只打印异常类型和异常值。

sys.**unraisablehook**(*unraisable*, */*)

处理一个无法抛出的异常。

它会在发生了一个异常但 Python 没有办法处理时被调用。例如，当一个析构器引发了异常，或在垃圾回收 (*gc.collect()*) 期间引发了异常。

unraisable 参数具有以下属性:

- exc_type: 异常类型。
- exc_value: 异常值，可以为 None。
- exc_traceback: 异常回溯，可以为 None。
- err_msg: 错误消息，可以为 None。
- object: 导致异常的对象，可以为 None。

默认的钩子会将 err_msg 和 object 格式化为: f'{err_msg}: {object!r}'；如果 err_msg 为 None 则会使用"Exception ignored in" 错误消息。

要改变无法抛出的异常的处理过程，可以重写*sys.unraisablehook()*。

参见

excepthook() 处理未捕获的异常。

警告

使用自定义钩子存储 exc_value 可能会创建引用循环。当该异常不再需要时应当显式地清空以打破引用循环。

使用自定义钩子存储 object 可能会在它被设为正在终结的对象时将其复活。为避免对象复活应当避免在自定义钩子完成后存储 object。

当发生无法处理的异常时将引发一个审计事件 sys.unraisablehook，附带参数 *hook*、*unraisable*。其中 *unraisable* 对象与传递给钩子的对象相同。如果没有设置钩子，那么 *hook* 可以为 None。

Added in version 3.8.

sys.**version**

一个包含 Python 解释器版本号加编译版本号以及所用编译器等额外信息的字符串。此字符串会在交互式解释器启动时显示。请不要从中提取版本信息，而应当使用*version_info* 以及*platform* 模块所提供的函数。

sys.**api_version**

这个解释器的 C API 版本。当你在调试 Python 及期扩展模板的版本冲突这个功能非常有用。

sys.**version_info**

一个包含版本号五部分的元组: *major*, *minor*, *micro*, *releaselevel* 和 *serial*。除 *releaselevel* 外的所有值均为整数；发布级别值则为 `'alpha'`, `'beta'`, `'candidate'` 或 `'final'`。对应于 Python 版本 2.0 的 `version_info` 值为 `(2, 0, 0, 'final', 0)`。这些部分也可按名称访问，因此 `sys.version_info[0]` 就等价于 `sys.version_info.major`，依此类推。

在 3.1 版本发生变更: 增加了以名称表示的各部分属性。

sys.**warnoptions**

这是警告框架的一个实现细节；请不要修改此值。有关警告框架的更多信息请参阅*warnings* 模块。

sys.**winver**

用于在 Windows 平台上作为注册表键的版本号。这在 Python DLL 中被存储为 1000 号字符串资源。其值通常是正在运行的 Python 解释器的主要和次要版本号。它在*sys* 模块中提供是为了信息展示目的；修改此值不会影响 Python 所使用的注册表键。

Availability: Windows.

sys.**monitoring**

包含用于注册回调和控制监控事件的函数和常量的命名空间。详情参见*sys.monitoring*。

sys.**_xoptions**

一个字典，包含通过 `-X` 命令行选项传递的旗标，这些旗标专属于各种具体实现。选项名称将会映射到对应的值（如果显式指定）或者*True*。例如:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

这是 CPython 专属的访问通过 `-X` 传递的选项的方式。其他实现可能会通过其他方式导出它们，或者完全不导出。

Added in version 3.2.

引用

30.2 `sys.monitoring` --- 执行事件监测

Added in version 3.12.

备注

sys.monitoring 是*sys* 模块内部的一个命名空间，而不是一个独立模块，因此不需要 `import sys.monitoring`，只要简单地 `import sys` 然后使用 `sys.monitoring`。

这个命名空间提供了对于激活和控制事件监控所需的函数和常量的访问。

在程序执行过程中，会发生对于监控执行的工具来说值得关注的事件。*sys.monitoring* 命名空间提供了在相应事件发生时接收回调的操作方式。

monitoring API 由三个部分组成:

- *Tool identifiers*

- *Events*
- 回调

30.2.1 工具标识符

工具标识符是一个整数及其所关联的名称。工具标识符被用来防止工具之间的相互干扰并允许同时操作多个工作。目前工具是完全独立的且不能被用于相互监控。这一限制在将来可能会被取消。

在注册或激活事件之前，工具应选择一个标识符。标识符是 0 到 5 的开区间内的整数。

注册和使用工具

sys.monitoring.**use_tool_id**(*tool_id:* int, *name:* str, /) → *None*

必须在 *tool_id* 可被使用之前调用。*tool_id* 必须在 0 到 5 的开区间内。如果 *tool_id* 已被使用则会引发 *ValueError*。

sys.monitoring.**free_tool_id**(*tool_id:* int, /) → *None*

应当在一个工具不再需要 *tool_id* 时被调用。

备注

free_tool_id() 将不会禁用关联到 *tool_id* 的全局或局部事件，也不会注销任何回调函数。此函数仅被设计用来通知虚拟机特定的 *tool_id* 已不再被使用。

sys.monitoring.**get_tool**(*tool_id:* int, /) → *str* | *None*

如果 *tool_id* 已被使用则返回工具名称，否则返回 None。*tool_id* 取值必须在 0 至 5 的开区间内。

虚拟机在处理事件时对所有 ID 都一视同仁，但为便于工具之间的协作而预定义了下列 ID:

```
sys.monitoring.DEBUGGER_ID = 0
sys.monitoring.COVERAGE_ID = 1
sys.monitoring.PROFILER_ID = 2
sys.monitoring.OPTIMIZER_ID = 5
```

30.2.2 事件

以下事件是受支持的:

sys.monitoring.events.**BRANCH**

条件分支被采用（或不采用）。

sys.monitoring.events.**CALL**

Python 代码中的调用（事件发生在调用之前）。

sys.monitoring.events.**C_RAISE**

从任意可调用对象引发的异常。Python 函数除外（事件发生在退出之后）。

sys.monitoring.events.**C_RETURN**

从任意可调用对象返回，Python 函数除外（事件在返回之后发生）。

sys.monitoring.events.**EXCEPTION_HANDLED**

一个异常被处理。

sys.monitoring.events.**INSTRUCTION**

一个 VM 指令即将被执行。

sys.monitoring.events.**JUMP**

在控制流图中进行一次无条件的跳转。

sys.monitoring.events.**LINE**
一条与之前指令行号不同的指令即将被执行。

sys.monitoring.events.**PY_RESUME**
恢复执行一个 Python 函数（用于生成器和协程函数），`throw()` 调用除外。

sys.monitoring.events.**PY_RETURN**
从一个 Python 函数返回（在返回之前立即发生，被调用方的帧将在栈中）。

sys.monitoring.events.**PY_START**
开始一个 Python 函数（在调用之后立即发生，被调用方的帧将在栈中）

sys.monitoring.events.**PY_THROW**
一个 Python 函数由 `throw()` 调用恢复执行。

sys.monitoring.events.**PY_UNWIND**
在异常解除期间从一个 Python 函数退出。

sys.monitoring.events.**PY_YIELD**
从一个 Python 函数产出数据（在产出之前立即发生，被调用方的帧将在栈中）。

sys.monitoring.events.**RAISE**
一个异常被引发，导致*STOP_ITERATION* 事件的异常除外。

sys.monitoring.events.**RERAISE**
一个异常被重新引发，例如在 `finally` 代码块结束的时候。

sys.monitoring.events.**STOP_ITERATION**
一个*StopIteration* 被人工引发；参见*the STOP_ITERATION event*。

将来可能会添加更多事件。

这些事件都是 `sys.monitoring.events` 命名空间的属性。每个事件用整数常量的 2 次幂来表示。要定义一组事件，只需对多个单独事件执行按位或运算即可。例如，要同时指定*PY_RETURN* 和*PY_START* 事件，则使用表达式 `PY_RETURN | PY_START`。

sys.monitoring.events.**NO_EVENTS**
代表 `0` 的别名以便用户可以这样执行显式比较:

```
if get_events(DEBUGGER_ID) == NO_EVENTS:
    ...
```

事件被分为三组：

本地事件

本地事件与程序的正常执行相关联并且发生在明确定义的位置上。所有本地事件都可以被禁用。本地事件包括：

- *PY_START*
- *PY_RESUME*
- *PY_RETURN*
- *PY_YIELD*
- *CALL*
- *LINE*
- *INSTRUCTION*
- *JUMP*
- *BRANCH*
- *STOP_ITERATION*

辅助事件

辅助事件可以像其他事件一样被监视，但是由另一个事件来控制：

- *C_RAISE*
- *C_RETURN*

C_RETURN 和*C_RAISE* 事件是由*CALL* 事件控制的。*C_RETURN* 和*C_RAISE* 事件只会在相应的*CALL* 事件被监控时才能被看到。

其他事件

其他事件不一定与程序中的特定位置相关联并且不能被单独禁用。

可以被监视的其他事件包括：

- *PY_THROW*
- *PY_UNWIND*
- *RAISE*
- *EXCEPTION_HANDLED*

STOP_ITERATION 事件

PEP 380 规定了当从生成器或协程返回值时可引发*StopIteration* 异常。不过，这是一种非常低效的返回值的方式，因此某些 Python 实现，比如 CPython 3.12+，只有在异常对其他代码可见时才会引发它。

为允许工具监视真正的异常而不会拖慢生成器和协程的运行，解释器提供了*STOP_ITERATION* 事件。*STOP_ITERATION* 可以被局部禁用，这与*RAISE* 不同。

30.2.3 开启和关闭事件

要监视一个事件，它必须被开启并注册相应的回调函数。可以通过将事件设置为全局的或针对特定代码对象的来开启或关闭事件。

全局设置事件

通过修改被监视的事件集可以对事件进行全局控制。

sys.monitoring.**get_events**(*tool_id:* int, /) → *int*

返回代表所有活动事件的 int。

sys.monitoring.**set_events**(*tool_id:* int, *event_set:* int, /) → *None*

激活在 *event_set* 中设置的所有事件。如果 *tool_id* 未被使用则会引发*ValueError*。

在默认情况下没有被激活的事件。

针对特定代码对象的事件

事件也可以基于每个代码对象来控制。下面定义的接受一个*types.CodeType* 的函数应当准备好接受来自不是在 Python 中定义的类似对象 (参见 c-api-monitoring)。

sys.monitoring.**get_local_events**(*tool_id:* int, *code:* CodeType, /) → *int*

返回 *code* 的所有局部事件

sys.monitoring.**set_local_events**(*tool_id:* int, *code:* CodeType, *event_set:* int, /) → *None*

激活在 *event_set* 中设置的针对 *code* 的所有局部事件。如果 *tool_id* 未被使用则会引发*ValueError*。

局部事件将添加到全局事件中，但不会屏蔽全局事件。换句话说，所有全局事件都会为代码对象触发，无论是否有局部事件。

禁用事件

sys.monitoring.**DISABLE**

一个可从回调函数返回以禁用当前代码位置上的事件的特殊值。

可从回调函数返回*sys.monitoring.DISABLE* 以禁用特定代码位置上的局部事件。这不会改变已设置的事件，也不会改变同一事件的任何其他代码位置。

禁用特定位置的事件对高性能的监控非常重要。例如，如果调试器禁用了除几个断点外的所有监控那么程序在调试器下运行时就不会产生额外的开销。

sys.monitoring.**restart_events**() → *None*

启用*sys.monitoring.DISABLE* 针对所有工具禁用的所有事件。

30.2.4 注册回调函数

要为事件注册一个可调用对象则要调用

sys.monitoring.**register_callback**(*tool_id:* int, *event:* int, *func:* Callable | None, /) → *Callable* | *None*

使用给定的 *tool_id* 为 *event* 注册可调用对象 *func*

如果已经为给定的 *tool_id* 和 *event* 注册了另一个回调，它将被注销并返回。在其他情况下*register_callback()* 将返回 None。

函数可以通过调用 sys.monitoring.register_callback(tool_id, event, None) 来注销。

回调函数可在任何时候被注册或注销。

注册或注销回调函数将生成一个*sys.audit()* 事件。

回调函数参数

sys.monitoring.**MISSING**

一个传给回调函数表明该调用不附带任何参数的特殊值。

当一个激活的事件发生时，已注册的回调函数将被调用。不同的事件将为回调函数提供不同的参数，如下所示：

- *PY_START* 和*PY_RESUME*:

```
func(code: CodeType, instruction_offset: int) -> DISABLE | Any
```

- *PY_RETURN* 和*PY_YIELD*:

```
func(code: CodeType, instruction_offset: int, retval: object) -> DISABLE | Any
```

- *CALL*, *C_RAISE* 和*C_RETURN*:

```
func(code: CodeType, instruction_offset: int, callable: object, arg0: object | MISSING)␣
↪-> DISABLE | Any
```

如果没有任何参数，则 *arg0* 将被设为*sys.monitoring.MISSING*。

- *RAISE*, *RERAISE*, *EXCEPTION_HANDLED*, *PY_UNWIND*, *PY_THROW* 和*STOP_ITERATION*:

```
func(code: CodeType, instruction_offset: int, exception: BaseException) -> DISABLE | Any
```

- *LINE*:

```
func(code: CodeType, line_number: int) -> DISABLE | Any
```

- *BRANCH* 和*JUMP*:

```
func(code: CodeType, instruction_offset: int, destination_offset: int) -> DISABLE | Any
```

请注意 *destination_offset* 是代码下一次执行的位置。对于未进入的分支这将为该分支之后的指令的偏移量。

- *INSTRUCTION*:

```
func(code: CodeType, instruction_offset: int) -> DISABLE | Any
```

30.3 sysconfig --- 提供对 Python 配置信息的访问

Added in version 3.2.

源代码: Lib/sysconfig

---

sysconfig 模块提供了对 Python 配置信息的访问支持，比如安装路径列表和有关当前平台的配置变量。

30.3.1 配置变量

一个包含 `Makefile` 和 `pyconfig.h` 头文件的 Python 分发版，这是构建 Python 二进制文件本身和用 `setuptools` 编译的第三方 C 扩展所必需的。

sysconfig 将这些文件中的所有变量放在一个字典对象中，可用*get_config_vars()* 或*get_config_var()* 访问。

请注意在 Windows 上，这是一个小得多的集合。

sysconfig.**get_config_vars**(**args*)

不带参数时，返回一个与当前平台相关的所有配置变量的字典。

带参数时，返回一个由在配置变量字典中查找每个参数的结果的值组成的列表。

对于每个参数，如果未找到值，则返回 `None`。

sysconfig.**get_config_var**(*name*)

返回单个变量 *name* 的值。等价于 `get_config_vars().get(name)`。

如果未找到 *name*，则返回 `None`。

用法示例:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

30.3.2 安装路径

Python 会使用根据平台和安装选项区别处理的安装方案。这些方案被存储在*sysconfig* 中基于*os.name* 返回的值来确定的唯一标识符下。软件包安装程序使用这些方案来确定将文件复制到何处。

Python 目前支持九种方案:

- *posix_prefix*: 针对 POSIX 平台如 Linux 或 macOS 的方案。这是在安装 Python 或者组件时的默认方案。
- *posix_home*: 当使用 *home* 选项时，针对 POSIX 平台的方案。该方案定义了位于特定 home 前缀下的路径。
- *posix_user*: 当使用 *user* 选项时，针对 POSIX 平台的方案。该方案定义了位于用户主目录 (*site.USER_BASE*) 下的路径。

- *posix_venv*: 针对 POSIX 平台上*Python 虚拟环境* 的方案；在默认情况下与 *posix_prefix* 相同。
- *nt*: 针对 Windows 的方案。这是在安装 Python 或其组件时的默认方案。
- *nt_user*: 针对 Windows, 当使用了 *user* 选项时的方案。
- *nt_venv*: 针对 Windows 上*Python 虚拟环境* 的方案；在默认情况下与 *nt* 相同。
- *venv*: 根据 Python 运行所在平台的不同来设置 *posix_venv* 或 *nt_venv* 的值的方案。
- *osx_framework_user*: 针对 macOS，当使用了 *user* 选项时的方案。

每个方案本身由一系列路径组成并且每个路径都有唯一的标识符。Python 目前使用了八个路径:

- *stdlib*: 包含非平台专属的标准 Python 库文件的目录。
- *platstdlib*: 包含平台专属的标准 Python 库文件的目录。
- *platlib*: 用于站点专属、平台专属的文件的目录。
- *purelib*: 用于站点专属、非平台专属的文件（‘纯’ Python）的目录。
- *include*: 针对用于 Python C-API 的非平台专属头文件的目录。
- *platinclude*: 针对用于 Python C-API 的平台专属头文件的目录。
- *scripts*: 用于脚本文件的目录。
- *data*: 用于数据文件的目录。

30.3.3 用户方案

此方案被设计为针对没有全局 site-packages 目录写入权限或不想安装到该目录的用户的最便捷解决方案。

文件将被安装到*site.USER_BASE* (以下称为 *userbase*) 的子目录中。此方案将在同一个位置 (或称*site.USER_SITE*) 中安装纯 Python 模块和扩展模块。

posix_user

Path	安装目录
stdlib	*userbase*/lib/python*X.Y*
platstdlib	*userbase*/lib/python*X.Y*
platlib	*userbase*/lib/python*X.Y*/site-packages
purelib	*userbase*/lib/python*X.Y*/site-packages
include	*userbase*/include/python*X.Y*
scripts	*userbase*/bin
data	*userbase*

nt_user

Path	安装目录
stdlib	*userbase*\Python*XY*
platstdlib	*userbase*\Python*XY*
platlib	*userbase*\Python*XY*\site-packages
purelib	*userbase*\Python*XY*\site-packages
include	*userbase*\Python*XY*\Include
scripts	*userbase*\Python*XY*\Scripts
data	*userbase*

osx_framework_user

Path	安装目录
stdlib	*userbase*/lib/python
platstdlib	*userbase*/lib/python
platlib	*userbase*/lib/python/site-packages
purelib	*userbase*/lib/python/site-packages
include	*userbase*/include/python*X.Y*
scripts	*userbase*/bin
data	*userbase*

30.3.4 主方案

“主方案”背后的理念是你可以构建并维护个人的 Python 模块集。该方案的名称源自 Unix 上“主目录”的概念，因为通常 Unix 用户会将其主目录的布局设置为与 `/usr/` 或 `/usr/local/` 相似。任何人都可以使用该方案，无论其安装的操作系统是什么。

posix_home

Path	安装目录
stdlib	*home*/lib/python
platstdlib	*home*/lib/python
platlib	*home*/lib/python
purelib	*home*/lib/python
include	*home*/include/python
platinclude	*home*/include/python
scripts	*home*/bin
data	*home*

30.3.5 前缀方案

“前缀方案”适用于当你希望使用一个 Python 安装程序来执行构建/安装（即运行 setup 脚本），但需要将模块安装到另一个 Python 安装版（或看起来类似于另一个 Python 安装版）的第三方模块目录中的情况。如果这听起来有点不寻常，确实如此 --- 这就是为什么要先介绍用户和主目录方案的原因。然而，至少有两种已知的情况会用到前缀方案。

首先，许多 Linux 发行版都会将 Python 放在 `/usr` 中，而不是传统的 `/usr/local` 中。这是完全适当的，因为在这些情况下，Python 是“系统”的一部分而不是本地的附加组件。但是，如果你从源代码安装 Python 模块，您可能会想要将它们放在 `/usr/local/lib/python2.`*X* 而不是 `/usr/lib/python2.`*X* 中。

另一种可能性是在用于写入远程目录的名称与用于读取该目录的名称不同的网络文件系统：例如，作为 `/usr/local/bin/python` 访问的 Python 解释器可能会在 `/usr/local/lib/python2.`*X* 中搜索模块，但这些模块又必须安装到 `/mnt/`*@server*`/export/lib/python2.`*X* 这样的地方。

posix_prefix

Path	安装目录
stdlib	*prefix*/lib/python*X.Y*
platstdlib	*prefix*/lib/python*X.Y*
platlib	*prefix*/lib/python*X.Y*/site-packages
purelib	*prefix*/lib/python*X.Y*/site-packages
include	*prefix*/include/python*X.Y*
platinclude	*prefix*/include/python*X.Y*
scripts	*prefix*/bin
data	*prefix*

nt

Path	安装目录
stdlib	*prefix*\Lib
platstdlib	*prefix*\Lib
platlib	*prefix*\Lib\site-packages
purelib	*prefix*\Lib\site-packages
include	*prefix*\Include
platinclude	*prefix*\Include
scripts	*prefix*\Scripts
data	*prefix*

30.3.6 安装路径函数

sysconfig 提供了一些函数来确定这些安装路径。

sysconfig.**get_scheme_names**()

返回一个包含*sysconfig* 目前支持的所有方案的元组。

sysconfig.**get_default_scheme**()

返回针对当前平台的默认方案的名称。

Added in version 3.10: 此函数之前被命名为 `_get_default_scheme()` 并被认为属性实现细节。

在 3.11 版本发生变更: 当 Python 运行于虚拟环境时，将返回 *venv* 方案。

sysconfig.**get_preferred_scheme**(*key*)

返回针对由 *key* 所指定的安装布局的推荐方案的名称。

key 必须为 `"prefix"`, `"home"` 或 `"user"`。

该返回值是*get_scheme_names()* 中列出的一个方案名称。它可以被传给接受 *scheme* 参数的*sysconfig* 函数，如*get_paths()*。

Added in version 3.10.

在 3.11 版本发生变更: 当 Python 运行于虚拟环境且 `key="prefix"` 时，将返回 *venv* 方案。

sysconfig.**_get_preferred_schemes**()

返回一个包含当前平台推荐的方案名称的字典。Python 的实现方和再分发方可以将他们推荐的方案添加到 `_INSTALL_SCHEMES` 模块层级全局值，并修改此函数以返回这些方案名称，例如为各种系统和语言的包管理器提供不同的方案，这样它们各自安装的包就不会彼此混淆。

最终用户不应使用此函数，而应改用*get_default_scheme()* 和*get_preferred_scheme()*。

Added in version 3.10.

sysconfig.**get_path_names**()

返回一个包含在*sysconfig* 中目前支持的所有路径名称的元组。

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])

返回一个对应于路径 *name*，来自名为 *scheme* 的安装方案的安装路径。

name 必须是一个来自*get_path_names()* 所返回的列表的值。

sysconfig 会针对每个平台保存与每个路径名称相对应的安装路径，并带有可扩展的变量。例如针对 *nt* 方案的 *stdlib* 路径是: `{base}/Lib`。

get_path() 将使用*get_config_vars()* 所返回的变量来扩展路径。所有变量对于每种平台都有相应的默认值因此使用者可以调用此函数来获取默认值。

如果提供了 *scheme*，则它必须是一个来自*get_scheme_names()* 所返回的列表的值。在其他情况下，将会使用针对当前平台的默认方案。

如果提供了 *vars*，则它必须是一个将要更新*get_config_vars()* 所返回的字典的变量字典。

如果 *expand* 被设为 `False`，则将不使用这些变量来扩展路径。

如果 *name* 未找到，则会引发*KeyError*。

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])

返回一个包含与特定安装方案对应的安装路径的字典。请参阅*get_path()* 了解详情。

如果未提供 *scheme*，则将使用针对当前平台的默认方案。

如果提供了 *vars*，则它必须是一个将要更新用于扩展的字典的变量字典。

如果 *expand* 被设为假值，则路径将不会被扩展。

如果 *scheme* 不是一个现有的方案，则*get_paths()* 将引发*KeyError*。

30.3.7 其他功能

sysconfig.**get_python_version**()

以字符串形式 `MAJOR.MINOR` 返回 Python 版本号。类似于 `'%d.%d' % sys.version_info[:2]`。

sysconfig.**get_platform**()

返回一个标识当前平台的字符串。

这主要被用来区分平台专属的构建目录和平台专属的构建分发版。通常包括 OS 名称和版本以及架构（即*os.uname()* 所提供的信息），但是实际包括的信息取决于具体 OS；例如，在 Linux 上，内核版本并不是特别重要。

返回值的示例：

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows 将返回以下之一：

- win-amd64 (在 AMD64 即 x86_64, Intel64 和 EM64T 上的 64 位 Windows)
- win-arm64 (在 ARM64 即 AArch64 上的 64 位 Windows)
- win32（所有其他的——确切地说，返回 sys.platform）

macOS 可以返回:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

对于其他非-POSIX 平台，目前只是返回*sys.platform* 。

sysconfig.**is_python_build**()

如果正在运行的 Python 解释器是使用源代码构建的并在其构建位置上运行，而不是在其他位置例如通过运行 `make install` 或通过二进制机器码安装程序安装则返回 `True`。

sysconfig.**parse_config_h**(*fp*[, *vars*])

解析一个 `config.h` 风格的文件。

fp 是一个指向 `config.h` 风格的文件的文件型对象。

返回一个包含名称/值对的字典。如果传入一个可选的字典作为第二个参数，则将使用它而不是新的字典，并使用从文件中读取的值更新它。

sysconfig.**get_config_h_filename**()

返回 `pyconfig.h` 的目录

sysconfig.**get_makefile_filename**()

返回 Makefile 的目录

30.3.8 将 sysconfig 作为脚本使用

You can use *sysconfig* as a script with Python's *-m* option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

此调用将把*get_platform()*, *get_python_version()*, *get_path()* 和*get_config_vars()* 所返回的信息打印至标准输出。

30.4 builtins --- 内置对象

此模块提供了对 Python 的所有‘内置’标识符的直接访问；例如，builtins.open 是内置函数 function *open()* 的完整名称。

大多数应用程序通常不会显式访问此模块，但在提供与内置值同名的对象的模块中可能很有用，但其中还需要内置该名称。例如，在一个想要实现*open()* 函数的模块中，它包装了内置的*open()* ，这个模块可以直接使用:

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''文件的包装类，将读取的内容转换为大写输出'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ..
```

作为一个实现细节，大多数模块都将名称 __builtins__ 作为其全局变量的一部分提供。__builtins__ 的值通常是这个模块或者这个模块的值 __dict__ 属性。由于这是一个实现细节，因此 Python 的替代实现可能不会使用它。

参见

- 内置常量
- 内置异常
- 内置函数
- 内置类型

30.5 __main__ --- 最高层级代码环境

Python 的特殊名 __main__ 用于两个重要的构造：

1. 程序的顶层环境的名称，可用表达式 __name__ == '__main__' 来检查；以及
2. Python 包中的文件 __main__.py。

这两个机制都与 Python 模块相关——用户与它们如何交互，及它们之间如何交互——下文详述。而教程的 tut-modules 一节则为初学者介绍了 Python 模块。

30.5.1 __name__ == '__main__'

当一个 Python 模块或包被导入时，__name__ 被设为模块的名称——通常为 Python 文件本身的名称去掉 .py 后缀：

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

如果文件是包的一部分，则 __name__ 还将包括父包的路径：

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

而若模块是在顶层代码环境中执行的，则其 __name__ 被设为字符串 '__main__'。

什么是“顶层代码环境”？

__main__ 是顶层代码运行环境的名称。“顶层代码”是指由用户指定的最先开始运行的那一个 Python 模块。之所以它是“顶层”，是因为它将导入程序所需的所有其它模块。有时“顶层代码”被称为应用程序的 入口点。

顶层代码环境可以是：

- 交互提示符的作用域：

```
>>> __name__
'__main__'
```

- 作为文件参数传给 Python 解释器的 Python 模块：

```
$ python helloworld.py
Hello, world!
```

- 与 -m 一起传给 Python 解释器的 Python 模块或包：

```
$ python -m tarfile
usage: tarfile.py [-h] [-v] (...)
```

- Python 解释器从标准输入中读取的 Python 代码：

```
$ echo "import this" | python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

- 与 -c 一起传给 Python 解释器的 Python 代码：

```
$ python -c "import this"
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

上述每种情况中的顶层模块的 __name__ 被设为 '__main__'。

作为结果，模块通过检查自己的 __name__ 可发现自己是否运行于顶层环境，使一些代码仅当模块不是被导入语句初始化的时候才执行：

```
if __name__ == '__main__':
    # 将在模块不是由于 import 语句被初始化的情况下执行。
    ...
```

参见

关于在所有情况下 __name__ 是被如何设置的，详见教程的 tut-modules 一节。

惯用法

有些模块包含了仅供脚本使用的代码，比如解析命令行参数或从标准输入获取数据。如果这样的模块被从不同的模块中导入，例如为了单元测试，脚本代码也会无意中执行。

这就是 `if __name__ == '__main__'` 代码块的用武之地。除非模块在顶层环境中被执行，否则该块内的代码不会运行。

将尽可能少的语句放在位于 `if __name__ == '__main__'` 之下的代码块中可以提高代码的清晰度和准确度。通常，将由一个名为 `main` 的函数来封装程序的主要行为：

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
   """一个对 print 的简单包装器。"""
   # 出于演示目的，你可以想象在此函数内部
   # 存在有价值且可重用的逻辑
   print(phrase)

def main() -> int:
    """将输入参数回显到标准输出"""
    phrase = shlex.join(sys.argv)
```

(续下页)

(接上页)

```
    echo(phrase)
    return 0

if __name__ == '__main__':
    sys.exit(main())  # 下一节将讲解 sys.exit 的使用
```

请注意，如果模块没有将代码封装在 main 函数内，而是直接放在 if __name__ == '__main__' 块内，那么这个 phrase 变量对整个模块来说就是全局变量。这很容易出错，因为模块内的其他函数可能会无意中使用全局变量而不是局部名称。一个 main 函数解决了这个问题。

使用 main 函数有一个额外的好处，就是 echo 函数本身是孤立的，可以在其他地方导入。当 echo.py 被导入时，echo 和 main 函数将被定义，但它们都不会被调用，因为 __name__ != '__main__' 。

打包考量

main 函数经常被用来创建命令行工具，把它们指定为控制台脚本的入口点。当这样做时，pip 将函数调用插入到模板脚本中，其中 main 的返回值被传递到*sys.exit()* 。例如:

```
sys.exit(main())
```

由于 main 调用被包裹在*sys.exit()* 中，期望你的函数将返回一些可被*sys.exit()* 作为输入而接受的值；通常为一个整数或 None （如果你的函数没有返回语句，则隐含返回）。

通过主动遵循这一惯例，我们的模块在直接运行时 (即 python echo.py) 会有相同的行为，当我们以后把它打包成可用 pip 安装的软件包的控制台脚本入口时也会如此。

特别的是，要小心从你的 main 函数中返回字符串。*sys.exit()* 将把一个字符串参数解释为失败信息，所以你的程序将有一个 1 的退出代码，表示失败。并且这个字符串将被写入*sys.stderr* 。前面的 echo.py 例子举例说明了使用 sys.exit(main()) 的约定。

参见

Python 打包用户指南 包含了一系列关于如何用现代工具分发和安装 Python 包的教程和参考资料。

30.5.2 Python 包中的 __main__.py

如果你不熟悉 Python 包，请参阅本教程的 tut-packages 一节。最常见的是，__main__.py 文件被用来为一个包提供命令行接口。假设有下面这个虚构的包，”bandclass”:

```
bandclass
  ├── __init__.py
  ├── __main__.py
  └── student.py
```

当使用 -m 标志从命令行直接调用软件包本身时，将执行 __main__.py 。比如说。

```
$ python -m bandclass
```

这个命令将导致 __main__.py 的运行。你如何利用这一机制将取决于你所编写的软件包的性质，但在这个假设的案例中，允许教师搜索学生可能是有意义的:

```
# bandclass/__main__.py

import sys
from .student import search_students

student_name = sys.argv[1] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

注意，`from .student import search_students` 是一个相对导入的例子。这种导入方式可以在引用一个包内的模块时使用。更多细节，请参见教程 tut-modules 中的 intra-package-references 一节。

惯用法

`__main__.py` 的内容通常不会用 `if __name__ == '__main__'` 块围起来。相反，这些文件会保持简短并从其他模块导入函数来执行。这样其他模块就可以很容易地进行单元测试并可以适当地重用。

如果使用，一个 `if __name__ == '__main__'` 区块仍然会像预期的那样对包内的 `__main__.py` 文件起作用，因为如果导入，它的 `__name__` 属性将包括包的路径:

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

但这对 `.zip` 文件的根目录中的 `__main__.py` 文件不起作用。因此，为了保持一致性，建议使用不带 `__name__` 检测的最小化 `__main__.py`。

参见

请参阅*venv* 以获取标准库中具有最小化 `__main__.py` 的软件包示例。它不包含 `if __name__ == '__main__'` 代码块。你可以用 `python -m venv [directory]` 来唤起。

参见*runpy* 以了解更多关于 `-m` 标志对解释器可执行包的细节。

参见*zipapp* 了解如何运行打包成 *.zip* 文件的应用程序。在这种情况下，Python 会在归档文件的根目录下寻找一个 `__main__.py` 文件。

30.5.3 `import __main__`

不管 Python 程序是用哪个模块启动的，在同一程序中运行的其他模块可以通过导入 `__main__` 模块来导入顶级环境的范围 (*namespace*)。这并不是导入一个 `__main__.py` 文件，而是导入使用特殊名称 `'__main__'` 的哪个模块。

下面是一个使用 `__main__` 命名空间的模块的例子:

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    if '__file__' in dir(__main__):
        print(__main__.my_name, "found in file", __main__.__file__)
    else:
        print(__main__.my_name)
```

该模块的用法示例如下:

```
# start.py

import sys

from namely import print_user_name

# my_name = "Dinsdale"
```

(续下页)

(接上页)

```
def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)

if __name__ == "__main__":
    sys.exit(main())
```

现在，如果我们启动我们的程序，结果会是这样的:

```
$ python start.py
Define the variable `my_name`!
```

该程序的退出代码为 1，表明有错误。取消对 `my_name = "Dinsdale"` 这一行的注释，就可以修复程序，现在它的退出状态代码为 0，表示成功。

```
$ python start.py
Dinsdale found in file /path/to/start.py
```

请注意，导入 `__main__` 不会导致无意中运行旨在用于脚本的顶层代码的问题，这些代码被放在模块 `start` 的 `if __name__ == "__main__"` 块中。为什么这样做?

Python 解释器启动时会在*sys.modules* 中插入一个空的 `__main__` 模块，并通过运行最高层级代码来填充它。在我们的例子中这就是 `start` 模块，它逐行运行并导入 `namely`。相应地，`namely` 会导入 `__main__` (它实际上就是 `start`)。这就是一个导入循环！幸运的是，由于部分填充的 `__main__` 模块存在于*sys.modules* 中，Python 会将其传递给 `namely`。请参阅导入系统的参考文档中 有关 __main__ 的特别考量来了解其中的详情。

Python REPL 是另一个" 顶层环境" 的例子，所以在 REPL 中定义的任何东西都成为 `__main__` 范围的一部分:

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
>>> namely.print_user_name()
Jabberwocky
```

注意，在这种情况下，`__main__` 范围不包含 `__file__` 属性，因为它是交互式的。

`__main__` 范围用于*pdb* 和*rlcompleter* 的实现。

30.6 warnings --- 警告信息控制

源代码: Lib/warnings.py

通常以下情况会引发警告：提醒用户注意程序中的某些情况，而这些情况（通常）还不值得触发异常并终止程序。例如，当程序用到了某个过时的模块时，就可能需要发出一条警告。

Python 程序员可调用本模块中定义的*warn()* 函数来发布警告。（C 语言程序员则用 `PyErr_WarnEx()`；详见 exceptionhandling ）。

警告信息通常会写入*sys.stderr*，但可以灵活改变，从忽略所有警告到变成异常都可以。警告的处理方式可以依据警告类型、警告信息的文本和发出警告的源位置而进行变化。同一源位置重复出现的警告通常会被抑制。

控制警告信息有两个阶段：首先，每次引发警告时，决定信息是否要发出；然后，如果要发出信息，就用可由用户设置的钩子进行格式化并打印输出。

警告过滤器控制着是否发出警告信息，也即一系列的匹配规则和动作。调用*filterwarnings()*可将规则加入过滤器，调用*resetwarnings()*则可重置为默认状态。

警告信息的打印输出是通过调用*showwarning()*完成的，该函数可被重写；默认的实现代码是调用*formatwarning()*进行格式化，自己编写的代码也可以调用此格式化函数。

参见

利用*logging.captureWarnings()*可以采用标准的日志架构处理所有警告。

30.6.1 警告类别

警告的类别由一些内置的异常表示。这种分类有助于对警告信息进行分组过滤。

虽然在技术上警告类别属于内置异常，但也只是在此记录一下而已，因为在概念上他们属于警告机制的一部分。

通过对某个标准的警告类别进行派生，用户代码可以定义其他的警告类别。警告类别必须是*Warning*类的子类。

目前已定义了以下警告类别的类：

类	描述
Warning	这是所有警告类别的基类。它是*Exception*的子类。
UserWarning	The default category for *warn()*.
DeprecationWarning	已废弃特性警告的基类，这些警告是为其他 Python 开发者准备的（默认会忽略，除非在 __main__ 中用代码触发）。
SyntaxWarning	用于警告可疑语法的基类。
RuntimeWarning	用于警告可疑运行时特性的基类。
FutureWarning	用于警告已废弃特性的基类，这些警告是为 Python 应用程序的最终用户准备的。
PendingDeprecationWarning	用于警告即将废弃功能的基类（默认忽略）。
ImportWarning	导入模块时触发的警告的基类（默认忽略）。
UnicodeWarning	用于 Unicode 相关警告的基类。
BytesWarning	*bytes*和*bytearray*相关警告的基类。
ResourceWarning	资源使用相关警告的基础类别（默认会被忽略）。ignored by default).

在 3.7 版本发生变更: 以前*DeprecationWarning*和*FutureWarning*是根据某个功能是否完全删除或改变其行为来区分的。现在是根据受众和默认警告过滤器的处理方式来区分的。

30.6.2 警告过滤器

警告过滤器控制着警告是否被忽略、显示或转为错误（触发异常）。

从概念上讲，警告过滤器维护着一个经过排序的过滤器类别列表；任何具体的警告都会依次与列表中的每种过滤器进行匹配，直到找到一个匹配项；过滤器决定了匹配项的处理方式。每个列表项均为 (*action* , *message* , *category* , *module* , *lineno*) 格式的元组，其中：

- *action* 是以下字符串之一：

值	处置
"default"	为发出警告的每个位置（模块 + 行号）打印第一个匹配警告
"error"	将匹配警告转换为异常
"ignore"	从不打印匹配的警告
"always"	总是打印匹配的警告
"module"	为发出警告的每个模块打印第一次匹配警告（无论行号如何）
"once"	无论位置如何，仅打印第一次出现的匹配警告

- *message* 是一个包含警告消息的开头需要匹配的正则表达式的字符串，对大小写不敏感。在 -W 和 PYTHONWARNINGS 中，*message* 是警告消息的开头需要包含的字符串字面值（对大小写不敏感），将忽略 *message* 开头和末尾的任何空格。
- *category* 是警告类别的类（*Warning* 的子类），警告类别必须是其子类，才能匹配。
- *module* 是一个包含完整限定模块名称的开头需要匹配的正则表达式的字符串，对大小写敏感。在 -W 和 PYTHONWARNINGS 中，*module* 是完整限定模块名称需要与之相等的字符串字面值（对大小写敏感），将忽略 *module* 开头和末尾的任何空格。
- *lineno* 是个整数，发生警告的行号必须与之匹配，或为 0 表示与所有行号匹配。

由于*Warning* 类是由内置类*Exception* 派生出来的，要把某个警告变成错误，只要触发 category(message) 即可。

如果警告不匹配所有已注册的过滤器，那就会应用“default”动作（正如其名）。

重复警告的屏蔽准则

屏蔽重复警告的过滤器将应用以下准则来确定一个警告是否会被视为重复:

- "default": 一个警告仅在 (*message*, *category*, *module*, *lineno*) 全部相同时会被视为重复。
- "module": 一个警告将在 (*message*, *category*, *module*) 相同时会被视为重复，即忽略行号。
- "once": 一个警告将在 (*message*, *category*) 相同时会被视为重复，即忽略模块和行号。

警告过滤器的介绍

警告过滤器由传给 Python 解释器的命令行 -W 选项和 PYTHONWARNINGS 环境变量初始化。解释器在 sys.warningoptions 中保存了所有给出的参数，但不作解释；*warnings* 模块在第一次导入时会解析这些参数（无效的选项被忽略，并会先向*sys.stderr* 打印一条信息）。

每个警告过滤器的设定格式为冒号分隔的字段序列:

```
action:message:category:module:line
```

这些字段的含义在警告过滤器 中描述。当一行中列出多个过滤器时（如 PYTHONWARNINGS），过滤器间用逗号隔开，后面的优先于前面的（因为是从左到右应用的，最近应用的过滤器优先于前面的）。

常用的警告过滤器适用于所有的警告、特定类别的警告、由特定模块和包引发的警告。下面是一些例子:

```
default                       # 显示所有警告（即使是默认被忽略的）
ignore                        # 忽略所有警告
error                         # 将所有警告转换为错误
error::ResourceWarning        # 将 ResourceWarning 消息视为错误
default::DeprecationWarning   # 显示 DeprecationWarning 消息
ignore,default:::mymodule     # 只报告由 "mymodule" 触发的警告
error:::mymodule              # 将 "mymodule" 中的警告转换为错误
```

默认警告过滤器

Python 默认安装了几个警告过滤器，可以通过 `-W` 命令行参数、`PYTHONWARNINGS` 环境变量及调用*filterwarnings()* 进行覆盖。

在常规发布的版本中，默认的警告过滤器包括（按优先顺序排列）：

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

在 调试版本中，默认警告过滤器的列表是空的。

在 3.2 版本发生变更: 除了*PendingDeprecationWarning* 之外，*DeprecationWarning* 现在默认会被忽略。

在 3.7 版本发生变更: *DeprecationWarning* 在被 `__main__` 中的代码直接触发时，默认会再次显示。

在 3.7 版本发生变更: 如果指定两次 `-b`，则*BytesWarning* 不再出现在默认的过滤器列表中，而是通过 `sys.warningoptions` 进行配置。

重写默认的过滤器

Python 应用程序的开发人员可能希望在默认情况下向用户隐藏 所有 Python 级别的警告，而只在运行测试或其他调试时显示这些警告。用于向解释器传递过滤器配置的 `sys.warningoptions` 属性可以作为一个标记，表示是否应该禁用警告：

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

建议 Python 代码测试的开发者使用如下代码，以确保被测代码默认显示 所有警告：

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # 更改此进程中的过滤器
    os.environ["PYTHONWARNINGS"] = "default" # 也会影响子进程
```

最后，建议在 `__main__` 以外的命名空间运行用户代码的交互式开发者，请确保*DeprecationWarning* 在默认情况下是可见的，可采用如下代码（这里 `user_ns` 是用于执行交互式输入代码的模块）：

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

30.6.3 暂时禁止警告

如果明知正在使用会引起警告的代码，比如某个废弃函数，但不想看到警告（即便警告已经通过命令行作了显式配置），那么可以使用*catch_warnings* 上下文管理器来抑制警告。

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
```

(续下页)

(接上页)

```
    warnings.simplefilter("ignore")
    fxn()
```

在上下文管理器中，所有的警告将被简单地忽略。这样就能使用已知的过时代码而又不必看到警告，同时也不会限制警告其他可能不知过时的代码。注意：只能保证在单线程应用程序中生效。如果两个以上的线程同时使用*catch_warnings* 上下文管理器，行为不可预知。

30.6.4 测试警告

要测试由代码引发的警告，请采用*catch_warnings* 上下文管理器。有了它，就可以临时改变警告过滤器以方便测试。例如，以下代码可捕获所有的警告以便查看：

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # 使得所有警告总是会被触发。
    warnings.simplefilter("always")
    # 触发一个警告。
    fxn()
    # 执行一些查验
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

也可以用 `error` 取代 `always`，让所有的警告都成为异常。需要注意的是，如果某条警告已经因为 `once` / `default` 规则而被引发，那么无论设置什么过滤器，该条警告都不会再出现，除非该警告有关的注册数据被清除。

一旦上下文管理器退出，警告过滤器将恢复到刚进此上下文时的状态。这样在多次测试时可防止意外改变警告过滤器，从而导致不确定的测试结果。模块中的*showwarning()* 函数也被恢复到初始值。注意：这只能在单线程应用程序中得到保证。如果两个以上的线程同时使用*catch_warnings* 上下文管理器，行为未定义。

当测试多项操作会引发同类警告时，重点是要确保每次操作都会触发新的警告（比如，将警告设置为异常并检查操作是否触发异常，检查每次操作后警告列表的长度是否有增加，否则就在每次新操作前将以前的警告列表项删除）。

30.6.5 为新版本的依赖关系更新代码

在默认情况下，主要针对 Python 开发者（而不是 Python 应用程序的最终用户）的警告类别，会被忽略。

值得注意的是，这个“默认忽略”的列表包含*DeprecationWarning*（适用于每个模块，除了 `__main__`），这意味着开发人员应该确保在测试代码时应将通常忽略的警告显示出来，以便未来破坏性 API 变化时及时收到通知（无论是在标准库还是第三方包）。

理想情况下，代码会有一个合适的测试套件，在运行测试时会隐含地启用所有警告（由*unittest* 模块提供的测试运行程序就是如此）。

在不太理想的情况下，可以通过向 Python 解释器传入 `-Wd`（这是 `-W default` 的简写）或设置环境变量 `PYTHONWARNINGS=default` 来检查应用程序是否用到了已弃用的接口。这样可以启用对所有警告的默认处理操作，包括那些默认忽略的警告。要改变遇到警告后执行的动作，可以改变传给 `-W` 的参数（例如 `-W error`）。请参阅 `-W` 旗标来了解更多的细节。

30.6.6 可用的函数

warnings.**warn**(*message*, *category=None*, *stacklevel=1*, *source=None*, *, *skip_file_prefixes=()*)

引发警告、忽略或者触发异常。如果给出 *category* 参数，则必须是警告类别类；默认为*UserWarning*。或者 *message* 可为*Warning* 的实例，这时 *category* 将被忽略，转而采用 `message.__class__`。在这种情况下，错误信息文本将是 `str(message)`。如果某条警告被警告过滤器 改成了错误，本函数将触发一条异常。参数 *stacklevel* 可供 Python 包装函数使用，比如:

```
def deprecated_api(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

这会让警告指向 `deprecated_api` 的调用者，而不是 `deprecated_api` 本身的来源（因为后者会破坏警告消息的目的）。

skip_file_prefixes 关键字参数可被用来指明在栈层级计数时哪些栈帧要被忽略。当常数 *stacklevel* 不能适应所有调用路径或在其他情况下难以维护如果你希望警告总是在一个包以外的调用位置上出现这将会很有用处。如果提供，则它必须是一个字符串元组。当提供了 prefixes 前缀时，stacklevel 会被隐式地覆盖为 `max(2, stacklevel)`。要使得一个警告被归因至当前包以外的调用方你可以这样写:

```
# example/lower.py
_warn_skips = (os.path.dirname(__file__),)

def one_way(r_luxury_yacht=None, t_wobbler_mangrove=None):
    if r_luxury_yacht:
        warnings.warn("Please migrate to t_wobbler_mangrove=.",
                      skip_file_prefixes=_warn_skips)

# example/higher.py
from . import lower

def another_way(**kw):
    lower.one_way(**kw)
```

这将使得警告同时指向 `example.lower.one_way()` 和来自 `example` 包以外的调用代码的 `package.higher.another_way()` 调用位置。

source 是发出*ResourceWarning* 的被销毁对象。

在 3.6 版本发生变更: 加入 *source* 参数。

在 3.12 版本发生变更: 增加了 *skip_file_prefixes*。

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

这是*warn()* 函数的底层接口，显式传入消息、类别、文件名和行号，以及可选的模块名和注册表（应为模块的 `__warningregistry__` 字典）。模块名称默认为去除了 `.py` 的文件名；如果未传递注册表，警告就不会被抑制。*message* 必须是个字符串，*category* 是*Warning* 的子类；或者 *message* 可为*Warning* 的实例，且 *category* 将被忽略。

module_globals 应为发出警告的代码所用的全局命名空间。（该参数用于从 zip 文件或其他非文件系统导入模块时显式源码）。

source 是发出*ResourceWarning* 的被销毁对象。

在 3.6 版本发生变更: 加入 *source* 参数。

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

将警告信息写入文件。默认的实现代码是调用 `formatwarning(message, category, filename, lineno, line)` 并将结果字符串写入 *file*，默认文件为*sys.stderr*。通过将任何可调用对象赋给 `warnings.showwarning` 可替换掉该函数。*line* 是要包含在警告信息中的一行源代码；如果未提供 *line*，*showwarning()* 将尝试读取由 *filename* 和 *lineno* 指定的行。

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*, *line=None*)

以标准方式格式化一条警告信息。将返回一个字符串，可能包含内嵌的换行符，并以换行符结束。如果未提供 *line*，*formatwarning()* 将尝试读取由 *filename* 和 *lineno* 指定的行。

warnings.**filterwarnings**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

在*警告过滤器种类* 列表中插入一条数据项。默认情况下，该数据项将被插到前面；如果 *append* 为 True，则会插到后面。这里会检查参数的类型，编译 *message* 和 *module* 正则表达式，并将他们作为一个元组插入警告过滤器的列表中。如果两者都与某种警告匹配，那么靠近列表前面的数据项就会覆盖后面的项。省略的参数默认匹配任意值。

warnings.**simplefilter**(*action*, *category=Warning*, *lineno=0*, *append=False*)

在*警告过滤器种类* 列表中插入一条简单数据项。函数参数的含义与*filterwarnings()* 相同，但不需要正则表达式，因为插入的过滤器总是匹配任何模块中的任何信息，只要类别和行号匹配即可。

warnings.**resetwarnings**()

重置警告过滤器。这将丢弃之前对*filterwarnings()* 的所有调用，包括 `-W` 命令行选项和对*simplefilter()* 的调用效果。

@warnings.**deprecated**(*msg*, ***, *category=DeprecationWarning*, *stacklevel=1*)

指明某个类、函数或重载已被弃用的装饰器。

当此装饰器被应用于某个对象时，在运行时当该对象被使用时将会发出弃用警告。*静态类型检查器* 也会生成已弃用对象的使用情况诊断报告。

用法：

```
from warnings import deprecated
from typing import overload

@deprecated("Use B instead")
class A:
    pass

@deprecated("Use g instead")
def f():
    pass

@overload
@deprecated("int support is deprecated")
def g(x: int) -> int: ...
@overload
def g(x: str) -> int: ...
```

在运行时当已弃用对象被使用时将发出由 *category* 所指明的警告。对于函数，这会在执行调用时发生；对于类，则会在实例化或创建子类时发生。如果 *category* 为 `None`，则不会在运行时发出警告。*stacklevel* 确定要在哪里发出警告。如为 `1` (默认值)，警告将在已弃用对象的直接调用方那里发出；如为更高的值，它将在栈的更高层级上发出。静态类型检查器的行为不会受到 *category* 和 *stacklevel* 参数的影响。

传给该装饰器的弃用消息保存在被装饰对象的 `__deprecated__` 属性中。如果应用于重载，该装饰器必须位于*@overload* 装饰器之后以使该属性存在于*typing.get_overloads()* 所返回的重载之中。

Added in version 3.13: 参见 **PEP 702**。

30.6.7 可用的上下文管理器

class warnings.**catch_warnings**(***, *record=False*, *module=None*, *action=None*, *category=Warning*, *lineno=0*, *append=False*)

该上下文管理器会复制警告过滤器和*showwarning()* 函数，并在退出时恢复。如果 *record* 参数是*False* (默认)，则在进入时会返回*None*。如果 *record* 为*True*，则返回一个列表，列表由自定

义*showwarning()* 函数所用对象逐步填充（该函数还会抑制 sys.stdout 的输出）。列表中每个对象的属性与*showwarning()* 的参数名称相同。

module 参数代表一个模块，当导入*warnings* 时，将被用于代替返回的模块，其过滤器将被保护。该参数主要是为了测试*warnings* 模块自身。

如果 *action* 参数不为 None，则剩余的参数会被传递给*simplefilter()* 就如同它在进入上下文时被立即调用一样。

请参阅警告过滤器 了解 *category* 和 *lineno* 形参的含义。

> **备注**
>
> *catch_warnings* 管理器的工作方式，是替换并随后恢复模块的*showwarning()* 函数和内部的过滤器种类列表。这意味着上下文管理器将会修改全局状态，因此不是线程安全的。

在 3.11 版本发生变更: 增加了 *action*, *category*, *lineno* 和 *append* 形参。

30.7 dataclasses --- 数据类

源码： Lib/dataclasses.py

这个模块提供了一个装饰器和一些函数，用于自动为用户自定义的类添加生成的特殊方法 例如 __init__() 和 __repr__()。它的初始描述见 **PEP 557**。

在这些生成的方法中使用的成员变量是使用 **PEP 526** 类型标注来定义的。例如以下代码：

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

将添加多项内容，包括如下所示的 __init__()：

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

请注意此方法会自动添加到类中：它不是在如上所示的 InventoryItem 定义中直接指定的。

Added in version 3.7.

30.7.1 模块内容

@dataclasses.**dataclass**(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*)

此函数是一个*decorator*，它被用于将生成的特殊方法 添加到类中，如下所述。

@dataclass 装饰器会检查类以找到其中的 field。field 被定义为具有类型标注 的类变量。除了下面所述的两个例外，在 @dataclass 中没有任何东西会去检查变量标注中指定的类型。

这些字段在所有生成的方法中的顺序，都是它们在类定义中出现的顺序。

@dataclass 装饰器将把各种“双下线”方法添加到类，具体如下所述。如果所添加的任何方法在类中已存在，其行为将取决于形参的值，具体如下所述。该装饰器将返回执行其调用的类而不会创建新类。

如果 @dataclass 仅被用作不带形参的简单装饰器，其行为相当于使用在此签名中记录的默认值。也就是说，这三种 @dataclass 的用法是等价的:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False,
           match_args=True, kw_only=False, slots=False, weakref_slot=False)
class C:
    ...
```

@dataclass 的形参有:

- *init*: 如为真值（默认），将生成 __init__() 方法。

  如果类已经定义了 __init__()，此形参将被忽略。

- *repr*: 如为真值（默认），将生成 __repr__() 方法。生成的 repr 字符串将带有类名及每个字符的名称和 repr，并按它们在类中定义的顺序排列。不包括被标记为从 repr 排除的字段。例如: InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)。

  如果类已经定义了 __repr__()，此形参将被忽略。

- *eq*: 如为真值（默认），将生成 __eq__() 方法。此方法将把类当作由其字段组成的元组那样按顺序进行比较。要比较的两个实例必须是相同的类型。

  如果类已经定义了 __eq__()，此形参将被忽略。

- *order*: 如为真值 (默认为 False)，将生成 __lt__(), __le__(), __gt__() 和 __ge__() 方法。这些方法将把类当作由其字段组成的元组那样按顺序进行比较。要比较的两个实例必须是相同的类型。如果 *order* 为真值且 *eq* 为假值，则会引发*ValueError*。

  如果类已经定义了 __lt__(), __le__(), __gt__() 或者 __ge__() 中的任意一个，将引发*TypeError*。

- *unsafe_hash*: 如为 False (默认值)，则会根据 *eq* 和 *frozen* 的设置情况生成 __hash__() 方法。

  __hash__() 会在对象被添加到哈希多项集如字典和集合时由内置的*hash()* 使用。具有 __hash__() 就意味着类的实例是不可变的。可变性是一个依赖于程序员的实际意图、__eq__() 是否存在及其具体行为，以及 @dataclass 装饰器中 *eq* 和 *frozen* 旗标的值的复杂特征属性。

  在默认情况下，@dataclass 不会隐式地添加 __hash__() 方法，除非这样做是安全的。它也没会添加或更改现有的显式定义的 __hash__() 方法。设置类属性 __hash__ = None 对 Python 具有特定含义，如 __hash__() 文档中所述。

  如果 __hash__() 没有被显式定义，或者它被设为 None，则 @dataclass 可能会添加一个隐式 __hash__() 方法。虽然并不推荐，但你可以用 unsafe_hash=True 来强制让 @dataclass 创建一个 __hash__() 方法。如果你的类在逻辑上不可变但却仍然可被修改那么可能就是这种情况一。这是一个特殊用例并且应当被小心地处理。

  以下是针对隐式创建 __hash__() 方法的规则。请注意你的数据类中不能既有显式的 __hash__() 方法又设置 unsafe_hash=True；这将导致*TypeError*。

  如果 *eq* 和 *frozen* 均为真值，则默认 @dataclass 将为你生成 __hash__() 方法。如果 *eq* 为真值而 *frozen* 为假值，则:meth:*!__hash__* 将被设为 None，即将其标记为不可哈希（因为它属于可变对象）。如果 *eq* 为假值，则 __hash__() 将保持不变，这意味着将使用超类的 __hash__() 方法（如果超类是*object*，这意味着它将回退为基于 id 的哈希）。

- *frozen*: 如为真值 (默认为 `False`)，则对字段赋值将引发异常。这模拟了只读的冻结实例。如果在类中定义了 `__setattr__()` 或 `__delattr__()`，则将引发*TypeError*。参见下文的讨论。
- *match_args*: 如为真值 (默认为 `True`)，则将根据传给生成的 `__init__()` 方法的非关键字形参列表来创建 the `__match_args__` 元组（即使没有生成 `__init__()`，见上文）。如为假值，或者如果 `__match_args__` 已在类中定义，则不会生成 `__match_args__`。

  Added in version 3.10.

- *kw_only*: 如为真值 (默认值为 `False`)，则所有字段都将被标记为仅限关键字的。如果一个字段被标记为仅限关键字的，则唯一的影响是由仅限关键字的字段生成的 `__init__()` 形参在 `__init__()` 被调用时必须以关键字形式来指定。详情参见*parameter* 术语表条目。另请参阅*KW_ONLY* 一节。

  仅限关键字字段不会被包括在 `__match_args__` 中。

  Added in version 3.10.

- *slots*: 如为真值 (默认为 `False`)，则将生成 `__slots__` 属性并返回一个新类而非原本的类。如果 `__slots__` 已在类中定义，则会引发*TypeError*。

  警告

  在使用 `slots=True` 的数据类中调用无参数的*super()* 将导致引发以下异常: `TypeError: super(type, obj): obj must be an instance or subtype of type`。可用的绕过方式是调用带两个参数的*super()*。请参阅 gh-90562 了解完整细节。

  警告

  在使用 `slots=True` 时向基类 `__init_subclass__()` 传入形参将导致*TypeError*。应使用不带参数的 `__init_subclass__` 或使用默认值的绕过方式。请参阅 gh-91126 了解完整细节。

  Added in version 3.10.

  在 3.11 版本发生变更: 如果某个字段名称已经包括在基类的 `__slots__` 中，它将不会被包括在生成的 `__slots__` 中以防止 重写它们。因此，请不要使用 `__slots__` 来获取数据类的字段名称。而应改用*fields()*。为了能够确定所继承的槽位，基类 `__slots__` 可以是任意可迭代对象，但是 不可以是迭代器。an iterator.

- *weakref_slot*: 如为真值 (默认为 `False`)，则添加一个名为"__weakref__" 的槽位，这是使得一个实例可以弱引用 所必需的。指定 `weakref_slot=True` 而不同时指定 `slots=True` 将会导致错误。

  Added in version 3.11.

可以用普通的 Python 语法为各个 `field` 指定默认值:

```
@dataclass
class C:
    a: int       # 'a' 没有默认值
    b: int = 0   # 为 'b' 赋默认值
```

在这个例子中，`a` 和 `b` 都将被包括在所添加的 `__init__()` 方法中，该方法将被定义为:

```
def __init__(self, a: int, b: int = 0):
```

如果在具有默认值的字段之后存在没有默认值的字段，将会引发*TypeError*。无论此情况是发生在单个类中还是作为类继承的结果，都是如此。

dataclasses.**field**(*, *default=MISSING*, *default_factory=MISSING*, *init=True*, *repr=True*, *hash=None*, *compare=True*, *metadata=None*, *kw_only=MISSING*)

对于常见和简单的用例，不需要其他的功能。但是，有些数据类的特性需要额外的每字段信息。为了满足这种对额外信息的需求，你可以通过调用所提供的 `field()` 函数来替换默认的字段值。例如:

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

如上所示，*MISSING* 值是一个哨兵对象，用于检测一些形参是否由用户提供。使用它是因为 `None` 对于一些形参来说是有效的用户值。任何代码都不应该直接使用*MISSING* 值。

传给 `field()` 的形参有:

- *default*: 如果提供，这将为该字段的默认值。设置此形参是因为 `field()` 调用本身会替换通常的默认值所在位置。
- *default_factory*: 如果提供，它必须是一个零参数的可调用对象，它将在该字段需要一个默认值时被调用。在其他目的以外，它还可被用于指定具有可变默认值的字段，如下所述。同时指定 *default* 和 *default_factory* 将会导致错误。
- *init*: 如为真值（默认），则该字段将作为一个形参被包括在生成的 `__init__()` 方法中。
- *repr*: 如为真值（默认值），则该字段将被包括在生成的 `__repr__()` 方法所返回的字符串中。
- *hash*: 这可以是一个布尔值或 `None`。如为真值，则此字段将被包括在所生成的 `__hash__()` 方法中。如为假值，则此字段将被排除在所生成的 `__hash__()` 之外。如为 `None` (默认值)，则使用 *compare* 的值：这通常是预期的行为，因为一个字段如果被用于比较那么就应当被包括在哈希运算中。不建议将该值设为 `None` 以外的任何值。

  设置 `hash=False` 但 `compare=True` 的一个合理情况是，一个计算哈希值的代价很高的字段是检验等价性需要的，且还有其他字段可以用于计算类型的哈希值。可以从哈希值中排除该字段，但仍令它用于比较。
- *compare*: 如为真值（默认），则该字段将被包括在生成的相等和比较方法中 (`__eq__()`, `__gt__()` 等等)。
- *metadata*: 这可以是一个映射或为 `None`。`None` 将被当作空字典来处理。这个值将被包装在*MappingProxyType()* 以便其为只读，并暴露在*Field* 对象上。它完全不被数据类所使用，并且是作为第三方扩展机制提供的。多个第三方可以各自拥有其本身的键，以用作元数据的命名空间。
- *kw_only*: 如为真值，则该字段将被标记为仅限关键字的。这将在计算所生成的 `__init__()` 方法的形参时被使用。

  仅限关键字字段也不会被包括在 `__match_args__` 中。

  Added in version 3.10.

如果通过对 `field()` 的调用来指定字段的默认值，那么该字段对应的类属性将被替换为指定的 *default* 值。如果没有提供 *default*，那么该类属性将被删除。其意图是在*@dataclass* 装饰器运行之后，该类属性将包含所有字段的默认值，就像直接指定了默认值本身一样。例如，在执行以下代码之后:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

类属性 C.z 将为 10，类属性 C.t 将为 20，类属性 C.x 和 C.y 将不被设置。

class dataclasses.**Field**

Field 对象描述每个已定义的字段。这些对象是在内部创建的，并会由*fields()* 模块块方法返回（见下文）。用户绝不应直接实例化 Field 对象。已写入文档的属性如下：

- name: 字段的名称。
- type: 字段的类型。
- default, default_factory, init, repr, hash, compare, metadata 和 kw_only 具有与*field()* 函数中对应参数相同的含义和值。

可能存在其他属性，但它们是私有的。用户不应检查或依赖于这些属性。

class dataclasses.**InitVar**

InitVar[T] 类型标注用于描述仅限初始化 变量。使用 InitVar 标注的字段将被视作伪字段，因此既不会被*fields()* 函数返回也不会在除了作为传给 __init__() 的和可选的*__post_init__()* 的形参添加之外以任何方式被使用。

dataclasses.**fields**(*class_or_instance*)

返回一个能描述此数据类所包含的字段的元组，元组的每一项都是*Field* 对象。接受数据类或数据类的实例。如果没有传递一个数据类或实例将引发*TypeError*。不返回 ClassVar 或 InitVar 等伪字段。

dataclasses.**asdict**(*obj*, *, *dict_factory=dict*)

将数据类 *obj* 转换为一个字典 (使用工厂函数 *dict_factory*)。每个数据类会被转换为以 name: value 键值对来存储其字段的字典。数据类、字典、列表和元组会被递归地处理。其他对象会通过*copy.deepcopy()* 来拷贝。

在嵌套的数据类上使用 asdict() 的例子:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: list[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

要创建一个浅拷贝，可以使用以下的变通方法：

```
{field.name: getattr(obj, field.name) for field in fields(obj)}
```

如果 *obj* 不是一个数据类实例则 asdict() 将引发*TypeError* 。

dataclasses.**astuple**(*obj*, *, *tuple_factory=tuple*)

将数据类 *obj* 转换为元组 (使用工厂函数 *tuple_factory*)。每个数据类将被转换为由其字段值组成的元组。数据类、字典、列表和元组会被递归地处理。其他对象会通过*copy.deepcopy()* 来拷贝。

继续前一个例子：

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

要创建一个浅拷贝，可以使用以下的变通方法：

```
tuple(getattr(obj, field.name) for field in dataclasses.fields(obj))
```

如果 *obj* 不是一个数据类实例则 `astuple()` 将引发*TypeError*。

dataclasses.**make_dataclass**(*cls_name*, *fields*, *, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*, *module=None*)

新建一个名为 *cls_name* 的数据类，其字段在 *fields* 中定义，其基类在 *bases* 中给出，并使用在 *namespace* 中给定的命名空间来初始化。*fields* 是一个可迭代对象，其中每个元素均为 `name`，`(name, type)` 或 `(name, type, Field)` 的形式。如果只提供了 `name`，则使用*typing.Any* 作为 `type`。*init*, *repr*, *eq*, *order*, *unsafe_hash*, *frozen*, *match_args*, *kw_only*, *slots* 和 *weakref_slot* 值的含义与*@dataclass* 中的同名参数一致。

如果定义了 *module*，则该数据类的 `__module__` 属性将被设为该值。在默认情况下，它将被设为调用方的模块名。

此函数不是必需的，因为任何用于创建带有 `__annotations__` 的新类的 Python 机制都可以进一步用*@dataclass* 函数将创建的类转换为数据类。提供此函数是为了方便。例如：

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x + 1})
```

等价于：

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

dataclasses.**replace**(*obj*, */*, ***changes*)

创建一个与 *obj* 类型相同的新对象，将字段替换为 *changes* 的值。如果 *obj* 不是数据类，则会引发*TypeError*。如果 *changes* 中的键不是给定数据类的字段名，则会引发*TypeError*。

新返回的对象是通过调用数据类的 `__init__()` 方法来创建的。这确保了如果存在*__post_init__()*，则它也会被调用。

如果存在任何没有默认值的仅初始化变量，那么必须在调用 `replace()` 时指定它们的值，以便它们可以被传递给 `__init__()` 和*__post_init__()*。

如果 *changes* 包含被任何定义为 defined as having `init=False` 的字段都会导致错误。在此情况下将引发*ValueError*。

需要预先注意 `init=False` 字段在对 `replace()` 的调用期间的行为。如果它们会被初始化，它们就不会从源对象拷贝，而是在*__post_init__()* 中初始化。通常预期 `init=False` 字段将很少能被正确地使用。如果要使用它们，那么更明智的做法是使用另外的类构造器，或者自定义的 `replace()` (或类似名称) 方法来处理实例的拷贝。

数据类支持也被泛型函数*copy.replace()* 所支持。

dataclasses.**is_dataclass**(*obj*)

如果其形参是一个 dataclass（包括 dataclass 的子类）或其实例则返回 True，否则返回 False。

如果你需要知道一个类是否是一个数据类的实例（而不是一个数据类本身），那么再添加一个 not isinstance(obj, type) 检查：

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

dataclasses.**MISSING**

一个指明“没有提供 default 或 default_factory”的监视值。

dataclasses.**KW_ONLY**

一个用途类型标的监视值。任何在伪字段之后的类型为 KW_ONLY 的字段会被标记为仅限关键字的字段。请注意在其他情况下 KW_ONLY 类型的伪字段会被完全忽略。这包括此类字段的名称。根据惯例，名称 _ 会被用作 KW_ONLY 字段。仅限关键字字段指明当类被实例化时 __init__() 形参必须以关键字形式来指定。

在这个例子中，字段 y 和 z 将被标记为仅限关键字字段：

```
@dataclass
class Point:
    x: float
    _: KW_ONLY
    y: float
    z: float

p = Point(0, y=1.5, z=2.0)
```

在单个数据类中，指定一个以上 KW_ONLY 类型的字段将导致错误。

Added in version 3.10.

exception dataclasses.**FrozenInstanceError**

在定义时设置了 frozen=True 的类上调用隐式定义的 __setattr__() 或 __delattr__() 时引发。这是*AttributeError* 的一个子类。

30.7.2 初始化后处理

dataclasses.**__post_init__**()

当在类上定义时，它将被所生成的 __init__() 调用，通常是以 self.__post_init__() 的形式。但是，如果定义了任何 InitVar 字段，它们也将按照它们在类中定义的顺序被传递给 __post_init__()。如果没有生成 __init__() 方法，那么 __post_init__() 将不会被自动调用。

在其他用途中，这允许初始化依赖于一个或多个其他字段的字段值。例如：

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

由*@dataclass* 生成的 __init__() 方法不会调用基类的 __init__() 方法。如果基类有必须被调用的 __init__() 方法，通常是在*__post_init__()* 方法中调用此方法：

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
```

(续下页)

(接上页)

```
        self.width = width

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

但是，请注意一般来说数据类生成的 `__init__()` 方法不需要被调用，因为派生的数据类将负责初始化任何本身为数据类的基类的所有字段。

请参阅下面有关仅初始化变量的小节来了解如何将形参传递给 `__post_init__()`。另请参阅关于*replace()* 如何处理 `init=False` 字段的警告。

30.7.3 类变量

少数几个*@dataclass* 会实际检查字段类型的地方之一是确定字段是否为如 **PEP 526** 所定义的类变量。它通过检查字段的类型是否为*typing.ClassVar* 来实现这一点。如果一个字段是 `ClassVar`，它将被排除在考虑范围之外并被数据类机制所忽略。这样的 `ClassVar` 伪字段将不会被模块层级的*fields()* 函数返回。

30.7.4 仅初始化变量

另一个*@dataclass* 会检查类型标注的地方是为了确定一个字段是否为仅限初始化的变量。它通过检查字段的类型是否为*InitVar* 类型来做到这一点。如果一个字段是*InitVar*，它会被当作是一个名为仅限初始化字段的伪字段。因为它不是一个真正的字段，所以它不会被模块层级的*fields()* 函数返回。仅限初始化字段会作为形参被添加到所生成的 `__init__()` 方法中，并会被传递给可选的*__post_init__()* 方法。在其他情况下它们将不会被类所使用。

例如，假设在创建类时没有为某个字段提供值，初始化时将从数据库中取值:

```
@dataclass
class C:
    i: int
    j: int | None = None
    database: InitVar[DatabaseType | None] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

在这种情况下，*fields()* 将返回*Field* 作为 `i` 和 `j`，但不包括 `database`。

30.7.5 冻结的实例

创建真正不可变的 Python 对象是不可能的。但是，你可以通过将 `frozen=True` 传递给*@dataclass* 装饰器来模拟出不可变性。在这种情况下，数据类将向类添加 `__setattr__()` 和 `__delattr__()` 方法。当被唤起时这些方法将会引发*FrozenInstanceError*。

在使用 `frozen=True` 时会有微小的性能损失: `__init__()` 不能使用简单赋值来初始化字段，而必须使用 `object.__setattr__()`。

30.7.6 继承

当数据类由 *@dataclass* 装饰器创建时，它会按反向 MRO 顺序（也就是说，从 *object* 开始）查看它的所有基类，并将找到的每个数据类的字段添加到一个有序映射中。所有生成的方法都将使用这个有序映射。字段会遵守它们被插入的顺序，因此派生类会重写基类。一个例子:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

最终的字段列表依次是 x, y, z。最终的 x 类型是 *int*，正如类 C 中所指定的。

为 C 生成的 __init__() 方法看起来像是这样:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

30.7.7 __init__() 中仅限关键字形参的重新排序

在计算出 __init__() 所需要的形参之后，任何仅限关键字形参会被移至所有常规（非仅限关键字）形参的后面。这是 Python 中实现仅限关键字形参所要求的：它们必须位于非仅限关键字形参之后。

在这个例子中，Base.y, Base.w 和 D.t 是仅限关键字字段，而 Base.x 和 D.z 是常规字段:

```
@dataclass
class Base:
    x: Any = 15.0
    _: KW_ONLY
    y: int = 0
    w: int = 1

@dataclass
class D(Base):
    z: int = 10
    t: int = field(kw_only=True, default=0)
```

为 D 生成的 __init__() 方法看起来像是这样:

```
def __init__(self, x: Any = 15.0, z: int = 10, *, y: int = 0, w: int = 1, t: int = 0):
```

请注意形参原来在字段列表中出现的位置已被重新排序：前面是来自常规字段的形参而后面是来自仅限关键字字段的形参。

仅限关键字形参的相对顺序会在重新排序的 __init__() 列表中保持不变。

30.7.8 默认工厂函数

如果一个 *field()* 指定了 *default_factory*，它将在该字段需要默认值时不带参数地被调用。例如，要创建一个列表的新实例，则使用:

```
mylist: list = field(default_factory=list)
```

如果一个字段被排除在 __init__() 之外 (使用 init=False) 并且该字段还指定了 *default_factory*，则默认的工厂函数将总是会从生成的 __init__() 函数中被调用。发生这种情况是因为没有其他方式能为字段提供初始值。

30.7.9 可变的默认值

Python 在类属性中存储默认成员变量值。思考这个例子，不使用数据类:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

请注意类 C 的两个实例将共享同一个类变量 x，正如预期的那样。

使用数据类，如果此代码有效:

```
@dataclass
class D:
    x: list = []      # 此代码将引发 ValueError
    def add(self, element):
        self.x.append(element)
```

它生成的代码类似于:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x.append(element)

assert D().x is D().x
```

这具有与使用 C 类的原始示例相同的问题。也就是说，当创建类实例时如果 D 类的两个实例没有为 x 指定值则将共享同一个 x 的副本。因为数据类只是使用普通的 Python 类创建方式所以它们也会共享此行为。数据类没有任何通用方式来检测这种情况。相反地，*@dataclass* 装饰器在检测到不可哈希的默认形参时将会引发*ValueError*。这一行为假定如果一个值是不可哈希的，则它就是可变对象。这是一个部分解决方案，但它确实能防止许多常见错误。

使用默认工厂函数是一种创建可变类型新实例的方法，并将其作为字段的默认值:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

在 3.11 版本发生变更: 现在不再是寻找并阻止使用类型为*list*, *dict* 或*set* 的对象，而是不允许将不可哈希的对象用作默认值。就是不可哈希性被作为不可变性的近似物了。

30.7.10 描述器类型的字段

当字段被 描述器对象赋值为默认值时会遵循以下行为:

- 传递给数据类的 __init__() 方法的字段值会被传递给描述器的 __set__() 方法而不会覆盖描述器对象。
- 类似地，当获取或设置字段值时，将调用描述器的 __get__() 或 __set__() 方法而不是返回或重写描述器对象。

- 为了确定一个字段是否包含默认值，*@dataclass* 会使用类访问形式调用描述器的 `__get__()` 方法: `descriptor.__get__(obj=None, type=cls)`。如果在此情况下描述器返回了一个值，它将被用作字段的默认值。另一方面，如果在此情况下描述器引发了*AttributeError*，则不会为字段提供默认值。

```
class IntConversionDescriptor:
    def __init__(self, *, default):
        self._default = default

    def __set_name__(self, owner, name):
        self._name = "_" + name

    def __get__(self, obj, type):
        if obj is None:
            return self._default

        return getattr(obj, self._name, self._default)

    def __set__(self, obj, value):
        setattr(obj, self._name, int(value))

@dataclass
class InventoryItem:
    quantity_on_hand: IntConversionDescriptor = IntConversionDescriptor(default=100)

i = InventoryItem()
print(i.quantity_on_hand)   # 100
i.quantity_on_hand = 2.5    # 调用 __set__ 并传入 2.5
print(i.quantity_on_hand)   # 2
```

若一个字段的类型是描述器，但其默认值并不是描述器对象，那么该字段只会像普通的字段一样工作。

30.8 contextlib --- 为 with 语句上下文提供的工具

源代码 Lib/contextlib.py

此模块为涉及 `with` 语句的常见任务提供了实用的工具。更多信息请参见上下文管理器类型 和 context-managers。

30.8.1 工具

提供的函数和类：

class contextlib.**AbstractContextManager**

一个为实现了 `object.__enter__()` 与 `object.__exit__()` 的类提供的*abstract base class*。为 `object.__enter__()` 提供的一个默认实现是返回 `self` 而 `object.__exit__()` 是一个默认返回 `None` 的抽象方法。参见上下文管理器类型 的定义。

Added in version 3.6.

class contextlib.**AbstractAsyncContextManager**

一个为实现了 `object.__aenter__()` 与 `object.__aexit__()` 的类提供的*abstract base class*。为 `object.__aenter__()` 提供的一个默认实现是返回 `self` 而 `object.__aexit__()` 是一个默认返回 `None` 的抽象方法。参见 async-context-managers 的定义。

Added in version 3.7.

@contextlib.**contextmanager**

此函数是一个*decorator*，它可被用来定义一个支持 `with` 语句上下文管理器的工厂函数，而无需创建一个类或单独的 `__enter__()` 和 `__exit__()` 方法。

尽管许多对象都原生支持在 with 语句中使用，但有时需要被管理的资源本身并不是上下文管理器，并且没有实现 `close()` 方法以配合 `contextlib.closing` 使用。

下面是一个抽象的示例，展示如何确保正确的资源管理:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # 获取资源的代码，例如：
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # 释放资源的代码，例如：
        release_resource(resource)
```

随后可以这样使用此函数:

```
>>> with managed_resource(timeout=3600) as resource:
...     # 资源将在此代码块的末尾被释放，
...     # 即使代码块中的代码引发了异常
```

被装饰的函数在被调用时，必须返回一个*generator* 迭代器。这个迭代器必须只 yield 一个值出来，这个值会被用在 `with` 语句中，绑定到 `as` 后面的变量，如果给定了的话。

当生成器发生 yield 时，嵌套在 `with` 语句中的语句体会被执行。语句体执行完毕离开之后，该生成器将被恢复执行。如果在该语句体中发生了未处理的异常，则该异常会在生成器发生 yield 时重新被引发。因此，你可以使用 `try...except...finally` 语句来捕获该异常（如果有的话），或确保进行了一些清理。如果仅出于记录日志或执行某些操作（而非完全抑制异常）的目的捕获了异常，生成器必须重新引发该异常。否则生成器的上下文管理器将向 `with` 语句指示该异常已经被处理，程序将立即在 `with` 语句之后恢复并继续执行。

`contextmanager()` 使用*`ContextDecorator`* 因此它创建的上下文管理器不仅可以用在 `with` 语句中，还可以用作一个装饰器。当它用作一个装饰器时，每一次函数调用时都会隐式创建一个新的生成器实例（这使得*`contextmanager()`* 创建的上下文管理器满足了支持多次调用以用作装饰器的需求，而非“一次性”的上下文管理器）。

在 3.2 版本发生变更: *`ContextDecorator`* 的使用。

`@contextlib.`**`asynccontextmanager`**

与*`contextmanager()`* 类似，但创建的是 asynchronous context manager 。

该函数是一个*decorator*，它可被用来定义一个使用 `async with` 语句的异步上下文管理器的工厂函数，而不需要创建一个类或单独的 `__aenter__()` 和 `__aexit__()` 方法。它必须应用在*asynchronous generator* 函数上。

一个简单的示例：

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

Added in version 3.7.

使用*asynccontextmanager()* 定义的上下文管理器可以用作装饰器，也可以在 `async with` 语句中使用。

```
import time
from contextlib import asynccontextmanager

@asynccontextmanager
async def timeit():
    now = time.monotonic()
    try:
        yield
    finally:
        print(f'it took {time.monotonic() - now}s to run')

@timeit()
async def main():
    # ... 异步代码 ...
```

用作装饰器时，每次函数调用都会隐式创建一个新的生成器实例。这使得由*asynccontextmanager()* 创建的“一次性”上下文管理器能够满足作为装饰器所需要的支持多次调用的要求。

在 3.10 版本发生变更: 使用*asynccontextmanager()* 创建的异步上下文管理器可以用作装饰器。

`contextlib.`**`closing`**(*thing*)

返回一个在语句块执行完成时关闭 *things* 的上下文管理器。这基本上等价于：

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

并允许你编写这样的代码：

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

而无需显式地关闭 `page` 。即使发生错误，在退出 `with` 语句块时，`page.close()` 也同样会被调用。

备注

大多数管理资源的类型都支持*context manager* 协议，它在离开 `with` 语句时将关闭 *thing*。因此，`closing()` 对不支持上下文管理器的第三方类型是最有用的。这个例子纯粹是为了说明问题，因为*urlopen()* 通常都会在上下文管理器中使用。

`contextlib.`**`aclosing`**(*thing*)

返回一个在语句块执行完成时调用 `aclose()` 方法来关闭 *things* 的异步上下文管理器。这基本上等价于：

```
from contextlib import asynccontextmanager

@asynccontextmanager
```

(续下页)

(接上页)

```
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

重要的是，`aclosing()` 支持在异步生成器因遭遇 `break` 或异常而提前退出时对其执行确定性的清理。例如:

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

此模块将确保生成器的异步退出代码在与其迭代相同的上下文中执行（这样异常和上下文变量将能按预期工作，并且退出代码不会在其所依赖的某些任务的生命期结束后继续运行）。

Added in version 3.10.

contextlib.**nullcontext**(*enter_result=None*)

返回一个从 `__enter__` 返回 *enter_result* 的上下文管理器，除此之外不执行任何操作。它旨在用于可选上下文管理器的一种替代，例如:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # 使用 suppress 来忽略所有异常。
        cm = contextlib.suppress(Exception)
    else:
        # 不忽略任何异常，cm 将没有影响。
        cm = contextlib.nullcontext()
    with cm:
        # 执行某些操作
```

一个使用 *enter_result* 的例子:

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # 调用方要负责关闭文件
        cm = nullcontext(file_or_path)

    with cm as file:
        # 在文件上执行处理
```

它也可以替代 asynchronous context managers：

```
async def send_http(session=None):
    if not session:
        # 如果没有 http 会话，则使用 aiohttp 创建它
        cm = aiohttp.ClientSession()
    else:
        # 调用方要负责关闭会话
        cm = nullcontext(session)

    async with cm as session:
        # 使用会话发送 http 请求
```

Added in version 3.7.

在 3.10 版本发生变更: 增加了对*asynchronous context manager* 的支持。

contextlib.**suppress**(**exceptions*)

返回一个当指定的异常在 `with` 语句体中发生时会屏蔽它们然后从 `with` 语句结束后的第一条语言开始恢复执行的上下文管理器。

与完全抑制异常的任何其他机制一样，该上下文管理器应当只用来抑制非常具体的错误，并确保该场景下静默地继续执行程序是通用的正确做法。

例如:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

这段代码等价于:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

该上下文管理器是*reentrant* 。

如果 `with` 语句块内的代码引发了`BaseExceptionGroup`，将从分组中移除被抑制的异常。该分组中任何未被抑制的异常会在一个使用原分组的`derive()` 方法创建的新分组中被重新引发。

Added in version 3.4.

在 3.12 版本发生变更: `suppress` 现在支持抑制作为`BaseExceptionGroup` 的组成部分被引发的异常。

contextlib.**redirect_stdout**(*new_target*)

用于将`sys.stdout` 临时重定向到一个文件或类文件对象的上下文管理器。

该工具给已有的将输出硬编码写到 stdout 的函数或类提供了额外的灵活性。

例如，`help()` 的输出通常会被发送到 *sys.stdout*。你可以通过将输出重定向到一个`io.StringIO` 对象来将该输出捕获到字符串。替换的流是由 `__enter__` 返回的因此可以被用作 `with` 语句的目标:

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

如果要把`help()` 的输出写到磁盘上的一个文件，重定向该输出到一个常规文件:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

如果要把`help()` 的输出写到 *sys.stderr*：

```
with redirect_stdout(sys.stderr):
    help(pow)
```

需要注意的点在于，*sys.stdout* 的全局副作用意味着此上下文管理器不适合在库代码和大多数多线程应用程序中使用。它对子进程的输出没有影响。不过对于许多工具脚本而言，它仍然是一个有用的方法。

该上下文管理器是*reentrant* 。

Added in version 3.4.

contextlib.**redirect_stderr**(*new_target*)

与*redirect_stdout()* 类似，不过是将*sys.stderr* 重定向到一个文件或类文件对象。

该上下文管理器是*reentrant* 。

Added in version 3.5.

contextlib.**chdir**(*path*)

用于改变当前工作目录的非并行安全的上下文管理器。由于这会改变一个全局状态，即工作目录，因此它不适合在大多数线程或异步上下文中使用。它也不适合大多数非线性的代码执行，比如生成器，在此类代码中程序的执行会被临时性挂起 -- 除非明确希望如此，当该上下文管理器被激活时你不应执行 yield 语句。

这是一个对*chdir()* 的简单包装器，它会在进入时改变当前工作目录并在退出时恢复。

该上下文管理器是*reentrant* 。

Added in version 3.11.

class contextlib.**ContextDecorator**

一个使上下文管理器能用作装饰器的基类。

与往常一样，继承自 ContextDecorator 的上下文管理器必须实现 __enter__ 与 __exit__ 。即使用作装饰器，__exit__ 依旧会保持可能的异常处理。

ContextDecorator 被用在*contextmanager()* 中，因此你自然获得了这项功能。

ContextDecorator 的示例:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```

随后可以这样使用该类:

```
>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

这个改动只是针对如下形式的一个语法糖:

```
def f():
    with cm():
        # 执行操作
```

ContextDecorator 使得你可以这样改写:

```
@cm()
def f():
    # 执行操作
```

这能清楚地表明，cm 作用于整个函数，而不仅仅是函数的一部分（同时也能保持不错的缩进层级）。

现有的上下文管理器即使已经有基类，也可以使用 ContextDecorator 作为混合类进行扩展:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

备注

由于被装饰的函数必须能够被多次调用，因此对应的上下文管理器必须支持在多个 with 语句中使用。如果不是这样，则应当使用原来的具有显式 with 语句的形式使用该上下文管理器。

Added in version 3.2.

class contextlib.**AsyncContextDecorator**

和*ContextDecorator* 类似，但是用于异步函数。

AsyncContextDecorator 的示例:

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False
```

随后可以这样使用该类:

```
>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
```

(续下页)

(接上页)

```
...     async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing
```

Added in version 3.10.

class contextlib.**ExitStack**

该上下文管理器的设计目标是使得在编码中组合其他上下文管理器和清理函数更加容易，尤其是那些可选的或由输入数据驱动的上下文管理器。

例如，通过一个如下的 with 语句可以很容易处理一组文件：

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # 所有已打开的文件都将在 with 语句结束时
    # 自动被关闭，即使此后打开列表中文件的
    # 尝试引发了异常
```

`__enter__()` 方法返回*ExitStack* 的实例，并且不会执行额外的操作。

每个实例维护一个注册了一组回调的栈，这些回调在实例关闭时以相反的顺序被调用（显式或隐式地在 `with` 语句的末尾）。请注意，当一个栈实例被垃圾回收时，这些回调将 *不会*被隐式调用。

通过使用这个基于栈的模型，那些通过 `__init__` 方法获取资源的上下文管理器（如文件对象）能够被正确处理。

由于注册的回调函数是按照与注册相反的顺序调用的，因此最终的行为就像多个嵌套的 `with` 语句用在这些注册的回调函数上。这个行为甚至扩展到了异常处理：如果内部的回调函数抑制或替换了异常，则外部回调收到的参数是基于该更新后的状态得到的。

这是一个相对底层的 API，它负责正确处理栈里回调退出时依次展开的细节。它为相对高层的上下文管理器提供了一个合适的基础，使得它能根据应用程序的需求使用特定方式操作栈。

Added in version 3.3.

enter_context(*cm*)

进入一个新的上下文管理器并将其 `__exit__()` 方法添加到回调栈中。返回值是该上下文管理器自己的 `__enter__()` 方法的输出结果。

这些上下文管理器可能会屏蔽异常就如当它们作为 `with` 语句的一部分直接使用时通常表现的行为一样。

在 3.11 版本发生变更: 如果 *cm* 不是上下文管理器则会引发*TypeError* 而不是*AttributeError*。

push(*exit*)

将一个上下文管理器的 `__exit__()` 方法添加到回调栈。

由于 `__enter__` 没有被唤起，此方法可以被用来通过一个上下文管理器自己的 `__exit__()` 方法覆盖一部分 `__enter__()` 的实现。

如果传入了一个不是上下文管理器的对象，此方法将假定它是一个具有与上下文管理器的 `__exit__()` 方法相同的签名的回调并会直接将其添加到回调栈中。

通过返回真值，这些回调可以通过与上下文管理器的 `__exit__()` 方法相同的方式抑制异常。

传入的对象将被该函数返回，允许此方法作为函数装饰器使用。

callback(*callback*, */*, **args*, ***kwds*)

接受一个任意的回调函数和参数并将其添加到回调栈。

与其他方法不同，通过此方式添加的回调无法屏蔽异常（因为异常的细节不会被传递给它们）。

传入的回调将被该函数返回，允许此方法作为函数装饰器使用。

pop_all()

将回调栈传输到一个新的*ExitStack* 实例并返回它。此操作不会唤起任何回调——作为替代，现在当新栈被关闭时它们将（显式地或是在一条 with 语句结束时隐式地）被唤起。

例如，一组文件可以像下面这样以“一个都不能少”的操作方式被打开:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # 持有 close 方法，但暂时不调用它。
    close_files = stack.pop_all().close
    # 如果任何文件打开失败，则所有之前打开的文件
    # 将自动被关闭。 如果所有文件都被成功地打开，
    # 即使在 with 语句结束后它们仍将保持打开状态。
    # 随后可以显式唤起 close_files() 来全部关闭它们。
```

close()

立即展开回调栈，按注册时的相反顺序唤起其中的回调。对于任何已注册的上下文管理器和退出回调，传入的参数将表明没有发生异常。

class contextlib.**AsyncExitStack**

一个 异步上下文管理器，类似于*ExitStack*，它支持组合同步和异步上下文管理器，并拥有针对清理逻辑的协程。

close() 方法未实现，必须改用*aclose()*。

async enter_async_context(*cm*)

类似于*ExitStack.enter_context()* 但是需要一个异步上下文管理器。

在 3.11 版本发生变更: 如果 *cm* 不是异步上下文管理器则会引发*TypeError* 而不是*AttributeError*。

push_async_exit(*exit*)

类似于*ExitStack.push()* 但是需要一个异步上下文管理器或协程函数。

push_async_callback(*callback*, */*, **args*, ***kwds*)

类似于*ExitStack.callback()* 但是需要一个协程函数。

async aclose()

类似于*ExitStack.close()* 但是能正确处理可等待对象。

从*asynccontextmanager()* 的例子继续:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # 所有已打开连接都将在 async with 语句结束时
    # 自动被关闭，即使此后打开列表中连接的尝试
    # 引发了异常。
```

Added in version 3.7.

30.8.2 例子和配方

本节描述了一些用于有效利用*contextlib* 所提供的工具的示例和步骤说明。

支持可变数量的上下文管理器

ExitStack 的主要应用场景已在该类的文档中给出：在单个 with 语句中支持可变数量的上下文管理器和其他清理操作。这个可变性可以来自通过用户输入驱动所需的上下文管理器数量（例如打开用户所指定的文件集），或者来自将某些上下文管理器作为可选项。

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # 执行使用了所获取资源的操作
```

如上所示，*ExitStack* 还让使用 with 语句来管理任意非原生支持上下文管理协议的资源变得相当容易。

捕获 __enter__ 方法产生的异常

有时人们会想要从 __enter__ 方法的实现中捕获异常，而 不会无意中捕获来自 with 语句体或上下文管理器的 __exit__ 方法的异常。通过使用*ExitStack* 可以将上下文管理协议中的步骤稍微分开以允许这样做:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # 处理 __enter__ 异常
else:
    with stack:
        # 处理正常情况
```

实际上需要这样做很可能表明下层的 API 应该提供一个直接的资源管理接口以便与 try/except/finally 语句配合使用，但并不是所有的 API 在这方面都设计得很好。当上下文管理器是所提供的唯一资源管理 API 时，则*ExitStack* 可以让处理各种无法在 with 语句中直接处理的情况变得更为容易。

在一个 __enter__ 方法的实现中进行清理

正如*ExitStack.push()* 的文档中指出的，如果在 __enter__() 实现中的后续步骤失败则此方法将可被用于清理已分配的资源。

下面是为一个上下文管理器做这件事的例子，该上下文管理器可接受资源获取和释放函数，以及可选的验证函数，并将它们映射到上下文管理协议:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # 验证检测通过且没有引发异常
            # 相应地，我们想要保留资源，并将其
            # 回传给调用方
            stack.pop_all()

    def __enter__(self):
```

(续下页)

(接上页)

```
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # 我们不需要复制任何的资源释放逻辑
        self.release_resource()
```

替换任何对 try-finally 和旗标变量的使用

一种你有时将看到的模式是 try-finally 语句附带一个旗标变量来指明 finally 子句体是否要被执行。在其最简单的形式中（它无法仅仅通过改用一条 except 子句来预先处理），看起来会是这样:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

就如任何基于 try 语句的代码一样，这可能会导致开发和审查方面的问题，因为设置代码和清理代码最终可能会被任意长的代码部分所分隔。

ExitStack 将允许选择在一条 with 语句末尾注册一个用于执行的回调的替代方式，等以后再决定是否跳过该回调的执行:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

这允许在事先显式地指明预期的清理行为，而不需要一个单独的旗标变量。

如果某个应用程序大量使用此模式，则可以通过使用一个较小的辅助类来进一步地简化它:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

如果资源清理操作尚未完善地捆绑到一个独立的函数中，则仍然可以使用*ExitStack.callback()* 的装饰器形式来提前声明资源清理:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

受装饰器协议工作方式的影响，以此方式声明的回调函数无法接受任何形参。作为替代，任何要释放的资源必须作为闭包变量来访问。

将上下文管理器作为函数装饰器使用

ContextDecorator 类允许将上下文管理器作为函数装饰器使用，而不仅在 `with` 语句块中使用。

例如，有时将函数或语句组包装在一个可以跟踪进入和退出时间的记录器中是很有用的。与其为任务同时编写函数装饰器和上下文管理器，不如继承*ContextDecorator* 在一个定义中同时提供这两种能力:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

这个类的实例既可以被用作上下文管理器:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

也可以被用作函数装饰器:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

请注意当使用上下文管理器作为函数装饰器时有一个额外的限制：没有任何办法可以访问 `__enter__()` 的返回值。如果需要该值，那么你仍然需要使用显式的 `with` 语句。

参见

PEP 343 - "with" 语句
Python `with` 语句的规范描述、背景和示例。

30.8.3 单独使用，可重用并可重进入的上下文管理器

大多数上下文管理器的编写方式意味着它们只能在一个 with 语句中被有效使用一次。这些一次性使用的上下文管理器必须在每次被使用时重新创建——试图第二次使用它们将会触发异常或是不能正确工作。

这个常见的限制意味着通常来说都建议在 with 语句开头上下文管理器被使用的位置直接创建它们（如下面所有的使用示例所显示的）。

文件是一个高效的单次使用上下文管理器的例子，因为第一个 with 语句将关闭文件，防止任何后续的使用该文件对象的 IO 操作。

使用*contextmanager()* 创建的上下文管理器也是单次使用的上下文管理器，并会在试图第二次使用它们时报告下层生成器无法执行产生操作:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

重进入上下文管理器

更复杂的上下文管理器可以“重进入”。这些上下文管理器不但可以被用于多个 with 语句中，还可以被用于已经在使用同一个上下文管理器的 with 语句 内部。

threading.RLock 是一个可重入上下文管理器的例子，*suppress()*，*redirect_stdout()* 和*chdir()* 也是。下面是一个非常简单的使用重入的示例:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

现实世界的可重入例子更可能涉及到多个函数的相互调用因此会比这个例子要复杂得多。

还要注意可重入与线程安全 不是一回事。举例来说，*redirect_stdout()* 肯定不是线程安全的，因为它会通过将*sys.stdout* 绑定到一个不同的流对系统状态做了全局性的修改。

可重用的上下文管理器

与单次使用和可重入上下文管理器都不同的还有“可重用”上下文管理器（或者，使用完全显式的表达应为“可重用，但不可重入”上下文管理器，因为可重入上下文管理器也会是可重用的）。这些上下文管理器支持多次使用，但如果特定的上下文管理器实例已经在包含它的 with 语句中被使用过则将失败（或者不能正确工作）。

threading.Lock 是一个可重用，但是不可重入的上下文管理器的例子（对于可重入锁，则有必要使用*threading.RLock* 来代替）。

另一个可重用，但不可重入的上下文管理器的例子是*ExitStack*，因为它在离开任何 with 语句时会唤起所有当前已注册的回调，不论回调是在哪里添加的:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

正如这个例子的输出所显示的，在多个 with 语句中重用一个单独的栈对象可以正确工作，但调试嵌套它们就将导致栈在最内层的 with 语句结束时被清空，这不大可能是符合期望的行为。

使用单独的*ExitStack* 实例而不是重复使用一个实例可以避免此问题:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

30.9 abc --- 抽象基类

源代码： Lib/abc.py

该模块提供了在 Python 中定义抽象基类 (ABC) 的组件，在 **PEP 3119** 中已有概述。查看 PEP 文档了解为什么需要在 Python 中增加这个模块。（也可查看 **PEP 3141** 以及 *numbers* 模块了解基于 ABC 的数字类型继承关系。）

collections 模块中有一些派生自 ABC 的实体类；当然，这些类还可以进一步被派生。此外，*collections.abc* 子模块中有一些可被用于测试一个类或实例是否提供了特定接口的 ABC，例如，它是否为*hashable* 或者是否为*mapping* 等。

该模块提供了一个元类 *ABCMeta*，可以用来定义抽象类，另外还提供一个工具类 *ABC*，可以用它以继承的方式定义抽象基类。

class abc.**ABC**

一个使用 *ABCMeta* 作为元类的辅助类。使用这个类，可以通过简单地从 ABC 派生的方式创建抽象基类，这将避免时常令人混淆的元类用法，例如:

```
from abc import ABC

class MyABC(ABC):
    pass
```

请注意 ABC 的类型仍然是 *ABCMeta*，因此继承 ABC 仍然需要考虑使用元类的注意事项，比如可能会导致元类冲突的多重继承。你也可以通过传入 metaclass 关键字并直接使用 ABCMeta 来定义抽象基类，例如:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Added in version 3.4.

class abc.**ABCMeta**

用于定义抽象基类（ABC）的元类。

使用该元类以创建抽象基类。抽象基类可以像 mix-in 类一样直接被子类继承。你也可以将不相关的具体类（包括内建类）和抽象基类注册为“抽象子类”——这些类以及它们的子类会被内建函数 *issubclass()* 识别为对应的抽象基类的子类，但是该抽象基类不会出现在其 MRO（Method Resolution Order，方法解析顺序）中，抽象基类中实现的方法也不可调用（即使通过 *super()* 调用也不行）。[1]

使用 ABCMeta 元类创建的类具有以下方法:

register(*subclass*)

将 *subclass* 注册为该 ABC 的“虚拟子类”。例如:

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

在 3.3 版本发生变更: 返回注册的子类，使其能够作为类装饰器。

在 3.4 版本发生变更: 要检测对 `register()` 的调用，你可以使用 *get_cache_token()* 函数。

你也可以在虚基类中重写这个方法。

[1] C++ 程序员需要注意：Python 中虚基类的概念和 C++ 中的并不相同。

__subclasshook__(*subclass*)

(必须定义为类方法。)

检查 *subclass* 是否为该 ABC 的子类。这意味着你可以进一步定制*issubclass()* 的行为而无需在每个你希望作为该 ABC 的子类的类上调用*register()*。(这个类方法是在该 ABC 的 __subclasscheck__() 方法上被调用的。)

该方法应为返回 True，False 或*NotImplemented* 。如果返回 True ，则 子类被视为该抽象基类的子类。如果返回 False ，则 子类不被视为此抽象基类的子类，即使它通常是。如果返回 NotImplemented ，则继续按照常规机制检查子类。

为了对这些概念做一演示，请看以下定义 ABC 的示例：

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

ABC MyIterable 将标准的可迭代对象方法 __iter__() 定义为一个抽象方法。这里给出的实现仍可以从子类中调用。get_iterator() 方法也是 MyIterable 抽象基类的一部分，但它并非必须在非抽象的派生类中被重写。

这里定义的*__subclasshook__()* 类方法指明任何在其 __dict__ (或在其通过 __mro__ 列表访问的基本) 中具有 __iter__() 方法的类也都会被视为 MyIterable。

最后，末尾的行使得 Foo 成为 MyIterable 的一个虚子类，即使它没有定义 __iter__() 方法(它使用了以 __len__() 和 __getitem__() 等方法定义的旧式可迭代协议)。请注意这将不会使 get_iterator 成为 Foo 可用的方法，所以它会被单独提供。

abc 模块还提供了下列装饰器：

@abc.**abstractmethod**

用于声明抽象方法的装饰器。

使用此装饰器要求类的元类是*ABCMeta* 或是其派生类。一个具有派生自 ABCMeta 的元类的类无法被实例化，除非它全部的抽象方法和特征属性均已被重载。抽象方法可通过任何普通的'super' 调用机制来调用。abstractmethod() 可被用于声明特征属性和描述器的抽象方法。

动态地添加抽象方法到一个类，或尝试在方法或类被创建后修改其抽象状态等操作仅在使用*update_abstractmethods()* 函数时受到支持。abstractmethod() 只会影响使用常规继承所派生的子类；通过 ABC 的*register()* 方法注册的“虚子类”不会受到影响。

当 abstractmethod() 与其他方法描述器配合应用时，它应当被应用为最内层的装饰器，如以下用法示例所示:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

为了正确地与抽象基类机制互操作，描述器必须使用 __isabstractmethod__ 将自身标识为抽象的。通常，如果组成描述器的任一方法是抽象的，那么此属性就应为 True。例如，Python 的内置*property* 所做的就等价于:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

备注

不同于 Java 抽象方法，这些抽象方法可能具有一个实现。这个实现可在重写它的类上通过*super()* 机制来调用。这在使用协作多重继承的框架中可以被用作 super 调用的一个终点。

abc 模块还支持下列旧式装饰器:

@abc.abstractclassmethod

Added in version 3.2.

自 3.3 版本弃用: 现在可以让*classmethod* 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*classmethod()* 的子类，指明一个抽象类方法。在其他方面它都类似于*abstractmethod()*。

这个特例已被弃用，因为现在当*classmethod()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

@abc.**abstractstaticmethod**

Added in version 3.2.

自 3.3 版本弃用: 现在可以让*staticmethod* 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*staticmethod()* 的子类，指明一个抽象静态方法。在其他方面它都类似于*abstractmethod()*。

这个特例已被弃用，因为现在当*staticmethod()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

@abc.**abstractproperty**

自 3.3 版本弃用: 现在可以让*property*, *property.getter()*, *property.setter()* 和*property.deleter()* 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*property()* 的子类，指明一个抽象特性属性。

这个特例已被弃用，因为现在当*property()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

上面的例子定义了一个只读特征属性；你也可以通过适当地将一个或多个下层方法标记为抽象的来定义可读写的抽象特征属性:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

如果只有某些组件是抽象的，则只需更新那些组件即可在子类中创建具体的特征属性:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

abc 模块还提供了下列函数:

abc.**get_cache_token**()

返回当前抽象基类的缓存令牌

此令牌是一个不透明对象（支持相等性测试），用于为虚子类标识抽象基类缓存的当前版本。此令牌会在任何 ABC 上每次调用*ABCMeta.register()* 时发生更改。

Added in version 3.4.

abc.**update_abstractmethods**(*cls*)

重新计算一个抽象类的抽象状态的函数。如果一个类的抽象方法在类被创建后被实现或被修改则应当调用此函数。通常，此函数应当在一个类装饰器内部被调用。

返回 *cls*，使其能够用作类装饰器。

如果 *cls* 不是*ABCMeta* 的子类，则不做任何操作。

备注

此函数会假定 *cls* 的上级类已经被更新。它不会更新任何子类。

Added in version 3.10.

备注

30.10 atexit --- 退出处理器

atexit 模块定义了清理函数的注册和反注册函数. 被注册的函数会在解释器正常终止时执行. *atexit* 会按照注册顺序的 * 逆序 * 执行; 如果你注册了 A, B 和 C, 那么在解释器终止时会依序执行 C, B, A.

注意: 通过该模块注册的函数, 在程序被未被 Python 捕获的信号杀死时并不会执行, 在检测到 Python 内部致命错误以及调用了*os._exit()* 时也不会执行.

注意: 在清理函数内部注册或注销函数可能产生的影响是未定义的。

在 3.7 版本发生变更: 当配合 C-API 子解释器使用时，已注册函数是它们所注册解释器中的局部对象。

atexit.**register**(*func*, **args*, ***kwargs*)

将 *func* 注册为终止时执行的函数. 任何传给 *func* 的可选的参数都应当作为参数传给*register()*. 可以多次注册同样的函数及参数.

在正常的程序终止时 (举例来说, 当调用了*sys.exit()* 或是主模块的执行完成时), 所有注册过的函数都会以后进先出的顺序执行. 这样做是假定更底层的模块通常会比高层模块更早引入, 因此需要更晚清理.

如果在 exit 处理器执行期间引发了异常，将会打印回溯信息 (除非引发的是*SystemExit*) 并且异常信息会被保存。在所有 exit 处理器都获得运行机会之后，所引发的最后一个异常会被重新引发。

这个函数返回 *func* 对象，可以把它当作装饰器使用。

警告

启动新线程或从已注册的函数调用*os.fork()* 可能导致主 Python 运行时线程释放线程状态而内部*threading* 例程或新进程试图使用该状态之间的竞争条件。这会造成程序崩溃而不是正常关闭。

在 3.12 版本发生变更: 现在尝试启动新线程或在已注册的函数中*os.fork()* 新进程会导致*RuntimeError*。

atexit.**unregister**(*func*)

将 *func* 移出当解释器关闭时要运行的函数列表。如果 *func* 之前未被注册则*unregister()* 将静默地不做任何事。如果 *func* 已被注册一次以上，则该函数每次在*atexit* 调用栈中的出现都将被移除。当取消注册时会在内部使用相等性比较 (==)，因而函数引用不需要具有匹配的标识号。

参见

模块*readline*
使用*atexit* 读写*readline* 历史文件的有用的例子。

30.10.1 atexit 示例

以下简单例子演示了一个模块在被导入时如何从文件初始化一个计数器，并在程序终结时自动保存计数器的更新值，此操作不依赖于应用在终结时对此模块进行显式调用。:

```
try:
    with open('counterfile') as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open('counterfile', 'w') as outfile:
        outfile.write('%d' % _count)

import atexit

atexit.register(savecounter)
```

位置和关键字参数也可传入*register()* 以便传递给被调用的已注册函数:

```
def goodbye(name, adjective):
    print('Goodbye %s, it was %s to meet you.' % (name, adjective))

import atexit

atexit.register(goodbye, 'Donny', 'nice')
# 或者:
atexit.register(goodbye, adjective='nice', name='Donny')
```

作为*decorator*: 使用:

```
import atexit

@atexit.register
def goodbye():
    print('You are now leaving the Python sector.')
```

只有在函数不需要任何参数调用时才能工作.

30.11 traceback --- 打印或读取栈回溯信息

源代码： Lib/traceback.py

本模块提供了提取、格式化和打印 Python 程序的栈回溯信息的标准接口。它比解释器默认的回溯显示更灵活，因而使得配置输出的特定部分成为可能。最后，它还包含用于捕获有关异常的足够信息以供稍后打印的工具，而无需保存对实际异常的引用。由于异常可作为大型对象图的根对象，此工具能够显著地提升内存管理效率。

本模块使用 回溯对象 --- 它们是类型为*types.TracebackType* 的对象，它们将被赋值给*BaseException* 实例的*__traceback__* 字段。

参见

模块*faulthandler*
用于在发生错误、超时或用户信号时显式地转储 Python 回溯信息。

模块*pdb*
用于 Python 程序的交互式源代码调试器。

本模块的 API 可分为两部分：

- 提供基本功能的模块级函数，对于异常和回溯的交互式检查来说很有用处。
- *TracebackException* 类及其辅助类*StackSummary* 和*FrameSummary*。这些类提供了生成输出的更大灵活性和存储稍后进行格式化所需信息的能力而无需持有对实际异常和回溯对象的引用。

30.11.1 模块级函数

`traceback.`**`print_tb`**(*tb*, *limit=None*, *file=None*)

如果 *limit* 为正值则打印来自 回溯对象 *tb* 的至多 *limit* 个栈回溯条目（从调用方的帧开始）。否则，打印最后 `abs(limit)` 个条目。如果 *limit* 被省略或为 `None`，则打印所有条目。如果 *file* 被省略或为 `None`，则会输出到*sys.stderr*；在其他情况下它应当是一个打开的文件 或*file-like object* 用来接受输出。

备注

limit 形参的含义不同于*sys.tracebacklimit* 的含义。负的 *limit* 值对应于正的 `sys.tracebacklimit` 值，而正的 *limit* 值的行为无法用 `sys.tracebacklimit` 来达成。

在 3.5 版本发生变更: 添加了对负数值 *limit* 的支持

`traceback.`**`print_exception`**(*exc*, */*, [*value*, *tb*,]*limit=None*, *file=None*, *chain=True*)

将来自 回溯对象 *tb* 的异常信息与栈跟踪条目打印到 *file*。这与*print_tb()* 相比有以下几方面的区别：

- 如果 *tb* 不为 `None`，它将打印头部 `Traceback (most recent call last):`
- 它将在栈回溯之后打印异常类型和 *value*
- 如果 *type(value)* 为*SyntaxError* 且 *value* 具有适当的格式，它会打印发生语法错误的行并用一个圆点来指明错误的大致位置。

从 Python 3.10 开始，可以不再传递 *value* 和 *tb*，而是传递一个异常对象作为第一个参数。如果提供了 *value* 和 *tb*，则第一个参数会被忽略以便提供向下兼容性。

可选的 *limit* 参数的含义与*print_tb()* 的相同。如果 *chain* 为真值（默认），则链式异常（异常的*__cause__* 或*__context__* 属性）也将被打印出来，就像解释器本身在打印未处理的异常时一样。

在 3.5 版本发生变更: *etype* 参数会被忽略并根据 *value* 推断出来。

在 3.10 版本发生变更: *etype* 形参已被重命名为 *exc* 并且现在是仅限位置形参。

`traceback.`**`print_exc`**(*limit=None*, *file=None*, *chain=True*)

这是 `print_exception(sys.exception(), limit=limit, file=file, chain=chain)` 的快捷方式。

traceback.**print_last**(*limit=None*, *file=None*, *chain=True*)

这是 `print_exception(sys.last_exc, limit=limit, file=file, chain=chain)` 的快捷方式。通常它将只在异常到达交互提示符之后才会起作用 (参见*sys.last_exc*)。

traceback.**print_stack**(*f=None*, *limit=None*, *file=None*)

如果 *limit* 为正数则打印至多 *limit* 个栈跟踪条目（从唤起点开始）。在其他情况下，则打印最后 `abs(limit)` 个条目。如果 *limit* 被省略或为 `None`，则会打印所有条目。可选的 *f* 参数可被用来指定一个替代 栈帧作为开始位置。可选的 *file* 参数的含义与*print_tb()* 的相同。

在 3.5 版本发生变更: 添加了对负数值 *limit* 的支持

traceback.**extract_tb**(*tb*, *limit=None*)

返回一个*StackSummary* 对象来代表从 回溯对象 *tb* 提取的 “预处理” 栈跟踪条目列表。它可用作栈跟踪的另一种格式化形式。可选的 *limit* 参数的含义与*print_tb()* 的相同。“预处理” 栈跟踪条目是一个*FrameSummary* 对象，其中包含代表通常针对栈跟踪打印的信息的*filename*, *lineno*, *name* 和*line* 等属性。

traceback.**extract_stack**(*f=None*, *limit=None*)

从当前的 栈帧提取原始回溯。返回值的格式与*extract_tb()* 的相同。可选的 *f* 和 *limit* 参数的含义与*print_stack()* 的相同。

traceback.**print_list**(*extracted_list*, *file=None*)

将*extract_tb()* 或*extract_stack()* 返回的元组列表以带格式的栈回溯形式打印到给定的文件。如果 *file* 为 `None`，则输出将被写到*sys.stderr*。

traceback.**format_list**(*extracted_list*)

给定一个由元组或如*extract_tb()* 或*extract_stack()* 所返回的*FrameSummary* 对象组成的列表，返回一个可打印的字符串列表。结果列表中的每个字符串都对应于参数列表中具有相同索引号的条目。每个字符串以一个换行符结束；对于那些源文本行不为 `None` 的条目，字符串也可能包含内部换行符。

traceback.**format_exception_only**(*exc*, */*, [*value*,]*, *show_group=False*)

使用*sys.last_value* 等给出的异常值来格式化回溯的异常部分。返回值是一个字符串列表，其中每一项都以换行符结束。该列表包含异常消息，它通常是一个字符串；但是，对于*SyntaxError* 异常，它将包含多行并且（当打印时）会显示语法错误发生位置的详细信息。在异常消息之后，该列表还包含了异常的注释。

从 Python 3.10 开始，可以不传入 *value*，而是传入一个异常对象作为第一个参数。如果提供了 *value*，则第一个参数将被忽略以便提供向下兼容性。

当 *show_group* 为 `True`，并且异常为*BaseExceptionGroup* 的实例时，还会递归地包括嵌套的异常，并根据它们的嵌套深度添加缩进。

在 3.10 版本发生变更: *etype* 形参已被重命名为 *exc* 并且现在是仅限位置形参。

在 3.11 版本发生变更: 返回的列表现在将包括关联到异常的任何注释。

在 3.13 版本发生变更: 增加了 *show_group* 形参。

traceback.**format_exception**(*exc*, */*, [*value*, *tb*,]*limit=None*, *chain=True*)

格式化一个栈跟踪和异常信息。参数的含义与传给*print_exception()* 的相应参数相同。返回值是一个字符串列表，每个字符串都以一个换行符结束且有些还包含内部换行符。当这些行被拼接并打印时，打印的文本与*print_exception()* 的完全相同。

在 3.5 版本发生变更: *etype* 参数会被忽略并根据 *value* 推断出来。

在 3.10 版本发生变更: 此函数的行为和签名已被修改以与*print_exception()* 相匹配。

traceback.**format_exc**(*limit=None*, *chain=True*)

这类似于 `print_exc(limit)` 但会返回一个字符串而不是打印到一个文件。

traceback.**format_tb**(*tb*, *limit=None*)

是 `format_list(extract_tb(tb, limit))` 的简写形式。

traceback.**format_stack**(*f=None*, *limit=None*)

是 format_list(extract_stack(f, limit)) 的简写形式。

traceback.**clear_frames**(*tb*)

通过调用每个 帧对象的 clear() 方法来清除 回溯 *tb* 中所有栈帧的局部变量。

Added in version 3.4.

traceback.**walk_stack**(*f*)

从给定的帧开始访问 f.f_back 之后的栈内容，产生每一个帧和帧对应的行号。如果 *f* 为 None，则会使用当前栈。这个辅助函数要与*StackSummary.extract()* 一起使用。

Added in version 3.5.

traceback.**walk_tb**(*tb*)

访问 tb_next 之后的回溯并产生每一个帧和帧对应的行号。这个辅助函数要与*StackSummary.extract()* 一起使用。

Added in version 3.5.

30.11.2 TracebackException 对象

Added in version 3.5.

TracebackException 对象基于实际异常创建以便捕获数据供稍后打印。它们通过避免持有对 回溯和 帧对象的引用提供了存储此信息的更轻量方法。此外，相比上文所述的模块级函数它们还公开了更多选项用于配置输出。

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*, *compact=False*, *max_group_width=15*, *max_group_depth=10*)

捕获异常以供稍后渲染。*limit*, *lookup_lines* 和 *capture_locals* 的含义与*StackSummary* 类的相同。

如果 *compact* 为真值，则只有 TracebackException 的*format()* 方法所需要的数据会被保存在类属性性。特别地，*__context__* 字段只有在*__cause__* 为 None 且*__suppress_context__* 为假值时才会被计算。

请注意当局部变量被捕获时，它们也会被显示在回溯中。

max_group_width 和 *max_group_depth* 控制异常组的格式化 (参见*BaseExceptionGroup*)。depth 是指分组的嵌套层级，而 width 是指一个异常组的异常数组的大小。格式化的输出在达到某个限制时将被截断。

在 3.10 版本发生变更: 增加了 *compact* 形参。

在 3.11 版本发生变更: 添加了 *max_group_width* 和 *max_group_depth* 形参。parameters.

__cause__

原始*__cause__* 的 TracebackException。

__context__

原始*__context__* 的 TracebackException。

exceptions

如果 self 代表一个*ExceptionGroup*，此字段将保存一个由代表被嵌套异常的 TracebackException 实例组成的列表。否则它将为 None。

Added in version 3.11.

__suppress_context__

来自原始异常的*__suppress_context__* 值。

__notes__

来自原始异常的*__notes__* 值，或者如果异常没有任何注释则为 None。如果它不为 None 则会在异常字符串之后的回溯中进行格式化。

Added in version 3.11.

stack

代表回溯的*StackSummary*。

exc_type

原始回溯的类。

自 3.13 版本弃用.

exc_type_str

原始异常类的字符串显示。

Added in version 3.13.

filename

针对语法错误——错误发生所在的文件名。

lineno

针对语法错误——错误发生所在的行号。

end_lineno

针对语法错误——错误发生所在的末尾行号。如不存在则可以为 None。

Added in version 3.10.

text

针对语法错误——错误发生所在的文本。

offset

针对语法错误——错误发生所在的文本内部的偏移量。

end_offset

针对语法错误——错误发生所在的文本末尾偏移量。如不存在则可以为 None。

Added in version 3.10.

msg

针对语法错误——编译器错误消息。

classmethod from_exception(*exc*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

捕获一个异常以便随后渲染。*limit*, *lookup_lines* 和 *capture_locals* 的含义与*StackSummary* 类的相同。

请注意当局部变量被捕获时，它们也会被显示在回溯中。

print(*, *file=None*, *chain=True*)

将*format()* 所返回的异常信息打印至 *file* (默认为 sys.stderr)。

Added in version 3.11.

format(*, *chain=True*)

格式化异常。

如果 *chain* 不为 True，则*__cause__* 和*__context__* 将不会被格式化。

返回值是一个字符串的生成器，其中每个字符串都以换行符结束并且有些还会包含内部换行符。*print_exception()* 是此方法的一个包装器，它只是将这些行打印到一个文件。

format_exception_only(*, *show_group=False*)

格式化回溯的异常部分。

返回值是一个字符串的生成器，每个字符串都以一个换行符结束。

当 *show_group* 为 `False` 时，生成器会发出异常消息并附带其注释（如果有的话）。异常消息通常是一个字符串；但是，对于*SyntaxError* 异常，它将由多行组成并且（当打印时）会显示语法错误发生位置的详细信息。

当 *show_group* 为 `True`，并且异常为*BaseExceptionGroup* 的实例时，还会递归地包括嵌套的异常，并根据它们的嵌套深度添加缩进。

在 3.11 版本发生变更: 异常的注释 现在将被包括在输出中。

在 3.13 版本发生变更: 增加了 *show_group* 形参。

30.11.3 StackSummary 对象

Added in version 3.5.

`StackSummary` 对象代表一个可被格式化的调用栈。

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

根据一个帧生成器（例如由*walk_stack()* 或*walk_tb()* 所返回的对象）构造 `StackSummary` 对象。

如果提供了 *limit*，则只从 *frame_gen* 提取该参数所指定数量的帧。如果 *lookup_lines* 为 `False`，则返回的*FrameSummary* 对象将不会读入它们的行，这使得创建 `StackSummary` 的开销更低（如果它不会被实际格式化这就很有价值）。如果 *capture_locals* 为 `True` 则每个 `FrameSummary` 中的局部变量会被捕获为对象表示形式。

在 3.12 版本发生变更: 在局部变量的*repr()* 上被引发的异常（当 *capture_locals* 为 `True` 时）不会再被传播给调用方。

classmethod from_list(*a_list*)

从所提供的*FrameSummary* 对象列表或旧式的元组列表构造一个 `StackSummary` 对象。每个元组都应当是以 *文件名*, *行号*, *名称*, *行为*元素的 4 元组。

format()

返回一个可打印的字符串列表。结果列表中的每个字符串各自对应来自栈的单独的 帧。每个字符串都以一个换行符结束；对于带有源文本行的条目来说，字符串还可能包含内部换行符。

对于同一帧与行的长序列，将显示前几个重复项，后面跟一个指明之后的实际重复次数的摘要行。

在 3.6 版本发生变更: 重复帧的长序列现在将被缩减。

format_frame_summary(*frame_summary*)

返回用于打印栈中涉及的某一个 帧的字符串。此方法会为每个要用*StackSummary.format()* 来打印的*FrameSummary* 对象进行调用。如果它返回 `None`，该帧将从输出中被省略。

Added in version 3.11.

30.11.4 FrameSummary 对象

Added in version 3.5.

`FrameSummary` 对象表示 回溯中的某一个 帧。

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *, *lookup_line=True*, *locals=None*, *line=None*, *end_lineno=None*, *colno=None*, *end_colno=None*)

代表 回溯或栈中被格式化或打印的一个单独 帧。它还可能带有包括在其中的帧局部变量的字符串化版本。如果 *lookup_line* 为 `False`，则源代码不会被查找直到 `FrameSummary` 的*line* 属性被访

问（这还会在将其转换为*tuple* 时发生）。*line* 可能会被直接提供，并将完全阻止行查找的发生。*locals* 是一个可选的局部变量映射，如果有提供的话这些变量的表示形式将被存储在概要中以便随后显示。

FrameSummary 实例具有以下属性：

filename

对应于该帧的源代码的文件名。等价于访问 帧对象 *f* 上的 f.f_code.co_filename。

lineno

对应于该帧的源代码的行号。

name

等价于访问 帧对象 *f* 上的 f.f_code.co_name。

line

代表该帧的源代码的字符串，开头和末尾的空白将被去除。如果源代码不可用，它将为 None。

end_lineno

该帧源代码的末尾行号。在默认情况下，它将被设为 lineno 且索引号从 1 开始。

在 3.13 版本发生变更: 默认值从 None 改为 lineno。

colno

该帧源代码的列号。在默认情况下，它将为 None 且索引号从 0 开始。

end_colno

该帧源代码的末尾列号。在默认情况下，它将为 None 且索引号从 0 开始。

30.11.5 使用模块级函数的例子

这个简单示例是一个基本的读取-求值-打印循环，类似于（但实用性小于）标准 Python 交互式解释器循环。对于解释器循环的更完整实现，请参阅*code* 模块。

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

下面的例子演示了打印和格式化异常与回溯的不同方式：

```
import sys, traceback

def lumberjack():
    bright_side_of_life()

def bright_side_of_life():
    return tuple()[0]

try:
    lumberjack()
except IndexError as exc:
```

(续下页)

(接上页)

```
    print("*** print_tb:")
    traceback.print_tb(exc.__traceback__, limit=1, file=sys.stdout)
    print("*** print_exception:")
    traceback.print_exception(exc, limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    print(repr(traceback.format_exception(exc)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc.__traceback__)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc.__traceback__)))
    print("*** tb_lineno:", exc.__traceback__.tb_lineno)
```

该示例的输出看起来像是这样的:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\n    ~~~~~~
↪~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return tuple()[0]\n    ␣
↪      ~~~~~~~^^^\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_life>]
*** format_tb:
['  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\n    ~~~~~~
↪~~~~~~~~~~~~~~^^\n',
```

(续下页)

(接上页)

```
'  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return tuple()[0]\n    ␣
↪      ~~~~~~~^^^\n']
*** tb_lineno: 10
```

下面的例子演示了打印和格式化栈的不同方式:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_stack()))\n']
```

最后这个例子演示了最后几个格式化函数:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(an_error)
['IndexError: tuple index out of range\n']
```

30.11.6 使用 TracebackException 的示例

使用辅助类，我们将有更多的选项:

```
>>> import sys
>>> from traceback import TracebackException
>>>
>>> def lumberjack():
...     bright_side_of_life()
...
>>> def bright_side_of_life():
...     t = "bright", "side", "of", "life"
...     return t[5]
...
>>> try:
...     lumberjack()
... except IndexError as e:
...     exc = e
...
>>> try:
```

(续下页)

(接上页)

```
...     try:
...         lumberjack()
...     except:
...         1/0
... except Exception as e:
...     chained_exc = e
...
>>> # 限制使用模块级函数
>>> TracebackException.from_exception(exc, limit=-2).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

>>> # capture_locals 添加帧中的局部变量
>>> TracebackException.from_exception(exc, limit=-2, capture_locals=True).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
    t = ("bright", "side", "of", "life")
IndexError: tuple index out of range

>>> # print() 的 *chain* 关键字参数控制是否要显示
>>> # 串连的异常
>>> TracebackException.from_exception(chained_exc).print()
Traceback (most recent call last):
  File "<python-input-19>", line 4, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<python-input-8>", line 7, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~^^
  File "<python-input-8>", line 11, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero

>>> TracebackException.from_exception(chained_exc).print(chain=False)
Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero
```

30.12 __future__ --- Future 语句定义

源代码： Lib/__future__.py

`from __future__ import feature` 形式的导入被称为 future 语句。它们会被 Python 编译器当作特例，通过包含 future 语句来允许新的 Python 特性在该特性成为语言标准之前发布的模块中使用。

虽然这些 future 语句被 Python 编译器赋予了额外的特殊含义，但它们仍然像会其他导入语句一样被执行，而*__future__* 的存在和被导入系统处理的方式与其他任何 Python 模块的相同。这种设计有三个目的：

- 避免混淆已有的分析 import 语句并查找 import 的模块的工具。
- 当引入不兼容的修改时，可以记录其引入的时间以及强制使用的时间。这是一种可执行的文档，并且可以通过 import *__future__* 来做程序性的检查。
- 确保 future 语句在 Python 2.1 之前的发布版上运行时至少能抛出运行时异常（对*__future__* 的导入将失败，因为 will fail, because there was no module of that name prior to 2.1 之前没有这个模块名称）。

30.12.1 模块内容

__future__ 中不会删除特性的描述。从 Python 2.1 中首次加入以来，通过这种方式引入了以下特性：

特性	可选版本	强制加入版本	效果
nested_scopes	2.1.0b1	2.2	**PEP 227**: *静态嵌套作用域*
generators	2.2.0a1	2.3	**PEP 255**: *简单生成器*
division	2.2.0a2	3.0	**PEP 238**: *修改除法运算符*
absolute_import	2.5.0a1	3.0	**PEP 328**: *导入：多行与绝对/相对*
with_statement	2.5.0a1	2.6	**PEP 343**: *"with" 语句*
print_function	2.6.0a2	3.0	**PEP 3105**: *print 改为函数*
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Python 3000 中的字节字面值*
generator_stop	3.5.0b1	3.7	**PEP 479**: *在生成器中处理 StopIteration*
annotations	3.7.0b1	TBD[1]	**PEP 563**: *Postponed evaluation of annotations*

class __future__.**_Feature**

__future__.py 中的每一条语句都是以下格式的：

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

通常 *OptionalRelease* 要比 *MandatoryRelease* 小，并且都是和*sys.version_info* 格式一致的 5 元素元组。

```
(PY_MAJOR_VERSION, # 2.1.0a3 中的 2; 一个整数
 PY_MINOR_VERSION, # 1; 一个整数
 PY_MICRO_VERSION, # 0; 一个整数
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" 或 "final"; 字符串
 PY_RELEASE_SERIAL # 3; 一个整数
)
```

_Feature.**getOptionalRelease**()

OptionalRelease 记录了一个特性首次发布时的 Python 版本。

_Feature.**getMandatoryRelease**()

在 *MandatoryRelases* 还没有发布时，*MandatoryRelease* 表示该特性会变成语言的一部分的预测时间。

其他情况下，*MandatoryRelease* 用来记录这个特性是何时成为语言的一部分的。从该版本往后，使用该特性将不需要 future 语句，不过很多人还是会加上对应的 import。

[1] 先前计划在 Python 3.10 中强制使用 `from __future__ import annotations`，但 Python 指导委员会两次决定推迟这一改变（Python 3.10 的公告；Python 3.11 的公告）。目前还没有做出最终决定。参见 **PEP 563** 和 **PEP 649**。

MandatoryRelease 也可能为 None，表示计划中的特性被撤销或尚未确定。

_Feature.**compiler_flag**

CompilerFlag 是一个（位）旗标，对于动态编译的代码应当将其作为第四个参数传给内置函数*compile()* 以启用相应的特性。该旗标存储在*_Feature* 实例的*_Feature.compiler_flag* 属性中。

参见

future
编译器怎样处理 future import。

PEP 236 - 回到 __future__
有关 __future__ 机制的最初提议。

30.13 gc --- 垃圾回收器接口

此模块提供可选的垃圾回收器的接口，提供的功能包括：关闭收集器、调整收集频率、设置调试选项。它同时提供对回收器找到但是无法释放的不可达对象的访问。由于 Python 使用了带有引用计数的回收器，如果你确定你的程序不会产生循环引用，你可以关闭回收器。可以通过调用 gc.disable() 关闭自动垃圾回收。若要调试一个存在内存泄漏的程序，调用 gc.set_debug(gc.DEBUG_LEAK)；需要注意的是，它包含 gc.DEBUG_SAVEALL，使得被垃圾回收的对象会被存放在 gc.garbage 中以待检查。

gc 模块提供下列函数：

gc.**enable**()
启用自动垃圾回收

gc.**disable**()
停用自动垃圾回收

gc.**isenabled**()
如果启用了自动回收则返回 True。

gc.**collect**(*generation=2*)
不带参数时，将运行完全的回收。可选的参数 *generation* 是一个指定要回收哪一代 (从 0 到 2) 的整数值。如果 generation 值无效则会引发*ValueError*。返回值为已回收对象和不可回收对象的总数。

每当运行完整收集或最高代 (2) 收集时，为多个内置类型所维护的空闲列表会被清空。由于特定类型特别是*float* 的实现，在某些空闲列表中并非所有项都会被释放。

当解释器已经在执行收集任务时调用 gc.collect() 的效果是未定义的。

gc.**set_debug**(*flags*)
设置垃圾回收器的调试标识位。调试信息会被写入 sys.stderr 。此文档末尾列出了各个标志位及其含义；可以使用位操作对多个标志位进行设置以控制调试器。

gc.**get_debug**()
返回当前调试标识位。

gc.**get_objects**(*generation=None*)
返回一个由垃圾回收器所跟踪的所有对象组成的列表，不包括已返回对象的列表。如果 *generation* 不为 None，则只返回垃圾回收器所跟踪的属于该 generation 的对象。

在 3.8 版本发生变更: 新的 *generation* 形参。

引发一个审计事件 gc.get_objects 并附带参数 generation。

gc.**get_stats**()

返回一个包含三个字典对象的列表，每个字典分别包含对应代的从解释器开始运行的垃圾回收统计数据。字典的键的数目在将来可能发生改变，目前每个字典包含以下内容：

- `collections` 是该代被回收的次数；
- `collected` 是该代中被回收的对象总数；
- `uncollectable` 是在这一代中被发现无法收集的对象总数（因此被移动到*garbage* 列表中）。

Added in version 3.4.

gc.**set_threshold**(*threshold0*[, *threshold1*[, *threshold2*]])

设置垃圾回收阈值（收集频率）。将 *threshold0* 设为零会禁用回收。

垃圾回收器把所有对象分类为三代，其依据是对象在多少次垃圾回收后幸存。新建对象会被放在最年轻代（第 0 代）。如果一个对象在一次垃圾回收后幸存，它会被移入下一个较老代。由于第 2 代是最老代，这一代的对象在一次垃圾回收后仍会保留原样。为了确定何时要运行，垃圾回收器会跟踪自上一次回收后对象分配和释放的数量。当分配数量减去释放数量的结果值大于 *threshold0* 时，垃圾回收就会开始。初始时只有第 0 代会被检查。如果自第 1 代被检查后第 0 代已被检查超过 *threshold1* 次，则第 1 也会被检查。对于第三代来说情况还会更复杂，请参阅 Collecting the oldest generation 来了解详情。

gc.**get_count**()

将当前回收计数以形为 `(count0, count1, count2)` 的元组返回。

gc.**get_threshold**()

将当前回收阈值以形为 `(threshold0, threshold1, threshold2)` 的元组返回。

gc.**get_referrers**(**objs*)

返回直接引用任意一个 *objs* 的对象列表。这个函数只定位支持垃圾回收的容器；引用了其它对象但不支持垃圾回收的扩展类型不会被找到。

需要注意的是，已经解除对 *objs* 引用的对象，但仍存在于循环引用中未被回收时，仍然会被作为引用者出现在返回的列表当中。若要获取当前正在引用 *objs* 的对象，需要调用*collect()* 然后再调用*get_referrers()* 。

> **警告**
>
> 在使用*get_referrers()* 返回的对象时必须要小心，因为其中一些对象可能仍在构造中因此处于暂时的无效状态。不要把*get_referrers()* 用于调试以外的其它目的。

引发一个审计事件 `gc.get_referrers` 并附带参数 `objs`。

gc.**get_referents**(**objs*)

返回被任意一个参数中的对象直接引用的对象的列表。返回的被引用对象是被参数中的对象的 C 语言级别方法（若存在）`tp_traverse` 访问到的对象，可能不是所有的实际直接可达对象。只有支持垃圾回收的对象支持 `tp_traverse` 方法，并且此方法只会在需要访问涉及循环引用的对象时使用。因此，可以有以下例子：一个整数对其中一个参数是直接可达的，这个整数有可能出现或不出现在返回的结果列表当中。

引发一个审计事件 `gc.get_referents` 并附带参数 `objs`。

gc.**is_tracked**(*obj*)

当对象正在被垃圾回收器监控时返回 `True` ，否则返回 `False` 。一般来说，原子类的实例不会被监控，而非原子类（如容器、用户自定义的对象）会被监控。然而，会有一些特定类型的优化以便减少垃圾回收器在简单实例（如只含有原子性的键和值的字典）上的消耗。

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
```

(续下页)

(接上页)

```
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

Added in version 3.1.

gc.**is_finalized**(*obj*)

如果给定对象已被垃圾回收器终结则返回 True，否则返回 False。

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

Added in version 3.9.

gc.**freeze**()

冻结由垃圾回收器追踪的所有对象；将它们移至永久世代并在所有未来的回收操作中忽略它们。

如果一个进程将执行 fork() 而不执行 exec()，则在子进程中避免不必要的写入时拷贝将最大化内存共享并减少总体内存使用。这需要同时在父进程的内存页中避免创建已释放的“空洞”并确保在子进程中的 GC 回收不会触及源自父进程的长寿对象的 gc_refs 计数器。要同时达成这两个目标，请在父进程中尽早调用 gc.disable()，在 fork() 之前调用 gc.freeze()，并在子进程中尽早调用 gc.enable()。

Added in version 3.7.

gc.**unfreeze**()

解冻永久代中的对象，并将它们放回到年老代中。

Added in version 3.7.

gc.**get_freeze_count**()

返回永久代中的对象数量。

Added in version 3.7.

提供以下变量仅供只读访问（你可以修改但不应该重绑定它们）：

gc.**garbage**

一个回收器发现不可达而又无法被释放的对象（不可回收对象）列表。从 Python 3.4 开始，该列表在大多数时候都应该是空的，除非使用了含有非 NULL tp_del 空位的 C 扩展类型的实例。

如果设置了*DEBUG_SAVEALL*，则所有不可访问对象将被添加至该列表而不会被释放。

在 3.2 版本发生变更: 当*interpreter shutdown* 即解释器关闭时，若此列表非空，会产生*ResourceWarning*，即资源警告，在默认情况下此警告不会被提醒。如果设置了*DEBUG_UNCOLLECTABLE*，所有无法被回收的对象会被打印。

在 3.4 版本发生变更: 根据 **PEP 442**，具有 __del__() 方法的对象不会再由*gc.garbage* 来处理。

gc.**callbacks**

在垃圾回收器开始前和完成后会被调用的一系列回调函数。这些回调函数在被调用时使用两个参数：*phase* 和 *info* 。

phase 可为以下两值之一：

"start": 垃圾回收即将开始。

"stop": 垃圾回收已结束。

info is a dict providing more information for the callback. The following keys are currently defined:

"generation"（代）：正在被回收的最久远的一代。

"collected"（已回收的）：当 *phase* 为"stop" 时，被成功回收的对象的数目。

"uncollectable"（不可回收的）：当 *phase* 为"stop" 时，不能被回收并被放入*garbage* 的对象的数目。

应用程序可以把他们自己的回调函数加入此列表。主要的使用场景有：

统计垃圾回收的数据，如：不同代的回收频率、回收所花费的时间。

使应用程序可以识别和清理他们自己的在*garbage* 中的不可回收类型的对象。

Added in version 3.3.

以下常量被用于*set_debug()* ：

gc.**DEBUG_STATS**

在回收完成后打印统计信息。当回收频率设置较高时，这些信息会比较有用。

gc.**DEBUG_COLLECTABLE**

当发现可回收对象时打印信息。

gc.**DEBUG_UNCOLLECTABLE**

打印找到的不可回收对象的信息（指不能被回收器回收的不可达对象）。这些对象会被添加到 garbage 列表中。

在 3.2 版本发生变更: 当*interpreter shutdown* 时，即解释器关闭时，若*garbage* 列表中存在对象，这些对象也会被打印输出。

gc.**DEBUG_SAVEALL**

设置后，所有回收器找到的不可达对象会被添加进 *garbage* 而不是直接被释放。这在调试一个内存泄漏的程序时会很有用。

gc.**DEBUG_LEAK**

调试内存泄漏的程序时，使回收器打印信息的调试标识位。（等价于 DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL ）。

30.14 inspect --- 检查当前对象

源代码: Lib/inspect.py

---

inspect 模块提供了一些有用的函数帮助获取对象的信息，例如模块、类、方法、函数、回溯、帧对象以及代码对象。例如它可以帮助你检查类的内容，获取某个方法的源代码，取得并格式化某个函数的参数列表，或者获取你需要显示的回溯的详细信息。

该模块提供了 4 种主要的功能：类型检查、获取源代码、检查类与函数、检查解释器的调用堆栈。

30.14.1 类型和成员

getmembers() 函数获取对象如类或模块的成员。名称以 “is” 打头的函数主要是作为传给*getmembers()* 的第二个参数的便捷选项提供的。它们还可帮助你确定你是否能找到下列特殊属性（请参阅 import-mod-attrs 了解有关模块属性的详情）：

类型	属性	描述
class -- 类	__doc__	文档字符串
	__name__	类定义时所使用的名称
	__qualname__	qualified name -- 限定名称
	__module__	该类型被定义时所在的模块的名称
	__type_params__	一个包含泛型类的 类型形参的元组
method -- 方法	__doc__	文档字符串
	__name__	该方法定义时所使用的名称
	__qualname__	qualified name -- 限定名称
	__func__	实现该方法的函数对象
	__self__	该方法被绑定的实例，若没有绑定则为 `None`
	__module__	定义此方法的模块的名称
function -- 函数	__doc__	文档字符串
	__name__	用于定义此函数的名称
	__qualname__	qualified name -- 限定名称
	__code__	包含已编译函数的代码对象*bytecode*
	__defaults__	所有位置或关键字参数的默认值的元组
	__kwdefaults__	保存仅关键字参数的所有默认值的映射
	__globals__	此函数定义所在的全局命名空间
	__builtins__	builtins 命名空间
	__annotations__	参数名称到注解的映射；保留键 `"return"` 用于返回值注解。
	__type_params__	一个包含泛型函数的 类型形参的元组
	__module__	此函数定义所在的模块名称
traceback -- 回溯	tb_frame	此层的帧对象
	tb_lasti	在字节码中最后尝试的指令的索引
	tb_lineno	当前行在 Python 源代码中的行号
	tb_next	下一个内部回溯对象（由本层调用）
frame -- 帧	f_back	下一个外部帧对象（此帧的调用者）
	f_builtins	此帧执行时所在的 builtins 命名空间
	f_code	在此帧中执行的代码对象
	f_globals	此帧执行时所在的全局命名空间
	f_lasti	在字节码中最后尝试的指令的索引
	f_lineno	当前行在 Python 源代码中的行号
	f_locals	此帧所看到的局部命名空间
	f_trace	此帧的追踪函数，或 `None`
code -- 代码	co_argcount	参数数量（不包括仅关键字参数、* 或 ** 参数）
	co_code	字符串形式的原始字节码
	co_cellvars	单元变量名称的元组 (通过包含作用域引用)
	co_consts	字节码中使用的常量元组
	co_filename	创建此代码对象的文件的名称
	co_firstlineno	第一行在 Python 源代码中的行号
	co_flags	`CO_*` 标志的位图，详见此处
	co_lnotab	编码的行号到字节码索引的映射
	co_freevars	自由变量的名字组成的元组（通过函数闭包引用）
	co_posonlyargcount	仅限位置参数的数量
	co_kwonlyargcount	仅限关键字参数的数量（不包括 ** 参数）
	co_name	定义此代码对象的名称
	co_qualname	定义此代码对象的完整限定名称
	co_names	除参数和函数局部变量之外的名称元组
	co_nlocals	局部变量的数量
	co_stacksize	需要虚拟机堆栈空间

表 2 – 接上页

类型	属性	描述
	co_varnames	参数名和局部变量的元组
generator -- 生成器	__name__	名称
	__qualname__	qualified name -- 限定名称
	gi_frame	frame -- 帧
	gi_running	生成器在运行吗?
	gi_code	code -- 代码
	gi_yieldfrom	通过 `yield from``迭代的对象，或``None`
异步生成器	__name__	名称
	__qualname__	qualified name -- 限定名称
	ag_await	正在等待的对象，或 `None`
	ag_frame	frame -- 帧
	ag_running	生成器在运行吗?
	ag_code	code -- 代码
coroutine -- 协程	__name__	名称
	__qualname__	qualified name -- 限定名称
	cr_await	正在等待的对象，或 `None`
	cr_frame	frame -- 帧
	cr_running	这个协程正在运行吗?
	cr_code	code -- 代码
	cr_origin	协程被创建的位置，或 `None`。参见*sys.set_coroutine_origin_tracking_*
builtin	__doc__	文档字符串
	__name__	此函数或方法的原始名称
	__qualname__	qualified name -- 限定名称
	__self__	方法绑定到的实例，或 `None`

在 3.5 版本发生变更: 为生成器添加 `__qualname__` 和 `gi_yieldfrom` 属性。

生成器的 `__name__` 属性现在由函数名称设置，而不是代码对象名称，并且现在可以被修改。

在 3.7 版本发生变更: 为协程添加 `cr_origin` 属性。

在 3.10 版本发生变更: 为函数添加 `__builtins__` 属性。

`inspect.`**`getmembers`**(*object*[, *predicate*])

返回一个对象上的所有成员，组成以（名称，值）对为元素的列表，按名称排序。如果提供了可选的 *predicate* 参数（会对每个成员的 值对象进行一次调用），则仅包含该断言为真的成员。

> **备注**
>
> 当参数是一个类且这些属性在元类的自定义方法 `__dir__()` 中列出时*getmembers()* 将只返回在元类中定义的类属性。

`inspect.`**`getmembers_static`**(*object*[, *predicate*])

将一个对象的所有成员作为由 `(name, value)` 对组成并按名称排序的列表返回而不触发通过描述器协议 __getattr__ 或 __getattribute__ 执行的动态查找。或是作为可选项，只返回满足给定预期的成员。

> **备注**
>
> *getmembers_static()* 可能无法获得 getmembers 所能获取的所有成员（如动态创建的属性）并且可能会找到一些 getmembers 所不能找到的成员（如会引发 AttributeError 的描述器）。在某些情况下它还能返回描述器对象而不是实例成员。

Added in version 3.11.

inspect.**getmodulename**(*path*)

返回由文件名 *path* 表示的模块名字，但不包括外层的包名。文件扩展名会检查是否在*importlib.machinery.all_suffixes()* 列出的条目中。如果符合，则文件路径的最后一个组成部分会去掉后缀名后被返回；否则返回 None。

值得注意的是，这个函数 仅返回可以作为 Python 模块的名字，而有可能指向一个 Python 包的路径仍然会返回 None。

在 3.3 版本发生变更: 该函数直接基于*importlib*。

inspect.**ismodule**(*object*)

当该对象是一个模块时返回 True。

inspect.**isclass**(*object*)

当该对象是一个类时返回 True，无论是内置类或者 Python 代码中定义的类。

inspect.**ismethod**(*object*)

当该对象是一个 Python 写成的绑定方法时返回 True。

inspect.**isfunction**(*object*)

当该对象是一个 Python 函数时（包括使用*lambda* 表达式创造的函数），返回 True。

inspect.**isgeneratorfunction**(*object*)

当该对象是一个 Python 生成器函数时返回 True。

在 3.8 版本发生变更: 对于使用*functools.partial()* 封装的函数，如果被封装的函数是一个 Python 生成器函数，现在也会返回 True。

在 3.13 版本发生变更: 现在对于使用*functools.partialmethod()* 包装的函数，如果被包装的函数是一个 Python 生成器函数则返回 True。

inspect.**isgenerator**(*object*)

当该对象是一个生成器时返回 True。

inspect.**iscoroutinefunction**(*object*)

如果该对象为*coroutine function* (使用 async def 语法定义的函数)，包装了*coroutine function* 的*functools.partial()* 或使用*markcoroutinefunction()* 标记的同步函数则返回 True。

Added in version 3.5.

在 3.8 版本发生变更: 对于使用*functools.partial()* 封装的函数，如果被封装的函数是一个协程函数，现在也会返回 True。

在 3.12 版本发生变更: 使用*markcoroutinefunction()* 标记的同步函数现在将返回 True。

在 3.13 版本发生变更: 现在对于使用*functools.partialmethod()* 包装的函数，如果被包装的函数是一个*coroutine function* 则返回 True。

inspect.**markcoroutinefunction**(*func*)

将一个可调用对象标记为*coroutine function* 的装饰器，如果它不会被*iscoroutinefunction()* 检测到的话。

这可被用于返回*coroutine* 的同步函数，如果该函数被传给需要*iscoroutinefunction()* 的 API 的话。

在可能的情况下，更推荐使用 async def 函数。调用该函数并使用*iscoroutine()* 来检测其返回值也是可接受的。

Added in version 3.12.

inspect.**iscoroutine**(*object*)

当该对象是一个由 async def 函数创建的协程 时返回 True。

Added in version 3.5.

inspect.**isawaitable**(*object*)

如果该对象可以在 await 表达式中使用时返回 True。

也可被用于区分基于生成器的协程和常规的生成器:

```
import types

def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Added in version 3.5.

inspect.**isasyncgenfunction**(*object*)

如果对象是一个*asynchronous generator* 函数则返回 True，例如:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Added in version 3.6.

在 3.8 版本发生变更: 对于使用*functools.partial()* 封装的函数，如果被封装的函数是一个异步生成器 函数现在也会返回 True。

在 3.13 版本发生变更: 现在对于使用*functools.partialmethod()* 包装的函数，如果被包装的函数是一个*coroutine function* 则返回 True。

inspect.**isasyncgen**(*object*)

如果该对象是一个由异步生成器 函数创建的异步生成器迭代器，则返回 True。

Added in version 3.6.

inspect.**istraceback**(*object*)

如果该对象是一个回溯则返回 True。

inspect.**isframe**(*object*)

如果该对象是一个帧对象则返回 True。

inspect.**iscode**(*object*)

如果该对象是一个代码对象则返回 True。

inspect.**isbuiltin**(*object*)

如果该对象是一个内置函数或一个绑定的内置方法，则返回 True。

inspect.**ismethodwrapper**(*object*)

如果对象类型为*MethodWrapperType* 则返回 True。

这些是*MethodWrapperType* 的实例，如 __str__(), __eq__() 和 __repr__()。

Added in version 3.11.

inspect.**isroutine**(*object*)

如果该对象是一个用户定义的或内置的函数或者方法，则返回 True。

inspect.**isabstract**(*object*)

如果该对象是一个抽象基类则返回 True。

inspect.**ismethoddescriptor**(*object*)

如果该对象是一个方法描述器，但*ismethod()* 、*isclass()* 、*isfunction()* 及*isbuiltin()* 均不为真，则返回 True。

例如，该函数对于 int.__add__ 为真值。一个通过此测试的对象会有 __get__() 方法，但没有 __set__() 方法或 __delete__() 方法。除此以外，属性集合是可变的。一个*__name__* 属性通常是合理的，而*__doc__* 通常也是如此。

以描述器实现的能够通过其他某个测试的函数对于*ismethoddescriptor()* 测试也会返回 False，这只是因为其他测试提供了更多保证——比如，当一个对象通过*ismethod()* 时你将能够使用 __func__ 等属性。

在 3.13 版本发生变更: 此函数将不再错误地将具有 __get__() 和 __delete__()，但没有 __set__() 的对象报告为方法描述器（这样的对象是数据描述器，不是方法描述器）。

inspect.**isdatadescriptor**(*object*)

如果该对象是一个数据描述器则返回 True。

数据描述器具有 __set__ 或 __delete__ 方法。例如特征属性（在 Python 中定义）、getset 和成员等。后两者是在 C 中定义并且有针对这些类型的更具体的测试，它们在不同 Python 实现中均能保持健壮性。通常，数据描述器还具有*__name__* 和 __doc__ 属性（特征属性、getset 和成员都同时具有这些属性），但并不保证这一点。

inspect.**isgetsetdescriptor**(*object*)

如果该对象是一个 getset 描述器则返回 True。

getset 是在扩展模块中通过 PyGetSetDef 结构体定义的属性。对于不包含这种类型的 Python 实现，这个方法将永远返回 False。

inspect.**ismemberdescriptor**(*object*)

如果该对象是一个成员描述器则返回 True。

成员描述器是在扩展模块中通过 PyMemberDef 结构体定义的属性。对于不包含这种类型的 Python 实现，这个方法将永远返回 False。

30.14.2 获取源代码

inspect.**getdoc**(*object*)

获取对象的文档字符串并通过*cleandoc()* 进行清理。如果对象本身并未提供文档字符串并且这个对象是一个类、一个方法、一个特性或者一个描述器，将通过继承层次结构获取文档字符串。如果文档字符串无效或缺失，则返回 None。

在 3.5 版本发生变更: 文档字符串没有被重写的话现在会被继承。

inspect.**getcomments**(*object*)

任意多行注释作为单一一个字符串返回。对于类、函数和方法，选取紧贴在该对象的源代码之前的注释；对于模块，选取 Python 源文件顶部的注释。如果对象的源代码不可获得，返回 None。这可能是因为对象是 C 语言中或者是在交互式命令行中定义的。

inspect.**getfile**(*object*)

返回定义了这个对象的文件名（包括文本文件或二进制文件）。如果该对象是一个内置模块、类或函数则会失败并引发一个*TypeError*。

inspect.**getmodule**(*object*)

尝试猜测一个对象是在哪个模块中定义的。如果无法确定模块则返回 None。

inspect.**getsourcefile**(*object*)

返回一个对象定义所在 Python 源文件的名称，如果无法获取源文件则返回 None。如果对象是一个内置模块、类或函数则将失败并引发*TypeError*。

inspect.**getsourcelines**(*object*)

返回一个源代码行的列表和对象的起始行号。参数可以是一个模块、类、方法、函数、回溯或者代码对象。源代码将以与该对象所对应的行的列表的形式返回并且行号指明其中第一行代码出现在初始源文件的那个位置。如果源代码无法被获取则会引发*OSError*。如果对象是一个内置模块、类或函数则会引发*TypeError*。

在 3.3 版本发生变更: 现在会引发*OSError* 而不是*IOError*，后者现在是前者的一个别名。

inspect.**getsource**(*object*)

返回对象的源代码文本。参数可以是一个模块、类、方法、函数、回溯帧或代码对象。源代码将以单个字符串的形式被返回。如果源代码无法被获取则会引发*OSError*。如果对象是一个内置模块、类或函数则会引发*TypeError*。

在 3.3 版本发生变更: 现在会引发*OSError* 而不是*IOError*，后者现在是前者的一个别名。

inspect.**cleandoc**(*doc*)

清理文档字符串中为对齐当前代码块进行的缩进

第一行的所有前缀空白符会被移除。从第二行开始所有可以被统一去除的空白符也会被去除。之后，首尾的空白行也会被移除。同时，所有制表符会被展开到空格。

30.14.3 使用 Signature 对象对可调用对象进行内省

Added in version 3.3.

Signature 对象代表一个可调用对象的调用签名及其返回值标。要获取一个 Signature 对象，可使用 signature() 函数。

inspect.**signature**(*callable*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*)

返回一个给定 *callable* 的*Signature* 对象:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b: int, **kwargs)'

>>> str(sig.parameters['b'])
'b: int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

接受各类的 Python 可调用对象，包括单纯的函数、类，到*functools.partial()* 对象。

对于使用字符化标注的模块中定义的对象 (from __future__ import annotations)，*signature()* 会尝试使用*get_annotations()* 自动地反字符串化这些标注。*globals*, *locals* 和 *eval_str* 等形参会在解析标注时被传入*get_annotations()*；请参阅*get_annotations()* 的文档获取如何使用这些形参的说明。

如果没有可提供的签名则会引发*ValueError*，而如果对象类型不受支持则会引发*TypeError*。同时，如果标注被字符串化，并且 *eval_str* 不为假值，则在*get_annotations()* 中调用 eval() 来反字符串化标注可能会引发任何种类的异常。

函数签名中的斜杠（/）表示在它之前的参数是仅限位置的。详见 编程常见问题中关于仅限位置参数的条目

在 3.5 版本发生变更: 添加了 *follow_wrapped* 形参。传入 False 以获得特定 *callable* 的签名 (callable.__wrapped__ 将不会被用来解包被装饰的可调用对象。)

在 3.10 版本发生变更: 添加了 *globals*, *locals* 和 *eval_str* 形参。

> **备注**
>
> 一些可调用对象可能在特定 Python 实现中无法被内省。例如，在 CPython 中，部分通过 C 语言定义的内置函数不提供关于其参数的元数据。

如果传递的对象有一个 `__signature__` 属性，我们可以用它来创建签名。确切的语义是实现的一个细节，可能会有未经宣布的更改。有关当前语义，请查阅源代码。

class inspect.**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)

`Signature` 对象表示一个函数的调用签名及其返回值标注。对于函数所接受的每个形参它会在其*parameters* 多项集中存储一个*Parameter* 对象。

可选参数 *parameters* 是一个*Parameter* 对象组成的序列，它会在之后被验证不存在名字重复的参数，并且参数处于正确的顺序，即仅限位置参数最前，之后紧接着可位置可关键字参数，并且有默认值参数在无默认值参数之前。

可选的 *return_annotation* 参数可以是任意 Python 对象。它表示可调用对象的"return" 标注。

`Signature` 对象是 不可变对象。使用*Signature.replace()* 或*copy.replace()* 来创建经修改的副本。

在 3.5 版本发生变更: `Signature` 对象现在是可 pickle 且*hashable* 的对象。

empty

该类的一个特殊标记来明确指出返回值标注缺失。

parameters

一个参数名字到对应*Parameter* 对象的有序映射。参数以严格的定义顺序出现，包括仅关键字参数。

在 3.7 版本发生变更: Python 从 3.7 版起才显式地保证了它保持仅关键字参数的定义顺序，尽管实践上在 Python 3 中一直保持了这个顺序。

return_annotation

可调用对象的 "返回值" 标注。如果可调用对象没有 "返回值" 标注，这个属性会被设置为*Signature.empty*。

bind(**args*, ***kwargs*)

构造一个位置和关键字实参到形参的映射。如果 `*args` 和 `**kwargs` 符合签名，则返回一个*BoundArguments*；否则引发一个*TypeError* 。

bind_partial(**args*, ***kwargs*)

与*Signature.bind()* 作用方式相同，但允许省略部分必要的参数（模仿*functools.partial()* 的行为）。返回*BoundArguments*，或者在传入参数不符合签名的情况下，引发一个*TypeError*。

replace(**[, parameters][, return_annotation]*)

根据唤起*replace()* 的实例新建一个*Signature* 实例。可以通过传入不同的 *parameters* 和/或 *return_annotation* 来覆盖基类签名的相应特征属性。要从拷贝的 `Signature` 中移除 `return_annotation`，可以传入*Signature.empty*。

```
>>> def test(a, b):
...     pass
...
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

Signature 对象也被泛型函数*copy.replace()* 所支持。

format(*, *max_width=None*)

创建一个*Signature*对象的字符串表示形式。

如果传入了 *max_width*，该方法将尝试将签名调整为每行至多 *max_width* 个字符的多行文本。如果签名长度超过 *max_width*，所有形参都将位于单独的行。

Added in version 3.13.

classmethod from_callable(*obj*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*)

返回给定的可调用对象 *obj* 的*Signature*(或其子类)。

该方法简化了*Signature*的子类化操作:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(sum)
assert isinstance(sig, MySignature)
```

其行为在其他方面都与*signature()*相同。

Added in version 3.5.

在 3.10 版本发生变更: 添加了 *globals*, *locals* 和 *eval_str* 形参。

class inspect.**Parameter**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)

`Parameter` 对象是 不可变对象。不要直接修改 `Parameter` 对象，你可以使用*Parameter.replace()*或*copy.replace()*来创建一个修改后的副本。

在 3.5 版本发生变更: 现在 Parameter 对象可以被 pickle 并且为*hashable*。

empty

该类的一个特殊标记来明确指出默认值和标注的缺失。

name

参数的名字字符串。这个名字必须是一个合法的 Python 标识符。

CPython 会为用于实现推导式和生成器表达式的代码对象构造形如 `.0` 的隐式形参名。

在 3.6 版本发生变更: 这些形参名会被此模块公开为 `implicit0` 这样的名字。

default

该参数的默认值。如果该参数没有默认值，这个属性会被设置为*Parameter.empty*。

annotation

该参数的标注。如果该参数没有标注，这个属性会被设置为*Parameter.empty*。

kind

描述参数值要如何绑定到形参。可能的取值可通过*Parameter*获得 (如 `Parameter.KEYWORD_ONLY`)，并支持比较与排序，基于以下顺序:

名称	含意
POSITIONAL_ONLY	值必须以位置参数的方式提供。仅限位置参数是在函数定义中出现在 `/` 之前（如果有）的条目。
POSITIONAL_OR_KEYWORD	值既可以以关键字参数的形式提供，也可以以位置参数的形式提供（这是 Python 写成的函数的标准绑定行为的）。
VAR_POSITIONAL	没有绑定到其他形参的位置实参组成的元组。这对应于 Python 函数定义中的 `*args` 形参。
KEYWORD_ONLY	值必须以关键字参数的形式提供。仅限关键字形参是在 Python 函数定义中出现在 `*` 或 `*args` 之后的条目。
VAR_KEYWORD	一个未绑定到其他形参的关键字参数的字典。这对应于 Python 函数定义中的 `**kwargs` 形参。

示例：打印全部没有默认值的仅限关键字参数：

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                       param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

kind.**description**

描述*Parameter.kind* 的枚举值。

Added in version 3.8.

示例：打印全部参数的描述：

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

replace(**[, name][, kind][, default][, annotation]*)

根据唤起 replace 的实例新建一个*Parameter* 实例。要覆盖一个 Parameter 属性，可以传入相应的参数。要从一个 Parameter 中移除默认值或/和标注，可以传入*Parameter.empty*。

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # 将创建一个 'param' 的浅拷贝
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo: 'spam'"
```

Parameter 对象也被泛型函数*copy.replace()* 所支持。

在 3.4 版本发生变更: 在 Python 3.3 中*Parameter* 对象在其 kind 被设为 POSITIONAL_ONLY 时允许将 name 设为 None。现在已不再允许这样做。

class inspect.**BoundArguments**

调用*Signature.bind()* 或*Signature.bind_partial()* 的结果。容纳实参到函数的形参的映射。

arguments

一个形参名到实参值的可变映射。仅包含显式绑定的参数。对*arguments* 的修改会反映到*args* 和*kwargs* 上。

应当在任何参数处理目的中与*Signature.parameters* 结合使用。

备注

Signature.bind() 和*Signature.bind_partial()* 中采用默认值的参数被跳过。然而，如果有需要的话，可以使用*BoundArguments.apply_defaults()* 来添加它们。

在 3.9 版本发生变更: *arguments* 现在的类型是*dict*。之前，它的类型是*collections.OrderedDict*。

args

位置参数的值的元组。由*arguments* 属性动态计算。

kwargs

由关键字参数值组成的字典。根据*arguments* 属性动态计算得到。可按位置传入的参数已改为包括在*args* 中。

signature

向所属*Signature* 对象的一个引用。

apply_defaults()

设置缺失的参数的默认值。

对于变长位置参数（*args），默认值是一个空元组。

对于变长关键字参数（**kwargs）默认值是一个空字典。

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
{'a': 'spam', 'b': 'ham', 'args': ()}
```

Added in version 3.5.

args 和*kwargs* 特征属性可被用于唤起函数：

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

参见

PEP 362 - 函数签名对象。
包含具体的规范，实现细节和样例。

30.14.4 类与函数

inspect.**getclasstree**(*classes*, *unique=False*)

将给定的类的列表组织成嵌套列表的层级结构。每当一个内层列表出现时，它包含的类均派生自紧接着该列表之前的条目的类。每个条目均是一个二元组，包含一个类和它的基类组成的元组。如果*unique* 参数为真值，则给定列表中的每个类将恰有一个对应条目。否则，运用了多重继承的类和它们的后代将出现多次。

inspect.**getfullargspec**(*func*)

获取一个 Python 函数的形参的名字和默认值。将返回一个具名元组：

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
annotations)
```

args 是一个位置参数的名字的列表。*varargs* 是 * 形参的名字或 None 表示不接受任意长位置参数时。*varkw* 是 ** 参数的名字，或 None 表示不接受任意关键字参数。*defaults* 是一个包含了默认参数值的 *n* 元组分别对应最后 *n* 个位置参数，或 None 则表示没有默认值。*kwonlyargs* 是一个仅关键词参数列表，保持定义时的顺序。*kwonlydefaults* 是一个字典映射自 *kwonlyargs* 中包含的形参名。*annotations* 是一个字典，包含形参值到标注的映射。其中包含一个特殊的键 "return" 代表函数返回值的标注（如果有的话）。

注意：*signature()* 和*Signature 对象* 提供可调用对象内省更推荐的 API，并且支持扩展模块 API 中可能出现的额外的行为（比如仅限位置参数）。该函数被保留的主要原因是保持兼容 Python 2 的 inspect 模块 API。

在 3.4 版本发生变更: 该函数现在基于*signature()* 但仍然忽略 __wrapped__ 属性，并且在签名中包含绑定方法中的第一个绑定参数。

在 3.6 版本发生变更: 该方法在 Python 3.5 中曾因*signature()* 被文档归为弃用。但该决定已被推翻以恢复一个明确受支持的标准接口，以便运用一份源码通用 Python 2/3 间遗留的 getargspec() API 的迁移。

在 3.7 版本发生变更: Python 从 3.7 版起才显式地保证了它保持仅关键字参数的定义顺序，尽管实践上在 Python 3 中一直保持了这个顺序。

inspect.**getargvalues**(*frame*)

获取传入特定的帧的实参的信息。将返回一个具名元组 ArgInfo(args, varargs, keywords, locals)。*args* 是一个参数名字的列表。*varargs* 和 *keyword* 是 * 和 ** 参数的名字或 None。*locals* 是给定的帧的局部环境字典。

> **备注**
>
> 该函数因疏忽在 Python 3.5 中被错误地标记为弃用。

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

将*getargvalues()* 返回的四个值格式化为美观的参数规格。format* 的参数是对应的可选格式化函数以转化名字和值为字符串。

> **备注**
>
> 该函数因疏忽在 Python 3.5 中被错误地标记为弃用。

inspect.**getmro**(*cls*)

返回由类 cls 的全部基类按方法解析顺序组成的元组，包括 cls 本身。所有类不会在此元组中出现多于一次。注意方法解析顺序取决于 cls 的类型。除非使用一个非常奇怪的用户定义元类型，否则 cls 会是元组的第一个元素。

inspect.**getcallargs**(*func*, */*, **args*, ***kwds*)

将 *args* 和 *kwds* 绑定到 Python 函数或方法 *func* 的参数名称，就像将它们作为调用时传入的参数一样。对于绑定方法，还会将第一个参数 (通常命名为 self) 绑定到关联的实例。将返回一个字典，该字典会将参数名称（包括 * 和 ** 参数的名称，如果有的话）绑定到 *args* 和 *kwds* 中的值。对于不正确地唤起 *func* 的情况，即 func(*args, **kwds) 因函数签名不兼容而引发异常的时候，将引发一个相同类型的异常并附带相同或相似的消息。例如：

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
...
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
```

(续下页)

(接上页)

```
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

Added in version 3.2.

自 3.5 版本弃用: 改使用*Signature.bind()* 和*Signature.bind_partial()*。

inspect.**getclosurevars**(*func*)

获取自 Python 函数或方法 *func* 引用的外部名字到它们的值的映射。返回一个具名元组 `ClosureVars(nonlocals, globals, builtins, unbound)`。*nonlocals* 映射引用的名字到词法闭包变量，*globals* 映射到函数的模块级全局，*builtins* 映射到函数体内可见的内置变量。*unbound* 是在函数中引用但不能解析到给定的模块全局和内置变量的名字的集合。

如果 *func* 不是 Python 函数或方法，将引发*TypeError*。

Added in version 3.3.

inspect.**unwrap**(*func*, *, *stop=None*)

获取 *func* 所包装的对象。它追踪 `__wrapped__` 属性链并返回最后一个对象。

stop 是一个可选的回调，接受包装链的一个对象作为唯一参数，以允许通过让回调返回真值使解包装更早中止。如果回调不曾返回一个真值，将如常返回链中的最后一个对象。例如，*signature()* 使用该参数来在遇到具有 `__signature__` 参数的对象时停止解包装。

如果遇到循环，则引发*ValueError*。

Added in version 3.4.

inspect.**get_annotations**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*)

计算一个对象的标注字典。

`obj` 可以是一个可调用对象、类或模块。传入其他类型的对象将引发*TypeError*。

返回一个字典。`get_annotations()` 每次调用均返回一个新的字典；对同一个对象调用两次将获得两个不同但相等的字典。

该函数帮助你处理若干细节：

- 如果 `eval_str` 为真值，`str` 类型的值将使用*eval()* 来反字符串化。这被设计用于字符串化的标注 (`from __future__ import annotations`)。
- 如果 `obj` 不包含一个标注字典，返回一个空字典。(函数和方法永远包含一个标注字典；类、模块和其他类型的可调用对象则可能没有。)
- 对于类，忽略继承而来的标注。如果一个类不包含自己的标注字典，返回一个空字典。
- 因安全原因，所有对于对象成员和字典值的访问将通过 `getattr()` 和 `dict.get()` 完成。
- 请确保始终、始终、始终返回一个新创建的字典。

`eval_str` 控制 `str` 类型的值是否要替换为在这些值上调用*eval()* 的结果：

- 如果 eval_str 为真值，*eval()* 会在 `str` 类型的值上被调用。(请注意 `get_annotations` 并不捕获异常；如果*eval()* 引发了异常，它将展开栈至跳过 `get_annotations` 调用。)
- 如果 eval_str 为假（默认值），`str` 类型的值将不会被改变。

`globals` 和 `locals` 会被传入*eval()*；详情参见*eval()* 的文档。如果 `globals` 或 `locals` 为 `None`，此函数可能将该值替换为特定上下文的默认值，基于 `type(obj)` 确定：

- 如果 `obj` 是一个模块，`globals` 默认为 `obj.__dict__`。
- 如果 `obj` 是一个类，`globals` 默认值为 `sys.modules[obj.__module__].__dict__` 并且 `locals` 默认值为 `obj` 的类命名空间。

- 如果 obj 是一个可调用对象，则 globals 默认为 obj.__globals__，而如果 obj 是一个包装函数 (使用了 *functools.update_wrapper()*) 则它会先打开包装。

调用 get_annotations 是获取任何对象的标注字典的最佳实践。关于标注的最佳实践的更多信息，参见 annotations-howto。

Added in version 3.10.

30.14.5 解释器栈

下列函数有些将返回 *FrameInfo* 对象。出于向下兼容性考虑这些对象允许在所有属性上执行元组类操作但 positions 除外。此行为已被弃用并可能会在未来被移除。

class inspect.**FrameInfo**

frame
记录所对应的 帧对象。

filename
关联到由此记录所对应的帧对象所执行的代码的文件名称。

lineno
关联到由此记录所对应的帧对象所执行的代码的当前行的行号。

function
由此记录所对应的帧所执行的函数的名称。

code_context
来自由此记录所对应的帧所执行的源代码的上下文行组成的列表。

index
在 *code_context* 列表中执行的当前行的索引号。

positions
包含关联到由此记录所对应的指令的起始行号，结束行号，起始列偏移量和结束列偏移量的 *dis.Positions* 对象。

在 3.5 版本发生变更: 返回一个 *named tuple* 而非 *tuple*。

在 3.11 版本发生变更: FrameInfo 现在是一个类实例（以便与之前的 *named tuple* 保持向下兼容）。

class inspect.**Traceback**

filename
关联到由此回溯所对应的帧所执行的代码的文件名称。

lineno
关联到由此回溯所对应的帧所执行的代码的当前行的行号。

function
由此回溯所对应的帧所执行的函数的名称。

code_context
来自由此回溯所对应的帧所执行的源代码的上下文行组成的列表。

index
在 *code_context* 列表中执行的当前行的索引号。

positions
包含关联到此回溯所对应的帧所执行的指令的起始行号，结束行号，起始列偏移量和结束列偏移量的 *dis.Positions* 对象。

在 3.11 版本发生变更: Traceback 现在是一个类实例（以便与之前的 *named tuple* 保持向下兼容）。

备注

保留帧对象的引用（可见于这些函数返回的帧记录的第一个元素）会导致你的程序产生循环引用。每当一个循环引用被创建，所有可从产生循环的对象访问的对象的生命周期将会被大幅度延长，即便 Python 的可选的循环检测器被启用。如果这类循环必须被创建，确保它们会被显式地打破以避免对象销毁被延迟从而导致占用内存增加。

尽管循环检测器能够处理这种情况，这些帧（包括其局部变量）的销毁可以通过在 `finally` 子句中移除循环来产生确定的行为。对于循环检测器在编译 Python 时被禁用或者使用*gc.disable()* 时，这样处理更加尤为重要。比如：

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # 对 frame 执行一些操作
    finally:
        del frame
```

如果你希望保持帧更长的时间（比如在之后打印回溯），你也可以通过 `frame.clear()` 方法打破循环引用。

大部分这些函数支持的可选的 *context* 参数指定返回时包含的上下文的行数，以当前行为中心。

`inspect.`**`getframeinfo`**(*frame*, *context=1*)

获取关于帧或回溯对象的信息。将返回一个*Traceback* 对象。

在 3.11 版本发生变更: 将返回一个*Traceback* 对象而非具名元组。

`inspect.`**`getouterframes`**(*frame*, *context=1*)

获取某个帧及其所有外部帧的*FrameInfo* 对象的列表。这些帧代表导致 *frame* 被创建的一系列调用。返回的列中的第一个条目代表 *frame*；最后一个条目代表在 *frame* 的栈上的最外层调用。

在 3.5 版本发生变更: 返回一个具名元组 `FrameInfo(frame, filename, lineno, function, code_context, index)` 的列表。

在 3.11 版本发生变更: 将返回一个*FrameInfo* 对象的列表。

`inspect.`**`getinnerframes`**(*traceback*, *context=1*)

获取一个回溯所在的帧及其所有内部帧的*FrameInfo* 对象的列表。这些帧代表作为 *frame* 的后续所执行的调用。列表中的第一个条目代表 *traceback*；最后一个条目代表引发异常的位置。

在 3.5 版本发生变更: 返回一个具名元组 `FrameInfo(frame, filename, lineno, function, code_context, index)` 的列表。

在 3.11 版本发生变更: 将返回一个*FrameInfo* 对象的列表。

`inspect.`**`currentframe`**()

返回调用者的栈帧对应的帧对象。

该函数依赖于 Python 解释器对于栈帧的支持，这并非在 Python 的所有实现中被保证。该函数在不支持 Python 栈帧的实现中运行会返回 `None`。

`inspect.`**`stack`**(*context=1*)

返回调用者的栈的*FrameInfo* 对象的列表。返回的列表中的第一个条目代表调用者；最后一个条目代表栈上的最外层调用。

在 3.5 版本发生变更: 返回一个具名元组 `FrameInfo(frame, filename, lineno, function, code_context, index)` 的列表。

在 3.11 版本发生变更: 将返回一个*FrameInfo* 对象的列表。

`inspect.`**`trace`**(*context=1*)

返回介于当前帧和引发了当前正在处理的异常所在的帧之间的栈的*FrameInfo* 对象的列表。列表中的第一个条目代表调用者；最后一个条目代表引发异常的位置。

在 3.5 版本发生变更: 返回一个具名元组 `FrameInfo(frame, filename, lineno, function, code_context, index)` 的列表。

在 3.11 版本发生变更: 将返回一个 *FrameInfo* 对象的列表。

30.14.6 静态地获取属性

getattr() 和 *hasattr()* 在获取或检查属性是否存在时可以触发代码执行。描述器，像特征属性一样，可能会被唤起而 `__getattr__()` 和 `__getattribute__()` 也可能会被调用。

对于你想要静态地内省的情况，比如文档工具，这会显得不方便。*getattr_static()* 拥有与 *getattr()* 相同的签名，但避免了获取属性时执行代码。

`inspect.`**`getattr_static`**(*obj*, *attr*, *default=None*)

获取属性而不触发通过描述器协议、`__getattr__()` 或 `__getattribute__()` 的动态查找。

注意：该函数可能无法获取 getattr 能获取的全部的属性（比如动态地创建的属性），并且可能发现一些 getattr 无法找到的属性（比如描述器会引发 AttributeError）。它也能够返回描述器对象本身而非实例成员。

如果实例的 `__dict__` 被其他成员遮盖（比如一个特性）则该函数无法找到实例成员。

Added in version 3.2.

getattr_static() 不解析描述器。比如槽描述器或 C 语言中实现的 getset 描述器。该描述器对象会被直接返回，而不处理底层属性。

你可以用类似下方的代码的方法处理此事。注意，对于任意 getset 描述符，使用这段代码仍可能触发代码执行。

```
# 解析内置描述器类型的示例代码
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # 描述器可以引发 AttributeError
        # 以提示下层的值不存在
        # 在这种情况下描述器本身必须
        # 执行处理
        pass
```

30.14.7 生成器、协程和异步生成器的当前状态

当实现协程调度器或其他更高级的生成器用途时，判断一个生成器是正在执行、等待启动或继续或执行，又或者已经被终止是非常有用的。*getgeneratorstate()* 允许方便地判断一个生成器的当前状态。

`inspect.`**`getgeneratorstate`**(*generator*)

获取生成器迭代器的当前状态。

可能的状态是：

- GEN_CREATED：等待开始执行。
- GEN_RUNNING：正在被解释器执行。
- GEN_SUSPENDED：当前挂起于一个 yield 表达式。

- GEN_CLOSED：执行已经完成。

Added in version 3.2.

inspect.**getcoroutinestate**(*coroutine*)

获取协程对象的当前状态。该函数设计为用于使用 `async def` 函数创建的协程函数，但也能接受任何包括 `cr_running` 和 `cr_frame` 的类似协程的对象。

可能的状态是：

- CORO_CREATED：等待开始执行。
- CORO_RUNNING：当前正在被解释器执行。
- CORO_SUSPENDED：当前挂起于一个 await 表达式。
- CORO_CLOSED：执行已经完成。

Added in version 3.5.

inspect.**getasyncgenstate**(*agen*)

获取一个异步生成器对象的当前状态。该函数设计为用于由 `async def` 函数创建的使用 `yield` 语句的异步迭代器对象，但也能接受任何具有 `ag_running` 和 `ag_frame` 属性的类似异步生成器的对象。

可能的状态是：

- AGEN_CREATED: 等待开始执行。
- AGEN_RUNNING: 当前正在被解释器执行。
- AGEN_SUSPENDED: 当前在 yield 表达式上挂起。
- AGEN_CLOSED: 执行已经完成。

Added in version 3.12.

生成器当前的内部状态也可以被查询。这通常在测试目的中最为有用，来保证内部状态如预期一样被更新：

inspect.**getgeneratorlocals**(*generator*)

获取 *generator* 里的实时局部变量到当前值的映射。返回一个由名字映射到值的字典。这与在生成器的主体内调用 *locals()* 是等效的，并且相同的警告也适用。

如果 *generator* 是一个没有关联帧的生成器，则返回一个空字典。如果 *generator* 不是一个 Python 生成器对象，则引发 *TypeError*。

该函数依赖于生成器为内省暴露一个 Python 栈帧，这并非在 Python 的所有实现中被保证。在这种情况下，该函数将永远返回一个空字典。

Added in version 3.3.

inspect.**getcoroutinelocals**(*coroutine*)

该函数可类比于 *getgeneratorlocals()*，只是作用于由 `async def` 函数创建的协程。

Added in version 3.5.

inspect.**getasyncgenlocals**(*agen*)

此函数类似于 *getgeneratorlocals()*，但只适用于由 `async def` 函数创建的使用 `yield` 语句的异步生成器对象。

Added in version 3.12.

30.14.8 代码对象位标志

Python 代码对象有一个 `co_flags` 属性，它是下列旗标的位映射：

inspect.**CO_OPTIMIZED**

代码对象已经经过优化，会采用快速局部变量。

inspect.**CO_NEWLOCALS**
: 如果设置，则当代码对象被执行时会新建一个字典作为帧的 f_locals。

inspect.**CO_VARARGS**
: 代码对象拥有一个变长位置形参（类似 *args）。

inspect.**CO_VARKEYWORDS**
: 代码对象拥有一个可变关键字形参 (类似 **kwrags)。

inspect.**CO_NESTED**
: 该标志当代码对象是一个嵌套函数时被置位。

inspect.**CO_GENERATOR**
: 当代码对象是一个生成器函数，即调用时会返回一个生成器对象，则该标志被置位。

inspect.**CO_COROUTINE**
: 当代码对象是一个协程函数时被置位。当代码对象被执行时它返回一个协程。详见 **PEP 492**。

  Added in version 3.5.

inspect.**CO_ITERABLE_COROUTINE**
: 该标志被用于将生成器转变为基于生成器的协程。包含此标志的生成器对象可以被用于 await 表达式，并可以 yield from 协程对象。详见 **PEP 492**。

  Added in version 3.5.

inspect.**CO_ASYNC_GENERATOR**
: 当代码对象是一个异步生成器函数时该标志被置位。当代码对象被运行时它将返回一个异步生成器对象。详见 **PEP 525**。

  Added in version 3.6.

> **备注**
>
> 这些标志特指于 CPython，并且在其他 Python 实现中可能从未被定义。更进一步地说，这些标志是一种实现细节，并且可能在将来的 Python 发行中被移除或弃用。推荐使用*inspect* 模块的公共 API 来进行任何内省需求。

30.14.9 缓冲区旗标

class inspect.**BufferFlags**
: 这是一个代表可被传给实现了 缓冲区协议的对象的 __buffer__() 方法的旗标的*enum.IntFlag*。

  这些旗标的含义的说明见 buffer-request-types。

  SIMPLE

  WRITABLE

  FORMAT

  ND

  STRIDES

  C_CONTIGUOUS

  F_CONTIGUOUS

  ANY_CONTIGUOUS

  INDIRECT

CONTIG

CONTIG_RO

STRIDED

STRIDED_RO

RECORDS

RECORDS_RO

FULL

FULL_RO

READ

WRITE

Added in version 3.12.

30.14.10 命令行界面

inspect 模块也提供一个从命令行使用基本的内省能力。

默认地，命令行接受一个模块的名字并打印模块的源代码。也可通过后缀一个冒号和目标对象的限定名称来打印一个类或者一个函数。

--details

打印特定对象的信息而非源码。

30.15 site --- 站点专属的配置钩子

源代码: Lib/site.py

这个模块将在初始化时被自动导入。此自动导入可以通过使用解释器的 `-S` 选项来屏蔽。

正常导入此模块将会把站点专属的路径添加到模块搜索路径并将一些可调用对象，包括*help()* 添加到内置命名空间。不过，Python 启动选项 `-S` 会阻止此行为且此模块可被安全地导入而不会自动修改模块搜索路径或添加内置对象。要显式地触发通常的站点专属附加项，请调用*main()* 函数。

在 3.3 版本发生变更: 在之前即便使用了 `-S`，导入此模块仍然会触发路径操纵。

它会以一个头部和尾部来构造至多四个目录作为起点。对于头部，它将使用 `sys.prefix` 和 `sys.exec_prefix`；空的头部会被跳过。对于尾部，它将使用空字符串然后是 `lib/site-packages`（在 Windows 上）或 `lib/pythonX.Y[t]/site-packages`（在 Unix 和 macOS 上）。（可选后缀"t" 表示*free threading* 构建版，并会在 `"t"` 存在于*sys.abiflags* 常量中时被添加。）对于每个不同的头部-尾部组合，它会查看其是否指向现有的目录，如果确实如此，则将其添加到 `sys.path` 并且还会检查新添加目录中的配置文件。

在 3.5 版本发生变更: 对"site-python" 目录的支持已被移除。

在 3.13 版本发生变更: 在 Unix 上，自由线程 Python 安装版是在版本专属的目录名称中以"t" 后缀来标识的，例如 `lib/python3.13t/`。

如果名为"pyvenv.cfg" 的文件存在于 sys.executable 之上的一个目录中，则 sys.prefix 和 sys.exec_prefix 将被设置为该目录，并且还会检查 site-packages （sys.base_prefix 和 sys.base_exec_prefix 始终是 Python 安装的"真实" 前缀）。如果"pyvenv.cfg" （引导程序配置文件）包含设置为非"true"（不区分大小写）的"include-system-site-packages" 键，则不会在系统级前缀中搜索 site-packages；反之则会。

一个路径配置文件是具有 *name*`.pth` 命名格式的文件，并且存在上面提到的四个目录之一中；它的内容是要添加到 `sys.path` 中的额外项目（每行一个）。不存在的项目不会添加到 `sys.path`，并且不会检查项

目指向的是目录还是文件。项目不会被添加到 sys.path 超过一次。空行和由 # 起始的行会被跳过。以 import 开始的行（跟着空格或 TAB）会被执行。

备注

每次启动 Python，在 .pth 文件中的可执行行都将会被运行，而不管特定的模块实际上是否需要被使用。因此，其影响应降至最低。可执行行的主要预期目的是使相关模块可导入（加载第三方导入钩子，调整 PATH 等）。如果它发生了，任何其他的初始化都应当在模块实际导入之前完成。将代码块限制为一行是一种有意采取的措施，不鼓励在此处放置更复杂的内容。

在 3.13 版本发生变更: 现在 .pth 文件会首先使用 UTF-8 来解码，如果失败会再改用*locale encoding* 来解码。

例如，假设 sys.prefix 和 sys.exec_prefix 已经被设置为 /usr/local。Python X.Y 的库之后被安装为 /usr/local/lib/pythonX.Y。假设有一个拥有三个孙目录 foo, bar 和 spam 的子目录 /usr/local/lib/pythonX.Y/site-packages，并且有两个路径配置文件 foo.pth 和 bar.pth。假定 foo.pth 内容如下:

```
# foo 包配置

foo
bar
bletch
```

并且 bar.pth 包含:

```
# bar 包配置

bar
```

则下面特定版目录将以如下顺序被添加到 sys.path。

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

请注意 bletch 已被省略因为它并不存在；bar 目前在 foo 目录之前因为 bar.pth 按字母顺序排在 foo.pth 之前；而 spam 已被省略因为它在两个路径配置文件中都未被提及。

30.15.1 sitecustomize

在这些路径操作之后，会尝试导入一个名为*sitecustomize* 的模块，它可以执行任意站点专属的定制。它通常是由系统管理员在 site-packages 目录下创建的。如果此导入失败并引发*ImportError* 或其子类的异常，并且异常的*name* 属性等于 'sitecustomize'，则它会被静默地忽略。如果 Python 是在没有可用输出流的情况下启动的，例如在 Windows 上使用 pythonw.exe (它被默认被用于 IDLE)，则来自*sitecustomize* 的输出尝试会被忽略。任何其他异常都会导致静默且可能令人困惑的进程失败。

30.15.2 usercustomize

在此之后，会尝试导入一个名为*usercustomize* 的模块，如果*ENABLE_USER_SITE* 为真值，则它可以执行任意的用户专属定制。这个文件应在用户的 site-packages 目录中创建（见下文），除非被 -s 所禁用，在其他情况下该目录都是 sys.path 的组成部分。如果此导入失败并引发*ImportError* 或其子类异常，并且异常的*name* 属性等于 'usercustomize'，它会被静默地忽略。

请注意对于某些非 Unix 系统来说，sys.prefix 和 sys.exec_prefix 均为空值，并且路径操作会被跳过；但是仍然会尝试导入*sitecustomize* 和*usercustomize*。

30.15.3 Readline 配置

在支持*readline*的系统上，这个模块也将导入并配置*rlcompleter*模块，如果 Python 是以 交互模式启动并且不带 `-S` 选项的话。默认的行为是启用 tab 键补全并使用 `~/.python_history` 作为历史存档文件。要禁用它，请删除（或重载）你的*sitecustomize*或*usercustomize*模块或 `PYTHONSTARTUP` 文件中的*sys.__interactivehook__*属性。

在 3.4 版本发生变更: rlcompleter 和 history 会被自动激活。

30.15.4 模块内容

`site.`**`PREFIXES`**

site-packages 目录的前缀列表。

`site.`**`ENABLE_USER_SITE`**

显示用户 site-packages 目录状态的旗标。`True` 意味着它被启用并被添加到 `sys.path`。`False` 意味着它按照用户请求被禁用 (通过 `-s` 或 `PYTHONNOUSERSITE`)。`None` 意味着它因安全理由（user 或 group id 和 effective id 之间不匹配）或是被管理员所禁用。

`site.`**`USER_SITE`**

运行中的 Python 的用户级 site-packages 的路径。它可以为 `None`，如果*getusersitepackages()*尚未被调用的话。默认值在 UNIX 和 macOS 非框架构建版上为 `~/.local/lib/pythonX.Y[t]/site-packages`，在 macOS 框架构建版上为 `~/Library/Python/X.Y/lib/python/site-packages`，而在 Windows 上为 `%APPDATA%\Python\PythonXY\site-packages`。可选的"t"表示自由线程构建版。此目录属于 site 目录，这意味着其中的 `.pth` 文件将会被处理。files in it will be processed.

`site.`**`USER_BASE`**

用户级 site-packages 目录的路径。如果尚未调用*getuserbase()*则它可以为 `None`。默认值在 Unix 和 macOS 非框架编译版上为 `~/.local`，在 macOS 框架编译版上为 `~/Library/Python/X.Y`，而在 Windows 上则为 `%APPDATA%\Python`。这个值会被用于计算针对*用户安装方案*的脚本、数据文件、Python 模块等的安装目录。另请参阅 `PYTHONUSERBASE`。

`site.`**`main`**`()`

将所有的标准站点专属目录添加到模块搜索路径。这个函数会在导入此模块时被自动调用，除非 Python 解释器启动时附带了 `-S` 旗标。

在 3.3 版本发生变更: 这个函数使用无条件调用。

`site.`**`addsitedir`**`(`*sitedir*, *known_paths=None*`)`

将一个目录添加到 sys.path 并处理其 `.pth` 文件。通常被用于*sitecustomize*或*usercustomize*(见下文)。

`site.`**`getsitepackages`**`()`

返回包含所有全局 site-packages 目录的列表。

Added in version 3.2.

`site.`**`getuserbase`**`()`

返回用户基准目录的路径*USER_BASE*。如果它尚未被初始化，则此函数还将参照 `PYTHONUSERBASE` 来设置它。

Added in version 3.2.

`site.`**`getusersitepackages`**`()`

返回用户专属 site-packages 目录的路径*USER_SITE*。如果它尚未被初始化，则此函数还将参照*USER_BASE*来设置它。要确定用户专属 site-packages 是否已被添加到 `sys.path` 则应当使用*ENABLE_USER_SITE*。

Added in version 3.2.

30.15.5 命令行界面

site 模块还提供了一个从命令行获取用户目录的方式:

```
$ python -m site --user-site
/home/user/.local/lib/python3.11/site-packages
```

如果它被不带参数地调用，它将在标准输出打印*sys.path* 的内容，再打印*USER_BASE* 的值以及该目录是否存在，然后打印*USER_SITE* 的相应信息，最后打印*ENABLE_USER_SITE* 的值。

--user-base

输出用户基本的路径。

--user-site

输出用户 site-packages 目录的路径。

如果同时给出了两个选项，则将打印用户基准目录和用户站点信息（总是按此顺序），并以*os.pathsep* 分隔。

如果给出了其中一个选项，脚本将退出并返回以下值中的一个：如果用户级 site-packages 目录被启用则为 `0`，如果它被用户禁用则为 `1`，如果它因安全理由或被管理员禁用则为 `2`，如果发生错误则为大于 2 的值。

参见

- **PEP 370** -- 分用户的 site-packages 目录
- *sys.path 模块搜索路径的初始化* -- *sys.path* 的初始化。

CHAPTER 31

自定义 Python 解释器

本章中描述的模块允许编写类似于 Python 的交互式解释器的接口。如果你想要一个支持附加某些特殊功能到 Python 语言的 Python 解释器，你应当看一下 *code* 模块。(*codeop* 模块是低层级的，用于支持编译可能不完整的 Python 代码块。

本章描述的完整模块列表如下：

31.1 code --- 解释器基类

源代码： Lib/code.py

`code` 模块提供了在 Python 中实现 read-eval-print 循环的功能。它包含两个类和一些快捷功能，可用于构建提供交互式解释器的应用程序。

class `code.`**InteractiveInterpreter**(*locals=None*)

这个类会处理解析和解释器状态（用户的命名空间）；它不会处理输入缓冲、提示或输入文件命名（文件名总是显式地传入）。可选的 *locals* 参数指定一个映射作为代码执行所在的命名空间；它默认是一个新创建的字典，该字典中的键 `'__name__'` 设为 `'__console__'` 而键 `'__doc__'` 设为 `None`。

class `code.`**InteractiveConsole**(*locals=None*, *filename='<console>'*, *local_exit=False*)

高度模仿交互式 Python 解释器的行为。这个类基于 *InteractiveInterpreter* 构建并使用熟悉的 `sys.ps1` 和 `sys.ps2` 来增加提示，以及输入缓冲功能。如果 *local_exit* 为真值，控制台中的 `exit()` 和 `quit()` 将不会引发 *SystemExit*，而是返回到调用方代码。

在 3.13 版本发生变更: 增加发 *local_exit* 形参。

`code.`**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*, *local_exit=False*)

运行一个读取-求值-打印循环的便捷函数。这会创建一个新的 *InteractiveConsole* 实例并设置 *readfunc* 作为 *InteractiveConsole.raw_input()* 方法，如果有提供的话。如果提供了 *local*，它将被传给 *InteractiveConsole* 构造器以用作解释器循环的默认命名空间。如果提供了 *local_exit*，它将被传给 *InteractiveConsole* 构造器。实例的 *interact()* 方法将随后运行并传入 *banner* 和 *exitmsg* 以作为标题和退出消息，如果有提供的话。控制台对象在使用后将被丢弃。

在 3.6 版本发生变更: 加入 *exitmsg* 参数。

在 3.13 版本发生变更: 增加发 *local_exit* 形参。

code.**compile_command**(*source*, *filename='<input>'*, *symbol='single'*)

这个函数主要用来模拟 Python 解释器的主循环（即 read-eval-print 循环）。难点的部分是当用户输入不完整命令时，判断能否通过之后的输入来完成（要么成为完整的命令，要么语法错误）。该函数几乎和实际的解释器主循环的判断是相同的。

source 是源字符串；*filename* 是可选的用作读取源的文件名，默认为 `'<input>'`；*symbol* 是可选的语法开启符号，应为 `'single'` (默认), `'eval'` 或 `'exec'`。

如果命令完整且有效则返回一个代码对象 (等价于 `compile(source, filename, symbol)`)；如果命令不完整则返回 `None`；如果命令完整但包含语法错误则会引发*SyntaxError* 或*OverflowError* 而如果命令包含无效字面值则将引发*ValueError*。

31.1.1 交互解释器对象

InteractiveInterpreter.**runsource**(*source*, *filename='<input>'*, *symbol='single'*)

在解释器中编译并运行一段源码。所用参数与*compile_command()* 一样；*filename* 的默认值为 `'<input>'`，*symbol* 则为 `'single'`。可能发生以下情况之一：

- 输入不正确；*compile_command()* 引发了一个异常 (*SyntaxError* 或*OverflowError*)。将通过调用*showsyntaxerror()* 方法打印语法回溯信息。*runsource()* 返回 `False`。
- 输入不完整，需要更多输入；函数*compile_command()* 返回 `None` 。方法*runsource()* 返回 `True` 。
- 输入完整；*compile_command()* 返回了一个代码对象。将通过调用*runcode()* 执行代码（该方法也会处理运行时异常，*SystemExit* 除外）。*runsource()* 返回 `False`。

该返回值用于决定使用 `sys.ps1` 还是 `sys.ps2` 来作为下一行的输入提示符。

InteractiveInterpreter.**runcode**(*code*)

执行一个代码对象。当发生异常时，调用*showtraceback()* 来显示回溯。除*SystemExit* （允许传播）以外的所有异常都会被捕获。

有关*KeyboardInterrupt* 的说明，该异常可能发生于此代码的其他位置，并且并不总能被捕获。调用者应当准备好处理它。

InteractiveInterpreter.**showsyntaxerror**(*filename=None*)

显示刚发生的语法错误。这不会显示堆栈回溯因为语法错误并无此种信息。如果给出了 *filename*，它会被放入异常来替代 Python 解析器所提供的默认文件名，因为它在从一个字符串读取时总是会使用 `'<string>'`。输出将由*write()* 方法来写入。

InteractiveInterpreter.**showtraceback**()

显示刚发生的异常。我们移除了第一个堆栈条目因为它从属于解释器对象的实现。输出将由*write()* 方法来写入。

在 3.5 版本发生变更: 将显示完整的链式回溯，而不只是主回溯。

InteractiveInterpreter.**write**(*data*)

将一个字符串写入到标准错误流 (`sys.stderr`)。所有派生类都应重写此方法以提供必要的正确输出处理。

31.1.2 交互式控制台对象

InteractiveConsole 类是*InteractiveInterpreter* 的子类，因此它提供了解释器对象的所有方法，还有以下的额外方法。

InteractiveConsole.**interact**(*banner=None*, *exitmsg=None*)

近似地模拟交互式 Python 终端。可选的 *banner* 参数指定要在第一次交互前打印的条幅；默认情况下会类似于标准 Python 解释器所打印的内容，并附上外加圆括号的终端对象类名（这样就不会与真正的解释器混淆——因为确实太像了！）

可选的 *exitmsg* 参数指定要在退出时打印的退出消息。传入空字符串可以屏蔽退出消息。如果 *exitmsg* 未给出或为 `None`，则将打印默认消息。

在 3.4 版本发生变更: 要禁止打印任何条幅消息，请传递一个空字符串。

在 3.6 版本发生变更: 退出时打印退出消息。

`InteractiveConsole.`**`push`**`(`*`line`*`)`

将一行源代码文本推入解释器。行内容不应带有末尾换行符；它可以有内部换行符。行内容会被添加到一缓冲区然后调用解释器的*runsource()* 方法并附带缓冲区内容的拼接结果作为源文本。如果提示命令已执行或不合法，缓冲区将被重置；在其他情况下，则命令结束，缓冲区将在添加行后保持原样。如果需要更多的输入则返回值为 `True`，如果行已按某种方式被处理则返回值为 `False` (这与 `runsource()` 相同)。

`InteractiveConsole.`**`resetbuffer`**`()`

从输入缓冲区中删除所有未处理的内容。

`InteractiveConsole.`**`raw_input`**`(`*`prompt=''`*`)`

输出提示并读取一行。返回的行不包含末尾的换行符。当用户输入 EOF 键序列时，会引发*EOFError* 异常。默认实现是从 `sys.stdin` 读取；子类可以用其他实现代替。

31.2 codeop --- 编译 Python 代码

源代码： Lib/codeop.py

---

codeop 模块提供了可以模拟 Python 读取-执行-打印循环的实用程序，就像在*code* 模块中一样。因此，您可能不希望直接使用该模块；如果你想在程序中包含这样一个循环，你可能需要使用*code* 模块。

这个任务有两个部分:

1. 能够判断一行输入是否完成了一条 Python 语句：简而言之，就是告诉我们接下来是要打印'>>>' 还是'...'。
2. 记住用户已输入了哪些 future 语句，这样后续的输入可以在这些语句生效的状态下被编译。

codeop 模块提供了分别以及同时执行这两个部分的方式。

只执行前一部分:

`codeop.`**`compile_command`**`(`*`source`*`, `*`filename='<input>'`*`, `*`symbol='single'`*`)`

尝试编译 *source*，这应当是一个 Python 代码字符串并且在 *source* 是有效的 Python 代码时返回一个对象对象。在此情况下，代码对象的 filename 属性将为 *filename*，其默认值为 `'<input>'`。如果 *source* 不是 *not* 有效的 Python 代码，而是有效的 Python 代码的一个前缀时将返回 `None`。

如果 *source* 存在问题，将引发异常。如果存在无效的 Python 语法将引发*SyntaxError*，而如果存在无效的字面值则将引发*OverflowError* 或*ValueError*。

symbol 参数确定 *source* 是作为一条语句 (`'single'`，为默认值)，作为一个*statement* 的序列 (`'exec'`) 还是作为一个*expression* (`'eval'`) 来进行编译。任何其他值都将导致引发*ValueError*。

> **备注**
>
> 解析器有可能（但很不常见）会在到达源码结尾之前停止解析并成功输出结果；在这种情况下，末尾的符号可能会被忽略而不是引发错误。例如，一个反斜杠加两个换行符之后可以跟随任何无意义的符号。一旦解析器 API 得到改进将修正这个问题。

`class` `codeop.`**`Compile`**

该类的实例具有 `__call__()` 方法，其签名与内置函数*compile()* 相似，区别在于如果该实例编译了包含*__future__* 语句的程序文本，则该实例会‘记住’并编译后续所有的所有包含该语句的程序文本。

class codeop.**CommandCompiler**

该类的实例具有 `__call__()` 方法，其签名与*compile_command()* 相似；区别在如果该实例编译了包含*__future__* 语句的程序文本，则该实例会‘记住’并编译后续所有的包含该语句的程序文本。

CHAPTER 32

导入模块

本章中介绍的模块提供了导入其他 Python 模块和挂钩以自定义导入过程的新方法。

本章描述的完整模块列表如下：

32.1 `zipimport` --- 从 Zip 归档导入模块

源代码： Lib/zipimport.py

此模块添加了从 ZIP 格式档案中导入 Python 模块（`*.py` ， `*.pyc` ）和包的能力。通常不需要明确地使用 *zipimport* 模块，内置的 `import` 机制会自动将此模块用于 ZIP 档案路径的 *sys.path* 项目上。

通常， *sys.path* 是字符串的目录名称列表。此模块同样允许 *sys.path* 的一项成为命名 ZIP 文件档案的字符串。ZIP 档案可以容纳子目录结构去支持包的导入，并且可以将归档文件中的路径指定为仅从子目录导入。比如说，路径 `example.zip/lib/` 将只会从档案中的 `lib/` 子目录导入。

任何文件都可以放到 ZIP 档案中，但只有 `.py` 和 `.pyc` 文件会触发导入器操作。动态模块 (`.pyd`, `.so`) 的 ZIP 导入是不被允许的。请注意如果一个档案只包含有 `.py` 文件，那么 Python 将不会尝试通过添加对应的 `.pyc` 文件来修改档案，这意味着如果一个 ZIP 档案不包含 `.pyc` 文件，则导入速度可能会相当慢。

在 3.13 版本发生变更: 已支持 ZIP64

在 3.8 版本发生变更: 以前，不支持带有档案注释的 ZIP 档案。

参见

PKZIP Application Note
Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

PEP 273 - 从 ZIP 压缩包导入模块
由 James C. Ahlstrom 编写，他也提供了实现。Python 2.3 遵循 **PEP 273** 的规范，但是使用 Just van Rossum 编写的使用了 **PEP 302** 中描述的导入钩的实现。

***importlib* - 导入机制的实现**
为所有导入器的实现提供相关协议的包。

此模块定义了一个异常：

exception zipimport.**ZipImportError**

异常由 zipimporter 对象引发。这是*ImportError* 的子类，因此，也可以捕获为*ImportError* 。

32.1.1 zipimporter 对象

zipimporter 是用于导入 ZIP 文件的类。

class zipimport.**zipimporter**(*archivepath*)

创建新的 zipimporter 实例。*archivepath* 必须是指向 ZIP 文件的路径，或者 ZIP 文件中的特定路径。例如，`foo/bar.zip/lib` 的 *archivepath* 将在 ZIP 文件 `foo/bar.zip` 中的 `lib` 目录中查找模块（只要它存在）。

如果 *archivepath* 没有指向一个有效的 ZIP 档案，引发*ZipImportError* 。

在 3.12 版本发生变更: 在 3.10 中已弃用的 `find_loader()` 和 `find_module()` 方法现在已被移除。请改用*find_spec()*。

create_module(*spec*)

返回*None* 来显式地请求默认语义的*importlib.abc.Loader.create_module()* 实现。

Added in version 3.10.

exec_module(*module*)

importlib.abc.Loader.exec_module() 的实现。

Added in version 3.10.

find_spec(*fullname*, *target=None*)

importlib.abc.PathEntryFinder.find_spec() 的实现。

Added in version 3.10.

get_code(*fullname*)

返回指定模块的代码对象。如果模块无法被导入则引发*ZipImportError*。

get_data(*pathname*)

返回与 *pathname* 相关联的数据。如果不能找到文件则引发*OSError* 错误。

在 3.3 版本发生变更: 过去触发的*IOError*，现在是*OSError* 的别名。

get_filename(*fullname*)

返回如果指定模块被导入则应当要设置的 `__file__` 值。如果模块无法被导入则引发*ZipImportError*。

Added in version 3.1.

get_source(*fullname*)

返回指定模块的源代码。如果没有找到模块则引发*ZipImportError* ，如果档案包含模块但是没有源代码，返回*None* 。

is_package(*fullname*)

如果由 *fullname* 指定的模块是一个包则返回 `True` 。如果不能找到模块则引发*ZipImportError* 错误。

load_module(*fullname*)

导入由 *fullname* 所指定的模块。*fullname* 必须为（带点号的）完整限定名称。成功时返回导入的模块，失败时引发*ZipImportError*。

自 3.10 版本弃用: 使用*exec_module()* 来代替。

invalidate_caches()

清除在 ZIP 归档文件中找到的相关文件信息的内部缓存。

Added in version 3.10.

archive

导入器关联的 ZIP 文件的文件名，没有可能的子路径。

prefix

ZIP 文件中搜索的模块的子路径。这是一个指向 ZIP 文件根目录的 zipimporter 对象的空字符串。

当与斜杠结合使用时，*archive* 和*prefix* 属性等价于赋予*zipimporter* 构造器的原始 *archivepath* 参数。

32.1.2 例子

这是一个从 ZIP 档案中导入模块的例子 - 请注意*zipimport* 模块不需要明确地使用。

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # 将 .zip 文件添加到 path 的开头
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

32.2 pkgutil --- 包扩展工具

源代码: Lib/pkgutil.py

该模块为导入系统提供了工具，尤其是在包支持方面。

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)

一个包含模块信息的简短摘要的命名元组。

Added in version 3.6.

pkgutil.**extend_path**(*path*, *name*)

扩展组成包的模块的搜索路径。预期用途是将以下代码放到包的 __init__.py 中:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

对于 For each directory on *sys.path* 中每个具有与该包名称相匹配的子目录的目录，将该子目录添加到包的 __path__。这在需要将单个逻辑包的不同部分拆分为多个目录的情况下很有用处。

它还会查找开头部分 * 匹配 *name* 参数的 *.pkg 文件。此特性与 *.pth 文件类似（参阅*site* 模块了解更多信息），区别在于它不会区别对待以 import 开头的行。*.pkg 文件将按外在值确定是否被信任：除了跳过空行和忽略注释，在 *.pkg 文件中找到的所有条目都会被添加到路径中，无论它们是否存在于文件系统中（这是个有意为之的特性）。

如果输入路径不是一个列表（已冻结包就是这种情况）则它将被原样返回。输入路径不会被修改；将返回一个扩展的副本。条目将被添加到副本的末尾。

sys.path 会被假定为一个序列。*sys.path* 中的条目如果不是指向现有目录的字符串则会被忽略。*sys.path* 上当用作文件名时会导致错误的 Unicode 条目可以会使得此函数引发异常（与*os.path.isdir()* 的行为一致）。

pkgutil.**find_loader**(*fullname*)

为给定的 *fullname* 获取一个模块*loader*。

这是针对*importlib.util.find_spec()* 的向下兼容包装器，它将大多数失败转换为*ImportError* 并且只返回加载器而不是完整的*importlib.machinery.ModuleSpec*。

在 3.3 版本发生变更: 更新为直接基于*importlib* 而不是依赖于包内部的 **PEP 302** 导入模拟。

在 3.4 版本发生变更: 更新为基于 **PEP 451**

Deprecated since version 3.12, will be removed in version 3.14: 使用*importlib.util.find_spec()* 来代替。

pkgutil.**get_importer**(*path_item*)

为给定的 *path_item* 获取一个*finder*。

返回的查找器如果是由一个路径钩子新建的则会被缓存至*sys.path_importer_cache*。

如果需要重新扫描*sys.path_hooks* 则缓存（或其一部分）可以被手动清空。

在 3.3 版本发生变更: 更新为直接基于*importlib* 而不是依赖于包内部的 **PEP 302** 导入模拟。

pkgutil.**get_loader**(*module_or_name*)

为 *module_or_name* 获取一个*loader*。

如果模块或包可通过正常导入机制来访问，则会返回该机制相关部分的包装器。如果模块无法找到或导入则返回 `None`。如果指定的模块尚未被导入，则包含它的包（如果存在）会被导入，以便建立包 `__path__`。

在 3.3 版本发生变更: 更新为直接基于*importlib* 而不是依赖于包内部的 **PEP 302** 导入模拟。

在 3.4 版本发生变更: 更新为基于 **PEP 451**

Deprecated since version 3.12, will be removed in version 3.14: 使用*importlib.util.find_spec()* 来代替。

pkgutil.**iter_importers**(*fullname=''*)

为给定的模块名称产生*finder* 对象。

如果完整名称包含一个 `'.'`，查找器将针对包含该完整名称的包，否则它们将被注册为最高层级查找器（即同时用于*sys.meta_path* 和*sys.path_hooks*）。

如果指定的模块位于一个包内，则该包会作为唤起此函数的附带影响被导入。

如果未指定模块名称，则会产生所有的最高层级查找器。

在 3.3 版本发生变更: 更新为直接基于*importlib* 而不是依赖于包内部的 **PEP 302** 导入模拟。

pkgutil.**iter_modules**(*path=None*, *prefix=''*)

为 *path* 上的所有子模块产生*ModuleInfo*，或者如果 *path* 为 `None`，则为*sys.path* 上的所有最高层级模块产生。

path 应当为 `None` 或一个作为查找模块目标的路径的列表。

prefix 是要在输出时输出到每个模块名称之前的字符串。

> **备注**
>
> 只适用于定义了 `iter_modules()` 方法的*finder*。该接口是非标准的，因此本模块还提供了针对*importlib.machinery.FileFinder* 和*zipimport.zipimporter* 的实现。

在 3.3 版本发生变更: 更新为直接基于*importlib* 而不是依赖于包内部的 **PEP 302** 导入模拟。

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)

在 *path* 上递归地为所有模块产生*ModuleInfo*，或者如果 *path* 为 None，则为所有可访问的模块产生。

path 应当为 None 或一个作为查找模块目标的路径的列表。

prefix 是要在输出时输出到每个模块名称之前的字符串。

请注意此函数必须导入给定 *path* 上所有的 *packages* (而不是所有的模块！)，以便能访问 __path__ 属性来查找子模块。

onerror 是在当试图导入包如果发生任何异常则将附带一个参数（被导入的包的名称）被调用的函数。如果没有提供 *onerror* 函数，则*ImportError* 会被捕获并被忽略，而其他异常则会被传播，导致模块搜索的终结。

示例:

```
# 列出 python 可访问的所有模块
walk_packages()

# 列出 ctypes 的所有子模块
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

备注

只适用于定义了 iter_modules() 方法的*finder*。该接口是非标准的，因此本模块还提供了针对*importlib.machinery.FileFinder* 和*zipimport.zipimporter* 的实现。

在 3.3 版本发生变更: 更新为直接基于*importlib* 而不是依赖于包内部的 **PEP 302** 导入模拟。

pkgutil.**get_data**(*package*, *resource*)

从包中获取一个资源。

这是一个针对*loader* *get_data* API 的包装器。*package* 参数应为一个标准模块格式 (foo.bar) 的包名称。*resource* 参数应为相对路径文件名的形式，使用 / 作为路径分隔符。父目录名 ..，以及根目录名 (以 / 打头) 均不允许使用。

返回指定资源内容的二进制串。

对于位于文件系统中，已经被导入的包来说，这大致等价于:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

如果指定的包无法被定位或加载，或者如果它使用了不支持*get_data* 的*loader*，则将返回 None。特别地，针对*命名空间包* 的*loader* 不支持*get_data*。

pkgutil.**resolve_name**(*name*)

将一个名称解析为对象。

此功能被用在标准库的许多地方 (参见 bpo-12915) ——并且等价的功能也被广泛用于第三方包例如 setuptools, Django 和 Pyramid。

预期 *name* 将为以下格式之一，其中 W 是一个有效的 Python 标识符的缩写而点号表示这些伪正则表达式中的句点字面值:

- W(.W)*
- W(.W)*:(W(.W)*)?

第一种形式只是为了保持向下兼容性。它假定带点号名称的某一部分是包，而其余部分则是该包内部的一个对象，并可能嵌套在其他对象之内。因为包和对象层级结构之间的分界点无法通过观察来确定，所以使用这种形式必须重复尝试导入。

在第二种形式中，调用方通过提供一个单独冒号来明确分界点：冒号左边的带点号名称是要导入的包，而冒号右边的带点号名称则是对象层级结构。使用这种形式只需要导入一次。如果它以冒号结尾，则将返回一个模块对象。

此函数将返回一个对象（可能为模块），或是引发下列异常之一:

ValueError -- 如果 *name* 不为可识别的格式。

ImportError -- 如果导入本应成功但却失败。

AttributeError -- 当在遍历所导入包的对象层级结构以获取想要的对象时遭遇失败。

Added in version 3.9.

32.3 modulefinder --- 查找脚本使用的模块

源码： Lib/modulefinder.py

该模块提供了一个*ModuleFinder* 类，可用于确定脚本导入的模块集。modulefinder.py 也可以作为脚本运行，给出 Python 脚本的文件名作为参数，之后将打印导入模块的报告。

modulefinder.**AddPackagePath**(*pkg_name*, *path*)

记录名为 *pkg_name* 的包可以在指定的 *path* 中找到。

modulefinder.**ReplacePackage**(*oldname*, *newname*)

允许指定名为 *oldname* 的模块实际上是名为 *newname* 的包。

class modulefinder.**ModuleFinder**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)

该类提供*run_script()* 和*report()* 方法，用于确定脚本导入的模块集。*path* 可以是搜索模块的目录列表；如果没有指定，则使用 sys.path 。*debug* 设置调试级别；更高的值使类打印调试消息，关于它正在做什么。*excludes* 是要从分析中排除的模块名称列表。*replace_paths* 是将在模块路径中替换的 (oldpath, newpath) 元组的列表。

report()

将报告打印到标准输出，列出脚本导入的模块及其路径，以及缺少或似乎缺失的模块。

run_script(*pathname*)

分析 *pathname* 文件的内容，该文件必须包含 Python 代码。

modules

一个将模块名称映射到模块的字典。请参阅*ModuleFinder 的示例用法*。

32.3.1 ModuleFinder 的示例用法

稍后将分析的脚本（bacon.py）：

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

将输出 bacon.py 报告的脚本:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

输出样例（可能因架构而异）：

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
re._compiler:  isstring,_sre,_optimize_unicode
_sre:
re._constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
re._parser:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
----------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

32.4 runpy --- 查找并执行 Python 模块

源代码： Lib/runpy.py

---

runpy 模块用于找到并运行 Python 的模块，而无需首先导入。主要用于实现 `-m` 命令行开关，以允许用 Python 模块命名空间而不是文件系统来定位脚本。

请注意，这 *并非*一个沙盒模块——所有代码都在当前进程中运行，所有副作用（如其他模块对导入操作进行了缓存）在函数返回后都会留存。

此外，在*runpy* 函数返回后，任何由已执行代码定义的函数和类都不能保证正确工作。如果某使用场景不能接收此限制，那么选用*importlib* 可能更合适些。

runpy 模块提供两个函数：

runpy.**run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

执行给定模块的代码并返回模块的全局 globals 字典作为结果。首先会使用标准的导入机制来定位该模块的代码（请参阅 **PEP 302** 了解详情）然后在新的模块命令空间中执行。

mod_name 参数应当是一个绝对模块名。如果模块名指向一个包而非普通模块，则会导入这个包然后执行这个包中的*__main__* 子模块再返回模块全局字典。

可选的字典参数 *init_globals* 可用来在代码执行前预填充模块的 globals 字典。*init_globals* 不会被修改。如果在 *init_globals* 中定义了下面的任何一个特殊全局变量，这些定义都会被*run_module()* 覆盖。

特殊全局变量 __name__, __spec__, __file__, __cached__, __loader__ and __package__ 会在模块代码被执行前在 globals 字典中设置。(请注意这是一个最小化的变量集合——作为解释器的实现细节其他变量有可能被隐式地设置。)

若可选参数 __name__ 不为*None* 则设为 *run_name*，若此名称的模块是一个包则设为 `mod_name + '.__main__'`，否则设为 *mod_name* 参数。

__spec__ 将针对 实际导入的模块进行适当的设置 (也就是说，__spec__.name 将始终为 *mod_name* 或 `mod_name + '.__main__'`，而不是 *run_name*)。

__file__ 、__cached__、__loader__ 和 __package__ 根据模块规格进行 常规设置

如果给出了参数 *alter_sys* 并且值为*True*，那么 `sys.argv[0]` 将被更新为 __file__ 的值，`sys.modules[__name__]` 将被更新为临时模块对象。在函数返回前，`sys.argv[0]` 和 `sys.modules[__name__]` 将会复原。

请注意对*sys* 的这种操作不是线程安全的。其他线程可能会看到部分初始化的模块，以及更改后的参数列表。建议当从线程中的代码调用此函数时不要使用 sys 模块。

参见

`-m` 选项由命令行提供相同功能。

在 3.1 版本发生变更: 增加了通过查找*__main__* 子模块来执行包的功能。

在 3.2 版本发生变更: 加入了 __cached__ 全局变量（参见 **PEP 3147**)。

在 3.4 版本发生变更: 充分利用 **PEP 451** 加入的模块规格功能。使得以这种方式运行的模块能够正确设置 __cached__，并确保真正的模块名称总是可以通过 __spec__.name 的形式访问。

在 3.12 版本发生变更: __cached__, __loader__ 和 __package__ 的设置已被弃用。替代设置参见*ModuleSpec*。

runpy.**run_path**(*path_name*, *init_globals=None*, *run_name=None*)

执行位于指定文件系统位置上的代码并返回模块的 globals 字典作为结果。与提供给 CPython 命令行的脚本名称一样，*file_path* 可以指向一个 Python 源文件、编译后的字节码文件或包含*__main__* 模块的有效*sys.path* 条目（例如一个包含最高层级 __main__.py 文件的 zip 文件)。

对于简单的脚本而言，只需在新的模块命名空间中执行指定的代码即可。对于一个有效的*sys.path* 条目（通常是一个 zip 文件或目录)，首先会将该条目添加到 sys.path 的开头。然后函数会使用更新后的路径查找并执行*__main__* 模块。请注意如果在指定的位置上没有 __main__ 模块那么在唤起位于 sys.path 中其他位置上的现有条目时也不会受到特殊保护。

可选的字典参数 *init_globals* 可用来在代码执行前预填充模块的 globals 字典。*init_globals* 不会被修改。如果在 *init_globals* 中定义了下面的任何一个特殊全局变量，这些定义都会被*run_path()* 覆盖。

特殊全局变量 __name__, __spec__, __file__, __cached__, __loader__ and __package__ 会在模块代码被执行前在 globals 字典中设置。(请注意这是一个最小化的变量集合——作为解释器的实现细节其他变量有可能被隐式地设置。)

如果该可选参数不为*None*，则 __name__ 被设为 *run_name*，否则为 `'<run_path>'`。

如果 *file_path* 直接指向一个脚本文件（无论是源码还是预编译的字节码)，则 __file__ 将被设为 *file_path*，而 __spec__, __cached__, __loader__ 和 __package__ 都将被设为*None*。

如果 *file_path* 是对一个有效*sys.path* 条目的引用，则 __spec__ 将针对导入的*__main__* 模块进行相应设置 (也就是说，__spec__.name 将始终为 __main__)。__file__, __cached__, __loader__ 和 __package__ 将根据模块规格说明 正常设置。

sys 模块也进行了多项改动。首先，*sys.path* 可能会有如上文所描述的调整，`sys.argv[0]` 会使用 *file_path* 的值进行更新而 `sys.modules[__name__]` 会使用对应于被执行模块的临时模块对象进行更新。在函数返回之前对*sys* 中条目的所有修改都会被复原。

请注意，与*run_module()* 不同，对*sys* 的修改在本函数中不是可选项，因为这些调整对于允许执行*sys.path* 条目来说是至关重要的。由于线程安全限制仍然适用，在线程代码中使用该函数应当使用导入锁进行序列化，或是委托给单独的进程。

参见

using-on-interface-options 用于在命令行上实现同等功能（`python path/to/script`）。

Added in version 3.2.

在 3.4 版本发生变更: 进行更新以便利用 **PEP 451** 加入的模块规格特性。这允许在 `__main__` 是从有效的*sys.path* 条目导入而不是直接执行的情况下能够正确地设置 `__cached__`。

在 3.12 版本发生变更: `__cached__`, `__loader__` 和 `__package__` 已被弃用。

参见

PEP 338 -- 将模块作为脚本执行
PEP 由 Nick Coghlan 撰写并实现。

PEP 366 ——主模块的显式相对导入
PEP 由 Nick Coghlan 撰写并实现。

PEP 451 ——导入系统采用的 ModuleSpec 类型
PEP 由 Eric Snow 撰写并实现。

using-on-general ——CPython 命令行详解

importlib.import_module() 函数

32.5 `importlib` --- `import` 的实现

Added in version 3.1.

源代码 Lib/importlib/__init__.py

32.5.1 概述

importlib 包具有三重目标。

一是在 Python 源代码中提供 `import` 语句的实现（并且因此而扩展*__import__()* 函数）。这提供了一个可移植到任何 Python 解释器的 `import` 实现。与使用 Python 以外的编程语言实现的方式相比这一实现也更易于理解。

第二个目的是实现 `import` 的部分被公开在这个包中，使得用户更容易创建他们自己的自定义对象 (通常被称为*importer*) 来参与到导入过程中。

三，这个包也包含了对外公开用于管理 Python 包的各个方面的附加功能的模块:

- *importlib.metadata* 代表对来自第三方发行版的元数据的访问。
- *importlib.resources* 提供了用于对来自 Python 包的非代码“资源”的访问的例程。

参见

import
`import` 语句的语言参考

包规格说明
包的初始规范。自从编写这个文档开始，一些语义已经发生改变了（比如基于*sys.modules* 中 None 的重定向）。

***__import__()* 函数**
import 语句是这个函数的语法糖。

***sys.path* 模块搜索路径的初始化**
sys.path 的初始化。

PEP 235
在忽略大小写的平台上进行导入

PEP 263
定义 Python 源代码编码

PEP 302
新导入钩子

PEP 328
导入：多行和绝对/相对

PEP 366
主模块显式相对导入

PEP 420
隐式命名空间包

PEP 451
导入系统的一个模块规范类型

PEP 488
消除 PYO 文件

PEP 489
多阶段扩展模块初始化

PEP 552
确定性的 pyc 文件

PEP 3120
使用 UTF-8 作为默认的源编码

PEP 3147
PYC 仓库目录

32.5.2 函数

importlib.**__import__**(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
内置*__import__()* 函数的实现。

> **备注**
> 程序式地导入模块应该使用*import_module()* 而不是这个函数。

importlib.**import_module**(*name*, *package=None*)
导入一个模块。参数 *name* 指定了以绝对或相对导入方式导入什么模块 (比如要么像这样 pkg.mod 或者这样 ..mod)。如果参数 name 使用相对导入的方式来指定，那么 *package* 参数必须设置为那个包名，这个包名作为解析这个包名的锚点 (比如 import_module('..mod', 'pkg.subpkg') 将会导入 pkg.mod)。

import_module() 函数是一个对*importlib.__import__()* 进行简化的包装器。这意味着该函数的所有语义都来自于*importlib.__import__()*。这两个函数之间最重要的不同点在

于*import_module()* 返回指定的包或模块 (例如 pkg.mod)，而*__import__()* 返回最高层级的包或模块 (例如 pkg)。

如果动态导入一个自解释器开始执行以来被创建的模块（即创建了一个 Python 源代码文件），为了让导入系统知道这个新模块，可能需要调用*invalidate_caches()*。

在 3.3 版本发生变更: 父包会被自动导入。

importlib.**invalidate_caches**()

使查找器存储在*sys.meta_path* 中的内部缓存无效。如果一个查找器实现了 invalidate_caches()，那么它会被调用来执行那个无效过程。如果创建/安装任何模块，同时正在运行的程序是为了保证所有的查找器知道新模块的存在，那么应该调用这个函数。

Added in version 3.3.

在 3.10 版本发生变更: 当注意到相同命名空间已被导入之后在不同*sys.path* 位置中创建/安装的命名空间包。

importlib.**reload**(*module*)

重新加载之前导入的 *module*。那个参数必须是一个模块对象，所以它之前必须已经成功导入了。这在你已经使用外部编辑器编辑过了那个模块的源代码文件并且想在退出 Python 解释器之前试验这个新版本的模块的时候将很适用。函数的返回值是那个模块对象（如果重新导入导致一个不同的对象放置在*sys.modules* 中，那么那个模块对象是有可能会不同）。

当执行*reload()* 的时候：

- Python 模块的代码会被重新编译并且那个模块级的代码被重新执行，通过重新使用一开始加载那个模块的*loader*，定义一个新的绑定在那个模块字典中的名称的对象集合。扩展模块的 init 函数不会被调用第二次。
- 与 Python 中的所有的其它对象一样，旧的对象只有在它们的引用计数为 0 之后才会被回收。
- 模块命名空间中的名称重新指向任何新的或更改后的对象。
- 其他旧对象的引用（例如那个模块的外部名称）不会被重新绑定到引用的新对象的，并且如果有需要，必须在出现的每个命名空间中进行更新。

有一些其他注意事项：

当一个模块被重新加载的时候，它的字典（包含了那个模块的全区变量）会被保留。名称的重新定义会覆盖旧的定义，所以通常来说这不是问题。如果一个新模块没有定义在旧版本模块中定义的名称，则将保留旧版本中的定义。这一特性可用于作为那个模块的优点，如果它维护一个全局表或者对象的缓存——使用 try 语句，就可以测试表的存在并且跳过它的初始化，如果有需要的话:

```
try:
    cache
except NameError:
    cache = {}
```

重新加载内置的或者动态加载模块，通常来说不是很有用处。不推荐重新加载"*sys*，*__main__*，*builtins* 和其它关键模块。在很多例子中，扩展模块并不是设计为不止一次的初始化，并且当重新加载时，可能会以任意方式失败。

如果一个模块使用 from ... import ... 导入的对象来自另外一个模块，给其它模块调用*reload()* 不会重新定义来自这个模块的对象——解决这个问题的一种方式是重新执行 from 语句，另一种方式是使用 import 和限定名称 (*module.name*) 来代替。

如果一个模块创建一个类的实例，重新加载定义那个类的模块不影响那些实例的方法定义——它们继续使用旧类中的定义。对于子类来说同样是正确的。

Added in version 3.4.

在 3.7 版本发生变更: 如果重新加载的模块缺少*ModuleSpec* ，则会触发*ModuleNotFoundError* 。

32.5.3 importlib.abc ——关于导入的抽象基类

源代码： Lib/importlib/abc.py

---

importlib.abc 模块包含了 import 使用到的所有核心抽象基类。在实现核心的 ABCs 中，核心抽象基类的一些子类也提供了帮助。

ABC 类的层次结构：

```
object
 +-- MetaPathFinder
 +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**MetaPathFinder**

一个代表*meta path finder* 的抽象基类。

Added in version 3.3.

在 3.10 版本发生变更: 不再是 Finder 的子类。

find_spec(*fullname*, *path*, *target=None*)

一个用于查找指定模块的*spec* 的抽象方法。如果这是最高层级的导入，*path* 将为 None。在其他情况下，这将是对子包或模块的搜索而 *path* 将是来自上级包的 __path__ 值。如果找不到 spec，则返回 None。在传入时，target 是一个被查找器用来对返回的 spec 进行更有依据的猜测的模块对象。在实现具体 MetaPathFinders 的时候*importlib.util.spec_from_loader()* 将会很有用处。

Added in version 3.4.

invalidate_caches()

当被调用的时候，一个可选的方法应该将查找器使用的任何内部缓存进行无效。将在*sys.meta_path* 上的所有查找器的缓存进行无效的时候，这个函数被*importlib.invalidate_caches()* 所使用。

在 3.4 版本发生变更: 当被调用时将返回 None 而不是*NotImplemented*。

class importlib.abc.**PathEntryFinder**

一个抽象基类，代表*path entry finder*。虽然与*MetaPathFinder* 有些相似之处，但 *PathEntryFinder* 仅用于*importlib.machinery.PathFinder* 提供的基于路径的导入子系统中。

Added in version 3.3.

在 3.10 版本发生变更: 不再是 Finder 的子类。

find_spec(*fullname*, *target=None*)

一个抽象方法，用于查找指定模块的*spec*。搜索器将只在指定的*path entry* 内搜索该模块。找不到则会返回 None。在实现具体的 PathEntryFinders 代码时，可能会用到*importlib.util.spec_from_loader()* 。

Added in version 3.4.

invalidate_caches()

可选方法，调用后应让查找器用到的所有内部缓存失效。要让所有缓存的查找器的缓存无效时，可供*importlib.machinery.PathFinder.invalidate_caches()* 调用。

class importlib.abc.**Loader**

loader 的抽象基类。关于一个加载器的实际定义请查看 **PEP 302**。

想要支持资源读取的加载器应当实现*importlib.resources.abc.ResourceReader* 所规定的 `get_resource_reader()` 方法。

在 3.7 版本发生变更: 引入了可选的 `get_resource_reader()` 方法。

create_module(*spec*)

当导入一个模块的时候，一个返回将要使用的那个模块对象的方法。这个方法可能返回 `None`，这暗示着应该发生默认的模块创建语义。”

Added in version 3.4.

在 3.6 版本发生变更: 当*exec_module()* 已定义时此方法将不再是可选项。

exec_module(*module*)

当一个模块被导入或重新加载时在自己的命名空间中执行该模块的的抽象方法。该模块在*exec_module()* 被调用时应该已经被初始化了。当此方法存在时，必须要定义*create_module()*。

Added in version 3.4.

在 3.6 版本发生变更: *create_module()* 也必须要定义。

load_module(*fullname*)

用于加载模块的传统方法。如果模块无法被导入，则会引发*ImportError*，在其他情况下将返回被加载的模块。

如果请求的模块已存在于*sys.modules* 中，则该模块应当被使用并重新加载。在其他情况下加载器应当创建一个新模块并在任何加载操作开始之前将其插入到*sys.modules* 中，以防止来自导入的无限递归。如果加载器插入了一个模块并且加载失败，则必须用加载器将其从*sys.modules* 中移除；在加载器开始执行之前已经存在于*sys.modules* 中的模块应当保持原样。

加载器应当在模块上设置几个属性（请注意在模块被重新加载时这些属性有几个可能发生改变）：

- `module.__name__`
- `module.__file__`
- `module.__cached__` *(已弃用)*
- `module.__path__`
- `module.__package__` *(已弃用)*
- `module.__loader__` *(已弃用)*

当*exec_module()* 可用的时候，那么则提供了向后兼容的功能。

在 3.4 版本发生变更: 当被调用时将引发*ImportError* 而不是*NotImplementedError*。在*exec_module()* 可用时提供的功能。

Deprecated since version 3.4, will be removed in version 3.15: 用于加载模块的推荐 API 是*exec_module()* (和*create_module()*)。加载器应该实现它而不是*load_module()*。当实现了*exec_module()* 时导入机制将会承担*load_module()* 的所有其他责任。

class importlib.abc.**ResourceLoader**

被 *TraversableResources* 取代

一个*loader* 的抽象基类，它实现了可选的 **PEP 302** 协议用于从存储后端加载任意资源。

自 3.7 版本弃用: 这个 ABC 已被弃用并转为通过*importlib.resources.abc.TraversableResources* 来支持资源加载。

abstractmethod get_data(*path*)

一个返回用来表示位于 *path* 的数据的字节串的抽象方法。具有允许存储任意数据的文件型存储后端的加载器能够实现这个抽象方法来直接访问被存储的数据。如果 *path* 无法找到则会引发*OSError*。*path* 应当使用一个模块的 module's `__file__` 属性或来自某个包的 `__path__` 中的条目来构造。

在 3.4 版本发生变更: 引发*OSError* 异常而不是*NotImplementedError* 异常。

class importlib.abc.**InspectLoader**

一个实现加载器检查模块可选的 **PEP 302** 协议的*loader* 的抽象基类。

get_code(*fullname*)

返回一个模块的代码对象，或如果模块没有一个代码对象（例如，对于内置的模块来说，这会是这种情况），则为 `None`。如果加载器不能找到请求的模块，则引发*ImportError* 异常。

> **备注**
>
> 当这个方法有一个默认的实现的时候，出于性能方面的考虑，如果有可能的话，建议覆盖它。

在 3.4 版本发生变更: 不再抽象并且提供一个具体的实现。

abstractmethod get_source(*fullname*)

一个返回模块源的抽象方法。使用*universal newlines* 作为文本字符串被返回，将所有可识别行分割符翻译成 `'\n'` 字符。如果没有可用的源（例如，一个内置模块），则返回 `None`。如果加载器不能找到指定的模块，则引发*ImportError* 异常。

在 3.4 版本发生变更: 引发*ImportError* 而不是*NotImplementedError*。

is_package(*fullname*)

可选方法，如果模块为包，则返回 True，否则返回 False。如果*loader* 找不到模块，则会触发*ImportError*。

在 3.4 版本发生变更: 引发*ImportError* 而不是*NotImplementedError*。

static source_to_code(*data*, *path='<string>'*)

创建一个来自 Python 源码的代码对象。

参数 *data* 可以是任意*compile()* 函数支持的类型（例如字符串或字节串）。参数 *path* 应该是源代码来源的路径，这可能是一个抽象概念（例如位于一个 zip 文件中）。

在有后续代码对象的情况下，可以在一个模块中通过运行 `exec(code, module.__dict__)` 来执行它。

Added in version 3.4.

在 3.5 版本发生变更: 使得这个方法变成静态的。

exec_module(*module*)

Loader.exec_module() 的实现。

Added in version 3.4.

load_module(*fullname*)

Loader.load_module() 的实现。

Deprecated since version 3.4, will be removed in version 3.15: 使用*exec_module()* 来代替。

class importlib.abc.**ExecutionLoader**

一个继承自*InspectLoader* 的抽象基类，当被实现时，帮助一个模块作为脚本来执行。这个抽象基类表示可选的 **PEP 302** 协议。

abstractmethod get_filename(*fullname*)

一个用于为指定模块返回 __file__ 的值的抽象方法。如果没有可用的路径，则会引发*ImportError*。

如果源代码可用，那么这个方法返回源文件的路径，不管是否是用来加载模块的字节码。

在 3.4 版本发生变更: 引发*ImportError* 而不是*NotImplementedError*。

class importlib.abc.**FileLoader**(*fullname*, *path*)

一个继承自*ResourceLoader* 和*ExecutionLoader*，提供*ResourceLoader.get_data()* 和*ExecutionLoader.get_filename()* 具体实现的抽象基类。

参数 *fullname* 是加载器要处理的模块的完全解析的名字。参数 *path* 是模块文件的路径。

Added in version 3.3.

name

加载器可以处理的模块的名字。

path

模块的文件路径

load_module(*fullname*)

调用 super 的 load_module()。

Deprecated since version 3.4, will be removed in version 3.15: 使用*Loader.exec_module()* 来代替。

abstractmethod get_filename(*fullname*)

返回*path*。

abstractmethod get_data(*path*)

读取 *path* 作为二进制文件并且返回来自它的字节数据。

class importlib.abc.**SourceLoader**

一个用于实现源文件（和可选地字节码）加载的抽象基类。这个类继承自*ResourceLoader* 和*ExecutionLoader*，需要实现：

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()***
  应该是只返回源文件的路径；不支持无源加载。

由这个类定义的抽象方法用来添加可选的字节码文件支持。不实现这些可选的方法（或导致它们引发*NotImplementedError* 异常）导致这个加载器只能与源代码一起工作。实现这些方法允许加载器能与源 和字节码文件一起工作。不允许只提供字节码的 无源式加载。字节码文件是通过移除 Python 编译器的解析步骤来加速加载的优化，并且因此没有开放出字节码专用的 API。

path_stats(*path*)

返回一个包含关于指定路径的元数据的*dict* 的可选的抽象方法。支持的字典键有：

- 'mtime' (必选项): 一个表示源码修改时间的整数或浮点数；
- 'size' (可选项)：源码的字节大小。

字典中任何其他键会被忽略，以允许将来的扩展。如果不能处理该路径，则会引发*OSError*。

Added in version 3.3.

在 3.4 版本发生变更: 引发*OSError* 而不是*NotImplemented*。

path_mtime(*path*)

返回指定文件路径修改时间的可选的抽象方法。

自 3.3 版本弃用: 在有了*path_stats()* 的情况下，这个方法被弃用了。没必要去实现它了，但是为了兼容性，它依然处于可用状态。如果文件路径不能被处理，则引发*OSError* 异常。

在 3.4 版本发生变更: 引发*OSError* 而不是*NotImplemented*。

set_data(*path*, *data*)

往一个文件路径写入指定字节的的可选的抽象方法。任何中间不存在的目录不会被自动创建。

当对路径的写入因路径为只读而失败时 (*errno.EACCES*/*PermissionError*)，不会传播异常。

在 3.4 版本发生变更: 当被调用时，不再引起*NotImplementedError* 异常。

get_code(*fullname*)

InspectLoader.get_code() 的具体实现。

exec_module(*module*)

Loader.exec_module() 的具体实现。

Added in version 3.4.

load_module(*fullname*)

Concrete implementation of *Loader.load_module()*.

Deprecated since version 3.4, will be removed in version 3.15: 使用*exec_module()* 来代替。

get_source(*fullname*)

InspectLoader.get_source() 的具体实现。

is_package(*fullname*)

InspectLoader.is_package() 的具体实现。一个模块被确定为一个包的条件是：它的文件路径（由*ExecutionLoader.get_filename()* 提供）当文件扩展名被移除时是一个命名为 __init__ 的文件，**并且**这个模块名字本身不是以 __init__ 结束。

class importlib.abc.**ResourceReader**

被 *TraversableResources* 取代

提供读取 *resources* 能力的一个*abstract base class* 。

从这个 ABC 的视角出发，*resource* 指一个包附带的二进制文件。常见的如在包的 __init__.py 文件旁的数据文件。这个类存在的目的是为了将对数据文件的访问进行抽象，这样包就和其数据文件的存储方式无关了。不论这些文件是存放在一个 zip 文件里还是直接在文件系统内。

对于该类中的任一方法，*resource* 参数的值都需要是一个在概念上表示文件名称的*path-like object*。这意味着任何子目录的路径都不该出现在 *resouce* 参数值内。因为对于阅读器而言，包的位置就代表着「目录」。因此目录和文件名就分别对应于包和资源。这也是该类的实例都需要和一个包直接关联（而不是潜在指代很多包或者一整个模块）的原因。

想支持资源读取的加载器需要提供一个返回实现了此 ABC 的接口的 get_resource_reader(fullname) 方法。如果通过全名指定的模块不是一个包，这个方法应该返回*None*。当指定的模块是一个包时，应该只返回一个与这个抽象类 ABC 兼容的对象。

Added in version 3.7.

Deprecated since version 3.12, will be removed in version 3.14: 使用*importlib.resources.abc.TraversableResources* 代替。

abstractmethod open_resource(*resource*)

返回一个打开的*file-like object* 用于 *resource* 的二进制读取。

如果无法找到资源，将会引发*FileNotFoundError*。

abstractmethod resource_path(*resource*)

返回 *resource* 的文件系统路径。

如果资源并不实际存在于文件系统中，将会引发*FileNotFoundError*。

abstractmethod is_resource(*name*)

如果 *name* 被视作资源，则返回 True。如果 *name* 不存在，则引发*FileNotFoundError* 异常。

abstractmethod contents()

反回由字符串组成的*iterable*，表示这个包的所有内容。请注意并不要求迭代器返回的所有名称都是实际的资源，例如返回*is_resource()* 为假值的名称也是可接受的。

允许非资源名字被返回是为了允许存储的一个包和它的资源的方式是已知先验的并且非资源名字会有用的情况。比如，允许返回子目录名字，目的是当得知包和资源存储在文件系统上面的时候，能够直接使用子目录的名字。

这个抽象方法返回了一个不包含任何内容的可迭代对象。

class importlib.abc.**Traversable**

一个具有*pathlib.Path* 中方法的子集并适用于遍历目录和打开文件的对象。

对于该对象在文件系统中的表示形式，请使用*importlib.resources.as_file()*。

Added in version 3.9.

Deprecated since version 3.12, will be removed in version 3.14: 使用*importlib.resources.abc.Traversable* 代替。

name

抽象属性。此对象的不带任何父引用的基本名称。

abstractmethod iterdir()

产出 `self` 中的 `Traversable` 对象。

abstractmethod is_dir()

如果 `self` 是一个目录则返回 `True`。

abstractmethod is_file()

如果 `self` 是一个文件则返回 `True`。

abstractmethod joinpath(*child*)

返回 `self` 中的 Traversable 子对象。

abstractmethod __truediv__(*child*)

返回 `self` 中的 `Traversable` 子对象。

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode 可以为'r' 或'rb' 即以文本或二进制模式打开。返回一个适用于读取的句柄（与*pathlib.Path.open* 样同）。

当以文件模式打开时，接受与*io.TextIOWrapper* 所接受的相同编码格式形参。

read_bytes()

以字节串形式读取 `self` 的内容。

read_text(*encoding=None*)

以文本形式读取 `self` 的内容。

class importlib.abc.**TraversableResources**

针对能够为*importlib.resources.files()* 接口提供服务的资源读取器的抽象基类。子类化*importlib.resources.abc.ResourceReader* 并为*importlib.resources.abc.ResourceReader* 的抽象方法提供具体实现。因此，任何提供了*importlib.abc.TraversableResources* 的加载器也会提供 ResourceReader。

需要支持资源读取的加载器应实现此接口。

Added in version 3.9.

Deprecated since version 3.12, will be removed in version 3.14: 使用*importlib.resources.abc.TraversableResources* 代替。

abstractmethod files()

为载入的包返回一个*importlib.resources.abc.Traversable* 对象。

32.5.4 importlib.machinery ——导入器和路径钩子函数。

源代码： Lib/importlib/machinery.py

本模块包含多个对象，以帮助 import 查找并加载模块。

importlib.machinery.**SOURCE_SUFFIXES**

一个字符串列表，表示源模块的可识别的文件后缀。

Added in version 3.3.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**

一个字符串列表，表示未经优化字节码模块的文件后缀。

Added in version 3.3.

自 3.5 版本弃用: 改用*BYTECODE_SUFFIXES*。

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**

一个字符串列表，表示已优化字节码模块的文件后缀。

Added in version 3.3.

自 3.5 版本弃用: 改用*BYTECODE_SUFFIXES*。

importlib.machinery.**BYTECODE_SUFFIXES**

一个字符串列表，表示字节码模块的可识别的文件后缀（包含前导的句点符号）。

Added in version 3.3.

在 3.5 版本发生变更: 该值不再依赖于 __debug__ 。

importlib.machinery.**EXTENSION_SUFFIXES**

一个字符串列表，表示扩展模块的可识别的文件后缀。

Added in version 3.3.

importlib.machinery.**all_suffixes**()

返回字符串的组合列表，代表标准导入机制可识别模块的所有文件后缀。这是个助手函数，只需知道某个文件系统路径是否会指向模块，而不需要任何关于模块种类的细节（例如*inspect.getmodulename()*）。

Added in version 3.3.

class importlib.machinery.**BuiltinImporter**

用于导入内置模块的*importer*。所有已知的内置模块都已列入*sys.builtin_module_names*。此类实现了*importlib.abc.MetaPathFinder* 和*importlib.abc.InspectLoader* 抽象基类。

此类只定义类的方法，以减轻实例化的开销。

在 3.5 版本发生变更: 作为 **PEP 489** 的一部分，现在内置模块导入器实现了 Loader.create_module() 和 Loader.exec_module()。

class importlib.machinery.**FrozenImporter**

用于已冻结模块的*importer*。此类实现了*importlib.abc.MetaPathFinder* 和*importlib.abc.InspectLoader* 抽象基类。

此类只定义类的方法，以减轻实例化的开销。

在 3.4 版本发生变更: 有了 create_module() 和 exec_module() 方法。

class importlib.machinery.**WindowsRegistryFinder**

Finder 用于查找在 Windows 注册表中声明的模块。该类实现了基础的*importlib.abc.MetaPathFinder*。

此类只定义类的方法，以减轻实例化的开销。

Added in version 3.3.

自 3.6 版本弃用: 改用*site* 配置。未来版本的 Python 可能不会默认启用该查找器。

class importlib.machinery.**PathFinder**

用于*sys.path* 和包的 __path__ 属性的*Finder* 。该类实现了基础的*importlib.abc.MetaPathFinder*。

此类只定义类的方法，以减轻实例化的开销。

classmethod find_spec(*fullname*, *path=None*, *target=None*)

类方法试图在*sys.path* 或 *path* 上为 *fullname* 指定的模块查找*spec*。对于每个路径条目，都会查看*sys.path_importer_cache* 。如果找到非 False 的对象，则将其用作*path entry finder* 来查找要搜索的模块。如果在*sys.path_importer_cache* 中没有找到条目，那会在*sys.path_hooks* 检索该路径条目的查找器，找到了则和查到的模块信息一起存入*sys.path_importer_cache* 。如果查找器没有找到，则缓存中的查找器和模块信息都存为 None ，然后返回。

Added in version 3.4.

在 3.5 版本发生变更: 如果当前工作目录不再有效（用空字符串表示），则返回 None，但在*sys.path_importer_cache* 中不会有缓存值。

classmethod invalidate_caches()

为所有存于*sys.path_importer_cache* 中的查找器，调用其*importlib.abc.PathEntryFinder.invalidate_caches()* 方法。*sys.path_importer_cache* 中为 None 的条目将被删除。

在 3.7 版本发生变更: *sys.path_importer_cache* 中为 None 的条目将被删除。

在 3.4 版本发生变更: 调用*sys.path_hooks* 中的对象，当前工作目录为 '' (即空字符串)。

class importlib.machinery.**FileFinder**(*path*, **loader_details*)

importlib.abc.PathEntryFinder 的一个具体实现，它会缓存来自文件系统的结果。

参数 *path* 是查找器负责搜索的目录。

loader_details 参数是数量不定的二元组，每个元组包含加载器及其可识别的文件后缀列表。加载器应为可调用对象，可接受两个参数，即模块的名称和已找到文件的路径。

查找器将按需对目录内容进行缓存，通过对每个模块的检索进行状态统计，验证缓存是否过期。因为缓存的滞后性依赖于操作系统文件系统状态信息的粒度，所以搜索模块、新建文件、然后搜索新文件代表的模块，这会存在竞争状态。如果这些操作的频率太快，甚至小于状态统计的粒度，那么模块搜索将会失败。为了防止这种情况发生，在动态创建模块时，请确保调用*importlib.invalidate_caches()*。

Added in version 3.3.

path

查找器将要搜索的路径。

find_spec(*fullname*, *target=None*)

尝试在*path* 中找到处理 *fullname* 的规格。

Added in version 3.4.

invalidate_caches()

清理内部缓存。

classmethod path_hook(**loader_details*)

一个类方法，返回供*sys.path_hooks* 使用的闭包。使用直接提供给闭包的路径参数和间接提供的 *loader_details* 闭包将返回一个*FileFinder* 的实例。

如果给闭包的参数不是已存在的目录，将会触发*ImportError*。

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)

importlib.abc.SourceLoader 的一个具体实现，该实现子类化了*importlib.abc.FileLoader* 并提供了其他一些方法的具体实现。

Added in version 3.3.

name

该加载器将要处理的模块名称。

path

源文件的路径

is_package(*fullname*)

如果*path* 看似包的路径，则返回 `True`。

path_stats(*path*)

importlib.abc.SourceLoader.path_stats() 的具体代码实现。

set_data(*path*, *data*)

importlib.abc.SourceLoader.set_data() 的具体代码实现。

load_module(*name=None*)

importlib.abc.Loader.load_module() 的具体代码实现，这里要加载的模块名是可选的。

Deprecated since version 3.6, will be removed in version 3.15: 改用*importlib.abc.Loader.exec_module()* 。

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)

importlib.abc.FileLoader 的具体代码实现，可导入字节码文件（也即源代码文件不存在）。

请注意，直接用字节码文件（而不是源代码文件），会让模块无法应用于所有的 Python 版本或字节码格式有所改动的新版本 Python。

Added in version 3.3.

name

加载器将要处理的模块名。

path

二进制码文件的路径。

is_package(*fullname*)

根据*path* 确定该模块是否为包。

get_code(*fullname*)

返回由*path* 创建的*name* 的代码对象。

get_source(*fullname*)

因为用此加载器时字节码文件没有源码文件，所以返回 `None`。

load_module(*name=None*)

importlib.abc.Loader.load_module() 的具体代码实现，这里要加载的模块名是可选的。

Deprecated since version 3.6, will be removed in version 3.15: 改用*importlib.abc.Loader.exec_module()* 。

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)

importlib.abc.ExecutionLoader 的具体代码实现，用于扩展模块。

参数 *fullname* 指定了加载器要支持的模块名。参数 *path* 是指向扩展模块文件的路径。

请注意，在默认情况下，在子解释器中导入未实现多阶段初始化的扩展模块 (参见 **PEP 489**) 将会失败，即使在其他情况下能够成功导入。

Added in version 3.3.

在 3.12 版本发生变更: 在子解释器中使用时需要多阶段初始化。

name

装载器支持的模块名。

path

扩展模块的路径。

create_module(*spec*)

根据 **PEP 489** ，由给定规范创建模块对象。

Added in version 3.5.

exec_module(*module*)

根据 **PEP 489**，初始化给定的模块对象。

Added in version 3.5.

is_package(*fullname*)

如果文件路径基于*EXTENSION_SUFFIXES* 指向某个包的 __init__ 模块则返回 True。

get_code(*fullname*)

返回 None，因为扩展模块缺少代码对象。

get_source(*fullname*)

返回 None，因为扩展模块没有源代码。

get_filename(*fullname*)

返回*path*。

Added in version 3.4.

class importlib.machinery.**NamespaceLoader**(*name*, *path*, *path_finder*)

一个针对命名空间包的*importlib.abc.InspectLoader* 具体实现。这是一个私有类的别名，仅为在命名空间包上内省 __loader__ 属性而被设为公有:

```
>>> from importlib.machinery import NamespaceLoader
>>> import my_namespace
>>> isinstance(my_namespace.__loader__, NamespaceLoader)
True
>>> import importlib.abc
>>> isinstance(my_namespace.__loader__, importlib.abc.Loader)
True
```

Added in version 3.11.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, ***, *origin=None*, *loader_state=None*, *is_package=None*)

针对特定模块的导入系统相关状态的规格说明。这通常是作为模块的 __spec__ 属性对外公开。这类属性中有许多在模块上也直接可用：例如，module.__spec__.origin == module.__file__。但是请注意，虽然 值通常是相等的，但它们也可以因为两个对象之间没有进行同步而不相等。举例来说，有可能在运行时更新模块的 __file__ 而这将不会自动反映在模块的*__spec__.origin* 中，反之亦然。

Added in version 3.4.

name

模块的完整限定名称 (参见 `module.__name__`)。*finder* 应当总是将该属性设为一个非空字符串。

loader

用于加载模块的*loader* (参见 `module.__loader__`)。*finder* 应当总是设置该属性。

origin

应当被*loader* 用于加载模块的位置 (参见 `module.__file__`)。例如，对于从 `.py` 新闻公报加载的模块来说这将为文件名。*finder* 应当总是将此属性设为一个有意义的值以供*loader* 使用。在没有可用值的少数情况下（如命名空间包），它应当被设为 `None`。

submodule_search_locations

一个（可能为空的）枚举用于查找包中的子模块的位置的由字符串组成的*sequence* (参见 `module.__path__`)。在大多数时候该列表中都将只有一个目录。

finder 应当将此属性设为一个序列，甚至可以为空序列，以向导入系统提示该模块是一个包。对于非包模块它应当被设为 `None`。对于命名空间包它会在稍后被自动设为一个特殊对象。

loader_state

finder 可以将此属性设为一个包含额外的模块专属数据的对象供加载模块时使用。在其他情况下应将其设为 `None`。

cached

模块代码的已编译版本的文件名 (参见 `module.__cached__`)。*finder* 总是应当设置此属性但对于不需要存储已编译代码的模块来说它可以为 `None`。

parent

（只读）模块所在的包的完整限定名称（或者对于最高层级模块来说则空字符串）。参见 `module.__package__`。如果模块是一个包则它将与*name* 相同。

has_location

如果 spec 的*origin* 指向一个可加载的位置则为 `True`，否则为 `False`。该值将确定如何解读 `origin` 以及如何填充模块的 `__file__`。

class importlib.machinery.**AppleFrameworkLoader**(*name*, *path*)

importlib.machinery.ExtensionFileLoader 的一个专用版本，能以 Framework 格式加载扩展模块。

为了保持与 iOS App Store 的兼容性，在 iOS app 中的 所有二进制模块都必须为动态库，包含在具有适当元数据的框架中，保存于被打包 app 的 `Frameworks` 文件夹下。每个框架只能有一个二进制模块，而在 Frameworks 文件夹之外不能有可执行的二进制模块。

为满足此要求，当在 iOS 上运行时，扩展模块二进制代码 不会被打包为 `sys.path` 中的 `.so` 文件，而是作为单个的独立框架。要发现这些框架，此加载器必须针对 `.fwork` 文件扩展名进行注册，并以一个 `.fwork` 文件作为 `sys.path` 中二进制代码的原始位置上的占位符。`.fwork` 文件包含 `Frameworks` 文件夹中实际二进制文件相对于 app 包的路径。为允许将框架打包的二进制代码解析到原始位置上，框架应当包含一个 `.origin` 文件，其中包含 `.fwork` 文件相对于 app 包的位置。

例如，考虑导入 `from foo.bar import _whiz` 的情况，其中 `_whiz` 是使用二进制模块 `sources/foo/bar/_whiz.abi3.so` 实现的，这里 `sources` 是在 `sys.path` 中注册的相对于 app 包的位置。此模块 必须发布为 `Frameworks/foo.bar._whiz.framework/foo.bar._whiz` (根据模块的完整导入路径创建框架名称)，并通过 `.framework` 目录中的 `Info.plist` 文件将二进制文件标识为一个框架。`foo.bar._whiz` 模块在原始位置中以一个 `sources/foo/bar/_whiz.abi3.fwork` 标记文件来代表，其中包含路径 `Frameworks/foo.bar._whiz/foo.bar._whiz`。该框架还要包含 `Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin`，其中包含 `.fwork` 文件的路径。

当使用此加载器加载一个模块时，模块的 `__file__` 将被报告为 `.fwork` 文件的位置。这允许代码使用模块的 `__file__` 作为文件系统遍历的锚点。不过，原始规范说明将会引用 `.framework` 文件夹下 实际二进制文件的位置。binary in the

构建 app 的 Xcode 项目要负责将存在于 PYTHONPATH 中的任何 .so 文件转换为 Frameworks 文件夹下的框架（包括从模块文件获取扩展，框架元数据的添加，以及结果框架的签名），并创建 .fwork 和 .origin 文件。这通常会在 Xcode 项目中使用一个构建步骤来完成；请参阅 iOS 文档了解有关如何构造此构建步骤的细节。

Added in version 3.13.

Availability: iOS.

name

装载器支持的模块名。

path

扩展模块 .fwork 文件的路径。

32.5.5 importlib.util ——导入器的工具程序代码

源代码： Lib/importlib/util.py

本模块包含了帮助构建*importer* 的多个对象。

importlib.util.**MAGIC_NUMBER**

代表字节码版本号的字节串。若要有助于加载/写入字节码，可考虑采用*importlib.abc.SourceLoader*。

Added in version 3.4.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, *, *optimization=None*)

返回 **PEP 3147/PEP 488** 定义的，与源 *path* 相关联的已编译字节码文件的路径。例如，如果 *path* 为 /foo/bar/baz.py 则 Python 3.2 中的返回值将是 /foo/bar/__pycache__/baz.cpython-32.pyc。字符串 cpython-32 来自于当前的魔法标签 (参见 get_tag()；如果 sys.implementation.cache_tag 未定义则将会引发*NotImplementedError*)。

参数 *optimization* 用于指定字节码文件的优化级别。空字符串代表没有优化，所以 *optimization* 为的 /foo/bar/baz.py，将会得到字节码路径为 /foo/bar/__pycache__/baz.cpython-32.pyc。None 会导致采用解释器的优化。任何其他字符串都会被采用，所以 *optimization* 为 '' 的 /foo/bar/baz.py 会导致字节码路径为 /foo/bar/__pycache__/baz.cpython-32.opt-2.pyc。*optimization* 字符串只能是字母数字，否则会触发*ValueError*。

debug_override 参数已废弃，可用于覆盖系统的 __debug__ 值。True 值相当于将 *optimization* 设为空字符串。False 则相当于 *optimization* 设为 1。如果 *debug_override* 和 *optimization* 都不为 None，则会触发*TypeError*。

Added in version 3.4.

在 3.5 版本发生变更: 增加了 *optimization* 参数，废弃了 *debug_override* 参数。

在 3.6 版本发生变更: 接受一个*path-like object*。

importlib.util.**source_from_cache**(*path*)

根据指向一个 **PEP 3147** 文件名的 *path*，返回相关联的源代码文件路径。举例来说，如果 *path* 为 /foo/bar/__pycache__/baz.cpython-32.pyc 则返回的路径将是 /foo/bar/baz.py。*path* 不需要已存在，但如果它未遵循 **PEP 3147** 或 **PEP 488** 的格式，则会引发*ValueError*。如果未定义 sys.implementation.cache_tag，则会引发*NotImplementedError*。

Added in version 3.4.

在 3.6 版本发生变更: 接受一个*path-like object*。

importlib.util.**decode_source**(*source_bytes*)

对代表源代码的字节串进行解码，并将其作为带有通用换行符的字符串返回（符合*importlib.abc.InspectLoader.get_source()* 要求）。

Added in version 3.4.

importlib.util.**resolve_name**(*name*, *package*)

将模块的相对名称解析为绝对名称。

如果 **name** 前面没有句点，那就简单地返回 **name**。这样就能采用 `importlib.util.resolve_name('sys', __spec__.parent)` 之类的写法，而无需检查是否需要 **package** 参数。

如果 **name** 是一个相对模块名称但 **package** 为假值（如为 `None` 或空字符串）则会引发*ImportError*。如果相对名称离开了其所在的包（如为从 `spam` 包请求 `..bacon` 的形式）则也会引发*ImportError*。

Added in version 3.3.

在 3.9 版本发生变更: 为了改善与 import 语句的一致性，对于无效的相对导入尝试会引发*ImportError* 而不是*ValueError*。

importlib.util.**find_spec**(*name*, *package=None*)

查找模块的*spec*，可选择相对于指定的 **package** 名称。如果该模块位于*sys.modules* 中，则会返回 `sys.modules[name].__spec__` (除非 spec 为 `None` 或未设置，在此情况下则会引发*ValueError*)。在其他情况下将使用*sys.meta_path* 进行搜索。如果找不到任何 spec 则返回 `None`。

如果 **name** 为一个子模块（带有一个句点），则会自动导入父级模块。

name 和 **package** 的用法与 `import_module()` 相同。

Added in version 3.4.

在 3.7 版本发生变更: 如果 **package** 实际上不是一个包（即缺少 `__path__` 属性）则会引发*ModuleNotFoundError* 而不是*AttributeError*。

importlib.util.**module_from_spec**(*spec*)

基于 **spec** 和*spec.loader.create_module* 创建一个新模块。

如果*spec.loader.create_module* 未返回 `None`，那么先前已存在的属性不会被重置。另外，如果*AttributeError* 是在访问 **spec** 或设置模块属性时触发的，则不会触发。

本函数比*types.ModuleType* 创建新模块要好，因为用到 **spec** 模块设置了尽可能多的导入控制属性。

Added in version 3.5.

importlib.util.**spec_from_loader**(*name*, *loader*, ***, *origin=None*, *is_package=None*)

一个工厂函数，用于创建基于加载器的*ModuleSpec* 实例。参数的含义与 ModuleSpec 的相同。该函数会利用当前可用的*loader* API，比如 `InspectLoader.is_package()`，以填充所有缺失的规格信息。

Added in version 3.4.

importlib.util.**spec_from_file_location**(*name*, *location*, ***, *loader=None*, *submodule_search_locations=None*)

一个工厂函数，根据文件路径创建*ModuleSpec* 实例。缺失的信息将根据 spec 进行填补，利用加载器 API，以及模块基于文件的隐含条件。

Added in version 3.4.

在 3.6 版本发生变更: 接受一个*path-like object*。

importlib.util.**source_hash**(*source_bytes*)

以字节串的形式返回 *source_bytes* 的哈希值。基于哈希值的 `.pyc` 文件在头部嵌入了对应源文件内容的*source_hash()*。

Added in version 3.7.

importlib.util.**_incompatible_extension_module_restrictions**(***, *disable_check*)

一个可以暂时跳过扩展模块兼容性检查的上下文管理器。在默认情况下该检查将被启用并且当在子解释器中导入单阶段初始化模块时该检查会失败。如果多阶段初始化模块没有显式地支持针对子解释器的 GIL，那么当它在一个有自己的 GIL 的解释器中被导入时，该检查也会失败。

请注意该函数是为了适应一种不寻常的情况；这种情况可能最终会消失。这很有可能不是你需要考虑的事情。

你可以通过实现多阶段初始化的基本接口 (**PEP 489**) 并假装支持多解释器 (或解释器级的 GIL) 来获得与该函数相同的效果。

警告

使用该函数来禁用检查可能会导致预期之外的行为甚至崩溃。它应当仅在扩展模块开发过程中使用。

Added in version 3.12.

class importlib.util.**LazyLoader**(*loader*)

此类会延迟执行模块加载器，直至该模块有一个属性被访问到。

此类 **仅仅**适用于定义*exec_module()* 作为需要控制模块使用何种模块类型的加载器。出于相同理由，加载器的*create_module()* 方法必须返回 None 或其 __class__ 属性可被改变并且不使用槽位 的类型。最后，用于替换已放入*sys.modules* 的对象的模块将无法工作因为没有办法安全地在整个解释器中正确替换模块引用；如果检测到这种替换则会引发*ValueError*。

备注

如果项目对启动时间要求很高，只要模块未被用过，此类能够最小化加载模块的开销。对于启动时间并不重要的项目来说，由于加载过程中产生的错误信息会被暂时搁置，因此强烈不建议使用此类。

Added in version 3.5.

在 3.6 版本发生变更: 开始调用*create_module()*，移除*importlib.machinery.BuiltinImporter* 和*importlib.machinery.ExtensionFileLoader* 的兼容性警告。

classmethod factory(*loader*)

一个返回创建延迟加载器的可调用对象的类方法。这专门被用于加载器由类而不是实例来传入的场合。

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

32.5.6 例子

用编程方式导入

要以编程方式导入一个模块，请使用*importlib.import_module()*：

```
import importlib

itertools = importlib.import_module('itertools')
```

检查某模块可否导入。

如果你需要在不实际执行导入的情况下确定某个模块是否可被导入，则你应当使用*importlib.util.find_spec()*。

请注意如果 name 是一个子模块 (即包含一个点号)，则*importlib.util.find_spec()* 将会导入父模块。

```
import importlib.util
import sys
```

(续下页)

(接上页)

```
# 出于展示目的。
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # 如果你选择执行实际的导入 ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

直接导入源码文件。

这个专用方案应当谨慎使用：它是直接指明文件路径而不搜索*sys.path* 的 import 语句的近似物。应当首先考虑其他替代方案，比如在需要某个特定模块时修改*sys.path*，或在运行某个 Python 得到正确的全局命名空间时使用*runpy.run_path()*。

要直接从某个路径导入 Python 源文件，请使用以下写法:

```
import importlib.util
import sys


def import_from_path(module_name, file_path):
    spec = importlib.util.spec_from_file_location(module_name, file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules[module_name] = module
    spec.loader.exec_module(module)
    return module


# 仅用于演示目的 (`json` 是随意选择的)。
import json
file_path = json.__file__
module_name = json.__name__

# 结果与 `import json` 类似。
json = import_from_path(module_name, file_path)
```

实现延迟导入

以下例子展示了如何实现延迟导入:

```
>>> import importlib.util
>>> import sys
>>> def lazy_import(name):
...     spec = importlib.util.find_spec(name)
...     loader = importlib.util.LazyLoader(spec.loader)
...     spec.loader = loader
...     module = importlib.util.module_from_spec(spec)
...     sys.modules[name] = module
...     loader.exec_module(module)
...     return module
...
>>> lazy_typing = lazy_import("typing")
>>> #lazy_typing is a real module object,
>>> #but it is not loaded in memory yet.
```

(续下页)

(接上页)

```
>>> lazy_typing.TYPE_CHECKING
False
```

导入器的配置

对于导入的深度定制，通常你需要实现一个*importer*。这意味着同时管理*finder* 和*loader* 两方面。对于查找器来说根据你的需求有两种类别可供选择: *meta path finder* 或*path entry finder*。前者你应当放到*sys.meta_path* 而后者是使用*path entry hook* 在*sys.path_hooks* 上创建并与*sys.path* 条目一起创建一个潜在的查找器。下面的例子将向你演示如何注册自己的导入器供导入机制使用 (关于自行创建导入器，请阅读在本包内定义的相应类的文档):

```
import importlib.machinery
import sys

# 仅用于展示目的。
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# 设置一个元路径查找器。
# 确保根据优先级别将查找器放在列表中正确的位置上。
sys.meta_path.append(SpamMetaPathFinder)

# 设置一个路径条目查找器。
# 确保根据优先级别将路径钩子放在列表中正确的位置上。
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

importlib.import_module() 的近似实现

导入过程本身是用 Python 代码实现的，这样就有可能通过 importlib 来对外公开大部分导入机制。以下代码通过提供*importlib.import_module()* 的近似实现来说明 importlib 所公开的几种 API:

```
import importlib.util
import sys

def import_module(name, package=None):
    """导入的近似实现。"""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
```

(续下页)

(接上页)

```
        setattr(parent_module, child_name, module)
    return module
```

32.6 importlib.resources -- 包资源的读取、打开和访问

源代码: Lib/importlib/resources/__init__.py

Added in version 3.7.

此模块调整了 Python 的导入系统以便提供对 包内部的 资源的访问。

“资源”是指 Python 中与模块或包相关联的文件类资源。资源可以直接包含在某个包中，包含在某个包的子目录中，或是与某个包外部的模块相邻。资源可以是文本或二进制数据。因此，从技术上说 Python 包的模块源代码文件 (.py) 和编译结果文件 (pycache) 就是包实际所包含的资源。但是，在实践中，资源主要是指包作者专门公开的非 Python 文件。

资源可以使用二进制或文本模式打开。

资源大致相当于目录内的文件，不过需要记住这只是一个比喻。资源和包 **不是**必须如文件系统上的物理文件和目录那样存在的：例如，一个包及其资源可使用*zipimport* 从一个 ZIP 文件导入。

备注

本模块提供了类似于 pkg_resources Basic Resource Access 的功能而没有那样高的性能开销。这使得读取包中的资源更为容易，并具有更为稳定和一致的语义。

此模块的独立向下移植版本在 using importlib.resources 和 migrating from pkg_resources to importlib.resources 中提供了更多信息。

想要支持资源读取的加载器应当实现*importlib.resources.abc.ResourceReader* 中规定的 `get_resource_reader(fullname)` 方法。

class importlib.resources.**Anchor**

代表资源的锚点，可以是一个模块对象 或字符串形式的模块名称不。定义为 `Union[str, ModuleType]`。

importlib.resources.**files**(*anchor: Anchor | None = None*)

返回一个代表资源容器（相当于目录）及其资源（相当于文件）的*Traversable* 对象。Traversable 可以包含其他容器（相当于子目录）。

anchor 是一个可选的*Anchor*。如果 anchor 是一个包，则会从这个包获取资源。如果是一个模块，则会从这个模块的相邻位置获取资源（在同一个包或包的根目录中）。如果省略了 anchor, 则会使用调用方的模块。

Added in version 3.9.

在 3.12 版本发生变更: *package* 形参被重命名为 *anchor*。*anchor* 现在可以是一个不为包的模块，如果被省略则默认为调用方的模块。为保持兼容性 *package* 仍然被接受但会引发*DeprecationWarning*。请考虑以位置参数方式传入或使用 `importlib_resources >= 5.10` 作为针对旧版 Python 的兼容接口。

importlib.resources.**as_file**(*traversable*)

给定一个代表文件或目录的*Traversable* 对象，通常是来自*importlib.resources.files()*，返回一个上下文管理器以供 `with` 语句使用。该上下文管理器提供一个*pathlib.Path* 对象。

退出上下文管理器后会清除从 zip 文件等提取资源时创建的任何临时文件或目录。

当 Traversable 的方法（如 `read_text` 等）不足以满足需要而需要文件系统中的真实文件或目录时请使用 `as_file`。

Added in version 3.9.

在 3.12 版本发生变更: 增加了对代表目录的 *traversable* 的支持。

32.6.1 函数式 API

提供了一套简化的、向下兼容的辅助工具。这些工具使得常用操作只需一次函数调用即可完成。

对于以下所有函数:

- *anchor* 是一个*Anchor*，就像在*files()* 中的一样。与在 `files` 中不同，它不可被省略。
- *path_names* 是相对于 anchor 的资源的路径名的各个组成部分。例如，要获取名为 `info.txt` 的资源的文本，则使用:

```
importlib.resources.read_text(my_module, "info.txt")
```

类似于*Traversable.joinpath*，单独的组成部分应当使用正斜杠 (/) 作为路径分隔符。例如，下面这些条目是等价的:

```
importlib.resources.read_binary(my_module, "pics/painting.png")
importlib.resources.read_binary(my_module, "pics", "painting.png")
```

出于保持向下兼容性的理由，如果给出了多个 *path_names* 则读取文本的函数需要一个显式的 *encoding* 参数。例如，要获取 `info/chapter1.txt` 的文本，则使用:

```
importlib.resources.read_text(my_module, "info", "chapter1.txt",
                              encoding='utf-8')
```

importlib.resources.**open_binary**(*anchor*, **path_names*)

打开指定的资源用于二进制读取。

请参阅说明文档 了解 *anchor* 和 *path_names* 的详情。

此函数返回一个*BinaryIO* 对象，即一个打开供读取的二进制流。

此函数大致等价于:

```
files(anchor).joinpath(*path_names).open('rb')
```

在 3.13 版本发生变更: 可接受多个 *path_names*。

importlib.resources.**open_text**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

打开指定的资源用于文本读取。在默认情况下，将使用严格 UTF-8 编码读取内容。

请参阅说明文档 了解 *anchor* 和 *path_names* 的详情。*encoding* 和 *errors* 的含义与在内置*open()* 中的相同。

出于保持向下兼容性的理由，如果有多个 *path_names* 则需要显式地给出 *encoding* 参数。此限制计划在 Python 3.15 中去除。

此函数返回一个*TextIO* 对象，即一个打开供读取的文本流。

此函数大致等价于:

```
files(anchor).joinpath(*path_names).open('r', encoding=encoding)
```

在 3.13 版本发生变更: 可接受多个 *path_names*。必须以关键字参数形式给出 *encoding* 和 *errors*。

importlib.resources.**read_binary**(*anchor*, **path_names*)

以*bytes* 形式读取并返回指定资源的内容。

请参阅说明文档 了解 *anchor* 和 *path_names* 的详情。

此函数大致等价于:

```
files(anchor).joinpath(*path_names).read_bytes()
```

在 3.13 版本发生变更: 可接受多个 *path_names*。

importlib.resources.**read_text**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

以*str* 形式读取并返回指定资源的内容。在默认情况下，将使用严格 UTF-8 编码读取内容。

请参阅说明文档 了解 *anchor* 和 *path_names* 的详情。*encoding* 和 *errors* 的含义与在内置*open()* 中的相同。

出于保持向下兼容性的理由，如果有多个 *path_names* 则需要显式地给出 *encoding* 参数。此限制计划在 Python 3.15 中去除。

此函数大致等价于:

```
files(anchor).joinpath(*path_names).read_text(encoding=encoding)
```

在 3.13 版本发生变更: 可接受多个 *path_names*。必须以关键字参数形式给出 *encoding* 和 *errors*。

importlib.resources.**path**(*anchor*, **path_names*)

以实际文件系统路径的形式提供 *resource* 的路径。此函数返回一个可在 `with` 语句中使用的上下文管理器。该上下文管理器将提供一个*pathlib.Path* 对象。

退出上下文管理器时将清理已创建的任何临时文件，例如在需要从 zip 文件提取资源时就将创建临时文件。

例如，*stat()* 方法需要一个实际的文件系统路径；可以这样使用它:

```
with importlib.resources.path(anchor, "resource.txt") as fspath:
    result = fspath.stat()
```

请参阅说明文档 了解 *anchor* 和 *path_names* 的详情。

此函数大致等价于:

```
as_file(files(anchor).joinpath(*path_names))
```

在 3.13 版本发生变更: 可接受多个 *path_names*。必须以关键字参数形式给出 *encoding* 和 *errors*。

importlib.resources.**is_resource**(*anchor*, **path_names*)

如果指定的资源存在则返回 `True`，否则返回 `False`。此函数不会考虑目录作为资源的情况。

请参阅说明文档 了解 *anchor* 和 *path_names* 的详情。

此函数大致等价于:

```
files(anchor).joinpath(*path_names).is_file()
```

在 3.13 版本发生变更: 可接受多个 *path_names*。

importlib.resources.**contents**(*anchor*, **path_names*)

返回一个用于遍历指定的包或路径内条目的可迭代对象。该可迭代对象将以*str* 的形式返回资源（例如文件）和非资源（例如目录）。该迭代器不会递归进入子目录。

请参阅说明文档 了解 *anchor* 和 *path_names* 的详情。

此函数大致等价于:

```
for resource in files(anchor).joinpath(*path_names).iterdir():
    yield resource.name
```

自 3.11 版本弃用: 首选上面的 `iterdir()`，它提供了对结果的更多控制和更丰富的功能。

32.7 importlib.resources.abc -- 资源的抽象基类

源代码: Lib/importlib/resources/abc.py

Added in version 3.11.

class importlib.resources.abc.**ResourceReader**

被 *TraversableResources* 取代

提供读取 *resources* 能力的一个*abstract base class* 。

从这个 ABC 的视角出发，*resource* 指一个包附带的二进制文件。常见的如在包的 `__init__.py` 文件旁的数据文件。这个类存在的目的是为了将对数据文件的访问进行抽象，这样包就和其数据文件的存储方式无关了。不论这些文件是存放在一个 zip 文件里还是直接在文件系统内。

对于该类中的任一方法，*resource* 参数的值都需要是一个在概念上表示文件名称的*path-like object*。这意味着任何子目录的路径都不该出现在 *resouce* 参数值内。因为对于阅读器而言，包的位置就代表着「目录」。因此目录和文件名就分别对应于包和资源。这也是该类的实例都需要和一个包直接关联（而不是潜在指代很多包或者一整个模块）的原因。

想支持资源读取的加载器需要提供一个返回实现了此 ABC 的接口的 `get_resource_reader(fullname)` 方法。如果通过全名指定的模块不是一个包，这个方法应该返回`None`。当指定的模块是一个包时，应该只返回一个与这个抽象类 ABC 兼容的对象。

自 3.12 版本弃用: 使用`importlib.resources.abc.TraversableResources` 代替。

abstractmethod open_resource(*resource*)

返回一个打开的*file-like object* 用于 *resource* 的二进制读取。

如果无法找到资源，将会引发`FileNotFoundError`。

abstractmethod resource_path(*resource*)

返回 *resource* 的文件系统路径。

如果资源并不实际存在于文件系统中，将会引发`FileNotFoundError`。

abstractmethod is_resource(*name*)

如果 *name* 被视作资源，则返回 True。如果 *name* 不存在，则引发`FileNotFoundError` 异常。

abstractmethod contents()

反回由字符串组成的*iterable*，表示这个包的所有内容。请注意并不要求迭代器返回的所有名称都是实际的资源，例如返回`is_resource()` 为假值的名称也是可接受的。

允许非资源名字被返回是为了允许存储的一个包和它的资源的方式是已知先验的并且非资源名字会有用的情况。比如，允许返回子目录名字，目的是当得知包和资源存储在文件系统上面的时候，能够直接使用子目录的名字。

这个抽象方法返回了一个不包含任何内容的可迭代对象。

class importlib.resources.abc.**Traversable**

一个具有`pathlib.Path` 中方法的子集并适用于遍历目录和打开文件的对象。

对于该对象在文件系统中的表示形式，请使用`importlib.resources.as_file()`。

name

抽象属性。此对象的不带任何父引用的基本名称。

abstractmethod iterdir()

产出自身内部的可遍历对象。

abstractmethod is_dir()

如果 self 是一个目录则返回 `True`。

abstractmethod is_file()

如果 self 是一个文件则返回 `True`。

abstractmethod joinpath(**pathsegments*)

按照 *pathsegments* 遍历目录并以 `Traversable` 形式返回结果。

每个 *pathsegments* 参数可能包含以正斜杠 (`/`, `posixpath.sep`) 分隔的多个名称。例如，以下值是等价的:

```
files.joinpath('subdir', 'subsuddir', 'file.txt')
files.joinpath('subdir/subsuddir/file.txt')
```

请注意某些 `Traversable` 实现可能没有升级到最新版本的协议。要与这样的实现保持兼容，可以向每个对 `joinpath` 的调用提供提供单个不带路径分隔符的参数。例如:

```
files.joinpath('subdir').joinpath('subsubdir').joinpath('file.txt')
```

在 3.11 版本发生变更: `joinpath` 接受多个 *pathsegments*，这些部分可以包含正斜杠作为路径分隔符。在之前版本中，只接受单个 *child* 参数。

abstractmethod __truediv__(*child*)

返回可遍历的子对象自身。等价于 `joinpath(child)`。

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode 可以为'r' 或'rb' 即以文本或二进制模式打开。返回一个适用于读取的句柄（与*pathlib.Path.open* 样同）。

当以文件模式打开时，接受与*io.TextIOWrapper* 所接受的相同编码格式形参。

read_bytes()

以字节串形式读取自身的内容。

read_text(*encoding=None*)

以文本形式读取自身的内容。

class importlib.resources.abc.**TraversableResources**

针对能够为*importlib.resources.files()* 接口提供服务的资源读取器的抽象基类。子类化*ResourceReader* 并为 `ResourceReader` 的抽象方法提供具体实现。因此，任何提供了 `TraversableResources` 的加载器也会提供 `ResourceReader`。

需要支持资源读取的加载器应实现此接口。

abstractmethod files()

为载入的包返回一个*importlib.resources.abc.Traversable* 对象。

32.8 importlib.metadata -- 访问软件包元数据

Added in version 3.8.

在 3.10 版本发生变更: `importlib.metadata` 不再是暂定的。

源代码: Lib/importlib/metadata/__init__.py

`importlib.metadata` 是一个提供对已安装的 分发包 的元数据的访问的库，如其入口点或其顶层名称 (导入包, 模块等，如果存在的话）。这个库部分构建于 Python 的导入系统之上，其目标是取代 `pkg_resources` 的中的 entry point API 和 metadata API。配合*importlib.resources* ，这个包使得较老旧且低效的 `pkg_resources` 包不再必要。

`importlib.metadata` 对 pip 等工具安装到 Python 的 `site-packages` 目录的第三方 分发包进行操作。具体来说，适用的分发包应带有可发现的 `dist-info` 或 `egg-info` 目录，以及 核心元数据规范说明 定义的元数据。

重要

它们 不一定等同或 1:1 对应于可在 Python 代码中导入的顶层 导入包名称。一个 分发包可以包含多个 导入包 (和单个模块)，如果是命名空间包，一个顶层 导入包可以映射到多个 分发包。您可以使用*packages_distributions()* 来获取它们之间的映射。

分发包元数据默认可存在于*sys.path* 下的文件系统或 zip 归档文件中。通过一个扩展机制，元数据可以存在于几乎任何地方。

参见

https://importlib-metadata.readthedocs.io/
importlib_metadata 的文档，它向下移植了 importlib.metadata。它包含该模块的类和函数的 API 参考，以及针对 pkg_resources 现有用户的 迁移指南。

32.8.1 概述

让我们假设你想要获取你使用 pip 安装的某个 分发包 的版本字符串。我们首先创建一个虚拟环境并在其中安装一些软件包：

```
$ python -m venv example
$ source example/bin/activate
(example) $ python -m pip install wheel
```

你可以通过运行以下代码得到 wheel 的版本字符串：

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

你还能得到可通过 EntryPoint 的属性 (通常为'group' 或'name') 来选择的入口点多项集，比如 console_scripts, distutils.commands。每个 group 包含一个由*EntryPoint* 对象组成的多项集。

你可以获得*分发的元数据*：

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-email',
↪'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL', 'Project-URL',
↪'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier
↪', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Requires-Python', 'Provides-Extra', 'Requires-Dist', 'Requires-Dist']
```

你也可以获得*分发包的版本号*，列出它的*构成文件*，并且得到分发包的*分发包的依赖* 列表。

exception importlib.metadata.**PackageNotFoundError**
当查询未在当前 Python 环境中安装的分发包时由此模块的某些函数所引发的*ModuleNotFoundError* 子类。

32.8.2 函数式 API

这个包的公开 API 提供了以下功能。

入口点

importlib.metadata.**entry_points**(***select_params*)
返回一个描述当前环境的入口点的*EntryPoints* 实例。所给出的任何关键字形参都将被传给 select() 方法以与单独的入口点定义的属性进行比较。

注意：目前无法基于 EntryPoint.dist 属性来查询入口点（因为不同的 Distribution 实例目前不可能相等，即使它们具有相同的属性）

class importlib.metadata.**EntryPoints**

已安装入口点多项集的详情。

还提供 .groups 属性用于报告所有已标识的入口点分组，以及 .names 属性用于报告所有已标识的入口点名称。

class importlib.metadata.**EntryPoint**

一个已安装入口点的详情。

每个 EntryPoint 实例都有 .name, .group 和 .value 属性以及 .load() 方法用于求值。此外还有 .module, .attr 和 .extras 属性用于获取 .value 属性的组成部分，以及 .dist 用于获取有关提供该入口点的分发包的信息。

查询所有的入口点：

```
>>> eps = entry_points()
```

entry_points() 函数返回一个 EntryPoints 对象，即由带 names 和 groups 属性的所有 EntryPoint 对象组成的多项集以方便使用：

```
>>> sorted(eps.groups)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.writers',
↪'setuptools.installation']
```

EntryPoints 具有 select() 方法用于选择匹配指定特征属性的入口点。如选择 console_scripts 组中的入口点：

```
>>> scripts = eps.select(group='console_scripts')
```

效果相同，因为 entry_points() 会传递关键字参数来选择：

```
>>> scripts = entry_points(group='console_scripts')
```

选出命名为“wheel”的特定脚本（可以在 wheel 项目中找到）：

```
>>> 'wheel' in scripts.names
True
>>> wheel = scripts['wheel']
```

等价地，在选择过程中查询对应的入口点：

```
>>> (wheel,) = entry_points(group='console_scripts', name='wheel')
>>> (wheel,) = entry_points().select(group='console_scripts', name='wheel')
```

检查解析得到的入口点：

```
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> wheel.module
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

group 和 name 是由包作者定义的任意值并且通常来说客户端会想要解析特定 group 的所有入口点。请参阅 the setuptools docs 了解有关入口点，其定义和用法的更多信息。

在 3.12 版本发生变更: "selectable" 入口点是在 importlib_metadata 3.6 和 Python 3.10 中引入的。在这项改变之前，entry_points 不接受任何形参并且总是返回一个由入口点组成的字典，字典的键为分组名。在 importlib_metadata 5.0 和 Python 3.12 中，entry_points 总是返回一个 EntryPoints 对象。请参阅 backports.entry_points_selectable 了解相关兼容性选项。

在 3.13 版本发生变更: EntryPoint 对象不再提供类似于元组的接口（__getitem__()）。

分发的元数据

importlib.metadata.**metadata**(*distribution_name*)

将对应于指定分发包的分发元数据作为*PackageMetadata* 实例返回。

如果指定的发布包未在当前 Python 环境中安装则会引发*PackageNotFoundError*。

class importlib.metadata.**PackageMetadata**

PackageMetadata 协议 的一个具体实现。

除了提供已定义的协议方法和属性，对实例的下标操作就相当于调用 get() 方法。

每个 分发包 都包括一些元数据，你可以使用 metadata() 函数来获取:

```
>>> wheel_metadata = metadata('wheel')
```

所返回数据结构的键指明了元数据关键字，而值将从分发的元数据中不加解析地返回:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

PackageMetadata 也还提供了一个按照 **PEP 566** 将所有元数据以 JSON 兼容形式返回的 json:

```
>>> wheel_metadata.json['requires_python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

此处并未描述可用元数据的完整集合。更多详情参见 PyPA 核心元数据规格说明。

在 3.10 版本发生变更: 当有效载荷中包含时，Description 以去除续行符的形式被包含于元数据中。

添加了 json 属性。

分发包的版本

importlib.metadata.**version**(*distribution_name*)

返回对应指定发布包的已安装发布包 版本。

如果指定的发布包未在当前 Python 环境中安装则会引发*PackageNotFoundError*。

version() 函数是获取字符串形式的 分发包 版本号的最快速方式:

```
>>> version('wheel')
'0.32.3'
```

分发包的文件

importlib.metadata.**files**(*distribution_name*)

返回包含在指定分发包内的完整文件集合。

如果指定的发布包未在当前 Python 环境中安装则会引发*PackageNotFoundError*。

如果找到了分发包但未找到报告与分发包相关联的文件的安装数据库记录则返回*None*。

class importlib.metadata.**PackagePath**

一个*pathlib.PurePath* 的派生对象，增加了对应于指定文件的分发包的安装元数据的 dist, size 和 hash 特征属性。

files() 函数接受一个 分发包 名称并返回此分发包所安装的全部文件。每个文件均报告为一个*PackagePath* 实例。例如:

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

当你获得了文件对象，你可以读取其内容:

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

你也可以使用 locate() 方法来获取文件的绝对路径:

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-packages/wheel/util.py')
```

对于列出文件的元数据文件 (RECORD 或 SOURCES.txt) 缺失的情况，files() 将返回*None*。调用者可能会想要把对 files() 的调用包装在 always_iterable 中或是用其他方式在尚未知晓目标分发元数据存在性时应对此情况。

分发包的依赖

importlib.metadata.**requires**(*distribution_name*)

返回指定分发包已声明的依赖描述。

如果指定的发布包未在当前 Python 环境中安装则会引发*PackageNotFoundError*。

要获取一个 分发包 的完整需求集，请使用 requires() 函数:

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

将导入映射到分发包

importlib.metadata.**packages_distributions**()

返回一个从最高层级模块和通过*sys.meta_path* 找到的导入包名称到提供相应文件的分发包名称（如果存在）的映射。

为了允许使用命名空间包（它可能包含由多个分发包所提供的成员），每个最高层级导入名称都映射到一个分发名称的列表而不是直接映射单个名称。

解析每个提供可导入的最高层级 Python 模块或 导入包 对应的 分发包 名称（对于命名空间包可能有多个名称）的快捷方法:

```
>>> packages_distributions()
{'importlib_metadata': ['importlib-metadata'], 'yaml': ['PyYAML'], 'jaraco': ['jaraco.classes
↪', 'jaraco.functools'], ...}
```

某些可编辑的安装 没有提供最高层级名称，因而此函数不适用于这样的安装。

Added in version 3.10.

32.8.3 分发包对象

importlib.metadata.**distribution**(*distribution_name*)

返回一个描述指定分发包的*Distribution* 实例。instance describing the named distribution package.

如果指定的发布包未在当前 Python 环境中安装则会引发*PackageNotFoundError*。

class importlib.metadata.**Distribution**

一个已安装分发包的详情。

注意：目前不同的 Distribution 实例在比较时肯定不相等，即使它们是关联到相同的已安装发布版因而具有相同的属性。

虽然上面描述的模块级 API 是最常见且便捷的用法，但你也可以从 Distribution 类获取所有信息。Distribution 是一个代表 Python 分发包 元数据的抽象对象。你可以通过调用*distribution()* 函数来获取对应某个已安装分发包的具体 Distribution 子类实例:

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
>>> type(dist)
<class 'importlib.metadata.PathDistribution'>
```

因此，一个获取版本号的替代方式是通过 Distribution 实例:

```
>>> dist.version
'0.32.3'
```

在 Distribution 实例上提供了所有类别的附加元数据:

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

对于可编辑包，origin 属性可能表示 **PEP 610** 元数据:

```
>>> dist.origin.url
'file:///path/to/wheel-0.32.3.editable-py3-none-any.whl'
```

此处并未描述可用元数据的完整集合。更多详情参见 PyPA 核心元数据规格说明。

Added in version 3.13: 增加了 .origin 特征属性。

32.8.4 分发包的发现

在默认情况下，这个包针对文件系统和 zip 文件 分发包 的元数据发现提供了内置支持。这个元数据查找器的搜索目标默认为 sys.path，但它对来自其他导入机制行为方式的解读会略有变化。特别地:

- importlib.metadata 不会识别 sys.path 上的*bytes* 对象。
- importlib.metadata 将顺带识别 sys.path 上的*pathlib.Path* 对象，即使这些值会被导入操作所忽略。

32.8.5 扩展搜索算法

因为 分发包 元数据不能通过*sys.path* 搜索，或是通过包加载器直接获得，一个分发包的元数据是通过导入系统的 查找器找到的。要找到分发包的元数据，importlib.metadata 将在*sys.meta_path* 上查询元路径查找器 的列表。

在默认情况下 importlib.metadata 会安装在文件系统中找到的分发包的查找器。这个查找器无法真正找出任何 分发包，但它能找到它们的元数据。

抽象基类*importlib.abc.MetaPathFinder* 定义了 Python 导入系统期望的查找器接口。importlib.metadata 通过寻找*sys.meta_path* 上查找器可选的 find_distributions 可调用的属性扩展这个协议，并将这个扩展接口作为 DistributionFinder 抽象基类提供，它定义了这个抽象方法:

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()):
    """返回一个由所有能够为指定 ``context`` 加载包的元数据的
    Distribution 实例组成的可迭代对象。
    """
```

DistributionFinder.Context 对象提供了指示搜索路径和匹配名称的属性 .path 和 .name，也可能提供其他相关的上下文。

这在实践中意味着要支持在文件系统外的其他位置查找分发包的元数据，你需要子类化 Distribution 并实现抽象方法，之后从一个自定义查找器的 find_distributions() 方法返回这个派生的 Distribution 实例。

示例

例如，考虑一个从数据库中加载 Python 模块的自定义查找器：

```
class DatabaseImporter(importlib.abc.MetaPathFinder):
    def __init__(self, db):
        self.db = db

    def find_spec(self, fullname, target=None) -> ModuleSpec:
        return self.db.spec_from_name(fullname)

sys.meta_path.append(DatabaseImporter(connect_db(...)))
```

该导入器现在大概可以从数据库中导入模块，但它不提供元数据或入口点。这个自定义导入器如果要提供元数据，它还需要实现 DistributionFinder：

```
from importlib.metadata import DistributionFinder

class DatabaseImporter(DistributionFinder):
    ...

    def find_distributions(self, context=DistributionFinder.Context()):
        query = dict(name=context.name) if context.name else {}
        for dist_record in self.db.query_distributions(query):
            yield DatabaseDistribution(dist_record)
```

这样一来，query_distributions 就会返回数据库中与查询匹配的每个分发包的记录。例如，如果数据库中有 requests-1.0，find_distributions 就会为 Context(name='requests') 或 “Context(name=None)“ 产生 DatabaseDistribution。

为简单起见，本例忽略了 context.path。path 属性默认为 sys.path，是搜索中考虑的导入路径集。一个 DatabaseImporter 可以在不考虑搜索路径的情况下运作。假设导入器不进行分区，那么”path” 就无关紧要了。为了说明 path 的作用，示例需要展示一个更复杂的 DatabaseImporter，它的行为随 sys.path/PYTHONPATH 而变化。在这种情况下，find_distributions 应该尊重 context.path，并且只产生与该路径相关的 Distribution。

那么，DatabaseDistribution 看起来就像是这样：

```
class DatabaseDistribution(importlib.metadata.Distribution):
    def __init__(self, record):
        self.record = record

    def read_text(self, filename):
        """
        为当前分发版读取文件型的 "METADATA"。
        """
        if filename == "METADATA":
            return f"""Name: {self.record.name}
Version: {self.record.version}
```

(续下页)

(接上页)

```
"""
        if filename == "entry_points.txt":
            return "\n".join(
              f"""[{ep.group}]\n{ep.name}={ep.value}"""
              for ep in self.record.entry_points)

    def locate_file(self, path):
        raise RuntimeError("This distribution has no file system")
```

这个基本实现应当为由 `DatabaseImporter` 进行服务的包提供元数据和入口点，假定 `record` 提供了适当的 `.name`, `.version` 和 `.entry_points` 属性。

`DatabaseDistribution` 还可能提供其他元数据文件，比如 `RECORD` (对 `Distribution.files` 来说需要) 或重写 `Distribution.files` 的实现。请参看源代码深入了解。

32.9 `sys.path` 模块搜索路径的初始化

模块搜索路径是在 Python 启动时被初始化的。这个模块搜索路径可通过*sys.path* 来访问。

模块搜索路径的第一个条目是包含输入脚本的目录，如果存在输入脚本的话。否则，第一个条目将是当前目录，当执行交互式 shell, `-c` 命令, 或 `-m` 模块时都属于这种情况。

`PYTHONPATH` 环境变量经常被用于将目录添加到搜索路径。如果发现了该环境变量则其内容将被添加到模块搜索路径中。

备注

`PYTHONPATH` 将影响所有已安装的 Python 版本/环境。在你的 shell 用户配置或全局环境变量中设置它时需要小心谨慎。*site* 模块提供了下文所述的更细微的技巧。

随后加入的条目是包含标准 Python 模块以及这些模块所依赖的任何*extension module* 的目录。扩展模块在 Windows 上为 `.pyd` 文件而在其他平台上则为 `.so` 文件。独立于平台的 Python 模块的目录称为 `prefix`。扩展模块的目录称为 `exec_prefix`。

`PYTHONHOME` 环境变量可以被用于设置 `prefix` 和 `exec_prefix` 的位置。在其他情况下这些目录将使用 Python 可执行文件作为起始点来确定然后再查找几处' 地标' 文件和目录。请注意任何符号链接也会被引入以便使用实际的 Python 可执行文件位置作为搜索起始点。这个 Python 可执行文件位置被称为 `home`。

一旦确定了 `home`，则 `prefix` 目录将通过首先查找 `pythonmajorversionminorversion.zip` (`python311.zip`) 来找到。在 Windows 上将会到 `home` 中搜索 zip 归档而在 Unix 上则会到 `lib` 中搜索它。请注意预期的 zip 归档位置即使在此归档不存在时仍然会被添加到模块搜索路径。如果未找到归档，在 Windows 上 Python 将继续通过查找 `Lib\os.py` 来搜索 `prefix`。在 Unix 上 Python 将查找 `lib/pythonmajorversion.minorversion/os.py` (`lib/python3.11/os.py`)。在 Windows 上 `prefix` 和 `exec_prefix` 是相同的，但是在其他平台上则会搜索 `lib/pythonmajorversion.minorversion/lib-dynload` (`lib/python3.11/lib-dynload`) 并将其用作 `exec_prefix` 的锚点。在某些平台上 `lib` 可能为 `lib64` 或其他值，请参阅*sys.platlibdir* 和 `PYTHONPLATLIBDIR`。

一旦找到，`prefix` 和 `exec_prefix` 将分别在*sys.prefix* 和*sys.exec_prefix* 上可用。

最后，将会处理*site* 模块并将 `site-packages` 目录添加到模块搜索路径。自定义搜索路径的一个常用方式是创建*sitecustomize* 或*usercustomize* 模块，如*site* 模块文档所描述的那样。

备注

特定的命令行选项可能对路径计算造成额外的影响。请参阅 `-E`, `-I`, `-s` 和 `-S` 了解更多细节。

32.9.1 从虚拟环境

如果 Python 运行在虚拟环境中（如 tut-venv 所描述）则 `prefix` 和 `exec_prefix` 都将是该虚拟环境专属的。

如果在主可执行文件的相同位置，或者在可执行文件的上一级目录中找到了 `pyvenv.cfg` 文件，则将应用以下变化形式:

- 如果 `home` 是一个绝对路径并且未设置 `PYTHONHOME`，则在推断 `prefix` 和 `exec_prefix` 时将使用此路径而不是主可执行文件的路径。

32.9.2 _pth 文件

若要完全覆盖*sys.path* 则请创建一个与共享库或可执行文件 (`python._pth` 或 `python311._pth`) 同名的 `._pth` 文件。共享库路径在 Windows 是始终是已知的，但这在其他平台上也许会不可用。请在 `._pth` 文件中为添加到*sys.path* 的每个路径指定对应的一行。基于共享库名称的文件会覆盖基于可执行文件的对应文件，这允许在必要时为任何加载运行时的程序限制路径。

当文件存在时，将忽略所有注册表和环境变量，启用隔离模式，并且：除非文件中的一行指定 `import site` ，否则不会导入*site* 。以 # 开头的空白路径和行将被忽略。每个路径可以是绝对的或相对于文件的位置。不允许使用除 `site` 以外的导入语句，并且不能指定任意代码。

请注意，当指定 `import site` 时，`.pth` 文件（没有前导下划线）将由*site* 模块正常处理。

32.9.3 嵌入式 Python

如果 Python 被嵌入其他应用程序中则 `Py_InitializeFromConfig()` 和 `PyConfig` 结构体可被用来初始化 Python。路径专属的细节描述见 init-path-config。

参见

- windows_finding_modules 了解更多有关 Windows 的细节说明。
- using-on-unix 了解 Unix 的相关细节。

CHAPTER 33

Python 语言服务

Python 提供了许多模块来帮助使用 Python 语言。这些模块支持标记化、解析、语法分析、字节码反汇编以及各种其他工具。

这些模块包括：

33.1 ast --- 抽象语法树

源代码： Lib/ast.py

ast 模块帮助 Python 程序处理 Python 语法的抽象语法树。抽象语法或许会随着 Python 的更新版发行而改变；该模块能够帮助理解当前语法在编程层面的样貌。

抽象语法树可通过将*ast.PyCF_ONLY_AST* 作为旗标传递给*compile()* 内置函数来生成，或是使用此模块中提供的*parse()* 辅助函数。返回结果将是一个由许多对象构成的树，这些对象所属的类都继承自*ast.AST*。抽象语法树可被内置的*compile()* 函数编译为一个 Python 代码对象。

33.1.1 抽象文法

抽象文法目前定义如下

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment, type_param* type_params)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns,
                             string? type_comment, type_param* type_params)
```

(续下页)

(接上页)

```
        | ClassDef(identifier name,
           expr* bases,
           keyword* keywords,
           stmt* body,
           expr* decorator_list,
           type_param* type_params)
        | Return(expr? value)

        | Delete(expr* targets)
        | Assign(expr* targets, expr value, string? type_comment)
        | TypeAlias(expr name, type_param* type_params, expr value)
        | AugAssign(expr target, operator op, expr value)
        -- 'simple' indicates that we annotate simple name without parens
        | AnnAssign(expr target, expr annotation, expr? value, int simple)

        -- use 'orelse' because else is a keyword in target languages
        | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
        | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_comment)
        | While(expr test, stmt* body, stmt* orelse)
        | If(expr test, stmt* body, stmt* orelse)
        | With(withitem* items, stmt* body, string? type_comment)
        | AsyncWith(withitem* items, stmt* body, string? type_comment)

        | Match(expr subject, match_case* cases)

        | Raise(expr? exc, expr? cause)
        | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
        | Assert(expr test, expr? msg)

        | Import(alias* names)
        | ImportFrom(identifier? module, alias* names, int? level)

        | Global(identifier* names)
        | Nonlocal(identifier* names)
        | Expr(expr value)
        | Pass | Break | Continue

        -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

        -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | NamedExpr(expr target, expr value)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension* generators)
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can occur
         | Await(expr value)
         | Yield(expr? value)
         | YieldFrom(expr value)
         -- need sequences for compare to distinguish between
         -- x < 4 < 3 and (x < 4) < 3
```

(续下页)

(接上页)

```
        | Compare(expr left, cmpop* ops, expr* comparators)
        | Call(expr func, expr* args, keyword* keywords)
        | FormattedValue(expr value, int conversion, expr? format_spec)
        | JoinedStr(expr* values)
        | Constant(constant value, string? kind)

        -- the following expression can appear in assignment context
        | Attribute(expr value, identifier attr, expr_context ctx)
        | Subscript(expr value, expr slice, expr_context ctx)
        | Starred(expr value, expr_context ctx)
        | Name(identifier id, expr_context ctx)
        | List(expr* elts, expr_context ctx)
        | Tuple(expr* elts, expr_context ctx)

        -- can appear only in Subscript
        | Slice(expr? lower, expr? upper, expr? step)

         -- col_offset is the byte offset in the utf8 string the parser uses
        attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    expr_context = Load | Store | Del

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                 expr* kw_defaults, arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation, string? type_comment)
           attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)
              attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
             attributes (int lineno, int col_offset, int? end_lineno, int? end_col_offset)

    withitem = (expr context_expr, expr? optional_vars)

    match_case = (pattern pattern, expr? guard, stmt* body)

    pattern = MatchValue(expr value)
            | MatchSingleton(constant value)
            | MatchSequence(pattern* patterns)
            | MatchMapping(expr* keys, pattern* patterns, identifier? rest)
            | MatchClass(expr cls, pattern* patterns, identifier* kwd_attrs, pattern* kwd_
↪patterns)
```

(续下页)

(接上页)

```
            | MatchStar(identifier? name)
            -- The optional "rest" MatchMapping parameter handles capturing extra mapping␣
↪keys

            | MatchAs(pattern? pattern, identifier? name)
            | MatchOr(pattern* patterns)

             attributes (int lineno, int col_offset, int end_lineno, int end_col_offset)

    type_ignore = TypeIgnore(int lineno, string tag)

    type_param = TypeVar(identifier name, expr? bound, expr? default_value)
               | ParamSpec(identifier name, expr? default_value)
               | TypeVarTuple(identifier name, expr? default_value)
              attributes (int lineno, int col_offset, int end_lineno, int end_col_offset)
}
```

33.1.2 节点类

class ast.**AST**

这是所有 AST 节点类的基类。实际的节点类都派生自 `Parser/Python.asdl` 文件，其完整内容如上所示。它们 `_ast` C 模块中定义并在*ast* 中重新导出。

抽象文法中的每个等号左边的符号（比方说，`ast.stmt` 或者 `ast.expr`）定义了一个类。另外，在等号右边，对每一个构造器也定义了一个类；这些类继承自等号左边的类。比如，*ast.BinOp* 继承自 `ast.expr`。对于多分支产生式（也就是含有“|”的产生式），左边的类是抽象的；只有具体构造器类的实例能够被 compile 函数构造。

_fields

每个实体类都具有属性 `_fields`，它给出了所有子节点的名字。

每个具体类的实例为自己的每个子节点都准备了一个属性来引用该子节点，属性的类型就是文法中所定义的。比如，*ast.BinOp* 的实例有个属性 `left`，类型是 `ast.expr`。

如果这些属性在文法中标记为可选（用问号标记），对应值可能会是 `None`。如果这些属性可有零或多个值（用星号标记），对应值会用 Python 的列表来表示。在用*compile()* 将 AST 编译为可执行代码时，所有的属性必须已经被赋值为有效的值。

_field_types

每次个实体类上的 `_field_types` 属性都是一个将字段名（与在*_fields* 中列出的相同）映射到其类型的字典。

```
>>> ast.TypeVar._field_types
{'name': <class 'str'>, 'bound': ast.expr | None, 'default_value': ast.expr | None}
```

Added in version 3.13.

lineno

col_offset

end_lineno

end_col_offset

`ast.expr` 和 `ast.stmt` 的子类的实例的属性包括*lineno*、*col_offset*、*end_lineno* 和*end_col_offset*。*lineno* 和*end_lineno* 是源码中属于该节点的部分从哪一行开始，到哪一行结束（数字 1 指第一行，以此类推）；*col_offset* 和*end_col_offset* 是第一个和最后一个属于该节点的 token 的 UTF-8 字节偏移量。记录 UTF-8 偏移量的原因是解析器内部使用 UTF-8。

注意编译器不需要结束位置，所以结束位置是可选的。结束偏移在最后一个符号之后，例如你可以通过 `source_line[node.col_offset : node.end_col_offset]` 获得一个单行表达

式节点的源码片段。

`ast.T` 类的构造器像下面这样解析它的参数：

- 如果只用位置参数，参数的数量必须和 `T._fields` 中的项一样多；它们会按顺序赋值到这些属性上。
- 如果有关键字参数，它们会为与其关键字同名的属性赋值。

比方说，要创建和填充节点*ast.UnaryOp*，你得用

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

如果从构造器中省略一个在语法中可选的字段，则其默认值为 `None`。如果省略一个列表字段，则其默认值为空列表。如果省略一个 `ast.expr_context` 类型的字段，则其默认值为*Load()*。如果省略任何其他字段，则会引发*DeprecationWarning* 并且 AST 节点将不包含此字段。在 Python 3.15，这种情况将引发一个错误。

在 3.8 版本发生变更: *ast.Constant* 类现在用于所有常量。

在 3.9 版本发生变更: 简单索引由它们的值表示，扩展切片表示为元组。

自 3.8 版本弃用: 原有的类 `ast.Num`, `ast.Str`, `ast.Bytes`, `ast.NameConstant` 和 `ast.Ellipsis` 仍然可用，但它们将在未来的 Python 发布版中被移除。同时，实例化它们将返回其他某个类的实例。

自 3.9 版本弃用: 原有的类 `ast.Index` and `ast.ExtSlice` 仍然可用，但它们将在未来的 Python 发布版中被移除。同时，实例化它们将返回其他某个类的实例。

Deprecated since version 3.13, will be removed in version 3.15: 之前版本的 Python 允许创建缺少必需字段的 AST 节点。类似地，AST 节点构造器也允许任意用于设置 AST 节点属性的关键字参数，即使它们不能匹配任何 AST 节点的字段。此行为已被弃用并将在 Python 3.15 中移除。

备注

在此显示的特定节点类的描述最初是改编自杰出的 Green Tree Snakes 项目及其所有贡献者。

根节点

class `ast.`**Module**(*body*, *type_ignores*)

一个 Python 模块，用于 文件输入。由*ast.parse()* 以默认 `"exec"` *mode* 生成的节点类型。

`body` 是由该模块的*语句* 组成的*list*。

`type_ignores` 是由该模块的类型忽略注释组成的*list*；请参阅*ast.parse()* 了解详情。

```
>>> print(ast.dump(ast.parse('x = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))])
```

class `ast.`**Expression**(*body*)

单个 Python 表达式输入。当 *mode* 为 `"eval"` 时由*ast.parse()* 所生成的节点类型。

`body` 为单独节点，是某一个*表达式类型*。

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**Interactive**(*body*)

单个 交互式输入，就像在 tut-interac 中一样。当 *mode* 为 "single" 时由*ast.parse()* 所生成的节点类型。

body 是由*语句节点* 组成的*list*。

```
>>> print(ast.dump(ast.parse('x = 1; y = 2', mode='single'), indent=4))
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
            value=Constant(value=2))])
```

class ast.**FunctionType**(*argtypes*, *returns*)

函数的旧风格类型注释表示形式，因为 Python 3.5 之前的版本不支持 **PEP 484** 标注。当 *mode* 为 "func_type" 时由*ast.parse()* 所生成的节点类型。

此种类型注释的形式是这样的:

```
def sum_two_number(a, b):
    # type: (int, int) -> int
    return a + b
```

argtypes 是由*表达式节点* 组成的*list*。

returns 是单独的*表达式节点*。

```
>>> print(ast.dump(ast.parse('(int, str) -> List[int]', mode='func_type'), indent=4))
FunctionType(
    argtypes=[
        Name(id='int', ctx=Load()),
        Name(id='str', ctx=Load())],
    returns=Subscript(
        value=Name(id='List', ctx=Load()),
        slice=Name(id='int', ctx=Load()),
        ctx=Load()))
```

Added in version 3.8.

字面值

class ast.**Constant**(*value*)

一个常量。Constant 字面值的 value 属性即为其代表的 Python 对象。它可以代表简单的数字，字符串或者 None 对象，但是也可以代表所有元素都是常量的不可变容器（例如元组或冻结集合）。

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**FormattedValue**(*value*, *conversion*, *format_spec*)

节点是以一个 f-字符串形式的格式化字段来代表的。如果该字符串只包含单个格式化字段而没有任何其他内容则节点可以被隔离，否则它将在*JoinedStr* 中出现。

- value 为任意的表达式节点（如一个字面值、变量或函数调用）。
- conversion 是一个整数:
  - -1: 无格式化

- 115: `!s` 字符串格式化
- 114: `!r` repr 格式化
- 97: `!a` ascii 格式化

- `format_spec` 是一个代表值的格式化的*JoinedStr* 节点，或者如果未指定格式则为 `None`。`conversion` 和 `format_spec` 可以被同时设置。

class `ast.`**JoinedStr**(*values*)

一个 f-字符串，由一系列*FormattedValue* 和*Constant* 节点组成。

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'), indent=4))
Expression(
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())]),
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='.3')]))])))
```

class `ast.`**List**(*elts*, *ctx*)

class `ast.`**Tuple**(*elts*, *ctx*)

一个列表或元组。`elts` 保存一个代表元素的节点的列表。`ctx` 在容器为赋值的目标时 (如 `(x, y)=something`) 是*Store*，否则是*Load*。

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'), indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'), indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class `ast.`**Set**(*elts*)

一个集合。`elts` 保存一个代表集合的元组的节点的列表。

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'), indent=4))
Expression(
    body=Set(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class ast.**Dict**(*keys*, *values*)

一个字典。keys 和 values 保存分别代表键和值的节点的列表，按照匹配的顺序（即当调用 dictionary.keys() 和 dictionary.values() 时将返回的结果）。

当使用字典字面值进行字典解包操作时要扩展的表达式放入 values 列表，并将 None 放入 keys 的对应位置。

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'), indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

变量

class ast.**Name**(*id*, *ctx*)

一个变量名。id 将名称保存为字符串，而 ctx 为下列类型之一。

class ast.**Load**

class ast.**Store**

class ast.**Del**

变量引用可被用来载入一个变量的值，为其赋一个新值，或是将其删除。变量引用会给出一个上下文来区分这几种情况。

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])])
```

class ast.**Starred**(*value*, *ctx*)

一个 *var 变量引用。value 保存变量，通常为一个*Name* 节点。此类型必须在构建*Call* 节点并传入 *args 时被使用。

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
```

(续下页)

(接上页)

```
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
                value=Name(id='it', ctx=Load()))])
```

表达式

class ast.**Expr**(*value*)

当一个表达式，例如函数调用，本身作为一个语句出现并且其返回值未被使用或存储时，它会被包装在此容器中。value 保存本节中的其他节点之一，一个*Constant*, *Name*, *Lambda*, *Yield* 或者*YieldFrom* 节点。

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load()))))])
```

class ast.**UnaryOp**(*op*, *operand*)

一个单目运算。op 是运算符，而 operand 是任意表达式节点。

class ast.**UAdd**

class ast.**USub**

class ast.**Not**

class ast.**Invert**

单目运算符对应的形符。*Not* 是 not 关键字，*Invert* 是 ~ 运算符。

```
>>> print(ast.dump(ast.parse('not x', mode='eval'), indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.**BinOp**(*left*, *op*, *right*)

一个双目运算（如相加或相减）。op 是运算符，而 left 和 right 是任意表达式节点。

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.**Add**

class ast.**Sub**

class ast.**Mult**

class ast.**Div**

class ast.**FloorDiv**

class ast.**Mod**

class ast.**Pow**

class ast.**LShift**

class ast.**RShift**

class ast.**BitOr**

class ast.**BitXor**

class ast.**BitAnd**

class ast.**MatMult**

双目运算符对应的形符。

class ast.**BoolOp**(*op*, *values*)

一个布尔运算，'or' 或者'and'。`op` 是*Or* 或者*And*。`values` 是参与运算的值。具有相同运算符的连续运算，如 `a or b or c`，会被折叠为具有多个值的单个节点。

这不包括 `not`，它属于*UnaryOp*。

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'), indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.**And**

class ast.**Or**

布尔运算符对应的形符。

class ast.**Compare**(*left*, *ops*, *comparators*)

两个或更多值之间的比较运算。`left` 是参加比较的第一个值，`ops` 是由运算符组成的列表，而 `comparators` 是由参加比较的第一个元素之后的值组成的列表。

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'), indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.**Eq**

class ast.**NotEq**

class ast.**Lt**

class ast.**LtE**

class ast.**Gt**

class ast.**GtE**

class ast.**Is**

class ast.**IsNot**

class ast.**In**

class ast.**NotIn**

比较运算符对应的形符。

class ast.**Call**(*func*, *args*, *keywords*)

一个函数调用。`func` 是函数，它通常是一个*Name* 或*Attribute* 对象。对于其参数:

- `args` 保存由按位置传入的参数组成的列表。
- `keywords` 保存了一个代表以关键字传入的参数的*keyword* 对象的列表。

`args` 和 `keywords` 参数是可选的并且默认为空列表。

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class ast.**keyword**(*arg*, *value*)

传给函数调用或类定义的关键字参数。arg 是形参名称对应的原始字符串，value 是要传入的节点。

class ast.**IfExp**(*test*, *body*, *orelse*)

一个表达式例如 `a if b else c`。每个字段保存一个单独节点，因而在下面的示例中，三个节点均为*Name* 节点。

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'), indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class ast.**Attribute**(*value*, *attr*, *ctx*)

属性访问，例如 `d.keys`。value 是一个节点，通常为*Name*。attr 是一个给出属性名称的纯字符串，而 ctx 根据属性操作的方式可以为*Load*, *Store* 或*Del*。

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'), indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class ast.**NamedExpr**(*target*, *value*)

一个带名称的表达式。此 AST 节点是由赋值表达式运算符（或称海象运算符）产生的。与第一个参数可以有多个节点的*Assign* 节点不同，在此情况下 target 和 value 都必须为单独节点。

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'), indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

Added in version 3.8.

抽取

class ast.**Subscript**(*value*, *slice*, *ctx*)

抽取操作，如 `l[1]`。value 是被抽取的对象（通常为序列或映射）。slice 是索引号、切片或键。它可以是一个包含*Slice* 的*Tuple*。ctx 根据抽取所执行的操作可以为*Load*, *Store* 或*Del*。

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.**Slice**(*lower*, *upper*, *step*)

常规切片 (形式如 `lower:upper` 或 `lower:upper:step`)。只能在*Subscript* 的 *slice* 字段内部出现，可以是直接切片对象或是作为*Tuple* 的元素。

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

推导式

class ast.**ListComp**(*elt*, *generators*)

class ast.**SetComp**(*elt*, *generators*)

class ast.**GeneratorExp**(*elt*, *generators*)

class ast.**DictComp**(*key*, *value*, *generators*)

列表和集合推导式、生成器表达式以及字典推导式。`elt` (或 `key` 和 `value`) 是一个代表将针对每个条目被求值的部分的单独节点。

`generators` 是一个由*comprehension* 节点组成的列表。

```
>>> print(ast.dump(
...     ast.parse('[x for x in numbers]', mode='eval'),
...     indent=4,
... ))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x: x**2 for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=DictComp(
        key=Name(id='x', ctx=Load()),
        value=BinOp(
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
```

(续下页)

(接上页)

```
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
```

class ast.**comprehension**(*target*, *iter*, *ifs*, *is_async*)

推导式中的一个 `for` 子句。`target` 是针对每个元素使用的引用——通常为一个*Name* 或*Tuple* 节点。`iter` 是要执行迭代的对象。`ifs` 是一个由测试表达式组成的列表：每个 `for` 子句都可以拥有多个 `ifs`。

`is_async` 表明推导式是异步的 (使用 `async for` 而不是 `for`)。它的值是一个整数 (0 或 1)。

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c in line]', mode='eval'),
...                indent=4)) # 多个推导式合一。
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())]),
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if n<10)', mode='eval'),
...                indent=4)) # 生成器推导式
Expression(
    body=GeneratorExp(
        elt=BinOp(
            left=Name(id='n', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='n', ctx=Store()),
                iter=Name(id='it', ctx=Load()),
                ifs=[
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
```

(续下页)

(接上页)

```
                            Constant(value=5)]),
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Lt()],
                        comparators=[
                            Constant(value=10)])],
                is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]', mode='eval'),
...                indent=4)) # 异步推导式
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                is_async=1)]))
```

语句

class ast.**Assign**(*targets*, *value*, *type_comment*)

一次赋值。`targets` 是一个由节点组成的列表，而 `value` 是一个单独节点。

`targets` 中有多个节点表示将同一个值赋给多个目标。解包操作是通过在 `targets` 中放入一个*Tuple* 或*List* 来表示的。

type_comment

`type_comment` 是带有以注释表示的类型标注的可选的字符串。

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) # 多重赋值
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) # 解包
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
            value=Name(id='c', ctx=Load()))])
```

class ast.**AnnAssign**(*target*, *annotation*, *value*, *simple*)

带有类型标注的赋值。`target` 是单独的节点并可以是一个*Name*, *Attribute* 或*Subscript*。`annotation` 是标注，例如一个*Constant* 或*Name* 节点。`value` 是单独的可选节点。

`simple` 将始终为 0 (表示一个“复杂”目标) 或 1 (表示一个“简单”目标)。“简单”目标仅由一个两边不带圆括号的*Name* 结点组成；所有其他目标均被视为复杂目标。只有简单目标会出现在模块和类的 `__annotations__` 字典中。

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) # 带有圆括号的标注
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
            simple=0)])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) # 属性标注
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) # 下标标注
Module(
    body=[
        AnnAssign(
            target=Subscript(
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])
```

class ast.**AugAssign**(*target*, *op*, *value*)

增强赋值，如 `a += 1`。在下面的例子中，`target` 是一个针对 `x` (带有*Store* 上下文) 的*Name* 节点，`op` 为*Add*，而 `value` 是一个值为 1 的*Constant*。

`target` 属性不可以是*Tuple* 或*List* 类，这与*Assign* 的目标不同。

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))])
```

class ast.**Raise**(*exc*, *cause*)

一条 `raise` 语句。`exc` 是要引发的异常，对于一个单独的 `raise` 通常为*Call* 或*Name*，或者为 `None`。`cause` 是针对 `raise x from y` 中 `y` 的可选部分。

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
    body=[
        Raise(
```

(续下页)

(接上页)

```
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))])
```

class ast.**Assert**(*test*, *msg*)

一条断言。test 保存条件，例如为一个*Compare* 节点。msg 保存失败消息。

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))])
```

class ast.**Delete**(*targets*)

代表一条 del 语句。targets 是一个由节点组成的列表，例如*Name*，*Attribute* 或*Subscript* 节点。

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
                Name(id='y', ctx=Del()),
                Name(id='z', ctx=Del())])])
```

class ast.**Pass**

一条 pass 语句。

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()])
```

class ast.**TypeAlias**(*name*, *type_params*, *value*)

通过 type 语句创建的类型别名。name 是别名的名称，type_params 是类型形参 的列表，而 value 是类型别名的值。

```
>>> print(ast.dump(ast.parse('type Alias = int'), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            value=Name(id='int', ctx=Load()))])
```

Added in version 3.12.

其他仅在函数或循环内部可用的语句将在其他小节中描述。

导入

class ast.**Import**(*names*)

一条导入语句。names 是一个由*alias* 节点组成的列表。

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
```

(续下页)

(接上页)

```
                alias(name='y'),
                alias(name='z')])])
```

class ast.**ImportFrom**(*module*, *names*, *level*)

代表 `from x import y`。`module` 是一个'from'名称的原始字符串，不带任何前导点号，或者为 `None` 表示 `from . import foo` 这样的语句。`level` 是一个保存相对导入层级的整数（0 表示绝对导入）。

```
>>> print(ast.dump(ast.parse('from y import x,y,z'), indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)])
```

class ast.**alias**(*name*, *asname*)

两个形参均为名称的原始字符串。如果要使用常规名称则 `asname` 可以为 `None`。

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b, c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)])
```

控制流

> **备注**
>
> 可选的子句如 `else` 如果不存在则会被存储为一个空列表。

class ast.**If**(*test*, *body*, *orelse*)

一条 `if` 语句。`test` 保存一个单独节点，如一个*Compare* 节点。`body` 和 `orelse` 各自保存一个节点列表。

`elif` 子句在 AST 中没有特别的表示形式，而是作为上文介绍的 `orelse` 部分之内的一个额外*If* 节点出现。

```
>>> print(ast.dump(ast.parse("""
... if x:
...     ...
... elif y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
```

(续下页)

(接上页)

```
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

class ast.**For**(*target*, *iter*, *body*, *orelse*, *type_comment*)

一个 `for` 循环。`target` 保存循环赋值的变量，是一个单独的*Name*, *Tuple*, *List*, *Attribute* 或*Subscript* 节点。`iter` 保存要被循环的条目，同样也是一个单独节点。`body` 和 `orelse` 包含要执行的节点列表。`orelse` 中的语句会在循环正常结束时被执行，而不是通过 `break` 语句执行。

type_comment

`type_comment` 是带有以注释表示的类型标注的可选的字符串。

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**While**(*test*, *body*, *orelse*)

一个 `while` 循环。`test` 保存条件，如一个*Compare* 节点。

```
>>> print(ast.dump(ast.parse("""
... while x:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**Break**

class ast.**Continue**

`break` 和 `continue` 语句。

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
                    orelse=[
                        Continue()])])])
```

class ast.**Try**(*body*, *handlers*, *orelse*, *finalbody*)

try 代码块。所有属性都是要执行的节点列表，除了 handlers，它是一个*ExceptHandler* 节点列表。

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except Exception:
...     ...
... except OtherException as e:
...     ...
... else:
...     ...
... finally:
...     ...
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException', ctx=Load()),
                    name='e',
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])],
            orelse=[
```

(续下页)

(接上页)

```
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**TryStar**(*body*, *handlers*, *orelse*, *finalbody*)

try 代码块后带有 except* 子句。包含的属性与*Try* 的相同但*ExceptHandler* 节点在 handlers 中会被解读为 except* 而不是 except 代码块。

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except* Exception:
...     ...
... """), indent=4))
Module(
    body=[
        TryStar(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.11.

class ast.**ExceptHandler**(*type*, *name*, *body*)

一个单独的 except 子句。type 是它将匹配的异常，通常为一个*Name* 节点（或 None 表示捕获全部的 except: 子句）。name 是一个用于存放异常的别名的原始字符串，或者如果子句没有 as foo 则为 None。body 为一个节点列表。

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])])])
```

class ast.**With**(*items*, *body*, *type_comment*)

一个 with 代码块。items 是一个代表上下文管理器的*withitem* 节点列表，而 body 是该上下文中的缩进代码块。

type_comment

type_comment 是带有以注释表示的类型标注的可选的字符串。

class ast.**withitem**(*context_expr*, *optional_vars*)

一个 with 代码块中单独的上下文管理器。context_expr 为上下文管理器，通常为一个*Call* 节点。optional_vars 为一个针对 as foo 部分的*Name*, *Tuple* 或*List*，或者如果未使用别名则为 None。

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...    something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b', ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d', ctx=Store()))],
            body=[
                Expr(
                    value=Call(
                        func=Name(id='something', ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
                            Name(id='d', ctx=Load())])])])
```

模式匹配

class ast.**Match**(*subject*, *cases*)

一条 match 语句。subject 保存匹配的目标（与 cases 相匹配的对象）而 cases 包含一个由不同分支的*match_case* 节点组成的可迭代对象。

Added in version 3.10.

class ast.**match_case**(*pattern*, *guard*, *body*)

一个 match 语句中单独的 case 模式。pattern 包含目标将要去匹配的匹配模式。请注意针对模式所产生的*AST* 节点不同于针对表达式所产生的节点，即使它们共享相同的语法。

guard 属性包含一个当模式与目标相匹配时将被求值的表达式。

body 包含一个当模式匹配并且对 guard 表达式求值的结果为真时要执行的节点列表。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] if x>0:
...         ...
...     case tuple():
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchAs(name='x')]),
                    guard=Compare(
```

(续下页)

(接上页)

```
                        left=Name(id='x', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=0)]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='tuple', ctx=Load())),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])]))
```

Added in version 3.10.

class ast.**MatchValue**(*value*)

一个按相等性进行比较的匹配字面值或值模式。`value` 为一个表达式节点。允许的值节点被限制为 match 语句文档中所描述的节点。如果匹配目标等于 value 的求值结果则模式匹配成功。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case "Relevant":
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchValue(
                        value=Constant(value='Relevant')),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchSingleton**(*value*)

一个按标识号进行比较的匹配字面值模式。`value` 为用于比较的单例对象: `None`, `True` 或 `False`。如果匹配目标为给定的常量则该模式匹配成功。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case None:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSingleton(value=None),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchSequence**(*patterns*)

一个匹配序列模式。patterns 包含当目标为一个序列时要与目标元素进行匹配的模式。如果某一子模式为 MatchStar 节点则将匹配一个变长度序列，否则将匹配一个固定长度序列。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchStar**(*name*)

匹配一个可变长度匹配序列模式中的剩余部分序列。如果 name 不为 None，则当整个序列模式匹配成功时将把一个包含剩余序列元素的列表绑定到该名称。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2, *rest]:
...         ...
...     case [*_]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2)),
                            MatchStar(name='rest')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchStar()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchMapping**(*keys*, *patterns*, *rest*)

一个匹配的映射模式。`keys` 为一个由表达式节点组成的序列。`patterns` 为一个由对应的模式节点组成的序列。`rest` 是一个可被指定用来捕获剩余映射元素的可选名称。允许的关键字表达式被限制为与 match 语句文档中所描述的一致。

如果目标为一个映射、所有被求值的表达式都存在于该映射中，并且对应于每个键的值都与对应的子模式相匹配则此模式匹配成功。如果 `rest` 不为 `None`，则当整个映射模式匹配成功时会将一个包含剩余映射元素的字典绑定到该名称。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case {1: _, 2: _}:
...         ...
...     case {**rest}:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchMapping(
                        keys=[
                            Constant(value=1),
                            Constant(value=2)],
                        patterns=[
                            MatchAs(),
                            MatchAs()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchMapping(rest='rest'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchClass**(*cls*, *patterns*, *kwd_attrs*, *kwd_patterns*)

一个 match 类模式。`cls` 为一个给出要匹配的名义类的表达式。`patterns` 为一个由要与该类所定义的模式匹配属性相匹配的模式节点组成的序列。`kwd_attrs` 为一个由要匹配的附加属性（指定为该类模式中的关键字参数）组成的序列，`kwd_patterns` 为对应的模式（指定为该类模式中的关键字值）。

如果目标为被指名类的一个实例、所有的位置模式都与对应的类定义属性相匹配，并且任何被指定的关键字属性都与其对应的模式相匹配则此模式匹配成功。

注意：类可能会定义一个返回自身的特征属性以便能将一个模式节点与被匹配的实例相匹配。某些内置类型也是以这种方式来匹配的，与 match 语句文档中所描述的一致。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case Point2D(0, 0):
...         ...
...     case Point3D(x=0, y=0, z=0):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
```

(续下页)

(接上页)

```
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchAs**(*pattern*, *name*)

一个匹配"as-模式"、捕获模式或通配符模式。`pattern` 包含将要与目标相匹配的匹配模式。如果模式为 `None`，则该节点代表一个捕获模式（即一个简单的名称）并将总是会成功。

`name` 属性包含当模式匹配成功时将要绑定的名称。如果 `name` 为 `None`，则 `pattern` 也必须为 `None` 并且该节点代表的是通配符模式。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] as y:
...         ...
...     case _:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchAs(
                        pattern=MatchSequence(
                            patterns=[
                                MatchAs(name='x')]),
                        name='y'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
```

(续下页)

(接上页)

```
                    pattern=MatchAs(),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchOr**(*patterns*)

一个匹配"or-模式"。or-模式会依次将其每个子模式与目标相匹配，直到有一个匹配成功。此时该 or-模式将被视为匹配成功。如果没有一个子模式匹配成功则该 or-模式匹配失败。`patterns` 属性包含一个由将与目标相匹配的匹配模式节点组成的列表。

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[
                                    MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

类型形参

类型形参可以存在于类、函数和类型别名中。

class ast.**TypeVar**(*name*, *bound*, *default_value*)

一个*typing.TypeVar*。`name` 是类型变量的名称。`bound` 是边界或约束，如果有的话。如果 `bound` 是一个*Tuple*，则它是表示约束；否则它是表示边界。`default_value` 是默认值；如果 `TypeVar` 没有默认值，该属性将被设为 `None`。

```
>>> print(ast.dump(ast.parse("type Alias[T: int = bool] = list[T]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVar(
                    name='T',
                    bound=Name(id='int', ctx=Load()),
                    default_value=Name(id='bool', ctx=Load()))],
            value=Subscript(
                value=Name(id='list', ctx=Load()),
                slice=Name(id='T', ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

在 3.13 版本发生变更: 增加了 *default_value* 形参。

class ast.**ParamSpec**(*name*, *default_value*)

一个*typing.ParamSpec*。name 是形参规则说明的名称。default_value 是默认值；如果 ParamSpec 没有默认值，该属性将被设为 None。

```
>>> print(ast.dump(ast.parse("type Alias[**P = [int, str]] = Callable[P, int]"),␣
↪indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                ParamSpec(
                    name='P',
                    default_value=List(
                        elts=[
                            Name(id='int', ctx=Load()),
                            Name(id='str', ctx=Load())],
                        ctx=Load()))],
            value=Subscript(
                value=Name(id='Callable', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Name(id='P', ctx=Load()),
                        Name(id='int', ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

在 3.13 版本发生变更: 增加了 *default_value* 形参。

class ast.**TypeVarTuple**(*name*, *default_value*)

一个*typing.TypeVarTuple*。name 是类型变量元组的名称。default_value 是默认值；如果 TypeVarTuple 没有默认值，该属性将被设为 None。

```
>>> print(ast.dump(ast.parse("type Alias[*Ts = ()] = tuple[*Ts]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVarTuple(
                    name='Ts',
                    default_value=Tuple(ctx=Load()))],
            value=Subscript(
                value=Name(id='tuple', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Starred(
                            value=Name(id='Ts', ctx=Load()),
                            ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

在 3.13 版本发生变更: 增加了 *default_value* 形参。

函数与类定义

class ast.**FunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

一个函数定义。

- name 是函数名称的原始字符串。
- args 是一个*arguments* 节点。
- body 是函数内部的节点列表。
- decorator_list 是要应用的装饰器列表，最外层的最先保存（即列表中的第一项将最后被应用）。
- returns 是返回标注。
- type_params 是一个类型形参 的列表。

type_comment

type_comment 是带有以注释表示的类型标注的可选的字符串。

在 3.12 版本发生变更: 增加了 type_params。

class ast.**Lambda**(*args*, *body*)

lambda 是可在表达式内部使用的最小化函数定义。不同于*FunctionDef*，body 是保存一个单独节点。

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    args=[
                        arg(arg='x'),
                        arg(arg='y')]),
                body=Constant(value=Ellipsis)))])
```

class ast.**arguments**(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)

函数的参数。

- posonlyargs, args 和 kwonlyargs 均为*arg* 节点的列表。
- vararg 和 kwarg 均为单独的*arg* 节点，指向 *args, **kwargs 形参。
- kw_defaults 是一个由仅限关键字参数默认值组成的列表。如果有一个为 None，则对应的参数为必须的参数。
- defaults 是一个由可按位置传入的参数的默认值组成的列表。如果默认值个数少于参数个数，则它们将对应最后 n 个参数。

class ast.**arg**(*arg*, *annotation*, *type_comment*)

列表中的一个单独参数。arg 为参数名称原始字符串；annotation 为其标，如一个*Name* 节点。

type_comment

type_comment 是一个可选的将注释用作类型标注的字符串。

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
```

(续下页)

(接上页)

```
            name='f',
            args=arguments(
                args=[
                    arg(
                        arg='a',
                        annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
                kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))])
```

class ast.**Return**(*value*)

一条 return 语句。

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))])
```

class ast.**Yield**(*value*)

class ast.**YieldFrom**(*value*)

一个 yield 或 yield from 表达式。因为这些属性表达式，所以如果发回的值未被使用则必须将它们包装在*Expr* 节点中。

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load()))))])

>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load()))))])
```

class ast.**Global**(*names*)

class ast.**Nonlocal**(*names*)

global 和 nonlocal 语句。names 为一个由原始字符串组成的列表。

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])])

>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
        Nonlocal(
            names=[
                'x',
                'y',
                'z'])])
```

class ast.**ClassDef**(*name*, *bases*, *keywords*, *body*, *decorator_list*, *type_params*)

一个类定义。

- `name` 为类名称的原始字符串。
- `bases` 为一个由显式指明的基类节点组成的列表。
- `keywords` 是一个*keyword* 节点的列表，主要用于‘元类’。其他关键字将被传给相应的元类，参见 **PEP 3115**。
- `body` 是一个由代表类定义内部代码的节点组成的列表。
- `decorator_list` 是一个节点的列表，与*FunctionDef* 中的一致。
- `type_params` 是一个*类型形参* 的列表。

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
...     pass
... """), indent=4))
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())])])
```

在 3.12 版本发生变更: 增加了 `type_params`。

async 与 await

class ast.**AsyncFunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

一个 async def 函数定义。具有与*FunctionDef* 相同的字段。

在 3.12 版本发生变更: 增加了 type_params。

class ast.**Await**(*value*)

一个 await 表达式。value 是它所等待的值。仅在*AsyncFunctionDef* 的函数体内可用。

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load())))])])
```

class ast.**AsyncFor**(*target*, *iter*, *body*, *orelse*, *type_comment*)

class ast.**AsyncWith**(*items*, *body*, *type_comment*)

async for 循环和 async with 上下文管理器。它们分别具有与*For* 和*With* 相同的字段。仅在*AsyncFunctionDef* 的函数体内可用。

备注

当一个字符串由*ast.parse()* 来解析时，所返回的树中的运算符节点 (为 ast.operator, ast.unaryop, ast.cmpop, ast.boolop 和 ast.expr_context 的子类) 将均为单例对象。对其中某一个 (例如*ast.Add*) 的修改将反映到同一个值所出现的其他位置上。

33.1.3 ast 中的辅助函数

除了节点类，*ast* 模块里为遍历抽象语法树定义了这些工具函数和类:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*, *, *type_comments=False*, *feature_version=None*, *optimize=-1*)

将 source 解析为一个 AST 节点。等价于 compile(source, filename, mode, flags=FLAGS_VALUE, optimize=optimize)，其中当 optimize <= 0 时 FLAGS_VALUE 为 ast.PyCF_ONLY_AST 否则为 ast.PyCF_OPTIMIZED_AST。

如果给出 type_comments=True，解析器会被修改只检查并返回 **PEP 484** 和 **PEP 526** 所规定的类型注释。这相当于将*ast.PyCF_TYPE_COMMENTS* 添加到传给*compile()* 的旗标中。这将报告针对未正确放置类型注释的语法错误。没有这个旗标，类型注释将被忽略，而选定的 AST 节点上的 type_comment 字段将始终为 None。此外，# type: ignore 注释的位置将作为*Module* 的 type_ignores 属性被返回（在其他情况下则总是为空列表）。attribute of (otherwise it is always an empty list).

并且，如果 mode 为 'func_type'，则输入语法会进行与 **PEP 484** ”签名类型注释” 对应的修改，例如 (str, int) -> List[str]。

将 feature_version 设为元组 (major, minor) 将导致使用相应 Python 版本的语法来 “尽力” 尝试解析。例如，设置 feature_version=(3, 9) 将尝试禁止解析 match 语句。目前 major 必须等于 3。最低的受支持版本为 (3, 7) (并且这可能在未来的 Python 版本中增高)；最高版本为 sys.version_info[0:2]。 “尽力” 尝试意味着不能保证解析（或解析成功）与在对应于 feature_version 的 Python 版本上运行时相同。

如果源包含一个空字符 (\0)，则会引发*ValueError*。

警告

请注意成功将源代码解析为 AST 对象并不能保证该源代码提供了可被执行的有效 Python 代码，因为编译步骤还可能引发其他的*SyntaxError* 异常。例如，对于源代码 `return 42` 可以生成一条有效的 return 语句 AST 节点，但它不能被单独编译（它必须位于一个函数节点之内）。

特别地，*ast.parse()* 不会执行任何作用域检查，这是由编译步骤来做的。

警告

足够复杂或是巨大的字符串可能导致 Python 解释器的崩溃，因为 Python 的 AST 编译器是有栈深限制的。

在 3.8 版本发生变更: 增加了 `type_comments`, `mode='func_type'` 和 `feature_version`。

在 3.13 版本发生变更: 现在最低的支持 `feature_version` 的版本为 `(3, 7)`。增加了 `optimize` 参数。

`ast.`**`unparse`**(*ast_obj*)

反向解析一个*ast.AST* 对象并生成一个包含当再次使用*ast.parse()* 解析时将产生同样的*ast.AST* 对象的代码的字符串。

警告

所产生的代码字符串将不一定与生成*ast.AST* 对象的原始代码完全一致（不带任何编译器优化，例如常量元组/冻结集合等）。

警告

尝试反向解析一个高度复杂的表达式可能会导致*RecursionError*。

Added in version 3.9.

`ast.`**`literal_eval`**(*node_or_string*)

对表达式节点或仅包含 Python 字面值或容器表示形式的字符串进行求值。所提供的字符串或节点可能只包含下列 Python 字面值结构：字符串、字节串、数字、元组、字典、集合、布尔值、`None` 和 `Ellipsis`。

这可被用于对包含 Python 值的字符串进行求值而不必解析这些值本身。它不能对任意的复杂表达式进行求值，例如涉及运算符或索引操作的表达式。

此函数过去曾被描述为“安全”但并未定义其含义。这是存在误导性的。此函数被专门设计为不执行 Python 代码，这与更通用的*eval()* 不同。它没有命名空间、没有名称查找或执行外部调用的能力。但是它并不能完全避免攻击：一个相对较小的输入有可能导致内存耗尽或 C 栈耗尽，使得进程崩溃。还可能在某些输入上出现过度消耗 CPU 的拒绝服务问题。因此不建议在未被信任的数据上调用它。

警告

有可能由于 Python 的 AST 编译器中的栈深度限制而导致 Python 解释器的崩溃。

根据输入错误的具体形式它可能引发*ValueError*, *TypeError*, *SyntaxError*, *MemoryError* 以及*RecursionError*。

在 3.2 版本发生变更: 目前支持字节和集合。

在 3.9 版本发生变更: 现在支持通过 `'set()'` 创建空集合。

在 3.10 版本发生变更: 对于字符串输入，打头的空格和制表符现在会被去除。

`ast.`**`get_docstring`**(*node*, *clean=True*)

返回给定 *node* (必须为*FunctionDef*, *AsyncFunctionDef*, *ClassDef* 或*Module* 节点) 的文档字符串，或者如果没有文档字符串则返回 `None`。如果 *clean* 为真值，则通过*inspect.cleandoc()* 清除文档字符串的缩进。

在 3.5 版本发生变更: 目前支持*AsyncFunctionDef*

`ast.`**`get_source_segment`**(*source*, *node*, ***, *padded=False*)

获取生成 *node* 的 *source* 的源代码段。如果丢失了某些位置信息 (*lineno*, *end_lineno*, *col_offset* 或*end_col_offset*)，则返回 `None`。

如果 *padded* 为 `True`，则多行语句的第一行将以与其初始位置相匹配的空格填充。

Added in version 3.8.

`ast.`**`fix_missing_locations`**(*node*)

当你使用*compile()* 来编译节点树时，编译器会接受每个支持*lineno* 和*col_offset* 属性的节点的相应信息。对于已生成的节点来说这是相当繁琐的，因此这个辅助工具会递归地为尚未设置这些属性的节点添加它们，具体做法是将其设为父节点的对应值。它将从 *node* 开始递归地执行。

`ast.`**`increment_lineno`**(*node*, *n=1*)

从 *node* 开始按 *n* 递增节点树中每个节点的行号和结束行号。这在“移动代码”到文件中的不同位置时很有用处。

`ast.`**`copy_location`**(*new_node*, *old_node*)

在可能的情况下将源位置 (*lineno*, *col_offset*, *end_lineno* 和*end_col_offset*) 从 *old_node* 拷贝到 *new_node*，并返回 *new_node*。

`ast.`**`iter_fields`**(*node*)

针对于 *node* 上在 `node._fields` 中出现的每个字段产生一个 `(fieldname, value)` 元组。

`ast.`**`iter_child_nodes`**(*node*)

产生 *node* 所有的直接子节点，也就是说，所有为节点的字段所有为节点列表的字段条目。

`ast.`**`walk`**(*node*)

递归地产生节点树中从 *node* 开始（包括 *node* 本身）的所有下级节点，没有确定的排序方式。这在你仅想要原地修改节点而不关心具体上下文时很有用处。

`class` `ast.`**`NodeVisitor`**

一个遍历抽象语法树并针对所找到的每个节点调用访问器函数的节点访问器基类。该函数可能会返回一个由*visit()* 方法所提供的值。

这个类应当被子类化，并由子类来添加访问器方法。

`visit`(*node*)

访问一个节点。默认实现会调用名为 `self.visit_`*classname* 的方法其中 *classname* 为节点类的名称，或者如果该方法不存在则为*generic_visit()*。

`generic_visit`(*node*)

该访问器会在节点的所有子节点上调用*visit()*。

请注意所有包含自定义访问器方法的节点的子节点将不会被访问除非访问器调用了*generic_visit()* 或是自行访问它们。

visit_Constant(*node*)

处理所有常量节点。

如果你想在遍历期间应用对节点的修改则请不要使用*NodeVisitor*。对此目的可使用一个允许修改的特殊访问器 (*NodeTransformer*)。

自 3.8 版本弃用: `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` 和 `visit_Ellipsis()` 等方法现在已被弃用并且在未来的 Python 版本中将不再被调用。请添加*visit_Constant()* 方法来处理所有常量节点。

class ast.**NodeTransformer**

子类*NodeVisitor* 用于遍历抽象语法树，并允许修改节点。

NodeTransformer 将遍历抽象语法树并使用 visitor 方法的返回值去替换或移除旧节点。如果 visitor 方法的返回值为 `None`，则该节点将从其位置移除，否则将替换为返回值。当返回值是原始节点时，无需替换。

如下是一个转换器示例，它将所有出现的名称 (`foo`) 重写为 `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Constant(value=node.id),
            ctx=node.ctx
        )
```

请记住如果你正在操作的节点具有子节点，你必须先自行转换这些子节点或是为该节点调用*generic_visit()* 方法。

对于属于语句集合（适用于所有语句节点）的节点，访问者还可以返回节点列表而不仅仅是单个节点。

如果*NodeTransformer* 引入了新的（不属于原节点树的一部分的）节点而没有给出它们的位置信息（如*lineno* 等），则应当调用*fix_missing_locations()* 并传入新的子节点树来重新计算位置信息:

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

通常你可以像这样使用转换器:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*, ***, *indent=None*, *show_empty=False*)

返回 *node* 中树结构的格式化转储。这主要适用于调试目的。如果 *annotate_fields* 为真值（默认），返回的字符串将显示字段的名称和值。如果 *annotate_fields* 为假值，结果字符串将通过省略无歧义的字段名称变得更为紧凑。默认情况下不会转储行号和列偏移等属性。如果需要，可将 *include_attributes* 设为真值。

如果 *indent* 是一个非负整数或者字符串，那么节点树将被美化输出为指定的缩进级别。如果缩进级别为 0、负数或者 `""` 则将只插入换行符。`None` (默认) 将选择最紧凑的表示形式。使用一个正整数将让每个级别缩进相应数量的空格。如果 *indent* 是一个字符串 (如 `"\t"`)，该字符串会被用于缩进每个级别。

如果 *show_empty* 为 `False` (默认值)，则空列表和值为 `None` 的字段将在输出中省略。

在 3.9 版本发生变更: 添加了 *indent* 选项。

在 3.13 版本发生变更: 增加了 *show_empty* 选项。

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4, show_empty=True))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()),
                            args=[],
                            keywords=[])))],
            decorator_list=[],
            type_params=[])],
    type_ignores=[])
```

33.1.4 编译器旗标

下列旗标可被传给*compile()* 用来改变程序编译的效果:

`ast.`**PyCF_ALLOW_TOP_LEVEL_AWAIT**

启用对最高层级 `await`, `async for`, `async with` 以及异步推导式的支持。

Added in version 3.8.

`ast.`**PyCF_ONLY_AST**

生成并返回一个抽象语法树而不是返回一个已编译的代码对象。

`ast.`**PyCF_OPTIMIZED_AST**

返回的 AST 已根据*compile()* 或*ast.parse()* 中的 *optimize* 参数进行了优化。

Added in version 3.13.

`ast.`**PyCF_TYPE_COMMENTS**

启用对 **PEP 484** 和 **PEP 526** 风格的类型注释的支持 (`# type: <type>`, `# type: ignore <stuff>`)。

Added in version 3.8.

33.1.5 命令行用法

Added in version 3.9.

ast 模块可以在命令行下作为脚本来执行。具体做法非常简单:

```
python -m ast [-m <mode>] [-a] [infile]
```

可以接受以下选项:

-h, --help

显示帮助信息并退出。

-m <mode>

--mode <mode>
指明哪种代码必须被编译，相当于*parse()* 中的 *mode* 参数。

--no-type-comments
不要解析类型注释。

-a, --include-attributes
包括属性如行号和列偏移。

-i <indent>
--indent <indent>
AST 中节点的缩进（空格数）。

如果指定了 `infile` 则其内容将被解析为 AST 并转储至 stdout。在其他情况下，将从 stdin 读取内容。

参见

Green Tree Snakes，一个外部文档资源，包含处理 Python AST 的完整细节。

ASTTokens 会为 Python AST 标注生成它们的源代码中的形符和文本的位置。这对执行源代码转换的工具很有帮助。

leoAst.py 通过在形符和 ast 节点之间插入双向链接统一了 python 程序基于形符的和基于解析树的视图。

LibCST 将代码解析为一个实体语法树（Concrete Syntax Tree），它看起来像是 ast 树而又保留了所有格式化细节。它对构建自动化重构（codemod）应用和代码质量检查工具很有用处。

Parso 是一个支持错误恢复和不同 Python 版本的（在多个 Python 版本中）往返解析的 Python 解析器。Parso 还能列出你的 Python 文件中的许多语法错误。

33.2 `symtable` --- 访问编译器的符号表

Source code： Lib/symtable.py

符号表由编译器在生成字节码之前根据 AST 生成。符号表负责计算代码中每个标识符的作用域。*symtable* 提供了一个查看这些表的接口。

33.2.1 符号表的生成

symtable.**symtable**(*code*, *filename*, *compile_type*)
返回 Python 源代码顶层的*SymbolTable*。*filename* 是代码文件名。*compile_type* 的含义类似*compile()* 的 *mode* 参数。

33.2.2 符号表的查看

class symtable.**SymbolTableType**
一个指明*SymbolTable* 对象的类型的枚举。

MODULE = "module"
用于模块的符号表。

FUNCTION = "function"
用于函数的符号表。

CLASS = "class"
用于类的符号表。

以下成员指向不同风格的 标注作用域。

ANNOTATION = "annotation"

当 `from __future__ import annotations` 被激活时用于标注。

TYPE_ALIAS = "type alias"

用于 `type` 构造的符号表。

TYPE_PARAMETERS = "type parameters"

用于 泛型函数或 泛型类的符号表。

TYPE_VARIABLE = "type variable"

用于正式意义下的绑定、约束元组或单个类型变量的默认值的符号变量的符号表，即 TypeVar, TypeVarTuple 或 ParamSpec 对象（后两者不支持绑定或约束元组）。

Added in version 3.13.

class symtable.**SymbolTable**

某个代码块的命名空间表。构造函数不公开。

get_type()

返回符号表的类型。可能的值为*SymbolTableType* 枚举的成员。

在 3.12 版本发生变更: 增加 `'annotation'`, `'TypeVar bound'`, `'type alias'` 和 `'type parameter'` 作为可能的返回值。

在 3.13 版本发生变更: 返回值为*SymbolTableType* 枚举的成员。

返回字符串的实际值可能在未来发生变化，因此，建议使用*SymbolTableType* 成员而不是硬编码的字符串。

get_id()

返回符号表的标识符

get_name()

返回表名称。如果表是针对类的则为类名；如果是针对函数的则为函数名；或者如果表是全局的 (*get_type()* 返回 `'module'`) 则为 `'top'`。对于类型形参作用域 (用于泛型类、函数和类型别名)，它将为底层类、函数或类型别名的名称。对于类型别名作用域，它将为类型别名的名称。对于*TypeVar* 绑定作用域，它将为 `TypeVar` 的名称。

get_lineno()

返回符号表所代表代码块的第一行编号。

is_optimized()

如果符号表中的局部变量可能被优化过，则返回 `True`。

is_nested()

如果代码块是嵌套类或函数，则返回 `True`。

has_children()

如果代码块中有嵌套的命名空间，则返回 `True`。可通过*get_children()* 读取。

get_identifiers()

返回一个包含表中符号名称的视图对象。参见视图对象文档。

lookup(*name*)

在符号表中查找 *name* 并返回一个*Symbol* 实例。

get_symbols()

返回符号表中所有符号的*Symbol* 实例的列表。

get_children()

返回嵌套符号表的列表。

class symtable.**Function**

函数或方法的命名空间。该类继承自*SymbolTable*。

get_parameters()
返回由函数的参数名组成的元组。

get_locals()
返回函数中局部变量名组成的元组。

get_globals()
返回函数中全局变量名组成的元组。

get_nonlocals()
返回一个包含在此函数中显式声明的非局部变量名称的元组。

get_frees()
返回一个包含在此函数中的*自由（闭包）变量* 名称的元组。

class symtable.**Class**
类的命名空间。该类继承自*SymbolTable*。

get_methods()
返回一个包含类中声明的方法型函数的名称的元组。

在这里，术语' 方法' 是指 任何在 class 语句体中通过 `def` 或 `async def` 定义的函数。

在更深的作用域（例如内部类）中定义的函数不会被*get_methods()* 所获取。

例如:

```
>>> import symtable
>>> st = symtable.symtable('''
... def outer(): pass
...
... class A:
...     def f():
...         def w(): pass
...
...     def g(self): pass
...
...     @classmethod
...     async def h(cls): pass
...
...     global outer
...     def outer(self): pass
... ''', 'test', 'exec')
>>> class_A = st.get_children()[1]
>>> class_A.get_methods()
('f', 'g', 'h')
```

虽然 `A().f()` 在运行时会引发*TypeError*，但 `A.f` 仍然被视为是方法型函数。

class symtable.**Symbol**
SymbolTable 中的数据项，对应于源码中的某个标识符。构造函数不公开。

get_name()
返回符号名

is_referenced()
如果符号在代码块中被引用了，则返回 `True`。

is_imported()
如果符号是由导入语句创建的，则返回 `True`。

is_parameter()
如果符号是参数，返回 `True`。

is_global()
如果符号是全局变量，则返回 True。

is_nonlocal()
如果符号为非局部变量，则返回 True。

is_declared_global()
如果符号用 global 声明为全局变量，则返回 True。

is_local()
如果符号是代码块内的局部变量，则返回 True。

is_annotated()
如果符号带有注解，则返回 True。

Added in version 3.6.

is_free()
如果符号在代码块中被引用，但未赋值，则返回 True。

is_assigned()
如果符号在代码块中赋值，则返回 True。

is_namespace()
如果符号名绑定引入了新的命名空间，则返回 True。

如果符号名用于函数或类定义语句，则为 True。

例如：

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

注意，一个符号名可以与多个对象绑定。如果结果为 True，则该符号名还可以绑定到其他对象上，比如 int 或 list，且不会引入新的命名空间。

get_namespaces()
返回与符号名绑定的命名空间的列表。

get_namespace()
返回绑定到这个名称的命名空间。如果有多个命名空间或没有命名空间被绑定到这个名称，则会引发*ValueError*。

33.2.3 命令行用法

Added in version 3.13.

symtable 模块可以在命令行下作为脚本来执行。

```
python -m symtable [infile...]
```

符号表将针对指定的 Python 文件生成并转储至 stdout。如果未指定输入文件，将从 stdin 读取内容。

33.3 token --- 用于 Python 解析树的常量

源码： Lib/token.py

该模块提供了一些代表解析树的叶子节点的数字值的常量（终端形符）。请参阅 Python 发布版中的 Grammar/Tokens 文件获取在该语言语法情境下的名称定义。这些名称所映射的特定数字值有可能在各 Python 版本间发生变化。

该模块还提供从数字代码到名称和一些函数的映射。这些函数镜像了 Python C 头文件中的定义。

请注意一个词元的值可能取决于分词器选项。例如，"+" 词元可能被报告为*PLUS* 或*OP*，而 "match" 词元可能被报告为*NAME* 或*SOFT_KEYWORD*。

token.**tok_name**

将此模块中定义的常量的数值映射回名称字符串的字典，允许生成更加人类可读的解析树表示。

token.**ISTERMINAL**(*x*)

对终端形符值返回 True。

token.**ISNONTERMINAL**(*x*)

对非终端形符值返回 True。

token.**ISEOF**(*x*)

如果 *x* 是表示输入结束的标记则返回 True。

形符常量有：

token.**NAME**

表示一个 标识符的词元值。请注意关键字也会被初始分词化为 NAME 词元。

token.**NUMBER**

表示一个 数字字面值的词元值。

token.**STRING**

表示一个 字符串或字节串字面值的词元值，不包括 格式化字符串字面值。该词元字符串不会被解读：它包括两边的引号以及前缀（如果有的话）；反斜杠将按照字面值被包括，而不会处理转义序列。

token.**OP**

表示一个 运算符或 分隔符的泛用词元。

这个值仅会由*tokenize* 模块报告。在内部，分词器会改用实际词元类型。

token.**COMMENT**

用于表示注释的词元值。解析器会忽略 COMMENT 词元。

token.**NEWLINE**

表示一个 逻辑行结束的词元值。

token.**NL**

用于表示非终结换行符的词元值。NL 词元会在一个代码逻辑行跨越了多个物理行时被生成。解析器会忽略 NL 词元。

token.**INDENT**

用于在一个 逻辑行的开头表示 缩进块的开始的词元值。

token.**DEDENT**

用于在一个 逻辑行的开头表示 缩进块的结束的词元值。

token.**FSTRING_START**

用于表示一个 格式化字符串字面值的开始的词元值。

该词元值包括前缀和开始引号，但不包括字面值的内容。

token.**FSTRING_MIDDLE**

用于一个 格式化字面值内部的字面文本包括格式说明的词元值。

替换字段（即格式化字符串的非字面值部分）使用与其他表达式相同的词元，并由*LBRACE*, *RBRACE*, *EXCLAMATION* 和*COLON* 词元来分隔。

token.**FSTRING_END**

用于表示一个 格式化字符串的结束的词元值。

该词元字符串包含结束引号。

token.**ENDMARKER**

表示输入结束的词元值。

token.**ENCODING**

指示用于将源字节解码为文本的编码的形符值。*tokenize.tokenize()* 返回的第一个形符将始终是一个 ENCODING 形符。

该词元类型不被 C 分词器所使用但却是*tokenize* 模块所需要的。

下列词元类型不会由*tokenize* 模块产生，它们是为分词器或解析器内的特殊用处而定义的。

token.**TYPE_IGNORE**

指明一个 `type: ignore` 注释已被识别的词元值。此种词元仅在设置了*PyCF_TYPE_COMMENTS* 旗标时会代替常规的*COMMENT* 词元被产生。

token.**TYPE_COMMENT**

指明一个类型注释已被识别的词元值。此种词元仅在设置了*PyCF_TYPE_COMMENTS* 旗标时会代替常规的*COMMENT* 词元被产生。

token.**SOFT_KEYWORD**

指明一个 软关键字的词元值。

分词器绝不会产生该值。要检测一个软关键字，请将一个*NAME* 词元字符串传给*keyword.issoftkeyword()*。

token.**ERRORTOKEN**

用于表示错误输入的词元值。

tokenize 模块通常用引发异常而不是发出这个词元来指明错误。它也可以发出*OP* 或*NAME* 等词元并附带将被解析器拒绝的字符串。

The remaining tokens represent specific operators and delimiters. (The *tokenize* module reports these as *OP*; see `exact_type` in the *tokenize* documentation for details.)

形符	值
token.**LPAR**	"("
token.**RPAR**	")"
token.**LSQB**	"["
token.**RSQB**	"]"
token.**COLON**	":"
token.**COMMA**	","
token.**SEMI**	";"

续下页

表 1 – 接上页

形符	值
token.**PLUS**	"+"
token.**MINUS**	"-"
token.**STAR**	"*"
token.**SLASH**	"/"
token.**VBAR**	"|"
token.**AMPER**	"&"
token.**LESS**	"<"
token.**GREATER**	">"
token.**EQUAL**	"="
token.**DOT**	"."
token.**PERCENT**	"%"
token.**LBRACE**	"{"
token.**RBRACE**	"}"
token.**EQEQUAL**	"=="
token.**NOTEQUAL**	"!="
token.**LESSEQUAL**	"<="
token.**GREATEREQUAL**	">="
token.**TILDE**	"~"

续下页

表 1 – 接上页

形符	值
token.**CIRCUMFLEX**	"^"
token.**LEFTSHIFT**	"<<"
token.**RIGHTSHIFT**	">>"
token.**DOUBLESTAR**	"**"
token.**PLUSEQUAL**	"+="
token.**MINEQUAL**	"-="
token.**STAREQUAL**	"*="
token.**SLASHEQUAL**	"/="
token.**PERCENTEQUAL**	"%="
token.**AMPEREQUAL**	"&="
token.**VBAREQUAL**	"|="
token.**CIRCUMFLEXEQUAL**	"^="
token.**LEFTSHIFTEQUAL**	"<<="
token.**RIGHTSHIFTEQUAL**	">>="
token.**DOUBLESTAREQUAL**	"**="
token.**DOUBLESLASH**	"//"
token.**DOUBLESLASHEQUAL**	"//="
token.**AT**	"@"

续下页

表 1 – 接上页

形符	值
token.**ATEQUAL**	"@="
token.**RARROW**	"->"
token.**ELLIPSIS**	"..."
token.**COLONEQUAL**	":="
token.**EXCLAMATION**	"!"

The following non-token constants are provided:

token.**N_TOKENS**

The number of token types defined in this module.

token.**EXACT_TOKEN_TYPES**

将形符字符串表示形式映射到其数字代码的字典。

Added in version 3.8.

在 3.5 版本发生变更: 增加了 AWAIT 和 ASYNC 形符。

在 3.7 版本发生变更: 形符*COMMENT*、*NL* 和*ENCODING* 形符。

在 3.7 版本发生变更: 移除了 AWAIT 和 ASYNC 形符。”async” 和”await” 被形符化为*NAME* 形符。

在 3.8 版本发生变更: 增加了*TYPE_COMMENT*, *TYPE_IGNORE*, *COLONEQUAL*。重新增加了 AWAIT 和 ASYNC 形符（需要用它们来支持解析*ast.parse()* 的 feature_version 设为 6 或更低的较旧 Python 版本）。

在 3.12 版本发生变更: Added *EXCLAMATION*.

在 3.13 版本发生变更: 重新移除了 AWAIT 和 ASYNC 形符。

33.4 keyword --- 检验 Python 关键字

源码： Lib/keyword.py

此模块使 Python 程序可以确定某个字符串是否为 关键字或 软关键字。

keyword.**iskeyword**(*s*)

如果 *s* 是一个 Python 关键字则返回 True。

keyword.**kwlist**

包含解释器定义的所有 关键字的序列。如果所定义的任何关键字仅在特定*__future__* 语句生效时被激活，它们也将被包含在内。

keyword.**issoftkeyword**(*s*)

如果 *s* 是一个 Python 软关键字则返回 True。

Added in version 3.9.

keyword.**softkwlist**

包含为解释器定义的所有软关键字的序列。如果有任何被定义的软关键字是仅在当特定*__future__*语句生效时才会被激活的，它们同样会被包括在内。

Added in version 3.9.

33.5 tokenize --- Python 源代码的分词器

源码： Lib/tokenize.py

---

tokenize 模块为 Python 源代码提供了一个词法扫描器，用 Python 实现。该模块中的扫描器也将注释作为标记返回，这使得它对于实现“漂亮的输出器”非常有用，包括用于屏幕显示的着色器。

为了简化标记流的处理，所有的 运算符和 定界符以及*Ellipsis* 返回时都会打上通用的*OP* 标记。可以通过*tokenize.tokenize()* 返回的*named tuple* 对象的 exact_type 属性来获得确切的标记类型。

警告

请注意本模块中的函数被设计为仅能解析符合语法的 Python 代码（当使用*ast.parse()* 解析代码时不会引发异常）。在提供无效的 Python 代码时本模块中函数的行为是 **未定义**的并可能在任何时候发生改变。

33.5.1 对输入进行解析标记

主要的入口是一个*generator*:

tokenize.**tokenize**(*readline*)

生成器*tokenize()* 需要一个 *readline* 参数，这个参数必须是一个可调用对象，且能提供与文件对象的*io.IOBase.readline()* 方法相同的接口。每次调用这个函数都要返回字节类型输入的一行数据。

生成器产生 5 个具有这些成员的元组：令牌类型；令牌字符串；指定令牌在源中开始的行和列的 2 元组 (srow, scol)；指定令牌在源中结束的行和列的 2 元组 (erow, ecol)；以及发现令牌的行。所传递的行（最后一个元组项）是 实际的行。5 个元组以*named tuple* 的形式返回，字段名是: type string start end line 。

返回的*named tuple* 有一个额外的属性，名为 exact_type ，包含了*OP* 标记的确切操作符类型。对于所有其他标记类型，exact_type 等于命名元组的 type 字段。

在 3.1 版本发生变更: 增加了对 named tuple 的支持。

在 3.3 版本发生变更: 添加了对 exact_type 的支持。

根据 **PEP 263** ，*tokenize()* 通过寻找 UTF-8 BOM 或编码 cookie 来确定文件的源编码。

tokenize.**generate_tokens**(*readline*)

对读取 unicode 字符串而不是字节的源进行标记。

和*tokenize()* 一样，*readline* 参数是一个返回单行输入的可调用参数。然而，*generate_tokens()* 希望 *readline* 返回一个 str 对象而不是字节。

其结果是一个产生具名元组的的迭代器，与*tokenize()* 完全一样。它不会产生*ENCODING* 标记。

所有来自*token* 模块的常量也可从*tokenize* 导出。

提供了另一个函数来逆转标记化过程。这对于创建对脚本进行标记、修改标记流并写回修改后脚本的工具很有用。

tokenize.**untokenize**(*iterable*)

将令牌转换为 Python 源代码。*iterable* 必须返回至少有两个元素的序列，即令牌类型和令牌字符串。任何额外的序列元素都会被忽略。

结果将保证会被词元化为与输入相匹配因此转换是无损的并且确保可以来回反复。此项保证只适用于词元类型和词元字符串因为词元之间的空位（列位置）可能发生改变。

它返回字节，使用*ENCODING* 标记进行编码，这是由*tokenize()* 输出的第一个标记序列。如果输入中没有编码令牌，它将返回一个字符串。

tokenize() 需要检测它所标记源文件的编码。它用来做这件事的函数是可用的：

tokenize.**detect_encoding**(*readline*)

detect_encoding() 函数用于检测解码 Python 源文件时应使用的编码。它需要一个参数，readline，与*tokenize()* 生成器的使用方式相同。

它最多调用 readline 两次，并返回所使用的编码（作为一个字符串）和它所读入的任何行（不是从字节解码的）的 list 。

它从 UTF-8 BOM 或编码 cookie 的存在中检测编码格式，如 **PEP 263** 所指明的。如果 BOM 和 cookie 都存在，但不一致，将会引发*SyntaxError*。请注意，如果找到 BOM ，将返回 `'utf-8-sig'` 作为编码格式。

如果没有指定编码，那么将返回默认的 `'utf-8'` 编码.

使用*open()* 来打开 Python 源文件：它使用*detect_encoding()* 来检测文件编码。

tokenize.**open**(*filename*)

使用由*detect_encoding()* 检测到的编码，以只读模式打开一个文件。

Added in version 3.2.

exception tokenize.**TokenError**

当文件中任何地方没有完成 docstring 或可能被分割成几行的表达式时触发，例如:

```
"""Beginning of
docstring
```

或者:

```
[1,
 2,
 3
```

33.5.2 命令行用法

Added in version 3.3.

tokenize 模块可以作为一个脚本从命令行执行。这很简单:

```
python -m tokenize [-e] [filename.py]
```

可以接受以下选项：

-h, --help

显示此帮助信息并退出

-e, --exact

使用确切的类型显示令牌名称

如果 `filename.py` 被指定，其内容会被标记到 stdout 。否则，标记化将在 stdin 上执行。

33.5.3 例子

脚本改写器的例子，它将 float 文本转换为 Decimal 对象:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """在一个语句字符串中用 Decimal 替代浮点数。

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    指数的格式继承自平台的 C 库。
    已知用例有 "e-007" (Windows) 和 "e-07" (非 Windows)。
    由于我们只显示 12 个数位，且第 13 位的值不到 5，因此
    输出的其余部分应当是依赖于具体平台的。

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    使用 Decimal 进行计算的输出应当在所有平台上保持一致。

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # 替换 NUMBER 形符
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

从命令行进行标记化的例子。脚本:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

将被标记为以下输出，其中第一列是发现标记的行 / 列坐标范围，第二列是标记的名称，最后一列是标记的值（如果有）。

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
```

(续下页)

(接上页)

```
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

可以使用*-e*选项来显示确切的标记类型名称。

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

以编程方式对文件进行标记的例子，用*generate_tokens()*读取 unicode 字符串而不是字节:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

或者通过*tokenize()*直接读取字节数据:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

33.6 tabnanny --- 检测有歧义的缩进

源代码: Lib/tabnanny.py

目前，该模块旨在作为脚本调用。但是可以使用下面描述的*check()*函数将其导入 IDE。

备注

此模块提供的 API 可能会在将来的版本中更改；此类更改可能无法向后兼容。

tabnanny.**check**(*file_or_dir*)

如果 *file_or_dir* 是目录而非符号链接，则递归地在名为 *file_or_dir* 的目录树中下行，沿途检查所有 `.py` 文件。如果 *file_or_dir* 是一个普通 Python 源文件，将检查其中的空格相关问题。诊断消息将使用*print()* 函数写入到标准输出。

tabnanny.**verbose**

此旗标指明是否打印详细消息。如果作为脚本调用则是通过 `-v` 选项来增加。

tabnanny.**filename_only**

此旗标指明是否只打印包含空格相关问题文件的文件名。如果作为脚本调用则是通过 `-q` 选项来设为真值。

exception tabnanny.**NannyNag**

如果检测到模糊缩进则由*process_tokens()* 引发。在*check()* 中捕获并处理。

tabnanny.**process_tokens**(*tokens*)

此函数由*check()* 用来处理由*tokenize* 模块所生成的标记。

参见

模块*tokenize*
用于 Python 源代码的词法扫描程序。

33.7 pyclbr --- Python 模块浏览器支持

源代码: Lib/pyclbr.py

---

pyclbr 模块提供了对于以 Python 编写的模块中定义的函数、类和方法的受限信息。这种信息足够用来实现一个模块浏览器。这种信息是从 Python 源代码中直接提取而非通过导入模块，因此该模块可以安全地用于不受信任的代码。此限制使得非 Python 实现的模块无法使用此模块，包括所有标准和可选的扩展模块。

pyclbr.**readmodule**(*module*, *path=None*)

返回一个将模块层级的类名映射到类描述器的字典。如果可能，将会包括已导入基类的描述器。形参 *module* 为要读取模块名称的字符串；它可能是某个包内部的模块名称。*path* 如果给出则为添加到 `sys.path` 开头的目录路径序列，它会被用于定位模块的源代码。

此函数为原始接口，仅保留用于向下兼容。它会返回以下内容的过滤版本。

pyclbr.**readmodule_ex**(*module*, *path=None*)

返回一个基于字典的树，其中包含与模块中每个用 `def` 或 `class` 语句定义的函数和类相对应的函数和类描述器。被返回的字典会将模块层级的函数和类名映射到它们的描述器。嵌套的对象会被输入到它们的上级子目录中。与 readmodule 一样，*module* 指明要读取的模块而 *path* 会被添加到 sys.path。如果被读取的模块是一个包，则返回的字典将具有 `'__path__'` 键，其值是一个包含包搜索路径的列表。

Added in version 3.7: 嵌套定义的描述器。它们通过新的子属性来访问。每个定义都会有一个新的上级属性。

这些函数所返回的描述器是 Function 和 Class 类的实例。用户不应自行创建这些类的实例。

33.7.1 Function 对象

class pyclbr.**Function**

Function 类的实例描述了由 def 语句所定义的函数。它们具有下列属性：

file

函数定义所在的文件名称。

module

定义了所描述函数的模块名称。

name

函数名称。

lineno

定义在文件中起始位置的行号。

parent

对于最高层级函数为 None。对于嵌套函数则为上级函数。

Added in version 3.7.

children

一个将名称映射到针对嵌套函数和类的描述器的字典。

Added in version 3.7.

is_async

True 针对使用 async 前缀定义的函数，其他情况下为 False。

Added in version 3.10.

33.7.2 Class 对象

class pyclbr.**Class**

Class 类的实例描述了由 class 语句所定义的类。它们具有与*Function* 相同的属性以及两个额外属性。

file

类定义所在的文件名称。

module

定义了所描述类的模块名称。

name

类名称。

lineno

定义在文件中起始位置的行号。

parent

对于最高层级类为 None。对于嵌套的类则为上级类。

Added in version 3.7.

children

将名称映射到嵌套函数和类描述器的字典。

Added in version 3.7.

super

一个由 Class 对象组成的列表，这些对象描述了相应类的直接基类。被指定为超类但无法被*readmodule_ex()* 发现的类会作为类名字符串而非 Class 对象列出。

methods

一个将方法名映射到行号的字典。此属性可从更新的 *children* 字典中获取，但被保留用于向下兼容。

33.8 py_compile --- 编译 Python 源文件

源代码: Lib/py_compile.py

---

py_compile 模块提供了用来从源文件生成字节码的函数和另一个用于当模块源文件作为脚本被调用时的函数。

虽然不太常用，但这个函数在安装共享模块时还是很有用的，特别是当一些用户可能没有权限在包含源代码的目录中写字节码缓存文件时。

exception py_compile.**PyCompileError**

当编译文件过程中发生错误时，抛出的异常。

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)

将源文件编译成字节码并写入字节码缓存文件。源代码将从名为 *file* 的文件中加载。字节码会被写入到 *cfile*，它默认为 **PEP 3147/PEP 488** 路径，以 `.pyc` 结尾。举例来说，如果 *file* 为 `/foo/bar/baz.py` 则对于 Python 3.2 *cfile* 将默认为 `/foo/bar/__pycache__/baz.cpython-32.pyc`。如果指定了 *dfile*，则将用它而不是 *file* 作为在异常回溯中获取并显示的源文件的名称。如果 *doraise* 为真值，则当编译 *file* 遇到错误时将引发 *PyCompileError*。如果 *doraise* 为（默认的）假值，则会将错误字符串写入到 `sys.stderr`，但不会引发异常。此函数返回已编译字节码文件的路径，即 *cfile* 所使用的值。

doraise 和 *quiet* 参数确定在编译文件时如何处理错误。如果 *quiet* 为 0 或 1，并且 *doraise* 为假值，则会启用默认行为：写入错误信息到 `sys.stderr`，并且函数将返回 `None` 而非一个路径。如果 *doraise* 为真值，则将改为引发 *PyCompileError*。但是如果 *quiet* 为 2，则不会写入消息，并且 *doraise* 也不会有效果。

如果 *cfile* 所表示（显式指定或计算得出）的路径为符号链接或非常规文件，则将引发 *FileExistsError*。此行为是用来警告如果允许写入编译后字节码文件到这些路径则导入操作将会把它们转为常规文件。这是使用文件重命名来将最终编译后字节码文件放置到位以防止并发文件写入问题的导入操作的附带效果。

optimize 控制优化级别并会被传给内置的 *compile()* 函数。默认值 `-1` 表示选择当前解释器的优化级别。

invalidation_mode 应当是 *PycInvalidationMode* 枚举的成员，它控制在运行时如何让已生成的字节码缓存失效。如果设置了 `SOURCE_DATE_EPOCH` 环境变量则默认值为 *PycInvalidationMode.CHECKED_HASH*，否则默认值为 *PycInvalidationMode.TIMESTAMP*。

在 3.2 版本发生变更: 将 *cfile* 的默认值改成与 **PEP 3147** 兼容。之前的默认值是 *file* + `'c'` (如果启用优化则为 `'o'`)。同时也添加了 *optimize* 形参。

在 3.4 版本发生变更: 将代码更改为使用 *importlib* 执行字节码缓存文件写入。这意味着文件创建/写入的语义现在与 *importlib* 所做的相匹配，例如权限、写入和移动语义等等。同时也添加了当 *cfile* 为符号链接或非常规文件时引发 *FileExistsError* 的预警设置。

在 3.7 版本发生变更: *invalidation_mode* 形参是根据 **PEP 552** 的描述添加的。如果设置了 `SOURCE_DATE_EPOCH` 环境变量，*invalidation_mode* 将被强制设为 *PycInvalidationMode.CHECKED_HASH*。

在 3.7.2 版本发生变更: `SOURCE_DATE_EPOCH` 环境变量不会再覆盖 *invalidation_mode* 参数的值，而改为确定其默认值。

在 3.8 版本发生变更: 增加了 *quiet* 形参。

class py_compile.**PycInvalidationMode**

一个由可用方法组成的枚举，解释器可以用它来确定字节码文件是否与源文件保持同步。.pyc 文件在其标头中指明了所需的失效模式。请参阅 pyc-invalidation 了解有关 Python 在运行时如何让 .pyc 文件失效的更多信息。

Added in version 3.7.

TIMESTAMP

.pyc 文件包括时间戳和源文件的大小，Python 将在运行时将其与源文件的元数据进行比较以确定 .pyc 文件是否需要重新生成。

CHECKED_HASH

.pyc 文件包括源文件内容的哈希值，Python 将在运行时将其与源文件内容进行比较以确定 .pyc 文件是否需要重新生成。

UNCHECKED_HASH

类似于*CHECKED_HASH*，.pyc 文件包括源文件内容的哈希值。但是，Python 将在运行时假定 .pyc 文件是最新的而完全不会将 .pyc 与源文件进行验证。

此选项适用于 .pycs 由 Python 以外的某个系统例如构建系统来确保最新的情况。

33.8.1 命令行接口

这个模块可作为脚本被唤起以编译多个源文件。在 *filenames* 中指定的文件会被编译并将结果字节码以普通方式进行缓存。这个程序不会搜索目录结构来定位源文件；它只编译显式指定的文件。如果某个文件无法被编译则退出状态为非零值。

<file> ... <fileN>

-

位置参数是要编译的文件。如果 - 是唯一的形参，则文件列表将从标准输入获取。

-q, --quiet

屏蔽错误输出。

在 3.2 版本发生变更: 添加了对 - 的支持。

在 3.10 版本发生变更: 添加了对*-q* 的支持。

参见

模块*compileall*
编译一个目录树中所有 Python 源文件的工具。

33.9 compileall --- 字节编译 Python 库

源代码: Lib/compileall.py

这个模块提供了一些工具函数来支持安装 Python 库。这些函数可以编译一个目录树中的 Python 源文件。这个模块可被用来在安装库时创建缓存的字节码文件，这使得它们对于没有库目录写入权限的用户来说也是可用的。

Availability: not WASI.

此模块在 WebAssembly 平台上无效或不可用。请参阅*WebAssembly 平台* 了解详情。

33.9.1 使用命令行

此模块可以作为脚本运行 (使用 **python -m compileall**) 来编译 Python 源代码。

directory ...

file ...

位置参数是要编译的文件或包含源文件的目录，目录将被递归地遍历。如果没有给出参数，则其行为如同使用了命令行 `-l <directories from sys.path>`。

-l

不要递归到子目录，只编译直接包含在指明或隐含的目录中的源代码文件。

-f

强制重新构建即使时间戳是最新的。

-q

不要打印已编译文件的列表。如果传入一次，则错误消息仍将被打印。如果传入两次 (`-qq`)，所有输出都会被屏蔽。

-d destdir

要附加到每个被编译文件的路径之前的目录。这将出现在编译时回溯信息中，并且还会被编译到字节码文件中，届时它将在字节码文件被执行而源文件已不存在的情况下被用于回溯和其他消息。

-s strip_prefix

-p prepend_prefix

移除 (`-s`) 或添加 (`-p`) 记录在 `.pyc` 文件中的给定路径前缀。不可与 `-d` 一同使用。

-x regex

regex 会被用于搜索每个要执行编译的文件的完整路径，而如果 regex 产生了一个匹配，则相应文件会被跳过。

-i list

读取文件 `list` 并将其包含的每一行添加到要编译的文件和目录列表中。如果 `list` 为 `-`，则从 `stdin` 读取行。

-b

将字节码写入到它们的传统位置和名称，这可能会覆盖由另一版本的 Python 所创建的字节码文件。默认是将文件写入到它们的 **PEP 3147** 位置和名称，这允许来自多个版本的 Python 字节码文件共存。

-r

控制子目录的最大递归层级。如果给出此选项，则 `-l` 选项将不会被考虑。**python -m compileall <directory> -r 0** 等价于 **python -m compileall <directory> -l**。

-j N

使用 *N* 个工作进程来编译给定目录中的文件。如果使用 `0`，则将使用*os.process_cpu_count()* 的结果。

--invalidation-mode [timestamp|checked-hash|unchecked-hash]

控制生成的字节码文件在运行时的失效规则。值为 `timestamp`，意味着将生成嵌入了源时间戳和大小的 `.pyc` 文件。`checked-hash` 和 `unchecked-hash` 等值将导致生成基于哈希的 pyc。基于哈希的 pyc 嵌入了源文件内容的哈希值而不是时间戳。请参阅 pyc-invalidation 了解有关 Python 在运行时如何验证字节码缓存文件的更多信息。如果未设置 `SOURCE_DATE_EPOCH` 环境变量则默认值为 `timestamp`，而如果设置了 `SOURCE_DATE_EPOCH` 则为 `checked-hash`。

-o level

使用给定的优化级别进行编译。可以多次使用来一次性地针对多个级别进行编译 (例如，`compileall -o 1 -o 2`)。

-e dir

忽略指向给定目录之外的符号链接。

--hardlink-dupes

如果两个不同优化级别的 .pyc 文件具有相同的内容，则使用硬链接来合并重复的文件。

在 3.2 版本发生变更: 增加了 -i, -b 和 -h 选项。

在 3.5 版本发生变更: 增加了 -j, -r 和 -qq 选项。-q 选项改为多级别值。-b 将总是产生以 .pyc 为后缀的字节码文件，而不是 .pyo。

在 3.7 版本发生变更: 增加了 --invalidation-mode 选项。

在 3.9 版本发生变更: 增加了 -s, -p, -e 和 --hardlink-dupes 选项。将默认的递归限制从 10 提高到*sys.getrecursionlimit()*。增加允许多次指定 -o 选项。

没有可以控制*compile()* 函数所使用的优化级别的命令行选项，因为 Python 解释器本身已经提供了该选项: **python -O -m compileall**。

类似地，*compile()* 函数会遵循*sys.pycache_prefix* 设置。所生成的字节码缓存将仅在*compile()* 附带与将在运行时使用的相同*sys.pycache_prefix* 时可用（如果存在该设置）。

33.9.2 公有函数

compileall.**compile_dir**(*dir*, *maxlevels=sys.getrecursionlimit()*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

递归地深入名为 *dir* 的目录树，在途中编译所有 .py 文件。如果所有文件都编译成功则返回真值，否则返回假值。

maxlevels 形参被用来限制递归深度；它默认为 sys.getrecursionlimit()。

如果给出了 *ddir*，它会被附加到每个被编译的文件的路径之前以便在编译时回溯中使用，同时还会被编译到字节码文件中，届时它将在字节码文件被执行而源文件已不存在的情况下被用于回溯和其他消息中。

如果 *force* 为真值，则即使时间戳为最新模块也会被重新编译。

如果给出了 *rx*，则它的 search 方法会在准备编译的每个文件的完整路径上被调用，并且如果它返回真值，则该文件会被跳过。这可被用来排除与一个正则表达式相匹配的文件，正则表达式将以*re.Pattern* 对象的形式给出。

如果 *quiet* 为 False 或 0 (默认值)，则文件名和其他信息将被打印到标准输出。如果设为 1，则只打印错误。如果设为 2，则屏蔽所有输出。

如果 *legacy* 为真值，则将字节码文件写入到它们的传统位置和名称，这可能会覆盖由另一版本的 Python 所创建的字节码文件。默认是将文件写入到它们的 **PEP 3147** 位置和名称，这允许来自多个版本的 Python 字节码文件共存。

optimize 指明编译器的优化级别。它会被传给内置*compile()* 函数。还接受一个优化层别列表这将在单次调用中多次编译一个 .py 文件。

参数 *workers* 指明要使用多少个工作进程来并行编译文件。默认设置不使用多个工作进程。如果平台不能使用多个工作进程而又给出了 *workers* 参数，则将回退为使用顺序编译。如果 *workers* 为 0，则会使用系统的核心数量。如果 *workers* 小于 0，则会引发*ValueError*。

invalidation_mode 应为*py_compile.PycInvalidationMode* 枚举的成员之一并将控制所生成的 pyc 在运行时以何种方式验证是否失效。

stripdir, *prependdir* 和 *limit_sl_dest* 参数分别对应上述的 -s, -p 和 -e 选项。它们可以被指定为 str 或*os.PathLike*。

如果 *hardlink_dupes* 为真值且两个使用不同优化级别的 .pyc 文件具有相同的内容，则会使用硬链接来合并重复的文件。

在 3.2 版本发生变更: 增加了 *legacy* 和 *optimize* 形参。

在 3.5 版本发生变更: 增加了 *workers* 形参。

在 3.5 版本发生变更: *quiet* 形参已改为多级别值。

在 3.5 版本发生变更: *legacy* 形参将只写入 .pyc 文件而非 .pyo 文件，无论 *optimize* 的值是什么。

在 3.6 版本发生变更: 接受一个*path-like object*。

在 3.7 版本发生变更: 增加了 *invalidation_mode* 形参。

在 3.7.2 版本发生变更: *invalidation_mode* 形参的默认值已更新为 None。

在 3.8 版本发生变更: 将 *workers* 设为 0 现在将会选择最优核心数量。

在 3.9 版本发生变更: 增加了 *stripdir*, *prependdir*, *limit_sl_dest* 和 *hardlink_dupes* 参数。*maxlevels* 的默认值从 10 改为 sys.getrecursionlimit()

compileall.**compile_file**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

编译路径为 *fullname* 的文件。如果文件编译成功则返回真值，否则返回假值。

如果给出了 *ddir*，它会被附加到被编译的文件的路径之前以便在编译时回溯中使用，同时还会被编译到字节码文件中，届时它将在字节码文件被执行而源文件已不存在的情况下被用于回溯和其他消息中。

如果给出了 *rx*，则会向它的 search 方法传入准备编译的文件的完整路径，并且如果它返回真值，则不编译该文件并返回 True。这可被用来排除与一个正则表达式相匹配的文件，正则表达式将以*re.Pattern* 对象的形式给出。

如果 *quiet* 为 False 或 0 (默认值)，则文件名和其他信息将被打印到标准输出。如果设为 1，则只打印错误。如果设为 2，则屏蔽所有输出。

如果 *legacy* 为真值，则将字节码文件写入到它们的传统位置和名称，这可能会覆盖由另一版本的 Python 所创建的字节码文件。默认是将文件写入到它们的 **PEP 3147** 位置和名称，这允许来自多个版本的 Python 字节码文件共存。

optimize 指明编译器的优化级别。它会被传给内置*compile()* 函数。还接受一个优化层别列表这将在单次调用中多次编译一个 .py 文件。

invalidation_mode 应为*py_compile.PycInvalidationMode* 枚举的成员之一并将控制所生成的 pyc 在运行时以何种方式验证是否失效。

stripdir, *prependdir* 和 *limit_sl_dest* 参数分别对应上述的 -s, -p 和 -e 选项。它们可以被指定为 str 或*os.PathLike*。

如果 *hardlink_dupes* 为真值且两个使用不同优化级别的 .pyc 文件具有相同的内容，则会使用硬链接来合并重复的文件。

Added in version 3.2.

在 3.5 版本发生变更: *quiet* 形参已改为多级别值。

在 3.5 版本发生变更: *legacy* 形参将只写入 .pyc 文件而非 .pyo 文件，无论 *optimize* 的值是什么。

在 3.7 版本发生变更: 增加了 *invalidation_mode* 形参。

在 3.7.2 版本发生变更: *invalidation_mode* 形参的默认值已更新为 None。

在 3.9 版本发生变更: 增加了 *stripdir*, *prependdir*, *limit_sl_dest* 和 *hardlink_dupes* 参数。

compileall.**compile_path**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

将在 sys.path 中找到的所有 .py 文件编译为字节码。如果所有文件编译成功则返回真值，否则返回假值。

如果 *skip_curdir* 为真值（默认），则当前目录不会被包括在搜索中。所有其他形参将被传递给*compile_dir()* 函数。请注意不同于其他编译函数，maxlevels 默认为 0。

在 3.2 版本发生变更: 增加了 *legacy* 和 *optimize* 形参。

在 3.5 版本发生变更: *quiet* 形参已改为多级别值。

在 3.5 版本发生变更: *legacy* 形参将只写入 .pyc 文件而非 .pyo 文件，无论 *optimize* 的值是什么。

在 3.7 版本发生变更: 增加了 *invalidation_mode* 形参。

在 3.7.2 版本发生变更: *invalidation_mode* 形参的默认值已更新为 `None`。

强制重新编译 `Lib/` 子目录及其所有子目录下的全部 `.py` 文件:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# 执行同样的编译，排除 .svn 目录中的文件。
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# 也可以使用 pathlib.Path 对象。
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

参见

模块*py_compile*
将单个源文件编译为字节码。

33.10 `dis` --- Python 字节码反汇编器

源代码: Lib/dis.py

---

dis 模块通过反汇编支持 CPython 的*bytecode* 分析。该模块作为输入的 CPython 字节码在文件 `Include/opcode.h` 中定义，并由编译器和解释器使用。

字节码是 CPython 解释器的实现细节。不保证不会在 Python 版本之间添加、删除或更改字节码。不应考虑将此模块的跨 Python VM 或 Python 版本的使用。

在 3.6 版本发生变更: 每条指令使用 2 个字节。以前字节数因指令而异。

在 3.10 版本发生变更: 跳转、异常处理和循环指令的参数现在将为指令偏移量而不是字节偏移量。

在 3.11 版本发生变更: 有些指令带有一个或多个内联缓存条目，它们是采用*CACHE* 指令的形式。这些指令默认是隐藏的，但可以通过将 `show_caches=True` 传给任何*dis* 工具对象来显示。此外，解释器现在会适配字节码以使其能针对不同的运行时条件实现专门化。适配的字节码可通过传入 `adaptive=True` 来显示。

在 3.12 版本发生变更: 跳转的参数是目标指令相对于紧接在跳转指令的*CACHE* 条目之后的指令的偏移量。

因此，*CACHE* 指令的存在对前向跳转是透明的但在处理后向跳转时则需要将其纳入考虑。

在 3.13 版本发生变更: 对于跳转目标和异常处理器输出将显示逻辑标签而不是指令偏移量。增加了 `-O` 命令行选项和 `show_offsets` 参数。

示例：给定函数 `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

可以使用以下命令显示 `myfunc()` 的反汇编:

```
>>> dis.dis(myfunc)
  2           RESUME                   0

  3           LOAD_GLOBAL              1 (len + NULL)
              LOAD_FAST                0 (alist)
```

(续下页)

(接上页)

```
              CALL                     1
              RETURN_VALUE
```

("2" 是行号)。

33.10.1 命令行接口

dis 模块可以在命令行下作为一个脚本来唤起:

```
python -m dis [-h] [-C] [-O] [infile]
```

可以接受以下选项:

-h, --help

显示用法并退出。

-C, --show-caches

显示内联缓存。

Added in version 3.13.

-O, --show-offsets

显示指令偏移量。

Added in version 3.13.

如果指定了 `infile`，其反汇编代码将被写入到标准输出。在其他情况下，反汇编将在从标准输入接收的已编译源代码上进行。

33.10.2 字节码分析

Added in version 3.4.

字节码分析 API 允许将 Python 代码片段包装在*Bytecode* 对象中，以便轻松访问已编译代码的详细信息。

class dis.**Bytecode**(*x*, ***, *first_line=None*, *current_offset=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*)

分析的字节码对应于函数、生成器、异步生成器、协程、方法、源代码字符串或代码对象(由*compile()* 返回)。

这是下面列出的许多函数的便利包装，最值得注意的是*get_instructions()*，迭代于*Bytecode* 的实例产生字节码操作*Instruction* 的实例。

如果 *first_line* 不是 `None`，则表示应该为反汇编代码中的第一个源代码行报告的行号。否则，源行信息（如果有的话）直接来自反汇编的代码对象。

如果 *current_offset* 不是 `None`，它指的就是汇编代码中的指令偏移量。设置它意味着*dis()* 将针对指定的操作码显示“当前指令”标记。

如果 *show_caches* 为 `True`，*dis()* 将显示解释器用来专门化字节码的内联缓存条目。

如果 *adaptive* 为 `True`，*dis()* 将显示可能不同于原始字节码的专门化字节码。

若 *show_offsets* 是 `True`，*dis()* 的输出将会显示指令偏移量。

classmethod from_traceback(*tb*, ***, *show_caches=False*)

从给定回溯构造一个*Bytecode* 实例，将设置 *current_offset* 为异常负责的指令。

codeobj

已编译的代码对象。

first_line

代码对象的第一个源代码行（如果可用）

dis()

返回字节码操作的格式化视图（与*dis.dis()* 打印相同，但作为多行字符串返回）。

info()

返回带有关于代码对象的详细信息的格式化多行字符串，如*code_info()* 。

在 3.7 版本发生变更: 现在可以处理协程和异步生成器对象。

在 3.11 版本发生变更: 增加了 *show_caches* 和 *adaptive* 形参。

示例:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
RESUME
LOAD_GLOBAL
LOAD_FAST
CALL
RETURN_VALUE
```

33.10.3 分析函数

dis 模块还定义了以下分析函数，它们将输入直接转换为所需的输出。如果只执行单个操作，它们可能很有用，因此中间分析对象没用:

dis.**code_info**(*x*)

返回格式化的多行字符串，其包含详细代码对象信息的用于被提供的函数、生成器、异步生成器、协程、方法、源代码字符串或代码对象。

请注意，代码信息字符串的确切内容是高度依赖于实现的，它们可能会在 Python VM 或 Python 版本中任意更改。

Added in version 3.2.

在 3.7 版本发生变更: 现在可以处理协程和异步生成器对象。

dis.**show_code**(*x*, *, *file=None*)

将提供的函数、方法。源代码字符串或代码对象的详细代码对象信息打印到 *file* （如果未指定 *file* ，则为 `sys.stdout` ）。

这是 `print(code_info(x), file=file)` 的便捷简写，用于在解释器提示符下进行交互式探索。

Added in version 3.2.

在 3.4 版本发生变更: 添加 *file* 形参。

dis.**dis**(*x=None*, *, *file=None*, *depth=None*, *show_caches=False*, *adaptive=False*)

反汇编 *x* 对象。*x* 可以表示模块、类、方法、函数、生成器、异步生成器、协程、代码对象、源代码字符串或原始字节码的字节序列。对于模块，它会反汇编所有函数。对于一个类，它会反汇编所有方法（包括类方法和静态方法）。对于代码对象或原始字节码序列，它会为每条字节码指令打印一行。它还会递归地反汇编嵌套代码对象。这些对象包括生成器表达式、嵌套函数、嵌套类的语句体以及用于 标注作用域的代码对象。在反汇编之前，首先使用*compile()* 内置函数将字符串编译为代码对象。如果未提供任何对象，则该函数将反汇编最后一次回溯。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 `sys.stdout` 。

递归的最大深度受 *depth* 限制，除非它是 `None` 。`depth=0` 表示没有递归。

如果 *show_caches* 为 `True`，此函数将显示解释器用来专门化字节码的内联缓存条目。

如果 *adaptive* 为 `True`，此函数将显示可能不同于原始字节码的专门化字节码。

在 3.4 版本发生变更: 添加 *file* 形参。

在 3.7 版本发生变更: 实现了递归反汇编并添加了 *depth* 参数。

在 3.7 版本发生变更: 现在可以处理协程和异步生成器对象。

在 3.11 版本发生变更: 增加了 *show_caches* 和 *adaptive* 形参。

distb(tb=None, *, file=None, show_caches=False, adaptive=False, show_offset=False)

如果没有传递，则使用最后一个回溯来反汇编回溯的堆栈顶部函数。指示了导致异常的指令。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 `sys.stdout` 。

在 3.4 版本发生变更: 添加 *file* 形参。

在 3.11 版本发生变更: 增加了 *show_caches* 和 *adaptive* 形参。

在 3.13 版本发生变更: 添加了 *show_offsets* 参数。

dis.**disassemble**(*code*, *lasti=-1*, *, *file=None*, *show_caches=False*, *adaptive=False*)

disco(code, lasti=-1, *, file=None, show_caches=False, adaptive=False, show_offsets=False)

反汇编代码对象，如果提供了 *lasti* ，则指示最后一条指令。输出分为以下几列：

1. 行号，用于每行的第一条指令
2. 当前指令，表示为 `-->` ，
3. 一个标记的指令，用 `>>` 表示，
4. 指令的地址，
5. 操作码名称，
6. 操作参数，和
7. 括号中参数的解释。

参数解释识别本地和全局变量名称、常量值、分支目标和比较运算符。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 `sys.stdout` 。

在 3.4 版本发生变更: 添加 *file* 形参。

在 3.11 版本发生变更: 增加了 *show_caches* 和 *adaptive* 形参。

在 3.13 版本发生变更: 添加了 *show_offsets* 参数。

dis.**get_instructions**(*x*, *, *first_line=None*, *show_caches=False*, *adaptive=False*)

在所提供的函数、方法、源代码字符串或代码对象中的指令上返回一个迭代器。

迭代器生成一系列 *Instruction* ，命名为元组，提供所提供代码中每个操作的详细信息。

如果 *first_line* 不是 `None` ，则表示应该为反汇编代码中的第一个源代码行报告的行号。否则，源行信息（如果有的话）直接来自反汇编的代码对象。

参数 *adaptive* 和其在 *dis()* 中的工作方式一样。

Added in version 3.4.

在 3.11 版本发生变更: 增加了 *show_caches* 和 *adaptive* 形参。

在 3.13 版本发生变更: *show_caches* 参数被弃用并且失效。迭代器保证 *Instruction* 实例一定有 *cache_info* 字段（无论 *show_caches* 传入什么），不再生成单独的缓存项。

dis.**findlinestarts**(*code*)

这个生成器函数使用 代码对象 *code* 的 `co_lines()` 方法来查找源代码中行开头的偏移量。它们将作为 `(offset, lineno)` 对被生成。

在 3.6 版本发生变更: 行号可能会减少。以前，他们总是在增加。

在 3.10 版本发生变更: 使用 **PEP 626** `co_lines()` 方法而不是 代码对象的 `co_firstlineno` 和 `co_lnotab` 属性。

在 3.13 版本发生变更: 若字节码不对应任何一行，行号可以是 “None“。

dis.**findlabels**(*code*)

检测作为跳转目标的原始编译后字节码字符串 *code* 中的所有偏移量，并返回这些偏移量的列表。

dis.**stack_effect**(*opcode*, *oparg=None*, *, *jump=None*)

使用参数 *oparg* 计算 *opcode* 的堆栈效果。

如果代码有一个跳转目标并且 *jump* 是 `True`，则 `drag_effect()` 将返回跳转的堆栈效果。如果 *jump* 是 `False`，它将返回不跳跃的堆栈效果。如果 *jump* 是 `None`（默认值），它将返回两种情况的最大堆栈效果。

Added in version 3.4.

在 3.8 版本发生变更: 添加 *jump* 参数。

在 3.13 版本发生变更: 如果 “oparg“ 被省略（或传入 “None“ ），过去如果字节码使用参数，此时会抛出错误，而现在会返回 “oparg=0“时的结果。当 “opcode“ 不使用整数 `oparg` 时，传入的 “oparg“将被忽略，不会抛出错误。

33.10.4 Python 字节码说明

get_instructions() 函数和*Bytecode* 类提供字节码指令的详细信息的*Instruction* 实例：

class dis.**Instruction**

字节码操作的详细信息

opcode

操作的数字代码，对应于下面列出的操作码值和操作码集合 中的字节码值。

opname

人类可读的操作名称

baseopcode

如果操作是专用的则为基本操作的数字码；否则等于*opcode*

baseopname

如果操作是专用的则为基本操作的人类易读的名称；否则等于*opname*

arg

操作的数字参数（如果有的话），否则为 `None`

oparg

arg 的别名

argval

已解析的 arg 值（如果有的话），否则为 `None`

argrepr

人类可读的操作参数（如果存在）的描述，否则为空字符串。

offset

在字节码序列中的起始操作索引

start_offset

在字节码序列中的起始索引，包括前面的 `EXTENDED_ARG` 操作（如有），否则和*offset* 相等。

cache_offset

在操作后面的缓存条目的起始索引

end_offset

操作后面的缓存条目的结束索引

starts_line

如果这个操作在源代码行的开始，为 “True“，否则为 “False“。

line_number

操作码对应的源代码行号（如有），否则为 “None“

is_jump_target

如果其他代码跳到这里，则为 `True` ，否则为 `False`

jump_target

跳转的目标的字节码索引，如果有跳转操作，否则为 “None“

positions

dis.Positions 对象保存了这条指令所涵盖的起始和结束位置。

cache_info

有关该指令的缓存条目的信息，为 `(name, size, data)` 形式的三元组，其中 `name` 和 `size` 描述缓存的格式而 data 为缓存的内容。如果指令没有缓存则 `cache_info` 为 `None`。

Added in version 3.4.

在 3.11 版本发生变更: 增加了 `positions` 字段。

在 3.13 版本发生变更: 更改了字段 “starts_line“。

添加了字段 “start_offset“、`cache_offset`、`end_offset`、`baseopname`、`baseopcode`、`jump_target`、`oparg`、`line_number``和``cache_info`

class dis.**Positions**

考虑到此信息不可用的情况，某些字段可能为 `None`。

lineno

end_lineno

col_offset

end_col_offset

Added in version 3.11.

Python 编译器当前生成以下字节码指令。

一般指令

在下文中，我们将把解释器栈称为 `STACK` 并像描述 Python 列表一样描述对它的操作。栈顶对应于该语言中的 `STACK[-1]`。

NOP

无操作代码。被字节码优化器用作占位符，以及生成行追踪事件。

POP_TOP

移除除堆栈顶部的项:

```
STACK.pop()
```

END_FOR

移除栈顶。等价于 `POP_TOP`。用于循环结束时的清理，因此而得名。

Added in version 3.12.

END_SEND

实现 `del STACK[-2]`。用于在生成器退出时进行清理。

Added in version 3.12.

COPY (*i*)

将第 *i* 项推入栈顶，并不移除原项：

```
assert i > 0
STACK.append(STACK[-i])
```

Added in version 3.11.

SWAP (*i*)

将栈顶的项与栈中第 *i* 项互换：

```
STACK[-i], STACK[-1] = STACK[-1], STACK[-i]
```

Added in version 3.11.

CACHE

此操作码不是真正的指令，它被用来为解释器标记额外空间以便在字节码中直接缓存有用的数据。它会被所有 `dis` 工具自动隐藏，但可以通过 `show_caches=True` 来查看。

从逻辑上说，此空间是之前的指令的组成部分。许多操作码都预期带有固定数量的缓存，并会指示解释器在运行时跳过它们。

被填充的缓存看起来可以像是任意的指令，因此在读取或修改包含快取数据的原始自适应字节码时应当非常小心。

Added in version 3.11.

一元操作

一元操作获取堆栈顶部元素，应用操作，并将结果推回堆栈。

UNARY_NEGATIVE

实现 `STACK[-1] = -STACK[-1]` 。

UNARY_NOT

实现 `STACK[-1] = not STACK[-1]` 。

在 3.13 版本发生变更: 现在，操作对象需要是*bool* 类型。

UNARY_INVERT

实现 `STACK[-1] = ~STACK[-1]` 。

GET_ITER

实现 `STACK[-1] = iter(STACK[-1])` 。

GET_YIELD_FROM_ITER

如果 `STACK[-1]` 是一个*generator iterator* 或*coroutine* 对象则它将保持原样。否则，将实现 `STACK[-1] = iter(STACK[-1])`。

Added in version 3.5.

TO_BOOL

实现 “STACK[-1] = bool(STACK[-1])“。

Added in version 3.13.

双目和原地操作

双目操作移除栈顶的两项（`STACK[-1]` 和 `STACK[-2]`），执行其运算操作，并将结果放回栈中。

原地操作类似于双目操作，但当 `STACK[-2]` 支持时，操作将在原地进行。结果 `STACK[-1]` 可能（但不一定）是原先 `STACK[-2]` 的值。

BINARY_OP (*op*)

实现双目和原地操作运算符（取决于 *op* 的值）：

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

Added in version 3.11.

BINARY_SUBSCR

实现：

```
key = STACK.pop()
container = STACK.pop()
STACK.append(container[key])
```

STORE_SUBSCR

实现：

```
key = STACK.pop()
container = STACK.pop()
value = STACK.pop()
container[key] = value
```

DELETE_SUBSCR

实现：

```
key = STACK.pop()
container = STACK.pop()
del container[key]
```

BINARY_SLICE

实现：

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
STACK.append(container[start:end])
```

Added in version 3.12.

STORE_SLICE

实现：

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
values = STACK.pop()
container[start:end] = value
```

Added in version 3.12.

协程操作码

GET_AWAITABLE(*where*)

实现 `STACK[-1] = get_awaitable(STACK[-1])` 。其中对于 `get_awaitable(o)` ，当 `o` 是一个有*CO_ITERABLE_COROUTINE* 旗标的协程对象或生成器对象时，返回 `o`，否则解析 `o.__await__` 。

如果 `where` 操作数为非零值，则表示指令所在的位置:

- `1`：在调用 `__aenter__` 之后
- `2`：在调用 `__aexit__` 之后

Added in version 3.5.

在 3.11 版本发生变更: 在之前版本中，该指令没有 oparg。

GET_AITER

实现 `STACK[-1] = STACK[-1].__aiter__()`。

Added in version 3.5.

在 3.7 版本发生变更: 已经不再支持从 `__aiter__` 返回可等待对象。

GET_ANEXT

对堆栈实现 `STACK.append(get_awaitable(STACK[-1].__anext__()))` 。关于 `get_awaitable` 的细节，见 `GET_AWAITABLE` 。

Added in version 3.5.

END_ASYNC_FOR

终结一个 `async for` 循环。在等待下一项时处理被引发的异常。栈包含了 `STACK[-2]` 中的异步可迭代对象和 `STACK[-1]` 中的已引发异常。两者都将被弹出。如果异常不是*StopAsyncIteration*，它会被重新引发。

Added in version 3.8.

在 3.11 版本发生变更: 栈中的异常表示形式现在将由一个而不是三个条目组成。

CLEANUP_THROW

处理当前帧中由调用 `throw()` 或 `close()` 引发的异常。如果 `STACK[-1]` 是*StopIteration* 的实例，则从栈中弹出三个值，并将其成员 `value` 的值推入栈中，否则重新引发 `STACK[-1]` 异常。

Added in version 3.12.

BEFORE_ASYNC_WITH

从 `STACK[-1]` 解析 `__aenter__` 和 `__aexit__` 。将 `__aexit__` 和 `__aenter__()` 的结果推入栈中:

```
STACK.extend((__aexit__, __aenter__())
```

Added in version 3.5.

其他操作码

SET_ADD (*i*)

实现:

```
item = STACK.pop()
set.add(STACK[-i], item)
```

用于实现集合推导式。

LIST_APPEND (*i*)

实现:

```
item = STACK.pop()
list.append(STACK[-i], item)
```

用于实现列表推导式。

MAP_ADD (*i*)

实现:

```
value = STACK.pop()
key = STACK.pop()
dict.__setitem__(STACK[-i], key, value)
```

用于实现字典推导式。

Added in version 3.1.

在 3.8 版本发生变更: 映射的值为 `STACK[-1]` ，映射的键为 `STACK[-2]` 。之前它们是反过来的。

对于所有*SET_ADD* 、*LIST_APPEND* 和*MAP_ADD* 指令，当弹出添加的值或键值对时，容器对象保留在堆栈上，以便它可用于循环的进一步迭代。

RETURN_VALUE

返回 `STACK[-1]` 给函数的调用者。

RETURN_CONST (*consti*)

返回 `co_consts[consti]` 给函数的调用者。

Added in version 3.12.

YIELD_VALUE

从*generator* 产生 `STACK.pop()`。

在 3.11 版本发生变更: oparg 被设为堆栈深度。

在 3.12 版本发生变更: oparg 被设为异常块的深度，以确保关闭生成器的效率。

在 3.13 版本发生变更: 如果该指令是 yield-from 或 await 的一部分则 oparg 为 `1`，否则为 `0`。

SETUP_ANNOTATIONS

检查 `__annotations__` 是否在 `locals()` 中定义，如果没有，它被设置为空 `dict` 。只有在类或模块体静态地包含*variable annotations* 时才会发出此操作码。

Added in version 3.6.

POP_EXCEPT

从栈中弹出一个值，它将被用来恢复异常状态。

在 3.11 版本发生变更: 栈中的异常表示形式现在将由一个而不是三个条目组成。

RERAISE

重新引发当前位于栈顶的异常。如果 oparg 为非零值，则从栈顶额外弹出一个值用来设置当前帧的 `f_lasti`。

Added in version 3.9.

在 3.11 版本发生变更: 栈中的异常表示形式现在将由一个而不是三个条目组成。

PUSH_EXC_INFO

从栈中弹出一个值。将当前异常推入栈顶。将原先被弹出的值推回栈。在异常处理器中使用。

Added in version 3.11.

CHECK_EXC_MATCH

为 `except` 执行异常匹配。检测 `STACK[-2]` 是否为匹配 `STACK[-1]` 的异常。弹出 `STACK[-1]` 并将测试的布尔值结果推入栈。

Added in version 3.11.

CHECK_EG_MATCH

为 `except*` 执行异常匹配。在代表 `STACK[-2]` 的异常组上应用 `split(STACK[-1])`。

在匹配的情况下，从栈中弹出两项并推入不匹配的子分组 (如完全匹配则为 `None`) 以及匹配的子分组。当没有任何匹配时，则弹出一项 (匹配类型) 并推入 `None`。

Added in version 3.11.

WITH_EXCEPT_START

调用栈中 4 号位置上的函数并附带代表位于栈顶的异常的参数 (type, val, tb)。用于在 `with` 语句内发生异常时实现调用 `context_manager.__exit__(*exc_info())`。

Added in version 3.9.

在 3.11 版本发生变更: `__exit__` 函数位于栈的 4 号位而不是 7 号位。栈中的异常表示形式现在由一项而不是三项组成。

LOAD_ASSERTION_ERROR

将*AssertionError* 推入栈顶。由 assert 语句使用。

Added in version 3.9.

LOAD_BUILD_CLASS

将 builtins.__build_class__() 推入栈中。之后它将会被调用来构造一个类。

BEFORE_WITH

此操作码会在 with 代码块开始之前执行多个操作。首先，它将从上下文管理器加载 __exit__() 并将其推入栈顶以供*WITH_EXCEPT_START* 后续使用。然后，将调用 __enter__()。最后，将调用 __enter__() 方法的结果推入栈顶。

Added in version 3.11.

GET_LEN

执行 STACK.append(len(STACK[-1]))。当需要与模式结构体进行比较时在 match 语句中使用。

Added in version 3.10.

MATCH_MAPPING

如果 STACK[-1] 是*collections.abc.Mapping* 的实例（或者更准确地说：如果在它的 tp_flags 中设置了 Py_TPFLAGS_MAPPING 旗标），则将 True 推入栈顶。否则，推入 False。

Added in version 3.10.

MATCH_SEQUENCE

如果 STACK[-1] 是*collections.abc.Sequence* 的实例而 不是*str*/*bytes*/*bytearray* 的实例（或者更准确地说：如果在它的 tp_flags 中设置了 Py_TPFLAGS_SEQUENCE 旗标），则将 True 推入栈顶。否则，推入 False。

Added in version 3.10.

MATCH_KEYS

STACK[-1] 是一个映射键的元组，而 STACK[-2] 是匹配目标。如果 STACK[-2] 包含 STACK[-1] 中的所有键，则推入一个包含对应值的*tuple*。在其他情况下，推入 None。

Added in version 3.10.

在 3.11 版本发生变更: 在之前的版本中，该指令还会推入一个表示成功 (True) 或失败 (False) 的布尔值。

STORE_NAME(*namei*)

实现 name = STACK.pop()。*namei* 是 *name* 在 代码对象的 co_names 属性中的索引。在可能的情况下编译器会尝试使用*STORE_FAST* 或*STORE_GLOBAL*。

DELETE_NAME(*namei*)

实现 del name，其中 *namei* 是 代码对象的 co_names 属性的索引。

UNPACK_SEQUENCE(*count*)

将 STACK[-1] 解包为 *count* 个单独的值，然后自右向左放入栈中。要求有确切的 *count* 值：

```
assert(len(STACK[-1]) == count)
STACK.extend(STACK.pop()[:-count-1:-1])
```

UNPACK_EX(*counts*)

实现带星号目标的赋值：将 STACK[-1] 中的可迭代对象解包为各个单独的值。值的总数可以小于可迭代对象的项数：其中会有一个值是存放剩下的项的列表。

在列表前后的值的数量被限制在 255。

列表前的值的数量被编码在操作码的参数之中。如果列表后有值，则其数量会被用 EXTENDED_ARG 编码。因此参数可以被认为是一个双字节值，其中低位字节代表列表前的值的数量，高位字节代表其后的值的数量。

提取出来的值被自右向左放入栈中，也就是说 `a, *b, c = d` 在执行完成之后会被这样储存: `STACK.extend((a, b, c))` 。

STORE_ATTR(*namei*)

实现:

```
obj = STACK.pop()
value = STACK.pop()
obj.name = value
```

其中 *namei* 是 name 在 代码对象的 `co_names` 中的索引。

DELETE_ATTR(*namei*)

实现:

```
obj = STACK.pop()
del obj.name
```

其中 *namei* 是 name 在 代码对象的 `co_names` 中的索引。

STORE_GLOBAL(*namei*)

类似于*STORE_NAME* 但会将 name 存储为全局变量。

DELETE_GLOBAL(*namei*)

类似于*DELETE_NAME* 但会删除一个全局变量。

LOAD_CONST(*consti*)

将 `co_consts[consti]` 推入栈顶。

LOAD_NAME(*namei*)

将 `co_names[namei]` 关联的值压入栈中。名称的查找范围包括局部变量，然后是全局变量，然后是内置量。

LOAD_LOCALS

将一个局部变量字典的引用压入栈中。被用于为*LOAD_FROM_DICT_OR_DEREF* 和*LOAD_FROM_DICT_OR_GLOBALS* 准备命名空间字典。

Added in version 3.12.

LOAD_FROM_DICT_OR_GLOBALS(*i*)

从栈中弹出一个映射，在其中查找 `co_names[namei]`。如果在此没有找到相应的名称，则在全局变量和内置量中查找，类似*LOAD_GLOBAL* 。被用于在类定义中的 标注作用域中加载全局变量。

Added in version 3.12.

BUILD_TUPLE(*count*)

创建一个使用来自栈的 *count* 个条目的元组，并将结果元组推入栈顶:

```
if count == 0:
    value = ()
else:
    value = tuple(STACK[-count:])
    STACK = STACK[:-count]

STACK.append(value)
```

BUILD_LIST(*count*)

类似于*BUILD_TUPLE* 但会创建一个列表。

BUILD_SET(*count*)

类似于*BUILD_TUPLE* 但会创建一个集合。

BUILD_MAP(*count*)

将一个新字典对象推入栈顶。弹出 `2 * count` 项使得字典包含 *count* 个条目: `{..., STACK[-4]: STACK[-3], STACK[-2]: STACK[-1]}` 。

在 3.5 版本发生变更: 字典是根据栈中的项创建而不是创建一个预设大小包含 *count* 项的空字典。

BUILD_CONST_KEY_MAP(*count*)

BUILD_MAP 版本专用于常量键。弹出的栈顶元素包含一个由键构成的元组，然后从 `STACK[-2]` 开始从构建字典的值中弹出 *count* 个值。

Added in version 3.6.

BUILD_STRING(*count*)

拼接 *count* 个来自栈的字符串并将结果字符串推入栈顶。

Added in version 3.6.

LIST_EXTEND(*i*)

实现:

```
seq = STACK.pop()
list.extend(STACK[-i], seq)
```

用于构建列表。

Added in version 3.9.

SET_UPDATE(*i*)

实现:

```
seq = STACK.pop()
set.update(STACK[-i], seq)
```

用于构建集合。

Added in version 3.9.

DICT_UPDATE(*i*)

实现:

```
map = STACK.pop()
dict.update(STACK[-i], map)
```

用于构建字典。

Added in version 3.9.

DICT_MERGE(*i*)

类似于*DICT_UPDATE* 但对于重复的键会引发异常。

Added in version 3.9.

LOAD_ATTR(*namei*)

如果 `namei` 的低位未设置,则将 `STACK[-1]` 替换为 `getattr(STACK[-1], co_names[namei>>1])` 。

如果 `namei` 的低位已设置，则会尝试从 `STACK[-1]` 对象加载名为 `co_names[namei>>1]` 的方法。`STACK[-1]` 会被弹出。此字节码会区分两种情况: 如果 `STACK[-1]` 具有一个名称正确的方法，字节码会推入未绑定的方法和 `STACK[-1]` 。`STACK[-1]` 将被*CALL* 或者*CALL_KW* 用作调用未绑定方法时的第一个参数 (`self`)。否则，将推入 `NULL` 和属性查询所返回的对象。

在 3.12 版本发生变更: 如果 `namei` 的低位已置，则会在属性或未绑定方法之前分别将 `NULL` 或 `self` 推入栈。

LOAD_SUPER_ATTR(*namei*)

该操作码实现了*super()*，包括零参数和双参数形式 (例如 `super().method()`, `super().attr` 和 `super(cls, self).method()`, `super(cls, self).attr`)。

它从栈弹出三个值（栈顶在前）：

- `self`：当前方法的第一个参数
- `cls`：当前方法被定义的类
- 全局的 “super“

对应于其参数，它的操作类似于*LOAD_ATTR*，区别在于 `namei` 左移了 2 位而不是 1 位。

`namei` 的低位发出尝试加载方法的信号，与*LOAD_ATTR*一样，其结果是推入 `NULL` 和所加载的方法。当其被取消设置时会将单个值推入栈。

`namei` 的次低比特位如果被设置，表示这是对*super()*附带两个参数的调用（未设置则表示附带零个参数）。

Added in version 3.12.

COMPARE_OP(*opname*)

执行布尔操作。操作名可以在 “cmp_op[opname » 5]“中找到。如果 “opname“的第五低位是 1（`opname & 16`），则结果将被强制转换为 “bool“。

在 3.13 版本发生变更: 现在参数的第五低位表示强制将结果转换为*bool*。

IS_OP(*invert*)

执行 `is` 比较，或者如果 `invert` 为 1 则执行 `is not`。

Added in version 3.9.

CONTAINS_OP(*invert*)

执行 `in` 比较，或者如果 `invert` 为 1 则执行 `not in`。

Added in version 3.9.

IMPORT_NAME(*namei*)

导入模块 `co_names[namei]`。会弹出 `STACK[-1]` 和 `STACK[-2]` 以提供 *fromlist* 和 *level* 参数给*__import__()*。模块对象会被推入栈顶。当前命名空间不受影响：对于一条标准 import 语句，会执行后续的*STORE_FAST* 指令来修改命名空间。

IMPORT_FROM(*namei*)

从在 `STACK[-1]` 内找到的模块中加载属性 `co_names[namei]`。结果对象会被推入栈顶，以便由后续的*STORE_FAST* 指令来保存。

JUMP_FORWARD(*delta*)

将字节码计数器的值增加 *delta*。

JUMP_BACKWARD(*delta*)

将字节码计数器减少 *delta*。检查中断。

Added in version 3.11.

JUMP_BACKWARD_NO_INTERRUPT(*delta*)

将字节码计数器减少 *delta*。不检查中断。

Added in version 3.11.

POP_JUMP_IF_TRUE(*delta*)

如果 `STACK[-1]` 为真值，则将字节码计数器增加 *delta*。`STACK[-1]` 将被弹出。

在 3.11 版本发生变更: 操作符的参数现在是一个相对的差值而不是一个绝对的目标量。此操作码是一个伪指令，在最终的字节码里被定向的版本（forward/backward ）取代。

在 3.12 版本发生变更: 该操作码现在不再是伪指令。

在 3.13 版本发生变更: 现在，操作对象需要是*bool* 类型。

POP_JUMP_IF_FALSE(*delta*)

如果 STACK[-1] 为假值，则将字节码计数器增加 *delta*。STACK[-1] 将被弹出。

在 3.11 版本发生变更: 操作符的参数现在是一个相对的差值而不是一个绝对的目标量。此操作码是一个伪指令，在最终的字节码里被定向的版本（forward/backward ）取代。

在 3.12 版本发生变更: 该操作码现在不再是伪指令。

在 3.13 版本发生变更: 现在，操作对象需要是*bool* 类型。

POP_JUMP_IF_NOT_NONE(*delta*)

如果 STACK[-1] 不为 None ，则将字节码计数器增加 *delta*。STACK[-1] 将被弹出。

Added in version 3.11.

在 3.12 版本发生变更: 该操作码现在不再是伪指令。

POP_JUMP_IF_NONE(*delta*)

如果 STACK[-1] 为 None ，则将字节码计数器增加 *delta*。STACK[-1] 将被弹出。

Added in version 3.11.

在 3.12 版本发生变更: 该操作码现在不再是伪指令。

FOR_ITER(*delta*)

STACK[-1] 是一个*iterator*。调用其*__next__()* 方法。如果产生一个新的值则压入栈中（把迭代器压下去）。如果迭代器已耗尽，则将字节码计数器增加 *delta* 。

在 3.12 版本发生变更: 直到 3.11 ，当迭代器耗尽时，它会被从栈中弹出。

LOAD_GLOBAL(*namei*)

将名为 co_names[namei>>1] 的全局对象加载到栈顶。

在 3.11 版本发生变更: 如果设置了 namei 的低比特位，则会在全局变量前将一个 NULL 推入栈。

LOAD_FAST(*var_num*)

将指向局部对象 co_varnames[var_num] 的引用推入栈顶。

在 3.12 版本发生变更: 这个操作码目前只用在保证局部变量被初始化的情况下使用。它不能引发*UnboundLocalError* 。

LOAD_FAST_LOAD_FAST(*var_nums*)

将指向 co_varnames[var_nums >> 4] 和 co_varnames[var_nums & 15] 的引用推入栈顶。

Added in version 3.13.

LOAD_FAST_CHECK(*var_num*)

将一个指向局部变量 co_varnames[var_num] 的引用推入栈中，如果该局部变量未被初始化，引发一个*UnboundLocalError* 。

Added in version 3.12.

LOAD_FAST_AND_CLEAR(*var_num*)

将一个指向局部变量 co_varnames[var_num] 的引用推入栈中（如果该局部变量未被初始化，则推入一个 NULL ）然后将 co_varnames[var_num] 设为 NULL 。

Added in version 3.12.

STORE_FAST(*var_num*)

将 STACK.pop() 存放到局部变量 co_varnames[var_num] 。

STORE_FAST_STORE_FAST(*var_nums*)

将 STACK[-1] 存入 co_varnames[var_nums >> 4] 并将 STACK[-2] 存入 co_varnames[var_nums & 15]。

Added in version 3.13.

STORE_FAST_LOAD_FAST(*var_nums*)

将 STACK.pop() 存入本地 co_varnames[var_nums >> 4] 并将指向本地 co_varnames[var_nums & 15] 的引用推入栈顶。

Added in version 3.13.

DELETE_FAST(*var_num*)

移除局部对象 co_varnames[var_num]。

MAKE_CELL(*i*)

在槽位 i 中创建一个新单元。如果该槽位为非空则该值将存储到新单元中。

Added in version 3.11.

LOAD_DEREF(*i*)

加载包含在"fast locals" 存储的 i 号槽位中的单元。将一个指向该单元所包含对象的引用推入栈。

在 3.11 版本发生变更: i 不再是 co_varnames 的长度的偏移量。

LOAD_FROM_DICT_OR_DEREF(*i*)

从栈中弹出一个映射并查找与该映射中"fast locals" 存储的槽位 i 相关联的名称。如果未在其中找到此名称，则从包含在槽位 i 中的单元加载它，与 *LOAD_DEREF* 类似。这被用于加载类语句体中的闭包变量 (在此之前是使用 LOAD_CLASSDEREF) 和类语句体中的 标注作用域。

Added in version 3.12.

STORE_DEREF(*i*)

将 STACK.pop() 存放到"fast locals" 存储中包含在 i 号槽位的单元内。

在 3.11 版本发生变更: i 不再是 co_varnames 的长度的偏移量。

DELETE_DEREF(*i*)

清空"fast locals" 存储中包含在 i 号槽位的单元。被用于 del 语句。

Added in version 3.2.

在 3.11 版本发生变更: i 不再是 co_varnames 的长度的偏移量。

COPY_FREE_VARS(*n*)

将 n 个自由（闭包）变量 从闭包拷贝到帧中。当调用闭包时不再需要调用方添加特殊的代码。

Added in version 3.11.

RAISE_VARARGS(*argc*)

使用 raise 语句的 3 种形式之一引发异常，具体形式取决于 *argc* 的值：

- 0: raise (重新引发之前的异常)
- 1: raise STACK[-1] （在 STACK[-1] 上引发异常实例或类型）
- 2: raise STACK[-2] from STACK[-1] （在 STACK[-2] 上引发异常实例或类型并将 __cause__ 设为 STACK[-1] ）

CALL(*argc*)

调用 callable 对象，以 "argc"指定的参数个数。堆栈上是（升序）：

- 可调用对象
- self``或者``NULL
- 其余的位置参数

argc 是位置参数的数量，不包括 self。

CALL 将把所有参数和可调用对象弹出栈，附带这些参数调用该可调用对象，并将该可调用对象的返回值推入栈。

Added in version 3.11.

在 3.13 版本发生变更: Callable 现在总是出现在栈的同一位置。

在 3.13 版本发生变更: 有关键字参数的调用现在由*CALL_KW* 处理。

CALL_KW(*argc*)

调用一个可调用对象并传入由 `argc` 指定数量的参数。在栈上为（升序）:

- 可调用对象
- `self``或者``NULL`
- 其余的位置参数
- 关键字参数
- 由关键字参数名称组成的*tuple*

`argc` 是位置参数和关键字参数的总数，不包括 `self`。关键字参数名称元组的长度为关键字参数的数量。

`CALL_KW` 会将所有参数、关键字名称和可调用对象从栈中弹出，用这些参数调用可调用对象，并将返回值推入栈。

Added in version 3.13.

CALL_FUNCTION_EX(*flags*)

调用一个可调用对象并附带位置参数和关键字参数变量集合。如果设置了 *flags* 的最低位，则栈顶包含一个由额外关键字参数组成的映射对象。在调用该可调用对象之前，映射对象和可迭代对象会被分别“解包”并将它们的内容分别作为关键字参数和位置参数传入。`CALL_FUNCTION_EX` 会中栈中弹出所有参数及可调用对象，附带这些参数调用该可调用对象，并将可调用对象所返回的返回值推入栈顶。

Added in version 3.6.

PUSH_NULL

将一个 `NULL` 推入栈。在调用序列中用来匹配*LOAD_METHOD* 针对非方法调用推入栈的 `NULL`。

Added in version 3.11.

MAKE_FUNCTION

在 `STACK[-1]` 位置上将一个新函数对象推入从对象代码构建的栈。

在 3.10 版本发生变更: 旗标值 `0x04` 是一个字符串元组而非字典。

在 3.11 版本发生变更: 位于 `STACK[-1]` 的限定名称已被移除。

在 3.13 版本发生变更: 操作参数曾指示额外的函数属性，现已被移除。现在使用*SET_FUNCTION_ATTRIBUTE*。

SET_FUNCTION_ATTRIBUTE(*flag*)

在函数对象上设置一个属性。将 “STACK[-1]“的元素作为函数对象，`STACK[-2]``作为要设置的属性值，消耗这两个元素，将函数对象留在``STACK[-1]`。参数 flag 决定了要设置哪个属性:

- `0x01` 一个默认值的元组，用于按位置排序的仅限位置形参以及位置或关键字形参
- `0x02` 一个仅限关键字形参的默认值的字典
- `0x04` 一个包含形参标注的字符串元组。
- `0x08` 一个包含用于自由变量的单元的元组，生成一个闭包

Added in version 3.13.

BUILD_SLICE(*argc*)

将一个切片对象推入栈中，*argc* 必须为 2 或 3。如果其为 2，则实现:

```
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end))
```

如果其为 3，则实现：

```
step = STACK.pop()
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end, step))
```

详见内置函数*slice()* 。

EXTENDED_ARG(*ext*)

为任意带有大到无法放入默认的单字节的参数的操作码添加前缀。*ext* 存放一个附加字节作为参数中的高比特位。对于每个操作码，最多允许三个 EXTENDED_ARG 前缀，构成两字节到三字节的参数。

CONVERT_VALUE(*oparg*)

将值转化为字符串，以 “oparg“指定的方式：

```
value = STACK.pop()
result = func(value)
STACK.append(result)
```

- oparg == 1: 在 *value* 上调用*str()*
- oparg == 2: 在 *value* 上调用*repr()*
- oparg == 3: 在 *value* 上调用*ascii()*

用于实现格式字符串字面值（f-字符串）。

Added in version 3.13.

FORMAT_SIMPLE

格式化栈顶的值:

```
value = STACK.pop()
result = value.__format__("")
STACK.append(result)
```

用于实现格式字符串字面值（f-字符串）。

Added in version 3.13.

FORMAT_WITH_SPEC

使用给定的格式说明来格式化给定的值:

```
spec = STACK.pop()
value = STACK.pop()
result = value.__format__(spec)
STACK.append(result)
```

用于实现格式字符串字面值（f-字符串）。

Added in version 3.13.

MATCH_CLASS(*count*)

STACK[-1] 是一个由关键字属性名称组成的元组，STACK[-2] 是要匹配的类，而 STACK[-3] 是匹配的目标主题。*count* 是位置子模式的数量。

弹出 STACK[-1]、STACK[-2] 和 STACK[-3]。如果 STACK[-3] 是 STACK[-2] 的实例并且具有 *count* 和 STACK[-1] 所要求的位置和关键字属性，则推入一个由已提取属性组成的元组。在其他情况下，则推入 None。

Added in version 3.10.

在 3.11 版本发生变更: 在之前的版本中，该指令还会推入一个表示成功 (True) 或失败 (False) 的布尔值。

RESUME (*context*)

空操作。执行内部追踪、调试和优化检查。

context 操作数由两部分组成。最低的两个比特位指明 RESUME 在何处发生：

- 0 在函数的开头。函数不能是生成器、协程或者异步生成器。
- 1 在 yield 表达式之后
- 2 在 yield from 表达式之后
- 3 在 await 表达式之后

如果 RESUME 在 except 深度 1 上发生则下一个比特位值为 1，否则为 0。

Added in version 3.11.

在 3.13 版本发生变更: oparg 值已被修改以包括有关 except 深度的信息

RETURN_GENERATOR

从当前帧中创建一个生成器，协程，或者异步生成器。被用作上述可调用对象的代码对象第一个操作码。清除当前帧，返回新创建的生成器。

Added in version 3.11.

SEND (*delta*)

等价于 STACK[-1] = STACK[-2].send(STACK[-1]) 。被用于 yield from 和 await 语句。

如果调用引发了*StopIteration*，则从栈中弹出最上面的值，推入异常的 value 属性，并将字节码计数器值递增 *delta*。

Added in version 3.11.

HAVE_ARGUMENT

这不是一个真正的操作码。它是 [0,255] 范围内使用与不使用参数的操作码（分别是 < HAVE_ARGUMENT 和 >= HAVE_ARGUMENT ）的分界线。

如果你的应用程序使用了伪指令或专用指令，请改用*hasarg* 多项集。

在 3.6 版本发生变更: 现在每条指令都带有参数，但操作码 < HAVE_ARGUMENT 会忽略它。之前仅限操作码 >= HAVE_ARGUMENT 带有参数。

在 3.12 版本发生变更: 伪指令被添加到*dis* 模块中，对于它们来说，“比较 HAVE_ARGUMENT 以确定其是否使用参数”不再有效。

自 3.13 版本弃用: 使用*hasarg* 来代替。

CALL_INTRINSIC_1

调用内联的函数并附带一个参数。传入 STACK[-1] 作为参数并将 STACK[-1] 设为结果。用于实现对性能不敏感的功能。

调用哪个内置函数取决于操作数：

操作数	描述
INTRINSIC_1_INVALID	无效
INTRINSIC_PRINT	将参数打印到标准输出。被用于 REPL 。
INTRINSIC_IMPORT_ST	为指定模块执行 `import *` 。
INTRINSIC_STOPITERA	从 `StopIteration` 异常中提取返回值。
INTRINSIC_ASYNC_GEN	包裹一个异常生成器值
INTRINSIC_UNARY_POS	执行单目运算符 +
INTRINSIC_LIST_TO_T	将一个列表转换为元组
INTRINSIC_TYPEVAR	创建一个*typing.TypeVar*
INTRINSIC_PARAMSPEC	创建一个*typing.ParamSpec*
INTRINSIC_TYPEVARTU	创建一个*typing.TypeVarTuple*
INTRINSIC_SUBSCRIPT	返回*typing.Generic* 取参数下标。
INTRINSIC_TYPEALIAS	创建一个*typing.TypeAliasType* ； 被用于 `type` 语句。参数是一个由类型别名的名称、类型形参和值组成的元组。

Added in version 3.12.

CALL_INTRINSIC_2

调用内联的函数并附带两个参数。用于实现对性能不敏感的功能:

```
arg2 = STACK.pop()
arg1 = STACK.pop()
result = intrinsic2(arg1, arg2)
STACK.append(result)
```

调用哪个内置函数取决于操作数:

操作数	描述
INTRINSIC_2_INVALID	无效
INTRINSIC_PREP_RERAISE_STAR	计算*ExceptionGroup* 以从 `try-except*` 中引发异常。
INTRINSIC_TYPEVAR_WITH_BOUND	创建一个带范围的*typing.TypeVar* 。
INTRINSIC_TYPEVAR_WITH_CONSTRAINTS	创建一个带约束的*typing.TypeVar* 。
INTRINSIC_SET_FUNCTION_TYPE_PARAMS	为一个函数设置 `__type_params__` 属性。

Added in version 3.12.

伪指令

这些操作码并不出现在 Python 的字节码之中。它们被编译器所使用，但在生成字节码之前会被替代成真正的操作码。

SETUP_FINALLY (*target*)

为下面的代码块设置一个异常处理器。如果发生异常，值栈的级别将恢复到当前状态并将控制权移交给位于 `target` 的异常处理器。

SETUP_CLEANUP (*target*)

与 `SETUP_FINALLY` 类似，但在出现异常的情况下也会将最后一条指令 (`lasti`) 推入栈以便 `RERAISE` 能恢复它。如果出现异常，栈级别值和帧上的最后一条指令将恢复为其当前状态，控制权将转移到 `target` 上的异常处理器。

SETUP_WITH (*target*)

与 `SETUP_CLEANUP` 类似，但在出现异常的情况下会从栈中再弹出一项然后将控制权转移到 `target` 上的异常处理器。

该变体形式用于 `with` 和 `async with` 结构，它们会将上下文管理器的 `__enter__()` 或 `__aenter__()` 的返回值推入栈。

POP_BLOCK

标记与最后一个 SETUP_FINALLY、SETUP_CLEANUP 或 SETUP_WITH 相关联的代码块的结束。

JUMP

JUMP_NO_INTERRUPT

非定向相对跳转指令会被汇编器转换为它们定向版本（forward/backward）

LOAD_CLOSURE(*i*)

将一个引用推入到"fast locals" 存储包含在 i 号槽位的单元内。

请注意在汇编程序中 LOAD_CLOSURE 将被替换为 LOAD_FAST。

在 3.13 版本发生变更: 该操作码现在是一个伪指令。

LOAD_METHOD

经优化的非绑定方法查找。以在 arg 中设置了旗标的 LOAD_ATTR 操作码的形式发出。

33.10.5 操作码集合

提供这些集合用于字节码指令的自动内省:

在 3.12 版本发生变更: 现在此集合还包含一些伪指令和工具化指令。这些操作码的值 >= MIN_PSEUDO_OPCODE 和 >= MIN_INSTRUMENTED_OPCODE。

dis.**opname**

操作名称的序列，可使用字节码来索引。

dis.**opmap**

映射操作名称到字节码的字典

dis.**cmp_op**

所有比较操作名称的序列。

dis.**hasarg**

所有使用参数的字节码的序列

Added in version 3.12.

dis.**hasconst**

访问常量的字节码序列。

dis.**hasfree**

访问自由（闭包）变量 的字节码序列。这里所说的‘自由’是指在当前作用域中被内层作用域所引用的名称，或在外层作用域中被此作用域所引用的名称。它 并不包括对全局或内置作用域的引用。

dis.**hasname**

按名称访问属性的字节码序列。

dis.**hasjump**

具有一个跳转目标的字节码序列。所有跳转都是相对的。

Added in version 3.13.

dis.**haslocal**

访问局部变量的字节码序列。

dis.**hascompare**

布尔运算的字节码序列。

dis.**hasexc**

设置一个异常处理器的字节码序列。

Added in version 3.12.

dis.**hasjrel**

具有相对跳转目标的字节码序列。

自 3.13 版本弃用: 所有跳转现在都是相对的。使用*hasjump*。

dis.**hasjabs**

具有绝对跳转目标的字节码序列。

自 3.13 版本弃用: 所有跳转现在都是相对的。该列表将为空。

33.11 pickletools --- pickle 开发者工具

源代码: Lib/pickletools.py

此模块包含与*pickle*模块内部细节有关的多个常量，一些关于具体实现的详细注释，以及一些能够分析封存数据的有用函数。此模块的内容对需要操作*pickle*的 Python 核心开发者来说很有用处；*pickle*的一般用户则可能会感觉*pickletools*模块与他们无关。

33.11.1 命令行语法

Added in version 3.2.

当从命令行唤起时，`python -m pickletools`将对一个或更多 pickle 文件的内容进行拆解。请注意如果你查看 pickle 中保存的 Python 对象而非 pickle 格式的细节，你可能需要改用`-m pickle`。但是，当你想检查的 pickle 文件来自某个不受信任的源时，`-m pickletools`是更安全的选择，因为它不会执行 pickle 字节码。

例如，对于一个封存在文件`x.pickle`中的元组`(1, 2)`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

命令行选项

-a, --annotate

使用简短的操作码描述来标注每一行。

-o, --output=<file>

输出应当写入到的文件名称。

-l, --indentlevel=<num>

一个新的 MARK 层级所需缩进的空格数。

-m, --memo

当反汇编多个对象时，保留各个反汇编的备忘记录。

-p, --preamble=<preamble>

当指定一个以上的 pickle 文件时，在每次反汇编之前打印给定的前言。

33.11.2 编程接口

pickletools.**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

将 pickle 的符号化反汇编数据输出到文件型对象 *out*，默认为 `sys.stdout`。*pickle* 可以是一个字符串或一个文件型对象。*memo* 可以是一个将被用作 pickle 的备忘记录的 Python 字典；它可被用来对由同一封存器创建的多个封存对象执行反汇编。由 `MARK` 操作码指明的每个连续级别将会缩进 *indentlevel* 个空格。如果为 *annotate* 指定了一个非零值，则输出中的每个操作码将以一个简短描述来标注。*annotate* 的值会被用作标注所应开始的列的提示。

在 3.2 版本发生变更: 增加了 *annotate* 形参。

pickletools.**genops**(*pickle*)

提供包含 pickle 中所有操作码的*iterator*，返回一个 `(opcode, arg, pos)` 三元组的序列。*opcode* 是 `OpcodeInfo` 类的一个实例；*arg* 是 Python 对象形式的 opcode 参数的已解码值；*pos* 是 opcode 所在的位置。*pickle* 可以是一个字符串或一个文件型对象。

pickletools.**optimize**(*picklestring*)

在消除未使用的 `PUT` 操作码之后返回一个新的等效 pickle 字符串。优化后的 pickle 将更为简短，耗费更少的传输时间，要求更少的存储空间并能更高效地解封。

CHAPTER 34

Windows 系统相关模块

本章节叙述的模块只在 Windows 平台上可用。

34.1 msvcrt --- 来自 MS VC++ 运行时的有用例程

这些函数提供了对 Windows 平台上一些有用功能的访问。一些更高层级的模块使用这些函数来构建其服务的 Windows 实现。例如，*getpass* 模块在*getpass()* 函数的实现中就用到了它。

关于这些函数的更多信息可以在平台 API 文档中找到。

该模块实现了控制台 I/O API 的普通和宽字符变体。普通的 API 只处理 ASCII 字符，国际化应用受限。应该尽可能地使用宽字符 API 。

在 3.3 版本发生变更: 此模块中过去会引发*IOError* 的操作现在将引发*OSError*。

34.1.1 文件操作

msvcrt.**locking**(*fd*, *mode*, *nbytes*)

基于文件描述符 *fd* 从 C 运行时锁定文件的某个部分。失败时将引发*OSError*。锁定的文件区域从当前文件位置扩展 *nbytes* 个字节，并可能持续到超出文件末尾。*mode* 必须为下面列出的 LK_* 常量之一。一个文件中的多个区域可以被同时锁定，但是不能重叠。相邻的区域不会被合并；它们必须被单独解锁。

引发一个审计事件 msvcrt.locking 并附带参数 fd, mode, nbytes。

msvcrt.**LK_LOCK**

msvcrt.**LK_RLCK**

锁定指定的字节数据。如果字节数据无法被锁定，程序会在 1 秒后立即重试。如果在 10 次尝试后字节数据仍无法被锁定，则会引发*OSError*。

msvcrt.**LK_NBLCK**

msvcrt.**LK_NBRLCK**

锁定指定的字节数据。如果字节数据无法被锁定，则会引发*OSError*。

msvcrt.**LK_UNLCK**

解锁指定的字节数据，该对象必须在之前被锁定。

msvcrt.**setmode**(*fd*, *flags*)

设置文件描述符 *fd* 的行结束符转写模式。要将其设为文本模式，则 *flags* 应当为*os.O_TEXT*；设为二进制模式，则应当为*os.O_BINARY*。

msvcrt.**open_osfhandle**(*handle*, *flags*)

基于文件句柄 *handle* 创建一个 C 运行时文件描述符。*flags* 形参应当是*os.O_APPEND*, *os.O_RDONLY*, *os.O_TEXT* 和*os.O_NOINHERIT* 按位 OR 的结果。返回的文件描述符可以被用作传给*os.fdopen()* 的形参以创建一个文件对象。

该文件描述符默认是不可继承的。传入*os.O_NOINHERIT* 旗标可使它成为非不可继承的。

引发一个审计事件 `msvcrt.open_osfhandle` 并附带参数 `handle`, `flags`。

msvcrt.**get_osfhandle**(*fd*)

返回文件描述符 *fd* 的文件句柄。如果 *fd* 不能被识别则会引发*OSError*。

引发一个审计事件 `msvcrt.get_osfhandle` 并附带参数 `fd`。

34.1.2 控制台 I/O

msvcrt.**kbhit**()

如果正在等待读取一个按键则返回一个非零值。在其他情况下，将返回 0。

msvcrt.**getch**()

读取一个按键并将结果字符以字节串形式返回。不会回显到控制台。如果没有任何按键可用则此调用将会阻塞，但它不会等待 `Enter` 被按下。如果按下的键是一个特殊功能键，此函数将返回 `'\000'` 或 `'\xe0'`；下一次调用将返回键代码。`Control-C` 按键无法使用此函数来读取。

msvcrt.**getwch**()

getch() 的宽字符版本，返回一个 Unicode 值。

msvcrt.**getche**()

类似于*getch()*，但如果按键是代表一个可打印字符则它将被回显。

msvcrt.**getwche**()

getche() 的宽字符版本，返回一个 Unicode 值。

msvcrt.**putch**(*char*)

将字符串 *char* 打印到终端，不使用缓冲区。

msvcrt.**putwch**(*unicode_char*)

putch() 的宽字符版本，接受一个 Unicode 值。

msvcrt.**ungetch**(*char*)

使得字节串 *char* 被“推回”终端缓冲区；它将是被*getch()* 或*getche()* 读取的下一个字符。

msvcrt.**ungetwch**(*unicode_char*)

ungetch() 的宽字符版本，接受一个 Unicode 值。

34.1.3 其他函数

msvcrt.**heapmin**()

强制 `malloc()` 堆清空自身并将未使用的块返回给操作系统。失败时，这将引发*OSError*。

msvcrt.**set_error_mode**(*mode*)

更改 C 运行时为可能终止程序的错误写入错误消息的位置。*mode* 必须是下面列出的 `OUT_*` 常量或*REPORT_ERRMODE* 中的一个。返回旧设置或者在有错误发生时返回 -1。仅在 Python 调试构建版中可用。

msvcrt.**OUT_TO_DEFAULT**

错误 sink 是由应用程序的类型决定的。仅在 Python 调试构建版中可用。

msvcrt.**OUT_TO_STDERR**

错误 sink 是某个标准错误。仅在 Python 调试构建版中可用。

msvcrt.**OUT_TO_MSGBOX**

错误 sink 是某个消息框。仅在 Python 调试构建版中可用。

msvcrt.**REPORT_ERRMODE**

报告当前错误模式值。仅在 Python 调试构建版中可用。

msvcrt.**CrtSetReportMode**(*type*, *mode*)

为 `_CrtDbgReport()` 在 MS VC++ 运行时中生成的特定报告类型指定一个目标或多个目标。*type* 必须是下面列出的 `CRT_*` 常量之一。*mode* 必须是下面列出的 `CRTDBG_*` 常量之一。仅在 Python 调试构建版中可用。

msvcrt.**CrtSetReportFile**(*type*, *file*)

在你使用*CrtSetReportMode()* 来指定*CRTDBG_MODE_FILE* 之后，你可以指定接收消息文本的文件句柄。*type* 必须是下面列出的 `CRT_*` 常量之一。*file* 应当是你希望指定的文件句柄。仅在 Python 调试构建版中可用。

msvcrt.**CRT_WARN**

不需要立即关注的警告、消息和信息。

msvcrt.**CRT_ERROR**

需要立即关注的错误、不可恢复的问题和议题。

msvcrt.**CRT_ASSERT**

断言失败

msvcrt.**CRTDBG_MODE_DEBUG**

将消息写至调试器的输出窗口。

msvcrt.**CRTDBG_MODE_FILE**

将消息写入用户提供的文件句柄。应当调用*CrtSetReportFile()* 来定义要用作写入目标的特定文件或流。

msvcrt.**CRTDBG_MODE_WNDW**

创建一个消息框来显示消息并提供 `Abort`, `Retry` 和 `Ignore` 等按钮。

msvcrt.**CRTDBG_REPORT_MODE**

返回指定 *type* 当前的 *mode*。

msvcrt.**CRT_ASSEMBLY_VERSION**

CRT 汇编版，来自 `crtassem.h` 头文件。

msvcrt.**VC_ASSEMBLY_PUBLICKEYTOKEN**

VC 汇编公钥凭据，来自 `crtassem.h` 头文件。

msvcrt.**LIBRARIES_ASSEMBLY_NAME_PREFIX**

库汇编名称前缀，来自 `crtassem.h` 头文件。

34.2 `winreg` --- Windows 注册表访问

这些函数将 Windows 注册表 API 暴露给 Python。为了确保即便程序员忘记显式关闭时也能够正确关闭，这里没有用整数作为注册表句柄，而是采用了*句柄对象*。

在 3.3 版本发生变更: 模块中有几个函数用于触发*WindowsError*，此异常现在是*OSError* 的别名。

34.2.1 函数

该模块提供了下列函数：

winreg.**CloseKey**(*hkey*)

关闭之前打开的注册表键。参数 *hkey* 指之前打开的键。

> **备注**
>
> 如果没有使用该方法关闭 *hkey* (或者通过*hkey.Close()*)，在对象 *hkey* 被 Python 销毁时会将其关闭。

winreg.**ConnectRegistry**(*computer_name*, *key*)

建立到另一台计算机上的预定义注册表句柄的连接，并返回一个句柄对象。

computer_name 是远程计算机的名称，以 `r"\\computername"` 的形式。如果是 `None` ，将会使用本地计算机。

key 是所连接到的预定义句柄。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

引发一个审计事件 `winreg.ConnectRegistry` 并附带参数 `computer_name`, `key`。

在 3.3 版本发生变更: 参考上文。

winreg.**CreateKey**(*key*, *sub_key*)

创建或打开特定的键，返回一个*handle* 对象。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key 是用于命名该方法所打开或创建的键的字符串。

如果 *key* 是预定义键之一，*sub_key* 可能会是 `None`。该情况下，返回的句柄就是传入函数的句柄。

如果键已经存在，则该函数打开已经存在的该键。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

引发一个审计事件 `winreg.CreateKey` 并附带参数 `key`, `sub_key`, `access`。

引发一个审计事件 `winreg.OpenKey/result` 并附带参数 `key`。

在 3.3 版本发生变更: 参考上文。

winreg.**CreateKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

创建或打开特定的键，返回一个*handle* 对象。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key 是用于命名该方法所打开或创建的键的字符串。

reserved 是一个保留的整数，必须是零。默认值为零。

access 为一个整数，用于给键的预期安全访问指定访问掩码。默认值为*KEY_WRITE*。参阅*Access Rights* 了解其它允许值。

如果 *key* 是预定义键之一，*sub_key* 可能会是 `None`。该情况下，返回的句柄就是传入函数的句柄。

如果键已经存在，则该函数打开已经存在的该键。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

引发一个审计事件 `winreg.CreateKey` 并附带参数 `key`, `sub_key`, `access`。

引发一个审计事件 `winreg.OpenKey/result` 并附带参数 `key`。

Added in version 3.2.

在 3.3 版本发生变更: 参考上文。

winreg.**DeleteKey**(*key*, *sub_key*)

删除指定的键。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 这个字符串必须是由 *key* 参数所指定键的一个子项。该值项不可以是 None，同时键也不可以有子项。

该方法不能删除带有子项的键。

如果方法成功，则整个键，包括其所有值项都会被移除。如果方法失败，则引发一个*OSError* 异常。

引发一个审计事件 winreg.DeleteKey 并附带参数 key, sub_key, access。

在 3.3 版本发生变更: 参考上文。

winreg.**DeleteKeyEx**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

删除指定的键。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 这个字符串必须是由 *key* 参数所指定键的一个子项。该值项不可以是 None，同时键也不可以有子项。

reserved 是一个保留的整数，必须是零。默认值为零。

access 是一个指定描述注册表键所需的安全权限的访问掩码的整数。默认值为*KEY_WOW64_64KEY*。在 32-bit Windows 上，WOW64 常量会被忽略。请参阅访问权限 了解其他可用的值。

该方法不能删除带有子项的键。

如果方法成功，则整个键，包括其所有值项都会被移除。如果方法失败，则引发一个*OSError* 异常。

在不支持的 Windows 版本之上，将会引发*NotImplementedError* 异常。

引发一个审计事件 winreg.DeleteKey 并附带参数 key, sub_key, access。

Added in version 3.2.

在 3.3 版本发生变更: 参考上文。

winreg.**DeleteValue**(*key*, *value*)

从某个注册键中删除一个命名值项。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

value 为标识所要删除值项的字符串。

引发一个审计事件 winreg.DeleteValue 并附带参数 key, value。

winreg.**EnumKey**(*key*, *index*)

列举某个已经打开注册表键的子项，并返回一个字符串。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

index 为一个整数，用于标识所获取键的索引。

每次调用该函数都会获取一个子项的名字。通常它会被反复调用，直到引发*OSError* 异常，这说明已经没有更多的可用值了。

引发一个审计事件 winreg.EnumKey 并附带参数 key, index。

在 3.3 版本发生变更: 参考上文。

winreg.**EnumValue**(*key*, *index*)

列举某个已经打开注册表键的值项，并返回一个元组。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

index 为一个整数，用于标识要获取值项的索引。

每次调用该函数都会获取一个子项的名字。通常它会被反复调用，直到引发*OSError* 异常，这说明已经没有更多的可用值了。

结果为 3 元素的元组。

索引	含意
0	用于标识值项名称的字符串。
1	保存值项数据的对象，其类型取决于背后的注册表类型。
2	标识值项数据类型的整数。（请查阅*SetValueEx()* 文档中的表格）

引发一个审计事件 `winreg.EnumValue` 并附带参数 `key`, `index`。

在 3.3 版本发生变更: 参考上文。

`winreg.`**`ExpandEnvironmentStrings`**(*str*)

Expands environment variable placeholders `%NAME%` in strings like *REG_EXPAND_SZ*:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

引发一个审计事件 `winreg.ExpandEnvironmentStrings` 并附带参数 `str`。

`winreg.`**`FlushKey`**(*key*)

将某个键的所有属性写入注册表。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

没有必要调用*FlushKey()* 去改动注册表键。注册表的变动是由其延迟刷新机制更新到磁盘的。在系统关机时，也会将注册表的变动写入磁盘。与*CloseKey()* 不同，*FlushKey()* 方法只有等到所有数据都写入注册表后才会返回。只有需要绝对确认注册表变动已写入磁盘时，应用程序才应去调用*FlushKey()*。

备注

如果不知道是否要调用*FlushKey()*，可能就是不需要。

`winreg.`**`LoadKey`**(*key*, *sub_key*, *file_name*)

在指定键之下创建一个子键，并将指定文件中的注册表信息存入该子键中。

key 是由*ConnectRegistry()* 返回的句柄，或者是常量*HKEY_USERS* 或*HKEY_LOCAL_MACHINE*。

sub_key 是个字符串，用于标识需要载入的子键。

file_name 是要加载注册表数据的文件名。该文件必须是用*SaveKey()* 函数创建的。在文件分配表（FAT）文件系统中，文件名可能不带扩展名。

如果调用*LoadKey()* 的进程没有 `SE_RESTORE_PRIVILEGE` 特权则调用将失败。请注意特权与权限是不同的 -- 更多细节请参阅 RegLoadKey 文档。

如果 *key* 是由*ConnectRegistry()* 返回的句柄，那么 *file_name* 指定的路径是相对于远程计算机而言的。

引发一个审计事件 `winreg.LoadKey` 并附带参数 `key`, `sub_key`, `file_name`。

`winreg.`**`OpenKey`**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

`winreg.`**`OpenKeyEx`**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

打开指定的注册表键，返回*handle 对象*。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是个字符串，标识了需要打开的子键。

reserved 是个保留整数，必须为零。默认值为零。

access 是个指定访问掩码的整数，掩码描述了注册表键所需的安全权限。默认值为*KEY_READ*。其他合法值参见访问权限。

返回结果为一个新句柄，指向指定的注册表键。

如果调用失败，则会触发*OSError* 。

引发一个审计事件 `winreg.OpenKey` 并附带参数 `key`, `sub_key`, `access`。

引发一个审计事件 `winreg.OpenKey/result` 并附带参数 `key`。

在 3.2 版本发生变更: 允许使用命名参数。

在 3.3 版本发生变更: 参考上文。

`winreg.`**`QueryInfoKey`**(*key*)

以元组形式返回某注册表键的信息。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

结果为 3 元素的元组。

索引	含意
0	整数值，给出了此注册表键的子键数量。
1	整数值，给出了此注册表键的值的数量。
2	整数值，给出了此注册表键的最后修改时间，单位为自 1601 年 1 月 1 日以来的 100 纳秒。

引发一个审计事件 `winreg.QueryInfoKey` 并附带参数 `key`。

`winreg.`**`QueryValue`**(*key*, *sub_key*)

读取某键的未命名值，形式为字符串。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是个字符串，用于保存与某个值相关的子键名称。如果本参数为 `None` 或空，函数将读取由*SetValue()* 方法为 *key* 键设置的值。

注册表中的值包含名称、类型和数据。本方法将读取注册表键值的第一个名称为 `NULL` 的数据。可是底层的 API 调用不会返回类型，所以只要有可能就一定要使用*QueryValueEx()*。

引发一个审计事件 `winreg.QueryValue` 并附带参数 `key`, `sub_key`, `value_name`。

`winreg.`**`QueryValueEx`**(*key*, *value_name*)

读取已打开注册表键指定值名称的类型和数据。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

value_name 是字符串，表示要查询的值。

结果为二元组:

索引	含意
0	注册表项的值。
1	整数值，给出该值的注册表类型（请查看文档中的表格了解*SetValueEx()* ）。

引发一个审计事件 `winreg.QueryValue` 并附带参数 `key`, `sub_key`, `value_name`。

`winreg.`**`SaveKey`**(*key*, *file_name*)

将指定注册表键及其所有子键存入指定的文件。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

file_name 是要保存注册表数据的文件名。该文件不能已存在。如果文件名包括扩展名，也不能在文件分配表（FAT）文件系统中用于*LoadKey()* 方法。

如果 *key* 是代表远程计算机上的注册表键，那么 *file_name* 所描述的路径就是相对于远程计算机的。本方法的调用方必须拥有 **SeBackupPrivilege** 安全特权。请注意特权与权限是不同的 -- 更多细节请参阅 Conflicts Between User Rights and Permissions 文档。

本函数将 `NULL` 传给 API 的 *security_attributes*。

引发一个审计事件 `winreg.SaveKey` 并附带参数 `key`, `file_name`。

`winreg.`**`SetValue`**(*key*, *sub_key*, *type*, *value*)

将值与指定的注册表键关联。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是个字符串，用于命名与该值相关的子键。

type 是个整数，用于指定数据的类型。目前这必须是*REG_SZ*，意味着只支持字符串。请用*SetValueEx()* 函数支持其他的数据类型。

value 是设置新值的字符串。

如果 *sub_key* 参数指定的注册表键不存在，SetValue 函数会创建一个。

值的长度受到可用内存的限制。较长的值（超过 2048 字节）应存为文件，并将文件名存入配置注册表。这有助于提高注册表的使用效率。

由 *key* 参数标识的注册表键，必须已用*KEY_SET_VALUE* 方式打开。

引发一个审计事件 `winreg.SetValue` 并附带参数 `key`, `sub_key`, `type`, `value`。

`winreg.`**`SetValueEx`**(*key*, *value_name*, *reserved*, *type*, *value*)

将数据存入已打开的注册表键的值中。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

value_name 是个字符串，用于命名与值相关的子键。

reserved 可以是任意数据——传给 API 的总是 0。

type 是个整数，用于指定数据的类型。请参阅*Value Types* 了解可用的类型。

value 是设置新值的字符串。

本方法也可为指定的注册表键设置额外的值和类型信息。注册表键必须已用*KEY_SET_VALUE* 方式打开。

请用*CreateKey()* 或*OpenKey()* 方法打开注册表键。

值的长度受到可用内存的限制。较长的值（超过 2048 字节）应存为文件，并将文件名存入配置注册表。这有助于提高注册表的使用效率。

引发一个审计事件 `winreg.SetValue` 并附带参数 `key`, `sub_key`, `type`, `value`。

`winreg.`**`DisableReflectionKey`**(*key*)

禁用运行于 64 位操作系统的 32 位进程的注册表重定向。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

如果在 32 位操作系统上执行，一般会触发*NotImplementedError*。

如果注册表键不在重定向列表中，函数会调用成功，但没有实际效果。禁用注册表键的重定向不会影响任何子键的重定向。

引发一个审计事件 `winreg.DisableReflectionKey` 并附带参数 `key`。

`winreg.`**`EnableReflectionKey`**(*key*)

恢复已禁用注册表键的重定向。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

如果在 32 位操作系统上执行，一般会触发*NotImplementedError*。

恢复注册表键的重定向不会影响任何子键的重定向。

引发一个审计事件 winreg.EnableReflectionKey 并附带参数 key。

winreg.**QueryReflectionKey**(*key*)

确定给定注册表键的重定向状况。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

如果重定向已禁用则返回 True。

如果在 32 位操作系统上执行，一般会触发*NotImplementedError*。

引发一个审计事件 winreg.QueryReflectionKey 并附带参数 key。

34.2.2 常量

以下常量被定义以供多个*winreg* 函数使用。

HKEY_* 常量

winreg.**HKEY_CLASSES_ROOT**

本注册表键下的注册表项定义了文件的类型（或类别）及相关属性。Shell 和 COM 应用程序将使用该注册表键下保存的信息。

winreg.**HKEY_CURRENT_USER**

属于该注册表键的表项定义了当前用户的偏好。这些偏好值包括环境变量设置、程序组数据、颜色、打印机、网络连接和应用程序参数。

winreg.**HKEY_LOCAL_MACHINE**

属于该注册表键的表项定义了计算机的物理状态，包括总线类型、系统内存和已安装软硬件等数据。

winreg.**HKEY_USERS**

属于该注册表键的表项定义了当前计算机中新用户的默认配置和当前用户配置。

winreg.**HKEY_PERFORMANCE_DATA**

属于该注册表键的表项可用于读取性能数据。这些数据其实并不存放于注册表中；注册表提供功能让系统收集数据。

winreg.**HKEY_CURRENT_CONFIG**

包含有关本地计算机系统当前硬件配置的信息。

winreg.**HKEY_DYN_DATA**

Windows 98 以上版本不使用该注册表键。

访问权限

更多信息，请参阅 注册表密钥安全和访问。

winreg.**KEY_ALL_ACCESS**

组合了 STANDARD_RIGHTS_REQUIRED 、*KEY_QUERY_VALUE* 、*KEY_SET_VALUE* 、*KEY_CREATE_SUB_KEY* 、*KEY_ENUMERATE_SUB_KEYS* 、*KEY_NOTIFY* 和*KEY_CREATE_LINK* 访问权限。

winreg.**KEY_WRITE**

组合了 STANDARD_RIGHTS_WRITE 、*KEY_SET_VALUE* 和*KEY_CREATE_SUB_KEY* 访问权限。

winreg.**KEY_READ**

组合了 STANDARD_RIGHTS_READ 、*KEY_QUERY_VALUE* 、*KEY_ENUMERATE_SUB_KEYS* 和*KEY_NOTIFY* 。

winreg.**KEY_EXECUTE**

等价于*KEY_READ*。

winreg.**KEY_QUERY_VALUE**
查询注册表键值时需要用到。

winreg.**KEY_SET_VALUE**
创建、删除或设置注册表值时需要用到。

winreg.**KEY_CREATE_SUB_KEY**
创建注册表键的子键时需要用到。

winreg.**KEY_ENUMERATE_SUB_KEYS**
枚举注册表键的子键时需要用到。

winreg.**KEY_NOTIFY**
为注册表键或子键请求修改通知时需要用到。

winreg.**KEY_CREATE_LINK**
保留给系统使用。

64 位系统特有

详情请参阅 Accessing an Alternate Registry View。

winreg.**KEY_WOW64_64KEY**
表示一个应用程序在 64 位 Windows 上应当在 64 位的注册表视图上进行操作。在 32 位 Windows 上，此常量会被忽略。

winreg.**KEY_WOW64_32KEY**
表示一个应用程序在 64 位 Windows 上应当在 32 位的注册表视图上进行操作。在 32 位 Windows 上，此常量会被忽略。

注册表值的类型

详情请参阅 Registry Value Types。

winreg.**REG_BINARY**
任意格式的二进制数据。

winreg.**REG_DWORD**
32 位数字。

winreg.**REG_DWORD_LITTLE_ENDIAN**
32 位低字节序格式的数字。相当于*REG_DWORD*。

winreg.**REG_DWORD_BIG_ENDIAN**
32 位高字节序格式的数字。

winreg.**REG_EXPAND_SZ**
包含环境变量（%PATH%）的字符串，以空字符结尾。

winreg.**REG_LINK**
Unicode 符号链接。

winreg.**REG_MULTI_SZ**
一串以空字符结尾的字符串，最后以两个空字符结尾。Python 会自动处理这种结尾形式。

winreg.**REG_NONE**
未定义的类型。

winreg.**REG_QWORD**
64 位数字。

Added in version 3.6.

winreg.**REG_QWORD_LITTLE_ENDIAN**

64 位低字节序格式的数字。相当于*REG_QWORD*。

Added in version 3.6.

winreg.**REG_RESOURCE_LIST**

设备驱动程序资源列表。

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**

硬件设置。

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**

硬件资源列表。

winreg.**REG_SZ**

空字符结尾的字符串。

34.2.3 注册表句柄对象

该对象封装了 Windows HKEY 对象，对象销毁时会自动关闭。为确保资源得以清理，可调用*Close()* 方法或*CloseKey()* 函数。

本模块中的所有注册表函数都会返回注册表句柄对象。

本模块中所有接受注册表句柄对象的注册表函数，也能接受一个整数，但鼓励大家使用句柄对象。

句柄对象为 __bool__() 提供语义——因此

```
if handle:
    print("Yes")
```

将会打印出 Yes 。

句柄对象还支持比较语义，因此若多个句柄对象都引用了同一底层 Windows 句柄值，那么比较操作结果将为 True。

句柄对象可转换为整数（如利用内置函数*int()*），这时会返回底层的 Windows 句柄值。用*Detach()* 方法也可返回整数句柄，同时会断开与 Windows 句柄的连接。

PyHKEY.**Close**()

关闭底层的 Windows 句柄。

如果句柄已关闭，不会引发错误。

PyHKEY.**Detach**()

断开与 Windows 句柄的连接。

结果为一个整数，存有被断开连接之前的句柄值。如果该句柄已断开连接或关闭，则返回 0。

调用本函数后，注册表句柄将被迅速禁用，但并没有关闭。当需要底层的 Win32 句柄在句柄对象的生命周期之后仍然存在时，可以调用这个函数。

引发一个审计事件 winreg.PyHKEY.Detach 并附带参数 key。

PyHKEY.**__enter__**()

PyHKEY.**__exit__**(**exc_info*)

HKEY 对象实现了 __enter__() 和 __exit__() 方法，因此支持 with 语句的上下文协议：

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # 使用 key 进行操作
```

在离开 with 语句块时，*key* 会自动关闭。

34.3 winsound --- 针对 Windows 的声音播放接口

通过*winsound*模块可访问 Windows 平台的基础音频播放机制。包括一些函数和几个常量。

winsound.**Beep**(*frequency*, *duration*)

让 PC 的扬声器发出提示音。*frequency* 参数可指定声音的频率，单位是赫兹，必须位于 37 到 32,767 之间。*duration* 参数则指定了声音应持续的毫秒数。若系统无法让扬声器发声，则会触发*RuntimeError*。

winsound.**PlaySound**(*sound*, *flags*)

由平台 API 调用底层的 PlaySound() 函数。*sound* 形参可以是一个文件名、系统声音别名、*bytes-like object* 形式的音频数据或者 None。对它的解读取决于 *flags* 的值，它可以为下述常量通过按位或运算得到的组合。如果 *sound* 形参为 None，则将停止当前播放的任何波形声音。如果系统提示错误，则会引发*RuntimeError*。

winsound.**MessageBeep**(*type=MB_OK*)

由平台 API 调用底层的 MessageBeep() 函数。这将播放在注册表中指定的声音。*type* 参数指定要播放的声音；可能的值为 -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION 和 MB_OK，如下文所述。值为 -1 将产生一个简单的“哔”；这是在无法播放其他声音时的最终回退项。如果系统提示错误，则会引发*RuntimeError*。

winsound.**SND_FILENAME**

参数 *sound* 指明 WAV 文件名。不要与*SND_ALIAS* 一起使用。

winsound.**SND_ALIAS**

参数 *sound* 是注册表内关联的音频名称。如果注册表中无此名称，则播放系统默认的声音，除非同时设定了*SND_NODEFAULT*。如果没有注册默认声音，则会触发*RuntimeError*。请勿与*SND_FILENAME* 一起使用。

所有的 Win32 系统至少支持以下音频名称；大多数系统支持的音频都多于这些：

PlaySound() name 参数	对应的控制面板音频名
'SystemAsterisk'	星号
'SystemExclamation'	感叹号
'SystemExit'	退出 Windows
'SystemHand'	关键性停止
'SystemQuestion'	问题

例如：

```
import winsound
# 播放 Windows 退出音效。
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# 可能会播放 Windows 默认音效，如果有注册的话
# （因为 "*" 可能不是任何音效的注册名称）。
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**

循环播放音频。为避免阻塞，必须同时使用*SND_ASYNC* 标志。不能与*SND_MEMORY* 一起使用。

winsound.**SND_MEMORY**

PlaySound() 的 *sound* 形参是一个 WAV 文件的内存镜像，作为一个*bytes-like object*。

> **备注**
>
> 本模块不支持异步播放音频的内存镜像，所以该标志和*SND_ASYNC* 的组合将触发*RuntimeError*。

winsound.**SND_PURGE**
停止播放指定音频的所有实例。

> **备注**
>
> 新版 Windows 平台不支持本标志。

winsound.**SND_ASYNC**
立即返回，允许异步播放音频。

winsound.**SND_NODEFAULT**
即便找不到指定的音频，也不播放系统默认音频。

winsound.**SND_NOSTOP**
不打断正在播放的音频。

winsound.**SND_NOWAIT**
如果音频驱动程序忙，则立即返回。

> **备注**
>
> 新版 Windows 平台不支持本标志。

winsound.**SND_APPLICATION**
sound 形参是注册表中应用程序专属的别名。该旗标可与*SND_ALIAS* 旗标组合以指定由应用程序定义的声音别名。

winsound.**MB_ICONASTERISK**
播放 `SystemDefault` 音频。

winsound.**MB_ICONEXCLAMATION**
播放 `SystemExclamation` 音频。

winsound.**MB_ICONHAND**
播放 `SystemHand` 音频。

winsound.**MB_ICONQUESTION**
播放 `SystemQuestion` 音频。

winsound.**MB_OK**
播放 `SystemDefault` 音频。

CHAPTER 35

Unix 专有服务

本章描述的模块提供了 Unix 操作系统独有特性的接口，在某些情况下也适用于它的某些或许多衍生版。以下为模块概览：

35.1 `posix` --- 最常见的 POSIX 系统调用

此模块提供了对基于 C 标准和 POSIX 标准（一种稍加修改的 Unix 接口）进行标准化的系统功能的访问。

Availability: Unix.

请勿直接导入此模块。而应导入*os* 模块，它提供了此接口的 可移植版本。在 Unix 上，*os* 模块提供了*posix* 接口的一个超集。在非 Unix 操作系统上*posix* 模块将不可用，但会通过*os* 接口提供它的一个可用子集。一旦导入了*os*，用它替代*posix* 时就 没有性能惩罚。此外，*os* 还提供了一些附加功能，例如在 `os.environ` 中的某个条目被修改时会自动调用*putenv()*。

错误将作为异常被报告；对于类型错误会给出普通异常，而系统调用所报告的异常则会引发*OSError*。

35.1.1 大文件支持

某些操作系统（包括 AIX 和 Solaris）可对 `int` 和 `long` 为 32 位值的 C 编程模型提供大于 2 GiB 文件的支持。这在通常情况下是以将相关数据的大小和偏移量类型定义为 64 位值的方式来实现的。这样的文件有时被称为 大文件。

Python 中的大文件支持会在 `off_t` 的大小超过 `long` 且 `long long` 的大小至少与 `off_t` 一样时被启用。要启用此模式可能必须在启用特定编译旗标的情况下配置和编译 Python。例如，在 Solaris 2.6 和 2.7 中你需要执行这样的操作:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

在支持大文件的 Linux 系统中，可以这样做:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

35.1.2 重要的模块内容

除了 *os* 模块文档已说明的许多函数，*posix* 还定义了下列数据项:

posix.**environ**

一个表示解释器启动时间点的字符串环境的字典。键和值的类型在 Unix 上为 bytes 而在 Windows 上为 str。例如，environ[b'HOME'] (Windows 上的 environ['HOME']) 是你的家目录的路径名，等价于 C 中的 getenv("HOME")。

修改此字典不会影响由 *execv()*, *popen()* 或 *system()* 所传入的字符串环境；如果你需要修改环境，请将 environ 传给 *execve()* 或者为 *system()* 或 *popen()* 的命令字符串添加变量赋值和 export 语句。

在 3.2 版本发生变更: 在 Unix 上，键和值为 bytes 类型。

备注

os 模块提供了对 environ 的替代实现，它会在被修改时更新环境。还要注意更新 *os.environ* 将导致此字典失效。推荐使用这个 *os* 模块版本而不是直接访问 *posix* 模块。

35.2 pwd --- 密码数据库

此模块可以访问 Unix 用户账户名及密码数据库，在所有 Unix 版本上均可使用。

Availability: Unix, not WASI, not iOS.

密码数据库中的条目以元组对象返回，属性对应 passwd 中的结构（属性如下所示，可参考 <pwd.h>）:

索引	属性	含意
0	pw_name	登录名
1	pw_passwd	密码，可能已经加密
2	pw_uid	用户 ID 数值
3	pw_gid	组 ID 数值
4	pw_gecos	用户名或备注
5	pw_dir	用户主目录
6	pw_shell	用户的命令解释器

其中 uid 和 gid 是整数，其他是字符串，如果找不到对应的项目，抛出 *KeyError* 异常。

备注

在传统的 Unix 中字段 pw_passwd 通常包含一个使用 DES 派生算法加密的口令。不过大多数现代 Unix 使用一种所谓的 影子口令系统。在这些 Unix 上 *pw_passwd* 字段只包含一个星号 ('*') 或字母 'x' 而加密的口令存储在非全局可读的文件 /etc/shadow 中。而 *pw_passwd* 字段是否包含任何有用内容则取决于具体的系统。

本模块定义如下内容:

pwd.**getpwuid**(*uid*)

给定用户的数值 ID，返回密码数据库的对应项目。

pwd.**getpwnam**(*name*)

给定用户名，返回密码数据库的对应项目。

pwd.**getpwall**()

返回密码数据库中所有项目的列表，顺序不是固定的。

参见

模块*grp*
针对用户组数据库的接口，与本模块类似。

35.3 grp --- 组数据库

该模块提供对 Unix 组数据库的访问。它在所有 Unix 版本上都可用。

Availability: Unix, not WASI, not Android, not iOS.

组数据库条目被报告为类似元组的对象，其属性对应于 group 结构的成员 (下面的属性字段，请参见 <grp.h>):

索引	属性	含意
0	gr_name	组名
1	gr_passwd	（加密的）组密码；通常为空
2	gr_gid	数字组 ID
3	gr_mem	组内所有成员的用户名

gid 是整数，名称和密码是字符串，成员列表是字符串列表。(注意，大多数用户未根据密码数据库显式列为所属组的成员。请检查两个数据库以获取完整的成员资格信息。还要注意，以 + 或 - 开头的 gr_name 可能是 YP/NIS 引用，可能无法通过*getgrnam()* 或*getgrgid()* 访问。)

本模块定义如下内容:

grp.**getgrgid**(*id*)

返回给定数字组 ID 的组数据库条目。如果请求的条目无法找到则会引发*KeyError*。

在 3.10 版本发生变更: 对于非整数参数如浮点数或字符串将引发*TypeError*。

grp.**getgrnam**(*name*)

返回给定组名的组数据库条目。如果找不到要求的条目，则会引发*KeyError* 错误。

grp.**getgrall**()

以任意顺序返回所有可用组条目的列表。

参见

模块*pwd*
用户数据库的接口，与此类似。

35.4 termios --- POSIX 风格的 tty 控制

此模块提供了针对 tty I/O 控制的 POSIX 调用的接口。有关此类调用的完整描述，请参阅 *termios(3)* Unix 指南页。它仅在当安装时配置了支持 POSIX *termios* 风格的 tty I/O 控制的 Unix 版本上可用。

Availability: Unix.

此模块中的所有函数均接受一个文件描述符 *fd* 作为第一个参数。这可以是一个整数形式的文件描述符，例如 `sys.stdin.fileno()` 所返回的对象，或是一个*file object*，例如 `sys.stdin` 本身。

这个模块还定义了与此处所提供的函数一起使用的所有必要的常量；这些常量与它们在 C 中的对应常量同名。请参考你的系统文档了解有关如何使用这些终端控制接口的更多信息。

这个模块定义了以下函数：

`termios.`**`tcgetattr`**(*fd*)

对于文件描述符 *fd* 返回一个包含 tty 属性的列表，形式如下：`[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]`，其中 *cc* 为一个包含 tty 特殊字符的列表（每一项都是长度为 1 的字符串，索引号为 `VMIN` 和 `VTIME` 的项除外，这些字段如有定义则应为整数）。对旗标和速度以及 *cc* 数组中索引的解读必须使用在*termios* 模块中定义的符号常量来完成。

`termios.`**`tcsetattr`**(*fd*, *when*, *attributes*)

根据 *attributes* 为文件描述符 *fd* 设置 tty 的属性，它是一个类似于*tcgetattr()* 的返回值的列表。*when* 参数决定这些属性在何时被更改：

`termios.`**`TCSANOW`**

立即更改属性。

`termios.`**`TCSADRAIN`**

传输完所有队列输出后再更改属性。

`termios.`**`TCSAFLUSH`**

传输完所有队列输出并丢弃所有队列输入后再更改属性。

`termios.`**`tcsendbreak`**(*fd*, *duration*)

在文件描述符 *fd* 上发送一个中断。*duration* 为零表示发送时长为 0.25--0.5 秒的中断；*duration* 非零值的含义取决于具体系统。

`termios.`**`tcdrain`**(*fd*)

进入等待状态直到写入文件描述符 *fd* 的所有输出都传送完毕。

`termios.`**`tcflush`**(*fd*, *queue*)

在文件描述符 *fd* 上丢弃队列数据。*queue* 选择器指定哪个队列：`TCIFLUSH` 表示输入队列，`TCOFLUSH` 表示输出队列，或 `TCIOFLUSH` 表示两个队列同时。

`termios.`**`tcflow`**(*fd*, *action*)

在文件描述符 *fd* 上挂起一战恢复输入或输出。*action* 参数可以为 `TCOOFF` 表示挂起输出，`TCOON` 表示重启输出，`TCIOFF` 表示挂起输入，或 `TCION` 表示重启输入。

`termios.`**`tcgetwinsize`**(*fd*)

返回一个包含文件描述符 *fd* 的 tty 窗口大小的元组 `(ws_row, ws_col)`。需要 `termios.TIOCGWINSZ` 或 `termios.TIOCGSIZE`。

Added in version 3.11.

`termios.`**`tcsetwinsize`**(*fd*, *winsize*)

将文件描述符 *fd* 的 tty 窗口大小设置为 *winsize*，这是一个包含两项的元组 `(ws_row, ws_col)`，如*tcgetwinsize()* 所返回的一样。要求至少定义了 (`termios.TIOCGWINSZ`, `termios.TIOCSWINSZ`); (`termios.TIOCGSIZE`, `termios.TIOCSSIZE`) 对之一。

Added in version 3.11.

参见

模块*tty*
针对常用终端控制操作的便捷函数。

35.4.1 示例

这个函数可提示输入密码并且关闭回显。请注意其采取的技巧是使用一个单独的*tcgetattr()* 调用和一个 try ... finally 语句来确保旧的 tty 属性无论在何种情况下都会被原样保存:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

35.5 tty --- 终端控制函数

Source code: Lib/tty.py

---

tty 模块定义了将 tty 放入 cbreak 和 raw 模式的函数。

Availability: Unix.

因为它需要*termios* 模块，所以只能在 Unix 上运行。

tty 模块定义了以下函数:

tty.**cfmakeraw**(*mode*)

操作 tty 属性列表 *mode*，它是一个与*termios.tcgetattr()* 的返回值类似的列表，将其转换为原始模式 tty 的属性列表。

Added in version 3.12.

tty.**cfmakecbreak**(*mode*)

操作 tty 属性列表 *mode*，它是一个与*termios.tcgetattr()* 的返回值类似的列表，将其转换为 cbreak 模式的 tty 的属性列表。

这将清除 *mode* 中的 ECHO 和 ICANON 本地模式旗标并将最小输入设为 1 字节且无延迟。

Added in version 3.12.

在 3.12.2 版本发生变更: ICRNL 旗标将不再被清除。这与 Linux 和 macOS 的 stty cbreak 行为以及*setcbreak()* 在历史上所做的相匹配。

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)

将文件描述符 *fd* 的模式改为 raw。如果 *when* 被省略，它将默认为*termios.TCSAFLUSH*，并将被传给*termios.tcsetattr()*。*termios.tcgetattr()* 的返回值在将 *fd* 设为 raw 模式前会被保存；该值将被返回。

在 3.12 版本发生变更: 现在返回值就是原本的 tty 属性，而不是 None。

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)

将文件描述符 *fd* 的模式改为 cbreak。如果 *when* 被省略，它将默认为*termios.TCSAFLUSH*，并将被传给*termios.tcsetattr()*。*termios.tcgetattr()* 的返回值在将 *fd* 设为 cbreak 模式前会被保存；该值将被返回。

这将清除 ECHO 和 ICANON 本地模式旗标并将最小输入设为 1 字节且无延迟。

在 3.12 版本发生变更: 现在返回值就是原本的 tty 属性，而不是 None。

在 3.12.2 版本发生变更: ICRNL 旗标将不再被清除。这恢复了 Python 3.11 及更早版本的行为并与 Linux, macOS 和 BSD 在它们的 stty(1) 指南页对于 cbreak 模式的描述相匹配。

参见

模块 termios
低级终端控制接口。

35.6 pty --- 伪终端工具

源代码: Lib/pty.py

---

pty 模块定义了一些处理“伪终端”概念的操作：启动另一个进程并能以程序方式在其控制终端中进行读写。

Availability: Unix.

伪终端处理高度依赖于具体平台。此代码主要针对 Linux, FreeBSD 和 macOS 进行了测试（它应当也能在其他 POSIX 平台上工作，但是未经充分测试）。

pty 模块定义了下列函数:

pty.**fork**()
分叉。将子进程的控制终端连接到一个伪终端。返回值为 (pid, fd)。请注意子进程获得 *pid* 0 而 *fd* 为 *invalid*。父进程返回值为子进程的 *pid* 而 *fd* 为一个连接到子进程的控制终端（并同时连接到子进程的标准输入和输出）的文件描述符。

警告

在 macOS 上将此函数与高层级的系统 API 混用是不安全的，包括 urllib.request。

pty.**openpty**()
打开一个新的伪终端对，如果可能将使用 os.openpty()，或是针对通用 Unix 系统的模拟代码。返回一个文件描述符对 (master, slave)，分别表示主从两端。

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])
生成一个进程，并将其控制终端连接到当前进程的标准 io。这常被用来应对坚持要从控制终端读取数据的程序。在 pty 背后生成的进程预期最后将被终止，而且当它被终止时 *spawn* 将会返回。

将当前进程的 STDIN 拷贝到子进程并将从子进程接收的数据拷贝到当前进程的 STDOUT 的循环。如果当前进程的 STDIN 关闭则它不会向子进程发信号。

master_read 和 *stdin_read* 函数会被传入一个文件描述符供它们读取内容，并且它们总是应当返回一个字节串。为了强制 spawn 在子进程退出之前返回，应当返回一个空字节数组来提示文件的结束。

两个函数的默认实现在每次函数被调用时将读取并返回至多 1024 个字节。会向 *master_read* 回调传入伪终端的主文件描述符以从子进程读取输出，而向 *stdin_read* 传入文件描述符 0 以从父进程的标准输入读取数据。

从两个回调返回空字节串会被解读为文件结束 (EOF) 条件，在此之后回调将不再被调用。如果 *stdin_read* 发出 EOF 信号则控制终端就不能再与父进程或子进程进行通信。除非子进程将不带任何输入就退出，否则随后 *spawn* 将一直循环下去。如果 *master_read* 发出 EOF 信号则会有相同的行为结果（至少是在 Linux 上）。

从子进程中的 os.waitpid() 返回退出状态值。

os.waitstatus_to_exitcode() 可被用来将退出状态转换为退出码。

引发一个审计事件 pty.spawn 并附带参数 argv。

在 3.4 版本发生变更: *spawn()* 现在会从子进程的*os.waitpid()* 返回状态值。

35.6.1 示例

以下程序的作用类似于 Unix 命令 *script(1)*，它使用一个伪终端来记录一个"typescript" 里终端进程的所有输入和输出:

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.7 fcntl --- fcntl 和 ioctl 系统调用

本模块基于文件描述符来执行文件和 I/O 控制。它是 `fcntl()` 和 `ioctl()` Unix 例程的接口。请参阅 *fcntl(2)* 和 *ioctl(2)* Unix 手册页了解详情。

Availability: Unix, not WASI.

本模块的所有函数都接受文件描述符 *fd* 作为第一个参数。可以是一个整数形式的文件描述符，比如 `sys.stdin.fileno()` 的返回结果，或为*io.IOBase* 对象，比如 `sys.stdin` 提供一个*fileno()*，可返回一个真正的文件描述符。

在 3.3 版本发生变更: 本模块的操作以前触发的是*IOError*，现在则会触发*OSError*。

在 3.8 版本发生变更: `fcntl` 模块现在包含 `F_ADD_SEALS`, `F_GET_SEALS` 和 `F_SEAL_*` 常量用于*os.memfd_create()* 文件描述符的封包。

在 3.9 版本发生变更: 在 macOS 上，`fcntl` 模块暴露了 `F_GETPATH` 常量，它可从文件描述符获取文件的路径。在 Linux(>=3.15) 上，`fcntl` 模块暴露了 `F_OFD_GETLK`, `F_OFD_SETLK` 和 `F_OFD_SETLKW` 常量，它们将在处理打开文件描述锁时被使用。

在 3.10 版本发生变更: 在 Linux >= 2.6.11 中，`fcntl` 模块暴露了 `F_GETPIPE_SZ` 和 `F_SETPIPE_SZ` 常量，它们分别允许检查和修改管道的大小。

在 3.11 版本发生变更: 在 FreeBSD 上，fcntl 模块会暴露 F_DUP2FD 和 F_DUP2FD_CLOEXEC 常量，它们允许复制文件描述符，后者还额外设置了 FD_CLOEXEC 旗标。

在 3.12 版本发生变更: 在 Linux >= 4.5 上，*fcntl* 模块将公开 FICLONE 和 FICLONERANGE 常量，这允许在某些系统上（例如 btrfs, OCFS2, 和 XFS）通过将一个文件引用链接到另一个文件来共享某些数据。此行为通常被称为“写入时拷贝”。

在 3.13 版本发生变更: 在 Linux >= 2.6.32 上，fcntl 模块会暴露 F_GETOWN_EX, F_SETOWN_EX, F_OWNER_TID, F_OWNER_PID, F_OWNER_PGRP 常量，它们允许针对特定线程、进程或进程组的直接 I/O 可用性信号。在 Linux >= 4.13 上，fcntl 模块会暴露 F_GET_RW_HINT, F_SET_RW_HINT, F_GET_FILE_RW_HINT, F_SET_FILE_RW_HINT 和 RWH_WRITE_LIFE_* 常量，它们允许向内核通知有关在给定 inode 上或通过特定的打开文件描述符写入的相对预计生命期。在 Linux >= 5.1 和 NetBSD 上，fcntl 模块会暴露 F_SEAL_FUTURE_WRITE 常量供 F_ADD_SEALS 和 F_GET_SEALS 操作使用。在 FreeBSD 上，fcntl 模块会暴露 F_READAHEAD, F_ISUNIONSTACK 和 F_KINFO 常量。在 macOS 和 FreeBSD 上，fcntl 模块会暴露 F_RDAHEAD 常量。在 NetBSD 和 AIX 上，fcntl 模块会暴露 F_CLOSEM 常量。在 NetBSD 上，fcntl 模块会暴露 F_MAXFD 常量。在 macOS 和 NetBSD 上，fcntl 模块会暴露 F_GETNOSIGPIPE 和 F_SETNOSIGPIPE 常量。

这个模块定义了以下函数：

fcntl.**fcntl**(*fd*, *cmd*, *arg=0*, /)

在文件描述符 *fd* (也可以是提供 *fileno()* 方法的文件对象) 上执行操作 *cmd*。用作 *cmd* 的值依赖于具体操作系统，并且会是 *fcntl* 模块中的常量，使用与相应 C 头文件中所使用的相同名称。参数 *arg* 可以是一个整数值，*bytes* 对象或字符串。*arg* 的类型和大小必须与相应 C 文档所规定操作的参数的类型和大小相匹配。

当 *arg* 是一个整数时，该函数将返回 C fcntl() 调用的整数返回值。

当参数是 bytes 对象时，它代表一个二进制数组结构，例如由 *struct.pack()* 所创建的数据。字符串值将使用 UTF-8 编码格式编码为二进制数据。该二进制数据会被拷贝到一个缓冲区，其地址将被传给 C fcntl() 调用。成功调用后的返回值将是该缓冲区的内容所转换成的 *bytes* 对象。被返回对象的长度将与 *arg* 参数的长度相同。对象长度上限为 1024 字节。

如果 fcntl() 调用失败，将引发 *OSError*。

> **备注**
>
> 如果 *arg* 的类型和大小与相应操作的类型和大小不匹配（例如，如果预期传入一个指针却传入了一个整数，或者操作系统返回的缓冲区中的信息大于 1024 字节），这就很可能导致段错误或更微妙的数据损坏。

引发一个审计事件 fcntl.fcntl 并附带参数 fd, cmd, arg。

fcntl.**ioctl**(*fd*, *request*, *arg=0*, *mutate_flag=True*, /)

本函数与 *fcntl()* 函数相同，只是参数的处理更加复杂。

request 形参被限制为能被放入 32 或 64 个比特位的值，具体取决于所在的平台。在 *termios* 模块中还包含一些可被用作 *request* 参数的额外常量，其名称与相关 C 语言头文件中所使用的相同。

形参 *arg* 可以是一个整数，*bytes-like object* 或者字符串。*arg* 的类型和大小必须与对应 C 文档中规定的参数的类型和大小相匹配。

如果 *arg* 不支持读写缓冲区接口或者 *mutate_flag* 为假值，则其行为与 *fcntl()* 函数一样。

如果 *arg* 支持读写缓冲区接口 (就像 *bytearray*) 并且 *mutate_flag* 为（默认的）真值，那么缓冲区（实际上）会被传给下层的 ioctl() 系统调用，后者的返回代码则会回传给调用方 Python 对象，而缓冲区的新内容将反映 ioctl() 的动作。这里做了一点简化，因为如果给出的缓冲区长度小于 1024 字节则它会先被拷贝到一个长度为 1024 字节的静态缓冲区然后再传给 *ioctl()* 并把结果拷贝回给出的缓冲区。

如果 ioctl() 调用失败，将引发 *OSError* 异常。

> **备注**
>
> 如果 *arg* 的类型和大小与对应操作的参数的类型和大小不相匹配（例如，如果预期传入一个指针却传入了一个整数，或者由操作系统返回的缓冲区中的信息大于 1024 字节），这就很有可能导致段错误或更微妙的数据损坏。

举个例子：

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

引发一个审计事件 `fcntl.ioctl` 并附带参数 `fd`, `request`, `arg`。

`fcntl.`**`flock`**(*fd*, *operation*, */*)

在文件描述符 *fd* 上执行加锁操作 *operation* (也接受能提供*`fileno()`* 方法的文件对象)。详见 Unix 手册 *`flock(2)`*。(在某些系统中，此函数是用 `fcntl()` 模拟出来的。)

如果 `flock()` 调用失败，将引发*`OSError`* 异常。

引发一个审计事件 `fcntl.flock` 并附带参数 `fd`, `operation`。

`fcntl.`**`lockf`**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*, */*)

本质上是对*`fcntl()`* 加锁调用的封装。*fd* 是要加解锁的文件描述符（也接受能提供*`fileno()`* 方法的文件对象），*cmd* 是以下值之一：

`fcntl.`**`LOCK_UN`**

释放一个已存在的锁。

`fcntl.`**`LOCK_SH`**

获取一个共享的锁。

`fcntl.`**`LOCK_EX`**

获得一个独占的锁。

`fcntl.`**`LOCK_NB`**

与其他三个 `LOCK_*` 常量中的任何一个进行位或操作，使请求不阻塞。

如果使用了 `LOCK_NB` ，但无法获取锁，则*`OSError`* 将被引发，异常将被 *errno* 属性设置为*`EACCES`* 或*`EAGAIN`* （取决于操作系统；为便于移植，请检查这两个值）。至少在某些系统中，只有当文件描述符指向一个已打开供写入的文件时，才能使用:const:*!LOCK_EX* 。

len 是要锁定的字节数，*start* 是自 *whence* 开始锁定的字节偏移量，*whence* 与*`io.IOBase.seek()`* 的定义一样。

- `0` -- 相对于文件开头 (*`os.SEEK_SET`*)
- `1` -- 相对于当前缓冲区位置 (*`os.SEEK_CUR`*)
- `2` -- 相对于文件末尾 (*`os.SEEK_END`*)

start 的默认值为 0，表示从文件起始位置开始。*len* 的默认值是 0，表示加锁至文件末尾。*whence* 的默认值也是 0。

引发一个审计事件 `fcntl.lockf` 并附带参数 `fd`, `cmd`, `len`, `start`, `whence`。

示例（都是运行于符合 SVR4 的系统）：

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

注意，在第一个例子中，返回值变量 *rv* 将存有整数；在第二个例子中，该变量中将存有一个*bytes* 对象。*lockdata* 变量的结构布局视系统而定——因此采用*flock()* 调用可能会更好。

参见

模块*os*
如果加锁旗标*O_SHLOCK* 和*O_EXLOCK* 存在于*os* 模块中（仅 BSD 专属），则*os.open()* 函数提供了对*lockf()* 和*flock()* 函数的替代。

35.8 resource --- 资源使用信息

该模块提供了测量和控制程序所利用的系统资源的基本机制。

Availability: Unix, not WASI.

符号常量被用来指定特定的系统资源，并要求获得关于当前进程或其子进程的使用信息。

当系统调用失败时，会触发一个*OSError* 。

exception resource.**error**
一个被弃用的*OSError* 的别名。

在 3.3 版本发生变更: 根据 **PEP 3151**，这个类是*OSError* 的别名。

35.8.1 资源限制

资源的使用可以通过下面描述的*setrlimit()* 函数来限制。每个资源都被一对限制所控制：一个软限制和一个硬限制。软限制是当前的限制，并且可以由一个进程随着时间的推移而降低或提高。软限制永远不能超过硬限制。硬限制可以降低到大于软限制的任何数值，但不能提高。(只有拥有超级用户有效 UID 的进程才能提高硬限制。)

可以被限制的具体资源取决于系统。它们在 man *getrlimit(2)* 中描述。下面列出的资源在底层操作系统支持的情况下被支持；那些不能被操作系统检查或控制的资源在本模块中没有为这些平台定义。

resource.**RLIM_INFINITY**
用来表示无限资源的极限的常数。

resource.**getrlimit**(*resource*)
返回一个包含 *resource* 当前软限制和硬限制的元组。如果指定了一个无效的资源，则触发*ValueError* ，如果底层系统调用意外失败，则引发*error* 。

resource.**setrlimit**(*resource*, *limits*)
设置 *resource* 的新的消耗极限。参数 *limits* 必须是一个由两个整数组成的元组 (soft, hard) ，描述了新的限制。*RLIM_INFINITY* 的值可以用来请求一个无限的限制。

如果指定了一个无效的资源，如果新的软限制超过了硬限制，或者如果一个进程试图提高它的硬限制，将触发*ValueError* 。当资源的硬限制或系统限制不是无限时，指定一个*RLIM_INFINITY* 的限制将导致*ValueError* 。一个有效 UID 为超级用户的进程可以请求任何有效的限制值，包括无限，但如果请求的限制超过了系统规定的限制，则仍然会产生*ValueError* 。

如果底层系统调用失败，setrlimit 也可能触发*error* 。

VxWorks 只支持设置*RLIMIT_NOFILE*。

引发一个审计事件 `resource.setrlimit` 并附带参数 `resource`, `limits`。

`resource.`**`prlimit`**(*pid*, *resource*[, *limits*])

将*setrlimit()*和*getrlimit()*合并为一个函数，支持获取和设置任意进程的资源限制。如果 *pid* 为 0，那么该调用适用于当前进程。*resource* 和 *limits* 的含义与*setrlimit()*相同，只是 *limits* 是可选的。

当 *limits* 没有给出时，该函数返回进程 *pid* 的 *resource* 限制。当 *limits* 被给定时，进程的 *resource* 限制被设置，并返回以前的资源限制。

当 *pid* 找不到时，触发*ProcessLookupError*；当用户没有进程的 `CAP_SYS_RESOURCE` 时，触发*PermissionError*。

引发一个审计事件 `resource.prlimit` 并附带参数 `pid`, `resource`, `limits`。

Availability: Linux >= 2.6.36 with glibc >= 2.13.

Added in version 3.4.

这些符号定义了资源的消耗可以通过下面描述的*setrlimit()*和*getrlimit()*函数来控制。这些符号的值正是 C 程序所使用的常数。

Unix man 页面 *getrlimit(2)* 列出了可用的资源。注意，并非所有系统都使用相同的符号或相同的值来表示相同的资源。本模块并不试图掩盖平台的差异——没有为某一平台定义的符号在该平台上将无法从本模块中获得。

`resource.`**`RLIMIT_CORE`**

当前进程可以创建的核心文件的最大大小（以字节为单位）。如果需要更大的核心文件来包含整个进程的镜像，这可能会导致创建一个部分核心文件。

`resource.`**`RLIMIT_CPU`**

一个进程可以使用的最大处理器时间（以秒为单位）。如果超过了这个限制，一个 `SIGXCPU` 信号将被发送给进程。（参见*signal*模块文档，了解如何捕捉这个信号并做一些有用的事情，例如，将打开的文件刷新到磁盘上）。

`resource.`**`RLIMIT_FSIZE`**

进程可能创建的文件的最大大小。

`resource.`**`RLIMIT_DATA`**

进程的堆的最大大小（以字节为单位）。

`resource.`**`RLIMIT_STACK`**

当前进程的调用堆栈的最大大小（字节）。这只影响到多线程进程中主线程的堆栈。

`resource.`**`RLIMIT_RSS`**

应该提供给进程的最大常驻内存大小。

`resource.`**`RLIMIT_NPROC`**

当前进程可能创建的最大进程数。

`resource.`**`RLIMIT_NOFILE`**

当前进程打开的文件描述符的最大数量。

`resource.`**`RLIMIT_OFILE`**

BSD 对*RLIMIT_NOFILE*的命名。

`resource.`**`RLIMIT_MEMLOCK`**

可能被锁定在内存中的最大地址空间。

`resource.`**`RLIMIT_VMEM`**

进程可能占用的最大映射内存区域。

Availability: FreeBSD >= 11.

resource.**RLIMIT_AS**
进程可能占用的地址空间的最大区域（以字节为单位）。

resource.**RLIMIT_MSGQUEUE**
可分配给 POSIX 消息队列的字节数。

Availability: Linux >= 2.6.8.

Added in version 3.4.

resource.**RLIMIT_NICE**
进程的 Nice 级别的上限（计算为 20 - rlim_cur ）。

Availability: Linux >= 2.6.12.

Added in version 3.4.

resource.**RLIMIT_RTPRIO**
实时优先级的上限。

Availability: Linux >= 2.6.12.

Added in version 3.4.

resource.**RLIMIT_RTTIME**
在实时调度下，一个进程在不进行阻塞性系统调用的情况下，可以花费的 CPU 时间限制（以微秒计）。

Availability: Linux >= 2.6.25.

Added in version 3.4.

resource.**RLIMIT_SIGPENDING**
进程可能排队的信号数量。

Availability: Linux >= 2.6.8.

Added in version 3.4.

resource.**RLIMIT_SBSIZE**
这个用户使用的套接字缓冲区的最大大小（字节数）。这限制了这个用户在任何时候都可以持有的网络内存数量，因此也限制了 mbufs 的数量。

Availability: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_SWAP**
这个用户 ID 的所有进程可能保留或使用的交换空间的大小上限（以字节数表示）。此限制只有在 vm.overcommit sysctl 的 1 号比特位被设置时才会生效。请参阅 tuning(7) 获取该 sysctl 的完整描述。

Availability: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_NPTS**
该用户 ID 创建的伪终端的最大数量。

Availability: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_KQUEUES**
这个用户 ID 被允许创建的最大 kqueue 数量。

Availability: FreeBSD >= 11.

Added in version 3.10.

35.8.2 资源用量

这些函数被用来检索资源使用信息。

resource.**getrusage**(*who*)

此函数返回一个描述当前进程或其子进程所消耗的资源的对象，它由 *who* 形参指定。*who* 形参应当使用下面介绍的 RUSAGE_* 常量之一来指定。

一个简单的示例：

```
from resource import *
import time

# 非 CPU 密集型任务
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# CPU 密集型任务
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

返回值的字段分别描述了某一特定系统资源的使用情况，例如，在用户模式下运行的时间或进程从主内存中换出的次数。有些值取决于内部的时钟周期，例如进程使用的内存量。

为了向后兼容，返回值也可以作为一个 16 个元素的元组来访问。

返回值中的 ru_utime 和 ru_stime 字段是浮点值，分别代表在用户模式下执行的时间和在系统模式下执行的时间。其余的值是整数。关于这些值的详细信息，请查阅 *getrusage(2)* man page 。这里提供一个简短的摘要。

索引	字段	资源
0	ru_utime	用户模式下的时间（浮点数秒）
1	ru_stime	系统模式下的时间（浮点数秒）
2	ru_maxrss	最大的常驻内存大小
3	ru_ixrss	共享内存大小
4	ru_idrss	未共享的内存大小
5	ru_isrss	未共享的堆栈大小
6	ru_minflt	不需要 I/O 的页面故障数
7	ru_majflt	需要 I/O 的页面故障数
8	ru_nswap	swap out 的数量
9	ru_inblock	块输入操作数
10	ru_oublock	块输出操作数
11	ru_msgsnd	发送消息数
12	ru_msgrcv	收到消息数
13	ru_nsignals	收到信号数
14	ru_nvcsw	主动上下文切换
15	ru_nivcsw	被动上下文切换

如果指定了一个无效的 *who* 参数，这个函数将触发一个 *ValueError* 。在特殊情况下，它也可能触发 *error* 异常。

resource.**getpagesize**()

返回一个系统页面的字节数。(这不需要和硬件页的大小相同)。

下面的 RUSAGE_* 符号将被传给 *getrusage()* 函数以指定应该为哪些进程提供信息。

resource.**RUSAGE_SELF**

传递给 *getrusage()* 以请求调用进程消耗的资源，这是进程中所有线程使用的资源总和。

resource.**RUSAGE_CHILDREN**

传递给*getrusage()* 以请求被终止和等待的调用进程的子进程所消耗的资源。

resource.**RUSAGE_BOTH**

传递给*getrusage()* 以请求当前进程和子进程所消耗的资源。并非所有系统都能使用。

resource.**RUSAGE_THREAD**

传递给*getrusage()* 以请求当前线程所消耗的资源。并非所有系统都能使用。

Added in version 3.2.

35.9 syslog --- Unix syslog 库例程

此模块提供一个接口到 Unix syslog 日常库. 参考 Unix 手册页关于 syslog 设施的详细描述.

Availability: Unix, not WASI, not iOS.

此模块包装了系统 syslog 例程族。一个能与 syslog 服务器对话的纯 Python 库则在*logging.handlers* 模块中以*SysLogHandler* 类的形式提供。

这个模块定义了以下函数:

syslog.**syslog**(*message*)

syslog.**syslog**(*priority*, *message*)

将字符串 *message* 发送到系统日志记录器。如有必要会添加末尾换行符。每条消息都带有一个由 *facility* 和 *level* 组成的优先级标价签。可选的 *priority* 参数默认值为*LOG_INFO*，它确定消息的优先级。如果未在 *priority* 中使用逻辑或 (LOG_INFO | LOG_USER) 对 facility 进行编码，则会使用在*openlog()* 调用中所给定的值。

如果*openlog()* 未在对*syslog()* 的调用之前被调用，则将不带参数地调用*openlog()*。

引发一个审计事件 syslog.syslog 并附带参数 priority, message。

在 3.2 版本发生变更: 在之前的版本中，如果*openlog()* 未在对*syslog()* 的调用之前被调用则它将不会被自动调用，而是由 syslog 实现来负责调用 openlog()。

在 3.12 版本发生变更: 此函数在子解释器中受到限制。(该限制只影响在多解释器中运行的代码因而与大多数用户无关。) *openlog()* 必须在子解释器使用*syslog()* 之前在主解释器中被调用。否则它将引发*RuntimeError*。

syslog.**openlog**([*ident*[, *logoption*[, *facility*]]])

后续*syslog()* 调用的日志选项可以通过调用*openlog()* 来设置。如果日志当前未打开则*syslog()* 将不带参数地调用*openlog()*。

可选的 *ident* 关键字参数是在每条消息前添加的字符串，默认为 sys.argv[0] 去除打头的路径部分。可选的 *logoption* 关键字参数（默认为 0）是一个位字段 -- 请参见下文了解可能的组合值。可选的 *facility* 关键字参数 (默认为*LOG_USER*) 为没有显式编码 facility 的消息设置默认的 facility。

引发一个审计事件 syslog.openlog 并附带参数 ident, logoption, facility。

在 3.2 版本发生变更: 在之前的版本中，不允许使用关键字参数，并且要求必须有 *ident*。

在 3.12 版本发生变更: 此函数在子解释器中受到限制。(该限制只影响在多解释器中运行的代码因而与大多数用户无关。) 此函数只能在主解释器中被调用。如果在子解释器中被调用它将引发*RuntimeError*。

syslog.**closelog**()

重置日志模块值并且调用系统库 closelog().

这使得此模块在初始导入时行为固定。例如，*openlog()* 将在首次调用*syslog()* 时被调用（如果*openlog()* 还未被调用过），并且 *ident* 和其他*openlog()* 形参会被重置为默认值。

引发一个不带参数的审计事件 syslog.closelog。

在 3.12 版本发生变更: 此函数在子解释器中受到限制。（该限制只影响在多解释器中运行的代码因而与大多数用户无关。）此函数只能在主解释器中被调用。如果在子解释器中被调用它将引发*RuntimeError*。

syslog.**setlogmask**(*maskpri*)

将优先级掩码设为 *maskpri* 并返回之前的掩码值。调用*syslog()* 并附带未在 *maskpri* 中设置的优先级将会被忽略。默认设置为记录所有优先级。函数 LOG_MASK(pri) 可计算单个优先级 *pri* 的掩码。函数 LOG_UPTO(pri) 可计算包括 *pri* 在内的所有优先级的掩码。

引发一个审计事件 syslog.setlogmask 并附带参数 maskpri。

此模块定义了一下常量:

syslog.**LOG_EMERG**

syslog.**LOG_ALERT**

syslog.**LOG_CRIT**

syslog.**LOG_ERR**

syslog.**LOG_WARNING**

syslog.**LOG_NOTICE**

syslog.**LOG_INFO**

syslog.**LOG_DEBUG**

优先级别（从高到低）。

syslog.**LOG_AUTH**

syslog.**LOG_AUTHPRIV**

syslog.**LOG_CRON**

syslog.**LOG_DAEMON**

syslog.**LOG_FTP**

syslog.**LOG_INSTALL**

syslog.**LOG_KERN**

syslog.**LOG_LAUNCHD**

syslog.**LOG_LPR**

syslog.**LOG_MAIL**

syslog.**LOG_NETINFO**

syslog.**LOG_NEWS**

syslog.**LOG_RAS**

syslog.**LOG_REMOTEAUTH**

syslog.**LOG_SYSLOG**

syslog.**LOG_USER**

syslog.**LOG_UUCP**

syslog.**LOG_LOCAL0**

syslog.**LOG_LOCAL1**

syslog.**LOG_LOCAL2**

syslog.**LOG_LOCAL3**

syslog.**LOG_LOCAL4**

syslog.**LOG_LOCAL5**

syslog.**LOG_LOCAL6**

syslog.**LOG_LOCAL7**

功能项，根据在 <syslog.h> 中*LOG_AUTHPRIV*, *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL* 和*LOG_RAS* 的可用性确定。

在 3.13 版本发生变更: 增加了*LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL*, *LOG_RAS* 和*LOG_LAUNCHD*。

syslog.**LOG_PID**

syslog.**LOG_CONS**

syslog.**LOG_NDELAY**

syslog.**LOG_ODELAY**

syslog.**LOG_NOWAIT**

syslog.**LOG_PERROR**

日志选项，根据在 <syslog.h> 中*LOG_ODELAY*, *LOG_NOWAIT* 和*LOG_PERROR* 的可用性确定。

35.9.1 例子

简单示例

一个简单的示例集:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

一个设置多种日志选项的示例，其中有在日志消息中包含进程 ID，以及将消息写入用于邮件日志记录的目标设施等:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CHAPTER 36

模块命令行界面（CLI）

下列模块具有命令行界面。

- *ast*
- *asyncio*
- `base64`
- *calendar*
- `code`
- *compileall*
- `cProfile`: 参见*profile*
- *difflib*
- *dis*
- *doctest*
- `encodings.rot_13`
- `ensurepip`
- `filecmp`
- `fileinput`
- `ftplib`
- *gzip*
- *http.server*
- `idlelib`
- *inspect*
- *json.tool*
- `mimetypes`
- `pdb`
- `pickle`

- *pickletools*
- `platform`
- `poplib`
- *profile*
- `pstats`
- *py_compile*
- `pyclbr`
- `pydoc`
- `quopri`
- *random*
- `runpy`
- *site*
- *sqlite3*
- 符号表
- *sysconfig*
- `tabnanny`
- *tarfile*
- `this`
- *timeit*
- *tokenize*
- *trace*
- `turtledemo`
- *unittest*
- *uuid*
- `venv`
- `webbrowser`
- *zipapp*
- *zipfile*

另请参阅 the Python 命令行界面。

CHAPTER 37

被取代的模块

本章介绍的模块在大部分场合下都已被其他模块所取代，它们被保留主要是为了能够向下兼容。

某些模块出现在本章中可能是因为它们只覆盖了某个问题空间的有限子集，而更为普遍应用的解决方案存在于标准库的其他地方 (例如，*getopt* 覆盖了非常特定的" 在 Python 中模拟 C `getopt()` API" 任务，而不是如*optparse* 和*argparse* 所提供的更为宽泛的命令行选项解析和参数解析功能)。

另外，某些模块出现在本章中也可能是因为它们已被弃用，并将在未来的发布版中被移除，或者它们已被*soft deprecated* 并且不再鼓励在新项目中使用。随着多个过时模块根据 **PEP 594** 被移除，目前已不存在属于此类情况的模块。

37.1 `getopt` --- C 风格的命令行选项解析器

源代码: Lib/getopt.py

> **备注**
>
> 该模块被认为具有完备的特性。在*optparse* 模块中提供了一个针对此 API 的更为声明式和可扩展的替代物。对命令行形参处理进一步的功能增强或是作为 PyPI 上的第三方模块，或是作为*argparse* 模块中的特性被提供。

此模块可协助脚本解析 `sys.argv` 中的命令行参数。它支持与 Unix `getopt()` 函数相同的惯例 (包括形式为'-' 和'--' 的参数的特殊含义)。也可以通过可选的第三个参数来使用类似于 GNU 软件所支持形式的长选项。

不熟悉 Unix `getopt()` 函数的用户应当考虑改用*argparse* 模块。熟悉 Unix `getopt()` 函数，但希望在获得等价的行为同时使用更少的代码又具有更好的帮助和错误信息的用户应当考虑使用*optparse* 模块。请参阅*选择参数解析库* 了解更多细节。

此模块提供了两个函数和一个异常:

`getopt.`**`getopt`**(*args*, *shortopts*, *longopts=[]*)

解析命令行选项与形参列表。*args* 是要解析的参数列表，不包含最开头的对正在运行的程序的引用。通常，这意味着 `sys.argv[1:]`。*shortopts* 是脚本要识别的选项字母，带有要求后缀一个冒号 (`':'`；即与 Unix `getopt()` 所用的格式相同) 的选项。

> **备注**
>
> 与 GNU `getopt()` 不同，在非选项参数之后，所有后续参数都会被视为非选项。这类似于非 GNU Unix 系统的运作方式。

如果指定了 *longopts*，则必须为一个由应当被支持的长选项名称组成的列表。开头的 `'--'` 字符不应被包括在选项名称中。要求参数的长选项后应当带一个等号 (`'='`)。可选参数不被支持。如果想仅接受长选项，则 *shortopts* 应为一个空字符串。命令行中的长选项只要提供了恰好能匹配可接受选项之一的选项名称前缀即可被识别。举例来说，如果 *longopts* 为 `['foo', 'frob']`，则选项 `--fo` 将匹配为 `--foo`，但 `--f` 将不能得到唯一匹配，因此将引发 *GetoptError*。

返回值由两个元素组成：第一个是 `(option, value)` 对的列表；第二个是在去除该选项列表后余下的程序参数列表（这也就是 *args* 的尾部切片）。每个被返回的选项与值对的第一个元素是选项，短选项前缀一个连字符 (例如 `'-x'`)，长选项则前缀两个连字符 (例如 `'--long-option'`)，第二个元素是选项参数，如果选项不带参数则为空字符串。列表中选项的排列顺序与它们被解析的顺序相同，因此允许多次出现。长选项与短选项可以混用。

`getopt.`**`gnu_getopt`**(*args*, *shortopts*, *longopts=[]*)

此函数与 *getopt()* 类似，区别在于它默认使用 GNU 风格的扫描模式。这意味着选项和非选项参数可能会混在一起。*getopt()* 函数将在遇到非选项参数时立即停止处理选项。

如果选项字符串的第一个字符为 `'+'`，或者如果设置了环境变量 `POSIXLY_CORRECT`，则选项处理会在遇到非选项参数时立即停止。

exception `getopt.`**`GetoptError`**

当参数列表中出现不可识别的选项或当一个需要参数的选项未带参数时将引发此异常。此异常的参数是一个指明错误原因的字符串。对于长选项，将一个参数传给不需要参数的选项也将导致此异常被引发。`msg` 和 `opt` 属性将给出错误消息和关联的选项；如果没有关联到此异常的特定选项，则 `opt` 将为空字符串。

exception `getopt.`**`error`**

GetoptError 的别名；用于向后兼容。

一个仅使用 Unix 风格选项的例子:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

使用长选项名也同样容易:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

在脚本中，典型的用法是这样的:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # 打印帮助信息并退出。
        print(err)  # 将打印 "option -a not recognized" 之类的消息
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    process(args, output=output, verbose=verbose)

if __name__ == "__main__":
    main()
```

请注意可以通过使用*optparse* 模块以更少的代码并附带更清晰的帮助和错误消息生成等价的命令行界面：

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

对这种情况也可以通过使用*argparse* 模块来生成大致等价的命令行界面：

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

请参阅选择参数解析库 了解有关此代码的 argparse 版本与 optparse (和 getopt) 版本间行为差异的详情。

参见

模块*optparse*
声明式命令行选项解析。

模块*argparse*
更有针对性的命令行选项和参数解析库。

CHAPTER 38

Removed Modules

The modules described in this chapter have been removed from the Python standard library. They are documented here to help people find replacements.

38.1 `aifc` --- Read and write AIFF and AIFC files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `aifc` module was Python 3.12.

38.2 `asynchat` --- Asynchronous socket command/response handler

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594**.

Applications should use the *`asyncio`* module instead.

The last version of Python that provided the `asynchat` module was Python 3.11.

38.3 `asyncore` --- Asynchronous socket handler

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594**.

Applications should use the *`asyncio`* module instead.

The last version of Python that provided the `asyncore` module was Python 3.11.

38.4 audioop --- Manipulate raw audio data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `audioop` module was Python 3.12.

38.5 cgi --- Common Gateway Interface support

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

A fork of the module on PyPI can be used instead: legacy-cgi. This is a copy of the cgi module, no longer maintained or supported by the core Python team.

The last version of Python that provided the `cgi` module was Python 3.12.

38.6 cgitb --- Traceback manager for CGI scripts

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

A fork of the module on PyPI can now be used instead: legacy-cgi. This is a copy of the cgi module, no longer maintained or supported by the core Python team.

The last version of Python that provided the `cgitb` module was Python 3.12.

38.7 chunk --- Read IFF chunked data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `chunk` module was Python 3.12.

38.8 crypt --- Function to check Unix passwords

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Applications can use the *hashlib* module from the standard library. Other possible replacements are third-party libraries from PyPI: legacycrypt, bcrypt, argon2-cffi, or passlib. These are not supported or maintained by the Python core team.

The last version of Python that provided the `crypt` module was Python 3.12.

38.9 distutils --- Building and installing Python modules

Deprecated since version 3.10, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.10. The removal was decided in **PEP 632**, which has migration advice.

The last version of Python that provided the `distutils` module was Python 3.11.

38.10 `imghdr` --- Determine the type of an image

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Possible replacements are third-party libraries from PyPI: filetype, puremagic, or python-magic. These are not supported or maintained by the Python core team.

The last version of Python that provided the `imghdr` module was Python 3.12.

38.11 `imp` --- Access the import internals

Deprecated since version 3.4, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.4.

The removal notice includes guidance for migrating code from `imp` to *`importlib`*.

The last version of Python that provided the `imp` module was Python 3.11.

38.12 `mailcap` --- Mailcap file handling

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `mailcap` module was Python 3.12.

38.13 `msilib` --- Read and write Microsoft Installer files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `msilib` module was Python 3.12.

38.14 `nis` --- Interface to Sun's NIS (Yellow Pages)

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `nis` module was Python 3.12.

38.15 `nntplib` --- NNTP protocol client

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `nntplib` module was Python 3.12.

38.16 ossaudiodev --- Access to OSS-compatible audio devices

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `ossaudiodev` module was Python 3.12.

38.17 pipes --- Interface to shell pipelines

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Applications should use the *subprocess* module instead.

The last version of Python that provided the `pipes` module was Python 3.12.

38.18 smtpd --- SMTP Server

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594**.

A possible replacement is the third-party aiosmtpd library. This library is not maintained or supported by the Python core team.

The last version of Python that provided the `smtpd` module was Python 3.11.

38.19 sndhdr --- Determine type of sound file

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Possible replacements are third-party modules from PyPI: filetype, puremagic, or python-magic. These are not supported or maintained by the Python core team.

The last version of Python that provided the `sndhdr` module was Python 3.12.

38.20 spwd --- The shadow password database

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

A possible replacement is the third-party library python-pam. This library is not supported or maintained by the Python core team.

The last version of Python that provided the `spwd` module was Python 3.12.

38.21 sunau --- Read and write Sun AU files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `sunau` module was Python 3.12.

38.22 `telnetlib` --- Telnet client

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

Possible replacements are third-party libraries from PyPI: telnetlib3 or Exscript. These are not supported or maintained by the Python core team.

The last version of Python that provided the `telnetlib` module was Python 3.12.

38.23 `uu` --- Encode and decode uuencode files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `uu` module was Python 3.12.

38.24 `xdrlib` --- Encode and decode XDR data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594**.

The last version of Python that provided the `xdrlib` module was Python 3.12.

CHAPTER 39

安全考量

下列模块具有专门的安全事项:

- *base64*: *base64* 安全事项，参见 **RFC 4648**
- *hashlib*: 所有构造器都接受一个*"usedforsecurity"* 仅限关键字参数以停用已知的不安全和已封禁的算法
- *http.server* 不适合生产用途，只实现了基本的安全检查。请参阅安全性考量。
- *logging*: 日志记录配置使用了 *eval()*
- *multiprocessing*: *Connection.recv()* 使用了 *pickle*
- *pickle*: 在 *pickle* 中限制全局变量
- *random* 不应当被用于安全目的，而应改用*secrets*
- *shelve*: *shelve* 是基于 *pickle* 的因此不适用于处理不受信任的源
- *ssl*: *SSL/TLS* 安全事项
- *subprocess*: 子进程安全事项
- *tempfile*: *mktemp* 由于存在竞争条件缺陷已被弃用
- *xml*: *XML* 安全缺陷
- *zipfile*: 恶意处理的*.zip* 文件可能导致硬盘空间耗尽

`-I` 命令行选项可被用来在隔离模式下运行 Python。当它无法使用时，可以使用 `-P` 选项或 `PYTHONSAFEPATH` 环境变量以避免在*sys.path* 中预置一个潜在的不安全路径，如当前目录、脚本的目录或一个空字符串。

APPENDIX A

术语对照表

>>>
: *interactive* shell 中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

...
: 具有以下含义：

  - *interactive* shell 中输入特殊代码时默认的 Python 提示符，特殊代码包括缩进的代码块，左右成对分隔符（圆括号、方括号、花括号或三重引号等）之内，或是在指定一个装饰器之后。
  - `Ellipsis` 内置常量。

abstract base class -- 抽象基类
: 抽象基类简称 ABC，是对*duck-typing* 的补充，它提供了一种定义接口的新方式，相比之下其他技巧例如`hasattr()` 显得过于笨拙或有微妙错误（例如使用 魔术方法）。ABC 引入了虚拟子类，这种类并非继承自其他类，但却仍能被`isinstance()` 和`issubclass()` 所认可；详见`abc` 模块文档。Python 自带许多内置的 ABC 用于实现数据结构（在`collections.abc` 模块中）、数字（在`numbers` 模块中）、流（在`io` 模块中）、导入查找器和加载器（在`importlib.abc` 模块中）。你可以使用`abc` 模块来创建自己的 ABC。

annotation -- 标注
: 关联到某个变量、类属性、函数形参或返回值的标签，被约定作为类型注解 来使用。

  局部变量的标注在运行时不可访问，但全局变量、类属性和函数的标注会分别存放模块、类和函数的 `__annotations__` 特殊属性中。

  参见*variable annotation*, *function annotation*, **PEP 484** 和 **PEP 526**，对此功能均有介绍。另请参见 annotations-howto 了解使用标注的最佳实践。

argument -- 参数
: 在调用函数时传给*function* （或*method* ）的值。参数分为两种：

  - 关键字参数: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对`complex()` 的调用中均属于关键字参数:

    ```
    complex(real=3, imag=5)
    complex(**{'real': 3, 'imag': 5})
    ```

  - 位置参数: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 `*` 的*iterable* 里的元素被传入。举例来说，`3` 和 `5` 在以下调用中均属于位置参数:

```
complex(3, 5)
complex(*(3, 5))
```

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见 calls 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

另参见*parameter* 术语表条目，常见问题中 参数与形参的区别以及 **PEP 362**。

asynchronous context manager -- 异步上下文管理器
此种对象通过定义 `__aenter__()` 和 `__aexit__()` 方法来对 `async with` 语句中的环境进行控制。由 **PEP 492** 引入。

asynchronous generator -- 异步生成器
返回值为*asynchronous generator iterator* 的函数。它与使用 `async def` 定义的协程函数很相似，不同之处在于它包含 `yield` 表达式以产生一系列可在 `async for` 循环中使用的值。

此术语通常是指异步生成器函数，但在某些情况下则可能是指 异步生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

一个异步生成器函数可能包含 `await` 表达式或者 `async for` 以及 `async with` 语句。

asynchronous generator iterator -- 异步生成器迭代器
asynchronous generator 函数所创建的对象。

此对象属于*asynchronous iterator*，当使用 `__anext__()` 方法调用时会返回一个可等待对象来执行异步生成器函数的函数体直到下一个 `yield` 表达式。

每个 `yield` 会临时暂停处理过程，记住执行状态（包括局部变量和挂起的 try 语句）。当 异步生成器迭代器通过 `__anext__()` 所返回的另一个可等待对象有效地恢复时，它会从离开位置恢复处理过程。参见 **PEP 492** 和 **PEP 525**。

asynchronous iterable -- 异步可迭代对象
一个可以在 `async for` 语句中使用的对象。必须通过它的 `__aiter__()` 方法返回一个*asynchronous iterator*。由 **PEP 492** 引入。

asynchronous iterator -- 异步迭代器
一个实现了 `__aiter__()` 和 `__anext__()` 方法的对象。`__anext__()` 必须返回一个*awaitable* 对象。`async for` 会处理异步迭代器的 `__anext__()` 方法所返回的可等待对象直到其引发一个*`StopAsyncIteration`* 异常。由 **PEP 492** 引入。

attribute -- 属性
关联到一个对象的值，通常使用点号表达式按名称来引用。举例来说，如果对象 *o* 具有属性 *a* 则可以用 *o.a* 来引用它。

如果对象允许，将未被定义为 identifiers 的非标识名称用作一个对象的属性也是可以的，例如使用*`setattr()`*。这样的属性将无法使用点号表达式来访问，而是必须通过*`getattr()`* 来获取。

awaitable -- 可等待对象
一个可在 `await` 表达式中使用的对象。可以是*coroutine* 或是具有 `__await__()` 方法的对象。参见 **PEP 492**。

BDFL
“终身仁慈独裁者”的英文缩写，即 Guido van Rossum，Python 的创造者。

binary file -- 二进制文件
file object 能够读写*字节型对象*。二进制文件的例子包括以二进制模式 (`'rb'`, `'wb'` 或 `'rb+'`) 打开的文件、*`sys.stdin.buffer`*、*`sys.stdout.buffer`* 以及*`io.BytesIO`* 和*`gzip.GzipFile`* 的实例。

另请参见*text file* 了解能够读写*`str`* 对象的文件对象。

borrowed reference -- 借入引用
在 Python 的 C API 中，借用引用是指一种对象引用，使用该对象的代码并不持有该引用。如果对象被销毁则它就会变成一个悬空指针。例如，垃圾回收器可以移除对象的最后一个*strong reference* 来销毁它。

推荐在*borrowed reference* 上调用 `Py_INCREF()` 以将其原地转换为*strong reference*，除非是当该对象无法在借入引用的最后一次使用之前被销毁。`Py_NewRef()` 函数可以被用来创建一个新的*strong reference*。

bytes-like object -- 字节型对象
支持 bufferobjects 并且能导出 C-*contiguous* 缓冲的对象。这包括所有`bytes`、`bytearray` 和`array.array` 对象，以及许多普通`memoryview` 对象。字节型对象可在多种二进制数据操作中使用；这些操作包括压缩、保存为二进制文件以及通过套接字发送等。

某些操作需要可变的二进制数据。这种对象在文档中常被称为“可读写字节类对象”。可变缓冲对象的例子包括`bytearray` 以及`bytearray` 的`memoryview`。其他操作要求二进制数据存放于不可变对象 (” 只读字节类对象”)；这种对象的例子包括`bytes` 以及`bytes` 对象的`memoryview`。

bytecode -- 字节码
Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字节码）。这种” 中间语言” 运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在*dis 模块* 的文档中查看。

callable -- 可调用对象
可调用对象就是可以执行调用运算的对象，并可能附带一组参数 (参见*argument*)，使用以下语法:

```
callable(argument1, argument2, argumentN)
```

function，还可扩展到*method* 等，就属于可调用对象。实现了 `__call__()` 方法的类的实例也属于可调用对象。

callback -- 回调
一个作为参数被传入以用以在未来的某个时刻被调用的子例程函数。

class -- 类
用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

class variable -- 类变量
在类中定义的变量，并且仅限在类的层级上修改 (而不是在类的实例中修改)。

closure variable -- 闭包变量
引用自*nested scope* 的*free variable*，它是在外层作用域中定义而不是在运行时自全局或内置命名空间中取得。可能使用 `nonlocal` 关键字显式地定义以允许写入访问，或者如果变量仅供读取则只需隐式地定义。

例如，在以下代码的 `inner` 函数中，`x` 和 `print` 均为*自由变量*，但只有 `x` 属于 *闭包变量*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

由于 `codeobject.co_freevars` 属性的存在（虽然名称如此，但其仅包括闭包变量名称而非列出所有被引用的自由变量），更一般化的术语*free variable* 有时在即便本意是专指闭包变量时也会被使用。

complex number -- 复数
对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要`math` 模块内对象的对应复数版本，请使用`cmath`，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

context -- 上下文
此术语根据其所在场合和使用方式的不同而具有不同的含义。一些常见的含义为:

- 通过 `with` 语句由一个*context manager* 所创建的临时环境或状态。
- 一组关联到特定`contextvars.Context` 对象并通过`ContextVar` 对象来访问的键值绑定。另请参见*context variable*。
- 一个`contextvars.Context` 对象。另请参见*current context*。

上下文管理协议
`__enter__()` 和 `__exit__()` 方法将由 `with` 语句来调用。参见 **PEP 343**。

context manager -- 上下文管理器
一个实现了*context management protocol* 并负责控制某个 `with` 语句内的环境的对象。参见 **PEP 343**。

context variable -- 上下文变量
一个具体值取决于哪个上下文是*current context* 的变量。这些值是通过`contextvars.ContextVar` 对象来访问的。上下文变量主要被用来隔离并发的异步任务之间的状态。

contiguous -- 连续
一个缓冲如果是 *C 连续*或 *Fortran 连续*就会被认为是连续的。零维缓冲是 C 和 Fortran 连续的。在一维数组中，所有条目必须在内存中彼此相邻地排列，采用从零开始的递增索引顺序。在多维 C-连续数组中，当按内存地址排列时用最后一个索引访问条目时速度最快。但是在 Fortran 连续数组中则是用第一个索引最快。

coroutine -- 协程
协程是子例程的更一般形式。子例程可以在某一点进入并在另一点退出。协程则可以在许多不同的点上进入、退出和恢复。它们可通过 `async def` 语句来实现。参见 **PEP 492**。

coroutine function -- 协程函数
返回一个*coroutine* 对象的函数。协程函数可通过 `async def` 语句来定义，并可能包含 `await`、`async for` 和 `async with` 关键字。这些特性是由 **PEP 492** 引入的。

CPython
Python 编程语言的规范实现，在 python.org 上发布。”CPython” 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

current context -- 当前上下文
在当前被`ContextVar` 对象用来访问 (获取或设置) 上下文变量 的值的*context* (`contextvars.Context` 对象)。每个线程都具有它自己的当前上下文。用于执行异步任务 (参见`asyncio`) 的框架会在每个任务开始或恢复执行时将任务关联到一个成为当前上下文的上下文。

decorator -- 装饰器
返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括`classmethod()` 和`staticmethod()`。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

同样的概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见 函数定义和 类定义的文档。

descriptor -- 描述器
任何定义了 `__get__()`, `__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时会在 *a* 类的字典中查找名称为 *b* 的对象，但如果 *b* 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、特征属性、类方法、静态方法以及对超类的引用等等。

有关描述器的方法的更多信息，请参阅 descriptors 或 描述器使用指南。

dictionary -- 字典
一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有 `__hash__()` 和 `__eq__()` 方法的对象。在 Perl 中称为 hash。

dictionary comprehension -- 字典推导式
处理一个可迭代对象中的所有或部分元素并返回结果字典的一种紧凑写法。`results = {n: n ** 2 for n in range(10)}` 将生成一个由键 `n` 到值 `n ** 2` 的映射构成的字典。参见 comprehensions。

dictionary view -- 字典视图
从 *dict.keys()*, *dict.values()* 和 *dict.items()* 返回的对象被称为字典视图。它们提供了字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。要将字典视图强制转换为真正的列表，可使用 `list(dictview)`。参见字典视图对象。

docstring -- 文档字符串
作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码块被执行时将被忽略，但会被编译器识别并放入所在类、函数或模块的 *__doc__* 属性中。由于它可用于代码内省，因此是存放对象的文档的规范位置。

duck-typing -- 鸭子类型
指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（“看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。”）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用 *type()* 或 *isinstance()* 检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用 *hasattr()* 检测或是 *EAFP* 编程。

EAFP
“求原谅比求许可更容易”的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓 *LBYL* 风格，常见于 C 等许多其他语言。

expression -- 表达式
可以求出某个值的语法单元。换句话说，一个表达式就是表达元素例如字面值、名称、属性访问、运算符或函数调用的汇总，它们最终都会返回一个值。与许多其他语言不同，并非所有语言构件都是表达式。还存在不能被用作表达式的 *statement*，例如 `while`。赋值也是属于语句而非表达式。

extension module -- 扩展模块
以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

f-string -- f-字符串
带有 `'f'` 或 `'F'` 前缀的字符串字面值通常被称为“f-字符串”即 格式化字符串字面值的简写。参见 **PEP 498**。

file object -- 文件对象
对外公开面向文件的 API（带有 `read()` 或 `write()` 等方法）以使用下层资源的对象。根据其创建方式的不同，文件对象可以处理对真实磁盘文件、其他类型的存储或通信设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等）。文件对象也被称为 *文件型对象* 或 *流*。

实际上共有三种类别的文件对象: 原始二进制文件, 缓冲二进制文件 以及文本文件。它们的接口定义均在 *io* 模块中。创建文件对象的规范方式是使用 *open()* 函数。

file-like object -- 文件型对象
file object 的同义词。

filesystem encoding and error handler -- 文件系统编码格式与错误处理器
Python 用来从操作系统解码字节串和向操作系统编码 Unicode 的编码格式与错误处理器。

文件系统编码格式必须保证能成功解码长度在 128 以下的所有字节串。如果文件系统编码格式无法提供此保证，则 API 函数可能会引发 *UnicodeError*。

sys.getfilesystemencoding() 和 *sys.getfilesystemencodeerrors()* 函数可被用来获取文件系统编码格式与错误处理器。

filesystem encoding and error handler 是在 Python 启动时通过 `PyConfig_Read()` 函数来配置的：请参阅 `PyConfig` 的 `filesystem_encoding` 和 `filesystem_errors` 等成员。

另请参见*locale encoding*。

finder -- 查找器
一种会尝试查找被导入模块的*loader* 的对象。

存在两种类型的查找器: *元路径查找器* 配合*sys.meta_path* 使用，以及*路径条目查找器* 配合*sys.path_hooks* 使用。

请参阅 finders-and-loaders 和*importlib* 以了解更多细节。

floor division -- 向下取整除法
向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 `//` 。例如，表达式 `11 // 4` 的计算结果是 `2` ，而与之相反的是浮点数的真正除法返回 `2.75` 。注意 `(-11) // 4` 会返回 `-3` 因为这是 `-2.75` 向下舍入得到的结果。见 **PEP 238** 。

free threading -- 自由线程
一种线程模型，在同一个解释器内部的多个线程可以同时运行 Python 字节码。与此相对的是*global interpreter lock*，在同一时刻只允许一个线程执行 Python 字节码。参见 **PEP 703**。

free variable -- 自由变量
在正式场合下，如 语言执行模型所定义的，自由变量是指在某个命名空间中被使用的不属于该命名空间中的局部变量的任何变量。参见*closure variable* 中的样例。在实际应用中，由于 `codeobject.co_freevars` 属性被如此命名，该术语有时也被用作*closure variable* 的同义词。

function -- 函数
可以向调用者返回某个值的一组语句。还可以向其传入零个或多个*参数* 并在函数体执行中被使用。另见*parameter*, *method* 和 function 等节。

function annotation -- 函数标注
即针对函数形参或返回值的*annotation* 。

函数标注通常用于*类型提示*：例如以下函数预期接受两个*int* 参数并预期返回一个*int* 值:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函数标注语法的详解见 function 一节。

参见*variable annotation* 和 **PEP 484**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标的最佳实践。

__future__
future 语句, `from __future__ import <feature>` 指示编译器使用将在未来的 Python 发布版中成为标准的语法和语义来编译当前模块。*__future__* 模块文档记录了可能的 *feature* 取值。通过导入此模块并对其变量求值，你可以看到每项新特性在何时被首次加入到该语言中以及它将（或已）在何时成为默认:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection -- 垃圾回收
释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用*gc* 模块来控制垃圾回收器。

generator -- 生成器
返回一个*generator iterator* 的函数。它看起来很像普通函数，不同点在于其包含 `yield` 表达式以便产生一系列值供给 for-循环使用或是通过*next()* 函数逐一获取。

通常是指生成器函数，但在某些情况下也可能是指 *生成器迭代器*。如果需要清楚表达具体含义，请使用全称以避免歧义。

generator iterator -- 生成器迭代器
generator 函数所创建的对象。

每个 `yield` 会临时暂停处理过程，记住执行状态（包括局部变量和挂起的 try 语句）。当 *生成器迭代器* 恢复时，它会从离开位置继续执行（不同于每次被唤起时都从新开始的普通函数）。

generator expression -- 生成器表达式
返回一个*iterator* 的*expression*。它看起来很像普通表达式后带有定义了一个循环变量、范围的 `for` 子句，以及一个可选的 `if` 子句。以下复合表达式会为外层函数生成一系列值:

```
>>> sum(i*i for i in range(10))         # 平方值 0, 1, 4, ... 81 之和
285
```

generic function -- 泛型函数
为不同的类型实现相同操作的多个函数所组成的函数。在调用时会由调度算法来确定应该使用哪个实现。

另请参见*single dispatch* 术语表条目、*functools.singledispatch()* 装饰器以及 **PEP 443**。

generic type -- 泛型
可参数化的*type*；通常为*list* 或*dict* 这样的 容器类。用于*类型提示* 和*注解*。

更多细节参见*泛型别名类型*、**PEP 483**、**PEP 484**、**PEP 585** 和*typing* 模块。

GIL
参见*global interpreter lock*。

global interpreter lock -- 全局解释器锁
CPython 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括*dict* 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

在 Python 3.13 中，GIL 可以使用 `--disable-gil` 构建配置选项来禁用。在使用此选项构建 Python 之后，代码必须附带 `-X gil=0` 或在设置 `PYTHON_GIL=0` 环境变量后运行。此特性将为多线程应用程序启用性能提升并让高效率地使用多核 CPU 更加容易。要了解详情，请参阅 **PEP 703**。

hash-based pyc -- 基于哈希的 pyc
使用对应源文件的哈希值而非最后修改时间来确定其有效性的字节码缓存文件。参见 pyc-invalidation。

hashable -- 可哈希
一个对象如果具有在其生命期内绝不改变的哈希值（它需要有 `__hash__()` 方法），并可以同其他对象进行比较（它需要有 `__eq__()` 方法）就被称为 *可哈希* 对象。可哈希对象必须具有相同的哈希值比较结果才会相等。

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

大多数 Python 中的不可变内置对象都是可哈希的；可变容器（例如列表或字典）都不可哈希；不可变容器（例如元组和 frozenset）仅当它们的元素均为可哈希时才是可哈希的。用户定义类的实例对象默认是可哈希的。它们在比较时一定不相同（除非是与自己比较），它们的哈希值的生成是基于它们的*id()*。

IDLE
Python 的集成开发与学习环境。*IDLE --- Python 编辑器和 shell* 是 Python 标准发行版附带的基本编辑器和解释器环境。

immortal -- 永生对象
*永生对象*是在 **PEP 683** 中引入的 CPython 实现细节。

如果对象属于永生对象，则它的*reference count* 永远不会被修改，因而它在解释器运行期间永远不会被取消分配。例如，*True* 和*None* 在 CPython 中都属于永生对象。

immutable -- 不可变对象
具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

import path -- 导入路径
由多个位置（或路径条目）组成的列表，会被模块的*path based finder* 用来查找导入目标。在导入时，此位置列表通常来自*sys.path*，但对次级包来说也可能来自上级包的 __path__ 属性。

importing -- 导入
令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

importer -- 导入器
查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

interactive -- 交互
Python 带有一个交互式解释器，这意味着你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 python 命令（也可以在你的计算机主菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候这会非常方便（记住 help(x) 函数）。有关交互模式的详情，参见 tut-interac。

interpreted -- 解释型
Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

interpreter shutdown -- 解释器关闭
当被要求关闭时，Python 解释器将进入一个特殊运行阶段并逐步释放所有已分配资源，例如模块和各种关键内部结构等。它还会多次调用垃圾回收器。这会触发用户定义析构器或弱引用回调中的代码执行。在关闭阶段执行的代码可能会遇到各种异常，因为其所依赖的资源已不再有效（常见的例子有库模块或警告机制等）。

解释器需要关闭的主要原因有 __main__ 模块或所运行的脚本已完成执行。

iterable -- 可迭代对象
一种能够逐个返回其成员项的对象。可迭代对象的例子包括所有序列类型（如*list*, *str* 和*tuple* 等）以及某些非序列类型如*dict*, 文件对象 以及任何定义了 __iter__() 方法或实现了*sequence* 语义的 __getitem__() 方法的自定义类的对象。

可迭代对象可被用于 for 循环以及许多其他需要一个序列的地方 (*zip()*, *map()*, ...)。当一个可迭代对象作为参数被传给内置函数*iter()* 时，它会返回该对象的迭代器。这种迭代器适用于对值集合的一次性遍历。在使用可迭代对象时，你通常不需要调用*iter()* 或者自己处理迭代器对象。for 语句会自动为你处理那些操作，创建一个临时的未命名变量用来在循环期间保存迭代器。参见*iterator*, *sequence* 和*generator*。

iterator -- 迭代器
用来表示一连串数据流的对象。重复调用迭代器的*__next__()* 方法 (或将其传给内置函数*next()*) 将逐个返回流中的项。当没有数据可用时则将引发*StopIteration* 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 __next__() 方法只会再次引发*StopIteration*。迭代器必须具有*__iter__()* 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象 (例如*list*) 在你每次将其传入*iter()* 函数或是在 for 循环中使用时都会产生一个新的迭代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看迭代器类型。

CPython 没有强制推行迭代器定义*__iter__()* 的要求。还要注意的是自由线程 CPython 并不保证迭代器操作的线程安全性。

key function -- 键函数
键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，*locale.strxfrm()* 可用于生成一个符合特定区域排序约定的排序键。

Python 中有许多工具都允许用键函数来控制元素的排位或分组方式。其中包括*min()*, *max()*, *sorted()*, *list.sort()*, *heapq.merge()*, *heapq.nsmallest()*, *heapq.nlargest()* 以及*itertools.groupby()*。

有多种方式可以创建一个键函数。例如，*str.lower()* 方法可以用作忽略大小写排序的键函数。或者，键函数也可通过 lambda 表达式来创建例如 lambda r: (r[0], r[2])。此外，*operator.*

attrgetter(), *operator.itemgetter()* 和*operator.methodcaller()* 是键函数的三个构造器。请查看 排序指引来获取创建和使用键函数的示例。

keyword argument -- 关键字参数
参见*argument*。

lambda
由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

LBYL
“先查看后跳跃”的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用 `if` 语句。

在多线程环境中，LBYL 方式会导致“查看”和“跳跃”之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

lexical analyzer -- 词法分析器
分词器的正式名称；参见*token*。

list -- 列表
一种 Python 内置*sequence*。虽然名为列表，但它更类似于其他语言中的数组而非链表，因为访问元素的时间复杂度为 *O*(1)。

list comprehension -- 列表推导式
处理一个序列中的所有或部分元素并返回结果列表的一种紧凑写法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 将生成一个 0 到 255 范围内的十六进制偶数对应字符串（0x..）的列表。其中 `if` 子句是可选的，如果省略则 `range(256)` 中的所有元素都会被处理。

loader -- 加载器
负责加载模块的对象。它必须定义 `exec_module()` 和 `create_module()` 方法以实现*Loader* 接口。加载器通常由一个*finder* 返回。另请参阅：

- finders-and-loaders
- *importlib.abc.Loader*
- **PEP 302**

locale encoding -- 语言区域编码格式
在 Unix 上，它是 LC_CTYPE 语言区域的编码格式。它可以通过*locale.setlocale(locale.LC_CTYPE, new_locale)* 来设置。

在 Windows 上，它是 ANSI 代码页 (如: `"cp1252"`)。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作为语言区域编码格式。

locale.getencoding() 可被用来获取语言区域编码格式。

另请参阅*filesystem encoding and error handler*。

magic method -- 魔术方法
special method 的非正式同义词。

mapping -- 映射
一种支持任意键查找并实现了*collections.abc.Mapping* 或*collections.abc.MutableMapping* *抽象基类* 所规定方法的容器对象。此类对象的例子包括*dict*, *collections.defaultdict*, *collections.OrderedDict* 以及*collections.Counter*。

meta path finder -- 元路径查找器
sys.meta_path 的搜索所返回的*finder*。元路径查找器与*path entry finders* 存在关联但并不相同。

请查看*importlib.abc.MetaPathFinder* 了解元路径查找器所实现的方法。

metaclass -- 元类
一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定

义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见 metaclasses。

method -- 方法
在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 `self`)。参见*function* 和*nested scope*。

method resolution order -- 方法解析顺序
方法解析顺序就是在查找成员时搜索基类的顺序。请参阅 python_2.3_mro 了解自 2.3 发布版起 Python 解释器所使用算法的详情。

module -- 模块
此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

module spec -- 模块规格
一个命名空间，其中包含用于加载模块的相关导入信息。是*importlib.machinery.ModuleSpec* 的实例。

另请参阅 module-specs。

MRO
参见*method resolution order*。

mutable -- 可变对象
可变对象可以在其*id()* 保持固定的情况下改变其取值。另请参见*immutable*。

named tuple -- 具名元组
术语“具名元组”可用于任何继承自元组，并且其中的可索引元素还能使用名称属性来访问的类型或类。这样的类型或类还可能拥有其他特性。

有些内置类型属于具名元组，包括*time.localtime()* 和*os.stat()* 的返回值。另一个例子是*sys.float_info*:

```
>>> sys.float_info[1]                   # 索引访问
1024
>>> sys.float_info.max_exp              # 命名字段访问
1024
>>> isinstance(sys.float_info, tuple)   # 属于元组
True
```

有些具名元组是内置类型（比如上面的例子）。此外，具名元组还可通过常规类定义从*tuple* 继承并定义指定名称的字段的方式来创建。这样的类可以手工编号，或者也可以通过继承*typing.NamedTuple*，或者使用工厂函数*collections.namedtuple()* 来创建。后一种方式还会添加一些手工编写或内置的具名元组所没有的额外方法。

namespace -- 命名空间
命名空间是存放变量的场所。命名空间有局部、全局和内置的，还有对象中的嵌套命名空间（在方法之内）。命名空间通过防止命名冲突来支持模块化。例如，函数*builtins.open* 与*os.open()* 可通过各自的命名空间来区分。命名空间还通过明确哪个模块实现那个函数来帮助提高可读性和可维护性。例如，*random.seed()* 或*itertools.islice()* 这种写法明确了这些函数是由*random* 与*itertools* 模块分别实现的。

namespace package -- 命名空间包
一种仅被用作子包的容器的*package*。命名空间包可以没有实体表示物，具体而言就是不同于*regular package* 因为它们没有 `__init__.py` 文件。

命名空间包允许几个可单独安装的包具有共同的父包。在其他情况下，则推荐使用*regular package*。

要了解更多信息，请参阅 **PEP 420** 和 reference-namespace-package。

另可参见*module*。

nested scope -- 嵌套作用域
在一个定义范围内引用变量的能力。例如，在另一函数之内定义的函数可以引用前者的变量。请注意嵌套作用域默认只对引用有效而对赋值无效。局部变量的读写都受限于最内层作用域。类似的，全局变量的读写则作用于全局命名空间。通过 `nonlocal` 关键字可允许写入外层作用域。

new-style class -- 新式类
对目前已被应用于所有类对象的类形式的旧称谓。在较早的 Python 版本中，只有新式类能够使用 Python 新增的更灵活我，如 `__slots__`、描述器、特征属性、`__getattribute__()`、类方法和静态方法等。

object -- 对象
任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

optimized scope -- 已优化的作用域
当代码被编译时编译器已充分知晓目标局部变量名称的作用域，这允许对这些名称的读写进行优化。针对函数、生成器、协程、推导式和生成器表达式的局部命名空间都是这样的已优化作用域。注意：大部分解释器优化将应用于所有作用域，只有那些依赖于已知的局部和非局部变量名称集合的优化会仅限于已优化的作用域。

package -- 包
一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是具有 `__path__` 属性的 Python 模块。

另参见*regular package* 和*namespace package*。

parameter -- 形参
function（或方法）定义中的命名实体，它指定函数可以接受的一个*argument*（或在某些情况下，多个实参）。有五种形参：

- *positional-or-keyword*：位置或关键字，指定一个可以作为*位置参数* 传入也可以作为*关键字参数* 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*：仅限位置，指定一个只能通过位置传入的参数。仅限位置形参可通过在函数定义的形参列表中它们之后包含一个 `/` 字符来定义，例如下面的 *posonly1* 和 *posonly2*：

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 `*` 来定义，例如下面的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 `*` 来定义，例如下面的 *args*：

```
def func(*args, **kwargs): ...
```

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 `**` 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

另参见*argument* 术语表条目、参数与形参的区别中的常见问题、`inspect.Parameter` 类、function 一节以及 **PEP 362**。

path entry -- 路径入口
import path 中的一个单独位置，会被*path based finder* 用来查找要导入的模块。

path entry finder -- 路径入口查找器
任一可调用对象使用`sys.path_hooks` (即*path entry hook*) 返回的*finder*，此种对象能通过*path entry* 来定位模块。

请参看*importlib.abc.PathEntryFinder* 以了解路径入口查找器所实现的各个方法。

path entry hook -- 路径入口钩子
一种可调用对象，它在知道如何查找特定*path entry* 中的模块的情况下能够使用*sys.path_hooks* 列表返回一个*path entry finder*。

path based finder -- 基于路径的查找器
默认的一种*元路径查找器*，可在一个*import path* 中查找模块。

path-like object -- 路径类对象
代表一个文件系统路径的对象。路径类对象可以是一个表示路径的*str* 或者*bytes* 对象，还可以是一个实现了*os.PathLike* 协议的对象。一个支持*os.PathLike* 协议的对象可通过调用*os.fspath()* 函数转换为*str* 或者*bytes* 类型的文件系统路径；*os.fsdecode()* 和*os.fsencode()* 可被分别用来确保获得*str* 或*bytes* 类型的结果。此对象是由 **PEP 519** 引入的。

PEP
“Python 增强提议”的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

portion -- 部分
构成一个命名空间包的单个目录内文件集合（也可能存放于一个 zip 文件内），具体定义见 **PEP 420**。

positional argument -- 位置参数
参见*argument*。

provisional API -- 暂定 API
暂定 API 是指被有意排除在标准库的向后兼容性保证之外的应用编程接口。虽然此类接口通常不会再有重大改变，但只要其被标记为暂定，就可能在核心开发者确定有必要的情况下进行向后不兼容的更改（甚至包括移除该接口）。此种更改并不会随意进行 -- 仅在 API 被加入之前未考虑到的严重基础性缺陷被发现时才可能会这样做。

即便是对暂定 API 来说，向后不兼容的更改也会被视为“最后的解决方案”——任何问题被确认时都会尽可能先尝试找到一种向后兼容的解决方案。

这种处理过程允许标准库持续不断地演进，不至于被有问题的长期性设计缺陷所困。详情见 **PEP 411**。

provisional package -- 暂定包
参见*provisional API*。

Python 3000
Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为“Py3k”。

Pythonic
指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 `for` 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

```
for i in range(len(food)):
    print(food[i])
```

而相应的更简洁更 Pythonic 的方法是这样的:

```
for piece in food:
    print(piece)
```

qualified name -- 限定名称
一个以点号分隔的名称，显示从模块的全局作用域到该模块中定义的某个类、函数或方法的“路径”，相关定义见 **PEP 3155**。对于最高层级的函数和类，限定名称与对象名称一致:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

当被用于引用模块时，完整限定名称意为标示该模块的以点号分隔的整个路径，其中包含其所有的父包，例如 `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count -- 引用计数
指向某个对象的引用的数量。当一个对象的引用计数降为零时，它就会被释放。特殊的*immortal* 对象具有永远不会被修改的引用计数，因此这种对象永远不会被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。程序员可以调用`sys.getrefcount()` 函数来返回特定对象的引用计数。

regular package -- 常规包
传统型的*package*，例如包含有一个 `__init__.py` 文件的目录。

另参见*namespace package*。

REPL
“读取-求值-打印循环” read-eval-print loop 的缩写，*interactive* 解释器 shell 的另一个名字。

__slots__
一种写在类内部的声明，通过预先声明实例属性等对象并移除实例字典来节省内存。虽然这种技巧很流行，但想要用好却并不容易，最好是只保留在少数情况下采用，例如极耗内存的应用程序，并且其中包含大量实例。

sequence -- 序列
一种*iterable*，它支持通过 `__getitem__()` 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 `__len__()` 方法。内置序列类型有`list`, `str`, `tuple` 和`bytes` 等。请注意虽然`dict` 也支持 `__getitem__()` 和 `__len__()`，但它被归类为映射而非序列，因为它使用任意的*hashable* 键而不是整数来查找元素。

`collections.abc.Sequence` 抽象基类定义了一个更丰富的接口，它在 `__getitem__()` 和 `__len__()` 之外，还添加了 `count()`, `index()`, `__contains__()` 和 `__reversed__()`。实现此扩展接口的类型可以使用`register()` 来显式地注册。要获取有关通用序列方法的更多文档，请参阅通用序列操作。

set comprehension -- 集合推导式
处理一个可迭代对象中的所有或部分元素并返回结果集合的一种紧凑写法。`results = {c for c in 'abracadabra' if c not in 'abc'}` 将生成字符串集合 `{'r', 'd'}`。参见 comprehensions。

single dispatch -- 单分派
一种*generic function* 分派形式，其实现是基于单个参数的类型来选择的。

slice -- 切片
通常只包含了特定*sequence* 的一部分的对象。切片是通过使用下标标记来创建的，在 `[]` 中给出几个以冒号分隔的数字，例如 `variable_name[1:3:5]`。方括号（下标）标记在内部使用`slice` 对象。

soft deprecated -- 软弃用
被软弃用的 API 不应在新编写的代码中使用，但在已有代码中使用仍是安全的。这样的 API 将保留在文档中并被测试，但不会再获得进一步的功能增强。

与普通的弃用不同，软弃用没有 API 移除计划也不会发出弃用警告。

参见 PEP 387: Soft Deprecation。

special method -- 特殊方法
一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见 specialnames。

statement -- 语句
语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如 `if`、`while` 或 `for`。

static type checker -- 静态类型检查器
读取 Python 代码并进行分析，以查找问题例如拼写错误的外部工具。另请参阅类型提示 以及*typing* 模块。

strong reference -- 强引用
在 Python 的 C API 中，强引用是指为持有引用的代码所拥有的对象的引用。在创建引用时可通过调用 `Py_INCREF()` 来获取强引用而在删除引用时可通过 `Py_DECREF()` 来释放它。

`Py_NewRef()` 函数可被用于创建一个对象的强引用。通常，必须在退出某个强引用的作用域时在该强引用上调用 `Py_DECREF()` 函数，以避免引用的泄漏。

另请参阅*borrowed reference*。

text encoding -- 文本编码格式
在 Python 中，一个字符串是一串 Unicode 代码点（范围为 `U+0000--U+10FFFF`）。为了存储或传输一个字符串，它需要被序列化为一串字节。

将一个字符串序列化为一个字节序列被称为“编码”，而从字节序列中重新创建字符串被称为“解码”。

有各种不同的文本序列化编码器，它们被统称为“文本编码格式”。

text file -- 文本文件
一种能够读写*str* 对象的*file object*。通常一个文本文件实际是访问一个面向字节的数据流并自动处理*text encoding*。文本文件的例子包括以文本模式（`'r'` 或 `'w'`）打开的文件、*sys.stdin*、*sys.stdout* 以及*io.StringIO* 的实例。

另请参看*binary file* 了解能够读写字节型对象 的文件对象。

形符
一个源代码小单元，由 词法分析器 (或称 分词器) 生成。名称、数字、字符串、运算符、换行符等均由词元来表示。

tokenize 模块对外暴露了 Python 的词法分析器。*token* 模块包含了有关各种词元类型的信息。

triple-quoted string -- 三引号字符串
首尾各带三个连续双引号（”）或者单引号（’）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

type -- 类型
Python 对象的类型决定它属于什么种类；每个对象都具有特定的类型。对象的类型可通过其 `__class__` 属性来访问或是用 `type(obj)` 来获取。

type alias -- 类型别名
一个类型的同义词，创建方式是把类型赋值给特定的标识符。

类型别名的作用是简化类型注解。例如:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以这样提高可读性:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

参见*typing* 和 **PEP 484**，其中有对此功能的详细描述。

type hint -- 类型注解
annotation 为变量、类属性、函数的形参或返回值指定预期的类型。

类型提示是可选的而不是 Python 的强制要求，但它们对静态类型检查器 很有用处。它们还能协助 IDE 实现代码补全与重构。

全局变量、类属性和函数的类型注解可以使用*typing.get_type_hints()* 来访问，但局部变量则不可以。

参见*typing* 和 **PEP 484**，其中有对此功能的详细描述。

universal newlines -- 通用换行
一种解读文本流的方式，将以下所有符号都识别为行结束标志：Unix 的行结束约定 `'\n'`、Windows 的约定 `'\r\n'` 以及旧版 Macintosh 的约定 `'\r'`。参见 **PEP 278** 和 **PEP 3116** 和*bytes.splitlines()* 了解更多用法说明。

variable annotation -- 变量标注
对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

```
class C:
    field: 'annotation'
```

变量标注通常被用作类型提示：例如以下变量预期接受*int* 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见 annassign 一节。

参见*function annotation*, **PEP 484** 和 **PEP 526**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标注的最佳实践。

virtual environment -- 虚拟环境
一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

另参见*venv*。

virtual machine -- 虚拟机
一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

Zen of Python -- Python 之禅
列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入”`import this`”。

APPENDIX B

关于本文档

Python 的文档是用 Sphinx 从 reStructuredText 源生成的，*Sphinx* 是一个专为处理 Python 文档而编写的文档生成器。

本文档及其工具链之开发，皆在于志愿者之努力，亦恰如 Python 本身。如果您想为此作出贡献，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，原始 Python 文档工具集之创造者，众多文档之作者；
- 用于创建 reStructuredText 和 Docutils 套件的 Docutils 项目；
- Fredrik Lundh 的 Alternative Python Reference 项目，为 Sphinx 提供许多好的点子。

B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码分发的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档——谢谢你们！

APPENDIX C

历史和许可证

C.1 该软件的历史

Python 是在 1990 年代初由 Guido van Rossum 在荷兰的 Stichting Mathematisch Centrum (CWI，见 https://www.cwi.nl) 作为一门名为 ABC 的语言的后继者创造的。Guido 仍然是 Python 的主要作者，尽管它也包括了许多来自其他人的贡献。

在 1995 年，Guido 在弗吉尼亚州 Reston 市的 Corporation for National Research Initiatives (CNRI，见 https://www.cnri.reston.va.us) 继续他对 Python 的工作，他在那里发布了该软件的多个版本。

在 2000 年 5 月，Guido 和 Python 核心开发团队移至 BeOpen.com 组建了 BeOpen PythonLabs 团队。同年 10 月，PythonLabs 团队移至 Digital Creations，后改名为 Zope Corporation。在 2001 年，Python Software Foundation (PSF，见 https://www.python.org/psf/) 成立，这是一个专门为持有与 Python 相关的知识产权而创立的非营利组织。Zope Corporation 是 PSF 的一个赞助成员。

所有 Python 发布版都是开源的 (有关开源的定义见 https://opensource.org)。在历史上，大多数但并非全部 Python 发布版还是 GPL 兼容的；下表总结了各个版本的情况。

发布版本	源自	年份	所有者	GPL 兼容? (1)
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	是 (2)
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

备注

(1) GPL 兼容并不意味着我们是基于 GPL 发布 Python。与 GPL 不同，所有 Python 许可证都允许你分发经修改的版本而无需开源你所做的修改。GPL 兼容的许可证使得 Python 可以与其他基于

GPL 发布的软件结合使用；其他许可证则不可以。

(2) 按照 Richard Stallman 的说法，1.6.1 不是 GPL 兼容的，因为它的许可证包含特定的法律条款。但是按照 CNRI 的说法，Stallman 的律师已告诉 CNRI 的律师 1.6.1 与 GPL " 并非不兼容"。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

Python 软件和文档的使用许可均为 Python Software Foundation License Version 2。

从 Python 3.8.6 开始，文档中的示例、操作指导和其他代码均采用 PSF License Version 2 和*Zero-Clause BSD license* 双重使用许可。

某些包含在 Python 中的软件基于不同的许可。这些许可会与相应许可之下的代码一同列出。有关这些许可的不完整列表请参阅收录软件的许可与鸣谢。

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
```

(续下页)

(接上页)

```
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(续下页)

(接上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 收录软件的许可与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

作为*random* 模块下层的 _random C 扩展包括基于从 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

(续下页)

(接上页)

```
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 套接字

socket 使用了 `getaddrinfo()` 和 `getnameinfo()` WIDE 项目的不同源文件中: https://www.wide.ad.jp/

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 异步套接字服务

`test.support.asynchat` 和 `test.support.asyncore` 模块包含以下说明。:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
```

(续下页)

(接上页)

```
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

http.cookies 模块包含以下声明:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 执行追踪

trace 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
```

(续下页)

(接上页)

```
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 与 UUdecode 函数

uu 编解码器包含以下声明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 远程过程调用

xmlrpc.client 模块包含以下声明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(续下页)

(接上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模块包含以下说明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

select 模块关于 kqueue 的接口包含以下声明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Python/pyhash.c 文件包含 Marek Majkowski' 对 Dan Bernstein 的 SipHash24 算法的实现。它包含以下声明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 和 dtoa

Python/dtoa.c 文件提供了 C 函数 dtoa 和 strtod，用于 C 双精度数值和字符串之间的转换，它派生自由 David M. Gay 编写的同名文件。目前该文件可在 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 访问。在 2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果操作系统有支持则*hashlib*, *posix* 和*ssl* 会使用 OpenSSL 库来提升性能。此外，Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本，所以我们也在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 发布版，及其后续衍生版本，均使用 Apache License v2:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/
```

(续下页)

(接上页)

```
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.
```

(续下页)

(接上页)

```
2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(续下页)

(接上页)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

pyexpat 扩展是使用所包括的 expat 源副本来构建的，除非配置了 `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
```

(续下页)

(接上页)

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

作为*ctypes* 模块下层的 `_ctypes` C 扩展是使用包括了 libffi 源的副本构建的，除非构建时配置了 `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建*zlib* 扩展:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
```

(续下页)

(接上页)

```
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

作为*decimal* 模块下层的 `_decimal` C 扩展是使用包括了 libmpdec 库的副本构建的，除非构建时配置了 `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

(续下页)

(接上页)

```
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 测试套件

test 包中的 C14N 2.0 测试集 (Lib/test/xmltestdata/c14n-20/) 提取自 W3C 网站 https://www.w3.org/TR/xml-c14n2-testcases/ 并根据 3 条款版 BSD 许可证发行:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT 许可证:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
```

(续下页)

(接上页)

```
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

asyncio 模块的某些部分来自 uvloop 0.16，它是基于 MIT 许可证发行的:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

文件 `Python/qsbr.c` 改编自 subr_smr.c 中 FreeBSD 的"Global Unbounded Sequences" 安全内存回收方案。该文件是基于 2 条款 BSD 许可证分发的:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
```

(续下页)

(接上页)

```
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX D

版权所有

Python 与这份文档：

版权所有 © 2001-2024 Python 软件基金会。保留所有权利。

版权所有 © 2000 BeOpen.com。保留所有权利。

版权所有 © 1995-2000 Corporation for National Research Initiatives。保留所有权利。

版权所有 © 1991-1995 Stichting Mathematisch Centrum。保留所有权利。

有关完整的许可证和许可信息，请参见历史和许可证。

Bibliography

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009. 该书的第三版不再包含 Python，但第一版极详细地覆盖了正则表达式模式串的编写。

[C99] ISO/IEC 9899:1999. "Programming languages -- C." 该标准的公开草案可从 https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf 获得。

Python 模块索引

_

__future__, 1871
__main__, 1821
_thread, 978
_tkinter, 1504

a

abc, 1855
aifc, 2055
argparse, 785
array, 266
ast, 1941
asynchat, 2055
asyncio, 981
asyncore, 2055
atexit, 1860
audioop, 2056

b

base64, 1241
bdb, 1731
binascii, 1244
bisect, 263
builtins, 1820
bz2, 532

c

calendar, 230
cgi, 2056
cgitb, 2056
chunk, 2056
cmath, 321
cmd, 1491
code, 1897
codecs, 173
codeop, 1899
collections, 238
collections.abc, 254
colorsys, 1439
compileall, 1992
concurrent.futures, 944
configparser, 576
contextlib, 1842
contextvars, 974
copy, 282
copyreg, 484
cProfile, 1749
crypt, 2056
csv, 569
ctypes, 754
curses (*Unix*), 854
curses.ascii, 879
curses.panel, 883
curses.textpad, 878

d

dataclasses, 1832
datetime, 191
dbm, 488
dbm.dumb, 493
dbm.gnu (*Unix*), 490
dbm.ndbm (*Unix*), 492
dbm.sqlite3 (*All*), 490
decimal, 325
difflib, 143
dis, 1996
distutils, 2056
doctest, 1601

e

email, 1159
email.charset, 1206
email.contentmanager, 1185
email.encoders, 1208
email.errors, 1179
email.generator, 1170
email.header, 1204
email.headerregistry, 1180
email.iterators, 1211
email.message, 1160
email.mime, 1201
email.mime.application, 1202
email.mime.audio, 1202
email.mime.base, 1201
email.mime.image, 1203
email.mime.message, 1203
email.mime.multipart, 1202

email.mime.nonmultipart, 1202
email.mime.text, 1203
email.parser, 1167
email.policy, 1173
email.utils, 1208
encodings.idna, 188
encodings.mbcs, 188
encodings.utf_8_sig, 189
ensurepip, 1771
enum, 291
errno, 745

f

faulthandler, 1736
fcntl (*Unix*), 2039
filecmp, 446
fileinput, 851
fnmatch, 456
fractions, 351
ftplib, 1359
functools, 393

g

gc, 1872
getopt, 2051
getpass, 851
gettext, 1441
glob, 453
graphlib, 305
grp (*Unix*), 2035
gzip, 528

h

hashlib, 599
heapq, 260
hmac, 610
html, 1249
html.entities, 1254
html.parser, 1249
http, 1348
http.client, 1352
http.cookiejar, 1401
http.cookies, 1398
http.server, 1392

i

idlelib, 1548
imaplib, 1368
imghdr, 2057
imp, 2057
importlib, 1909
importlib.abc, 1912
importlib.machinery, 1918
importlib.metadata, 1932
importlib.resources, 1928
importlib.resources.abc, 1931
importlib.util, 1923
inspect, 1875
io, 677
ipaddress, 1422
itertools, 377

j

json, 1212
json.tool, 1220

k

keyword, 1984

l

linecache, 457
locale, 1448
logging, 701
logging.config, 718
logging.handlers, 729
lzma, 536

m

mailbox, 1221
mailcap, 2057
marshal, 487
math, 312
mimetypes, 1239
mmap, 1152
modulefinder, 1906
msilib, 2057
msvcrt (*Windows*), 2019
multiprocessing, 898
multiprocessing.connection, 927
multiprocessing.dummy, 930
multiprocessing.managers, 918
multiprocessing.pool, 924
multiprocessing.shared_memory, 939
multiprocessing.sharedctypes, 916

n

netrc, 594
nis, 2057
nntplib, 2057
numbers, 309

o

operator, 402
optparse, 825
os, 615
os.path, 434
ossaudiodev, 2058

p

pathlib, 411
pdb, 1738
pickle, 469
pickletools, 2017
pipes, 2058
pkgutil, 1903
platform, 742

plistlib, 595
poplib, 1365
posix (*Unix*), 2033
pprint, 283
profile, 1749
pstats, 1750
pty (*Unix*), 2038
pwd (*Unix*), 2034
py_compile, 1991
pyclbr, 1989
pydoc, 1597

q

queue, 971
quopri, 1246

r

random, 354
re, 122
readline (*Unix*), 161
reprlib, 288
resource (*Unix*), 2042
rlcompleter, 165
runpy, 1907

s

sched, 969
secrets, 611
select, 1134
selectors, 1140
shelve, 484
shlex, 1496
shutil, 458
signal, 1144
site, 1893
sitecustomize, 1894
smtpd, 2058
smtplib, 1374
sndhdr, 2058
socket, 1075
socketserver, 1384
spwd, 2058
sqlite3, 494
ssl, 1101
stat, 440
statistics, 363
string, 111
stringprep, 159
struct, 167
subprocess, 951
sunau, 2058
symtable, 1976
sys, 1787
sys.monitoring, 1810
sysconfig, 1815
syslog (*Unix*), 2046

t

tabnanny, 1988
tarfile, 552
telnetlib, 2059
tempfile, 448
termios (*Unix*), 2035
test, 1707
test.regrtest, 1709
test.support, 1710
test.support.bytecode_helper, 1720
test.support.import_helper, 1723
test.support.os_helper, 1721
test.support.script_helper, 1719
test.support.socket_helper, 1718
test.support.threading_helper, 1720
test.support.warnings_helper, 1724
textwrap, 154
threading, 885
time, 690
timeit, 1754
tkinter, 1501
tkinter.colorchooser (*Tk*), 1513
tkinter.commondialog (*Tk*), 1517
tkinter.dnd (*Tk*), 1520
tkinter.filedialog (*Tk*), 1515
tkinter.font (*Tk*), 1513
tkinter.messagebox (*Tk*), 1517
tkinter.scrolledtext (*Tk*), 1519
tkinter.simpledialog (*Tk*), 1514
tkinter.ttk, 1520
token, 1979
tokenize, 1985
tomllib, 592
trace, 1758
traceback, 1861
tracemalloc, 1760
tty (*Unix*), 2037
turtle, 1457
turtledemo, 1490
types, 276
typing, 1549

u

unicodedata, 158
unittest, 1622
unittest.mock, 1650
urllib, 1321
urllib.error, 1347
urllib.parse, 1338
urllib.request, 1321
urllib.response, 1338
urllib.robotparser, 1347
usercustomize, 1894
uu, 2059
uuid, 1380

v

venv, 1773

w

warnings, 1825

wave, 1437
weakref, 269
webbrowser, 1309
winreg (*Windows*), 2021
winsound (*Windows*), 2030
wsgiref, 1312
wsgiref.handlers, 1316
wsgiref.headers, 1314
wsgiref.simple_server, 1315
wsgiref.types, 1320
wsgiref.util, 1312
wsgiref.validate, 1316

X

xdrlib, 2059
xml, 1254
xml.dom, 1273
xml.dom.minidom, 1283
xml.dom.pulldom, 1286
xml.etree.ElementInclude, 1266
xml.etree.ElementTree, 1256
xml.parsers.expat, 1299
xml.parsers.expat.errors, 1306
xml.parsers.expat.model, 1305
xml.sax, 1288
xml.sax.handler, 1290
xml.sax.saxutils, 1295
xml.sax.xmlreader, 1296
xmlrpc, 1409
xmlrpc.client, 1409
xmlrpc.server, 1417

Z

zipapp, 1782
zipfile, 541
zipimport, 1901
zlib, 525
zoneinfo, 226

非字母

??
    在正则表达式中, 123
..
    在 pathnames 中, 675
..., **2063**
    placeholder, 157, 284, 289
    在 doctests 中, 1608
    省略符字面值, 33, 95
    解释器提示符, 1605, 1804
! (*pdb command*), 1745
- (减号)
    使用 glob 风格的通配符, 453, 456
    使用 printf 风格的格式化, 59, 73
    单目运算符, 36
    双目运算符, 36
    在 doctests 中, 1609
    在字符串格式化中, 115
    在正则表达式中, 124
! (叹号)
    格式字符串字面值形式, 57
! (感叹号)
    使用 glob 风格的通配符, 453, 456
    在 curses 模块中, 882
    在命令解释器中, 1492
    在字符串格式化中, 113
    在结构格式字符串中 in struct format strings, 168
. (点号)
    使用 glob 风格的通配符, 453
    使用 printf 风格的格式化, 58, 73
    在 pathnames 中, 675, 676
    在字符串格式化中, 113
    在正则表达式中, 122
? (问号)
    使用 glob 风格的通配符, 453, 456
    在 argparse 模块中, 795
    在 AST 语法中, 1944
    在 SQL 语句中, 510
    在命令解释器中, 1492
    在正则表达式中, 123
    在结构格式字符串中 in struct format strings, 170
    替代字符, 176
! f-字符串, 57
! 格式化字符串字面值, 57
(*hash*)
    使用 printf 风格的格式化, 59, 73
    在 doctests 中, 1609
    在字符串格式化中, 115
    在正则表达式中, 130
    注释, 1893
$ (货币符号)
    在模板字符串中, 120
    在正则表达式中, 122
    在配置文件中的插值, 580
    环境变量扩展, 436
% (百分号)
    operator, 36
    printf 风格的格式化, 58, 72
    在配置文件中的插值, 580
    日期时间格式, 222, 694, 696
    环境变量扩展 (*Windows*), 436, 2024
& (和号)
    operator, 37
(?
    在正则表达式中, 124
(?!
    在正则表达式中, 126
(?#
    在正则表达式中, 126
(?(
    在正则表达式中, 126
() (圆括号)
    使用 printf 风格的格式化, 58, 73
    在正则表达式中, 124
(?:
    在正则表达式中, 125
(?<!
    在正则表达式中, 126
(?<=
    在正则表达式中, 126
(?=
    在正则表达式中, 126
(?P<
    在正则表达式中, 125
(?P=

    在正则表达式中, 126
*?
    在正则表达式中, 123
* (星号)
    operator, 36
    使用 glob 风格的通配符, 453, 456
    使用 printf 风格的格式化, 58, 73
    在 argparse 模块中, 796
    在 AST 语法中, 1944
    在正则表达式中, 123
**
    operator, 36
    使用 glob 风格的通配符, 454
*+
    在正则表达式中, 123
+?
    在正则表达式中, 123
?+
    在正则表达式中, 123
+ (加号)
    使用 printf 风格的格式化, 59, 73
    单目运算符, 36
    双目运算符, 36
    在 argparse 模块中, 796
    在 doctests 中, 1609
    在字符串格式化中, 115
    在正则表达式中, 123
++
    在正则表达式中, 123
, (逗号)
    在字符串格式化中, 115
-
    idle 命令行选项, 1544
    python--m-py_compile 命令行选项, 1992
/ (斜杠)
    operator, 36
    在 pathnames 中, 675
//
    operator, 36
2 位数表示年份, 690
2038 年, 690
: (冒号)
    在 SQL 语句中, 510
    在字符串格式化中, 113
    格式字符串字面值形式, 57
    路径分隔符 (*POSIX*), 676
; (分号), 676
< (小于)
    operator, 36
    在结构格式字符串中 in struct format strings, 168
< (小于号)
    在字符串格式化中, 114
<<
    operator, 37
<=
    operator, 36
<BLANKLINE>, 1608
<file>
    python--m-py_compile 命令行选项, 1992
!=
    operator, 36
= (等于号)
    在字符串格式化中, 114
    在结构格式字符串中 in struct format strings, 168
    用于帮助使用字符串字面值进行调试, 57
==
    operator, 36
> (大于)
    operator, 36
    在结构格式字符串中 in struct format strings, 168
> (大于号)
    在字符串格式化中, 114
>=
    operator, 36
>>
    operator, 37
>>>, **2063**
    解释器提示符, 1605, 1804
@ (*at*)
    在结构格式字符串中 in struct format strings, 168
[] (方括号)
    使用 glob 风格的通配符, 453, 456
    在字符串格式化中, 113
    在正则表达式中, 124
\ (反斜杠)
    在正则表达式中, 123, 124, 127
    在路径名称中 (*Windows*), 675
    转义序列, 176
\\
    在正则表达式中, 128
\A
    在正则表达式中, 127
\a
    在正则表达式中, 128
\B
    在正则表达式中, 127
\b
    在正则表达式中, 127, 128
\D
    在正则表达式中, 127
\d
    在正则表达式中, 127
\f
    在正则表达式中, 128
\g
    在正则表达式中, 132
\N
    在正则表达式中, 128
    转义序列, 177
\n
    在正则表达式中, 128
\r

在正则表达式中, 128
\S
在正则表达式中, 128
\s
在正则表达式中, 127
\t
在正则表达式中, 128
\U
在正则表达式中, 128
转义序列, 176
\u
在正则表达式中, 128
转义序列, 176
\v
在正则表达式中, 128
\W
在正则表达式中, 128
\w
在正则表达式中, 128
\x
在正则表达式中, 128
转义序列, 176
\Z
在正则表达式中, 128
^ (脱字号)
operator, 37
在 curses 模块中, 882
在字符串格式化中, 114
在正则表达式中, 122, 124
标记, 1607, 1862
_ (下划线)
gettext, 1441
在字符串格式化中, 115
__abs__() (在 operator 模块中), 403
__add__() (在 operator 模块中), 403
__and__() (enum.Flag 方法), 299
__and__() (在 operator 模块中), 403
__args__ (genericalias 属性), 92
__bound__ (typing.TypeVar 属性), 1573
__breakpointhook__() (在 sys 模块中), 1791
__bytes__() (email.message.EmailMessage 方法), 1161
__bytes__() (email.message.Message 方法), 1195
__call__() (argparse.Action 方法), 801
__call__() (email.headerregistry.HeaderRegistry 方法), 1184
__call__() (enum.EnumType 方法), 293
__call__() (weakref.finalize 方法), 272
__call__() (在 operator 模块中), 405
__callback__ (weakref.ref 属性), 270
__cause__ (异常属性), 99
__cause__ (BaseException 属性), 99
__cause__ (traceback.TracebackException 属性), 1864
__ceil__() (fractions.Fraction 方法), 353
__class__ (unittest.mock.Mock 属性), 1660
__code__ (函数对象属性), 95
__concat__() (在 operator 模块中), 404
__constraints__ (typing.TypeVar 属性), 1573
__contains__() (email.message.EmailMessage 方法), 1161
__contains__() (email.message.Message 方法), 1197
__contains__() (enum.EnumType 方法), 294
__contains__() (enum.Flag 方法), 299
__contains__() (mailbox.Mailbox 方法), 1223
__contains__() (在 operator 模块中), 404
__context__ (异常属性), 99
__context__ (BaseException 属性), 99
__context__ (traceback.TracebackException 属性), 1864
__contravariant__ (typing.TypeVar 属性), 1573
__copy__() (拷贝协议), 282
__covariant__ (typing.TypeVar 属性), 1572
__debug__ (内置变量), 34
__deepcopy__() (拷贝协议), 282
__default__ (typing.ParamSpec 属性), 1576
__default__ (typing.TypeVar 属性), 1573
__default__ (typing.TypeVarTuple 属性), 1575
__del__() (io.IOBase 方法), 682
__delitem__() (email.message.EmailMessage 方法), 1162
__delitem__() (email.message.Message 方法), 1197
__delitem__() (mailbox.Mailbox 方法), 1222
__delitem__() (mailbox.MH 方法), 1228
__delitem__() (在 operator 模块中), 405
__dir__() (enum.Enum 方法), 295
__dir__() (enum.EnumType 方法), 294
__dir__() (unittest.mock.Mock 方法), 1656
__displayhook__() (在 sys 模块中), 1791
__doc__ (definition 属性), 96
__enter__() (contextmanager 方法), 88
__enter__() (winreg.PyHKEY 方法), 2029
__eq__() (实例方法), 36
__eq__() (email.charset.Charset 方法), 1207
__eq__() (email.header.Header 方法), 1205
__eq__() (memoryview 方法), 76
__eq__() (在 operator 模块中), 402
__excepthook__() (在 sys 模块中), 1791
__excepthook__() (在 threading 模块中), 886
__exit__() (contextmanager 方法), 88
__exit__() (winreg.PyHKEY 方法), 2029
__floor__() (fractions.Fraction 方法), 353
__floordiv__() (在 operator 模块中), 403
__format__, 16
__format__() (datetime.date 方法), 200
__format__() (datetime.datetime 方法), 209
__format__() (datetime.time 方法), 214
__format__() (enum.Enum 方法), 297

__format__() (fractions.Fraction 方法), 353
__format__() (ipaddress.IPv4Address 方法), 1425
__format__() (ipaddress.IPv6Address 方法), 1426
__fspath__() (os.PathLike 方法), 618
__future__, **2068**
  module, 1871
__ge__() (实例方法), 36
__ge__() (在 operator 模块中), 402
__getitem__() (email.headerregistry.HeaderRegistry 方法), 1184
__getitem__() (email.message.EmailMessage 方法), 1162
__getitem__() (email.message.Message 方法), 1197
__getitem__() (enum.EnumType 方法), 294
__getitem__() (mailbox.Mailbox 方法), 1223
__getitem__() (re.Match 方法), 136
__getitem__() (在 operator 模块中), 405
__getnewargs__() (object 方法), 475
__getnewargs_ex__() (object 方法), 475
__getstate__() (*copy* 协议), 479
__getstate__() (object 方法), 475
__gt__() (实例方法), 36
__gt__() (在 operator 模块中), 402
__iadd__() (在 operator 模块中), 408
__iand__() (在 operator 模块中), 408
__iconcat__() (在 operator 模块中), 408
__ifloordiv__() (在 operator 模块中), 408
__ilshift__() (在 operator 模块中), 408
__imatmul__() (在 operator 模块中), 408
__imod__() (在 operator 模块中), 408
__import__()
  built-in function, 31
__import__() (在 importlib 模块中), 1910
__imul__() (在 operator 模块中), 408
__index__() (在 operator 模块中), 403
__infer_variance__ (typing.TypeVar 属性), 1573
__init__() (asyncio.Future 方法), 1065
__init__() (asyncio.Task 方法), 1065
__init__() (difflib.HtmlDiff 方法), 143
__init__() (enum.Enum 方法), 296
__init__() (logging.Handler 方法), 707
__init_subclass__() (enum.Enum 方法), 296
__interactivehook__() (在 sys 模块中), 1800
__inv__() (在 operator 模块中), 403
__invert__() (在 operator 模块中), 403
__ior__() (在 operator 模块中), 408
__ipow__() (在 operator 模块中), 408
__irshift__() (在 operator 模块中), 408
__isub__() (在 operator 模块中), 409
__iter__() (container 方法), 42
__iter__() (enum.EnumType 方法), 294
__iter__() (iterator 方法), 43
__iter__() (mailbox.Mailbox 方法), 1222
__iter__() (unittest.TestSuite 方法), 1641
__itruediv__() (在 operator 模块中), 409
__ixor__() (在 operator 模块中), 409
__le__() (实例方法), 36
__le__() (在 operator 模块中), 402
__len__() (email.message.EmailMessage 方法), 1161
__len__() (email.message.Message 方法), 1197
__len__() (enum.EnumType 方法), 294
__len__() (mailbox.Mailbox 方法), 1223
__lshift__() (在 operator 模块中), 404
__lt__() (实例方法), 36
__lt__() (在 operator 模块中), 402
__main__
  module, 1821, 1907, 1908
__matmul__() (在 operator 模块中), 404
__members__ (enum.EnumType 属性), 294
__missing__(), 84
__missing__() (collections.defaultdict 方法), 247
__mod__() (在 operator 模块中), 404
__module__ (definition 属性), 96
__module__ (typing.NewType 属性), 1579
__module__ (typing.TypeAliasType 属性), 1577
__mul__() (在 operator 模块中), 404
__mutable_keys__ (typing.TypedDict 属性), 1583
__name__ (definition 属性), 96
__name__ (property 属性), 26
__name__ (typing.NewType 属性), 1579
__name__ (typing.ParamSpec 属性), 1576
__name__ (typing.TypeAliasType 属性), 1577
__name__ (typing.TypeVar 属性), 1572
__name__ (typing.TypeVarTuple 属性), 1575
__ne__() (实例方法), 36
__ne__() (email.charset.Charset 方法), 1207
__ne__() (email.header.Header 方法), 1205
__ne__() (在 operator 模块中), 402
__neg__() (在 operator 模块中), 404
__new__() (enum.Enum 方法), 297
__next__() (csv.csvreader 方法), 574
__next__() (iterator 方法), 43
__not__() (在 operator 模块中), 403
__notes__ (BaseException 属性), 100
__notes__ (traceback.TracebackException 属性), 1864
__optional_keys__ (typing.TypedDict 属性), 1582
__or__() (enum.Flag 方法), 299
__or__() (在 operator 模块中), 404
__origin__ (genericalias 属性), 92
__parameters__ (genericalias 属性), 92
__pos__() (在 operator 模块中), 404
__post_init__() (在 dataclasses 模块中), 1838

__pow__() (在 operator 模块中), 404
__qualname__ (definition 属性), 96
__readonly_keys__ (typing.TypedDict 属性), 1583
__reduce__() (object 方法), 476
__reduce_ex__() (object 方法), 476
__replace__() (*replace* 协议), 282
__repr__() (enum.Enum 方法), 297
__repr__() (multiprocessing.managers.BaseProxy 方法), 924
__repr__() (netrc.netrc 方法), 595
__required_keys__ (typing.TypedDict 属性), 1582
__reversed__() (enum.EnumType 方法), 294
__round__() (fractions.Fraction 方法), 353
__rshift__() (在 operator 模块中), 404
__setitem__() (email.message.EmailMessage 方法), 1162
__setitem__() (email.message.Message 方法), 1197
__setitem__() (mailbox.Mailbox 方法), 1222
__setitem__() (mailbox.Maildir 方法), 1226
__setitem__() (在 operator 模块中), 405
__setstate__() (*copy* 协议), 479
__setstate__() (object 方法), 475
__slots__, **2075**
__stderr__() (在 sys 模块中), 1808
__stdin__() (在 sys 模块中), 1808
__stdout__() (在 sys 模块中), 1808
__str__() (datetime.date 方法), 199
__str__() (datetime.datetime 方法), 209
__str__() (datetime.time 方法), 214
__str__() (email.charset.Charset 方法), 1207
__str__() (email.header.Header 方法), 1205
__str__() (email.headerregistry.Address 方法), 1185
__str__() (email.headerregistry.Group 方法), 1185
__str__() (email.message.EmailMessage 方法), 1161
__str__() (email.message.Message 方法), 1195
__str__() (enum.Enum 方法), 297
__str__() (multiprocessing.managers.BaseProxy 方法), 924
__sub__() (在 operator 模块中), 404
__subclasshook__() (abc.ABCMeta 方法), 1856
__supertype__ (typing.NewType 属性), 1579
__suppress_context__ (异常属性), 99
__suppress_context__ (BaseException 属性), 99
__suppress_context__ (traceback.TracebackException 属性), 1864
__total__ (typing.TypedDict 属性), 1582
__traceback__ (BaseException 属性), 100
__truediv__() (importlib.abc.Traversable 方法), 1917
__truediv__() (importlib.resources.abc.Traversable 方法), 1932
__truediv__() (在 operator 模块中), 404
__type_params__ (definition 属性), 96
__type_params__ (typing.TypeAliasType 属性), 1577
__unpacked__ (genericalias 属性), 92
__unraisablehook__() (在 sys 模块中), 1791
__value__ (typing.TypeAliasType 属性), 1577
__version__() (在 curses 模块中), 866
__xor__() (enum.Flag 方法), 300
__xor__() (在 operator 模块中), 404
_add_alias_() (enum.EnumType 方法), 294
_add_value_alias_() (enum.EnumType 方法), 294
align (ctypes.Structure 属性), 782
anonymous (ctypes.Structure 属性), 782
_asdict() (collections.somenamedtuple 方法), 249
_b_base_ (ctypes._CData 属性), 779
_b_needsfree_ (ctypes._CData 属性), 779
_callmethod() (multiprocessing.managers.BaseProxy 方法), 923
_CData (ctypes 中的类), 778
_CFuncPtr (ctypes 中的类), 773
_clear_internal_caches() (在 sys 模块中), 1789
_clear_type_cache() (在 sys 模块中), 1789
_current_exceptions() (在 sys 模块中), 1789
_current_frames() (在 sys 模块中), 1789
_debugmallocstats() (在 sys 模块中), 1790
_emscripten_info() (在 sys 模块中), 1790
_enablelegacywindowsfsencoding() (在 sys 模块中), 1807
_enter_task() (在 asyncio 模块中), 1066
_exit() (在 os 模块中), 662
_Feature (__future__ 中的类), 1871
_field_defaults (collections.somenamedtuple 属性), 250
_field_types (ast.AST 属性), 1944
fields (ctypes.Structure 属性), 782
_fields (ast.AST 属性), 1944
_fields (collections.somenamedtuple 属性), 250
_flush() (wsgiref.handlers.BaseHandler 方法), 1317
_generate_next_value_() (enum.Enum 方法), 296
_get_child_mock() (unittest.mock.Mock 方法), 1656
_get_preferred_schemes() (在 sysconfig 模块中), 1818
_getframe() (在 sys 模块中), 1797
_getframemodulename() (在 sys 模块中), 1797
_getvalue() (multiprocessing.managers.BaseProxy 方法), 924

_handle (ctypes.PyDLL 属性), 772
ignore (enum.Enum 属性), 295
_incompatible_extension_module_restrictions() (在 importlib.util 模块中), 1924
_is_gil_enabled() (在 sys 模块中), 1801
_is_interned() (在 sys 模块中), 1801
_leave_task() (在 asyncio 模块中), 1066
length (ctypes.Array 属性), 783
_locale
    module, 1448
_log (logging.LoggerAdapter 属性), 713
_make()(collections.somenamedtuple 类方法), 249
_makeResult() (unittest.TextTestRunner 方法), 1646
missing() (enum.Enum 方法), 296
name (enum.Enum 属性), 295
_name (ctypes.PyDLL 属性), 772
_numeric_repr_() (enum.Flag 方法), 300
_objects (ctypes._CData 属性), 779
order (enum.Enum 属性), 295
pack (ctypes.Structure 属性), 782
_parse() (gettext.NullTranslations 方法), 1443
_Pointer (ctypes 中的类), 783
_register_task() (在 asyncio 模块中), 1066
_replace() (collections.somenamedtuple 方法), 249
_setroot() (xml.etree.ElementTree.ElementTree 方法), 1269
_SimpleCData (ctypes 中的类), 779
_structure() (在 email.iterators 模块中), 1211
_thread
    module, 978
_tkinter
    module, 1504
type (ctypes._Pointer 属性), 784
type (ctypes.Array 属性), 783
_unregister_task()(在 asyncio 模块中), 1066
value (enum.Enum 属性), 295
_write() (wsgiref.handlers.BaseHandler 方法), 1317
_xoptions() (在 sys 模块中), 1810
{} (花括号)
    在字符串格式化中, 113
    在正则表达式中, 123
    格式字符串字面值形式, 57
| (竖线)
    operator, 37
    在正则表达式中, 124
~ (波浪号)
    operator, 37
    主目录扩展, 436
上下文管理协议, 88, **2066**
下标
    operation, 43
    赋值, 45
不带缓冲的 I/O, 23
中断点, 1541
临时
    文件, 448
    文件名, 448
事件排期, 969
二元信号量, 978
二进制
    字面值, 36
    数据, 打包, 167
二进制模式, 23
互联网, 1309
代码对象, 95, 487
信号量, 二元, 978
特殊
    method -- 方法, 2076
环境
    虚拟, 1773
环境变量
    BROWSER, 1309, 1310
    COLUMNS, 860
    COMSPEC, 669, 956
    DISPLAY, 1503
    HOME, 436, 1503
    HOMEDRIVE, 436
    HOMEPATH, 436
    IDLESTARTUP, 1544
    LANG, 1441, 1442, 1448, 1452
    LANGUAGE, 1441, 1442
    LC_ALL, 1441, 1442
    LC_MESSAGES, 1441, 1442
    LINES, 855, 860
    LOGNAME, 619, 851
    MANPAGER, 1598
    no_proxy, 1325
    PAGER, 1598
    PATH, 433, 462, 661, 666, 667, 676, 955, 1309, 1775, 1894
    PATHEXT, 462
    PYTHON_CPU_COUNT, 675, 910
    PYTHON_DOM, 1274
    PYTHON_GIL, 2069
    PYTHONASYNCIODEBUG, 1038, 1073, 1599
    PYTHONBREAKPOINT, 9, 1789, 1790
    PYTHONCASEOK, 32
    PYTHONCOERCECLOCALE, 617
    PYTHONDEVMODE, 1598
    PYTHONDONTWRITEBYTECODE, 1790
    PYTHONFAULTHANDLER, 1599, 1736
    PYTHONHOME, 1719, 1939, 1940
    PYTHONINTMAXSTRDIGITS, 97, 1800
    PYTHONIOENCODING, 616, 1808
    PYTHONLEGACYWINDOWSFSENCODING, 1807
    PYTHONLEGACYWINDOWSSTDIO, 1808
    PYTHONMALLOC, 1599
    PYTHONNOUSERSITE, 1895
    PYTHONPATH, 1719, 1802, 1939
    PYTHONPLATLIBDIR, 1939

PYTHONPYCACHEPREFIX, 1791
PYTHONSAFEPATH, 1802, 2061
PYTHONSTARTUP, 163, 982, 1544, 1800, 1895
PYTHONTRACEMALLOC, 1761, 1766
PYTHONTZPATH, 227, 230
PYTHONUNBUFFERED, 1808
PYTHONUSERBASE, 1895
PYTHONUSERSITE, 1719
PYTHONUTF8, 616, 617, 1808
PYTHONWARNDEFAULTENCODING, 679
PYTHONWARNINGS, 1599, 1827, 1828
SOURCE_DATE_EPOCH, 1991, 1993
SSLKEYLOGFILE, 1103
SystemRoot, 958
TEMP, 451
TERM, 859
TMP, 451
TMPDIR, 451
TZ, 698, 699
USER, 851
USERNAME, 436, 619, 851
USERPROFILE, 436
删除, 623
设置, 621
相关
网址, 1338
真值
value, 35
确定性性能分析, 1746
移位
操作, 37
简单邮件传输协议, 1374
粘贴, 1541
结构
C, 167
缓冲区大小, I/O, 23
缓冲协议
str (内置类), 49
二进制序列类型, 60
编解码器, 173
decode, 173
encode, 173
编译
内置函数, 95, 278
网址, 1338, 1347, 1392
相关, 1338
解析, 1338
自纪元以来的秒数, 690
虚拟
环境, 1773
解析
网址, 1338
解释器提示符, 1804
设置断点, 1541
语言
C, 36
调试, 1738
调试器, 887, 1540, 1798, 1805
配置 文件, 1741
赋值
slice -- 切片, 45
下标, 45
路径浏览器, 1538
转换
数字, 37
轻量级进程, 978
运算目标
dictionary -- 字典 type, 83
integer types, 37
list type, 45
mapping -- 映射 types, 83
sequence types, 43, 45
数字 types, 37
运行脚本, 1539
进程, 轻量级, 978
迭代器协议, 42
追踪函数, 887, 1798, 1805
配置
文件, 576
文件, path, 1894
文件, 调试器, 1741
配置信息, 1815
重复
operation, 43
错误
logging, 701
错误处理器名称
backslashreplace, 176
ignore, 176
namereplace, 177
replace, 176
strict, 176
surrogateescape, 176
surrogatepass, 177
xmlcharrefreplace, 177
集成开发环境, 1537
魔术
method -- 方法, 2071

A

-a
ast 命令行选项, 1976
pickletools 命令行选项, 2017
A() (在 re 模块中), 128
a2b_base64() (在 binascii 模块中), 1245
a2b_hex() (在 binascii 模块中), 1246
a2b_qp() (在 binascii 模块中), 1245
a2b_uu() (在 binascii 模块中), 1245
a85decode() (在 base64 模块中), 1243
a85encode() (在 base64 模块中), 1243
A_ALTCHARSET() (在 curses 模块中), 868
A_ATTRIBUTES() (在 curses 模块中), 869
A_BLINK() (在 curses 模块中), 868
A_BOLD() (在 curses 模块中), 868
A_CHARTEXT() (在 curses 模块中), 869
A_COLOR() (在 curses 模块中), 869

A_DIM() (在 curses 模块中), 868
A_HORIZONTAL() (在 curses 模块中), 868
A_INVIS() (在 curses 模块中), 868
A_ITALIC() (在 curses 模块中), 868
A_LEFT() (在 curses 模块中), 868
A_LOW() (在 curses 模块中), 868
A_NORMAL() (在 curses 模块中), 868
A_PROTECT() (在 curses 模块中), 868
A_REVERSE() (在 curses 模块中), 868
A_RIGHT() (在 curses 模块中), 868
A_STANDOUT() (在 curses 模块中), 868
A_TOP() (在 curses 模块中), 868
A_UNDERLINE() (在 curses 模块中), 868
A_VERTICAL() (在 curses 模块中), 868
abc
    module, 1855
ABCMeta (abc 中的类), 1856
ABC (abc 中的类), 1856
ABDAY_1() (在 locale 模块中), 1450
ABDAY_2() (在 locale 模块中), 1450
ABDAY_3() (在 locale 模块中), 1450
ABDAY_4() (在 locale 模块中), 1450
ABDAY_5() (在 locale 模块中), 1450
ABDAY_6() (在 locale 模块中), 1450
ABDAY_7() (在 locale 模块中), 1450
abiflags() (在 sys 模块中), 1787
ABMON_1() (在 locale 模块中), 1450
ABMON_2() (在 locale 模块中), 1450
ABMON_3() (在 locale 模块中), 1450
ABMON_4() (在 locale 模块中), 1450
ABMON_5() (在 locale 模块中), 1450
ABMON_6() (在 locale 模块中), 1450
ABMON_7() (在 locale 模块中), 1450
ABMON_8() (在 locale 模块中), 1450
ABMON_9() (在 locale 模块中), 1450
ABMON_10() (在 locale 模块中), 1450
ABMON_11() (在 locale 模块中), 1450
ABMON_12() (在 locale 模块中), 1450
abort() (asyncio.Barrier 方法), 1014
abort() (asyncio.DatagramTransport 方法), 1051
abort() (asyncio.WriteTransport 方法), 1051
abort() (ftplib.FTP 方法), 1361
abort() (threading.Barrier 方法), 897
abort() (在 os 模块中), 661
ABORT() (在 tkinter.messagebox 模块中), 1518
abort_clients() (asyncio.Server 方法), 1041
ABORTRETRYIGNORE() (在 tkinter.messagebox 模块中), 1519
above() (curses.panel.Panel 方法), 883
ABOVE_NORMAL_PRIORITY_CLASS() (在 subprocess 模块中), 963
abs()
    built-in function, 8
abs() (decimal.Context 方法), 338
abs() (在 operator 模块中), 403
absolute() (pathlib.Path 方法), 423
AbsoluteLinkError, 554
AbsolutePathError, 554
abspath() (在 os.path 模块中), 435
abstract base class -- 抽象基类, **2063**
AbstractAsyncContextManager (contextlib 中的类), 1842
AbstractBasicAuthHandler (urllib.request 中的类), 1325
AbstractChildWatcher (asyncio 中的类), 1062
abstractclassmethod() (在 abc 模块中), 1858
AbstractContextManager (contextlib 中的类), 1842
AbstractDigestAuthHandler (urllib.request 中的类), 1325
AbstractEventLoopPolicy (asyncio 中的类), 1061
AbstractEventLoop (asyncio 中的类), 1042
abstractmethod() (在 abc 模块中), 1857
abstractproperty() (在 abc 模块中), 1859
AbstractSet (typing 中的类), 1594
abstractstaticmethod() (在 abc 模块中), 1859
accept() (multiprocessing.connection.Listener 方法), 927
accept() (socket.socket 方法), 1091
access() (在 os 模块中), 636
accumulate() (在 itertools 模块中), 378
ACK() (在 curses.ascii 模块中), 880
aclose() (contextlib.AsyncExitStack 方法), 1850
aclosing() (在 contextlib 模块中), 1844
acos() (在 cmath 模块中), 323
acos() (在 math 模块中), 319
acosh() (在 cmath 模块中), 323
acosh() (在 math 模块中), 319
acquire() (_thread.lock 方法), 979
acquire() (asyncio.Condition 方法), 1012
acquire() (asyncio.Lock 方法), 1010
acquire() (asyncio.Semaphore 方法), 1013
acquire() (logging.Handler 方法), 707
acquire() (multiprocessing.Lock 方法), 914
acquire() (multiprocessing.RLock 方法), 914
acquire() (threading.Condition 方法), 893
acquire() (threading.Lock 方法), 891
acquire() (threading.RLock 方法), 891
acquire() (threading.Semaphore 方法), 894
ACS_BBSS() (在 curses 模块中), 875
ACS_BLOCK() (在 curses 模块中), 875
ACS_BOARD() (在 curses 模块中), 875
ACS_BSBS() (在 curses 模块中), 875
ACS_BSSB() (在 curses 模块中), 875
ACS_BSSS() (在 curses 模块中), 875
ACS_BTEE() (在 curses 模块中), 875
ACS_BULLET() (在 curses 模块中), 875
ACS_CKBOARD() (在 curses 模块中), 875
ACS_DARROW() (在 curses 模块中), 875
ACS_DEGREE() (在 curses 模块中), 875

ACS_DIAMOND() (在 curses 模块中), 875
ACS_GEQUAL() (在 curses 模块中), 875
ACS_HLINE() (在 curses 模块中), 875
ACS_LANTERN() (在 curses 模块中), 875
ACS_LARROW() (在 curses 模块中), 875
ACS_LEQUAL() (在 curses 模块中), 875
ACS_LLCORNER() (在 curses 模块中), 875
ACS_LRCORNER() (在 curses 模块中), 876
ACS_LTEE() (在 curses 模块中), 876
ACS_NEQUAL() (在 curses 模块中), 876
ACS_PI() (在 curses 模块中), 876
ACS_PLMINUS() (在 curses 模块中), 876
ACS_PLUS() (在 curses 模块中), 876
ACS_RARROW() (在 curses 模块中), 876
ACS_RTEE() (在 curses 模块中), 876
ACS_S1() (在 curses 模块中), 876
ACS_S3() (在 curses 模块中), 876
ACS_S7() (在 curses 模块中), 876
ACS_S9() (在 curses 模块中), 876
ACS_SBBS() (在 curses 模块中), 876
ACS_SBSB() (在 curses 模块中), 876
ACS_SBSS() (在 curses 模块中), 876
ACS_SSBB() (在 curses 模块中), 876
ACS_SSBS() (在 curses 模块中), 876
ACS_SSSB() (在 curses 模块中), 876
ACS_SSSS() (在 curses 模块中), 877
ACS_STERLING() (在 curses 模块中), 877
ACS_TTEE() (在 curses 模块中), 877
ACS_UARROW() (在 curses 模块中), 877
ACS_ULCORNER() (在 curses 模块中), 877
ACS_URCORNER() (在 curses 模块中), 877
ACS_VLINE() (在 curses 模块中), 877
ACTIONS (optparse.Option 属性), 849
Action (argparse 中的类), 801
action (optparse.Option 属性), 838
activate_stack_trampoline() (在 sys 模块中), 1806
active_children() (在 multiprocessing 模块中), 910
active_count() (在 threading 模块中), 885
actual() (tkinter.font.Font 方法), 1514
add() (decimal.Context 方法), 338
add() (frozenset 方法), 83
add() (graphlib.TopologicalSorter 方法), 306
add() (mailbox.Mailbox 方法), 1222
add() (mailbox.Maildir 方法), 1226
add() (pstats.Stats 方法), 1750
add() (tarfile.TarFile 方法), 558
add() (tkinter.ttk.Notebook 方法), 1526
add() (在 operator 模块中), 403
add_alias() (在 email.charset 模块中), 1207
add_alternative() (email.message.EmailMessage 方法), 1166
add_argument() (argparse.ArgumentParser 方法), 792
add_argument_group() (argparse.ArgumentParser 方法), 808
add_attachment() (email.message.EmailMessage 方法), 1166
add_cgi_vars() (wsgiref.handlers.BaseHandler 方法), 1318
add_charset() (在 email.charset 模块中), 1207
add_child_handler() (asyncio.AbstractChildWatcher 方法), 1062
add_codec() (在 email.charset 模块中), 1207
add_cookie_header() (http.cookiejar.CookieJar 方法), 1403
add_dll_directory() (在 os 模块中), 661
add_done_callback() (asyncio.Future 方法), 1046
add_done_callback() (asyncio.Task 方法), 999
add_done_callback() (concurrent.futures.Future 方法), 949
add_fallback() (gettext.NullTranslations 方法), 1443
add_flag() (mailbox.Maildir 方法), 1225
add_flag() (mailbox.MaildirMessage 方法), 1231
add_flag() (mailbox.mboxMessage 方法), 1233
add_flag() (mailbox.MMDFMessage 方法), 1236
add_folder() (mailbox.Maildir 方法), 1225
add_folder() (mailbox.MH 方法), 1228
add_get_handler() (email.contentmanager.ContentManager 方法), 1186
add_handler() (urllib.request.OpenerDirector 方法), 1327
add_header() (email.message.EmailMessage 方法), 1162
add_header() (email.message.Message 方法), 1197
add_header() (urllib.request.Request 方法), 1327
add_header() (wsgiref.headers.Headers 方法), 1314
add_history() (在 readline 模块中), 162
add_label() (mailbox.BabylMessage 方法), 1235
add_mutually_exclusive_group() (argparse.ArgumentParser 方法), 809
add_note() (BaseException 方法), 100
add_option() (optparse.OptionParser 方法), 837
add_parent() (urllib.request.BaseHandler 方法), 1328

add_password() (urllib.request.HTTPPasswordMgr 方法), 1331
add_password() (urllib.request.HTTPPasswordMgrWithPriorAuth 方法), 1331
add_reader() (asyncio.loop 方法), 1033
add_related() (email.message.EmailMessage 方法), 1166
add_section() (configparser.ConfigParser 方法), 589
add_section() (configparser.RawConfigParser 方法), 591
add_sequence() (mailbox.MHMessage 方法), 1234
add_set_handler() (email.contentmanager.ContentManager 方法), 1186
add_signal_handler() (asyncio.loop 方法), 1036
add_subparsers() (argparse.ArgumentParser 方法), 804
add_type() (mimetypes.MimeTypes 方法), 1241
add_type() (在 mimetypes 模块中), 1240
add_unredirected_header() (urllib.request.Request 方法), 1327
add_writer() (asyncio.loop 方法), 1033
addAsyncCleanup() (unittest.IsolatedAsyncioTestCase 方法), 1640
addaudithook() (在 sys 模块中), 1787
addch() (curses.window 方法), 861
addClassCleanup() (unittest.TestCase 类方法), 1639
addCleanup() (unittest.TestCase 方法), 1639
addcomponent() (turtle.Shape 方法), 1486
addDuration() (unittest.TestResult 方法), 1646
addError() (unittest.TestResult 方法), 1645
addExpectedFailure() (unittest.TestResult 方法), 1646
addFailure() (unittest.TestResult 方法), 1645
addfile() (tarfile.TarFile 方法), 558
addFilter() (logging.Handler 方法), 707
addFilter() (logging.Logger 方法), 706
addHandler() (logging.Logger 方法), 706
addinfourl (urllib.response 中的类), 1338
addLevelName() (在 logging 模块中), 714
addModuleCleanup() (在 unittest 模块中), 1649
addnstr() (curses.window 方法), 861
AddPackagePath() (在 modulefinder 模块中), 1906
addr_spec (email.headerregistry.Address 属性), 1185
Address_exclude() (ipaddress.IPv4Network 方法), 1429
Address_exclude() (ipaddress.IPv6Network 方法), 1431
address_family (socketserver.BaseServer 属性), 1387
address_string() (http.server.BaseHTTPRequestHandler 方法), 1395
addresses (email.headerregistry.AddressHeader 属性), 1182
addresses (email.headerregistry.Group 属性), 1185
AddressHeader (email.headerregistry 中的类), 1182
addressof() (在 ctypes 模块中), 776
AddressValueError, 1435
Address (email.headerregistry 中的类), 1184
address (email.headerregistry.SingleAddressHeader 属性), 1182
address (multiprocessing.connection.Listener 属性), 928
address (multiprocessing.managers.BaseManager 属性), 919
addshape() (在 turtle 模块中), 1484
addsitedir() (在 site 模块中), 1895
addSkip() (unittest.TestResult 方法), 1646
addstr() (curses.window 方法), 861
addSubTest() (unittest.TestResult 方法), 1646
addSuccess() (unittest.TestResult 方法), 1645
addTest() (unittest.TestSuite 方法), 1641
addTests() (unittest.TestSuite 方法), 1641
addTypeEqualityFunc() (unittest.TestCase 方法), 1637
addUnexpectedSuccess() (unittest.TestResult 方法), 1646
Add (ast 中的类), 1949
adjust_int_max_str_digits() (在 test.support 模块中), 1718
adjusted() (decimal.Decimal 方法), 330
adler32() (在 zlib 模块中), 525
AF_ALG() (在 socket 模块中), 1082
AF_CAN() (在 socket 模块中), 1081
AF_DIVERT() (在 socket 模块中), 1081
AF_HYPERV() (在 socket 模块中), 1083
AF_INET() (在 socket 模块中), 1079
AF_INET6() (在 socket 模块中), 1079
AF_LINK() (在 socket 模块中), 1082
AF_PACKET() (在 socket 模块中), 1082
AF_QIPCRTR() (在 socket 模块中), 1083
AF_RDS() (在 socket 模块中), 1082
AF_UNIX() (在 socket 模块中), 1079
AF_UNSPEC() (在 socket 模块中), 1079
AF_VSOCK() (在 socket 模块中), 1082
aifc
    module, 2055
aiter()

built-in function, 8
alarm() (在 signal 模块中), 1147
ALERT_DESCRIPTION_HANDSHAKE_FAILURE() (在 ssl 模块中), 1112
ALERT_DESCRIPTION_INTERNAL_ERROR() (在 ssl 模块中), 1112
AlertDescription (ssl 中的类), 1112
algorithms_available() (在 hashlib 模块中), 601
algorithms_guaranteed() (在 hashlib 模块中), 601
algorithm (sys.hash_info 属性), 1799
Alias
Generic, 89
alias (*pdb command*), 1744
alias (ast 中的类), 1957
alignment() (在 ctypes 模块中), 776
alive (weakref.finalize 属性), 272
all()
built-in function, 8
ALL_COMPLETED() (在 asyncio 模块中), 996
ALL_COMPLETED() (在 concurrent.futures 模块中), 950
all_errors() (在 ftplib 模块中), 1365
all_features() (在 xml.sax.handler 模块中), 1291
all_frames (tracemalloc.Filter 属性), 1767
all_properties() (在 xml.sax.handler 模块中), 1292
all_suffixes() (在 importlib.machinery 模块中), 1918
all_tasks() (在 asyncio 模块中), 998
allocate_lock() (在 _thread 模块中), 979
allow_reuse_address (socketserver.BaseServer 属性), 1387
allowed_domains() (http.cookiejar.DefaultCookiePolicy 方法), 1406
alt() (在 curses.ascii 模块中), 882
ALT_DIGITS() (在 locale 模块中), 1451
--altinstall
ensurepip 命令行选项, 1772
altsep() (在 os 模块中), 675
altzone() (在 time 模块中), 701
ALWAYS_EQ() (在 test.support 模块中), 1711
ALWAYS_TYPED_ACTIONS (optparse.Option 属性), 849
AmbiguousOptionError, 851
AMPER() (在 token 模块中), 1982
AMPEREQUAL() (在 token 模块中), 1983
Anchor (importlib.resources 中的类), 1928
anchor (pathlib.PurePath 属性), 416
and
operator, 35, 36
and_() (在 operator 模块中), 403
android_ver() (在 platform 模块中), 745
And (ast 中的类), 1950
anext()
built-in function, 8
AnnAssign (ast 中的类), 1954
--annotate
pickletools 命令行选项, 2017
Annotated() (在 typing 模块中), 1566
annotation -- 标注
类型标注; 类型提示 type hint, 89
annotation -- 标注, **2063**
annotation (inspect.Parameter 属性), 1883
ANNOTATION (symtable.SymbolTableType 属性), 1976
answer_challenge() (在 multiprocessing.connection 模块中), 927
anticipate_failure() (在 test.support 模块中), 1715
any()
built-in function, 9
Any() (在 typing 模块中), 1560
ANY() (在 unittest.mock 模块中), 1682
ANY_CONTIGUOUS (inspect.BufferFlags 属性), 1892
AnyStr() (在 typing 模块中), 1560
api_version() (在 sys 模块中), 1809
apilevel() (在 sqlite3 模块中), 499
apop() (poplib.POP3 方法), 1366
append() (序列方法), 45
append() (array.array 方法), 267
append() (collections.deque 方法), 244
append() (email.header.Header 方法), 1204
append() (imaplib.IMAP4 方法), 1369
append() (xml.etree.ElementTree.Element 方法), 1267
append_history_file() (在 readline 模块中), 162
appendChild() (xml.dom.Node 方法), 1276
appendleft() (collections.deque 方法), 244
AppleFrameworkLoader (importlib.machinery 中的类), 1922
application_uri() (在 wsgiref.util 模块中), 1312
apply() (multiprocessing.pool.Pool 方法), 925
apply_async() (multiprocessing.pool.Pool 方法), 925
apply_defaults() (inspect.BoundArguments 方法), 1885
APRIL() (在 calendar 模块中), 236
architecture() (在 platform 模块中), 742
archive (zipimport.zipimporter 属性), 1902
AREGTYPE() (在 tarfile 模块中), 554
aRepr() (在 reprlib 模块中), 288
argparse
module, 785
args (*pdb command*), 1743
args_from_interpreter_flags() (在 test.support 模块中), 1713

args (BaseException 属性), 100
args (functools.partial 属性), 402
args (inspect.BoundArguments 属性), 1885
args (subprocess.CompletedProcess 属性), 952
args (subprocess.Popen 属性), 960
args (typing.ParamSpec 属性), 1575
argtypes (ctypes._CFuncPtr 属性), 773
argument -- 参数, **2063**
ArgumentDefaultsHelpFormatter(argparse 中的类), 788
ArgumentError, 774, 812
ArgumentParser (argparse 中的类), 786
arguments (ast 中的类), 1968
arguments (inspect.BoundArguments 属性), 1884
ArgumentTypeError, 812
argv() (在 sys 模块中), 1788
arg (ast 中的类), 1968
arithmetic, 36
ArithmeticError, 100
array
    module, 60, 266
Array() (multiprocessing.managers.SyncManager 方法), 920
ARRAY() (在 ctypes 模块中), 783
Array() (在 multiprocessing 模块中), 916
Array() (在 multiprocessing.sharedctypes 模块中), 916
arraysize (sqlite3.Cursor 属性), 512
array (array 中的类), 267
Array (ctypes 中的类), 783
as_bytes() (email.message.EmailMessage 方法), 1161
as_bytes() (email.message.Message 方法), 1195
as_completed() (在 asyncio 模块中), 996
as_completed() (在 concurrent.futures 模块中), 950
as_file() (在 importlib.resources 模块中), 1928
as_integer_ratio() (decimal.Decimal 方法), 330
as_integer_ratio() (float 方法), 40
as_integer_ratio() (fractions.Fraction 方法), 352
as_integer_ratio() (int 方法), 40
as_posix() (pathlib.PurePath 方法), 418
as_string() (email.message.EmailMessage 方法), 1160
as_string() (email.message.Message 方法), 1194
as_tuple() (decimal.Decimal 方法), 330
as_uri() (pathlib.Path 方法), 423
ascii()
    built-in function, 9
ascii() (在 curses.ascii 模块中), 882
ASCII() (在 re 模块中), 128
ascii_letters() (在 string 模块中), 111
ascii_lowercase() (在 string 模块中), 111
ascii_uppercase() (在 string 模块中), 111
asctime() (在 time 模块中), 691
asdict() (在 dataclasses 模块中), 1836
asin() (在 cmath 模块中), 323
asin() (在 math 模块中), 319
asinh() (在 cmath 模块中), 323
asinh() (在 math 模块中), 319
askcolor() (在 tkinter.colorchooser 模块中), 1513
askdirectory() (在 tkinter.filedialog 模块中), 1516
askfloat() (在 tkinter.simpledialog 模块中), 1514
askinteger() (在 tkinter.simpledialog 模块中), 1514
askokcancel() (在 tkinter.messagebox 模块中), 1518
askopenfile() (在 tkinter.filedialog 模块中), 1515
askopenfilename() (在 tkinter.filedialog 模块中), 1515
askopenfilenames() (在 tkinter.filedialog 模块中), 1515
askopenfiles() (在 tkinter.filedialog 模块中), 1515
askquestion() (在 tkinter.messagebox 模块中), 1518
askretrycancel() (在 tkinter.messagebox 模块中), 1518
asksaveasfile() (在 tkinter.filedialog 模块中), 1515
asksaveasfilename() (在 tkinter.filedialog 模块中), 1515
askstring() (在 tkinter.simpledialog 模块中), 1514
askyesno() (在 tkinter.messagebox 模块中), 1518
askyesnocancel() (在 tkinter.messagebox 模块中), 1518
assert
    statement -- 语句, 101
assert_any_await() (unittest.mock.AsyncMock 方法), 1664
assert_any_call() (unittest.mock.Mock 方法), 1654
assert_awaited() (unittest.mock.AsyncMock 方法), 1662
assert_awaited_once() (unittest.mock.AsyncMock 方法), 1663
assert_awaited_once_with() (unittest.mock.AsyncMock 方法), 1663
assert_awaited_with() (unittest.mock.AsyncMock 方法),

1663
assert_called() (unittest.mock.Mock 方法), 1653
assert_called_once() (unittest.mock.Mock 方法), 1653
assert_called_once_with() (unittest.mock.Mock 方法), 1654
assert_called_with() (unittest.mock.Mock 方法), 1654
assert_has_awaits() (unittest.mock.AsyncMock 方法), 1664
assert_has_calls() (unittest.mock.Mock 方法), 1654
assert_never() (在 typing 模块中), 1584
assert_not_awaited() (unittest.mock.AsyncMock 方法), 1664
assert_not_called() (unittest.mock.Mock 方法), 1655
assert_python_failure() (在 test.support.script_helper 模块中), 1719
assert_python_ok() (在 test.support.script_helper 模块中), 1719
assert_type() (在 typing 模块中), 1584
assertAlmostEqual() (unittest.TestCase 方法), 1636
assertCountEqual() (unittest.TestCase 方法), 1637
assertDictEqual() (unittest.TestCase 方法), 1638
assertEqual() (unittest.TestCase 方法), 1633
assertFalse() (unittest.TestCase 方法), 1633
assertGreater() (unittest.TestCase 方法), 1637
assertGreaterEqual() (unittest.TestCase 方法), 1637
assertIn() (unittest.TestCase 方法), 1633
assertInBytecode() (test.support.bytecode_helper.BytecodeTestCase 方法), 1720
AssertionError, 101
assertIs() (unittest.TestCase 方法), 1633
assertIsInstance() (unittest.TestCase 方法), 1633
assertIsNone() (unittest.TestCase 方法), 1633
assertIsNot() (unittest.TestCase 方法), 1633
assertIsNotNone() (unittest.TestCase 方法), 1633
assertLess() (unittest.TestCase 方法), 1637
assertLessEqual() (unittest.TestCase 方法), 1637
assertListEqual() (unittest.TestCase 方法), 1638
assertLogs() (unittest.TestCase 方法), 1635
assertMultiLineEqual() (unittest.TestCase 方法), 1637
assertNoLogs() (unittest.TestCase 方法), 1636
assertNotAlmostEqual() (unittest.TestCase 方法), 1636
assertNotEqual() (unittest.TestCase 方法), 1633
assertNotIn() (unittest.TestCase 方法), 1633
assertNotInBytecode() (test.support.bytecode_helper.BytecodeTestCase 方法), 1720
assertNotIsInstance() (unittest.TestCase 方法), 1633
assertNotRegex() (unittest.TestCase 方法), 1637
assertRaises() (unittest.TestCase 方法), 1634
assertRaisesRegex() (unittest.TestCase 方法), 1634
assertRegex() (unittest.TestCase 方法), 1637
assertSequenceEqual() (unittest.TestCase 方法), 1637
assertSetEqual() (unittest.TestCase 方法), 1638
assertTrue() (unittest.TestCase 方法), 1633
assertTupleEqual() (unittest.TestCase 方法), 1638
assertWarns() (unittest.TestCase 方法), 1635
assertWarnsRegex() (unittest.TestCase 方法), 1635
Assert (ast 中的类), 1956
Assign (ast 中的类), 1954
ast
    test.support.ast_helper.ASTTestCase, 1941
astimezone() (datetime.datetime 方法), 206
astuple() (在 dataclasses 模块中), 1836
ast 命令行选项
    -a, 1976
    -h, 1975
    --help, 1975
    -i, 1976
    --include-attributes, 1976
    --indent, 1976
    -m, 1975
    --mode, 1975
    --no-type-comments, 1976
AST (ast 中的类), 1944
AsyncContextDecorator (contextlib 中的类),

1848
asynccontextmanager() (在 contextlib 模块中), 1843
AsyncContextManager (typing 中的类), 1597
AsyncExitStack (contextlib 中的类), 1850
AsyncFor (ast 中的类), 1971
AsyncFunctionDef (ast 中的类), 1971
AsyncGeneratorType() (在 types 模块中), 278
AsyncGenerator(collections.abc 中的类), 258
AsyncGenerator (typing 中的类), 1595
asynchat
    module, 2055
asynchronous context manager -- 异步上下文管理器, **2064**
asynchronous generator -- 异步生成器, **2064**
asynchronous generator iterator -- 异步生成器迭代器, **2064**
asynchronous iterable -- 异步可迭代对象, **2064**
asynchronous iterator -- 异步迭代器, **2064**
asyncio
    module, 981
asyncio.subprocess.DEVNULL (内置变量), 1016
asyncio.subprocess.PIPE (内置变量), 1016
asyncio.subprocess.Process (内置类), 1016
asyncio.subprocess.STDOUT (内置变量), 1016
AsyncIterable (collections.abc 中的类), 258
AsyncIterable (typing 中的类), 1595
AsyncIterator (collections.abc 中的类), 258
AsyncIterator (typing 中的类), 1595
AsyncMock (unittest.mock 中的类), 1662
asyncore
    module, 2055
AsyncResult (multiprocessing.pool 中的类), 926
asyncSetUp() (unittest.IsolatedAsyncioTestCase 方法), 1640
asyncTearDown() (unittest.IsolatedAsyncioTestCase 方法), 1640
AsyncWith (ast 中的类), 1971
AT() (在 token 模块中), 1983
at_eof() (asyncio.StreamReader 方法), 1005
atan() (在 cmath 模块中), 323
atan() (在 math 模块中), 319
atan2() (在 math 模块中), 319
atanh() (在 cmath 模块中), 323
atanh() (在 math 模块中), 319
ATEQUAL() (在 token 模块中), 1984
atexit
    module, 1860
atexit (weakref.finalize 属性), 272
atof() (在 locale 模块中), 1453
atoi() (在 locale 模块中), 1453
attach() (email.message.Message 方法), 1195
attach_loop() (asyncio.AbstractChildWatcher 方法), 1062
attach_mock() (unittest.mock.Mock 方法), 1656
attempted (doctest.TestResults 属性), 1617
AttlistDeclHandler() (xml.parsers.expat.xmlparser 方法), 1303
attrgetter() (在 operator 模块中), 405
attribute -- 属性, **2064**
AttributeError, 101
AttributesImpl (xml.sax.xmlreader 中的类), 1296
AttributesNSImpl(xml.sax.xmlreader 中的类), 1296
attributes (xml.dom.Node 属性), 1275
Attribute (ast 中的类), 1951
attrib (xml.etree.ElementTree.Element 属性), 1267
attroff() (curses.window 方法), 861
attron() (curses.window 方法), 861
attrset() (curses.window 方法), 861
audioop
    module, 2056
audit() (在 sys 模块中), 1788
AugAssign (ast 中的类), 1955
AUGUST() (在 calendar 模块中), 236
auth() (ftplib.FTP_TLS 方法), 1364
auth() (smtplib.SMTP 方法), 1377
authenticate() (imaplib.IMAP4 方法), 1369
AuthenticationError, 907
authenticators() (netrc.netrc 方法), 595
authkey (multiprocessing.Process 属性), 906
autocommit (sqlite3.Connection 属性), 509
autorange() (timeit.Timer 方法), 1755
auto (enum 中的类), 304
available_timezones() (在 zoneinfo 模块中), 230
avoids_symlink_attacks (shutil.rmtree 属性), 461
await_args_list (unittest.mock.AsyncMock 属性), 1665
await_args (unittest.mock.AsyncMock 属性), 1665
await_count (unittest.mock.AsyncMock 属性), 1664
awaitable -- 可等待对象, **2064**
Awaitable (collections.abc 中的类), 258
Awaitable (typing 中的类), 1595
Await (ast 中的类), 1971

B

-b
    compileall 命令行选项, 1993
    http.server 命令行选项, 1397
    unittest 命令行选项, 1625
b2a_base64() (在 binascii 模块中), 1245
b2a_hex() (在 binascii 模块中), 1246
b2a_qp() (在 binascii 模块中), 1245
b2a_uu() (在 binascii 模块中), 1245

b16decode() (在 base64 模块中), 1243
b16encode() (在 base64 模块中), 1243
b32decode() (在 base64 模块中), 1242
b32encode() (在 base64 模块中), 1242
b32hexdecode() (在 base64 模块中), 1243
b32hexencode() (在 base64 模块中), 1243
b64decode() (在 base64 模块中), 1242
b64encode() (在 base64 模块中), 1242
b85decode() (在 base64 模块中), 1243
b85encode() (在 base64 模块中), 1243
BabylMessage (mailbox 中的类), 1234
Babyl (mailbox 中的类), 1229
back() (在 turtle 模块中), 1464
backend() (在 readline 模块中), 161
backslashreplace
    错误处理器名称, 176
backslashreplace_errors() (在 codecs 模块中), 178
backup() (sqlite3.Connection 方法), 506
backward() (在 turtle 模块中), 1464
BadGzipFile, 529
BadOptionError, 850
BadStatusLine, 1353
BadZipFile, 541
BadZipfile, 541
Barrier() (multiprocessing.managers.SyncManager 方法), 919
Barrier (asyncio 中的类), 1013
Barrier (multiprocessing 中的类), 913
Barrier (threading 中的类), 897
base64
    encoding, 1241
    module, 1241, 1244
base_exec_prefix() (在 sys 模块中), 1788
base_prefix() (在 sys 模块中), 1788
BaseCGIHandler (wsgiref.handlers 中的类), 1317
BaseCookie (http.cookies 中的类), 1398
BaseException, 100
BaseExceptionGroup, 108
BaseHandler (urllib.request 中的类), 1324
BaseHandler (wsgiref.handlers 中的类), 1317
BaseHeader (email.headerregistry 中的类), 1181
BaseHTTPRequestHandler(http.server 中的类), 1392
BaseManager(multiprocessing.managers 中的类), 918
basename() (在 os.path 模块中), 435
BaseProtocol (asyncio 中的类), 1052
BaseProxy(multiprocessing.managers 中的类), 923
BaseRequestHandler (socketserver 中的类), 1388
BaseRotatingHandler(logging.handlers 中的类), 731
BaseSelector (selectors 中的类), 1141
BaseServer (socketserver 中的类), 1386
BaseTransport (asyncio 中的类), 1049
basicConfig() (在 logging 模块中), 715
BasicContext() (在 decimal 模块中), 336
BasicInterpolation (configparser 中的类), 580
batched() (在 itertools 模块中), 379
baudrate() (在 curses 模块中), 854
bbox() (tkinter.ttk.Treeview 方法), 1530
BDADDR_ANY() (在 socket 模块中), 1083
BDADDR_LOCAL() (在 socket 模块中), 1083
bdb
    module, 1731, 1738
BdbQuit, 1731
Bdb (bdb 中的类), 1732
BDFL, **2064**
beep() (在 curses 模块中), 854
Beep() (在 winsound 模块中), 2030
BEFORE_ASYNC_WITH (*opcode*), 2004
BEFORE_WITH (*opcode*), 2006
begin_fill() (在 turtle 模块中), 1473
begin_poly() (在 turtle 模块中), 1478
BEL() (在 curses.ascii 模块中), 880
below() (curses.panel.Panel 方法), 883
BELOW_NORMAL_PRIORITY_CLASS() (在 subprocess 模块中), 963
benchmarking, 693, 698
--best
    gzip 命令行选项, 531
betavariate() (在 random 模块中), 357
bgcolor() (在 turtle 模块中), 1479
bgpic() (在 turtle 模块中), 1479
bidirectional()(在 unicodedata 模块中), 158
bigaddrspacetest()(在 test.support 模块中), 1716
BigEndianStructure (ctypes 中的类), 782
BigEndianUnion (ctypes 中的类), 782
bigmemtest() (在 test.support 模块中), 1716
bin()
    built-in function, 9
binary file -- 二进制文件, **2064**
BINARY_OP (*opcode*), 2002
BINARY_SLICE (*opcode*), 2003
BINARY_SUBSCR (*opcode*), 2003
BinaryIO (typing 中的类), 1584
Binary (xmlrpc.client 中的类), 1413
binascii
    module, 1244
--bind
    http.server 命令行选项, 1397
bind (部件), 1511
bind() (inspect.Signature 方法), 1882
bind() (socket.socket 方法), 1091
bind_partial() (inspect.Signature 方法), 1882
bind_port()(在 test.support.socket_helper 模块中), 1719
bind_textdomain_codeset() (在 locale 模块中), 1455

bind_unix_socket() (在 test.support.socket_helper 模块中), 1719
bindtextdomain() (在 gettext 模块中), 1441
bindtextdomain() (在 locale 模块中), 1455
binomialvariate() (在 random 模块中), 357
BinOp (ast 中的类), 1949
bisect
    module, 263
bisect() (在 bisect 模块中), 263
bisect_left() (在 bisect 模块中), 263
bisect_right() (在 bisect 模块中), 263
bit_count() (int 方法), 38
bit_length() (int 方法), 38
BitAnd (ast 中的类), 1949
BitOr (ast 中的类), 1949
bits_per_digit (sys.int_info 属性), 1800
bitwise
    操作, 37
BitXor (ast 中的类), 1949
bk() (在 turtle 模块中), 1464
bkgd() (curses.window 方法), 861
bkgdset() (curses.window 方法), 861
blake2b() (在 hashlib 模块中), 603
blake2b, blake2s, 603
blake2b.MAX_DIGEST_SIZE() (在 hashlib 模块中), 604
blake2b.MAX_KEY_SIZE() (在 hashlib 模块中), 604
blake2b.PERSON_SIZE() (在 hashlib 模块中), 604
blake2b.SALT_SIZE() (在 hashlib 模块中), 604
blake2s() (在 hashlib 模块中), 603
blake2s.MAX_DIGEST_SIZE() (在 hashlib 模块中), 604
blake2s.MAX_KEY_SIZE() (在 hashlib 模块中), 604
blake2s.PERSON_SIZE() (在 hashlib 模块中), 604
blake2s.SALT_SIZE() (在 hashlib 模块中), 604
BLKTYPE() (在 tarfile 模块中), 555
blobopen() (sqlite3.Connection 方法), 501
Blob (sqlite3 中的类), 513
block_on_close(socketserver.ThreadingMixIn 属性), 1385
block_size (hmac.HMAC 属性), 611
blocked_domains() (http.cookiejar.DefaultCookiePolicy 方法), 1406
BlockingIOError, 106, 679
blocksize (http.client.HTTPConnection 属性), 1356
body() (tkinter.simpledialog.Dialog 方法), 1515
body_encode() (email.charset.Charset 方法), 1207
body_encoding (email.charset.Charset 属性), 1206
body_line_iterator() (在 email.iterators 模块中), 1211
BOLD() (在 tkinter.font 模块中), 1513
BOM() (在 codecs 模块中), 176
BOM_BE() (在 codecs 模块中), 176
BOM_LE() (在 codecs 模块中), 176
BOM_UTF8() (在 codecs 模块中), 176
BOM_UTF16() (在 codecs 模块中), 176
BOM_UTF16_BE() (在 codecs 模块中), 176
BOM_UTF16_LE() (在 codecs 模块中), 176
BOM_UTF32() (在 codecs 模块中), 176
BOM_UTF32_BE() (在 codecs 模块中), 176
BOM_UTF32_LE() (在 codecs 模块中), 176
BOOLEAN_STATES (configparser.ConfigParser 属性), 585
BooleanOptionalAction(argparse 中的类), 794
BoolOp (ast 中的类), 1950
bool (内置类), 9
bootstrap() (在 ensurepip 模块中), 1772
border() (curses.window 方法), 861
borrowed reference -- 借入引用, **2064**
bottom() (curses.panel.Panel 方法), 883
bottom_panel()(在 curses.panel 模块中), 883
BoundArguments (inspect 中的类), 1884
BoundaryError, 1179
BoundedSemaphore() (multiprocessing.managers.SyncManager 方法), 919
BoundedSemaphore (asyncio 中的类), 1013
BoundedSemaphore (multiprocessing 中的类), 913
BoundedSemaphore (threading 中的类), 895
box() (curses.window 方法), 862
bpbynumber (bdb.Breakpoint 属性), 1732
bpformat() (bdb.Breakpoint 方法), 1731
bplist (bdb.Breakpoint 属性), 1732
bpprint() (bdb.Breakpoint 方法), 1731
BRANCH (*monitoring event*), 1811
break (*pdb command*), 1741
break_anywhere() (bdb.Bdb 方法), 1733
break_here() (bdb.Bdb 方法), 1733
break_long_words (textwrap.TextWrapper 属性), 157
break_on_hyphens (textwrap.TextWrapper 属性), 157
breakpoint()
    built-in function, 9
breakpointhook() (在 sys 模块中), 1789
Breakpoint (bdb 中的类), 1731
Break (ast 中的类), 1958
broadcast_address (ipaddress.IPv4Network 属性), 1428
broadcast_address (ipaddress.IPv6Network 属性), 1431
BrokenBarrierError, 897, 1014
BrokenExecutor, 950
BrokenPipeError, 106
BrokenProcessPool, 951

BrokenThreadPool, 951
broken (asyncio.Barrier 属性), 1014
broken (threading.Barrier 属性), 897
BROWSER, 1309, 1310
BS() (在 curses.ascii 模块中), 880
BsdDbShelf (shelve 中的类), 486
--buffer
    unittest 命令行选项, 1625
buffer_info() (array.array 方法), 267
buffer_size (xml.parsers.expat.xmlparser 属性), 1302
buffer_text (xml.parsers.expat.xmlparser 属性), 1302
buffer_updated() (asyncio.BufferedProtocol 方法), 1054
buffer_used (xml.parsers.expat.xmlparser 属性), 1302
BufferedIOBase (io 中的类), 683
BufferedProtocol (asyncio 中的类), 1052
BufferedRandom (io 中的类), 686
BufferedReader (io 中的类), 685
BufferedRWPair (io 中的类), 686
BufferedWriter (io 中的类), 685
BufferError, 100
BufferFlags (inspect 中的类), 1892
BufferingFormatter (logging 中的类), 709
BufferingHandler (logging.handlers 中的类), 738
BufferTooShort, 907
Buffer (collections.abc 中的类), 258
buffer (io.TextIOBase 属性), 686
buffer (unittest.TestResult 属性), 1644
buf(multiprocessing.shared_memory.SharedMemory 属性), 940
BUILD_CONST_KEY_MAP (*opcode*), 2008
BUILD_LIST (*opcode*), 2007
BUILD_MAP (*opcode*), 2007
build_opener() (在 urllib.request 模块中), 1323
BUILD_SET (*opcode*), 2007
BUILD_SLICE (*opcode*), 2012
BUILD_STRING (*opcode*), 2008
BUILD_TUPLE (*opcode*), 2007
built-in function
    __import__(), 31
    abs(), 8
    aiter(), 8
    all(), 8
    anext(), 8
    any(), 9
    ascii(), 9
    bin(), 9
    breakpoint(), 9
    callable(), 10
    chr(), 10
    classmethod(), 10
    compile(), 11
    delattr(), 12
    dir(), 12
    divmod(), 13
    enumerate(), 13
    eval(), 14
    exec(), 14
    filter(), 15
    format(), 16
    getattr(), 17
    globals(), 17
    hasattr(), 17
    hash(), 17
    help(), 17
    hex(), 17
    id(), 18
    input(), 18
    isinstance(), 19
    issubclass(), 19
    iter(), 19
    len(), 20
    locals(), 20
    map(), 20
    max(), 20
    min(), 21
    multiprocessing.Manager(), 918
    next(), 21
    oct(), 21
    open(), 22
    ord(), 24
    pow(), 24
    print(), 24
    property.deleter(), 25
    property.getter(), 25
    property.setter(), 25
    repr(), 26
    reversed(), 26
    round(), 26
    setattr(), 27
    sorted(), 27
    staticmethod(), 28
    sum(), 28
    vars(), 30
    zip(), 30
builtin_module_names()(在 sys 模块中), 1788
BuiltinFunctionType()(在 types 模块中), 278
BuiltinImporter (importlib.machinery 中的类), 1918
BuiltinMethodType() (在 types 模块中), 278
builtins
    module, 31, 1820
busy_retry() (在 test.support 模块中), 1712
BUTTON_ALT() (在 curses 模块中), 877
BUTTON_CTRL() (在 curses 模块中), 877
BUTTON_SHIFT() (在 curses 模块中), 877
buttonbox() (tkinter.simpledialog.Dialog 方法), 1515
BUTTONn_CLICKED() (在 curses 模块中), 877
BUTTONn_DOUBLE_CLICKED()(在 curses 模块中), 877

BUTTONn_PRESSED() (在 curses 模块中), 877
BUTTONn_RELEASED() (在 curses 模块中), 877
BUTTONn_TRIPLE_CLICKED() (在 curses 模块中), 877
bye() (在 turtle 模块中), 1485
byref() (在 ctypes 模块中), 776
bytearray
    object -- 对象, 45, 60, 61
    插值, 72
    方法, 62
    格式化, 72
bytearray (内置类), 61
bytecode -- 字节码, **2065**
BYTECODE_SUFFIXES() (在 importlib.machinery 模块中), 1918
Bytecode.codeobj() (在 dis 模块中), 1997
Bytecode.first_line() (在 dis 模块中), 1997
BytecodeTestCase(test.support.bytecode_helper 中的类), 1720
Bytecode (dis 中的类), 1997
byteorder() (在 sys 模块中), 1788
bytes-like object -- 字节型对象, **2065**
bytes_le (uuid.UUID 属性), 1381
bytes_warning (sys.flags 属性), 1793
BytesFeedParser (email.parser 中的类), 1168
BytesGenerator (email.generator 中的类), 1171
BytesHeaderParser (email.parser 中的类), 1169
BytesIO (io 中的类), 684
BytesParser (email.parser 中的类), 1168
ByteString (collections.abc 中的类), 257
ByteString (typing 中的类), 1594
byteswap() (array.array 方法), 267
BytesWarning, 107
bytes (uuid.UUID 属性), 1381
bytes (内置类), 60
bz2
    module, 532
BZ2Compressor (bz2 中的类), 534
BZ2Decompressor (bz2 中的类), 534
BZ2File (bz2 中的类), 532

C

C
    结构, 167
    语言, 36
-C
    dis 命令行选项, 1997
    trace 命令行选项, 1759
-c
    calendar 命令行选项, 238
    idle 命令行选项, 1544
    pdb 命令行选项, 1739
    random 命令行选项, 362
    tarfile 命令行选项, 565
    trace 命令行选项, 1759
    unittest 命令行选项, 1625
    zipapp 命令行选项, 1783
    zipfile 命令行选项, 551
C 连续, 2066
C14NWriterTarget(xml.etree.ElementTree 中的类), 1271
c_bool (ctypes 中的类), 781
c_byte (ctypes 中的类), 780
c_char_p (ctypes 中的类), 780
c_char (ctypes 中的类), 780
C_CONTIGUOUS (inspect.BufferFlags 属性), 1892
c_contiguous (memoryview 属性), 81
c_double (ctypes 中的类), 780
c_float (ctypes 中的类), 780
c_int8 (ctypes 中的类), 780
c_int16 (ctypes 中的类), 780
c_int32 (ctypes 中的类), 780
c_int64 (ctypes 中的类), 780
c_int (ctypes 中的类), 780
c_longdouble (ctypes 中的类), 780
c_longlong (ctypes 中的类), 780
c_long (ctypes 中的类), 780
C_RAISE (*monitoring event*), 1811
C_RETURN (*monitoring event*), 1811
c_short (ctypes 中的类), 780
c_size_t (ctypes 中的类), 780
c_ssize_t (ctypes 中的类), 780
c_time_t (ctypes 中的类), 780
c_ubyte (ctypes 中的类), 781
c_uint8 (ctypes 中的类), 781
c_uint16 (ctypes 中的类), 781
c_uint32 (ctypes 中的类), 781
c_uint64 (ctypes 中的类), 781
c_uint (ctypes 中的类), 781
c_ulonglong (ctypes 中的类), 781
c_ulong (ctypes 中的类), 781
c_ushort (ctypes 中的类), 781
c_void_p (ctypes 中的类), 781
c_wchar_p (ctypes 中的类), 781
c_wchar (ctypes 中的类), 781
CACHE (*opcode*), 2002
cache() (在 functools 模块中), 393
cache_from_source() (在 importlib.util 模块中), 1923
cached_property() (在 functools 模块中), 393
cached (importlib.machinery.ModuleSpec 属性), 1922
CacheFTPHandler (urllib.request 中的类), 1326
calcobjsize() (在 test.support 模块中), 1714
calcsize() (在 struct 模块中), 168
calcvobjsize() (在 test.support 模块中), 1714
calendar
    module, 230
calendar() (在 calendar 模块中), 235
calendar 命令行选项
    -c, 238

--css, 238
-e, 237
--encoding, 237
-f, 237
--first-weekday, 237
-h, 237
--help, 237
-L, 237
-l, 238
--lines, 238
--locale, 237
-m, 238
month, 237
--months, 238
-s, 238
--spacing, 238
-t, 237
--type, 237
-w, 238
--width, 238
year, 237
Calendar (calendar 中的类), 231
CALL (*monitoring event*), 1811
CALL (*opcode*), 2011
call() (在 operator 模块中), 405
call() (在 subprocess 模块中), 964
call() (在 unittest.mock 模块中), 1681
call_args_list (unittest.mock.Mock 属性), 1659
call_args (unittest.mock.Mock 属性), 1658
call_at() (asyncio.loop 方法), 1026
call_count (unittest.mock.Mock 属性), 1656
call_exception_handler() (asyncio.loop 方法), 1037
CALL_FUNCTION_EX (*opcode*), 2012
CALL_INTRINSIC_1 (*opcode*), 2014
CALL_INTRINSIC_2 (*opcode*), 2015
CALL_KW (*opcode*), 2012
call_later() (asyncio.loop 方法), 1026
call_list() (unittest.mock.call 方法), 1681
call_soon() (asyncio.loop 方法), 1025
call_soon_threadsafe() (asyncio.loop 方法), 1025
call_tracing() (在 sys 模块中), 1789
callable -- 可调用对象, **2065**
callable()
built-in function, 10
Callable() (在 typing 模块中), 1596
CallableProxyType()(在 weakref 模块中), 272
Callable (collections.abc 中的类), 257
callback -- 回调, **2065**
callback() (contextlib.ExitStack 方法), 1849
callback_args (optparse.Option 属性), 839
callback_kwargs (optparse.Option 属性), 839
callbacks() (在 gc 模块中), 1874
callback (optparse.Option 属性), 839
CalledProcessError, 953
called (unittest.mock.Mock 属性), 1656
Call (ast 中的类), 1950
CAN() (在 curses.ascii 模块中), 881
CAN_BCM() (在 socket 模块中), 1081
can_change_color() (在 curses 模块中), 855
can_fetch() (urllib.robotparser.RobotFileParser 方法), 1348
CAN_ISOTP() (在 socket 模块中), 1081
CAN_J1939() (在 socket 模块中), 1081
CAN_RAW_FD_FRAMES()(在 socket 模块中), 1081
CAN_RAW_JOIN_FILTERS() (在 socket 模块中), 1081
can_symlink() (在 test.support.os_helper 模块中), 1722
can_write_eof() (asyncio.StreamWriter 方法), 1006
can_write_eof() (asyncio.WriteTransport 方法), 1051
can_xattr() (在 test.support.os_helper 模块中), 1722
cancel() (asyncio.Future 方法), 1047
cancel() (asyncio.Handle 方法), 1040
cancel() (asyncio.Task 方法), 1000
cancel() (concurrent.futures.Future 方法), 948
cancel() (sched.scheduler 方法), 970
cancel() (threading.Timer 方法), 896
cancel() (tkinter.dnd.DndHandler 方法), 1520
CANCEL() (在 tkinter.messagebox 模块中), 1519
cancel_command() (tkinter.filedialog.FileDialog 方法), 1516
cancel_dump_traceback_later() (在 faulthandler 模块中), 1737
cancel_join_thread() (multiprocessing.Queue 方法), 909
cancelled() (asyncio.Future 方法), 1046
cancelled() (asyncio.Handle 方法), 1040
cancelled() (asyncio.Task 方法), 1001
cancelled() (concurrent.futures.Future 方法), 949
CancelledError, 950, 1022
cancelling() (asyncio.Task 方法), 1002
CannotSendHeader, 1353
CannotSendRequest, 1353
canonic() (bdb.Bdb 方法), 1732
canonical() (decimal.Context 方法), 338
canonical() (decimal.Decimal 方法), 330
canonicalize() (在 xml.etree.ElementTree 模块中), 1262
capa() (poplib.POP3 方法), 1366
capitalize() (bytearray 方法), 68
capitalize() (bytes 方法), 68
capitalize() (str 方法), 49
CapsuleType (types 中的类), 280

captured_stderr() (在 test.support 模块中), 1713
captured_stdin() (在 test.support 模块中), 1713
captured_stdout() (在 test.support 模块中), 1713
captureWarnings() (在 logging 模块中), 717
capwords() (在 string 模块中), 121
casefold() (str 方法), 49
cast() (memoryview 方法), 78
cast() (在 ctypes 模块中), 776
cast() (在 typing 模块中), 1584
--catch
    unittest 命令行选项, 1625
catch_threading_exception() (在 test.support.threading_helper 模块中), 1721
catch_unraisable_exception() (在 test.support 模块中), 1716
catch_warnings (warnings 中的类), 1831
category() (在 unicodedata 模块中), 158
cbreak() (在 curses 模块中), 855
cbrt() (在 math 模块中), 317
ccc() (ftplib.FTP_TLS 方法), 1364
cdf() (statistics.NormalDist 方法), 374
CDLL (ctypes 中的类), 771
ceil() (在 *math* 模块中), 37
ceil() (在 math 模块中), 314
CellType() (在 types 模块中), 278
center() (bytearray 方法), 66
center() (bytes 方法), 66
center() (str 方法), 49
CERT_NONE() (在 ssl 模块中), 1106
CERT_OPTIONAL() (在 ssl 模块中), 1107
CERT_REQUIRED() (在 ssl 模块中), 1107
cert_store_stats() (ssl.SSLContext 方法), 1118
cert_time_to_seconds()(在 ssl 模块中), 1105
CertificateError, 1105
certificates, 1125
cfmakecbreak() (在 tty 模块中), 2037
cfmakeraw() (在 tty 模块中), 2037
CFUNCTYPE() (在 ctypes 模块中), 774
cget() (tkinter.font.Font 方法), 1514
cgi
    module, 2056
--cgi
    http.server 命令行选项, 1397
cgi_directories(http.server.CGIHTTPRequestHandler 属性), 1397
CGIHandler (wsgiref.handlers 中的类), 1316
CGIHTTPRequestHandler (http.server 中的类), 1396
cgitb
    module, 2056
CGIXMLRPCRequestHandler (xmlrpc.server 中的类), 1417
chain() (在 itertools 模块中), 380
chaining
    异常, 99
    比较, 36
ChainMap (collections 中的类), 238
ChainMap (typing 中的类), 1593
change_cwd()(在 test.support.os_helper 模块中), 1722
CHANNEL_BINDING_TYPES() (在 ssl 模块中), 1111
CHAR_MAX() (在 locale 模块中), 1454
CharacterDataHandler() (xml.parsers.expat.xmlparser 方法), 1303
characters() (xml.sax.handler.ContentHandler 方法), 1293
characters_written (BlockingIOError 属性), 106
charset() (gettext.NullTranslations 方法), 1444
Charset (email.charset 中的类), 1206
chdir() (在 contextlib 模块中), 1847
chdir() (在 os 模块中), 637
check() (imaplib.IMAP4 方法), 1370
check() (在 tabnanny 模块中), 1989
check__all__() (在 test.support 模块中), 1717
check_call() (在 subprocess 模块中), 964
check_disallow_instantiation() (在 test.support 模块中), 1718
CHECK_EG_MATCH (*opcode*), 2005
CHECK_EXC_MATCH (*opcode*), 2005
check_free_after_iterating() (在 test.support 模块中), 1717
check_hostname (ssl.SSLContext 属性), 1122
check_impl_detail() (在 test.support 模块中), 1713
check_no_resource_warning() (在 test.support.warnings_helper 模块中), 1724
check_output() (doctest.OutputChecker 方法), 1619
check_output() (在 subprocess 模块中), 964
check_returncode() (subprocess.CompletedProcess 方法), 952
check_syntax_error()(在 test.support 模块中), 1716
check_syntax_warning() (在 test.support.warnings_helper 模块中), 1725
check_unused_args() (string.Formatter 方法), 113
check_warnings() (在 test.support.warnings_helper 模块中), 1725
checkcache() (在 linecache 模块中), 457
CHECKED_HASH(py_compile.PycInvalidationMode 属性), 1992

checkfunctionname() (在 bdb 模块中), 1735
checksizeof() (在 test.support 模块中), 1714
checksum
    循环冗余检测, 526
check (lzma.LZMADecompressor 属性), 539
chflags() (在 os 模块中), 637
chgat() (curses.window 方法), 862
childNodes (xml.dom.Node 属性), 1276
ChildProcessError, 106
children (pyclbr.Class 属性), 1990
children (pyclbr.Function 属性), 1990
children (tkinter.Tk 属性), 1503
chksum (tarfile.TarInfo 属性), 560
chmod() (pathlib.Path 方法), 432
chmod() (在 os 模块中), 638
--choice
    random 命令行选项, 362
choice() (在 random 模块中), 356
choice() (在 secrets 模块中), 612
choices() (在 random 模块中), 356
choices (optparse.Option 属性), 839
Chooser (tkinter.colorchooser 中的类), 1513
chown() (在 os 模块中), 639
chown() (在 shutil 模块中), 462
chr()
    built-in function, 10
chroot() (在 os 模块中), 639
CHRTYPE() (在 tarfile 模块中), 555
chunk
    module, 2056
cipher() (ssl.SSLSocket 方法), 1115
circle() (在 turtle 模块中), 1466
CIRCUMFLEX() (在 token 模块中), 1983
CIRCUMFLEXEQUAL() (在 token 模块中), 1983
Clamped (decimal 中的类), 342
class -- 类, **2065**
class variable -- 类变量, **2065**
ClassDef (ast 中的类), 1970
classmethod()
    built-in function, 10
ClassMethodDescriptorType() (在 types 模块中), 278
ClassVar() (在 typing 模块中), 1565
Class (pyclbr 中的类), 1990
Class (symtable 中的类), 1978
CLASS (symtable.SymbolTableType 属性), 1976
CLD_CONTINUED() (在 os 模块中), 672
CLD_DUMPED() (在 os 模块中), 672
CLD_EXITED() (在 os 模块中), 672
CLD_KILLED() (在 os 模块中), 672
CLD_STOPPED() (在 os 模块中), 672
CLD_TRAPPED() (在 os 模块中), 672
clean() (mailbox.Maildir 方法), 1225
cleandoc() (在 inspect 模块中), 1881
CleanImport (test.support.import_helper 中的类), 1724
cleanup() (tempfile.TemporaryDirectory 方法), 450
CLEANUP_THROW (*opcode*), 2004
clear (*pdb command*), 1742
clear() (序列方法), 45
clear() (array.array 方法), 268
clear() (asyncio.Event 方法), 1011
clear() (collections.deque 方法), 244
clear() (curses.window 方法), 862
clear() (dbm.gnu.gdbm 方法), 492
clear() (dbm.ndbm.ndbm 方法), 493
clear() (dict 方法), 85
clear() (email.message.EmailMessage 方法), 1166
clear() (frozenset 方法), 83
clear() (http.cookiejar.CookieJar 方法), 1404
clear() (mailbox.Mailbox 方法), 1223
clear() (threading.Event 方法), 895
clear() (xml.etree.ElementTree.Element 方法), 1267
clear() (在 turtle 模块中), 1474
clear_all_breaks() (bdb.Bdb 方法), 1734
clear_all_file_breaks() (bdb.Bdb 方法), 1734
clear_bpbynumber() (bdb.Bdb 方法), 1734
clear_break() (bdb.Bdb 方法), 1734
clear_cache() (在 filecmp 模块中), 447
clear_cache() (zoneinfo.ZoneInfo 类方法), 228
clear_content() (email.message.EmailMessage 方法), 1167
clear_flags() (decimal.Context 方法), 337
clear_frames() (在 traceback 模块中), 1864
clear_history() (在 readline 模块中), 162
clear_overloads() (在 typing 模块中), 1587
clear_session_cookies() (http.cookiejar.CookieJar 方法), 1404
clear_traces() (在 tracemalloc 模块中), 1765
clear_traps() (decimal.Context 方法), 337
clearcache() (在 linecache 模块中), 457
clearok() (curses.window 方法), 862
clearscreen() (在 turtle 模块中), 1480
clearstamp() (在 turtle 模块中), 1467
clearstamps() (在 turtle 模块中), 1468
Client() (在 multiprocessing.connection 模块中), 927
client_address (http.server.BaseHTTPRequestHandler 属性), 1393
client_address (socketserver.BaseRequestHandler 属性), 1388
CLOCK_BOOTTIME() (在 time 模块中), 699
clock_getres() (在 time 模块中), 691
clock_gettime() (在 time 模块中), 691
clock_gettime_ns() (在 time 模块中), 691
CLOCK_HIGHRES() (在 time 模块中), 699
CLOCK_MONOTONIC() (在 time 模块中), 699
CLOCK_MONOTONIC_RAW() (在 time 模块中), 699

CLOCK_MONOTONIC_RAW_APPROX() (在 time 模块中), 700
CLOCK_PROCESS_CPUTIME_ID() (在 time 模块中), 700
CLOCK_PROF() (在 time 模块中), 700
CLOCK_REALTIME() (在 time 模块中), 700
clock_seq_hi_variant (uuid.UUID 属性), 1381
clock_seq_low (uuid.UUID 属性), 1381
clock_seq (uuid.UUID 属性), 1381
clock_settime() (在 time 模块中), 691
clock_settime_ns() (在 time 模块中), 692
CLOCK_TAI() (在 time 模块中), 700
CLOCK_THREAD_CPUTIME_ID() (在 time 模块中), 700
CLOCK_UPTIME() (在 time 模块中), 700
CLOCK_UPTIME_RAW() (在 time 模块中), 700
CLOCK_UPTIME_RAW_APPROX() (在 time 模块中), 700
clone() (email.generator.BytesGenerator 方法), 1171
clone() (email.generator.Generator 方法), 1172
clone() (email.policy.Policy 方法), 1175
clone() (在 turtle 模块中), 1478
CLONE_FILES() (在 os 模块中), 624
CLONE_FS() (在 os 模块中), 624
CLONE_NEWCGROUP() (在 os 模块中), 624
CLONE_NEWIPC() (在 os 模块中), 624
CLONE_NEWNET() (在 os 模块中), 624
CLONE_NEWNS() (在 os 模块中), 624
CLONE_NEWPID() (在 os 模块中), 624
CLONE_NEWTIME() (在 os 模块中), 624
CLONE_NEWUSER() (在 os 模块中), 624
CLONE_NEWUTS() (在 os 模块中), 624
CLONE_SIGHAND() (在 os 模块中), 624
CLONE_SYSVSEM() (在 os 模块中), 624
CLONE_THREAD() (在 os 模块中), 624
CLONE_VM() (在 os 模块中), 624
cloneNode() (xml.dom.Node 方法), 1277
close() (asyncio.AbstractChildWatcher 方法), 1063
close() (asyncio.BaseTransport 方法), 1049
close() (asyncio.loop 方法), 1024
close() (asyncio.Runner 方法), 983
close() (asyncio.Server 方法), 1041
close() (asyncio.StreamWriter 方法), 1006
close() (asyncio.SubprocessTransport 方法), 1052
close() (contextlib.ExitStack 方法), 1850
close() (dbm.dumb.dumbdbm 方法), 494
close() (dbm.gnu.gdbm 方法), 492
close() (dbm.ndbm.ndbm 方法), 493
close() (email.parser.BytesFeedParser 方法), 1168
close() (ftplib.FTP 方法), 1363
close() (html.parser.HTMLParser 方法), 1250
close() (http.client.HTTPConnection 方法), 1356
close() (imaplib.IMAP4 方法), 1370
close() (io.IOBase 方法), 681
close() (logging.FileHandler 方法), 730
close() (logging.Handler 方法), 708
close() (logging.handlers.MemoryHandler 方法), 738
close() (logging.handlers.NTEventLogHandler 方法), 737
close() (logging.handlers.SocketHandler 方法), 734
close() (logging.handlers.SysLogHandler 方法), 736
close() (mailbox.Mailbox 方法), 1224
close() (mailbox.Maildir 方法), 1226
close() (mailbox.MH 方法), 1228
close() (mmap.mmap 方法), 1154
close() (multiprocessing.connection.Connection 方法), 912
close() (multiprocessing.connection.Listener 方法), 927
close() (multiprocessing.pool.Pool 方法), 926
close() (multiprocessing.Process 方法), 906
close() (multiprocessing.Queue 方法), 909
close() (multiprocessing.shared_memory.SharedMemory 方法), 940
close() (multiprocessing.SimpleQueue 方法), 909
close() (os.scandir 方法), 645
close() (select.devpoll 方法), 1136
close() (select.epoll 方法), 1137
close() (select.kqueue 方法), 1138
close() (selectors.BaseSelector 方法), 1142
close() (shelve.Shelf 方法), 485
close() (socket.socket 方法), 1091
close() (sqlite3.Blob 方法), 513
close() (sqlite3.Connection 方法), 501
close() (sqlite3.Cursor 方法), 511
close() (tarfile.TarFile 方法), 559
close() (urllib.request.BaseHandler 方法), 1328
close() (wave.Wave_read 方法), 1438
close() (wave.Wave_write 方法), 1439
Close() (winreg.PyHKEY 方法), 2029
close() (xml.etree.ElementTree.TreeBuilder 方法), 1270
close() (xml.etree.ElementTree.XMLParser 方法), 1271
close() (xml.etree.ElementTree.XMLPullParser 方法), 1272
close() (xml.sax.xmlreader.IncrementalParser 方法), 1298
close() (zipfile.ZipFile 方法), 544
close() (在 fileinput 模块中), 853

close() (在 os 模块中), 624
close() (在 socket 模块中), 1086
close_clients() (asyncio.Server 方法), 1041
close_connection(http.server.BaseHTTPRequestHandler 属性), 1393
CloseBoundaryNotFoundDefect, 1180
closed (http.client.HTTPResponse 属性), 1357
closed (io.IOBase 属性), 681
closed (mmap.mmap 属性), 1154
closed (select.devpoll 属性), 1136
closed (select.epoll 属性), 1137
closed (select.kqueue 属性), 1138
CloseKey() (在 winreg 模块中), 2022
closelog() (在 syslog 模块中), 2046
closerange() (在 os 模块中), 624
closing() (在 contextlib 模块中), 1844
closure variable -- 闭包变量, **2065**
clrtobot() (curses.window 方法), 862
clrtoeol() (curses.window 方法), 862
cmath
    module, 321
cmd
    module, 1491, 1738
cmdloop() (cmd.Cmd 方法), 1491
cmdqueue (cmd.Cmd 属性), 1493
Cmd (cmd 中的类), 1491
cmd (subprocess.CalledProcessError 属性), 953
cmd (subprocess.TimeoutExpired 属性), 953
cmp() (在 filecmp 模块中), 446
cmp_op() (在 dis 模块中), 2016
cmp_to_key() (在 functools 模块中), 394
cmpfiles() (在 filecmp 模块中), 446
CMSG_LEN() (在 socket 模块中), 1089
CMSG_SPACE() (在 socket 模块中), 1089
CO_ASYNC_GENERATOR() (在 inspect 模块中), 1892
CO_COROUTINE() (在 inspect 模块中), 1892
CO_GENERATOR() (在 inspect 模块中), 1892
CO_ITERABLE_COROUTINE() (在 inspect 模块中), 1892
CO_NESTED() (在 inspect 模块中), 1892
CO_NEWLOCALS() (在 inspect 模块中), 1891
CO_OPTIMIZED() (在 inspect 模块中), 1891
CO_VARARGS() (在 inspect 模块中), 1892
CO_VARKEYWORDS() (在 inspect 模块中), 1892
code
    module, 1897
code_context (inspect.FrameInfo 属性), 1888
code_context (inspect.Traceback 属性), 1888
code_info() (在 dis 模块中), 1998
CodecInfo (codecs 中的类), 174
codecs
    module, 173
Codec (codecs 中的类), 178
coded_value (http.cookies.Morsel 属性), 1400
codeop
    module, 1899
codepoint2name() (在 html.entities 模块中), 1254
codes() (在 xml.parsers.expat.errors 模块中), 1306
CODESET() (在 locale 模块中), 1449
CodeType (types 中的类), 278
code (SystemExit 属性), 104
code (urllib.error.HTTPError 属性), 1347
code (urllib.response.addinfourl 属性), 1338
code (xml.etree.ElementTree.ParseError 属性), 1273
code (xml.parsers.expat.ExpatError 属性), 1304
col_offset (ast.AST 属性), 1944
collapse_addresses() (在 ipaddress 模块中), 1434
collapse_rfc2231_value() (在 email.utils 模块中), 1210
collect() (在 gc 模块中), 1872
collectedDurations (unittest.TestResult 属性), 1644
collections
    module, 238
collections.abc
    module, 254
Collection (collections.abc 中的类), 257
Collection (typing 中的类), 1594
colno (json.JSONDecodeError 属性), 1218
colno (re.PatternError 属性), 134
colno (traceback.FrameSummary 属性), 1867
COLON() (在 token 模块中), 1981
COLONEQUAL() (在 token 模块中), 1984
colon (mailbox.Maildir 属性), 1224
color() (在 turtle 模块中), 1473
COLOR_BLACK() (在 curses 模块中), 878
COLOR_BLUE() (在 curses 模块中), 878
color_content() (在 curses 模块中), 855
COLOR_CYAN() (在 curses 模块中), 878
COLOR_GREEN() (在 curses 模块中), 878
COLOR_MAGENTA() (在 curses 模块中), 878
color_pair() (在 curses 模块中), 855
COLOR_PAIRS() (在 curses 模块中), 867
COLOR_RED() (在 curses 模块中), 878
COLOR_WHITE() (在 curses 模块中), 878
COLOR_YELLOW() (在 curses 模块中), 878
colormode() (在 turtle 模块中), 1484
COLORS() (在 curses 模块中), 867
colorsys
    module, 1439
COLS() (在 curses 模块中), 867
column() (tkinter.ttk.Treeview 方法), 1530
columnize() (cmd.Cmd 方法), 1492
COLUMNS, 860

columns (os.terminal_size 属性), 635
comb() (在 math 模块中), 313
combinations() (在 itertools 模块中), 380
combinations_with_replacement() (在 itertools 模块中), 381
combine() (datetime.datetime 类方法), 203
combining() (在 unicodedata 模块中), 158
Combobox (tkinter.ttk 中的类), 1524
COMMA() (在 token 模块中), 1981
--command
    pdb 命令行选项, 1739
CommandCompiler (codeop 中的类), 1899
commands (*pdb command*), 1742
command(http.server.BaseHTTPRequestHandler 属性), 1393
comment() (xml.etree.ElementTree.TreeBuilder 方法), 1270
comment() (xml.sax.handler.LexicalHandler 方法), 1294
COMMENT() (在 token 模块中), 1980
Comment()(在 xml.etree.ElementTree 模块中), 1263
comment_url (http.cookiejar.Cookie 属性), 1408
commenters (shlex.shlex 属性), 1498
CommentHandler() (xml.parsers.expat.xmlparser 方法), 1304
comment (http.cookiejar.Cookie 属性), 1408
comment (http.cookies.Morsel 属性), 1399
comment (zipfile.ZipFile 属性), 547
comment (zipfile.ZipInfo 属性), 550
commit() (sqlite3.Connection 方法), 501
Common Vulnerabilities and Exposures
    CVE 2020-10735, 96
    CVE 2023-52425, 1255
Common Weakness Enumeration
    CWE 257, 613
common_dirs (filecmp.dircmp 属性), 447
common_files (filecmp.dircmp 属性), 448
common_funny (filecmp.dircmp 属性), 448
common_types() (在 mimetypes 模块中), 1240
commonpath() (在 os.path 模块中), 435
commonprefix() (在 os.path 模块中), 435
common (filecmp.dircmp 属性), 447
communicate() (asyncio.subprocess.Process 方法), 1017
communicate() (subprocess.Popen 方法), 960
--compact
    json.tool 命令行选项, 1221
compare() (decimal.Context 方法), 338
compare() (decimal.Decimal 方法), 330
compare() (difflib.Differ 方法), 150
compare_digest() (在 hmac 模块中), 611
compare_digest() (在 secrets 模块中), 613
compare_networks() (ipaddress.IPv4Network 方法), 1430
compare_networks() (ipaddress.IPv6Network 方法), 1432
COMPARE_OP (*opcode*), 2009
compare_signal() (decimal.Context 方法), 338
compare_signal() (decimal.Decimal 方法), 330
compare_to() (tracemalloc.Snapshot 方法), 1767
compare_total() (decimal.Context 方法), 338
compare_total() (decimal.Decimal 方法), 330
compare_total_mag() (decimal.Context 方法), 338
compare_total_mag() (decimal.Decimal 方法), 331
Compare (ast 中的类), 1950
COMPARISON_FLAGS()(在 doctest 模块中), 1609
compat32() (在 email.policy 模块中), 1179
Compat32 (email.policy 中的类), 1178
compile()
    built-in function, 11
compile() (在 py_compile 模块中), 1991
compile() (在 re 模块中), 130
compile_command() (在 code 模块中), 1897
compile_command() (在 codeop 模块中), 1899
compile_dir() (在 compileall 模块中), 1994
compile_file() (在 compileall 模块中), 1995
compile_path() (在 compileall 模块中), 1995
compileall
    module, 1992
compileall 命令行选项
    -b, 1993
    -d, 1993
    directory, 1993
    -e, 1993
    -f, 1993
    file, 1993
    --hardlink-dupes, 1993
    -i, 1993
    --invalidation-mode, 1993
    -j, 1993
    -l, 1993
    -o, 1993
    -p, 1993
    -q, 1993
    -r, 1993
    -s, 1993
    -x, 1993
compiler_flag (__future__._Feature 属性), 1872
Compile (codeop 中的类), 1899
complete() (rlcompleter.Completer 方法), 165
complete_statement() (在 sqlite3 模块中), 497
completedefault() (cmd.Cmd 方法), 1492
CompletedProcess (subprocess 中的类), 952
Completer (rlcompleter 中的类), 165

complex number -- 复数
    object -- 对象, 36
    字面值, 36
complex number -- 复数, **2065**
Complex (numbers 中的类), 309
complex (内置类), 11
comprehension (ast 中的类), 1953
--compress
    zipapp 命令行选项, 1783
compress() (bz2.BZ2Compressor 方法), 534
compress() (lzma.LZMACompressor 方法), 538
compress() (zlib.Compress 方法), 527
compress() (在 bz2 模块中), 535
compress() (在 gzip 模块中), 530
compress() (在 itertools 模块中), 381
compress() (在 lzma 模块中), 539
compress() (在 zlib 模块中), 525
compress_size (zipfile.ZipInfo 属性), 550
compress_type (zipfile.ZipInfo 属性), 550
compressed (ipaddress.IPv4Address 属性), 1423
compressed (ipaddress.IPv4Network 属性), 1429
compressed (ipaddress.IPv6Address 属性), 1425
compressed (ipaddress.IPv6Network 属性), 1431
compression() (ssl.SSLSocket 方法), 1115
CompressionError, 554
compressobj() (在 zlib 模块中), 526
COMSPEC, 669, 956
concat() (在 operator 模块中), 404
Concatenate() (在 typing 模块中), 1564
concurrent.futures
    module, 944
condition (*pdb command*), 1742
Condition() (multiprocessing.managers.SyncManager 方法), 919
Condition (asyncio 中的类), 1011
Condition (multiprocessing 中的类), 914
Condition (threading 中的类), 893
cond (bdb.Breakpoint 属性), 1732
config() (tkinter.font.Font 方法), 1514
configparser
    module, 576
ConfigParser (configparser 中的类), 587
configure() (tkinter.ttk.Style 方法), 1533
configure_mock() (unittest.mock.Mock 方法), 1656
CONFORM (enum.FlagBoundary 属性), 302
confstr() (在 os 模块中), 674
confstr_names() (在 os 模块中), 675
conjugate() (复数方法), 37
conjugate() (decimal.Decimal 方法), 331
conjugate() (numbers.Complex 方法), 309
connect() (ftplib.FTP 方法), 1360
connect() (http.client.HTTPConnection 方法), 1355
connect() (multiprocessing.managers.BaseManager 方法), 919
connect() (smtplib.SMTP 方法), 1376
connect() (socket.socket 方法), 1091
connect() (在 sqlite3 模块中), 496
connect_accepted_socket() (asyncio.loop 方法), 1031
connect_ex() (socket.socket 方法), 1092
connect_read_pipe() (asyncio.loop 方法), 1035
connect_write_pipe() (asyncio.loop 方法), 1035
connection_lost() (asyncio.BaseProtocol 方法), 1053
connection_made() (asyncio.BaseProtocol 方法), 1053
ConnectionAbortedError, 106
ConnectionError, 106
ConnectionRefusedError, 106
ConnectionResetError, 106
Connection (multiprocessing.connection 中的类), 912
Connection (sqlite3 中的类), 500
connection (sqlite3.Cursor 属性), 512
ConnectRegistry() (在 winreg 模块中), 2022
Constant (ast 中的类), 1946
constructor() (在 copyreg 模块中), 484
const (optparse.Option 属性), 839
consumed (asyncio.LimitOverrunError 属性), 1022
Container (collections.abc 中的类), 257
Container (typing 中的类), 1594
contains() (在 operator 模块中), 404
CONTAINS_OP (*opcode*), 2009
content type
    MIME, 1239
content_disposition (email.headerregistry.ContentDispositionHeader 属性), 1183
content_manager (email.policy.EmailPolicy 属性), 1177
content_type(email.headerregistry.ContentTypeHeader 属性), 1183
ContentDispositionHeader (email.headerregistry 中的类), 1183
ContentHandler (xml.sax.handler 中的类), 1290
ContentManager (email.contentmanager 中的类), 1185
contents() (importlib.abc.ResourceReader 方法), 1917
contents() (importlib.resources.abc.ResourceReader 方法), 1931
contents() (在 importlib.resources 模块中), 1930
contents (ctypes._Pointer 属性), 784
ContentTooShortError, 1347

ContentTransferEncoding (email.headerregistry 中的类), 1183
ContentTypeHeader (email.headerregistry 中的类), 1183
content (urllib.error.ContentTooShortError 属性), 1347
context -- 上下文, **2066**
context manager -- 上下文管理器, 88
context manager -- 上下文管理器, **2066**
context variable -- 上下文变量, **2066**
context_diff() (在 difflib 模块中), 144
ContextDecorator (contextlib 中的类), 1847
contextlib
    module, 1842
contextmanager() (在 contextlib 模块中), 1842
ContextManager (typing 中的类), 1596
contextvars
    module, 974
ContextVar (contextvars 中的类), 974
Context (contextvars 中的类), 976
Context (decimal 中的类), 336
context (ssl.SSLSocket 属性), 1116
CONTIG_RO (inspect.BufferFlags 属性), 1893
contiguous -- 连续, **2066**
contiguous (memoryview 属性), 81
CONTIG (inspect.BufferFlags 属性), 1892
continue (*pdb command*), 1743
Continue (ast 中的类), 1958
CONTINUOUS (enum.EnumCheck 属性), 301
control() (select.kqueue 方法), 1138
controlnames() (在 curses.ascii 模块中), 882
CONTTYPE() (在 tarfile 模块中), 555
convert_arg_line_to_args() (argparse.ArgumentParser 方法), 811
convert_field() (string.Formatter 方法), 113
CONVERT_VALUE (*opcode*), 2013
CookieError, 1398
CookieJar (http.cookiejar 中的类), 1402
cookiejar (urllib.request.HTTPCookieProcessor 属性), 1330
CookiePolicy (http.cookiejar 中的类), 1402
Cookie (http.cookiejar 中的类), 1402
Coordinated Universal Time, 690
copy
    module, 282, 484
    协议, 476
COPY (*opcode*), 2001
copy() (序列方法), 45
copy() (collections.deque 方法), 244
copy() (contextvars.Context 方法), 977
copy() (decimal.Context 方法), 337
copy() (dict 方法), 85
copy() (frozenset 方法), 82
copy() (hashlib.hash 方法), 601
copy() (hmac.HMAC 方法), 611
copy() (http.cookies.Morsel 方法), 1400
copy() (imaplib.IMAP4 方法), 1370
copy() (tkinter.font.Font 方法), 1514
copy() (types.MappingProxyType 方法), 280
copy() (zlib.Compress 方法), 527
copy() (zlib.Decompress 方法), 528
copy() (在 copy 模块中), 282
copy() (在 multiprocessing.sharedctypes 模块中), 917
copy() (在 shutil 模块中), 459
copy2() (在 shutil 模块中), 459
copy_abs() (decimal.Context 方法), 338
copy_abs() (decimal.Decimal 方法), 331
copy_context() (在 contextvars 模块中), 975
copy_decimal() (decimal.Context 方法), 337
copy_file_range() (在 os 模块中), 625
COPY_FREE_VARS (*opcode*), 2011
copy_location() (在 ast 模块中), 1973
copy_negate() (decimal.Context 方法), 338
copy_negate() (decimal.Decimal 方法), 331
copy_sign() (decimal.Context 方法), 338
copy_sign() (decimal.Decimal 方法), 331
copyfile() (在 shutil 模块中), 458
copyfileobj() (在 shutil 模块中), 458
copymode() (在 shutil 模块中), 458
copyreg
    module, 484
copyright() (在 sys 模块中), 1789
copyright (内置变量), 34
copysign() (在 math 模块中), 315
copystat() (在 shutil 模块中), 459
copytree() (在 shutil 模块中), 460
coroutine -- 协程, **2066**
coroutine function -- 协程函数, **2066**
coroutine() (在 types 模块中), 281
CoroutineType() (在 types 模块中), 278
Coroutine (collections.abc 中的类), 258
Coroutine (typing 中的类), 1595
correlation() (在 statistics 模块中), 371
cos() (在 cmath 模块中), 323
cos() (在 math 模块中), 319
cosh() (在 cmath 模块中), 323
cosh() (在 math 模块中), 319
--count
    trace 命令行选项, 1759
count() (序列方法), 43
count() (array.array 方法), 267
count() (bytearray 方法), 62
count() (bytes 方法), 62
count() (collections.deque 方法), 244
count() (multiprocessing.shared_memory.ShareableList 方法), 943
count() (str 方法), 49
count() (在 itertools 模块中), 381
count_diff (tracemalloc.StatisticDiff 属性), 1769
Counter (collections 中的类), 241

Counter (typing 中的类), 1593
countOf() (在 operator 模块中), 405
countTestCases() (unittest.TestCase 方法), 1638
countTestCases() (unittest.TestSuite 方法), 1641
count (tracemalloc.Statistic 属性), 1768
count (tracemalloc.StatisticDiff 属性), 1768
covariance() (在 statistics 模块中), 371
CoverageResults (trace 中的类), 1760
--coverdir
    trace 命令行选项, 1759
cProfile
    module, 1749
cProfile 命令行选项
    -m, 1747
    -o, 1747
    -s, 1747
CPU time, 693, 698
cpu_count() (在 multiprocessing 模块中), 910
cpu_count() (在 os 模块中), 675
CPython, **2066**
cpython_only() (在 test.support 模块中), 1715
CR() (在 curses.ascii 模块中), 880
crawl_delay() (urllib.robotparser.RobotFileParser 方法), 1348
crc32() (在 binascii 模块中), 1245
crc32() (在 zlib 模块中), 526
crc_hqx() (在 binascii 模块中), 1245
CRC (zipfile.ZipInfo 属性), 550
--create
    tarfile 命令行选项, 565
    zipfile 命令行选项, 551
create() (imaplib.IMAP4 方法), 1370
create() (venv.EnvBuilder 方法), 1776
create() (在 venv 模块中), 1778
create_aggregate() (sqlite3.Connection 方法), 502
create_archive() (在 zipapp 模块中), 1783
create_autospec() (在 unittest.mock 模块中), 1682
CREATE_BREAKAWAY_FROM_JOB() (在 subprocess 模块中), 963
create_collation() (sqlite3.Connection 方法), 504
create_configuration() (venv.EnvBuilder 方法), 1777
create_connection() (asyncio.loop 方法), 1027
create_connection() (在 socket 模块中), 1085
create_datagram_endpoint() (asyncio.loop 方法), 1029
create_decimal() (decimal.Context 方法), 337
create_decimal_from_float() (decimal.Context 方法), 337
create_default_context() (在 ssl 模块中), 1103
CREATE_DEFAULT_ERROR_MODE() (在 subprocess 模块中), 963
create_eager_task_factory() (在 asyncio 模块中), 992
create_empty_file() (在 test.support.os_helper 模块中), 1722
create_function() (sqlite3.Connection 方法), 501
create_future() (asyncio.loop 方法), 1027
create_git_ignore_file() (venv.EnvBuilder 方法), 1778
create_module() (importlib.abc.Loader 方法), 1913
create_module() (importlib.machinery.ExtensionFileLoade 方法), 1921
create_module() (zipimport.zipimporter 方法), 1902
CREATE_NEW_CONSOLE() (在 subprocess 模块中), 963
CREATE_NEW_PROCESS_GROUP() (在 subprocess 模块中), 963
CREATE_NO_WINDOW() (在 subprocess 模块中), 963
create_server() (asyncio.loop 方法), 1030
create_server() (在 socket 模块中), 1085
create_stats() (profile.Profile 方法), 1749
create_string_buffer() (在 ctypes 模块中), 776
create_subprocess_exec() (在 asyncio 模块中), 1015
create_subprocess_shell() (在 asyncio 模块中), 1016
create_system (zipfile.ZipInfo 属性), 550
create_task() (asyncio.loop 方法), 1027
create_task() (asyncio.TaskGroup 方法), 989
create_task() (在 asyncio 模块中), 988
create_unicode_buffer() (在 ctypes 模块中), 777
create_unix_connection() (asyncio.loop 方法), 1029
create_unix_server() (asyncio.loop 方法), 1031
create_version (zipfile.ZipInfo 属性), 550
create_window_function() (sqlite3.Connection 方法), 503
createAttribute() (xml.dom.Document 方法), 1278
createAttributeNS() (xml.dom.Document 方法), 1278
createComment() (xml.dom.Document 方法), 1278
createDocument() (xml.dom.DOMImplementation 方法), 1275

createDocumentType() (xml.dom.DOMImplementation 方法), 1275
createElement() (xml.dom.Document 方法), 1278
createElementNS() (xml.dom.Document 方法), 1278
createfilehandler() (_tkinter.Widget.tk 方法), 1512
CreateKey() (在 winreg 模块中), 2022
CreateKeyEx() (在 winreg 模块中), 2022
createLock() (logging.Handler 方法), 707
createLock() (logging.NullHandler 方法), 730
createProcessingInstruction() (xml.dom.Document 方法), 1278
createSocket() (logging.handlers.SocketHandler 方法), 734
createSocket() (logging.handlers.SysLogHandler 方法), 736
createTextNode() (xml.dom.Document 方法), 1278
credits (内置变量), 34
critical() (logging.Logger 方法), 706
CRITICAL() (在 logging 模块中), 707
critical() (在 logging 模块中), 714
CRNCYSTR() (在 locale 模块中), 1451
CRT_ASSEMBLY_VERSION() (在 msvcrt 模块中), 2021
CRT_ASSERT() (在 msvcrt 模块中), 2021
CRT_ERROR() (在 msvcrt 模块中), 2021
CRT_WARN() (在 msvcrt 模块中), 2021
CRTDBG_MODE_DEBUG() (在 msvcrt 模块中), 2021
CRTDBG_MODE_FILE() (在 msvcrt 模块中), 2021
CRTDBG_MODE_WNDW() (在 msvcrt 模块中), 2021
CRTDBG_REPORT_MODE() (在 msvcrt 模块中), 2021
CrtSetReportFile() (在 msvcrt 模块中), 2021
CrtSetReportMode() (在 msvcrt 模块中), 2021
crypt
    module, 2056
--css
    calendar 命令行选项, 238
cssclass_month_head(calendar.HTMLCalendar 属性), 233
cssclass_month (calendar.HTMLCalendar 属性), 233
cssclass_noday (calendar.HTMLCalendar 属性), 233
cssclass_year_head (calendar.HTMLCalendar 属性), 233
cssclass_year (calendar.HTMLCalendar 属性), 233
cssclasses_weekday_head (calendar.HTMLCalendar 属性), 233
cssclasses (calendar.HTMLCalendar 属性), 233
csv, 569
    module, 569
cte_type (email.policy.Policy 属性), 1174
ctermid() (在 os 模块中), 617
cte(email.headerregistry.ContentTransferEncoding 属性), 1183
ctime() (datetime.date 方法), 199
ctime() (datetime.datetime 方法), 209
ctime() (在 time 模块中), 692
ctrl() (在 curses.ascii 模块中), 882
CTRL_BREAK_EVENT() (在 signal 模块中), 1146
CTRL_C_EVENT() (在 signal 模块中), 1146
ctypes
    module, 754
curdir() (在 os 模块中), 675
currency() (在 locale 模块中), 1453
current context -- 当前上下文, 2066
current() (tkinter.ttk.Combobox 方法), 1524
current_process() (在 multiprocessing 模块中), 910
current_task() (在 asyncio 模块中), 998
current_thread() (在 threading 模块中), 886
CurrentByteIndex(xml.parsers.expat.xmlparser 属性), 1302
CurrentColumnNumber (xml.parsers.expat.xmlparser 属性), 1302
currentframe() (在 inspect 模块中), 1889
CurrentLineNumber (xml.parsers.expat.xmlparser 属性), 1302
curs_set() (在 curses 模块中), 855
curses
    module, 854
curses.ascii
    module, 879
curses.panel
    module, 883
curses.textpad
    module, 878
cursor() (sqlite3.Connection 方法), 500
Cursor (sqlite3 中的类), 510
cursyncup() (curses.window 方法), 862
cwd() (ftplib.FTP 方法), 1363
cwd() (pathlib.Path 类方法), 423
cycle() (在 itertools 模块中), 382
CycleError, 308

D

-d
    compileall 命令行选项, 1993
    gzip 命令行选项, 531
    http.server 命令行选项, 1397
    idle 命令行选项, 1544
D_FMT() (在 locale 模块中), 1449
D_T_FMT() (在 locale 模块中), 1449

daemon_threads(socketserver.ThreadingMixIn 属性), 1385
daemon (multiprocessing.Process 属性), 905
daemon (threading.Thread 属性), 890
data() (xml.etree.ElementTree.TreeBuilder 方法), 1270
data_filter() (在 tarfile 模块中), 562
data_open() (urllib.request.DataHandler 方法), 1332
data_received() (asyncio.Protocol 方法), 1053
DatabaseError, 514
dataclass() (在 dataclasses 模块中), 1832
dataclass_transform() (在 typing 模块中), 1585
dataclasses
    module, 1832
DataError, 514
datagram_received() (asyncio.DatagramProtocol 方法), 1054
DatagramHandler (logging.handlers 中的类), 735
DatagramProtocol (asyncio 中的类), 1053
DatagramRequestHandler (socketserver 中的类), 1388
DatagramTransport (asyncio 中的类), 1049
DataHandler (urllib.request 中的类), 1326
data (collections.UserDict 属性), 253
data (collections.UserList 属性), 254
data (collections.UserString 属性), 254
data (plistlib.UID 属性), 596
data (select.kevent 属性), 1140
data (selectors.SelectorKey 属性), 1141
data (urllib.request.Request 属性), 1326
data (xml.dom.Comment 属性), 1280
data (xml.dom.ProcessingInstruction 属性), 1280
data (xml.dom.Text 属性), 1280
data (xmlrpc.client.Binary 属性), 1413
date() (datetime.datetime 方法), 206
date_time_string() (http.server.BaseHTTPRequestHandler 方法), 1395
date_time (zipfile.ZipInfo 属性), 550
DateHeader (email.headerregistry 中的类), 1181
datetime
    module, 191
datetime (datetime 中的类), 201
datetime (email.headerregistry.DateHeader 属性), 1182
DateTime (xmlrpc.client 中的类), 1412
date (datetime 中的类), 196
DAY_1() (在 locale 模块中), 1450
DAY_2() (在 locale 模块中), 1450
DAY_3() (在 locale 模块中), 1450
DAY_4() (在 locale 模块中), 1450
DAY_5() (在 locale 模块中), 1450
DAY_6() (在 locale 模块中), 1450
DAY_7() (在 locale 模块中), 1450
day_abbr() (在 calendar 模块中), 235
day_name() (在 calendar 模块中), 235
Daylight Saving Time, 690
daylight() (在 time 模块中), 701
days (datetime.timedelta 属性), 194
Day (calendar 中的类), 235
day (datetime.date 属性), 198
day (datetime.datetime 属性), 204
DbfilenameShelf (shelve 中的类), 486
dbm
    module, 488
dbm.dumb
    module, 493
dbm.gnu
    module, 485, 490
dbm.ndbm
    module, 485, 492
dbm.sqlite3
    module, 490
DC1() (在 curses.ascii 模块中), 880
DC2() (在 curses.ascii 模块中), 881
DC3() (在 curses.ascii 模块中), 881
DC4() (在 curses.ascii 模块中), 881
dcgettext() (在 locale 模块中), 1454
deactivate_stack_trampoline() (在 sys 模块中), 1807
debug (*pdb command*), 1745
debug() (logging.Logger 方法), 704
debug() (unittest.TestCase 方法), 1632
debug() (unittest.TestSuite 方法), 1641
debug() (在 doctest 模块中), 1620
DEBUG() (在 logging 模块中), 707
debug() (在 logging 模块中), 713
DEBUG() (在 re 模块中), 129
DEBUG_BYTECODE_SUFFIXES() (在 importlib.machinery 模块中), 1918
DEBUG_COLLECTABLE() (在 gc 模块中), 1875
DEBUG_LEAK() (在 gc 模块中), 1875
DEBUG_SAVEALL() (在 gc 模块中), 1875
debug_src() (在 doctest 模块中), 1621
DEBUG_STATS() (在 gc 模块中), 1875
DEBUG_UNCOLLECTABLE() (在 gc 模块中), 1875
debuglevel (http.client.HTTPResponse 属性), 1357
DebugRunner (doctest 中的类), 1621
debug (imaplib.IMAP4 属性), 1373
debug (shlex.shlex 属性), 1499
debug (sys.flags 属性), 1793
debug (zipfile.ZipFile 属性), 547
DECEMBER() (在 calendar 模块中), 236
decimal
    module, 325
decimal() (在 unicodedata 模块中), 158
DecimalException (decimal 中的类), 342
Decimal (decimal 中的类), 329

decode
    编解码器, 173
decode() (bytearray 方法), 63
decode() (bytes 方法), 63
decode() (codecs.Codec 方法), 178
decode() (codecs.IncrementalDecoder 方法), 180
decode() (json.JSONDecoder 方法), 1217
decode() (xmlrpc.client.Binary 方法), 1413
decode() (xmlrpc.client.DateTime 方法), 1412
decode() (在 base64 模块中), 1244
decode() (在 codecs 模块中), 174
decode() (在 quopri 模块中), 1247
decode_header() (在 email.header 模块中), 1205
decode_params() (在 email.utils 模块中), 1211
decode_rfc2231() (在 email.utils 模块中), 1210
decode_source() (在 importlib.util 模块中), 1923
decodebytes() (在 base64 模块中), 1244
DecodedGenerator (email.generator 中的类), 1172
decodestring() (在 quopri 模块中), 1247
decode (codecs.CodecInfo 属性), 174
decomposition()(在 unicodedata 模块中), 158
--decompress
    gzip 命令行选项, 531
decompress() (bz2.BZ2Decompressor 方法), 534
decompress() (lzma.LZMADecompressor 方法), 539
decompress() (zlib.Decompress 方法), 527
decompress() (在 bz2 模块中), 535
decompress() (在 gzip 模块中), 530
decompress() (在 lzma 模块中), 539
decompress() (在 zlib 模块中), 526
decompressobj() (在 zlib 模块中), 527
decorator -- 装饰器, **2066**
dedent() (在 textwrap 模块中), 155
DEDENT() (在 token 模块中), 1980
deepcopy() (在 copy 模块中), 282
def_prog_mode() (在 curses 模块中), 855
def_shell_mode() (在 curses 模块中), 855
default() (cmd.Cmd 方法), 1492
default() (json.JSONEncoder 方法), 1218
default() (在 email.policy 模块中), 1178
DEFAULT() (在 unittest.mock 模块中), 1681
DEFAULT_BUFFER_SIZE() (在 io 模块中), 679
default_bufsize() (在 xml.dom.pulldom 模块中), 1288
default_exception_handler() (asyncio.loop 方法), 1037
default_factory (collections.defaultdict 属性), 247
DEFAULT_FORMAT() (在 tarfile 模块中), 555
DEFAULT_IGNORES() (在 filecmp 模块中), 448
default_loader() (在 xml.etree.ElementInclude 模块中), 1266
default_max_str_digits (sys.int_info 属性), 1800
default_open() (urllib.request.BaseHandler 方法), 1329
DEFAULT_PROTOCOL() (在 pickle 模块中), 471
DEFAULT_TIMEOUT(unittest.mock.ThreadingMock 属性), 1666
default_timer() (在 timeit 模块中), 1754
DefaultContext() (在 decimal 模块中), 336
DefaultCookiePolicy(http.cookiejar 中的类), 1402
defaultdict (collections 中的类), 247
DefaultDict (typing 中的类), 1593
DefaultEventLoopPolicy (asyncio 中的类), 1061
DefaultHandler() (xml.parsers.expat.xmlparser 方法), 1304
DefaultHandlerExpand() (xml.parsers.expat.xmlparser 方法), 1304
--default-pip
    ensurepip 命令行选项, 1772
defaults() (configparser.ConfigParser 方法), 589
DefaultSelector (selectors 中的类), 1142
defaultTestLoader() (在 unittest 模块中), 1646
defaultTestResult() (unittest.TestCase 方法), 1638
default (inspect.Parameter 属性), 1883
default (optparse.Option 属性), 838
defects (email.headerregistry.BaseHeader 属性), 1181
defects (email.message.EmailMessage 属性), 1167
defects (email.message.Message 属性), 1201
defpath() (在 os 模块中), 676
DefragResultBytes (urllib.parse 中的类), 1344
DefragResult (urllib.parse 中的类), 1344
degrees() (在 math 模块中), 318
degrees() (在 turtle 模块中), 1470
del
    statement -- 语句, 45, 83
DEL() (在 curses.ascii 模块中), 881
del_param() (email.message.EmailMessage 方法), 1163
del_param() (email.message.Message 方法), 1199
delattr()
    built-in function, 12
delay() (在 turtle 模块中), 1481
delay_output() (在 curses 模块中), 855
delayload(http.cookiejar.FileCookieJar 属

性），1404
delch() （curses.window 方法），862
dele() （poplib.POP3 方法），1367
delete() （ftplib.FTP 方法），1362
delete() （imaplib.IMAP4 方法），1370
delete() （tkinter.ttk.Treeview 方法），1530
DELETE_ATTR (*opcode*), 2007
DELETE_DEREF (*opcode*), 2011
DELETE_FAST (*opcode*), 2011
DELETE_GLOBAL (*opcode*), 2007
DELETE_NAME (*opcode*), 2006
DELETE_SUBSCR (*opcode*), 2003
deleteacl() （imaplib.IMAP4 方法），1370
deletefilehandler() （_tkinter.Widget.tk 方法），1513
DeleteKey() （在 winreg 模块中），2022
DeleteKeyEx() （在 winreg 模块中），2023
deleteln() （curses.window 方法），862
deleteMe() （bdb.Breakpoint 方法），1731
DeleteValue() （在 winreg 模块中），2023
Delete （ast 中的类），1956
delimiter （csv.Dialect 属性），573
delitem() （在 operator 模块中），405
deliver_challenge() （在 multiprocessing.connection 模块中），927
delocalize() （在 locale 模块中），1453
Del （ast 中的类），1948
demo_app() （在 wsgiref.simple_server 模块中），1315
denominator （fractions.Fraction 属性），352
denominator （numbers.Rational 属性），310
deprecated() （在 warnings 模块中），1831
DeprecationWarning, 107
dequeue() （logging.handlers.QueueListener 方法），741
deque （collections 中的类），243
Deque （typing 中的类），1593
DER_cert_to_PEM_cert()（在 ssl 模块中），1106
derive() （BaseExceptionGroup 方法），108
derwin() （curses.window 方法），862
description （inspect.Parameter.kind 属性），1884
description （sqlite3.Cursor 属性），512
descriptor -- 描述器, **2066**
deserialize() （sqlite3.Connection 方法），508
dest （optparse.Option 属性），838
detach() （io.BufferedIOBase 方法），683
detach() （io.TextIOBase 方法），686
detach() （socket.socket 方法），1092
detach() （tkinter.ttk.Treeview 方法），1530
detach() （weakref.finalize 方法），272
Detach() （winreg.PyHKEY 方法），2029
DETACHED_PROCESS() （在 subprocess 模块中），963
--details
    inspect 命令行选项, 1893
detect_api_mismatch() （在 test.support 模块中），1717
detect_encoding()（在 tokenize 模块中），1986
dev_mode （sys.flags 属性），1793
device_encoding() （在 os 模块中），625
devmajor （tarfile.TarInfo 属性），560
devminor （tarfile.TarInfo 属性），560
devnull() （在 os 模块中），676
DEVNULL() （在 subprocess 模块中），953
devpoll() （在 select 模块中），1134
DevpollSelector （selectors 中的类），1143
dgettext() （在 gettext 模块中），1442
dgettext() （在 locale 模块中），1454
Dialect （csv 中的类），571
dialect （csv.csvreader 属性），574
dialect （csv.csvwriter 属性），574
Dialog （tkinter.commondialog 中的类），1517
Dialog （tkinter.simpledialog 中的类），1515
dict() （multiprocessing.managers.SyncManager 方法），920
DICT_MERGE (*opcode*), 2008
DICT_UPDATE (*opcode*), 2008
DictComp （ast 中的类），1952
dictConfig()（在 logging.config 模块中），718
dictionary -- 字典
    object -- 对象, 83
    type, 运算目标, 83
dictionary -- 字典, **2067**
dictionary comprehension -- 字典推导式, **2067**
dictionary view -- 字典视图, **2067**
DictReader （csv 中的类），570
DictWriter （csv 中的类），571
Dict （ast 中的类），1947
Dict （typing 中的类），1592
dict （内置类），83
diff_bytes() （在 difflib 模块中），146
diff_files （filecmp.dircmp 属性），448
difference() （frozenset 方法），82
difference_update() （frozenset 方法），83
Differ （difflib 中的类），143
difflib
    module, 143
digest() （hashlib.hash 方法），601
digest() （hashlib.shake 方法），602
digest() （hmac.HMAC 方法），610
digest() （在 hmac 模块中），610
digest_size （hmac.HMAC 属性），611
digit() （在 unicodedata 模块中），158
digits() （在 string 模块中），111
dig （sys.float_info 属性），1795
dir()
    built-in function, 12
dir() （ftplib.FTP 方法），1362
dircmp （filecmp 中的类），447
directory
    compileall 命令行选项, 1993
    site-packages, 1893

    traversal, 653, 654
    修改, 637
    创建, 642
    删除, 460, 644
    walking, 653, 654
--directory
    http.server 命令行选项, 1397
Directory (tkinter.filedialog 中的类), 1516
DirEntry (os 中的类), 645
dirname() (在 os.path 模块中), 435
dirs_double_event()
        (tkinter.filedialog.FileDialog
        方法), 1516
dirs_select_event()
        (tkinter.filedialog.FileDialog
        方法), 1516
DirsOnSysPath (test.support.import_helper
        中的类), 1724
DIRTYPE() (在 tarfile 模块中), 555
dis
    module, 1996
dis() (dis.Bytecode 方法), 1997
dis() (在 dis 模块中), 1998
dis() (在 pickletools 模块中), 2018
disable (*pdb command*), 1742
disable() (bdb.Breakpoint 方法), 1731
disable() (profile.Profile 方法), 1749
disable() (在 faulthandler 模块中), 1737
disable() (在 gc 模块中), 1872
disable() (在 logging 模块中), 714
DISABLE() (在 sys.monitoring 模块中), 1814
disable_faulthandler() (在 test.support 模
        块中), 1713
disable_gc() (在 test.support 模块中), 1714
disable_interspersed_args()
        (optparse.OptionParser 方 法),
        842
disabled (logging.Logger 属性), 703
DisableReflectionKey() (在 winreg 模块中),
        2026
disassemble() (在 dis 模块中), 1999
discard() (frozenset 方法), 83
discard() (mailbox.Mailbox 方法), 1222
discard() (mailbox.MH 方法), 1228
discard (http.cookiejar.Cookie 属性), 1408
discover() (unittest.TestLoader 方 法),
        1643
disk_usage() (在 shutil 模块中), 461
dispatch_call() (bdb.Bdb 方法), 1733
dispatch_exception() (bdb.Bdb 方法), 1733
dispatch_line() (bdb.Bdb 方法), 1733
dispatch_return() (bdb.Bdb 方法), 1733
dispatch_table (pickle.Pickler 属性), 472
DISPLAY, 1503
display (*pdb command*), 1743
display_name (email.headerregistry.Address
        属性), 1185
display_name (email.headerregistry.Group
        属性), 1185
displayhook() (在 sys 模块中), 1790
dist() (在 math 模块中), 318
distance() (在 turtle 模块中), 1469
distribution() (在 importlib.metadata 模块
        中), 1937
Distribution (importlib.metadata 中的类),
        1937
distutils
    module, 2056
dis 命令行选项
    -C, 1997
    -h, 1997
    --help, 1997
    -O, 1997
    --show-caches, 1997
    --show-offsets, 1997
divide() (decimal.Context 方法), 338
divide_int() (decimal.Context 方法), 338
DivisionByZero (decimal 中的类), 342
divmod()
    built-in function, 13
divmod() (decimal.Context 方法), 338
Div (ast 中的类), 1949
DLE() (在 curses.ascii 模块中), 880
DllCanUnloadNow() (在 ctypes 模块中), 777
DllGetClassObject() (在 ctypes 模块中), 777
dllhandle() (在 sys 模块中), 1790
dnd_start() (在 tkinter.dnd 模块中), 1520
DndHandler (tkinter.dnd 中的类), 1520
dngettext() (在 gettext 模块中), 1442
dnpgettext() (在 gettext 模块中), 1442
do_clear() (bdb.Bdb 方法), 1734
do_command() (curses.textpad.Textbox 方
        法), 878
do_GET() (http.server.SimpleHTTPRequestHandler
        方法), 1396
do_handshake() (ssl.SSLSocket 方法), 1114
do_HEAD() (http.server.SimpleHTTPRequestHandler
        方法), 1396
do_help() (cmd.Cmd 方法), 1492
do_POST() (http.server.CGIHTTPRequestHandler
        方法), 1397
doc_header (cmd.Cmd 属性), 1493
DocCGIXMLRPCRequestHandler (xmlrpc.server
        中的类), 1421
DocFileSuite() (在 doctest 模块中), 1613
doClassCleanups() (unittest.TestCase 类 方
        法), 1639
doCleanups() (unittest.TestCase 方 法),
        1639
docmd() (smtplib.SMTP 方法), 1376
docstring -- 文档字符串, **2067**
docstring (doctest.DocTest 属性), 1615
doctest
    module, 1601
DocTestFailure, 1621

DocTestFinder (doctest 中的类), 1616
DocTestParser (doctest 中的类), 1617
DocTestRunner (doctest 中的类), 1617
DocTestSuite() (在 doctest 模块中), 1613
doctest 命令行选项
    -f, 1604
    --fail-fast, 1604
    -o, 1604
    --option, 1604
    -v, 1604
    --verbose, 1604
DocTest (doctest 中的类), 1615
doctype() (xml.etree.ElementTree.TreeBuilder 方法), 1271
documentElement (xml.dom.Document 属性), 1278
DocXMLRPCRequestHandler (xmlrpc.server 中的类), 1421
DocXMLRPCServer(xmlrpc.server 中的类), 1421
doc (json.JSONDecodeError 属性), 1218
domain_initial_dot (http.cookiejar.Cookie 属性), 1408
domain_return_ok() (http.cookiejar.CookiePolicy 方法), 1405
domain_specified(http.cookiejar.Cookie 属性), 1408
DomainFilter (tracemalloc 中的类), 1766
DomainLiberal(http.cookiejar.DefaultCookiePolicy 属性), 1407
DomainRFC2965Match (http.cookiejar.DefaultCookiePolicy 属性), 1407
DomainStrictNoDots (http.cookiejar.DefaultCookiePolicy 属性), 1407
DomainStrictNonDomain (http.cookiejar.DefaultCookiePolicy 属性), 1407
DomainStrict(http.cookiejar.DefaultCookiePolicy 属性), 1407
domain (email.headerregistry.Address 属性), 1185
domain (http.cookiejar.Cookie 属性), 1408
domain (http.cookies.Morsel 属性), 1399
domain (tracemalloc.DomainFilter 属性), 1766
domain (tracemalloc.Filter 属性), 1766
domain (tracemalloc.Trace 属性), 1769
DOMEventStream (xml.dom.pulldom 中的类), 1288
DOMException, 1280
doModuleCleanups() (在 unittest 模块中), 1649
DomstringSizeErr, 1281
done() (asyncio.Future 方法), 1046
done() (asyncio.Task 方法), 999
done() (concurrent.futures.Future 方法), 949
done() (graphlib.TopologicalSorter 方法), 307
done() (在 turtle 模块中), 1483
DONT_ACCEPT_BLANKLINE()(在 doctest 模块中), 1608
DONT_ACCEPT_TRUE_FOR_1() (在 doctest 模块中), 1608
dont_write_bytecode() (在 sys 模块中), 1790
dont_write_bytecode (sys.flags 属性), 1793
doRollover() (logging.handlers.RotatingFileHandler 方法), 732
doRollover() (logging.handlers.TimedRotatingFileHandler 方法), 733
DOT() (在 token 模块中), 1982
dot() (在 turtle 模块中), 1467
DOTALL() (在 re 模块中), 130
doublequote (csv.Dialect 属性), 573
DOUBLESLASH() (在 token 模块中), 1983
DOUBLESLASHEQUAL() (在 token 模块中), 1983
DOUBLESTAR() (在 token 模块中), 1983
DOUBLESTAREQUAL() (在 token 模块中), 1983
doupdate() (在 curses 模块中), 855
down (*pdb command*), 1741
down() (在 turtle 模块中), 1470
dpgettext() (在 gettext 模块中), 1442
drain() (asyncio.StreamWriter 方法), 1006
drive (pathlib.PurePath 属性), 415
drop_whitespace (textwrap.TextWrapper 属性), 156
dropwhile() (在 itertools 模块中), 382
dst() (datetime.datetime 方法), 207
dst() (datetime.time 方法), 214
dst() (datetime.timezone 方法), 222
dst() (datetime.tzinfo 方法), 216
DTDHandler (xml.sax.handler 中的类), 1290
duck-typing -- 鸭子类型, **2067**
dump() (pickle.Pickler 方法), 472
dump() (tracemalloc.Snapshot 方法), 1767
dump() (在 ast 模块中), 1974
dump() (在 json 模块中), 1214
dump() (在 marshal 模块中), 487
dump() (在 pickle 模块中), 471
dump() (在 plistlib 模块中), 596
dump() (在 xml.etree.ElementTree 模块中), 1263
dump_stats() (profile.Profile 方法), 1750
dump_stats() (pstats.Stats 方法), 1750
dump_traceback() (在 faulthandler 模块中), 1736
dump_traceback_later() (在 faulthandler 模块中), 1737
dumps() (在 json 模块中), 1215
dumps() (在 marshal 模块中), 488
dumps() (在 pickle 模块中), 471
dumps() (在 plistlib 模块中), 596
dumps() (在 xmlrpc.client 模块中), 1416
dup() (socket.socket 方法), 1092

dup() (在 os 模块中), 625
dup2() (在 os 模块中), 625
DuplicateOptionError, 592
DuplicateSectionError, 592
--durations
    unittest 命令行选项, 1625
dwFlags (subprocess.STARTUPINFO 属性), 961
DynamicClassAttribute() (在 types 模块中), 281

E

-e
    calendar 命令行选项, 237
    compileall 命令行选项, 1993
    idle 命令行选项, 1544
    tarfile 命令行选项, 565
    tokenize 命令行选项, 1986
    zipfile 命令行选项, 551
e() (在 cmath 模块中), 324
e() (在 math 模块中), 320
E2BIG() (在 errno 模块中), 746
EACCES() (在 errno 模块中), 746
EADDRINUSE() (在 errno 模块中), 750
EADDRNOTAVAIL() (在 errno 模块中), 750
EADV() (在 errno 模块中), 749
EAFNOSUPPORT() (在 errno 模块中), 750
EAFP, **2067**
EAGAIN() (在 errno 模块中), 746
eager_task_factory() (在 asyncio 模块中), 992
EALREADY() (在 errno 模块中), 751
east_asian_width() (在 unicodedata 模块中), 158
EAUTH() (在 errno 模块中), 752
EBADARCH() (在 errno 模块中), 752
EBADE() (在 errno 模块中), 748
EBADEXEC() (在 errno 模块中), 752
EBADF() (在 errno 模块中), 746
EBADFD() (在 errno 模块中), 749
EBADMACHO() (在 errno 模块中), 752
EBADMSG() (在 errno 模块中), 749
EBADR() (在 errno 模块中), 748
EBADRPC() (在 errno 模块中), 753
EBADRQC() (在 errno 模块中), 748
EBADSLT() (在 errno 模块中), 748
EBFONT() (在 errno 模块中), 748
EBUSY() (在 errno 模块中), 746
ECANCELED() (在 errno 模块中), 753
ECHILD() (在 errno 模块中), 746
echo() (在 curses 模块中), 855
echochar() (curses.window 方法), 862
ECHRNG() (在 errno 模块中), 748
ECOMM() (在 errno 模块中), 749
ECONNABORTED() (在 errno 模块中), 750
ECONNREFUSED() (在 errno 模块中), 751
ECONNRESET() (在 errno 模块中), 750
EDEADLK() (在 errno 模块中), 747
EDEADLOCK() (在 errno 模块中), 748
EDESTADDRREQ() (在 errno 模块中), 750
EDEVERR() (在 errno 模块中), 752
edit() (curses.textpad.Textbox 方法), 878
EDOM() (在 errno 模块中), 747
EDOTDOT() (在 errno 模块中), 749
EDQUOT() (在 errno 模块中), 751
EEXIST() (在 errno 模块中), 746
EFAULT() (在 errno 模块中), 746
EFBIG() (在 errno 模块中), 747
EFD_CLOEXEC() (在 os 模块中), 656
EFD_NONBLOCK() (在 os 模块中), 657
EFD_SEMAPHORE() (在 os 模块中), 657
effective() (在 bdb 模块中), 1735
EFTYPE() (在 errno 模块中), 752
ehlo() (smtplib.SMTP 方法), 1376
ehlo_or_helo_if_needed() (smtplib.SMTP 方法), 1376
EHOSTDOWN() (在 errno 模块中), 751
EHOSTUNREACH() (在 errno 模块中), 751
EIDRM() (在 errno 模块中), 747
EILSEQ() (在 errno 模块中), 749
EINPROGRESS() (在 errno 模块中), 751
EINTR() (在 errno 模块中), 746
EINVAL() (在 errno 模块中), 746
EIO() (在 errno 模块中), 746
EISCONN() (在 errno 模块中), 751
EISDIR() (在 errno 模块中), 746
EISNAM() (在 errno 模块中), 751
EJECT (enum.FlagBoundary 属性), 302
EKEYEXPIRED() (在 errno 模块中), 752
EKEYREJECTED() (在 errno 模块中), 752
EKEYREVOKED() (在 errno 模块中), 752
EL2HLT() (在 errno 模块中), 748
EL2NSYNC() (在 errno 模块中), 748
EL3HLT() (在 errno 模块中), 748
EL3RST() (在 errno 模块中), 748
element_create() (tkinter.ttk.Style 方法), 1535
element_names() (tkinter.ttk.Style 方法), 1536
element_options() (tkinter.ttk.Style 方法), 1536
ElementDeclHandler() (xml.parsers.expat.xmlparser 方法), 1303
elements() (collections.Counter 方法), 241
ElementTree (xml.etree.ElementTree 中的类), 1269
Element(xml.etree.ElementTree 中的类), 1266
ELIBACC() (在 errno 模块中), 749
ELIBBAD() (在 errno 模块中), 749
ELIBEXEC() (在 errno 模块中), 749
ELIBMAX() (在 errno 模块中), 749
ELIBSCN() (在 errno 模块中), 749
ELLIPSIS() (在 doctest 模块中), 1608
ELLIPSIS() (在 token 模块中), 1984
EllipsisType() (在 types 模块中), 279
Ellipsis (内置变量), 33

ELNRNG() (在 errno 模块中), 748
ELOCKUNMAPPED() (在 errno 模块中), 752
ELOOP() (在 errno 模块中), 747
EM() (在 curses.ascii 模块中), 881
email
    module, 1159
email.charset
    module, 1206
email.contentmanager
    module, 1185
email.encoders
    module, 1208
email.errors
    module, 1179
email.generator
    module, 1170
email.header
    module, 1204
email.headerregistry
    module, 1180
email.iterators
    module, 1211
email.message
    module, 1160
EmailMessage (email.message 中的类), 1160
email.mime
    module, 1201
email.mime.application
    module, 1202
email.mime.audio
    module, 1202
email.mime.base
    module, 1201
email.mime.image
    module, 1203
email.mime.message
    module, 1203
email.mime.multipart
    module, 1202
email.mime.nonmultipart
    module, 1202
email.mime.text
    module, 1203
email.parser
    module, 1167
email.policy
    module, 1173
EmailPolicy (email.policy 中的类), 1176
email.utils
    module, 1208
EMEDIUMTYPE() (在 errno 模块中), 752
EMFILE() (在 errno 模块中), 747
emit() (logging.FileHandler 方法), 730
emit() (logging.Handler 方法), 708
emit() (logging.handlers.BufferingHandler 方法), 738
emit() (logging.handlers.DatagramHandler 方法), 735
emit() (logging.handlers.HTTPHandler 方法), 739
emit() (logging.handlers.NTEventLogHandler 方法), 737
emit() (logging.handlers.QueueHandler 方法), 740
emit() (logging.handlers.RotatingFileHandler 方法), 732
emit() (logging.handlers.SMTPHandler 方法), 738
emit() (logging.handlers.SocketHandler 方法), 734
emit() (logging.handlers.SysLogHandler 方法), 736
emit() (logging.handlers.TimedRotatingFileHandler 方法), 734
emit() (logging.handlers.WatchedFileHandler 方法), 731
emit() (logging.NullHandler 方法), 730
emit() (logging.StreamHandler 方法), 729
EMLINK() (在 errno 模块中), 747
Empty, 972
empty() (asyncio.Queue 方法), 1019
empty() (multiprocessing.Queue 方法), 908
empty() (multiprocessing.SimpleQueue 方法), 909
empty() (queue.Queue 方法), 972
empty() (queue.SimpleQueue 方法), 974
empty() (sched.scheduler 方法), 970
EMPTY_NAMESPACE() (在 xml.dom 模块中), 1274
emptyline() (cmd.Cmd 方法), 1492
empty (inspect.Parameter 属性), 1883
empty (inspect.Signature 属性), 1882
emscripten_version (sys._emscripten_info 属性), 1790
EMSGSIZE() (在 errno 模块中), 750
EMULTIHOP() (在 errno 模块中), 749
enable (*pdb command*), 1742
enable() (bdb.Breakpoint 方法), 1731
enable() (imaplib.IMAP4 方法), 1370
enable() (profile.Profile 方法), 1749
enable() (在 faulthandler 模块中), 1736
enable() (在 gc 模块中), 1872
enable_callback_tracebacks() (在 sqlite3 模块中), 498
enable_interspersed_args() (optparse.OptionParser 方法), 843
enable_load_extension() (sqlite3.Connection 方法), 505
enable_traversal() (tkinter.ttk.Notebook 方法), 1527
ENABLE_USER_SITE() (在 site 模块中), 1895
enabled (bdb.Breakpoint 属性), 1732
EnableReflectionKey() (在 winreg 模块中), 2026
ENAMETOOLONG() (在 errno 模块中), 747
ENAVAIL() (在 errno 模块中), 751

enclose() (curses.window 方法), 862
encode
    编解码器, 173
encode() (codecs.Codec 方法), 178
encode() (codecs.IncrementalEncoder 方法), 179
encode() (email.header.Header 方法), 1205
encode() (json.JSONEncoder 方法), 1218
encode() (str 方法), 49
encode() (xmlrpc.client.Binary 方法), 1413
encode() (xmlrpc.client.DateTime 方法), 1412
encode() (在 base64 模块中), 1244
encode() (在 codecs 模块中), 173
encode() (在 quopri 模块中), 1247
encode_7or8bit() (在 email.encoders 模块中), 1208
encode_base64() (在 email.encoders 模块中), 1208
encode_noop() (在 email.encoders 模块中), 1208
encode_quopri() (在 email.encoders 模块中), 1208
encode_rfc2231() (在 email.utils 模块中), 1210
encodebytes() (在 base64 模块中), 1244
EncodedFile() (在 codecs 模块中), 175
encodePriority() (logging.handlers.SysLogHandler 方法), 736
encodestring() (在 quopri 模块中), 1247
encoder (codecs.CodecInfo 属性), 174
encoding
    base64, 1241
    quoted-printable, 1246
--encoding
    calendar 命令行选项, 237
ENCODING() (在 tarfile 模块中), 554
ENCODING() (在 token 模块中), 1981
encodings_map() (在 mimetypes 模块中), 1240
encodings_map (mimetypes.MimeTypes 属性), 1240
encodings.idna
    module, 188
encodings.mbcs
    module, 188
encodings.utf_8_sig
    module, 189
EncodingWarning, 107
encoding (curses.window 属性), 863
encoding (io.TextIOBase 属性), 686
encoding (UnicodeError 属性), 105
end() (re.Match 方法), 137
end() (xml.etree.ElementTree.TreeBuilder 方法), 1270
END_ASYNC_FOR (*opcode*), 2004
end_col_offset (ast.AST 属性), 1944
end_colno (traceback.FrameSummary 属性), 1867
end_fill() (在 turtle 模块中), 1473
END_FOR (*opcode*), 2001
end_headers() (http.server.BaseHTTPRequestHandler 方法), 1395
end_lineno (ast.AST 属性), 1944
end_lineno (SyntaxError 属性), 104
end_lineno (traceback.FrameSummary 属性), 1867
end_lineno (traceback.TracebackException 属性), 1865
end_ns() (xml.etree.ElementTree.TreeBuilder 方法), 1271
end_offset (SyntaxError 属性), 104
end_offset (traceback.TracebackException 属性), 1865
end_poly() (在 turtle 模块中), 1478
END_SEND (*opcode*), 2001
endCDATA() (xml.sax.handler.LexicalHandler 方法), 1295
EndCdataSectionHandler() (xml.parsers.expat.xmlparser 方法), 1304
EndDoctypeDeclHandler() (xml.parsers.expat.xmlparser 方法), 1303
endDocument() (xml.sax.handler.ContentHandler 方法), 1292
endDTD() (xml.sax.handler.LexicalHandler 方法), 1294
endElement() (xml.sax.handler.ContentHandler 方法), 1293
EndElementHandler() (xml.parsers.expat.xmlparser 方法), 1303
endElementNS() (xml.sax.handler.ContentHandler 方法), 1293
endheaders() (http.client.HTTPConnection 方法), 1356
ENDMARKER() (在 token 模块中), 1981
EndNamespaceDeclHandler() (xml.parsers.expat.xmlparser 方法), 1304
endpos (re.Match 属性), 137
endPrefixMapping() (xml.sax.handler.ContentHandler 方法), 1292
endswith() (bytearray 方法), 64
endswith() (bytes 方法), 64
endswith() (str 方法), 50
endwin() (在 curses 模块中), 855
end (UnicodeError 属性), 105
ENEEDAUTH() (在 errno 模块中), 752
ENETDOWN() (在 errno 模块中), 750
ENETRESET() (在 errno 模块中), 750
ENETUNREACH() (在 errno 模块中), 750
ENFILE() (在 errno 模块中), 747
ENOANO() (在 errno 模块中), 748
ENOATTR() (在 errno 模块中), 752

ENOBUFS() (在 errno 模块中), 751
ENOCSI() (在 errno 模块中), 748
ENODATA() (在 errno 模块中), 748
ENODEV() (在 errno 模块中), 746
ENOENT() (在 errno 模块中), 746
ENOEXEC() (在 errno 模块中), 746
ENOKEY() (在 errno 模块中), 752
ENOLCK() (在 errno 模块中), 747
ENOLINK() (在 errno 模块中), 749
ENOMEDIUM() (在 errno 模块中), 751
ENOMEM() (在 errno 模块中), 746
ENOMSG() (在 errno 模块中), 747
ENONET() (在 errno 模块中), 748
ENOPKG() (在 errno 模块中), 749
ENOPOLICY() (在 errno 模块中), 752
ENOPROTOOPT() (在 errno 模块中), 750
ENOSPC() (在 errno 模块中), 747
ENOSR() (在 errno 模块中), 748
ENOSTR() (在 errno 模块中), 748
ENOSYS() (在 errno 模块中), 747
ENOTACTIVE() (在 errno 模块中), 752
ENOTBLK() (在 errno 模块中), 746
ENOTCAPABLE() (在 errno 模块中), 753
ENOTCONN() (在 errno 模块中), 751
ENOTDIR() (在 errno 模块中), 746
ENOTEMPTY() (在 errno 模块中), 747
ENOTNAM() (在 errno 模块中), 751
ENOTRECOVERABLE() (在 errno 模块中), 753
ENOTSOCK() (在 errno 模块中), 750
ENOTSUP() (在 errno 模块中), 750
ENOTTY() (在 errno 模块中), 747
ENOTUNIQ() (在 errno 模块中), 749
ENQ() (在 curses.ascii 模块中), 880
enqueue() (logging.handlers.QueueHandler 方法), 740
enqueue_sentinel() (logging.handlers.QueueListener 方法), 741
ensure_directories() (venv.EnvBuilder 方法), 1777
ensure_future() (在 asyncio 模块中), 1045
ensurepip
    module, 1771
ensurepip 命令行选项
    --altinstall, 1772
    --default-pip, 1772
    --root, 1772
    --user, 1772
enter() (sched.scheduler 方法), 970
enter_async_context() (contextlib.AsyncExitStack 方法), 1850
enter_context() (contextlib.ExitStack 方法), 1849
enterabs() (sched.scheduler 方法), 970
enterAsyncContext() (unittest.IsolatedAsyncioTestCase 方法), 1640
enterClassContext() (unittest.TestCase 类方法), 1639
enterContext() (unittest.TestCase 方法), 1639
enterModuleContext() (在 unittest 模块中), 1649
entities (xml.dom.DocumentType 属性), 1277
EntityDeclHandler() (xml.parsers.expat.xmlparser 方法), 1303
entitydefs() (在 html.entities 模块中), 1254
EntityResolver (xml.sax.handler 中的类), 1290
entry_points() (在 importlib.metadata 模块中), 1933
EntryPoints (importlib.metadata 中的类), 1934
EntryPoint (importlib.metadata 中的类), 1934
enum
    module, 291
enum_certificates() (在 ssl 模块中), 1106
enum_crls() (在 ssl 模块中), 1106
EnumCheck (enum 中的类), 301
EnumDict (enum 中的类), 303
enumerate()
    built-in function, 13
enumerate() (在 threading 模块中), 886
EnumKey() (在 winreg 模块中), 2023
EnumType (enum 中的类), 293
EnumValue() (在 winreg 模块中), 2023
Enum (enum 中的类), 294
EnvBuilder (venv 中的类), 1776
environ() (在 os 模块中), 617
environ() (在 posix 模块中), 2034
environb() (在 os 模块中), 617
EnvironmentError, 105
EnvironmentVarGuard (test.support.os_helper 中的类), 1722
ENXIO() (在 errno 模块中), 746
eof_received() (asyncio.BufferedProtocol 方法), 1054
eof_received() (asyncio.Protocol 方法), 1053
EOFError, 101
eof (bz2.BZ2Decompressor 属性), 534
eof (lzma.LZMADecompressor 属性), 539
eof (shlex.shlex 属性), 1499
eof (ssl.MemoryBIO 属性), 1131
eof (zlib.Decompress 属性), 527
EOPNOTSUPP() (在 errno 模块中), 750
EOT() (在 curses.ascii 模块中), 880
EOVERFLOW() (在 errno 模块中), 749
EOWNERDEAD() (在 errno 模块中), 753
EPERM() (在 errno 模块中), 745
EPFNOSUPPORT() (在 errno 模块中), 750
epilogue (email.message.EmailMessage 属性), 1167

epilogue (email.message.Message 属性), 1201
EPIPE() (在 errno 模块中), 747
epoch, 690
epoll() (在 select 模块中), 1134
EpollSelector (selectors 中的类), 1143
EPROCLIM() (在 errno 模块中), 753
EPROCUNAVAIL() (在 errno 模块中), 753
EPROGMISMATCH() (在 errno 模块中), 753
EPROGUNAVAIL() (在 errno 模块中), 753
EPROTO() (在 errno 模块中), 749
EPROTONOSUPPORT() (在 errno 模块中), 750
EPROTOTYPE() (在 errno 模块中), 750
epsilon (sys.float_info 属性), 1795
EPWROFF() (在 errno 模块中), 753
eq() (在 operator 模块中), 402
EQEQUAL() (在 token 模块中), 1982
EQFULL() (在 errno 模块中), 751
EQUAL() (在 token 模块中), 1982
Eq (ast 中的类), 1950
ERA() (在 locale 模块中), 1451
ERA_D_FMT() (在 locale 模块中), 1451
ERA_D_T_FMT() (在 locale 模块中), 1451
ERA_T_FMT() (在 locale 模块中), 1451
ERANGE() (在 errno 模块中), 747
erase() (curses.window 方法), 863
erasechar() (在 curses 模块中), 855
EREMCHG() (在 errno 模块中), 749
EREMOTE() (在 errno 模块中), 749
EREMOTEIO() (在 errno 模块中), 751
ERESTART() (在 errno 模块中), 749
erf() (在 math 模块中), 319
erfc() (在 math 模块中), 319
ERFKILL() (在 errno 模块中), 752
EROFS() (在 errno 模块中), 747
ERPCMISMATCH() (在 errno 模块中), 753
ERR() (在 curses 模块中), 866
errcheck (ctypes._CFuncPtr 属性), 774
errcode (xmlrpc.client.ProtocolError 属性), 1414
errmsg (xmlrpc.client.ProtocolError 属性), 1414
errno
    module, 102, 745
errno (OSError 属性), 102
Error, 282, 462, 514, 573, 592, 1237, 1246, 1310, 1437, 1448
error, 167, 488, 491493, 525, 615, 854, 978, 1079, 1134, 1300, 2042, 2052
error() (argparse.ArgumentParser 方法), 811
error() (logging.Logger 方法), 705
error() (urllib.request.OpenerDirector 方法), 1328
error() (xml.sax.handler.ErrorHandler 方法), 1294
ERROR() (在 logging 模块中), 707
error() (在 logging 模块中), 714
ERROR() (在 tkinter.messagebox 模块中), 1519
error_body (wsgiref.handlers.BaseHandler 属性), 1319
error_content_type (http.server.BaseHTTPRequestHandler 属性), 1393
error_headers (wsgiref.handlers.BaseHandler 属性), 1319
error_leader() (shlex.shlex 方法), 1498
error_message_format (http.server.BaseHTTPRequestHandler 属性), 1393
error_output() (wsgiref.handlers.BaseHandler 方法), 1318
error_perm, 1364
error_proto, 1364, 1366
error_received() (asyncio.DatagramProtocol 方法), 1054
error_reply, 1364
error_status (wsgiref.handlers.BaseHandler 属性), 1319
error_temp, 1364
ErrorByteIndex (xml.parsers.expat.xmlparser 属性), 1302
errorcode() (在 errno 模块中), 745
ErrorCode (xml.parsers.expat.xmlparser 属性), 1302
ErrorColumnNumber (xml.parsers.expat.xmlparser 属性), 1302
ErrorHandler (xml.sax.handler 中的类), 1290
errorlevel (tarfile.TarFile 属性), 558
ErrorLineNumber (xml.parsers.expat.xmlparser 属性), 1302
ErrorStream (wsgiref.types 中的类), 1320
ErrorString() (在 xml.parsers.expat 模块中), 1300
errors (io.TextIOBase 属性), 686
errors (unittest.TestLoader 属性), 1642
errors (unittest.TestResult 属性), 1644
ERRORTOKEN() (在 token 模块中), 1981
ESC() (在 curses.ascii 模块中), 881
escape() (在 glob 模块中), 454
escape() (在 html 模块中), 1249
escape() (在 re 模块中), 133
escape() (在 xml.sax.saxutils 模块中), 1295
escapechar (csv.Dialect 属性), 573
escapedquotes (shlex.shlex 属性), 1498
escape (shlex.shlex 属性), 1498
ESHLIBVERS() (在 errno 模块中), 753
ESHUTDOWN() (在 errno 模块中), 751
ESOCKTNOSUPPORT() (在 errno 模块中), 750
ESPIPE() (在 errno 模块中), 747
ESRCH() (在 errno 模块中), 746
ESRMNT() (在 errno 模块中), 749
ESTALE() (在 errno 模块中), 751
ESTRPIPE() (在 errno 模块中), 750
ETB() (在 curses.ascii 模块中), 881
ETH_P_ALL() (在 socket 模块中), 1082

ETHERTYPE_ARP() (在 socket 模块中), 1083
ETHERTYPE_IP() (在 socket 模块中), 1083
ETHERTYPE_IPV6() (在 socket 模块中), 1083
ETHERTYPE_VLAN() (在 socket 模块中), 1083
ETIME() (在 errno 模块中), 748
ETIMEDOUT() (在 errno 模块中), 751
Etiny() (decimal.Context 方法), 338
ETOOMANYREFS() (在 errno 模块中), 751
Etop() (decimal.Context 方法), 338
ETX() (在 curses.ascii 模块中), 880
ETXTBSY() (在 errno 模块中), 747
EUCLEAN() (在 errno 模块中), 751
EUNATCH() (在 errno 模块中), 748
EUSERS() (在 errno 模块中), 750
eval
    内置函数, 95, 284, 285
eval()
    built-in function, 14
Event() (multiprocessing.managers.SyncManager 方法), 920
EVENT_READ() (在 selectors 模块中), 1141
EVENT_WRITE() (在 selectors 模块中), 1141
eventfd() (在 os 模块中), 656
eventfd_read() (在 os 模块中), 656
eventfd_write() (在 os 模块中), 656
EventLoop (asyncio 中的类), 1042
events (部件), 1511
events (selectors.SelectorKey 属性), 1141
Event (asyncio 中的类), 1010
Event (multiprocessing 中的类), 914
Event (threading 中的类), 895
EWOULDBLOCK() (在 errno 模块中), 747
EX_CANTCREAT() (在 os 模块中), 663
EX_CONFIG() (在 os 模块中), 663
EX_DATAERR() (在 os 模块中), 662
EX_IOERR() (在 os 模块中), 663
EX_NOHOST() (在 os 模块中), 662
EX_NOINPUT() (在 os 模块中), 662
EX_NOPERM() (在 os 模块中), 663
EX_NOTFOUND() (在 os 模块中), 663
EX_NOUSER() (在 os 模块中), 662
EX_OK() (在 os 模块中), 662
EX_OSERR() (在 os 模块中), 663
EX_OSFILE() (在 os 模块中), 663
EX_PROTOCOL() (在 os 模块中), 663
EX_SOFTWARE() (在 os 模块中), 662
EX_TEMPFAIL() (在 os 模块中), 663
EX_UNAVAILABLE() (在 os 模块中), 662
EX_USAGE() (在 os 模块中), 662
--exact
    tokenize 命令行选项, 1986
EXACT_TOKEN_TYPES() (在 token 模块中), 1984
examples (doctest.DocTest 属性), 1615
Example (doctest 中的类), 1615
example (doctest.DocTestFailure 属性), 1621
example (doctest.UnexpectedException 属性), 1621
exc_info() (在 sys 模块中), 1792
exc_info (doctest.UnexpectedException 属性), 1621
exc_msg (doctest.Example 属性), 1616
exc_type_str(traceback.TracebackException 属性), 1865
exc_type (traceback.TracebackException 属性), 1865
excel_tab (csv 中的类), 572
excel (csv 中的类), 572
except
    statement -- 语句, 99
ExceptHandler (ast 中的类), 1960
excepthook() (在 sys 模块中), 1791
excepthook() (在 threading 模块中), 886
Exception, 100
exception() (asyncio.Future 方法), 1047
exception() (asyncio.Task 方法), 999
exception() (concurrent.futures.Future 方法), 949
exception() (logging.Logger 方法), 706
EXCEPTION() (在 _tkinter 模块中), 1513
exception() (在 logging 模块中), 714
exception() (在 sys 模块中), 1791
EXCEPTION_HANDLED (*monitoring event*), 1811
ExceptionGroup, 108
exceptions (*pdb command*), 1745
exceptions (BaseExceptionGroup 属性), 108
exceptions (traceback.TracebackException 属性), 1864
EXCLAMATION() (在 token 模块中), 1984
EXDEV() (在 errno 模块中), 746
exec
    内置函数, 14, 95
exec()
    built-in function, 14
exec_module() (importlib.abc.InspectLoader 方法), 1914
exec_module() (importlib.abc.Loader 方法), 1913
exec_module() (importlib.abc.SourceLoader 方法), 1916
exec_module() (importlib.machinery.ExtensionFileLoader 方法), 1921
exec_module() (zipimport.zipimporter 方法), 1902
exec_prefix() (在 sys 模块中), 1792
execl() (在 os 模块中), 661
execle() (在 os 模块中), 661
execlp() (在 os 模块中), 661
execlpe() (在 os 模块中), 661
executable() (在 sys 模块中), 1792
execute() (sqlite3.Connection 方法), 501
execute() (sqlite3.Cursor 方法), 510
executemany() (sqlite3.Connection 方法), 501
executemany() (sqlite3.Cursor 方法), 510
executescript() (sqlite3.Connection 方法), 501

executescript() (sqlite3.Cursor 方法), 511
ExecutionLoader(importlib.abc 中的类), 1914
Executor (concurrent.futures 中的类), 945
execv() (在 os 模块中), 661
execve() (在 os 模块中), 661
execvp() (在 os 模块中), 661
execvpe() (在 os 模块中), 661
EXFULL() (在 errno 模块中), 748
exists() (pathlib.Path 方法), 425
exists() (tkinter.ttk.Treeview 方法), 1531
exists() (zipfile.Path 方法), 548
exists() (在 os.path 模块中), 436
exit() (argparse.ArgumentParser 方法), 811
exit() (在 _thread 模块中), 979
exit() (在 sys 模块中), 1792
exitcode (multiprocessing.Process 属性), 905
exitonclick() (在 turtle 模块中), 1485
ExitStack (contextlib 中的类), 1849
exit (内置变量), 34
exp() (decimal.Context 方法), 339
exp() (decimal.Decimal 方法), 331
exp() (在 cmath 模块中), 322
exp() (在 math 模块中), 317
exp2() (在 math 模块中), 317
expand() (re.Match 方法), 135
expand_tabs (textwrap.TextWrapper 属性), 156
ExpandEnvironmentStrings() (在 winreg 模块中), 2024
expandNode() (xml.dom.pulldom.DOMEventStream 方法), 1288
expandtabs() (bytearray 方法), 68
expandtabs() (bytes 方法), 68
expandtabs() (str 方法), 50
expanduser() (pathlib.Path 方法), 423
expanduser() (在 os.path 模块中), 436
expandvars() (在 os.path 模块中), 436
Expat, 1300
ExpatError, 1300
expectedFailure() (在 unittest 模块中), 1630
expectedFailures (unittest.TestResult 属性), 1644
expected (asyncio.IncompleteReadError 属性), 1022
expired() (asyncio.Timeout 方法), 994
expires (http.cookiejar.Cookie 属性), 1408
expires (http.cookies.Morsel 属性), 1399
exploded (ipaddress.IPv4Address 属性), 1424
exploded (ipaddress.IPv4Network 属性), 1429
exploded (ipaddress.IPv6Address 属性), 1426
exploded (ipaddress.IPv6Network 属性), 1431
expm1() (在 math 模块中), 317
expovariate() (在 random 模块中), 357
expression -- 表达式, **2067**
Expression (ast 中的类), 1945
Expr (ast 中的类), 1949
expunge() (imaplib.IMAP4 方法), 1370
extend() (序列方法), 45
extend() (array.array 方法), 267
extend() (collections.deque 方法), 244
extend() (xml.etree.ElementTree.Element 方法), 1267
extend_path() (在 pkgutil 模块中), 1903
EXTENDED_ARG (*opcode*), 2013
ExtendedContext() (在 decimal 模块中), 336
ExtendedInterpolation(configparser 中的类), 580
extendleft() (collections.deque 方法), 244
extension module -- 扩展模块, **2067**
EXTENSION_SUFFIXES() (在 importlib.machinery 模块中), 1918
ExtensionFileLoader (importlib.machinery 中的类), 1920
extensions_map(http.server.SimpleHTTPRequestHandler 属性), 1395
External Data Representation, 470
external_attr (zipfile.ZipInfo 属性), 550
ExternalClashError, 1237
ExternalEntityParserCreate() (xml.parsers.expat.xmlparser 方法), 1301
ExternalEntityRefHandler() (xml.parsers.expat.xmlparser 方法), 1304
--extract
    tarfile 命令行选项, 565
    zipfile 命令行选项, 551
extract() (tarfile.TarFile 方法), 557
extract() (zipfile.ZipFile 方法), 544
extract() (traceback.StackSummary 类方法), 1866
extract_cookies() (http.cookiejar.CookieJar 方法), 1403
extract_stack() (在 traceback 模块中), 1863
extract_tb() (在 traceback 模块中), 1863
extract_version (zipfile.ZipInfo 属性), 550
extractall() (tarfile.TarFile 方法), 557
extractall() (zipfile.ZipFile 方法), 545
ExtractError, 554
extractfile() (tarfile.TarFile 方法), 558
extraction_filter (tarfile.TarFile 属性), 558
extra (zipfile.ZipInfo 属性), 550
extsep() (在 os 模块中), 676

F

-f
    calendar 命令行选项, 237
    compileall 命令行选项, 1993
    doctest 命令行选项, 1604
    random 命令行选项, 362
    trace 命令行选项, 1759
    unittest 命令行选项, 1625
f 字符串, 57

f-string -- f-字符串, **2067**
F_CONTIGUOUS (inspect.BufferFlags 属性), 1892
f_contiguous (memoryview 属性), 81
F_LOCK() (在 os 模块中), 627
F_OK() (在 os 模块中), 637
F_TEST() (在 os 模块中), 627
F_TLOCK() (在 os 模块中), 627
F_ULOCK() (在 os 模块中), 627
fabs() (在 math 模块中), 314
factorial() (在 math 模块中), 313
factory() (importlib.util.LazyLoader 类方法), 1925
fail() (unittest.TestCase 方法), 1638
FAIL_FAST() (在 doctest 模块中), 1609
failed (doctest.TestResults 属性), 1617
--failfast
    unittest 命令行选项, 1625
--fail-fast
    doctest 命令行选项, 1604
failfast (unittest.TestResult 属性), 1645
failureException, 1614
failureException (unittest.TestCase 属性), 1638
failures (doctest.DocTestRunner 属性), 1618
failures (unittest.TestResult 属性), 1644
FakePath (test.support.os_helper 中的类), 1722
False, 35, 42
false, 35
False (内置对象), 35
False (内置变量), 33
families() (在 tkinter.font 模块中), 1514
family (socket.socket 属性), 1097
FancyURLopener (urllib.request 中的类), 1337
--fast
    gzip 命令行选项, 531
FastChildWatcher (asyncio 中的类), 1063
fast (pickle.Pickler 属性), 473
fatalError() (xml.sax.handler.ErrorHandler 方法), 1294
faultCode (xmlrpc.client.Fault 属性), 1414
faulthandler
    module, 1736
faultString (xmlrpc.client.Fault 属性), 1414
Fault (xmlrpc.client 中的类), 1414
fchdir() (在 os 模块中), 639
fchmod() (在 os 模块中), 625
fchown() (在 os 模块中), 626
fcntl
    module, 2039
fcntl() (在 fcntl 模块中), 2040
fd() (在 turtle 模块中), 1464
fd_count() (在 test.support.os_helper 模块中), 1722
fdatasync() (在 os 模块中), 626
fdopen() (在 os 模块中), 624
fd (selectors.SelectorKey 属性), 1141
feature_external_ges() (在 xml.sax.handler 模块中), 1291
feature_external_pes() (在 xml.sax.handler 模块中), 1291
feature_namespace_prefixes() (在 xml.sax.handler 模块中), 1290
feature_namespaces() (在 xml.sax.handler 模块中), 1290
feature_string_interning() (在 xml.sax.handler 模块中), 1291
feature_validation() (在 xml.sax.handler 模块中), 1291
FEBRUARY() (在 calendar 模块中), 236
feed() (email.parser.BytesFeedParser 方法), 1168
feed() (html.parser.HTMLParser 方法), 1250
feed() (xml.etree.ElementTree.XMLParser 方法), 1271
feed() (xml.etree.ElementTree.XMLPullParser 方法), 1272
feed() (xml.sax.xmlreader.IncrementalParser 方法), 1298
feed_eof() (asyncio.StreamReader 方法), 1004
FeedParser (email.parser 中的类), 1168
fetch() (imaplib.IMAP4 方法), 1370
fetchall() (sqlite3.Cursor 方法), 511
fetchmany() (sqlite3.Cursor 方法), 511
fetchone() (sqlite3.Cursor 方法), 511
FF() (在 curses.ascii 模块中), 880
fflags (select.kevent 属性), 1139
field() (在 dataclasses 模块中), 1835
field_size_limit() (在 csv 模块中), 570
fieldnames (csv.DictReader 属性), 574
fields() (在 dataclasses 模块中), 1836
fields (uuid.UUID 属性), 1381
Field (dataclasses 中的类), 1836
FIFOTYPE() (在 tarfile 模块中), 555
file
    compileall 命令行选项, 1993
    gzip 命令行选项, 531
--file
    trace 命令行选项, 1759
file object -- 文件对象
    io 模块, 677
    open() 内置函数, 22
file object -- 文件对象, **2067**
file-like object -- 文件型对象, **2067**
FILE_ATTRIBUTE_ARCHIVE() (在 stat 模块中), 446
FILE_ATTRIBUTE_COMPRESSED() (在 stat 模块中), 446
FILE_ATTRIBUTE_DEVICE() (在 stat 模块中), 446
FILE_ATTRIBUTE_DIRECTORY() (在 stat 模块中), 446
FILE_ATTRIBUTE_ENCRYPTED() (在 stat 模块中)

, 446
FILE_ATTRIBUTE_HIDDEN() (在 stat 模块中), 446
FILE_ATTRIBUTE_INTEGRITY_STREAM() (在 stat 模块中), 446
FILE_ATTRIBUTE_NO_SCRUB_DATA() (在 stat 模块中), 446
FILE_ATTRIBUTE_NORMAL() (在 stat 模块中), 446
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED() (在 stat 模块中), 446
FILE_ATTRIBUTE_OFFLINE() (在 stat 模块中), 446
FILE_ATTRIBUTE_READONLY() (在 stat 模块中), 446
FILE_ATTRIBUTE_REPARSE_POINT() (在 stat 模块中), 446
FILE_ATTRIBUTE_SPARSE_FILE() (在 stat 模块中), 446
FILE_ATTRIBUTE_SYSTEM() (在 stat 模块中), 446
FILE_ATTRIBUTE_TEMPORARY() (在 stat 模块中), 446
FILE_ATTRIBUTE_VIRTUAL() (在 stat 模块中), 446
file_digest() (在 hashlib 模块中), 602
file_open() (urllib.request.FileHandler 方法), 1332
file_size (zipfile.ZipInfo 属性), 551
filecmp
    module, 446
fileConfig() (在 logging.config 模块中), 719
FileCookieJar (http.cookiejar 中的类), 1402
FileDialog (tkinter.filedialog 中的类), 1516
FileExistsError, 106
FileFinder (importlib.machinery 中的类), 1919
FileHandler (logging 中的类), 730
FileHandler (urllib.request 中的类), 1326
fileinput
    module, 851
FileInput (fileinput 中的类), 853
FileIO (io 中的类), 684
filelineno() (在 fileinput 模块中), 852
FileLoader (importlib.abc 中的类), 1915
filemode() (在 stat 模块中), 442
filename() (在 fileinput 模块中), 852
filename2 (OSError 属性), 103
filename_only() (在 tabnanny 模块中), 1989
filename_pattern (tracemalloc.Filter 属性), 1767
filename (doctest.DocTest 属性), 1615
filename (http.cookiejar.FileCookieJar 属性), 1404
filename (inspect.FrameInfo 属性), 1888
filename (inspect.Traceback 属性), 1888
filename (netrc.NetrcParseError 属性), 594
filename (OSError 属性), 103
filename (SyntaxError 属性), 104
filename (traceback.FrameSummary 属性), 1867
filename (traceback.TracebackException 属性), 1865
filename (tracemalloc.Frame 属性), 1767
filename (zipfile.ZipFile 属性), 547
filename (zipfile.ZipInfo 属性), 550
fileno() (bz2.BZ2File 方法), 533
fileno() (http.client.HTTPResponse 方法), 1357
fileno() (io.IOBase 方法), 681
fileno() (multiprocessing.connection.Connection 方法), 912
fileno() (select.devpoll 方法), 1136
fileno() (select.epoll 方法), 1137
fileno() (select.kqueue 方法), 1138
fileno() (selectors.DevpollSelector 方法), 1143
fileno() (selectors.EpollSelector 方法), 1143
fileno() (selectors.KqueueSelector 方法), 1143
fileno() (socketserver.BaseServer 方法), 1386
fileno() (socket.socket 方法), 1092
fileno() (在 fileinput 模块中), 852
FileNotFoundError, 106
fileobj (selectors.SelectorKey 属性), 1141
files() (importlib.abc.TraversableResources 方法), 1918
files() (importlib.resources.abc.TraversableResources 方法), 1932
files() (在 importlib.metadata 模块中), 1935
files() (在 importlib.resources 模块中), 1928
files_double_event() (tkinter.filedialog.FileDialog 方法), 1516
files_select_event() (tkinter.filedialog.FileDialog 方法), 1516
filesystem encoding and error handler -- 文件系统编码格式与错误处理器, **2067**
FileType (argparse 中的类), 807
FileWrapper (wsgiref.types 中的类), 1320
FileWrapper (wsgiref.util 中的类), 1313
file (bdb.Breakpoint 属性), 1732
file (pyclbr.Class 属性), 1990
file (pyclbr.Function 属性), 1990
fill() (textwrap.TextWrapper 方法), 157
fill() (在 textwrap 模块中), 154
fillcolor() (在 turtle 模块中), 1472
filling() (在 turtle 模块中), 1473
fillvalue (reprlib.Repr 属性), 289
--filter
    tarfile 命令行选项, 565
filter()

built-in function, 15
filter() (logging.Filter 方法), 710
filter() (logging.Handler 方法), 708
filter() (logging.Logger 方法), 706
filter() (在 curses 模块中), 855
filter() (在 fnmatch 模块中), 456
filter_command() (tkinter.filedialog.FileDialog 方法), 1516
FILTER_DIR() (在 unittest.mock 模块中), 1683
filter_traces() (tracemalloc.Snapshot 方法), 1767
FilterError, 554
filterfalse() (在 itertools 模块中), 382
filterwarnings() (在 warnings 模块中), 1831
Filter (logging 中的类), 710
filter (select.kevent 属性), 1139
Filter (tracemalloc 中的类), 1766
Final() (在 typing 模块中), 1566
final() (在 typing 模块中), 1588
finalize (weakref 中的类), 271
find() (bytearray 方法), 64
find() (bytes 方法), 64
find() (doctest.DocTestFinder 方法), 1616
find() (mmap.mmap 方法), 1154
find() (str 方法), 50
find() (xml.etree.ElementTree.Element 方法), 1267
find() (xml.etree.ElementTree.ElementTree 方法), 1269
find() (在 gettext 模块中), 1442
find_class() (*pickle* 协议), 482
find_class() (pickle.Unpickler 方法), 473
find_library() (在 ctypes.util 模块中), 777
find_loader() (在 pkgutil 模块中), 1903
find_longest_match() (difflib.SequenceMatcher 方法), 147
find_msvcrt() (在 ctypes.util 模块中), 777
find_spec() (importlib.abc.MetaPathFinder 方法), 1912
find_spec() (importlib.abc.PathEntryFinder 方法), 1912
find_spec() (importlib.machinery.FileFinder 方法), 1919
find_spec() (zipimport.zipimporter 方法), 1902
find_spec() (importlib.machinery.PathFinder 类方法), 1919
find_spec() (在 importlib.util 模块中), 1924
find_unused_port() (在 test.support.socket_helper 模块中), 1718
find_user_password() (urllib.request.HTTPPasswordMgr 方法), 1331
find_user_password() (urllib.request.HTTPPasswordMgrWithPriorAuth 方法), 1331
findall() (re.Pattern 方法), 135
findall() (xml.etree.ElementTree.Element 方法), 1267
findall() (xml.etree.ElementTree.ElementTree 方法), 1269
findall() (在 re 模块中), 132
findCaller() (logging.Logger 方法), 706
finder -- 查找器, **2068**
findfile() (在 test.support 模块中), 1713
finditer() (re.Pattern 方法), 135
finditer() (在 re 模块中), 132
findlabels() (在 dis 模块中), 2000
findlinestarts() (在 dis 模块中), 1999
findtext() (xml.etree.ElementTree.Element 方法), 1267
findtext() (xml.etree.ElementTree.ElementTree 方法), 1269
finish() (socketserver.BaseRequestHandler 方法), 1388
finish() (tkinter.dnd.DndHandler 方法), 1520
finish_request() (socketserver.BaseServer 方法), 1387
FIRST_COMPLETED() (在 asyncio 模块中), 996
FIRST_COMPLETED() (在 concurrent.futures 模块中), 950
FIRST_EXCEPTION() (在 asyncio 模块中), 996
FIRST_EXCEPTION() (在 concurrent.futures 模块中), 950
firstChild (xml.dom.Node 属性), 1276
FirstHeaderLineIsContinuationDefect, 1180
firstkey() (dbm.gnu.gdbm 方法), 491
--first-weekday
calendar 命令行选项, 237
firstweekday() (在 calendar 模块中), 234
firstweekday (calendar.Calendar 属性), 231
fix_missing_locations() (在 ast 模块中), 1973
fix_sentence_endings (textwrap.TextWrapper 属性), 157
flag_bits (zipfile.ZipInfo 属性), 550
FlagBoundary (enum 中的类), 302
flags() (在 sys 模块中), 1792
flags (re.Pattern 属性), 135
flags (select.kevent 属性), 1139
Flag (enum 中的类), 298
flash() (在 curses 模块中), 855
flatten() (email.generator.BytesGenerator 方法), 1171
flatten() (email.generator.Generator 方法), 1172
flattening
objects, 469
float
内置函数, 36
--float
random 命令行选项, 362
float_info() (在 sys 模块中), 1794

float_repr_style() (在 sys 模块中), 1796
FloatingPointError, 101
FloatOperation (decimal 中的类), 343
float (内置类), 16
flock() (在 fcntl 模块中), 2041
floor division -- 向下取整除法, **2068**
floor() (在 *math* 模块中), 37
floor() (在 math 模块中), 314
floordiv() (在 operator 模块中), 403
FloorDiv (ast 中的类), 1949
flush() (bz2.BZ2Compressor 方法), 534
flush() (io.BufferedWriter 方法), 686
flush() (io.IOBase 方法), 681
flush() (logging.Handler 方法), 708
flush() (logging.handlers.BufferingHandler 方法), 738
flush() (logging.handlers.MemoryHandler 方法), 739
flush() (logging.StreamHandler 方法), 729
flush() (lzma.LZMACompressor 方法), 538
flush() (mailbox.Mailbox 方法), 1224
flush() (mailbox.Maildir 方法), 1226
flush() (mailbox.MH 方法), 1228
flush() (mmap.mmap 方法), 1155
flush() (xml.etree.ElementTree.XMLParser 方法), 1271
flush() (xml.etree.ElementTree.XMLPullParser 方法), 1272
flush() (zlib.Compress 方法), 527
flush() (zlib.Decompress 方法), 528
flush_headers() (http.server.BaseHTTPRequestHandler 方法), 1395
flush_std_streams() (在 test.support 模块中), 1714
flushinp() (在 curses 模块中), 856
FlushKey() (在 winreg 模块中), 2024
fma() (decimal.Context 方法), 339
fma() (decimal.Decimal 方法), 331
fma() (在 math 模块中), 314
fmean() (在 statistics 模块中), 365
fmod() (在 math 模块中), 314
FMT_BINARY() (在 plistlib 模块中), 596
FMT_XML() (在 plistlib 模块中), 596
fnmatch
    module, 456
fnmatch() (在 fnmatch 模块中), 456
fnmatchcase() (在 fnmatch 模块中), 456
focus() (tkinter.ttk.Treeview 方法), 1531
fold() (email.headerregistry.BaseHeader 方法), 1181
fold() (email.policy.Compat32 方法), 1178
fold() (email.policy.EmailPolicy 方法), 1177
fold() (email.policy.Policy 方法), 1176
fold_binary() (email.policy.Compat32 方法), 1179
fold_binary() (email.policy.EmailPolicy 方法), 1177
fold_binary() (email.policy.Policy 方法), 1176
fold (datetime.datetime 属性), 205
fold (datetime.time 属性), 212
Font (tkinter.font 中的类), 1513
FOR_ITER (*opcode*), 2010
forget() (tkinter.ttk.Notebook 方法), 1526
forget() (在 test.support.import_helper 模块中), 1723
fork() (在 os 模块中), 663
fork() (在 pty 模块中), 2038
ForkingMixIn (socketserver 中的类), 1385
ForkingTCPServer (socketserver 中的类), 1385
ForkingUDPServer (socketserver 中的类), 1385
ForkingUnixDatagramServer (socketserver 中的类), 1385
ForkingUnixStreamServer (socketserver 中的类), 1385
forkpty() (在 os 模块中), 664
format()
    built-in function, 16
format() (inspect.Signature 方法), 1882
format() (logging.BufferingFormatter 方法), 710
format() (logging.Formatter 方法), 709
format() (logging.Handler 方法), 708
format() (pprint.PrettyPrinter 方法), 285
format() (str 方法), 50
format() (string.Formatter 方法), 112
format() (traceback.StackSummary 方法), 1866
format() (traceback.TracebackException 方法), 1865
format() (tracemalloc.Traceback 方法), 1769
format_datetime() (在 email.utils 模块中), 1210
format_exc() (在 traceback 模块中), 1863
format_exception() (在 traceback 模块中), 1863
format_exception_only() (traceback.TracebackException 方法), 1865
format_exception_only() (在 traceback 模块中), 1863
format_field() (string.Formatter 方法), 113
format_frame_summary() (traceback.StackSummary 方法), 1866
format_help() (argparse.ArgumentParser 方法), 810
format_list() (在 traceback 模块中), 1863
format_map() (str 方法), 51
FORMAT_SIMPLE (*opcode*), 2013
format_stack() (在 traceback 模块中), 1863
format_stack_entry() (bdb.Bdb 方法), 1735
format_string() (在 locale 模块中), 1452

format_tb() (在 traceback 模块中), 1863
format_usage() (argparse.Action 方法), 801
format_usage() (argparse.ArgumentParser 方法), 810
FORMAT_WITH_SPEC (*opcode*), 2013
formataddr() (在 email.utils 模块中), 1209
formatargvalues() (在 inspect 模块中), 1886
formatdate() (在 email.utils 模块中), 1210
formatday() (calendar.TextCalendar 方法), 232
FormatError, 1237
FormatError() (在 ctypes 模块中), 777
formatException() (logging.Formatter 方法), 709
formatFooter() (logging.BufferingFormatter 方法), 710
formatHeader() (logging.BufferingFormatter 方法), 710
formatmonth() (calendar.HTMLCalendar 方法), 233
formatmonth() (calendar.TextCalendar 方法), 232
formatmonthname() (calendar.HTMLCalendar 方法), 233
formatmonthname() (calendar.TextCalendar 方法), 232
formatStack() (logging.Formatter 方法), 709
FormattedValue (ast 中的类), 1946
Formatter (logging 中的类), 708
Formatter (string 中的类), 112
formatTime() (logging.Formatter 方法), 709
formatwarning() (在 warnings 模块中), 1830
formatweek() (calendar.TextCalendar 方法), 232
formatweekday() (calendar.TextCalendar 方法), 232
formatweekheader() (calendar.TextCalendar 方法), 232
formatyear() (calendar.HTMLCalendar 方法), 233
formatyear() (calendar.TextCalendar 方法), 232
formatyearpage() (calendar.HTMLCalendar 方法), 233
FORMAT (inspect.BufferFlags 属性), 1892
format (memoryview 属性), 80
format(multiprocessing.shared_memory.ShareableList 属性), 943
format (struct.Struct 属性), 173
Fortran 连续, 2066
forward() (在 turtle 模块中), 1464
ForwardRef (typing 中的类), 1591
For (ast 中的类), 1958
fpathconf() (在 os 模块中), 626
fp (urllib.error.HTTPError 属性), 1347
fractions
    module, 351
Fraction (fractions 中的类), 351
FrameInfo (inspect 中的类), 1888
FrameSummary (traceback 中的类), 1866
FrameType() (在 types 模块中), 279
frame (inspect.FrameInfo 属性), 1888
frame (tkinter.scrolledtext.ScrolledText 属性), 1519
Frame (tracemalloc 中的类), 1767
free threading -- 自由线程, **2068**
free variable -- 自由变量, **2068**
free_tool_id() (在 sys.monitoring 模块中), 1811
freedesktop_os_release() (在 platform 模块中), 745
freeze() (在 gc 模块中), 1874
freeze_support() (在 multiprocessing 模块中), 910
frexp() (在 math 模块中), 315
FRIDAY() (在 calendar 模块中), 235
from_address() (ctypes._CData 方法), 779
from_buffer() (ctypes._CData 方法), 778
from_buffer_copy() (ctypes._CData 方法), 779
from_bytes() (int 类方法), 39
from_callable() (inspect.Signature 类方法), 1883
from_decimal() (fractions.Fraction 类方法), 353
from_exception()(traceback.TracebackException 类方法), 1865
from_file() (zipfile.ZipInfo 类方法), 549
from_file() (zoneinfo.ZoneInfo 类方法), 228
from_float() (decimal.Decimal 类方法), 331
from_float() (fractions.Fraction 类方法), 352
from_iterable() (itertools.chain 类方法), 380
from_list()(traceback.StackSummary 类方法), 1866
from_param() (ctypes._CData 方法), 779
from_samples() (statistics.NormalDist 类方法), 373
from_traceback()(dis.Bytecode 类方法), 1997
from_uri() (pathlib.Path 类方法), 422
frombuf() (tarfile.TarInfo 类方法), 559
frombytes() (array.array 方法), 268
fromfd() (select.epoll 方法), 1137
fromfd() (select.kqueue 方法), 1138
fromfd() (在 socket 模块中), 1085
fromfile() (array.array 方法), 268
fromhex() (bytearray 类方法), 61
fromhex() (bytes 类方法), 60
fromhex() (float 类方法), 40
fromisocalendar() (datetime.date 类方法), 197
fromisocalendar() (datetime.datetime 类方法), 204
fromisoformat() (datetime.date 类方法), 197

fromisoformat() (datetime.datetime 类方法), 203
fromisoformat() (datetime.time 类方法), 213
fromkeys() (collections.Counter 方法), 242
fromkeys() (dict 类方法), 85
fromlist() (array.array 方法), 268
fromordinal() (datetime.date 类方法), 197
fromordinal() (datetime.datetime 类方法), 203
fromshare() (在 socket 模块中), 1086
fromstring() (在 xml.etree.ElementTree 模块中), 1263
fromstringlist() (在 xml.etree.ElementTree 模块中), 1263
fromtarfile() (tarfile.TarInfo 类方法), 559
fromtimestamp() (datetime.date 类方法), 197
fromtimestamp() (datetime.datetime 类方法), 202
fromunicode() (array.array 方法), 268
fromutc() (datetime.timezone 方法), 222
fromutc() (datetime.tzinfo 方法), 217
FrozenImporter(importlib.machinery 中的类), 1918
FrozenInstanceError, 1838
FrozenSet (typing 中的类), 1592
frozenset (内置类), 81
FS() (在 curses.ascii 模块中), 881
fs_is_case_insensitive() (在 test.support.os_helper 模块中), 1722
FS_NONASCII() (在 test.support.os_helper 模块中), 1721
fsdecode() (在 os 模块中), 618
fsencode() (在 os 模块中), 618
fspath() (在 os 模块中), 618
fstat() (在 os 模块中), 626
fstatvfs() (在 os 模块中), 626
fstring, 57
FSTRING_END() (在 token 模块中), 1980
FSTRING_MIDDLE() (在 token 模块中), 1980
FSTRING_START() (在 token 模块中), 1980
fsum() (在 math 模块中), 318
fsync() (在 os 模块中), 626
FTP, 1338
    ftplib (标准模块), 1359
    协议, 1337, 1359
ftp_open() (urllib.request.FTPHandler 方法), 1332
FTP_TLS (ftplib 中的类), 1363
FTPHandler (urllib.request 中的类), 1326
ftplib
    module, 1359
FTP (ftplib 中的类), 1359
ftruncate() (在 os 模块中), 626
Full, 972
full() (asyncio.Queue 方法), 1019
full() (multiprocessing.Queue 方法), 908
full() (queue.Queue 方法), 972
full_match() (pathlib.PurePath 方法), 419
FULL_RO (inspect.BufferFlags 属性), 1893
full_url (urllib.request.Request 属性), 1326
fullmatch() (re.Pattern 方法), 134
fullmatch() (在 re 模块中), 131
fully_trusted_filter() (在 tarfile 模块中), 562
FULL (inspect.BufferFlags 属性), 1893
funcname (bdb.Breakpoint 属性), 1732
function -- 函数, **2068**
function annotation -- 函数标注, **2068**
FunctionDef (ast 中的类), 1968
FunctionTestCase (unittest 中的类), 1641
FunctionType() (在 types 模块中), 278
FunctionType (ast 中的类), 1946
function (inspect.FrameInfo 属性), 1888
function (inspect.Traceback 属性), 1888
Function (pyclbr 中的类), 1990
Function (symtable 中的类), 1977
FUNCTION (symtable.SymbolTableType 属性), 1976
functools
    module, 393
func (functools.partial 属性), 402
funny_files (filecmp.dircmp 属性), 448
FutureWarning, 107
Future (asyncio 中的类), 1046
Future (concurrent.futures 中的类), 948
fwalk() (在 os 模块中), 654

G

-g
    trace 命令行选项, 1759
gaierror, 1079
gamma() (在 math 模块中), 320
gammavariate() (在 random 模块中), 357
garbage collection -- 垃圾回收, **2068**
garbage() (在 gc 模块中), 1874
gather() (curses.textpad.Textbox 方法), 879
gather() (在 asyncio 模块中), 991
gauss() (在 random 模块中), 358
gc
    module, 1872
gc_collect() (在 test.support 模块中), 1714
gcd() (在 math 模块中), 314
ge() (在 operator 模块中), 402
generate_tokens() (在 tokenize 模块中), 1985
generator -- 生成器, 2068
generator -- 生成器, **2068**
generator expression -- 生成器表达式, 2069
generator expression -- 生成器表达式, **2069**
generator iterator -- 生成器迭代器, **2068**
GeneratorExit, 101
GeneratorExp (ast 中的类), 1952
GeneratorType() (在 types 模块中), 278
Generator (collections.abc 中的类), 257

Generator (email.generator 中的类), 1171
Generator (typing 中的类), 1596
Generic
    Alias, 89
generic function -- 泛型函数, **2069**
generic type -- 泛型, **2069**
generic_visit() (ast.NodeVisitor 方法), 1973
GenericAlias
    object -- 对象, 89
GenericAlias (types 中的类), 279
Generic (typing 中的类), 1571
genops() (在 pickletools 模块中), 2018
geometric_mean() (在 statistics 模块中), 365
get() (asyncio.Queue 方法), 1020
get() (configparser.ConfigParser 方法), 590
get() (contextvars.Context 方法), 977
get() (contextvars.ContextVar 方法), 975
get() (dict 方法), 85
get() (email.message.EmailMessage 方法), 1162
get() (email.message.Message 方法), 1197
get() (mailbox.Mailbox 方法), 1223
get() (multiprocessing.pool.AsyncResult 方法), 926
get() (multiprocessing.Queue 方法), 908
get() (multiprocessing.SimpleQueue 方法), 909
get() (queue.Queue 方法), 972
get() (queue.SimpleQueue 方法), 974
get() (tkinter.ttk.Combobox 方法), 1524
get() (tkinter.ttk.Spinbox 方法), 1525
get() (types.MappingProxyType 方法), 280
get() (xml.etree.ElementTree.Element 方法), 1267
get() (在 webbrowser 模块中), 1310
GET_AITER (*opcode*), 2004
get_all() (email.message.EmailMessage 方法), 1162
get_all() (email.message.Message 方法), 1197
get_all() (wsgiref.headers.Headers 方法), 1314
get_all_breaks() (bdb.Bdb 方法), 1735
get_all_start_methods() (在 multiprocessing 模块中), 911
GET_ANEXT (*opcode*), 2004
get_annotations() (在 inspect 模块中), 1887
get_app() (wsgiref.simple_server.WSGIServer 方法), 1315
get_archive_formats() (在 shutil 模块中), 464
get_args() (在 typing 模块中), 1590
get_asyncgen_hooks() (在 sys 模块中), 1798
get_attribute() (在 test.support 模块中), 1716
GET_AWAITABLE (*opcode*), 2003
get_begidx() (在 readline 模块中), 163
get_blocking() (在 os 模块中), 627
get_body() (email.message.EmailMessage 方法), 1165
get_body_encoding() (email.charset.Charset 方法), 1206
get_boundary() (email.message.EmailMessage 方法), 1164
get_boundary() (email.message.Message 方法), 1199
get_bpbynumber() (bdb.Bdb 方法), 1734
get_break() (bdb.Bdb 方法), 1734
get_breaks() (bdb.Bdb 方法), 1735
get_buffer() (asyncio.BufferedProtocol 方法), 1054
get_bytes() (mailbox.Mailbox 方法), 1223
get_bytes() (mailbox.mbox 方法), 1227
get_bytes() (mailbox.MMDF 方法), 1230
get_ca_certs() (ssl.SSLContext 方法), 1119
get_cache_token() (在 abc 模块中), 1859
get_channel_binding() (ssl.SSLSocket 方法), 1115
get_charset() (email.message.Message 方法), 1196
get_charsets() (email.message.EmailMessage 方法), 1164
get_charsets() (email.message.Message 方法), 1200
get_child_watcher() (asyncio.AbstractEventLoopPolicy 方法), 1061
get_child_watcher() (在 asyncio 模块中), 1062
get_children() (symtable.SymbolTable 方法), 1977
get_children() (tkinter.ttk.Treeview 方法), 1530
get_ciphers() (ssl.SSLContext 方法), 1119
get_clock_info() (在 time 模块中), 692
get_close_matches() (在 difflib 模块中), 145
get_code() (importlib.abc.InspectLoader 方法), 1914
get_code() (importlib.abc.SourceLoader 方法), 1916
get_code() (importlib.machinery.ExtensionFileLoader 方法), 1921
get_code() (importlib.machinery.SourcelessFileLoader 方法), 1920
get_code() (zipimport.zipimporter 方法), 1902
get_completer() (在 readline 模块中), 163
get_completer_delims() (在 readline 模块中), 163
get_completion_type() (在 readline 模块中), 163
get_config_h_filename() (在 sysconfig 模块中), 1819

get_config_var()(在 sysconfig 模块中), 1815
get_config_vars() (在 sysconfig 模块中), 1815
get_content() (email.contentmanager.ContentManager 方法), 1186
get_content() (email.message.EmailMessage 方法), 1166
get_content() (在 email.contentmanager 模块中), 1186
get_content_charset() (email.message.EmailMessage 方法), 1164
get_content_charset() (email.message.Message 方法), 1200
get_content_disposition() (email.message.EmailMessage 方法), 1164
get_content_disposition() (email.message.Message 方法), 1200
get_content_maintype() (email.message.EmailMessage 方法), 1163
get_content_maintype() (email.message.Message 方法), 1198
get_content_subtype() (email.message.EmailMessage 方法), 1163
get_content_subtype() (email.message.Message 方法), 1198
get_content_type() (email.message.EmailMessage 方法), 1163
get_content_type() (email.message.Message 方法), 1198
get_context() (asyncio.Handle 方法), 1040
get_context() (asyncio.Task 方法), 1000
get_context()(在 multiprocessing 模块中), 911
get_coro() (asyncio.Task 方法), 1000
get_coroutine_origin_tracking_depth() (在 sys 模块中), 1799
get_count() (在 gc 模块中), 1873
get_current_history_length() (在 readline 模块中), 162
get_data() (importlib.abc.FileLoader 方法), 1915
get_data() (importlib.abc.ResourceLoader 方法), 1913
get_data() (zipimport.zipimporter 方法), 1902
get_data() (在 pkgutil 模块中), 1905
get_date() (mailbox.MaildirMessage 方法), 1231
get_debug() (asyncio.loop 方法), 1038
get_debug() (在 gc 模块中), 1872
get_default() (argparse.ArgumentParser 方法), 810
get_default_scheme() (在 sysconfig 模块中), 1818
get_default_type() (email.message.EmailMessage 方法), 1163
get_default_type() (email.message.Message 方法), 1198
get_default_verify_paths() (在 ssl 模块中), 1106
get_dialect() (在 csv 模块中), 570
get_disassembly_as_string() (test.support.bytecode_helper.BytecodeTestCase 方法), 1720
get_docstring() (在 ast 模块中), 1973
get_doctest() (doctest.DocTestParser 方法), 1617
get_endidx() (在 readline 模块中), 163
get_environ() (wsgiref.simple_server.WSGIRequestHandler 方法), 1315
get_errno() (在 ctypes 模块中), 777
get_escdelay() (在 curses 模块中), 859
get_event_loop() (asyncio.AbstractEventLoopPolicy 方法), 1061
get_event_loop() (在 asyncio 模块中), 1023
get_event_loop_policy()(在 asyncio 模块中), 1061
get_events() (在 sys.monitoring 模块中), 1813
get_examples() (doctest.DocTestParser 方法), 1617
get_exception_handler() (asyncio.loop 方法), 1037
get_exec_path() (在 os 模块中), 618
get_extra_info() (asyncio.BaseTransport 方法), 1049
get_extra_info() (asyncio.StreamWriter 方法), 1006
get_field() (string.Formatter 方法), 112
get_file() (mailbox.Babyl 方法), 1229
get_file() (mailbox.Mailbox 方法), 1223
get_file() (mailbox.Maildir 方法), 1226
get_file() (mailbox.mbox 方法), 1227
get_file() (mailbox.MH 方法), 1228
get_file() (mailbox.MMDF 方法), 1230
get_file_breaks() (bdb.Bdb 方法), 1735
get_filename() (email.message.EmailMessage 方法), 1164
get_filename() (email.message.Message 方法), 1199
get_filename() (importlib.abc.ExecutionLoader 方法), 1914
get_filename() (importlib.abc.FileLoader 方法), 1915
get_filename() (importlib.machinery.ExtensionFileLoader 方法), 1921

get_filename() (zipimport.zipimporter 方法), 1902
get_filter() (tkinter.filedialog.FileDialog 方法), 1516
get_flags() (mailbox.Maildir 方法), 1225
get_flags() (mailbox.MaildirMessage 方法), 1231
get_flags() (mailbox.mboxMessage 方法), 1233
get_flags() (mailbox.MMDFMessage 方法), 1236
get_folder() (mailbox.Maildir 方法), 1225
get_folder() (mailbox.MH 方法), 1228
get_frees() (symtable.Function 方法), 1978
get_freeze_count() (在 gc 模块中), 1874
get_from() (mailbox.mboxMessage 方法), 1232
get_from() (mailbox.MMDFMessage 方法), 1236
get_full_url() (urllib.request.Request 方法), 1327
get_globals() (symtable.Function 方法), 1978
get_grouped_opcodes() (difflib.SequenceMatcher 方法), 148
get_handle_inheritable() (在 os 模块中), 636
get_header() (urllib.request.Request 方法), 1327
get_history_item() (在 readline 模块中), 162
get_history_length() (在 readline 模块中), 162
get_id() (symtable.SymbolTable 方法), 1977
get_ident() (在 _thread 模块中), 979
get_ident() (在 threading 模块中), 886
get_identifiers() (string.Template 方法), 120
get_identifiers() (symtable.SymbolTable 方法), 1977
get_importer() (在 pkgutil 模块中), 1904
get_info() (mailbox.Maildir 方法), 1225
get_info() (mailbox.MaildirMessage 方法), 1231
get_inheritable() (socket.socket 方法), 1092
get_inheritable() (在 os 模块中), 636
get_instructions() (在 dis 模块中), 1999
get_int_max_str_digits() (在 sys 模块中), 1797
get_interpreter() (在 zipapp 模块中), 1784
GET_ITER (*opcode*), 2002
get_key() (selectors.BaseSelector 方法), 1142
get_labels() (mailbox.Babyl 方法), 1229
get_labels() (mailbox.BabylMessage 方法), 1235
get_last_error() (在 ctypes 模块中), 777
GET_LEN (*opcode*), 2006
get_line_buffer() (在 readline 模块中), 161
get_lineno() (symtable.SymbolTable 方法), 1977
get_loader() (在 pkgutil 模块中), 1904
get_local_events() (在 sys.monitoring 模块中), 1813
get_locals() (symtable.Function 方法), 1978
get_logger() (在 multiprocessing 模块中), 930
get_loop() (asyncio.Future 方法), 1047
get_loop() (asyncio.Runner 方法), 983
get_loop() (asyncio.Server 方法), 1041
get_makefile_filename() (在 sysconfig 模块中), 1819
get_map() (selectors.BaseSelector 方法), 1142
get_matching_blocks() (difflib.SequenceMatcher 方法), 148
get_message() (mailbox.Mailbox 方法), 1223
get_method() (urllib.request.Request 方法), 1327
get_methods() (symtable.Class 方法), 1978
get_mixed_type_key() (在 ipaddress 模块中), 1434
get_name() (asyncio.Task 方法), 1000
get_name() (symtable.Symbol 方法), 1978
get_name() (symtable.SymbolTable 方法), 1977
get_namespace() (symtable.Symbol 方法), 1979
get_namespaces() (symtable.Symbol 方法), 1979
get_native_id() (在 _thread 模块中), 979
get_native_id() (在 threading 模块中), 886
get_nonlocals() (symtable.Function 方法), 1978
get_nonstandard_attr() (http.cookiejar.Cookie 方法), 1409
get_nowait() (asyncio.Queue 方法), 1020
get_nowait() (multiprocessing.Queue 方法), 909
get_nowait() (queue.Queue 方法), 972
get_nowait() (queue.SimpleQueue 方法), 974
get_object_traceback() (在 tracemalloc 模块中), 1765
get_objects() (在 gc 模块中), 1872
get_opcodes() (difflib.SequenceMatcher 方法), 148
get_option() (optparse.OptionParser 方法), 843
get_option_group() (optparse.OptionParser 方法), 834
get_origin() (在 typing 模块中), 1590
get_original_bases() (在 types 模块中), 277
get_original_stdout() (在 test.support 模

块中), 1713
get_osfhandle() (在 msvcrt 模块中), 2020
get_output_charset() (email.charset.Charset 方法), 1207
get_overloads() (在 typing 模块中), 1587
get_pagesize() (在 test.support 模块中), 1713
get_param() (email.message.Message 方法), 1198
get_parameters() (symtable.Function 方法), 1977
get_params() (email.message.Message 方法), 1198
get_path() (在 sysconfig 模块中), 1818
get_path_names() (在 sysconfig 模块中), 1818
get_paths() (在 sysconfig 模块中), 1819
get_payload() (email.message.Message 方法), 1195
get_pid() (asyncio.SubprocessTransport 方法), 1052
get_pipe_transport() (asyncio.SubprocessTransport 方法), 1052
get_platform() (在 sysconfig 模块中), 1819
get_poly() (在 turtle 模块中), 1478
get_preferred_scheme() (在 sysconfig 模块中), 1818
get_protocol() (asyncio.BaseTransport 方法), 1050
get_protocol_members() (在 typing 模块中), 1590
get_proxy_response_headers() (http.client.HTTPConnection 方法), 1355
get_python_version() (在 sysconfig 模块中), 1819
get_ready() (graphlib.TopologicalSorter 方法), 307
get_referents() (在 gc 模块中), 1873
get_referrers() (在 gc 模块中), 1873
get_request() (socketserver.BaseServer 方法), 1387
get_returncode() (asyncio.SubprocessTransport 方法), 1052
get_running_loop() (在 asyncio 模块中), 1023
get_scheme() (wsgiref.handlers.BaseHandler 方法), 1318
get_scheme_names() (在 sysconfig 模块中), 1818
get_selection() (tkinter.filedialog.FileDialog 方法), 1516
get_sequences() (mailbox.MH 方法), 1228
get_sequences() (mailbox.MHMessage 方法), 1234
get_server() (multiprocessing.managers.BaseManager 方法), 918
get_server_certificate() (在 ssl 模块中), 1105
get_shapepoly() (在 turtle 模块中), 1477
get_source() (importlib.abc.InspectLoader 方法), 1914
get_source() (importlib.abc.SourceLoader 方法), 1916
get_source() (importlib.machinery.ExtensionFileLoader 方法), 1921
get_source() (importlib.machinery.SourcelessFileLoader 方法), 1920
get_source() (zipimport.zipimporter 方法), 1902
get_source_segment() (在 ast 模块中), 1973
get_stack() (asyncio.Task 方法), 1000
get_stack() (bdb.Bdb 方法), 1735
get_start_method() (在 multiprocessing 模块中), 911
get_starttag_text() (html.parser.HTMLParser 方法), 1251
get_stats() (在 gc 模块中), 1872
get_stats_profile() (pstats.Stats 方法), 1752
get_stderr() (wsgiref.handlers.BaseHandler 方法), 1318
get_stderr() (wsgiref.simple_server.WSGIRequestHandler 方法), 1315
get_stdin() (wsgiref.handlers.BaseHandler 方法), 1317
get_string() (mailbox.Mailbox 方法), 1223
get_string() (mailbox.mbox 方法), 1227
get_subdir() (mailbox.MaildirMessage 方法), 1231
get_symbols() (symtable.SymbolTable 方法), 1977
get_tabsize() (在 curses 模块中), 859
get_task_factory() (asyncio.loop 方法), 1027
get_terminal_size() (在 os 模块中), 635
get_terminal_size() (在 shutil 模块中), 467
get_threshold() (在 gc 模块中), 1873
get_token() (shlex.shlex 方法), 1497
get_tool() (在 sys.monitoring 模块中), 1811
get_traceback_limit() (在 tracemalloc 模块中), 1765
get_traced_memory() (在 tracemalloc 模块中), 1765
get_tracemalloc_memory() (在 tracemalloc 模块中), 1765
get_type() (symtable.SymbolTable 方法), 1977
get_type_hints() (在 typing 模块中), 1589
get_unixfrom() (email.message.EmailMessage 方法), 1161
get_unixfrom() (email.message.Message 方法), 1195
get_unpack_formats() (在 shutil 模块中), 465
get_unverified_chain() (ssl.SSLSocket 方

法），1115
get_usage() （optparse.OptionParser 方法），844
get_value() （string.Formatter 方法），112
get_verified_chain() （ssl.SSLSocket 方法），1115
get_version() （optparse.OptionParser 方法），834
get_visible() （mailbox.BabylMessage 方法），1235
get_wch() （curses.window 方法），863
get_write_buffer_limits() （asyncio.WriteTransport 方法），1051
get_write_buffer_size() （asyncio.WriteTransport 方法），1051
GET_YIELD_FROM_ITER (*opcode*), 2002
getacl() （imaplib.IMAP4 方法），1370
getaddresses()（在 email.utils 模块中），1209
getaddrinfo() （asyncio.loop 方法），1035
getaddrinfo() （在 socket 模块中），1086
getallocatedblocks() （在 sys 模块中），1796
getandroidapilevel() （在 sys 模块中），1796
getannotation() （imaplib.IMAP4 方法），1370
getargvalues() （在 inspect 模块中），1886
getasyncgenlocals() （在 inspect 模块中），1891
getasyncgenstate()（在 inspect 模块中），1891
getatime() （在 os.path 模块中），436
getattr()
    built-in function, 17
getattr_static() （在 inspect 模块中），1890
getAttribute() （xml.dom.Element 方法），1279
getAttributeNode() （xml.dom.Element 方法），1279
getAttributeNodeNS() （xml.dom.Element 方法），1279
getAttributeNS() （xml.dom.Element 方法），1279
GetBase() （xml.parsers.expat.xmlparser 方法），1301
getbegyx() （curses.window 方法），863
getbkgd() （curses.window 方法），863
getblocking() （socket.socket 方法），1092
getboolean() （configparser.ConfigParser 方法），590
getbuffer() （io.BytesIO 方法），684
getByteStream() （xml.sax.xmlreader.InputSource 方法），1298
getcallargs() （在 inspect 模块中），1886
getcanvas() （在 turtle 模块中），1484
getch() （curses.window 方法），863
getch() （在 msvcrt 模块中），2020
getCharacterStream() （xml.sax.xmlreader.InputSource 方法），1299
getche() （在 msvcrt 模块中），2020
getChild() （logging.Logger 方法），704
getChildren() （logging.Logger 方法），704
getclasstree() （在 inspect 模块中），1885
getclosurevars() （在 inspect 模块中），1887
getcode() （http.client.HTTPResponse 方法），1357
getcode() （urllib.response.addinfourl 方法），1338
getColumnNumber() （xml.sax.xmlreader.Locator 方法），1298
getcomments() （在 inspect 模块中），1880
getcompname() （wave.Wave_read 方法），1438
getcomptype() （wave.Wave_read 方法），1438
getconfig() （sqlite3.Connection 方法），508
getContentHandler() （xml.sax.xmlreader.XMLReader 方法），1297
getcontext() （在 decimal 模块中），335
getcoroutinelocals() （在 inspect 模块中），1891
getcoroutinestate() （在 inspect 模块中），1891
getctime() （在 os.path 模块中），436
getcwd() （在 os 模块中），639
getcwdb() （在 os 模块中），639
getdecoder() （在 codecs 模块中），174
getdefaultencoding() （在 sys 模块中），1796
getdefaultlocale() （在 locale 模块中），1451
getdefaulttimeout()（在 socket 模块中），1089
getdlopenflags() （在 sys 模块中），1796
getdoc() （在 inspect 模块中），1880
getDOMImplementation() （在 xml.dom 模块中），1274
getDTDHandler() （xml.sax.xmlreader.XMLReader 方法），1297
getEffectiveLevel() （logging.Logger 方法），704
getegid() （在 os 模块中），618
getElementsByTagName() （xml.dom.Document 方法），1278
getElementsByTagName() （xml.dom.Element 方法），1278
getElementsByTagNameNS() （xml.dom.Document 方法），1278
getElementsByTagNameNS() （xml.dom.Element 方法），1279
getencoder() （在 codecs 模块中），174
getEncoding() （xml.sax.xmlreader.InputSource 方法），1298
getencoding() （在 locale 模块中），1452
getEntityResolver() （xml.sax.xmlreader.XMLReader 方法），1297
getenv() （在 os 模块中），618
getenvb() （在 os 模块中），618
getErrorHandler()

(xml.sax.xmlreader.XMLReader 方法), 1297
geteuid() (在 os 模块中), 619
getEvent() (xml.dom.pulldom.DOMEventStream 方法), 1288
getEventCategory() (logging.handlers.NTEventLogHandler 方法), 737
getEventType() (logging.handlers.NTEventLogHandler 方法), 737
getException() (xml.sax.SAXException 方法), 1290
getFeature() (xml.sax.xmlreader.XMLReader 方法), 1297
getfile() (在 inspect 模块中), 1880
getFilesToDelete() (logging.handlers.TimedRotatingFileHandler 方法), 734
getfilesystemencodeerrors() (在 sys 模块中), 1796
getfilesystemencoding() (在 sys 模块中), 1796
getfirstweekday() (calendar.Calendar 方法), 231
getfloat() (configparser.ConfigParser 方法), 590
getfqdn() (在 socket 模块中), 1087
getframeinfo() (在 inspect 模块中), 1889
getframerate() (wave.Wave_read 方法), 1438
getfullargspec() (在 inspect 模块中), 1885
getgeneratorlocals() (在 inspect 模块中), 1891
getgeneratorstate() (在 inspect 模块中), 1890
getgid() (在 os 模块中), 619
getgrall() (在 grp 模块中), 2035
getgrgid() (在 grp 模块中), 2035
getgrnam() (在 grp 模块中), 2035
getgrouplist() (在 os 模块中), 619
getgroups() (在 os 模块中), 619
getHandlerByName() (在 logging 模块中), 715
getHandlerNames() (在 logging 模块中), 715
getheader() (http.client.HTTPResponse 方法), 1356
getheaders() (http.client.HTTPResponse 方法), 1357
gethostbyaddr() (在 *socket* 模块中), 623
gethostbyaddr() (在 socket 模块中), 1087
gethostbyname() (在 socket 模块中), 1087
gethostbyname_ex() (在 socket 模块中), 1087
gethostname() (在 *socket* 模块中), 623
gethostname() (在 socket 模块中), 1087
getincrementaldecoder() (在 codecs 模块中), 174
getincrementalencoder() (在 codecs 模块中), 174
getinfo() (zipfile.ZipFile 方法), 544
getinnerframes() (在 inspect 模块中), 1889
GetInputContext() (xml.parsers.expat.xmlparser 方法), 1301
getint() (configparser.ConfigParser 方法), 590
getitem() (在 operator 模块中), 405
getitimer() (在 signal 模块中), 1149
getkey() (curses.window 方法), 863
GetLastError() (在 ctypes 模块中), 777
getLength() (xml.sax.xmlreader.Attributes 方法), 1299
getLevelName() (在 logging 模块中), 714
getLevelNamesMapping() (在 logging 模块中), 714
getlimit() (sqlite3.Connection 方法), 507
getline() (在 linecache 模块中), 457
getLineNumber() (xml.sax.xmlreader.Locator 方法), 1298
getloadavg() (在 os 模块中), 675
getlocale() (在 locale 模块中), 1452
getLogger() (在 logging 模块中), 713
getLoggerClass() (在 logging 模块中), 713
getlogin() (在 os 模块中), 619
getLogRecordFactory() (在 logging 模块中), 713
getMandatoryRelease() (__future__._Feature 方法), 1871
getmark() (wave.Wave_read 方法), 1438
getmarkers() (wave.Wave_read 方法), 1438
getmaxyx() (curses.window 方法), 863
getmember() (tarfile.TarFile 方法), 556
getmembers() (tarfile.TarFile 方法), 556
getmembers() (在 inspect 模块中), 1877
getmembers_static() (在 inspect 模块中), 1877
getMessage() (logging.LogRecord 方法), 711
getMessage() (xml.sax.SAXException 方法), 1290
getMessageID() (logging.handlers.NTEventLogHandler 方法), 738
getmodule() (在 inspect 模块中), 1880
getmodulename() (在 inspect 模块中), 1877
getmouse() (在 curses 模块中), 856
getmro() (在 inspect 模块中), 1886
getmtime() (在 os.path 模块中), 436
getName() (threading.Thread 方法), 889
getNameByQName() (xml.sax.xmlreader.AttributesNS 方法), 1299
getnameinfo() (asyncio.loop 方法), 1035
getnameinfo() (在 socket 模块中), 1087
getnames() (tarfile.TarFile 方法), 557
getNames() (xml.sax.xmlreader.Attributes 方法), 1299
getnchannels() (wave.Wave_read 方法), 1438
getnframes() (wave.Wave_read 方法), 1438
getnode() (在 uuid 模块中), 1382
getobjects() (在 sys 模块中), 1797
getopt

    module, 2051
getopt() (在 getopt 模块中), 2051
GetoptError, 2052
getOptionalRelease() (__future__._Feature 方法), 1871
getouterframes() (在 inspect 模块中), 1889
getoutput() (在 subprocess 模块中), 968
getpagesize() (在 resource 模块中), 2045
getparams() (wave.Wave_read 方法), 1438
getparyx() (curses.window 方法), 863
getpass
    module, 851
getpass() (在 getpass 模块中), 851
GetPassWarning, 851
getpeercert() (ssl.SSLSocket 方法), 1114
getpeername() (socket.socket 方法), 1092
getpen() (在 turtle 模块中), 1478
getpgid() (在 os 模块中), 619
getpgrp() (在 os 模块中), 619
getpid() (在 os 模块中), 619
getpos() (html.parser.HTMLParser 方法), 1251
getppid() (在 os 模块中), 619
getpreferredencoding() (在 locale 模块中), 1452
getpriority() (在 os 模块中), 620
getprofile() (在 sys 模块中), 1798
getprofile() (在 threading 模块中), 887
getProperty() (xml.sax.xmlreader.XMLReader 方法), 1297
getprotobyname() (在 socket 模块中), 1088
getproxies() (在 urllib.request 模块中), 1323
getPublicId() (xml.sax.xmlreader.InputSource 方法), 1298
getPublicId() (xml.sax.xmlreader.Locator 方法), 1298
getpwall() (在 pwd 模块中), 2034
getpwnam() (在 pwd 模块中), 2034
getpwuid() (在 pwd 模块中), 2034
getQNameByName() (xml.sax.xmlreader.AttributesNS 方法), 1299
getQNames() (xml.sax.xmlreader.AttributesNS 方法), 1299
getquota() (imaplib.IMAP4 方法), 1370
getquotaroot() (imaplib.IMAP4 方法), 1370
getrandbits() (random.Random 方法), 358
getrandbits() (在 random 模块中), 356
getrandom() (在 os 模块中), 676
getreader() (在 codecs 模块中), 175
getrecursionlimit() (在 sys 模块中), 1797
getrefcount() (在 sys 模块中), 1797
GetReparseDeferralEnabled() (xml.parsers.expat.xmlparser 方法), 1302
getresgid() (在 os 模块中), 620
getresponse() (http.client.HTTPConnection 方法), 1355
getresuid() (在 os 模块中), 620
getrlimit() (在 resource 模块中), 2042
getroot() (xml.etree.ElementTree.ElementTree 方法), 1269
getrusage() (在 resource 模块中), 2045
getsampwidth() (wave.Wave_read 方法), 1438
getscreen() (在 turtle 模块中), 1478
getservbyname() (在 socket 模块中), 1088
getservbyport() (在 socket 模块中), 1088
GetSetDescriptorType() (在 types 模块中), 279
getshapes() (在 turtle 模块中), 1484
getsid() (在 os 模块中), 622
getsignal() (在 signal 模块中), 1147
getsitepackages() (在 site 模块中), 1895
getsize() (在 os.path 模块中), 436
getsizeof() (在 sys 模块中), 1797
getsockname() (socket.socket 方法), 1092
getsockopt() (socket.socket 方法), 1092
getsource() (在 inspect 模块中), 1881
getsourcefile() (在 inspect 模块中), 1880
getsourcelines() (在 inspect 模块中), 1880
getstate() (codecs.IncrementalDecoder 方法), 180
getstate() (codecs.IncrementalEncoder 方法), 179
getstate() (random.Random 方法), 358
getstate() (在 random 模块中), 355
getstatusoutput() (在 subprocess 模块中), 968
getstr() (curses.window 方法), 863
getSubject() (logging.handlers.SMTPHandler 方法), 738
getswitchinterval() (在 sys 模块中), 1797
getSystemId() (xml.sax.xmlreader.InputSource 方法), 1298
getSystemId() (xml.sax.xmlreader.Locator 方法), 1298
getsyx() (在 curses 模块中), 856
gettarinfo() (tarfile.TarFile 方法), 558
gettempdir() (在 tempfile 模块中), 451
gettempdirb() (在 tempfile 模块中), 451
gettempprefix() (在 tempfile 模块中), 452
gettempprefixb() (在 tempfile 模块中), 452
getTestCaseNames() (unittest.TestLoader 方法), 1643
gettext
    module, 1441
gettext() (gettext.GNUTranslations 方法), 1444
gettext() (gettext.NullTranslations 方法), 1443
gettext() (在 gettext 模块中), 1441
gettext() (在 locale 模块中), 1454
gettimeout() (socket.socket 方法), 1092
gettrace() (在 sys 模块中), 1798
gettrace() (在 threading 模块中), 887
getturtle() (在 turtle 模块中), 1478

getType() (xml.sax.xmlreader.Attributes 方法), 1299
getuid() (在 os 模块中), 620
getunicodeinternedsize() (在 sys 模块中), 1796
geturl() (http.client.HTTPResponse 方法), 1357
geturl() (urllib.parse.urllib.parse.SplitResult 方法), 1344
geturl() (urllib.response.addinfourl 方法), 1338
getuser() (在 getpass 模块中), 851
getuserbase() (在 site 模块中), 1895
getusersitepackages()(在 site 模块中), 1895
getvalue() (io.BytesIO 方法), 685
getvalue() (io.StringIO 方法), 689
getValue() (xml.sax.xmlreader.Attributes 方法), 1299
getValueByQName() (xml.sax.xmlreader.AttributesNS 方法), 1299
getwch() (在 msvcrt 模块中), 2020
getwche() (在 msvcrt 模块中), 2020
getweakrefcount() (在 weakref 模块中), 270
getweakrefs() (在 weakref 模块中), 270
getwelcome() (ftplib.FTP 方法), 1360
getwelcome() (poplib.POP3 方法), 1366
getwin() (在 curses 模块中), 856
getwindowsversion() (在 sys 模块中), 1798
getwriter() (在 codecs 模块中), 175
getxattr() (在 os 模块中), 660
getyx() (curses.window 方法), 863
gid (tarfile.TarInfo 属性), 560
GIL, **2069**
glob
    module, 453, 456
glob() (pathlib.Path 方法), 427
glob() (在 glob 模块中), 454
global interpreter lock -- 全局解释器锁, **2069**
global_enum() (在 enum 模块中), 305
globals()
    built-in function, 17
Global (ast 中的类), 1969
globs (doctest.DocTest 属性), 1615
gmtime() (在 time 模块中), 692
gname (tarfile.TarInfo 属性), 560
GNOME, 1445
GNU_FORMAT() (在 tarfile 模块中), 555
gnu_getopt() (在 getopt 模块中), 2052
GNUTranslations (gettext 中的类), 1444
GNUTYPE_LONGLINK() (在 tarfile 模块中), 555
GNUTYPE_LONGNAME() (在 tarfile 模块中), 555
GNUTYPE_SPARSE() (在 tarfile 模块中), 555
go() (tkinter.filedialog.FileDialog 方法), 1516
goto() (在 turtle 模块中), 1464
got (doctest.DocTestFailure 属性), 1621
grantpt() (在 os 模块中), 627
graphlib
    module, 305
GREATER() (在 token 模块中), 1982
GREATEREQUAL() (在 token 模块中), 1982
Greenwich Mean Time, 690
GRND_NONBLOCK() (在 os 模块中), 677
GRND_RANDOM() (在 os 模块中), 677
group() (pathlib.Path 方法), 431
group() (re.Match 方法), 135
groupby() (在 itertools 模块中), 382
groupdict() (re.Match 方法), 137
groupindex (re.Pattern 属性), 135
groups() (re.Match 方法), 136
groups(email.headerregistry.AddressHeader 属性), 1182
groups (re.Pattern 属性), 135
Group (email.headerregistry 中的类), 1185
grp
    module, 2035
GS() (在 curses.ascii 模块中), 881
gt() (在 operator 模块中), 402
GtE (ast 中的类), 1950
Gt (ast 中的类), 1950
guess_all_extensions() (mimetypes.MimeTypes 方法), 1241
guess_all_extensions() (在 mimetypes 模块中), 1239
guess_extension() (mimetypes.MimeTypes 方法), 1241
guess_extension() (在 mimetypes 模块中), 1239
guess_file_type() (mimetypes.MimeTypes 方法), 1241
guess_file_type() (在 mimetypes 模块中), 1239
guess_scheme() (在 wsgiref.util 模块中), 1312
guess_type() (mimetypes.MimeTypes 方法), 1241
guess_type() (在 mimetypes 模块中), 1239
GUI, 1501
gzip
    module, 528
GzipFile (gzip 中的类), 529
gzip 命令行选项
    --best, 531
    -d, 531
    --decompress, 531
    --fast, 531
    file, 531
    -h, 531
    --help, 531

H

-h
    ast 命令行选项, 1975
    calendar 命令行选项, 237

dis 命令行选项, 1997
gzip 命令行选项, 531
idle 命令行选项, 1544
json.tool 命令行选项, 1221
python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令行选项, 516
random 命令行选项, 362
timeit 命令行选项, 1756
tokenize 命令行选项, 1986
uuid 命令行选项, 1383
zipapp 命令行选项, 1783
halfdelay() (在 curses 模块中), 856
handle() (http.server.BaseHTTPRequestHandler 方法), 1394
handle() (logging.Handler 方法), 708
handle() (logging.handlers.QueueListener 方法), 741
handle() (logging.Logger 方法), 706
handle() (logging.NullHandler 方法), 730
handle() (socketserver.BaseRequestHandler 方法), 1388
handle() (wsgiref.simple_server.WSGIRequestHandler 方法), 1316
handle_charref() (html.parser.HTMLParser 方法), 1251
handle_comment() (html.parser.HTMLParser 方法), 1251
handle_data() (html.parser.HTMLParser 方法), 1251
handle_decl() (html.parser.HTMLParser 方法), 1251
handle_defect() (email.policy.Policy 方法), 1175
handle_endtag() (html.parser.HTMLParser 方法), 1251
handle_entityref() (html.parser.HTMLParser 方法), 1251
handle_error() (socketserver.BaseServer 方法), 1387
handle_expect_100() (http.server.BaseHTTPRequestHandler 方法), 1394
handle_one_request() (http.server.BaseHTTPRequestHandler 方法), 1394
handle_pi() (html.parser.HTMLParser 方法), 1251
handle_request() (socketserver.BaseServer 方法), 1386
handle_request() (xmlrpc.server.CGIXMLRPCRequestHandler 方法), 1421
handle_startendtag() (html.parser.HTMLParser 方法), 1251
handle_starttag() (html.parser.HTMLParser 方法), 1251
handle_timeout() (socketserver.BaseServer 方法), 1387
handleError() (logging.Handler 方法), 708
handleError() (logging.handlers.SocketHandler 方法), 734
handlers (logging.Logger 属性), 703
Handlers (signal 中的类), 1144
Handler (logging 中的类), 707
Handle (asyncio 中的类), 1040
hardlink_to() (pathlib.Path 方法), 430
--hardlink-dupes
compileall 命令行选项, 1993
harmonic_mean() (在 statistics 模块中), 366
HAS_ALPN() (在 ssl 模块中), 1111
has_children() (symtable.SymbolTable 方法), 1977
has_colors() (在 curses 模块中), 856
has_default() (typing.ParamSpec 方法), 1576
has_default() (typing.TypeVar 方法), 1573
has_default() (typing.TypeVarTuple 方法), 1575
has_dualstack_ipv6() (在 socket 模块中), 1085
HAS_ECDH() (在 ssl 模块中), 1111
has_extended_color_support() (在 curses 模块中), 856
has_extn() (smtplib.SMTP 方法), 1376
has_header() (csv.Sniffer 方法), 572
has_header() (urllib.request.Request 方法), 1327
has_ic() (在 curses 模块中), 856
has_il() (在 curses 模块中), 856
has_ipv6() (在 socket 模块中), 1083
has_key() (在 curses 模块中), 856
has_location(importlib.machinery.ModuleSpec 属性), 1922
HAS_NEVER_CHECK_COMMON_NAME() (在 ssl 模块中), 1111
has_nonstandard_attr() (http.cookiejar.Cookie 方法), 1408
HAS_NPN() (在 ssl 模块中), 1111
has_option() (configparser.ConfigParser 方法), 589
has_option() (optparse.OptionParser 方法), 843
HAS_PSK() (在 ssl 模块中), 1111
has_section() (configparser.ConfigParser 方法), 589
HAS_SNI() (在 ssl 模块中), 1111
HAS_SSLv2() (在 ssl 模块中), 1111
HAS_SSLv3() (在 ssl 模块中), 1111
has_ticket (ssl.SSLSession 属性), 1132
HAS_TLSv1() (在 ssl 模块中), 1111
HAS_TLSv1_1() (在 ssl 模块中), 1111
HAS_TLSv1_2() (在 ssl 模块中), 1111
HAS_TLSv1_3() (在 ssl 模块中), 1111
hasarg() (在 dis 模块中), 2016

hasattr()
    built-in function, 17
hasAttribute() (xml.dom.Element 方法), 1279
hasAttributeNS() (xml.dom.Element 方法), 1279
hasAttributes() (xml.dom.Node 方法), 1276
hasChildNodes() (xml.dom.Node 方法), 1276
hascompare() (在 dis 模块中), 2016
hasconst() (在 dis 模块中), 2016
hasexc() (在 dis 模块中), 2016
hasFeature() (xml.dom.DOMImplementation 方法), 1275
hasfree() (在 dis 模块中), 2016
hash
    内置函数, 45
hash()
    built-in function, 17
hash-based pyc -- 基于哈希的 pyc, **2069**
hash_bits (sys.hash_info 属性), 1799
hash_info() (在 sys 模块中), 1799
hash_randomization (sys.flags 属性), 1793
hashable -- 可哈希, **2069**
Hashable (collections.abc 中的类), 257
Hashable (typing 中的类), 1596
hasHandlers() (logging.Logger 方法), 706
hash.block_size() (在 hashlib 模块中), 601
hash.digest_size() (在 hashlib 模块中), 601
hashlib
    module, 599
hasjabs() (在 dis 模块中), 2017
hasjrel() (在 dis 模块中), 2016
hasjump() (在 dis 模块中), 2016
haslocal() (在 dis 模块中), 2016
hasname() (在 dis 模块中), 2016
HAVE_ARGUMENT (*opcode*), 2014
HAVE_CONTEXTVAR() (在 decimal 模块中), 342
HAVE_DOCSTRINGS() (在 test.support 模块中), 1711
HAVE_THREADS() (在 decimal 模块中), 341
HCI_DATA_DIR() (在 socket 模块中), 1083
HCI_FILTER() (在 socket 模块中), 1083
HCI_TIME_STAMP() (在 socket 模块中), 1083
header_encode() (email.charset.Charset 方法), 1207
header_encode_lines() (email.charset.Charset 方法), 1207
header_encoding (email.charset.Charset 属性), 1206
header_factory (email.policy.EmailPolicy 属性), 1177
header_fetch_parse() (email.policy.Compat32 方法), 1178
header_fetch_parse() (email.policy.EmailPolicy 方法), 1177
header_fetch_parse() (email.policy.Policy 方法), 1176
header_items() (urllib.request.Request 方法), 1327
header_max_count() (email.policy.EmailPolicy 方法), 1177
header_max_count() (email.policy.Policy 方法), 1175
header_offset (zipfile.ZipInfo 属性), 550
header_source_parse() (email.policy.Compat32 方法), 1178
header_source_parse() (email.policy.EmailPolicy 方法), 1177
header_source_parse() (email.policy.Policy 方法), 1175
header_store_parse() (email.policy.Compat32 方法), 1178
header_store_parse() (email.policy.EmailPolicy 方法), 1177
header_store_parse() (email.policy.Policy 方法), 1176
HeaderDefect, 1179
HeaderError, 554
HeaderParseError, 1179
HeaderParser (email.parser 中的类), 1169
HeaderRegistry (email.headerregistry 中的类), 1183
headers (http.client.HTTPResponse 属性), 1357
headers(http.server.BaseHTTPRequestHandler 属性), 1393
headers (urllib.error.HTTPError 属性), 1347
headers (urllib.response.addinfourl 属性), 1338
Headers (wsgiref.headers 中的类), 1314
headers (xmlrpc.client.ProtocolError 属性), 1414
HeaderWriteError, 1179
Header (email.header 中的类), 1204
heading() (tkinter.ttk.Treeview 方法), 1531
heading() (在 turtle 模块中), 1469
heapify() (在 heapq 模块中), 260
heapmin() (在 msvcrt 模块中), 2020
heappop() (在 heapq 模块中), 260
heappush() (在 heapq 模块中), 260
heappushpop() (在 heapq 模块中), 260
heapq
    module, 260
heapreplace() (在 heapq 模块中), 260
helo() (smtplib.SMTP 方法), 1376
help
    在线, 1597

--help
    ast 命令行选项, 1975
    calendar 命令行选项, 237
    dis 命令行选项, 1997
    gzip 命令行选项, 531
    json.tool 命令行选项, 1221
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令行选项, 516
    random 命令行选项, 362
    timeit 命令行选项, 1756
    tokenize 命令行选项, 1986
    trace 命令行选项, 1758
    uuid 命令行选项, 1383
    zipapp 命令行选项, 1783
help (*pdb command*), 1741
help()
    built-in function, 17
help (optparse.Option 属性), 839
herror, 1079
hex()
    built-in function, 17
hex() (bytearray 方法), 62
hex() (bytes 方法), 61
hex() (float 方法), 40
hex() (memoryview 方法), 77
hexdigest() (hashlib.hash 方法), 601
hexdigest() (hashlib.shake 方法), 602
hexdigest() (hmac.HMAC 方法), 610
hexdigits() (在 string 模块中), 111
hexlify() (在 binascii 模块中), 1246
hexversion() (在 sys 模块中), 1799
hex (uuid.UUID 属性), 1381
hidden() (curses.panel.Panel 方法), 883
hide() (curses.panel.Panel 方法), 883
hide() (tkinter.ttk.Notebook 方法), 1526
hide_cookie2 (http.cookiejar.CookiePolicy 属性), 1406
hideturtle() (在 turtle 模块中), 1474
HierarchyRequestErr, 1281
HIGH_PRIORITY_CLASS() (在 subprocess 模块中), 963
HIGHEST_PROTOCOL() (在 pickle 模块中), 471
hits (bdb.Breakpoint 属性), 1732
HKEY_CLASSES_ROOT() (在 winreg 模块中), 2027
HKEY_CURRENT_CONFIG() (在 winreg 模块中), 2027
HKEY_CURRENT_USER() (在 winreg 模块中), 2027
HKEY_DYN_DATA() (在 winreg 模块中), 2027
HKEY_LOCAL_MACHINE() (在 winreg 模块中), 2027
HKEY_PERFORMANCE_DATA() (在 winreg 模块中), 2027
HKEY_USERS() (在 winreg 模块中), 2027
hline() (curses.window 方法), 863
hls_to_rgb() (在 colorsys 模块中), 1440
hmac
    module, 610
HOME, 436, 1503
home() (pathlib.Path 类方法), 423
home() (在 turtle 模块中), 1466
HOMEDRIVE, 436
HOMEPATH, 436
hook_compressed() (在 fileinput 模块中), 853
hook_encoded() (在 fileinput 模块中), 853
hostmask (ipaddress.IPv4Network 属性), 1428
hostmask (ipaddress.IPv6Network 属性), 1431
hostname_checks_common_name (ssl.SSLContext 属性), 1123
hosts() (ipaddress.IPv4Network 方法), 1429
hosts() (ipaddress.IPv6Network 方法), 1431
hosts (netrc.netrc 属性), 595
host (urllib.request.Request 属性), 1326
hour (datetime.datetime 属性), 204
hour (datetime.time 属性), 212
HRESULT (ctypes 中的类), 781
hStdError (subprocess.STARTUPINFO 属性), 962
hStdInput (subprocess.STARTUPINFO 属性), 961
hStdOutput (subprocess.STARTUPINFO 属性), 961
hsv_to_rgb() (在 colorsys 模块中), 1440
HT() (在 curses.ascii 模块中), 880
ht() (在 turtle 模块中), 1474
HTML, 1249, 1338
html
    module, 1249
html5() (在 html.entities 模块中), 1254
HTMLCalendar (calendar 中的类), 232
HtmlDiff (difflib 中的类), 143
html.entities
    module, 1254
html.parser
    module, 1249
HTMLParser (html.parser 中的类), 1249
htonl() (在 socket 模块中), 1088
htons() (在 socket 模块中), 1088
HTTP
    http (标准模块), 1348
    http.client (标准模块), 1352
    协议, 1337, 1338, 1348, 1352, 1392
http
    module, 1348
HTTP() (在 email.policy 模块中), 1178
http_error_301() (urllib.request.HTTPRedirectHandler 方法), 1330
http_error_302() (urllib.request.HTTPRedirectHandler 方法), 1330
http_error_303() (urllib.request.HTTPRedirectHandler 方法), 1330
http_error_307() (urllib.request.HTTPRedirectHandler 方法), 1330
http_error_308() (urllib.request.HTTPRedirectHandler 方法), 1330
http_error_401() (urllib.request.HTTPBasicAuthHandler 方法), 1331

http_error_401() (urllib.request.HTTPDigestAuthHandler 方法), 1332
http_error_407() (urllib.request.ProxyBasicAuthHandler 方法), 1331
http_error_407() (urllib.request.ProxyDigestAuthHandler 方法), 1332
http_error_auth_reqed() (urllib.request.AbstractBasicAuthHandler 方法), 1331
http_error_auth_reqed() (urllib.request.AbstractDigestAuthHandler 方法), 1331
http_error_default() (urllib.request.BaseHandler 方法), 1329
http_open() (urllib.request.HTTPHandler 方法), 1332
HTTP_PORT() (在 http.client 模块中), 1354
http_response() (urllib.request.HTTPErrorProcessor 方法), 1333
http_version (wsgiref.handlers.BaseHandler 属性), 1319
HTTPBasicAuthHandler (urllib.request 中的类), 1325
http.client
    module, 1352
HTTPConnection (http.client 中的类), 1352
http.cookiejar
    module, 1401
HTTPCookieProcessor (urllib.request 中的类), 1324
http.cookies
    module, 1398
httpd, 1392
HTTPDefaultErrorHandler (urllib.request 中的类), 1324
HTTPDigestAuthHandler (urllib.request 中的类), 1325
HTTPError, 1347
HTTPErrorProcessor (urllib.request 中的类), 1326
HTTPException, 1353
HTTPHandler (logging.handlers 中的类), 739
HTTPHandler (urllib.request 中的类), 1326
HTTPMessage (http.client 中的类), 1359
HTTPMethod (http 中的类), 1351
httponly (http.cookies.Morsel 属性), 1399
HTTPPasswordMgrWithDefaultRealm (urllib.request 中的类), 1325
HTTPPasswordMgrWithPriorAuth (urllib.request 中的类), 1325
HTTPPasswordMgr (urllib.request 中的类), 1325
HTTPRedirectHandler (urllib.request 中的类), 1324
HTTPResponse (http.client 中的类), 1353
https_open() (urllib.request.HTTPSHandler 方法), 1332
HTTPS_PORT() (在 http.client 模块中), 1354
https_response() (urllib.request.HTTPErrorProcessor 方法), 1333
HTTPSConnection (http.client 中的类), 1352
http.server
    module, 1392
    安全, 1398
http.server 命令行选项
    -b, 1397
    --bind, 1397
    --cgi, 1397
    -d, 1397
    --directory, 1397
    -p, 1397
    port, 1397
    --protocol, 1397
HTTPServer (http.server 中的类), 1392
HTTPSHandler (urllib.request 中的类), 1326
HTTPStatus (http 中的类), 1349
HV_GUID_BROADCAST() (在 socket 模块中), 1083
HV_GUID_CHILDREN() (在 socket 模块中), 1083
HV_GUID_LOOPBACK() (在 socket 模块中), 1083
HV_GUID_PARENT() (在 socket 模块中), 1083
HV_GUID_WILDCARD() (在 socket 模块中), 1083
HV_GUID_ZERO() (在 socket 模块中), 1083
HV_PROTOCOL_RAW() (在 socket 模块中), 1083
HVSOCKET_ADDRESS_FLAG_PASSTHRU() (在 socket 模块中), 1083
HVSOCKET_CONNECT_TIMEOUT() (在 socket 模块中), 1083
HVSOCKET_CONNECT_TIMEOUT_MAX() (在 socket 模块中), 1083
HVSOCKET_CONNECTED_SUSPEND() (在 socket 模块中), 1083
hypot() (在 math 模块中), 318

I

-i
    ast 命令行选项, 1976
    compileall 命令行选项, 1993
    idle 命令行选项, 1544
    random 命令行选项, 362
I() (在 re 模块中), 129
I/O 控制
    POSIX, 2035
    tty, 2035
    UNIX, 2039
    缓冲, 23, 1093
iadd() (在 operator 模块中), 408
iand() (在 operator 模块中), 408
iconcat() (在 operator 模块中), 408
id()
    built-in function, 18
id() (unittest.TestCase 方法), 1639
idcok() (curses.window 方法), 863
identchars (cmd.Cmd 属性), 1493
identify() (tkinter.ttk.Notebook 方法), 1526

identify() (tkinter.ttk.Treeview 方法), 1531
identify() (tkinter.ttk.Widget 方法), 1523
identify_column() (tkinter.ttk.Treeview 方法), 1531
identify_element() (tkinter.ttk.Treeview 方法), 1531
identify_region() (tkinter.ttk.Treeview 方法), 1531
identify_row() (tkinter.ttk.Treeview 方法), 1531
ident (select.kevent 属性), 1139
ident (threading.Thread 属性), 890
IDLE, 1537, **2069**
IDLE_PRIORITY_CLASS() (在 subprocess 模块中), 963
idlelib
    module, 1548
IDLESTARTUP, 1544
idle 命令行选项
    -, 1544
    -c, 1544
    -d, 1544
    -e, 1544
    -h, 1544
    -i, 1544
    -r, 1544
    -s, 1544
    -t, 1544
idlok() (curses.window 方法), 863
id (ssl.SSLSession 属性), 1132
if
    statement -- 语句, 35
if_indextoname() (在 socket 模块中), 1090
if_nameindex() (在 socket 模块中), 1090
if_nametoindex() (在 socket 模块中), 1090
IfExp (ast 中的类), 1951
ifloordiv() (在 operator 模块中), 408
If (ast 中的类), 1957
iglob() (在 glob 模块中), 454
ignorableWhitespace() (xml.sax.handler.ContentHandler 方法), 1293
ignore
    错误处理器名称, 176
ignore (*pdb command*), 1742
IGNORE() (在 tkinter.messagebox 模块中), 1519
ignore_environment (sys.flags 属性), 1793
ignore_errors() (在 codecs 模块中), 178
IGNORE_EXCEPTION_DETAIL() (在 doctest 模块中), 1608
ignore_patterns() (在 shutil 模块中), 460
ignore_warnings() (在 test.support.warnings_helper 模块中), 1724
IGNORECASE() (在 re 模块中), 129
--ignore-dir
    trace 命令行选项, 1759
--ignore-module
    trace 命令行选项, 1759
ignore (bdb.Breakpoint 属性), 1732
IISCGIHandler (wsgiref.handlers 中的类), 1317
IllegalMonthError, 236
IllegalWeekdayError, 236
ilshift() (在 operator 模块中), 408
imag (numbers.Complex 属性), 309
imag (sys.hash_info 属性), 1799
imap() (multiprocessing.pool.Pool 方法), 925
IMAP4
    协议, 1368
IMAP4_SSL
    协议, 1368
IMAP4_SSL (imaplib 中的类), 1368
IMAP4_stream
    协议, 1368
IMAP4_stream (imaplib 中的类), 1369
IMAP4.abort, 1368
IMAP4.error, 1368
IMAP4.readonly, 1368
IMAP4 (imaplib 中的类), 1368
imap_unordered() (multiprocessing.pool.Pool 方法), 925
imaplib
    module, 1368
imatmul() (在 operator 模块中), 408
imghdr
    module, 2057
immedok() (curses.window 方法), 864
immortal -- 永生对象, **2069**
immutable -- 不可变对象
    sequence types, 45
immutable -- 不可变对象, **2069**
imod() (在 operator 模块中), 408
imp
    module, 2057
impl_detail()(在 test.support 模块中), 1715
implementation() (在 sys 模块中), 1800
import
    statement -- 语句, 31, 1893
import path -- 导入路径, **2070**
import_fresh_module() (在 test.support.import_helper 模块中), 1723
IMPORT_FROM (*opcode*), 2009
import_module() (在 importlib 模块中), 1910
import_module() (在 test.support.import_helper 模块中), 1724
IMPORT_NAME (*opcode*), 2009
importer -- 导入器, **2070**
ImportError, 101
ImportFrom (ast 中的类), 1957
importing -- 导入, **2070**

importlib
    module, 1909
importlib.abc
    module, 1912
importlib.machinery
    module, 1918
importlib.metadata
    module, 1932
importlib.resources
    module, 1928
importlib.resources.abc
    module, 1931
importlib.util
    module, 1923
ImportWarning, 107
Import (ast 中的类), 1956
ImproperConnectionState, 1353
imul() (在 operator 模块中), 408
in
    operator, 36, 43
in_dll() (ctypes._CData 方法), 779
in_table_a1() (在 stringprep 模块中), 160
in_table_b1() (在 stringprep 模块中), 160
in_table_c3() (在 stringprep 模块中), 160
in_table_c4() (在 stringprep 模块中), 160
in_table_c5() (在 stringprep 模块中), 160
in_table_c6() (在 stringprep 模块中), 160
in_table_c7() (在 stringprep 模块中), 160
in_table_c8() (在 stringprep 模块中), 160
in_table_c9() (在 stringprep 模块中), 160
in_table_c11() (在 stringprep 模块中), 160
in_table_c11_c12() (在 stringprep 模块中), 160
in_table_c12() (在 stringprep 模块中), 160
in_table_c21() (在 stringprep 模块中), 160
in_table_c21_c22() (在 stringprep 模块中), 160
in_table_c22() (在 stringprep 模块中), 160
in_table_d1() (在 stringprep 模块中), 160
in_table_d2() (在 stringprep 模块中), 160
in_transaction (sqlite3.Connection 属性), 509
inch() (curses.window 方法), 864
include() (在 xml.etree.ElementInclude 模块中), 1266
--include-attributes
    ast 命令行选项, 1976
inclusive (tracemalloc.DomainFilter 属性), 1766
inclusive (tracemalloc.Filter 属性), 1767
Incomplete, 1246
IncompleteRead, 1353
IncompleteReadError, 1022
increment_lineno() (在 ast 模块中), 1973
IncrementalDecoder (codecs 中的类), 179
incrementaldecoder (codecs.CodecInfo 属性), 174
IncrementalEncoder (codecs 中的类), 179
incrementalencoder (codecs.CodecInfo 属性), 174
IncrementalNewlineDecoder (io 中的类), 689
IncrementalParser (xml.sax.xmlreader 中的类), 1296
--indent
    ast 命令行选项, 1976
    json.tool 命令行选项, 1221
indent() (在 textwrap 模块中), 155
INDENT() (在 token 模块中), 1980
indent() (在 xml.etree.ElementTree 模块中), 1263
IndentationError, 104
--indentlevel
    pickletools 命令行选项, 2017
indent (doctest.Example 属性), 1616
indent (reprlib.Repr 属性), 289
index() (序列方法), 43
index() (array.array 方法), 268
index() (bytearray 方法), 64
index() (bytes 方法), 64
index() (collections.deque 方法), 244
index() (multiprocessing.shared_memory.ShareableList 方法), 943
index() (str 方法), 51
index() (tkinter.ttk.Notebook 方法), 1526
index() (tkinter.ttk.Treeview 方法), 1531
index() (在 operator 模块中), 403
IndexError, 101
indexOf() (在 operator 模块中), 405
IndexSizeErr, 1281
index (inspect.FrameInfo 属性), 1888
index (inspect.Traceback 属性), 1888
INDIRECT (inspect.BufferFlags 属性), 1892
inet_aton() (在 socket 模块中), 1088
inet_ntoa() (在 socket 模块中), 1088
inet_ntop() (在 socket 模块中), 1089
inet_pton() (在 socket 模块中), 1089
Inexact (decimal 中的类), 342
inf() (在 cmath 模块中), 324
inf() (在 math 模块中), 320
infile
    json.tool 命令行选项, 1220
infile (shlex.shlex 属性), 1498
Infinity, 16
infj() (在 cmath 模块中), 324
--info
    zipapp 命令行选项, 1783
info() (dis.Bytecode 方法), 1998
info() (gettext.NullTranslations 方法), 1444
info() (http.client.HTTPResponse 方法), 1357
info() (logging.Logger 方法), 705
info() (urllib.response.addinfourl 方法), 1338
INFO() (在 logging 模块中), 707
info() (在 logging 模块中), 713

INFO() (在 tkinter.messagebox 模块中), 1519
infolist() (zipfile.ZipFile 方法), 544
inf (sys.hash_info 属性), 1799
.ini
    文件, 576
ini 文件, 576
init() (在 mimetypes 模块中), 1239
init_color() (在 curses 模块中), 856
init_pair() (在 curses 模块中), 856
inited() (在 mimetypes 模块中), 1240
initgroups() (在 os 模块中), 620
initial_indent (textwrap.TextWrapper 属性), 157
initscr() (在 curses 模块中), 856
InitVar (dataclasses 中的类), 1836
inode() (os.DirEntry 方法), 646
input()
    built-in function, 18
input() (在 fileinput 模块中), 852
input_charset (email.charset.Charset 属性), 1206
input_codec (email.charset.Charset 属性), 1206
InputSource(xml.sax.xmlreader 中的类), 1296
InputStream (wsgiref.types 中的类), 1320
insch() (curses.window 方法), 864
insdelln() (curses.window 方法), 864
insert() (序列方法), 45
insert() (array.array 方法), 268
insert() (collections.deque 方法), 244
insert() (tkinter.ttk.Notebook 方法), 1526
insert() (tkinter.ttk.Treeview 方法), 1531
insert() (xml.etree.ElementTree.Element 方法), 1267
insert_text() (在 readline 模块中), 161
insertBefore() (xml.dom.Node 方法), 1276
insertln() (curses.window 方法), 864
insnstr() (curses.window 方法), 864
insort() (在 bisect 模块中), 264
insort_left() (在 bisect 模块中), 264
insort_right() (在 bisect 模块中), 264
inspect
    module, 1875
InspectLoader (importlib.abc 中的类), 1914
inspect 命令行选项
    --details, 1893
inspect (sys.flags 属性), 1793
insstr() (curses.window 方法), 864
install() (gettext.NullTranslations 方法), 1444
install() (在 gettext 模块中), 1443
install_opener() (在 urllib.request 模块中), 1323
install_scripts() (venv.EnvBuilder 方法), 1778
installHandler() (在 unittest 模块中), 1650
instate() (tkinter.ttk.Widget 方法), 1523
instr() (curses.window 方法), 864
instream (shlex.shlex 属性), 1498
INSTRUCTION (*monitoring event*), 1811
Instruction.arg() (在 dis 模块中), 2000
Instruction.argrepr() (在 dis 模块中), 2000
Instruction.argval() (在 dis 模块中), 2000
Instruction.baseopcode() (在 dis 模块中), 2000
Instruction.baseopname() (在 dis 模块中), 2000
Instruction.cache_info() (在 dis 模块中), 2001
Instruction.cache_offset() (在 dis 模块中), 2000
Instruction.end_offset() (在 dis 模块中), 2000
Instruction.is_jump_target() (在 dis 模块中), 2001
Instruction.jump_target() (在 dis 模块中), 2001
Instruction.line_number() (在 dis 模块中), 2000
Instruction.offset() (在 dis 模块中), 2000
Instruction.oparg() (在 dis 模块中), 2000
Instruction.opcode() (在 dis 模块中), 2000
Instruction.opname() (在 dis 模块中), 2000
Instruction.positions() (在 dis 模块中), 2001
Instruction.start_offset() (在 dis 模块中), 2000
Instruction.starts_line() (在 dis 模块中), 2000
Instruction (dis 中的类), 2000
int
    内置函数, 36
Int2AP() (在 imaplib 模块中), 1369
int_info() (在 sys 模块中), 1800
int_max_str_digits (sys.flags 属性), 1793
integer
    object -- 对象, 36
    types, 运算目标, 37
    字面值, 36
--integer
    random 命令行选项, 362
Integral (numbers 中的类), 310
IntegrityError, 514
IntEnum (enum 中的类), 298
interact (*pdb command*), 1744
interact() (code.InteractiveConsole 方法), 1898
interact() (在 code 模块中), 1897
interactive -- 交互, **2070**
InteractiveConsole (code 中的类), 1897
InteractiveInterpreter (code 中的类), 1897
Interactive (ast 中的类), 1945
interactive (sys.flags 属性), 1793
InterfaceError, 514
intern() (在 sys 模块中), 1801
internal_attr (zipfile.ZipInfo 属性), 550

Internaldate2tuple() (在 imaplib 模块中), 1369
InternalError, 514
internalSubset (xml.dom.DocumentType 属性), 1277
INTERNET_TIMEOUT() (在 test.support 模块中), 1710
InterpolationDepthError, 592
InterpolationError, 592
InterpolationMissingOptionError, 592
InterpolationSyntaxError, 592
interpreted -- 解释型, **2070**
interpreter shutdown -- 解释器关闭, **2070**
interpreter_requires_environment() (在 test.support.script_helper 模块中), 1719
interrupt() (sqlite3.Connection 方法), 504
interrupt_main() (在 _thread 模块中), 978
InterruptedError, 106
intersection() (frozenset 方法), 82
intersection_update() (frozenset 方法), 83
IntFlag (enum 中的类), 300
intro (cmd.Cmd 属性), 1493
int (uuid.UUID 属性), 1381
int (内置类), 18
InuseAttributeErr, 1281
inv() (在 operator 模块中), 403
inv_cdf() (statistics.NormalDist 方法), 374
InvalidAccessErr, 1281
invalidate_caches() (importlib.abc.MetaPathFinder 方法), 1912
invalidate_caches() (importlib.abc.PathEntryFinder 方法), 1912
invalidate_caches() (importlib.machinery.FileFinder 方法), 1919
invalidate_caches() (zipimport.zipimporter 方法), 1902
invalidate_caches() (importlib.machinery.PathFinder 类方法), 1919
invalidate_caches() (在 importlib 模块中), 1911
--invalidation-mode
    compileall 命令行选项, 1993
InvalidBase64CharactersDefect, 1180
InvalidBase64LengthDefect, 1180
InvalidBase64PaddingDefect, 1180
InvalidCharacterErr, 1281
InvalidDateDefect, 1180
InvalidFileException, 597
InvalidModificationErr, 1281
InvalidOperation (decimal 中的类), 343
InvalidStateErr, 1281
InvalidStateError, 951, 1022
InvalidTZPathWarning, 230
InvalidURL, 1353
invert() (在 operator 模块中), 403
Invert (ast 中的类), 1949
In (ast 中的类), 1950
io
    module, 677
IO_REPARSE_TAG_APPEXECLINK() (在 stat 模块中), 446
IO_REPARSE_TAG_MOUNT_POINT() (在 stat 模块中), 446
IO_REPARSE_TAG_SYMLINK() (在 stat 模块中), 446
IOBase (io 中的类), 681
ioctl() (socket.socket 方法), 1093
ioctl() (在 fcntl 模块中), 2040
IOCTL_VM_SOCKETS_GET_LOCAL_CID() (在 socket 模块中), 1082
IOError, 105
ior() (在 operator 模块中), 408
ios_ver() (在 platform 模块中), 744
io.StringIO
    object -- 对象, 48
IO (typing 中的类), 1584
ip_address() (在 ipaddress 模块中), 1422
ip_interface() (在 ipaddress 模块中), 1423
ip_network() (在 ipaddress 模块中), 1423
ipaddress
    module, 1422
ipow() (在 operator 模块中), 408
ipv4_mapped (ipaddress.IPv6Address 属性), 1426
IPv4Address (ipaddress 中的类), 1423
IPv4Interface (ipaddress 中的类), 1433
IPv4Network (ipaddress 中的类), 1428
IPV6_ENABLED() (在 test.support.socket_helper 模块中), 1718
ipv6_mapped (ipaddress.IPv4Address 属性), 1425
IPv6Address (ipaddress 中的类), 1425
IPv6Interface (ipaddress 中的类), 1433
IPv6Network (ipaddress 中的类), 1430
ip (ipaddress.IPv4Interface 属性), 1433
ip (ipaddress.IPv6Interface 属性), 1433
irshift() (在 operator 模块中), 408
is
    operator, 36
is not
    operator, 36
is_() (在 operator 模块中), 403
is_absolute() (pathlib.PurePath 方法), 418
is_active() (asyncio.AbstractChildWatcher 方法), 1062
is_active() (graphlib.TopologicalSorter 方法), 307

is_alive() (multiprocessing.Process 方法), 905
is_alive() (threading.Thread 方法), 890
is_android() (在 test.support 模块中), 1710
is_annotated() (symtable.Symbol 方法), 1979
is_assigned() (symtable.Symbol 方法), 1979
is_async (pyclbr.Function 属性), 1990
is_attachment() (email.message.EmailMessage 方法), 1164
is_authenticated() (urllib.request.HTTPPasswordMgrWithPriorAuth 方法), 1331
is_block_device() (pathlib.Path 方法), 426
is_blocked() (http.cookiejar.DefaultCookiePolicy 方法), 1406
is_canonical() (decimal.Context 方法), 339
is_canonical() (decimal.Decimal 方法), 332
is_char_device() (pathlib.Path 方法), 426
IS_CHARACTER_JUNK() (在 difflib 模块中), 146
is_check_supported() (在 lzma 模块中), 539
is_closed() (asyncio.loop 方法), 1024
is_closing() (asyncio.BaseTransport 方法), 1049
is_closing() (asyncio.StreamWriter 方法), 1006
is_dataclass() (在 dataclasses 模块中), 1837
is_declared_global() (symtable.Symbol 方法), 1979
is_dir() (importlib.abc.Traversable 方法), 1917
is_dir() (importlib.resources.abc.Traversable 方法), 1931
is_dir() (os.DirEntry 方法), 646
is_dir() (pathlib.Path 方法), 425
is_dir() (zipfile.Path 方法), 547
is_dir() (zipfile.ZipInfo 方法), 549
is_enabled() (在 faulthandler 模块中), 1737
is_expired() (http.cookiejar.Cookie 方法), 1409
is_fifo() (pathlib.Path 方法), 425
is_file() (importlib.abc.Traversable 方法), 1917
is_file() (importlib.resources.abc.Traversable 方法), 1932
is_file() (os.DirEntry 方法), 646
is_file() (pathlib.Path 方法), 425
is_file() (zipfile.Path 方法), 547
is_finalized() (在 gc 模块中), 1874
is_finalizing() (在 sys 模块中), 1801
is_finite() (decimal.Context 方法), 339
is_finite() (decimal.Decimal 方法), 332
is_free() (symtable.Symbol 方法), 1979
is_global() (symtable.Symbol 方法), 1978
is_global (ipaddress.IPv4Address 属性), 1424
is_global (ipaddress.IPv6Address 属性), 1426
is_hop_by_hop() (在 wsgiref.util 模块中), 1313
is_imported() (symtable.Symbol 方法), 1978
is_infinite() (decimal.Context 方法), 339
is_infinite() (decimal.Decimal 方法), 332
is_integer() (float 方法), 40
is_integer() (fractions.Fraction 方法), 352
is_integer() (int 方法), 40
is_junction() (os.DirEntry 方法), 647
is_junction() (pathlib.Path 方法), 425
is_jython() (在 test.support 模块中), 1710
IS_LINE_JUNK() (在 difflib 模块中), 146
is_linetouched() (curses.window 方法), 864
is_link_local (ipaddress.IPv4Address 属性), 1425
is_link_local (ipaddress.IPv4Network 属性), 1428
is_link_local (ipaddress.IPv6Address 属性), 1426
is_link_local (ipaddress.IPv6Network 属性), 1431
is_local() (symtable.Symbol 方法), 1979
is_loopback (ipaddress.IPv4Address 属性), 1425
is_loopback (ipaddress.IPv4Network 属性), 1428
is_loopback (ipaddress.IPv6Address 属性), 1426
is_loopback (ipaddress.IPv6Network 属性), 1431
is_mount() (pathlib.Path 方法), 425
is_multicast (ipaddress.IPv4Address 属性), 1424
is_multicast (ipaddress.IPv4Network 属性), 1428
is_multicast (ipaddress.IPv6Address 属性), 1426
is_multicast (ipaddress.IPv6Network 属性), 1431
is_multipart() (email.message.EmailMessage 方法), 1161
is_multipart() (email.message.Message 方法), 1195
is_namespace() (symtable.Symbol 方法), 1979
is_nan() (decimal.Context 方法), 339
is_nan() (decimal.Decimal 方法), 332
is_nested() (symtable.SymbolTable 方法), 1977
is_nonlocal() (symtable.Symbol 方法), 1979
is_normal() (decimal.Context 方法), 339
is_normal() (decimal.Decimal 方法), 332
is_normalized() (在 unicodedata 模块中), 159
is_not() (在 operator 模块中), 403
is_not_allowed() (http.cookiejar.DefaultCookiePolicy 方法), 1407
IS_OP (*opcode*), 2009

is_optimized() (symtable.SymbolTable 方法), 1977
is_package() (importlib.abc.InspectLoader 方法), 1914
is_package() (importlib.abc.SourceLoader 方法), 1916
is_package() (importlib.machinery.ExtensionFileLoader 方法), 1921
is_package() (importlib.machinery.SourceFileLoader 方法), 1920
is_package() (importlib.machinery.SourcelessFileLoader 方法), 1920
is_package() (zipimport.zipimporter 方法), 1902
is_parameter() (symtable.Symbol 方法), 1978
is_private (ipaddress.IPv4Address 属性), 1424
is_private (ipaddress.IPv4Network 属性), 1428
is_private (ipaddress.IPv6Address 属性), 1426
is_private (ipaddress.IPv6Network 属性), 1431
is_protocol() (在 typing 模块中), 1590
is_python_build() (在 sysconfig 模块中), 1819
is_qnan() (decimal.Context 方法), 339
is_qnan() (decimal.Decimal 方法), 332
is_reading() (asyncio.ReadTransport 方法), 1050
is_referenced() (symtable.Symbol 方法), 1978
is_relative_to() (pathlib.PurePath 方法), 418
is_reserved() (pathlib.PurePath 方法), 418
is_reserved (ipaddress.IPv4Address 属性), 1425
is_reserved (ipaddress.IPv4Network 属性), 1428
is_reserved (ipaddress.IPv6Address 属性), 1426
is_reserved (ipaddress.IPv6Network 属性), 1431
is_resource() (importlib.abc.ResourceReader 方法), 1916
is_resource() (importlib.resources.abc.ResourceReader 方法), 1931
is_resource() (在 importlib.resources 模块中), 1930
is_resource_enabled() (在 test.support 模块中), 1712
is_running() (asyncio.loop 方法), 1024
is_safe (uuid.UUID 属性), 1382
is_serving() (asyncio.Server 方法), 1042
is_set() (asyncio.Event 方法), 1011
is_set() (threading.Event 方法), 895
is_signed() (decimal.Context 方法), 339
is_signed() (decimal.Decimal 方法), 332
is_site_local (ipaddress.IPv6Address 属性), 1426
is_site_local (ipaddress.IPv6Network 属性), 1432
is_skipped_line() (bdb.Bdb 方法), 1733
is_socket() (pathlib.Path 方法), 425
is_stack_trampoline_active() (在 sys 模块中), 1807
is_subnormal() (decimal.Context 方法), 339
is_subnormal() (decimal.Decimal 方法), 332
is_symlink() (os.DirEntry 方法), 646
is_symlink() (pathlib.Path 方法), 425
is_symlink() (zipfile.Path 方法), 547
is_tarfile() (在 tarfile 模块中), 554
is_term_resized() (在 curses 模块中), 857
is_tracing() (在 tracemalloc 模块中), 1765
is_tracked() (在 gc 模块中), 1873
is_typeddict() (在 typing 模块中), 1590
is_unspecified (ipaddress.IPv4Address 属性), 1425
is_unspecified (ipaddress.IPv4Network 属性), 1428
is_unspecified (ipaddress.IPv6Address 属性), 1426
is_unspecified (ipaddress.IPv6Network 属性), 1431
is_valid() (string.Template 方法), 120
is_wintouched() (curses.window 方法), 864
is_zero() (decimal.Context 方法), 339
is_zero() (decimal.Decimal 方法), 332
is_zipfile() (在 zipfile 模块中), 542
isabs() (在 os.path 模块中), 436
isabstract() (在 inspect 模块中), 1879
IsADirectoryError, 106
isalnum() (bytearray 方法), 69
isalnum() (bytes 方法), 69
isalnum() (str 方法), 51
isalnum() (在 curses.ascii 模块中), 881
isalpha() (bytearray 方法), 69
isalpha() (bytes 方法), 69
isalpha() (str 方法), 51
isalpha() (在 curses.ascii 模块中), 881
isascii() (bytearray 方法), 69
isascii() (bytes 方法), 69
isascii() (str 方法), 51
isascii() (在 curses.ascii 模块中), 882
isasyncgen() (在 inspect 模块中), 1879
isasyncgenfunction() (在 inspect 模块中), 1879
isatty() (io.IOBase 方法), 681
isatty() (在 os 模块中), 627
isawaitable() (在 inspect 模块中), 1878
isblank() (在 curses.ascii 模块中), 882
isblk() (tarfile.TarInfo 方法), 561
isbuiltin() (在 inspect 模块中), 1879

ischr() (tarfile.TarInfo 方法), 561
isclass() (在 inspect 模块中), 1878
isclose() (在 cmath 模块中), 323
isclose() (在 math 模块中), 315
iscntrl() (在 curses.ascii 模块中), 882
iscode() (在 inspect 模块中), 1879
iscoroutine() (在 asyncio 模块中), 998
iscoroutine() (在 inspect 模块中), 1878
iscoroutinefunction() (在 inspect 模块中), 1878
isctrl() (在 curses.ascii 模块中), 882
isDaemon() (threading.Thread 方法), 890
isdatadescriptor()(在 inspect 模块中), 1880
isdecimal() (str 方法), 51
isdev() (tarfile.TarInfo 方法), 561
isdevdrive() (在 os.path 模块中), 437
isdigit() (bytearray 方法), 69
isdigit() (bytes 方法), 69
isdigit() (str 方法), 51
isdigit() (在 curses.ascii 模块中), 882
isdir() (tarfile.TarInfo 方法), 561
isdir() (在 os.path 模块中), 437
isdisjoint() (frozenset 方法), 82
isdown() (在 turtle 模块中), 1471
iselement() (在 xml.etree.ElementTree 模块中), 1263
isenabled() (在 gc 模块中), 1872
isEnabledFor() (logging.Logger 方法), 704
isendwin() (在 curses 模块中), 857
ISEOF() (在 token 模块中), 1980
isfifo() (tarfile.TarInfo 方法), 561
isfile() (tarfile.TarInfo 方法), 561
isfile() (在 os.path 模块中), 437
isfinite() (在 cmath 模块中), 323
isfinite() (在 math 模块中), 316
isfirstline() (在 fileinput 模块中), 852
isframe() (在 inspect 模块中), 1879
isfunction() (在 inspect 模块中), 1878
isfuture() (在 asyncio 模块中), 1045
isgenerator() (在 inspect 模块中), 1878
isgeneratorfunction() (在 inspect 模块中), 1878
isgetsetdescriptor() (在 inspect 模块中), 1880
isgraph() (在 curses.ascii 模块中), 882
isidentifier() (str 方法), 51
isinf() (在 cmath 模块中), 323
isinf() (在 math 模块中), 316
isinstance()
    built-in function, 19
isjunction() (在 os.path 模块中), 437
iskeyword() (在 keyword 模块中), 1984
isleap() (在 calendar 模块中), 234
islice() (在 itertools 模块中), 383
islink() (在 os.path 模块中), 437
islnk() (tarfile.TarInfo 方法), 561
islower() (bytearray 方法), 69
islower() (bytes 方法), 69
islower() (str 方法), 52
islower() (在 curses.ascii 模块中), 882
ismemberdescriptor() (在 inspect 模块中), 1880
ismeta() (在 curses.ascii 模块中), 882
ismethod() (在 inspect 模块中), 1878
ismethoddescriptor() (在 inspect 模块中), 1879
ismethodwrapper() (在 inspect 模块中), 1879
ismodule() (在 inspect 模块中), 1878
ismount() (在 os.path 模块中), 437
isnan() (在 cmath 模块中), 323
isnan() (在 math 模块中), 316
ISNONTERMINAL() (在 token 模块中), 1980
IsNot (ast 中的类), 1950
isnumeric() (str 方法), 52
isocalendar() (datetime.date 方法), 199
isocalendar() (datetime.datetime 方法), 208
isoformat() (datetime.date 方法), 199
isoformat() (datetime.datetime 方法), 208
isoformat() (datetime.time 方法), 213
IsolatedAsyncioTestCase (unittest 中的类), 1640
isolated (sys.flags 属性), 1793
isolation_level (sqlite3.Connection 属性), 509
isoweekday() (datetime.date 方法), 199
isoweekday() (datetime.datetime 方法), 208
isprint() (在 curses.ascii 模块中), 882
isprintable() (str 方法), 52
ispunct() (在 curses.ascii 模块中), 882
isqrt() (在 math 模块中), 314
isreadable() (pprint.PrettyPrinter 方法), 285
isreadable() (在 pprint 模块中), 284
isrecursive() (pprint.PrettyPrinter 方法), 285
isrecursive() (在 pprint 模块中), 284
isreg() (tarfile.TarInfo 方法), 561
isreserved() (在 os.path 模块中), 437
isReservedKey() (http.cookies.Morsel 方法), 1400
isroutine() (在 inspect 模块中), 1879
isSameNode() (xml.dom.Node 方法), 1276
issoftkeyword() (在 keyword 模块中), 1984
isspace() (bytearray 方法), 70
isspace() (bytes 方法), 70
isspace() (str 方法), 52
isspace() (在 curses.ascii 模块中), 882
isstdin() (在 fileinput 模块中), 852
issubclass()
    built-in function, 19
issubset() (frozenset 方法), 82
issuperset() (frozenset 方法), 82
issym() (tarfile.TarInfo 方法), 561
ISTERMINAL() (在 token 模块中), 1980
istitle() (bytearray 方法), 70

istitle() (bytes 方法), 70
istitle() (str 方法), 52
istraceback() (在 inspect 模块中), 1879
isub() (在 operator 模块中), 409
isupper() (bytearray 方法), 70
isupper() (bytes 方法), 70
isupper() (str 方法), 52
isupper() (在 curses.ascii 模块中), 882
isvisible() (在 turtle 模块中), 1474
isxdigit() (在 curses.ascii 模块中), 882
Is (ast 中的类), 1950
ITALIC() (在 tkinter.font 模块中), 1513
item() (tkinter.ttk.Treeview 方法), 1532
item() (xml.dom.NamedNodeMap 方法), 1280
item() (xml.dom.NodeList 方法), 1277
itemgetter() (在 operator 模块中), 406
items() (configparser.ConfigParser 方法), 590
items() (contextvars.Context 方法), 977
items() (dict 方法), 85
items() (email.message.EmailMessage 方法), 1162
items() (email.message.Message 方法), 1197
items() (mailbox.Mailbox 方法), 1223
items() (types.MappingProxyType 方法), 280
items() (xml.etree.ElementTree.Element 方法), 1267
itemsize (array.array 属性), 267
itemsize (memoryview 属性), 80
ItemsView (collections.abc 中的类), 258
ItemsView (typing 中的类), 1594
iter()
    built-in function, 19
iter() (xml.etree.ElementTree.Element 方法), 1267
iter() (xml.etree.ElementTree.ElementTree 方法), 1269
iter_attachments() (email.message.EmailMessage 方法), 1165
iter_child_nodes() (在 ast 模块中), 1973
iter_fields() (在 ast 模块中), 1973
iter_importers() (在 pkgutil 模块中), 1904
iter_modules() (在 pkgutil 模块中), 1904
iter_parts() (email.message.EmailMessage 方法), 1166
iter_unpack() (struct.Struct 方法), 173
iter_unpack() (在 struct 模块中), 168
iterable -- 可迭代对象, **2070**
Iterable (collections.abc 中的类), 257
Iterable (typing 中的类), 1596
iterator -- 迭代器, **2070**
Iterator (collections.abc 中的类), 257
Iterator (typing 中的类), 1596
iterdecode() (在 codecs 模块中), 176
iterdir() (importlib.abc.Traversable 方法), 1917
iterdir() (importlib.resources.abc.Traversable 方法), 1931
iterdir() (pathlib.Path 方法), 427
iterdir() (zipfile.Path 方法), 547
iterdump() (sqlite3.Connection 方法), 506
iterencode() (json.JSONEncoder 方法), 1218
iterencode() (在 codecs 模块中), 175
iterfind() (xml.etree.ElementTree.Element 方法), 1268
iterfind() (xml.etree.ElementTree.ElementTree 方法), 1269
iteritems() (mailbox.Mailbox 方法), 1222
iterkeys() (mailbox.Mailbox 方法), 1222
itermonthdates() (calendar.Calendar 方法), 231
itermonthdays() (calendar.Calendar 方法), 231
itermonthdays2() (calendar.Calendar 方法), 231
itermonthdays3() (calendar.Calendar 方法), 231
itermonthdays4() (calendar.Calendar 方法), 231
iterparse() (在 xml.etree.ElementTree 模块中), 1263
itertext() (xml.etree.ElementTree.Element 方法), 1268
itertools
    module, 377
itervalues() (mailbox.Mailbox 方法), 1222
iterweekdays() (calendar.Calendar 方法), 231
ITIMER_PROF() (在 signal 模块中), 1147
ITIMER_REAL() (在 signal 模块中), 1147
ITIMER_VIRTUAL() (在 signal 模块中), 1147
ItimerError, 1147
itruediv() (在 operator 模块中), 409
ixor() (在 operator 模块中), 409

J

-j
    compileall 命令行选项, 1993
JANUARY() (在 calendar 模块中), 236
java_ver() (在 platform 模块中), 743
join() (asyncio.Queue 方法), 1020
join() (bytearray 方法), 64
join() (bytes 方法), 64
join() (multiprocessing.JoinableQueue 方法), 910
join() (multiprocessing.pool.Pool 方法), 926
join() (multiprocessing.Process 方法), 905
join() (queue.Queue 方法), 973
join() (str 方法), 52
join() (threading.Thread 方法), 889
join() (在 os.path 模块中), 438
join() (在 shlex 模块中), 1496

join_thread() (multiprocessing.Queue 方法), 909
join_thread()(在 test.support.threading_helper 模块中), 1720
JoinableQueue(multiprocessing 中的类), 909
JoinedStr (ast 中的类), 1947
joinpath() (importlib.abc.Traversable 方法), 1917
joinpath() (importlib.resources.abc.Traversable 方法), 1932
joinpath() (pathlib.PurePath 方法), 419
joinpath() (zipfile.Path 方法), 548
js_output() (http.cookies.BaseCookie 方法), 1399
js_output() (http.cookies.Morsel 方法), 1400
json
    module, 1212
JSONDecodeError, 1218
JSONDecoder (json 中的类), 1216
JSONEncoder (json 中的类), 1217
--json-lines
    json.tool 命令行选项, 1221
json.tool
    module, 1220
json.tool 命令行选项
    --compact, 1221
    -h, 1221
    --help, 1221
    --indent, 1221
    infile, 1220
    --json-lines, 1221
    --no-ensure-ascii, 1221
    --no-indent, 1221
    outfile, 1220
    --sort-keys, 1221
    --tab, 1221
JULY() (在 calendar 模块中), 236
JUMP (*monitoring event*), 1811
JUMP (*opcode*), 2016
jump (*pdb command*), 1743
JUMP_BACKWARD (*opcode*), 2009
JUMP_BACKWARD_NO_INTERRUPT (*opcode*), 2009
JUMP_FORWARD (*opcode*), 2009
JUMP_NO_INTERRUPT (*opcode*), 2016
JUNE() (在 calendar 模块中), 236

K

-k
    unittest 命令行选项, 1625
kbhit() (在 msvcrt 模块中), 2020
kde() (在 statistics 模块中), 366
kde_random() (在 statistics 模块中), 367
KEEP (enum.FlagBoundary 属性), 303
kevent() (在 select 模块中), 1135
key function -- 键函数, **2070**
KEY_A1() (在 curses 模块中), 870
KEY_A3() (在 curses 模块中), 871
KEY_ALL_ACCESS() (在 winreg 模块中), 2027
KEY_B2() (在 curses 模块中), 871
KEY_BACKSPACE() (在 curses 模块中), 869
KEY_BEG() (在 curses 模块中), 871
KEY_BREAK() (在 curses 模块中), 869
KEY_BTAB() (在 curses 模块中), 871
KEY_C1() (在 curses 模块中), 871
KEY_C3() (在 curses 模块中), 871
KEY_CANCEL() (在 curses 模块中), 871
KEY_CATAB() (在 curses 模块中), 870
KEY_CLEAR() (在 curses 模块中), 870
KEY_CLOSE() (在 curses 模块中), 871
KEY_COMMAND() (在 curses 模块中), 871
KEY_COPY() (在 curses 模块中), 871
KEY_CREATE() (在 curses 模块中), 871
KEY_CREATE_LINK() (在 winreg 模块中), 2028
KEY_CREATE_SUB_KEY() (在 winreg 模块中), 2028
KEY_CTAB() (在 curses 模块中), 870
KEY_DC() (在 curses 模块中), 869
KEY_DL() (在 curses 模块中), 869
KEY_DOWN() (在 curses 模块中), 869
KEY_EIC() (在 curses 模块中), 870
KEY_END() (在 curses 模块中), 871
KEY_ENTER() (在 curses 模块中), 870
KEY_ENUMERATE_SUB_KEYS()(在 winreg 模块中), 2028
KEY_EOL() (在 curses 模块中), 870
KEY_EOS() (在 curses 模块中), 870
KEY_EXECUTE() (在 winreg 模块中), 2027
KEY_EXIT() (在 curses 模块中), 871
KEY_F0() (在 curses 模块中), 869
KEY_FIND() (在 curses 模块中), 871
KEY_Fn() (在 curses 模块中), 869
KEY_HELP() (在 curses 模块中), 871
KEY_HOME() (在 curses 模块中), 869
KEY_IC() (在 curses 模块中), 870
KEY_IL() (在 curses 模块中), 869
KEY_LEFT() (在 curses 模块中), 869
KEY_LL() (在 curses 模块中), 870
KEY_MARK() (在 curses 模块中), 871
KEY_MAX() (在 curses 模块中), 874
KEY_MESSAGE() (在 curses 模块中), 871
KEY_MIN() (在 curses 模块中), 869
KEY_MOUSE() (在 curses 模块中), 874
KEY_MOVE() (在 curses 模块中), 871
KEY_NEXT() (在 curses 模块中), 872
KEY_NOTIFY() (在 winreg 模块中), 2028
KEY_NPAGE() (在 curses 模块中), 870
KEY_OPEN() (在 curses 模块中), 872
KEY_OPTIONS() (在 curses 模块中), 872
KEY_PPAGE() (在 curses 模块中), 870
KEY_PREVIOUS() (在 curses 模块中), 872
KEY_PRINT() (在 curses 模块中), 870
KEY_QUERY_VALUE() (在 winreg 模块中), 2027
KEY_READ() (在 winreg 模块中), 2027
KEY_REDO() (在 curses 模块中), 872
KEY_REFERENCE() (在 curses 模块中), 872

KEY_REFRESH() (在 curses 模块中), 872
KEY_REPLACE() (在 curses 模块中), 872
KEY_RESET() (在 curses 模块中), 870
KEY_RESIZE() (在 curses 模块中), 874
KEY_RESTART() (在 curses 模块中), 872
KEY_RESUME() (在 curses 模块中), 872
KEY_RIGHT() (在 curses 模块中), 869
KEY_SAVE() (在 curses 模块中), 872
KEY_SBEG() (在 curses 模块中), 872
KEY_SCANCEL() (在 curses 模块中), 872
KEY_SCOMMAND() (在 curses 模块中), 872
KEY_SCOPY() (在 curses 模块中), 872
KEY_SCREATE() (在 curses 模块中), 872
KEY_SDC() (在 curses 模块中), 872
KEY_SDL() (在 curses 模块中), 872
KEY_SELECT() (在 curses 模块中), 873
KEY_SEND() (在 curses 模块中), 873
KEY_SEOL() (在 curses 模块中), 873
KEY_SET_VALUE() (在 winreg 模块中), 2028
KEY_SEXIT() (在 curses 模块中), 873
KEY_SF() (在 curses 模块中), 870
KEY_SFIND() (在 curses 模块中), 873
KEY_SHELP() (在 curses 模块中), 873
KEY_SHOME() (在 curses 模块中), 873
KEY_SIC() (在 curses 模块中), 873
KEY_SLEFT() (在 curses 模块中), 873
KEY_SMESSAGE() (在 curses 模块中), 873
KEY_SMOVE() (在 curses 模块中), 873
KEY_SNEXT() (在 curses 模块中), 873
KEY_SOPTIONS() (在 curses 模块中), 873
KEY_SPREVIOUS() (在 curses 模块中), 873
KEY_SPRINT() (在 curses 模块中), 873
KEY_SR() (在 curses 模块中), 870
KEY_SREDO() (在 curses 模块中), 873
KEY_SREPLACE() (在 curses 模块中), 873
KEY_SRESET() (在 curses 模块中), 870
KEY_SRIGHT() (在 curses 模块中), 873
KEY_SRSUME() (在 curses 模块中), 874
KEY_SSAVE() (在 curses 模块中), 874
KEY_SSUSPEND() (在 curses 模块中), 874
KEY_STAB() (在 curses 模块中), 870
KEY_SUNDO() (在 curses 模块中), 874
KEY_SUSPEND() (在 curses 模块中), 874
KEY_UNDO() (在 curses 模块中), 874
KEY_UP() (在 curses 模块中), 869
KEY_WOW64_32KEY() (在 winreg 模块中), 2028
KEY_WOW64_64KEY() (在 winreg 模块中), 2028
KEY_WRITE() (在 winreg 模块中), 2027
KeyboardInterrupt, 101
KeyError, 101
keylog_filename (ssl.SSLContext 属性), 1122
keyname() (在 curses 模块中), 857
keypad() (curses.window 方法), 864
keyrefs() (weakref.WeakKeyDictionary 方法), 271
keys() (contextvars.Context 方法), 977
keys() (dict 方法), 85
keys() (email.message.EmailMessage 方法), 1162
keys() (email.message.Message 方法), 1197
keys() (mailbox.Mailbox 方法), 1222
keys() (sqlite3.Row 方法), 513
keys() (types.MappingProxyType 方法), 280
keys() (xml.etree.ElementTree.Element 方法), 1267
KeysView (collections.abc 中的类), 258
KeysView (typing 中的类), 1594
keyword
    module, 1984
keyword argument -- 关键字参数, **2071**
keywords (functools.partial 属性), 402
keyword (ast 中的类), 1951
key (http.cookies.Morsel 属性), 1400
key (zoneinfo.ZoneInfo 属性), 228
kill() (asyncio.subprocess.Process 方法), 1018
kill() (asyncio.SubprocessTransport 方法), 1052
kill() (multiprocessing.Process 方法), 906
kill() (subprocess.Popen 方法), 960
kill() (在 os 模块中), 664
kill_python()(在 test.support.script_helper 模块中), 1720
killchar() (在 curses 模块中), 857
killpg() (在 os 模块中), 664
kind (inspect.Parameter 属性), 1883
knownfiles() (在 mimetypes 模块中), 1240
kqueue() (在 select 模块中), 1135
KqueueSelector (selectors 中的类), 1143
KW_ONLY() (在 dataclasses 模块中), 1838
kwargs (inspect.BoundArguments 属性), 1885
kwargs (typing.ParamSpec 属性), 1576
kwlist() (在 keyword 模块中), 1984

L

-L
    calendar 命令行选项, 237
-l
    calendar 命令行选项, 238
    compileall 命令行选项, 1993
    pickletools 命令行选项, 2017
    tarfile 命令行选项, 565
    trace 命令行选项, 1759
    zipfile 命令行选项, 551
L() (在 re 模块中), 129
lambda, **2071**
LambdaType() (在 types 模块中), 278
Lambda (ast 中的类), 1968
LANG, 1441, 1442, 1448, 1452
LANGUAGE, 1441, 1442
LARGEST() (在 test.support 模块中), 1711
LargeZipFile, 542
last_accepted(multiprocessing.connection.Listener 属性), 928
last_exc() (在 sys 模块中), 1801

last_traceback() (在 sys 模块中), 1801
last_type() (在 sys 模块中), 1801
last_value() (在 sys 模块中), 1801
lastChild (xml.dom.Node 属性), 1276
lastcmd (cmd.Cmd 属性), 1493
lastgroup (re.Match 属性), 137
lastindex (re.Match 属性), 137
lastResort() (在 logging 模块中), 717
lastrowid (sqlite3.Cursor 属性), 512
layout() (tkinter.ttk.Style 方法), 1534
lazycache() (在 linecache 模块中), 457
LazyLoader (importlib.util 中的类), 1925
LBRACE() (在 token 模块中), 1982
LBYL, **2071**
LC_ALL, 1441, 1442
LC_ALL() (在 locale 模块中), 1454
LC_COLLATE() (在 locale 模块中), 1453
LC_CTYPE() (在 locale 模块中), 1453
LC_MESSAGES, 1441, 1442
LC_MESSAGES() (在 locale 模块中), 1453
LC_MONETARY() (在 locale 模块中), 1453
LC_NUMERIC() (在 locale 模块中), 1453
LC_TIME() (在 locale 模块中), 1453
lchflags() (在 os 模块中), 639
lchmod() (pathlib.Path 方法), 432
lchmod() (在 os 模块中), 639
lchown() (在 os 模块中), 640
lcm() (在 math 模块中), 314
ldexp() (在 math 模块中), 316
le() (在 operator 模块中), 402
leapdays() (在 calendar 模块中), 234
leaveok() (curses.window 方法), 864
left() (在 turtle 模块中), 1464
left_list (filecmp.dircmp 属性), 447
left_only (filecmp.dircmp 属性), 447
LEFTSHIFT() (在 token 模块中), 1983
LEFTSHIFTEQUAL() (在 token 模块中), 1983
left (filecmp.dircmp 属性), 447
LEGACY_TRANSACTION_CONTROL() (在 sqlite3 模块中), 498
len
    内置函数, 43, 83
len()
    built-in function, 20
length_hint() (在 operator 模块中), 405
length (xml.dom.NamedNodeMap 属性), 1280
length (xml.dom.NodeList 属性), 1277
LESS() (在 token 模块中), 1982
LESSEQUAL() (在 token 模块中), 1982
level (logging.Logger 属性), 703
lexical analyzer -- 词法分析器, **2071**
LexicalHandler (xml.sax.handler 中的类), 1290
lexists() (在 os.path 模块中), 436
LF() (在 curses.ascii 模块中), 880
lgamma() (在 math 模块中), 320
libc_ver() (在 platform 模块中), 744
LIBRARIES_ASSEMBLY_NAME_PREFIX() (在 msvcrt 模块中), 2021
library() (在 dbm.ndbm 模块中), 492
LibraryLoader (ctypes 中的类), 772
library (ssl.SSLError 属性), 1104
license (内置变量), 34
LifoQueue (asyncio 中的类), 1021
LifoQueue (queue 中的类), 971
limit_denominator() (fractions.Fraction 方法), 353
LimitOverrunError, 1022
LINE (*monitoring event*), 1811
line_buffering (io.TextIOWrapper 属性), 688
line_num (csv.csvreader 属性), 574
linear_regression() (在 statistics 模块中), 372
linecache
    module, 457
lineno() (在 fileinput 模块中), 852
lineno (ast.AST 属性), 1944
lineno (doctest.DocTest 属性), 1615
lineno (doctest.Example 属性), 1616
lineno (inspect.FrameInfo 属性), 1888
lineno (inspect.Traceback 属性), 1888
lineno (json.JSONDecodeError 属性), 1218
lineno (netrc.NetrcParseError 属性), 594
lineno (pyclbr.Class 属性), 1990
lineno (pyclbr.Function 属性), 1990
lineno (re.PatternError 属性), 134
lineno (shlex.shlex 属性), 1499
lineno (SyntaxError 属性), 104
lineno (traceback.FrameSummary 属性), 1867
lineno (traceback.TracebackException 属性), 1865
lineno (tracemalloc.Filter 属性), 1767
lineno (tracemalloc.Frame 属性), 1767
lineno (xml.parsers.expat.ExpatError 属性), 1305
LINES, 855, 860
--lines
    calendar 命令行选项, 238
LINES() (在 curses 模块中), 867
linesep() (在 os 模块中), 676
linesep (email.policy.Policy 属性), 1174
lines (os.terminal_size 属性), 635
lineterminator (csv.Dialect 属性), 573
LineTooLong, 1353
line (bdb.Breakpoint 属性), 1732
line (traceback.FrameSummary 属性), 1867
link() (在 os 模块中), 640
linkname (tarfile.TarInfo 属性), 560
LinkOutsideDestinationError, 554
list
    object -- 对象, 45
    type, 运算目标, 45
--list
    tarfile 命令行选项, 565
    zipfile 命令行选项, 551

list (*pdb command*), 1743
list -- 列表, **2071**
list comprehension -- 列表推导式, **2071**
list() (imaplib.IMAP4 方法), 1370
list() (multiprocessing.managers.SyncManager 方法), 920
list() (poplib.POP3 方法), 1366
list() (tarfile.TarFile 方法), 557
LIST_APPEND (*opcode*), 2004
list_dialects() (在 csv 模块中), 570
LIST_EXTEND (*opcode*), 2008
list_folders() (mailbox.Maildir 方法), 1225
list_folders() (mailbox.MH 方法), 1228
ListComp (ast 中的类), 1952
listdir() (在 os 模块中), 640
listdrives() (在 os 模块中), 640
listen() (socket.socket 方法), 1093
listen() (在 logging.config 模块中), 719
listen() (在 turtle 模块中), 1482
listener (logging.handlers.QueueHandler 属性), 740
Listener (multiprocessing.connection 中的类), 927
--listfuncs
    trace 命令行选项, 1759
listMethods() (xmlrpc.client.ServerProxy.system 方法), 1411
listmounts() (在 os 模块中), 641
listvolumes() (在 os 模块中), 641
listxattr() (在 os 模块中), 660
List (ast 中的类), 1947
List (typing 中的类), 1592
list (内置类), 45
Literal() (在 typing 模块中), 1565
literal_eval() (在 ast 模块中), 1972
LiteralString() (在 typing 模块中), 1561
LittleEndianStructure (ctypes 中的类), 782
LittleEndianUnion (ctypes 中的类), 782
ljust() (bytearray 方法), 66
ljust() (bytes 方法), 66
ljust() (str 方法), 52
LK_LOCK() (在 msvcrt 模块中), 2019
LK_NBLCK() (在 msvcrt 模块中), 2019
LK_NBRLCK() (在 msvcrt 模块中), 2019
LK_RLCK() (在 msvcrt 模块中), 2019
LK_UNLCK() (在 msvcrt 模块中), 2019
ll (*pdb command*), 1743
LMTP (smtplib 中的类), 1375
ln() (decimal.Context 方法), 339
ln() (decimal.Decimal 方法), 332
LNKTYPE() (在 tarfile 模块中), 554
load() (http.cookiejar.FileCookieJar 方法), 1404
load() (http.cookies.BaseCookie 方法), 1399
load() (pickle.Unpickler 方法), 473
load() (tracemalloc.Snapshot 类方法), 1768
load() (在 json 模块中), 1215
load() (在 marshal 模块中), 487
load() (在 pickle 模块中), 471
load() (在 plistlib 模块中), 595
load() (在 tomllib 模块中), 593
LOAD_ASSERTION_ERROR (*opcode*), 2005
LOAD_ATTR (*opcode*), 2008
LOAD_BUILD_CLASS (*opcode*), 2006
load_cert_chain() (ssl.SSLContext 方法), 1118
LOAD_CLOSURE (*opcode*), 2016
LOAD_CONST (*opcode*), 2007
load_default_certs() (ssl.SSLContext 方法), 1118
LOAD_DEREF (*opcode*), 2011
load_dh_params() (ssl.SSLContext 方法), 1121
load_extension() (sqlite3.Connection 方法), 506
LOAD_FAST (*opcode*), 2010
LOAD_FAST_AND_CLEAR (*opcode*), 2010
LOAD_FAST_CHECK (*opcode*), 2010
LOAD_FAST_LOAD_FAST (*opcode*), 2010
LOAD_FROM_DICT_OR_DEREF (*opcode*), 2011
LOAD_FROM_DICT_OR_GLOBALS (*opcode*), 2007
LOAD_GLOBAL (*opcode*), 2010
LOAD_LOCALS (*opcode*), 2007
LOAD_METHOD (*opcode*), 2016
load_module() (importlib.abc.FileLoader 方法), 1915
load_module() (importlib.abc.InspectLoader 方法), 1914
load_module() (importlib.abc.Loader 方法), 1913
load_module() (importlib.abc.SourceLoader 方法), 1916
load_module() (importlib.machinery.SourceFileLoader 方法), 1920
load_module() (importlib.machinery.SourcelessFileLoader 方法), 1920
load_module() (zipimport.zipimporter 方法), 1902
LOAD_NAME (*opcode*), 2007
load_package_tests() (在 test.support 模块中), 1716
LOAD_SUPER_ATTR (*opcode*), 2008
load_verify_locations() (ssl.SSLContext 方法), 1118
loader -- 加载器, **2071**
loader_state (importlib.machinery.ModuleSpec 属性), 1922
LoadError, 1402
Loader (importlib.abc 中的类), 1912
loader (importlib.machinery.ModuleSpec 属性), 1922
LoadFileDialog (tkinter.filedialog 中的类), 1517
LoadKey() (在 winreg 模块中), 2024

LoadLibrary() (ctypes.LibraryLoader 方法), 772
loads() (在 json 模块中), 1216
loads() (在 marshal 模块中), 488
loads() (在 pickle 模块中), 471
loads() (在 plistlib 模块中), 596
loads() (在 tomllib 模块中), 593
loads() (在 xmlrpc.client 模块中), 1416
loadTestsFromModule() (unittest.TestLoader 方法), 1642
loadTestsFromName() (unittest.TestLoader 方法), 1642
loadTestsFromNames() (unittest.TestLoader 方法), 1643
loadTestsFromTestCase() (unittest.TestLoader 方法), 1642
Load (ast 中的类), 1948
LOCAL_CREDS() (在 socket 模块中), 1083
LOCAL_CREDS_PERSISTENT() (在 socket 模块中), 1083
localcontext() (在 decimal 模块中), 336
locale
    module, 1448
--locale
    calendar 命令行选项, 237
locale encoding -- 语言区域编码格式, **2071**
LOCALE() (在 re 模块中), 129
localeconv() (在 locale 模块中), 1448
LocaleHTMLCalendar (calendar 中的类), 234
LocaleTextCalendar (calendar 中的类), 234
localize() (在 locale 模块中), 1453
localName (xml.dom.Attr 属性), 1279
localName (xml.dom.Node 属性), 1276
--locals
    unittest 命令行选项, 1625
locals()
    built-in function, 20
localtime() (在 email.utils 模块中), 1208
localtime() (在 time 模块中), 692
local (threading 中的类), 888
Locator (xml.sax.xmlreader 中的类), 1296
lock() (mailbox.Babyl 方法), 1229
lock() (mailbox.Mailbox 方法), 1224
lock() (mailbox.Maildir 方法), 1226
lock() (mailbox.mbox 方法), 1227
lock() (mailbox.MH 方法), 1228
lock() (mailbox.MMDF 方法), 1230
Lock() (multiprocessing.managers.SyncManager 方法), 920
LOCK_EX() (在 fcntl 模块中), 2041
LOCK_NB() (在 fcntl 模块中), 2041
LOCK_SH() (在 fcntl 模块中), 2041
LOCK_UN() (在 fcntl 模块中), 2041
locked() (_thread.lock 方法), 980
locked() (asyncio.Condition 方法), 1012
locked() (asyncio.Lock 方法), 1010
locked() (asyncio.Semaphore 方法), 1013
locked() (threading.Lock 方法), 891
lockf() (在 fcntl 模块中), 2041
lockf() (在 os 模块中), 627
locking() (在 msvcrt 模块中), 2019
LockType() (在 _thread 模块中), 978
Lock (asyncio 中的类), 1010
Lock (multiprocessing 中的类), 914
lock (sys.thread_info 属性), 1809
Lock (threading 中的类), 890
log() (logging.Logger 方法), 706
log() (在 cmath 模块中), 322
log() (在 logging 模块中), 714
log() (在 math 模块中), 317
log1p() (在 math 模块中), 317
log2() (在 math 模块中), 317
log10() (decimal.Context 方法), 339
log10() (decimal.Decimal 方法), 332
log10() (在 cmath 模块中), 322
log10() (在 math 模块中), 317
LOG_ALERT() (在 syslog 模块中), 2047
LOG_AUTH() (在 syslog 模块中), 2047
LOG_AUTHPRIV() (在 syslog 模块中), 2047
LOG_CONS() (在 syslog 模块中), 2047
LOG_CRIT() (在 syslog 模块中), 2047
LOG_CRON() (在 syslog 模块中), 2047
LOG_DAEMON() (在 syslog 模块中), 2047
log_date_time_string() (http.server.BaseHTTPRequestHandler 方法), 1395
LOG_DEBUG() (在 syslog 模块中), 2047
LOG_EMERG() (在 syslog 模块中), 2047
LOG_ERR() (在 syslog 模块中), 2047
log_error() (http.server.BaseHTTPRequestHandler 方法), 1395
log_exception() (wsgiref.handlers.BaseHandler 方法), 1318
LOG_FTP() (在 syslog 模块中), 2047
LOG_INFO() (在 syslog 模块中), 2047
LOG_INSTALL() (在 syslog 模块中), 2047
LOG_KERN() (在 syslog 模块中), 2047
LOG_LAUNCHD() (在 syslog 模块中), 2047
LOG_LOCAL0() (在 syslog 模块中), 2047
LOG_LOCAL1() (在 syslog 模块中), 2047
LOG_LOCAL2() (在 syslog 模块中), 2047
LOG_LOCAL3() (在 syslog 模块中), 2047
LOG_LOCAL4() (在 syslog 模块中), 2047
LOG_LOCAL5() (在 syslog 模块中), 2047
LOG_LOCAL6() (在 syslog 模块中), 2047
LOG_LOCAL7() (在 syslog 模块中), 2047
LOG_LPR() (在 syslog 模块中), 2047
LOG_MAIL() (在 syslog 模块中), 2047
log_message() (http.server.BaseHTTPRequestHandler 方法), 1395
LOG_NDELAY() (在 syslog 模块中), 2047
LOG_NETINFO() (在 syslog 模块中), 2047
LOG_NEWS() (在 syslog 模块中), 2047
LOG_NOTICE() (在 syslog 模块中), 2047
LOG_NOWAIT() (在 syslog 模块中), 2047
LOG_ODELAY() (在 syslog 模块中), 2047

LOG_PERROR() (在 syslog 模块中), 2047
LOG_PID() (在 syslog 模块中), 2047
LOG_RAS() (在 syslog 模块中), 2047
LOG_REMOTEAUTH() (在 syslog 模块中), 2047
log_request() (http.server.BaseHTTPRequestHandler 方法), 1395
LOG_SYSLOG() (在 syslog 模块中), 2047
log_to_stderr() (在 multiprocessing 模块中), 930
LOG_USER() (在 syslog 模块中), 2047
LOG_UUCP() (在 syslog 模块中), 2047
LOG_WARNING() (在 syslog 模块中), 2047
logb() (decimal.Context 方法), 339
logb() (decimal.Decimal 方法), 332
LoggerAdapter (logging 中的类), 712
Logger (logging 中的类), 702
logging
    module, 701
    错误, 701
logging.config
    module, 718
logging.handlers
    module, 729
logical_and() (decimal.Context 方法), 339
logical_and() (decimal.Decimal 方法), 332
logical_invert() (decimal.Context 方法), 339
logical_invert() (decimal.Decimal 方法), 332
logical_or() (decimal.Context 方法), 339
logical_or() (decimal.Decimal 方法), 332
logical_xor() (decimal.Context 方法), 339
logical_xor() (decimal.Decimal 方法), 332
login() (ftplib.FTP 方法), 1360
login() (imaplib.IMAP4 方法), 1370
login() (smtplib.SMTP 方法), 1377
login_cram_md5() (imaplib.IMAP4 方法), 1370
login_tty() (在 os 模块中), 627
LOGNAME, 619, 851
lognormvariate() (在 random 模块中), 358
logout() (imaplib.IMAP4 方法), 1371
LogRecord (logging 中的类), 710
LONG_TIMEOUT() (在 test.support 模块中), 1711
longMessage (unittest.TestCase 属性), 1638
longname() (在 curses 模块中), 857
lookup() (symtable.SymbolTable 方法), 1977
lookup() (tkinter.ttk.Style 方法), 1534
lookup() (在 codecs 模块中), 174
lookup() (在 unicodedata 模块中), 158
lookup_error() (在 codecs 模块中), 177
LookupError, 100
loop_factory(unittest.IsolatedAsyncioTestCase 属性), 1640
LOOPBACK_TIMEOUT() (在 test.support 模块中), 1710
lower() (bytearray 方法), 70
lower() (bytes 方法), 70
lower() (str 方法), 52
LPAR() (在 token 模块中), 1981
lpAttributeList(subprocess.STARTUPINFO 属性), 962
lru_cache() (在 functools 模块中), 394
lseek() (在 os 模块中), 627
lshift() (在 operator 模块中), 404
LShift (ast 中的类), 1949
LSQB() (在 token 模块中), 1981
lstat() (pathlib.Path 方法), 425
lstat() (在 os 模块中), 641
lstrip() (bytearray 方法), 66
lstrip() (bytes 方法), 66
lstrip() (str 方法), 52
lsub() (imaplib.IMAP4 方法), 1371
lt() (在 operator 模块中), 402
lt() (在 turtle 模块中), 1464
LtE (ast 中的类), 1950
Lt (ast 中的类), 1950
LWPCookieJar (http.cookiejar 中的类), 1405
lzma
    module, 536
LZMACompressor (lzma 中的类), 537
LZMADecompressor (lzma 中的类), 538
LZMAError, 536
LZMAFile (lzma 中的类), 537

M

-m
    ast 命令行选项, 1975
    calendar 命令行选项, 238
    cProfile 命令行选项, 1747
    pdb 命令行选项, 1739
    pickletools 命令行选项, 2017
    trace 命令行选项, 1759
    zipapp 命令行选项, 1782
M() (在 re 模块中), 129
mac_ver() (在 platform 模块中), 744
machine() (在 platform 模块中), 742
macros (netrc.netrc 属性), 595
MADV_AUTOSYNC() (在 mmap 模块中), 1156
MADV_CORE() (在 mmap 模块中), 1156
MADV_DODUMP() (在 mmap 模块中), 1156
MADV_DOFORK() (在 mmap 模块中), 1156
MADV_DONTDUMP() (在 mmap 模块中), 1156
MADV_DONTFORK() (在 mmap 模块中), 1156
MADV_DONTNEED() (在 mmap 模块中), 1156
MADV_FREE() (在 mmap 模块中), 1156
MADV_FREE_REUSABLE() (在 mmap 模块中), 1156
MADV_FREE_REUSE() (在 mmap 模块中), 1156
MADV_HUGEPAGE() (在 mmap 模块中), 1156
MADV_HWPOISON() (在 mmap 模块中), 1156
MADV_MERGEABLE() (在 mmap 模块中), 1156
MADV_NOCORE() (在 mmap 模块中), 1156
MADV_NOHUGEPAGE() (在 mmap 模块中), 1156
MADV_NORMAL() (在 mmap 模块中), 1156
MADV_NOSYNC() (在 mmap 模块中), 1156

MADV_PROTECT() (在 mmap 模块中), 1156
MADV_RANDOM() (在 mmap 模块中), 1156
MADV_REMOVE() (在 mmap 模块中), 1156
MADV_SEQUENTIAL() (在 mmap 模块中), 1156
MADV_SOFT_OFFLINE() (在 mmap 模块中), 1156
MADV_UNMERGEABLE() (在 mmap 模块中), 1156
MADV_WILLNEED() (在 mmap 模块中), 1156
madvise() (mmap.mmap 方法), 1155
magic method -- 魔术方法, **2071**
MAGIC_NUMBER() (在 importlib.util 模块中), 1923
MagicMock (unittest.mock 中的类), 1679
mailbox
    module, 1221
Mailbox (mailbox 中的类), 1221
mailcap
    module, 2057
MaildirMessage (mailbox 中的类), 1230
Maildir (mailbox 中的类), 1224
--main
    zipapp 命令行选项, 1782
main() (在 site 模块中), 1895
main() (在 unittest 模块中), 1647
main_thread() (在 threading 模块中), 887
mainloop() (在 turtle 模块中), 1483
maintype(email.headerregistry.ContentTypeHeader 属性), 1183
major() (在 os 模块中), 643
major(email.headerregistry.MIMEVersionHeader 属性), 1183
make_alternative() (email.message.EmailMessage 方法), 1166
make_archive() (在 shutil 模块中), 463
make_bad_fd() (在 test.support.os_helper 模块中), 1722
MAKE_CELL (*opcode*), 2011
make_cookies() (http.cookiejar.CookieJar 方法), 1403
make_dataclass() (在 dataclasses 模块中), 1837
make_file() (difflib.HtmlDiff 方法), 143
MAKE_FUNCTION (*opcode*), 2012
make_header()(在 email.header 模块中), 1205
make_legacy_pyc() (在 test.support.import_helper 模块中), 1724
make_mixed() (email.message.EmailMessage 方法), 1166
make_msgid() (在 email.utils 模块中), 1209
make_parser() (在 xml.sax 模块中), 1289
make_pkg() (在 test.support.script_helper 模块中), 1720
make_related() (email.message.EmailMessage 方法), 1166
make_script()(在 test.support.script_helper 模块中), 1720
make_server()(在 wsgiref.simple_server 模块中), 1315
make_table() (difflib.HtmlDiff 方法), 144
make_zip_pkg() (在 test.support.script_helper 模块中), 1720
make_zip_script() (在 test.support.script_helper 模块中), 1720
makedev() (在 os 模块中), 643
makedirs() (在 os 模块中), 642
makeelement() (xml.etree.ElementTree.Element 方法), 1268
makefile() (socket.socket 方法), 1093
makeLogRecord() (在 logging 模块中), 715
makePickle() (logging.handlers.SocketHandler 方法), 734
makeRecord() (logging.Logger 方法), 706
makeSocket() (logging.handlers.DatagramHandler 方法), 735
makeSocket() (logging.handlers.SocketHandler 方法), 734
maketrans() (bytearray 静态方法), 64
maketrans() (bytes 静态方法), 64
maketrans() (str 静态方法), 53
MalformedHeaderDefect, 1180
manager (logging.LoggerAdapter 属性), 712
mangle_from_ (email.policy.Compat32 属性), 1178
mangle_from_ (email.policy.Policy 属性), 1175
MANPAGER, 1598
mant_dig (sys.float_info 属性), 1795
map()
    built-in function, 20
map() (concurrent.futures.Executor 方法), 945
map() (multiprocessing.pool.Pool 方法), 925
map() (tkinter.ttk.Style 方法), 1533
MAP_32BIT() (在 mmap 模块中), 1157
MAP_ADD (*opcode*), 2004
MAP_ALIGNED_SUPER() (在 mmap 模块中), 1157
MAP_ANON() (在 mmap 模块中), 1157
MAP_ANONYMOUS() (在 mmap 模块中), 1157
map_async() (multiprocessing.pool.Pool 方法), 925
MAP_CONCEAL() (在 mmap 模块中), 1157
MAP_DENYWRITE() (在 mmap 模块中), 1157
MAP_EXECUTABLE() (在 mmap 模块中), 1157
MAP_HASSEMAPHORE() (在 mmap 模块中), 1157
MAP_JIT() (在 mmap 模块中), 1157
MAP_NOCACHE() (在 mmap 模块中), 1157
MAP_NOEXTEND() (在 mmap 模块中), 1157
MAP_NORESERVE() (在 mmap 模块中), 1157
MAP_POPULATE() (在 mmap 模块中), 1157
MAP_PRIVATE() (在 mmap 模块中), 1157
MAP_RESILIENT_CODESIGN() (在 mmap 模块中), 1157

MAP_RESILIENT_MEDIA() (在 mmap 模块中), 1157
MAP_SHARED() (在 mmap 模块中), 1157
MAP_STACK() (在 mmap 模块中), 1157
map_table_b2() (在 stringprep 模块中), 160
map_table_b3() (在 stringprep 模块中), 160
map_to_type() (email.headerregistry.HeaderRegistry 方法), 1184
MAP_TPRO() (在 mmap 模块中), 1157
MAP_TRANSLATED_ALLOW_EXECUTE() (在 mmap 模块中), 1157
MAP_UNIX03() (在 mmap 模块中), 1157
mapLogRecord() (logging.handlers.HTTPHandler 方法), 739
mapping -- 映射
  object -- 对象, 83
  types, 运算目标, 83
mapping -- 映射, **2071**
MappingProxyType (types 中的类), 280
MappingView (collections.abc 中的类), 258
MappingView (typing 中的类), 1594
Mapping (collections.abc 中的类), 258
Mapping (typing 中的类), 1594
mapPriority() (logging.handlers.SysLogHandler 方法), 737
maps (collections.ChainMap 属性), 239
MARCH() (在 calendar 模块中), 236
markcoroutinefunction() (在 inspect 模块中), 1878
marshal
  module, 487
marshalling
  objects, 469
master (tkinter.Tk 属性), 1503
match() (pathlib.PurePath 方法), 419
match() (re.Pattern 方法), 134
match() (在 re 模块中), 131
match_case (ast 中的类), 1961
MATCH_CLASS (*opcode*), 2013
MATCH_KEYS (*opcode*), 2006
MATCH_MAPPING (*opcode*), 2006
MATCH_SEQUENCE (*opcode*), 2006
match_value() (test.support.Matcher 方法), 1718
MatchAs (ast 中的类), 1965
MatchClass (ast 中的类), 1964
Matcher (test.support 中的类), 1718
matches() (test.support.Matcher 方法), 1718
MatchMapping (ast 中的类), 1963
MatchOr (ast 中的类), 1966
MatchSequence (ast 中的类), 1962
MatchSingleton (ast 中的类), 1962
MatchStar (ast 中的类), 1963
MatchValue (ast 中的类), 1962
Match (ast 中的类), 1961
Match (re 中的类), 135
Match (typing 中的类), 1593
math
  module, 37, 312, 324
matmul() (在 operator 模块中), 404
MatMult (ast 中的类), 1949
max
  内置函数, 43
max() (decimal.Context 方法), 339
max() (decimal.Decimal 方法), 332
max_10_exp (sys.float_info 属性), 1795
max_count (email.headerregistry.BaseHeader 属性), 1181
MAX_EMAX() (在 decimal 模块中), 341
max_exp (sys.float_info 属性), 1795
MAX_INTERPOLATION_DEPTH() (在 configparser 模块中), 591
max_line_length (email.policy.Policy 属性), 1174
max_lines (textwrap.TextWrapper 属性), 157
max_mag() (decimal.Context 方法), 339
max_mag() (decimal.Decimal 方法), 333
max_memuse() (在 test.support 模块中), 1711
MAX_PREC() (在 decimal 模块中), 341
max_prefixlen (ipaddress.IPv4Address 属性), 1423
max_prefixlen (ipaddress.IPv4Network 属性), 1428
max_prefixlen (ipaddress.IPv6Address 属性), 1426
max_prefixlen (ipaddress.IPv6Network 属性), 1431
MAX_Py_ssize_t() (在 test.support 模块中), 1711
maxarray (reprlib.Repr 属性), 289
maxdeque (reprlib.Repr 属性), 289
maxdict (reprlib.Repr 属性), 289
maxDiff (unittest.TestCase 属性), 1638
maxfrozenset (reprlib.Repr 属性), 289
MAXIMUM_SUPPORTED (ssl.TLSVersion 属性), 1113
maximum_version (ssl.SSLContext 属性), 1123
maxlen (collections.deque 属性), 245
maxlevel (reprlib.Repr 属性), 289
maxlist (reprlib.Repr 属性), 289
maxlong (reprlib.Repr 属性), 289
maxother (reprlib.Repr 属性), 289
maxset (reprlib.Repr 属性), 289
maxsize() (在 sys 模块中), 1801
maxsize (asyncio.Queue 属性), 1019
maxstring (reprlib.Repr 属性), 289
maxtuple (reprlib.Repr 属性), 289
maxunicode() (在 sys 模块中), 1801
MAXYEAR() (在 datetime 模块中), 192
max (datetime.date 属性), 197
max (datetime.datetime 属性), 204
max (datetime.time 属性), 212
max (datetime.timedelta 属性), 194
max (sys.float_info 属性), 1795

MAY() (在 calendar 模块中), 236
MB_ICONASTERISK() (在 winsound 模块中), 2031
MB_ICONEXCLAMATION() (在 winsound 模块中), 2031
MB_ICONHAND() (在 winsound 模块中), 2031
MB_ICONQUESTION() (在 winsound 模块中), 2031
MB_OK() (在 winsound 模块中), 2031
mboxMessage (mailbox 中的类), 1232
mbox (mailbox 中的类), 1227
md5() (在 hashlib 模块中), 600
mean() (在 statistics 模块中), 364
mean (statistics.NormalDist 属性), 373
measure() (tkinter.font.Font 方法), 1514
median() (在 statistics 模块中), 367
median_grouped() (在 statistics 模块中), 368
median_high() (在 statistics 模块中), 368
median_low() (在 statistics 模块中), 368
median (statistics.NormalDist 属性), 373
member() (在 enum 模块中), 305
member_names (enum.EnumDict 属性), 303
MemberDescriptorType() (在 types 模块中), 280
memfd_create() (在 os 模块中), 655
memmove() (在 ctypes 模块中), 777
--memo
    pickletools 命令行选项, 2017
MemoryBIO (ssl 中的类), 1131
MemoryError, 102
MemoryHandler (logging.handlers 中的类), 738
memoryview
    object -- 对象, 60
memoryview (内置类), 75
memset() (在 ctypes 模块中), 778
merge() (在 heapq 模块中), 260
message_factory (email.policy.Policy 属性), 1175
message_from_binary_file() (在 email 模块中), 1169
message_from_bytes() (在 email 模块中), 1169
message_from_file() (在 email 模块中), 1170
message_from_string() (在 email 模块中), 1170
MessageBeep() (在 winsound 模块中), 2030
MessageClass (http.server.BaseHTTPRequestHandler 属性), 1394
MessageDefect, 1179
MessageError, 1179
MessageParseError, 1179
messages() (在 xml.parsers.expat.errors 模块中), 1306
message (BaseExceptionGroup 属性), 108
Message (email.message 中的类), 1194
Message (mailbox 中的类), 1230
Message (tkinter.messagebox 中的类), 1517
meta path finder -- 元路径查找器, **2071**
meta() (在 curses 模块中), 857
meta_path() (在 sys 模块中), 1801
metaclass -- 元类, **2071**
metadata() (在 importlib.metadata 模块中), 1935
--metadata-encoding
    zipfile 命令行选项, 551
MetaPathFinder (importlib.abc 中的类), 1912
MetavarTypeHelpFormatter (argparse 中的类), 788
metavar (optparse.Option 属性), 839
method -- 方法
    object -- 对象, 95
    特殊, 2076
    魔术, 2071
method -- 方法, **2072**
method resolution order -- 方法解析顺序, **2072**
method_calls (unittest.mock.Mock 属性), 1659
methodcaller() (在 operator 模块中), 406
MethodDescriptorType() (在 types 模块中), 278
methodHelp() (xmlrpc.client.ServerProxy.system 方法), 1412
methodSignature() (xmlrpc.client.ServerProxy.system 方法), 1411
methods (pyclbr.Class 属性), 1990
MethodType() (在 types 模块中), 278
MethodWrapperType() (在 types 模块中), 278
method (urllib.request.Request 属性), 1327
metrics() (tkinter.font.Font 方法), 1514
MFD_ALLOW_SEALING() (在 os 模块中), 655
MFD_CLOEXEC() (在 os 模块中), 655
MFD_HUGE_1GB() (在 os 模块中), 655
MFD_HUGE_1MB() (在 os 模块中), 655
MFD_HUGE_2GB() (在 os 模块中), 655
MFD_HUGE_2MB() (在 os 模块中), 655
MFD_HUGE_8MB() (在 os 模块中), 655
MFD_HUGE_16GB() (在 os 模块中), 655
MFD_HUGE_16MB() (在 os 模块中), 655
MFD_HUGE_32MB() (在 os 模块中), 655
MFD_HUGE_64KB() (在 os 模块中), 655
MFD_HUGE_256MB() (在 os 模块中), 655
MFD_HUGE_512KB() (在 os 模块中), 655
MFD_HUGE_512MB() (在 os 模块中), 655
MFD_HUGE_MASK() (在 os 模块中), 655
MFD_HUGE_SHIFT() (在 os 模块中), 655
MFD_HUGETLB() (在 os 模块中), 655
MHMessage (mailbox 中的类), 1234
MH (mailbox 中的类), 1227
microseconds (datetime.timedelta 属性), 195
microsecond (datetime.datetime 属性), 205
microsecond (datetime.time 属性), 212
MIME
    base64 编码格式, 1241
    content type, 1239
    quoted-printable 编码格式, 1246
    标头, 1239

MIMEApplication(email.mime.application 中的类), 1202
MIMEAudio (email.mime.audio 中的类), 1202
MIMEBase (email.mime.base 中的类), 1201
MIMEImage (email.mime.image 中的类), 1203
MIMEMessage (email.mime.message 中的类), 1203
MIMEMultipart(email.mime.multipart 中的类), 1202
MIMENonMultipart (email.mime.nonmultipart 中的类), 1202
MIMEPart (email.message 中的类), 1167
MIMEText (email.mime.text 中的类), 1203
mimetypes
    module, 1239
MimeTypes (mimetypes 中的类), 1240
MIMEVersionHeader(email.headerregistry 中的类), 1182
min
    内置函数, 43
min()
    built-in function, 21
min() (decimal.Context 方法), 340
min() (decimal.Decimal 方法), 333
min_10_exp (sys.float_info 属性), 1795
MIN_EMIN() (在 decimal 模块中), 341
MIN_ETINY() (在 decimal 模块中), 341
min_exp (sys.float_info 属性), 1795
min_mag() (decimal.Context 方法), 340
min_mag() (decimal.Decimal 方法), 333
MINEQUAL() (在 token 模块中), 1983
MINIMUM_SUPPORTED (ssl.TLSVersion 属性), 1113
minimum_version(ssl.SSLContext 属性), 1123
minor() (在 os 模块中), 643
minor(email.headerregistry.MIMEVersionHeader 属性), 1183
minus() (decimal.Context 方法), 340
MINUS() (在 token 模块中), 1982
minute (datetime.datetime 属性), 205
minute (datetime.time 属性), 212
MINYEAR() (在 datetime 模块中), 192
min (datetime.date 属性), 197
min (datetime.datetime 属性), 204
min (datetime.time 属性), 212
min (datetime.timedelta 属性), 194
min (sys.float_info 属性), 1795
mirrored() (在 unicodedata 模块中), 158
misc_header (cmd.Cmd 属性), 1493
MisplacedEnvelopeHeaderDefect, 1180
--missing
    trace 命令行选项, 1759
MISSING() (在 dataclasses 模块中), 1838
MISSING() (在 sys.monitoring 模块中), 1814
MISSING_C_DOCSTRINGS() (在 test.support 模块中), 1711
missing_compiler_executable() (在 test.support 模块中), 1717
MissingHeaderBodySeparatorDefect, 1180
MissingSectionHeaderError, 592
MISSING (contextvars.Token 属性), 975
mkd() (ftplib.FTP 方法), 1363
mkdir() (pathlib.Path 方法), 430
mkdir() (zipfile.ZipFile 方法), 546
mkdir() (在 os 模块中), 641
mkdtemp() (在 tempfile 模块中), 451
mkfifo() (在 os 模块中), 642
mknod() (在 os 模块中), 642
mkstemp() (在 tempfile 模块中), 450
mktemp() (在 tempfile 模块中), 453
mktime() (在 time 模块中), 692
mktime_tz() (在 email.utils 模块中), 1210
mlsd() (ftplib.FTP 方法), 1362
mmap
    module, 1152
mmap (mmap 中的类), 1153
MMDFMessage (mailbox 中的类), 1236
MMDF (mailbox 中的类), 1229
mock_add_spec() (unittest.mock.Mock 方法), 1655
mock_calls (unittest.mock.Mock 属性), 1659
mock_open() (在 unittest.mock 模块中), 1684
Mock (unittest.mock 中的类), 1653
mod() (在 operator 模块中), 404
--mode
    ast 命令行选项, 1975
mode() (在 statistics 模块中), 368
mode() (在 turtle 模块中), 1484
mode (bz2.BZ2File 属性), 533
mode (gzip.GzipFile 属性), 530
mode (io.FileIO 属性), 684
mode (lzma.LZMAFile 属性), 537
mode (statistics.NormalDist 属性), 373
mode (tarfile.TarInfo 属性), 559
modf() (在 math 模块中), 315
modified() (urllib.robotparser.RobotFileParser 方法), 1348
modify() (select.devpoll 方法), 1136
modify() (select.epoll 方法), 1137
modify() (selectors.BaseSelector 方法), 1142
modify() (select.poll 方法), 1138
module
    __future__, 1871
    __main__, 1821, 1907, 1908
    _locale, 1448
    _thread, 978
    _tkinter, 1504
    abc, 1855
    aifc, 2055
    argparse, 785
    array, 60, 266
    ast, 1941
    asynchat, 2055
    asyncio, 981
    asyncore, 2055

atexit, 1860
audioop, 2056
base64, 1241, 1244
bdb, 1731, 1738
binascii, 1244
bisect, 263
builtins, 31, 1820
bz2, 532
calendar, 230
cgi, 2056
cgitb, 2056
chunk, 2056
cmath, 321
cmd, 1491, 1738
code, 1897
codecs, 173
codeop, 1899
collections, 238
collections.abc, 254
colorsys, 1439
compileall, 1992
concurrent.futures, 944
configparser, 576
contextlib, 1842
contextvars, 974
copy, 282, 484
copyreg, 484
cProfile, 1749
crypt, 2056
csv, 569
ctypes, 754
curses, 854
curses.ascii, 879
curses.panel, 883
curses.textpad, 878
dataclasses, 1832
datetime, 191
dbm, 488
dbm.dumb, 493
dbm.gnu, 485, 490
dbm.ndbm, 485, 492
dbm.sqlite3, 490
decimal, 325
difflib, 143
dis, 1996
distutils, 2056
doctest, 1601
email, 1159
email.charset, 1206
email.contentmanager, 1185
email.encoders, 1208
email.errors, 1179
email.generator, 1170
email.header, 1204
email.headerregistry, 1180
email.iterators, 1211
email.message, 1160
email.mime, 1201
email.mime.application, 1202
email.mime.audio, 1202
email.mime.base, 1201
email.mime.image, 1203
email.mime.message, 1203
email.mime.multipart, 1202
email.mime.nonmultipart, 1202
email.mime.text, 1203
email.parser, 1167
email.policy, 1173
email.utils, 1208
encodings.idna, 188
encodings.mbcs, 188
encodings.utf_8_sig, 189
ensurepip, 1771
enum, 291
errno, 102, 745
faulthandler, 1736
fcntl, 2039
filecmp, 446
fileinput, 851
fnmatch, 456
fractions, 351
ftplib, 1359
functools, 393
gc, 1872
getopt, 2051
getpass, 851
gettext, 1441
glob, 453, 456
graphlib, 305
grp, 2035
gzip, 528
hashlib, 599
heapq, 260
hmac, 610
html, 1249
html.entities, 1254
html.parser, 1249
http, 1348
http.client, 1352
http.cookiejar, 1401
http.cookies, 1398
http.server, 1392
idlelib, 1548
imaplib, 1368
imghdr, 2057
imp, 2057
importlib, 1909
importlib.abc, 1912
importlib.machinery, 1918
importlib.metadata, 1932
importlib.resources, 1928
importlib.resources.abc, 1931
importlib.util, 1923
inspect, 1875
io, 677
ipaddress, 1422

itertools, 377
json, 1212
json.tool, 1220
keyword, 1984
linecache, 457
locale, 1448
logging, 701
logging.config, 718
logging.handlers, 729
lzma, 536
mailbox, 1221
mailcap, 2057
marshal, 487
math, 37, 312, 324
mimetypes, 1239
mmap, 1152
modulefinder, 1906
msilib, 2057
msvcrt, 2019
multiprocessing, 898
multiprocessing.connection, 927
multiprocessing.dummy, 930
multiprocessing.managers, 918
multiprocessing.pool, 924
multiprocessing.shared_memory, 939
multiprocessing.sharedctypes, 916
netrc, 594
nis, 2057
nntplib, 2057
numbers, 309
operator, 402
optparse, 825
os, 615, 2033
os.path, 434
ossaudiodev, 2058
pathlib, 411
pdb, 1738
pickle, 282, 469, 484, 487
pickletools, 2017
pipes, 2058
pkgutil, 1903
platform, 742
plistlib, 595
poplib, 1365
posix, 2033
pprint, 283
profile, 1749
pstats, 1750
pty, 630, 2038
pwd, 436, 2034
py_compile, 1991
pyclbr, 1989
pydoc, 1597
pyexpat, 1300
queue, 971
quopri, 1246
random, 354
re, 49, 122, 456
readline, 161
reprlib, 288
resource, 2042
rlcompleter, 165
runpy, 1907
sched, 969
secrets, 611
select, 1134
selectors, 1140
shelve, 484, 487
shlex, 1496
shutil, 458
signal, 980, 1144
site, 1893
sitecustomize, 1894
smtpd, 2058
smtplib, 1374
sndhdr, 2058
socket, 1075, 1309
socketserver, 1384
spwd, 2058
sqlite3, 494
ssl, 1101
stat, 440, 647
statistics, 363
string, 111
stringprep, 159
struct, 167, 1097
subprocess, 951
sunau, 2058
symtable, 1976
sys, 23, 1787
sysconfig, 1815
syslog, 2046
sys.monitoring, 1810
tabnanny, 1988
tarfile, 552
telnetlib, 2059
tempfile, 448
termios, 2035
test, 1707
test.regrtest, 1709
test.support, 1710
test.support.bytecode_helper, 1720
test.support.import_helper, 1723
test.support.os_helper, 1721
test.support.script_helper, 1719
test.support.socket_helper, 1718
test.support.threading_helper, 1720
test.support.warnings_helper, 1724
textwrap, 154
threading, 885
time, 690
timeit, 1754
tkinter, 1501
tkinter.colorchooser, 1513
tkinter.commondialog, 1517
tkinter.dnd, 1520

tkinter.filedialog, 1515
tkinter.font, 1513
tkinter.messagebox, 1517
tkinter.scrolledtext, 1519
tkinter.simpledialog, 1514
tkinter.ttk, 1520
token, 1979
tokenize, 1985
tomllib, 592
trace, 1758
traceback, 1861
tracemalloc, 1760
tty, 2037
turtle, 1457
turtledemo, 1490
types, 95, 276
typing, 1549
unicodedata, 158
unittest, 1622
unittest.mock, 1650
urllib, 1321
urllib.error, 1347
urllib.parse, 1338
urllib.request, 1321, 1352
urllib.response, 1338
urllib.robotparser, 1347
usercustomize, 1894
uu, 2059
uuid, 1380
venv, 1773
warnings, 1825
wave, 1437
搜索 path, 457, 1802, 1893
weakref, 269
webbrowser, 1309
winreg, 2021
winsound, 2030
wsgiref, 1312
wsgiref.handlers, 1316
wsgiref.headers, 1314
wsgiref.simple_server, 1315
wsgiref.types, 1320
wsgiref.util, 1312
wsgiref.validate, 1316
xdrlib, 2059
xml, 1254
xml.dom, 1273
xml.dom.minidom, 1283
xml.dom.pulldom, 1286
xml.etree.ElementInclude, 1266
xml.etree.ElementTree, 1256
xml.parsers.expat, 1299
xml.parsers.expat.errors, 1306
xml.parsers.expat.model, 1305
xmlrpc, 1409
xmlrpc.client, 1409
xmlrpc.server, 1417
xml.sax, 1288
xml.sax.handler, 1290
xml.sax.saxutils, 1295
xml.sax.xmlreader, 1296
zipapp, 1782
zipfile, 541
zipimport, 1901
zlib, 525
zoneinfo, 226
module -- 模块, **2072**
module spec -- 模块规格, **2072**
module_from_spec() (在 importlib.util 模块中), 1924
modulefinder
module, 1906
ModuleFinder (modulefinder 中的类), 1906
ModuleInfo (pkgutil 中的类), 1903
ModuleNotFoundError, 101
modules() (在 sys 模块中), 1802
modules_cleanup() (在 test.support.import_helper 模块中), 1724
modules_setup() (在 test.support.import_helper 模块中), 1724
ModuleSpec (importlib.machinery 中的类), 1921
modules (modulefinder.ModuleFinder 属性), 1906
ModuleType (types 中的类), 278
Module (ast 中的类), 1945
module (pyclbr.Class 属性), 1990
module (pyclbr.Function 属性), 1990
MODULE (symtable.SymbolTableType 属性), 1976
modulus (sys.hash_info 属性), 1799
Mod (ast 中的类), 1949
MON_1() (在 locale 模块中), 1450
MON_2() (在 locale 模块中), 1450
MON_3() (在 locale 模块中), 1450
MON_4() (在 locale 模块中), 1450
MON_5() (在 locale 模块中), 1450
MON_6() (在 locale 模块中), 1450
MON_7() (在 locale 模块中), 1450
MON_8() (在 locale 模块中), 1450
MON_9() (在 locale 模块中), 1450
MON_10() (在 locale 模块中), 1450
MON_11() (在 locale 模块中), 1450
MON_12() (在 locale 模块中), 1450
MONDAY() (在 calendar 模块中), 235
monotonic() (在 time 模块中), 692
monotonic_ns() (在 time 模块中), 693
month
calendar 命令行选项, 237
month() (在 calendar 模块中), 234
month_abbr() (在 calendar 模块中), 235
month_name() (在 calendar 模块中), 235
monthcalendar() (在 calendar 模块中), 234

monthdatescalendar() (calendar.Calendar 方法), 231
monthdays2calendar() (calendar.Calendar 方法), 231
monthdayscalendar() (calendar.Calendar 方法), 232
monthrange() (在 calendar 模块中), 234
--months
    calendar 命令行选项, 238
Month (calendar 中的类), 236
month (calendar.IllegalMonthError 属性), 236
month (datetime.date 属性), 198
month (datetime.datetime 属性), 204
Morsel (http.cookies 中的类), 1399
most_common() (collections.Counter 方法), 241
mouseinterval() (在 curses 模块中), 857
mousemask() (在 curses 模块中), 857
move() (curses.panel.Panel 方法), 883
move() (curses.window 方法), 864
move() (mmap.mmap 方法), 1155
move() (tkinter.ttk.Treeview 方法), 1532
move() (在 shutil 模块中), 461
move_to_end() (collections.OrderedDict 方法), 251
MozillaCookieJar (http.cookiejar 中的类), 1405
MRO, **2072**
msg (http.client.HTTPResponse 属性), 1357
msg (json.JSONDecodeError 属性), 1218
msg (netrc.NetrcParseError 属性), 594
msg (re.PatternError 属性), 133
msg (traceback.TracebackException 属性), 1865
msilib
    module, 2057
msvcrt
    module, 2019
mtime() (urllib.robotparser.RobotFileParser 方法), 1348
mtime (gzip.GzipFile 属性), 530
mtime (tarfile.TarInfo 属性), 559
mul() (在 operator 模块中), 404
MultiCall (xmlrpc.client 中的类), 1415
MULTILINE() (在 re 模块中), 129
MultilineContinuationError, 592
MultiLoopChildWatcher(asyncio 中的类), 1063
multimode() (在 statistics 模块中), 369
MultipartConversionError, 1179
MultipartInvariantViolationDefect, 1180
multiply() (decimal.Context 方法), 340
multiprocessing
    module, 898
multiprocessing.connection
    module, 927
multiprocessing.dummy
    module, 930
multiprocessing.Manager()
    built-in function, 918
multiprocessing.managers
    module, 918
multiprocessing.pool
    module, 924
multiprocessing.shared_memory
    module, 939
multiprocessing.sharedctypes
    module, 916
Mult (ast 中的类), 1949
mutable -- 可变对象
    sequence types, 45
mutable -- 可变对象, **2072**
MutableMapping(collections.abc 中的类), 258
MutableMapping (typing 中的类), 1594
MutableSequence (collections.abc 中的类), 257
MutableSequence (typing 中的类), 1594
MutableSet (collections.abc 中的类), 257
MutableSet (typing 中的类), 1595
mvderwin() (curses.window 方法), 864
mvwin() (curses.window 方法), 864
myrights() (imaplib.IMAP4 方法), 1371

N

-N
    uuid 命令行选项, 1383
-n
    timeit 命令行选项, 1756
    uuid 命令行选项, 1383
    webbrowser 命令行选项, 1309
N_TOKENS() (在 token 模块中), 1984
n_waiting (asyncio.Barrier 属性), 1014
n_waiting (threading.Barrier 属性), 897
NAK() (在 curses.ascii 模块中), 881
--name
    uuid 命令行选项, 1383
name() (在 os 模块中), 616
NAME() (在 token 模块中), 1980
name() (在 unicodedata 模块中), 158
name2codepoint() (在 html.entities 模块中), 1254
named tuple -- 具名元组, **2072**
NAMED_FLAGS (enum.EnumCheck 属性), 301
NamedExpr (ast 中的类), 1951
NamedTemporaryFile() (在 tempfile 模块中), 449
namedtuple() (在 collections 模块中), 248
NamedTuple (typing 中的类), 1577
NameError, 102
namelist() (zipfile.ZipFile 方法), 544
nameprep() (在 encodings.idna 模块中), 188
namereplace
    错误处理器名称, 177
namereplace_errors()(在 codecs 模块中), 178
namer(logging.handlers.BaseRotatingHandler 属性), 731

names() (在 tkinter.font 模块中), 1514
--namespace
    uuid 命令行选项, 1383
namespace -- 命名空间, **2072**
namespace package -- 命名空间包, **2072**
namespace() (imaplib.IMAP4 方法), 1371
Namespace() (multiprocessing.managers.SyncManager 方法), 920
NAMESPACE_DNS() (在 uuid 模块中), 1382
NAMESPACE_OID() (在 uuid 模块中), 1382
NAMESPACE_URL() (在 uuid 模块中), 1382
NAMESPACE_X500() (在 uuid 模块中), 1382
NamespaceErr, 1281
NamespaceLoader (importlib.machinery 中的类), 1921
namespaceURI (xml.dom.Node 属性), 1276
Namespace (argparse 中的类), 804
Namespace(multiprocessing.managers 中的类), 920
nametofont() (在 tkinter.font 模块中), 1514
Name (ast 中的类), 1948
name (bz2.BZ2File 属性), 533
name (codecs.CodecInfo 属性), 174
name (contextvars.ContextVar 属性), 975
name (doctest.DocTest 属性), 1615
name (email.headerregistry.BaseHeader 属性), 1181
name (enum.Enum 属性), 295
name (gzip.GzipFile 属性), 530
name (hashlib.hash 属性), 601
name (hmac.HMAC 属性), 611
name (http.cookiejar.Cookie 属性), 1408
name (ImportError 属性), 101
name (importlib.abc.FileLoader 属性), 1915
name (importlib.abc.Traversable 属性), 1917
name(importlib.machinery.AppleFrameworkLoader 属性), 1923
name(importlib.machinery.ExtensionFileLoader 属性), 1921
name (importlib.machinery.ModuleSpec 属性), 1921
name(importlib.machinery.SourceFileLoader 属性), 1920
name(importlib.machinery.SourcelessFileLoader 属性), 1920
name (importlib.resources.abc.Traversable 属性), 1931
name (inspect.Parameter 属性), 1883
name (io.FileIO 属性), 684
name (logging.Logger 属性), 702
name (lzma.LZMAFile 属性), 537
name (multiprocessing.Process 属性), 905
name(multiprocessing.shared_memory.SharedMemory 属性), 940
name (os.DirEntry 属性), 646
name (pathlib.PurePath 属性), 417
name (pyclbr.Class 属性), 1990
name (pyclbr.Function 属性), 1990
name (sys.thread_info 属性), 1808
name (tarfile.TarInfo 属性), 559
name (tempfile.TemporaryDirectory 属性), 450
name (threading.Thread 属性), 889
name (traceback.FrameSummary 属性), 1867
name (webbrowser.controller 属性), 1311
name (xml.dom.Attr 属性), 1279
name (xml.dom.DocumentType 属性), 1277
name (zipfile.Path 属性), 547
NaN, 16
nan() (在 cmath 模块中), 324
nan() (在 math 模块中), 320
nanj() (在 cmath 模块中), 324
NannyNag, 1989
nan (sys.hash_info 属性), 1799
napms() (在 curses 模块中), 857
nargs (optparse.Option 属性), 838
native_id (threading.Thread 属性), 890
nbytes (memoryview 属性), 80
ncurses_version() (在 curses 模块中), 866
ndiff() (在 difflib 模块中), 145
ndim (memoryview 属性), 80
ND (inspect.BufferFlags 属性), 1892
ne() (在 operator 模块中), 402
needs_input (bz2.BZ2Decompressor 属性), 534
needs_input (lzma.LZMADecompressor 属性), 539
neg() (在 operator 模块中), 404
nested scope -- 嵌套作用域, **2073**
NetmaskValueError, 1435
netmask (ipaddress.IPv4Network 属性), 1428
netmask (ipaddress.IPv6Network 属性), 1431
netrc
    module, 594
NetrcParseError, 594
netrc (netrc 中的类), 594
netscape (http.cookiejar.CookiePolicy 属性), 1406
network_address (ipaddress.IPv4Network 属性), 1428
network_address (ipaddress.IPv6Network 属性), 1431
network (ipaddress.IPv4Interface 属性), 1433
network (ipaddress.IPv6Interface 属性), 1433
Never() (在 typing 模块中), 1561
NEVER_EQ() (在 test.support 模块中), 1711
new() (在 hashlib 模块中), 600
new() (在 hmac 模块中), 610
new-style class -- 新式类, **2073**
newchild() (collections.ChainMap 方法), 239
new_class() (在 types 模块中), 276
new_event_loop() (asyncio.AbstractEventLoopPolicy 方法), 1061
new_event_loop() (在 asyncio 模块中), 1023

new_panel() (在 curses.panel 模块中), 883
NEWLINE() (在 token 模块中), 1980
newlines (io.TextIOBase 属性), 686
newpad() (在 curses 模块中), 857
--new-tab
    webbrowser 命令行选项, 1309
NewType (typing 中的类), 1579
newwin() (在 curses 模块中), 857
--new-window
    webbrowser 命令行选项, 1309
next (*pdb command*), 1742
next()
    built-in function, 21
next() (tarfile.TarFile 方法), 557
next() (tkinter.ttk.Treeview 方法), 1532
next_minus() (decimal.Context 方法), 340
next_minus() (decimal.Decimal 方法), 333
next_plus() (decimal.Context 方法), 340
next_plus() (decimal.Decimal 方法), 333
next_toward() (decimal.Context 方法), 340
next_toward() (decimal.Decimal 方法), 333
nextafter() (在 math 模块中), 316
nextfile() (在 fileinput 模块中), 852
nextkey() (dbm.gnu.gdbm 方法), 491
nextSibling (xml.dom.Node 属性), 1276
ngettext() (gettext.GNUTranslations 方法), 1444
ngettext() (gettext.NullTranslations 方法), 1443
ngettext() (在 gettext 模块中), 1442
nice() (在 os 模块中), 664
nis
    module, 2057
nl() (在 curses 模块中), 857
NL() (在 curses.ascii 模块中), 880
NL() (在 token 模块中), 1980
nl_langinfo() (在 locale 模块中), 1449
nlargest() (在 heapq 模块中), 261
nlst() (ftplib.FTP 方法), 1362
nntplib
    module, 2057
NO() (在 tkinter.messagebox 模块中), 1519
no_cache() (zoneinfo.ZoneInfo 类方法), 228
NO_EVENTS (*monitoring event*), 1812
no_proxy, 1325
no_site (sys.flags 属性), 1793
no_tracing() (在 test.support 模块中), 1715
no_type_check() (在 typing 模块中), 1588
no_type_check_decorator() (在 typing 模块中), 1588
no_user_site (sys.flags 属性), 1793
NoBoundaryInMultipartDefect, 1179
nocbreak() (在 curses 模块中), 858
NoDataAllowedErr, 1281
node() (在 platform 模块中), 742
NoDefault() (在 typing 模块中), 1591
nodelay() (curses.window 方法), 865
nodeName (xml.dom.Node 属性), 1276
NodeTransformer (ast 中的类), 1974
nodeType (xml.dom.Node 属性), 1275
nodeValue (xml.dom.Node 属性), 1276
NodeVisitor (ast 中的类), 1973
node (uuid.UUID 属性), 1381
noecho() (在 curses 模块中), 858
--no-ensure-ascii
    json.tool 命令行选项, 1221
NOEXPR() (在 locale 模块中), 1451
NOFLAG() (在 re 模块中), 129
--no-indent
    json.tool 命令行选项, 1221
NoModificationAllowedErr, 1281
NonCallableMagicMock(unittest.mock 中的类), 1679
NonCallableMock(unittest.mock 中的类), 1660
None (内置对象), 35
NoneType() (在 types 模块中), 277
None (内置变量), 33
nonl() (在 curses 模块中), 858
Nonlocal (ast 中的类), 1969
nonmember() (在 enum 模块中), 305
noop() (imaplib.IMAP4 方法), 1371
noop() (poplib.POP3 方法), 1367
NoOptionError, 592
NOP (*opcode*), 2001
noqiflush() (在 curses 模块中), 858
noraw() (在 curses 模块中), 858
--no-report
    trace 命令行选项, 1759
NoReturn() (在 typing 模块中), 1561
NORMAL() (在 tkinter.font 模块中), 1513
NORMAL_PRIORITY_CLASS() (在 subprocess 模块中), 963
NormalDist (statistics 中的类), 373
normalize() (decimal.Context 方法), 340
normalize() (decimal.Decimal 方法), 333
normalize() (xml.dom.Node 方法), 1277
normalize() (在 locale 模块中), 1452
normalize() (在 unicodedata 模块中), 158
NORMALIZE_WHITESPACE() (在 doctest 模块中), 1608
normalvariate() (在 random 模块中), 358
normcase() (在 os.path 模块中), 438
normpath() (在 os.path 模块中), 438
NoSectionError, 592
NoSuchMailboxError, 1237
not
    operator, 36
not in
    operator, 36, 43
not_() (在 operator 模块中), 403
NotADirectoryError, 106
notationDecl() (xml.sax.handler.DTDHandler 方法), 1294
NotationDeclHandler()
    (xml.parsers.expat.xmlparser 方法), 1303

notations (xml.dom.DocumentType 属性), 1278
NotConnected, 1353
Notebook (tkinter.ttk 中的类), 1526
NotEmptyError, 1237
NOTEQUAL() (在 token 模块中), 1982
NotEq (ast 中的类), 1950
NotFoundErr, 1281
notify() (asyncio.Condition 方法), 1012
notify() (threading.Condition 方法), 894
notify_all() (asyncio.Condition 方法), 1012
notify_all() (threading.Condition 方法), 894
notimeout() (curses.window 方法), 865
NotImplementedError, 102
NotImplementedType() (在 types 模块中), 278
NotImplemented (内置变量), 33
NotIn (ast 中的类), 1950
NotRequired() (在 typing 模块中), 1566
NOTSET() (在 logging 模块中), 707
NotStandaloneHandler() (xml.parsers.expat.xmlparser 方法), 1304
NotSupportedErr, 1281
NotSupportedError, 514
--no-type-comments
    ast 命令行选项, 1976
Not (ast 中的类), 1949
noutrefresh() (curses.window 方法), 865
NOVEMBER() (在 calendar 模块中), 236
now() (datetime.datetime 类方法), 202
npgettext() (gettext.GNUTranslations 方法), 1445
npgettext() (gettext.NullTranslations 方法), 1443
npgettext() (在 gettext 模块中), 1442
NSIG() (在 signal 模块中), 1146
nsmallest() (在 heapq 模块中), 261
NTEventLogHandler(logging.handlers 中的类), 737
ntohl() (在 socket 模块中), 1088
ntohs() (在 socket 模块中), 1088
ntransfercmd() (ftplib.FTP 方法), 1362
NUL() (在 curses.ascii 模块中), 879
nullcontext() (在 contextlib 模块中), 1845
NullHandler (logging 中的类), 730
NullTranslations (gettext 中的类), 1443
num_addresses (ipaddress.IPv4Network 属性), 1429
num_addresses (ipaddress.IPv6Network 属性), 1431
num_tickets (ssl.SSLContext 属性), 1123
--number
    timeit 命令行选项, 1756
NUMBER() (在 token 模块中), 1980
number_class() (decimal.Context 方法), 340
number_class() (decimal.Decimal 方法), 333
numbers
    module, 309
Number (numbers 中的类), 309
numerator (fractions.Fraction 属性), 352
numerator (numbers.Rational 属性), 310
numeric() (在 unicodedata 模块中), 158
numinput() (在 turtle 模块中), 1483

O

-O
    dis 命令行选项, 1997
-o
    compileall 命令行选项, 1993
    cProfile 命令行选项, 1747
    doctest 命令行选项, 1604
    pickletools 命令行选项, 2017
    zipapp 命令行选项, 1782
O_APPEND() (在 os 模块中), 629
O_ASYNC() (在 os 模块中), 629
O_BINARY() (在 os 模块中), 629
O_CLOEXEC() (在 os 模块中), 629
O_CREAT() (在 os 模块中), 629
O_DIRECT() (在 os 模块中), 629
O_DIRECTORY() (在 os 模块中), 629
O_DSYNC() (在 os 模块中), 629
O_EVTONLY() (在 os 模块中), 629
O_EXCL() (在 os 模块中), 629
O_EXLOCK() (在 os 模块中), 629
O_FSYNC() (在 os 模块中), 629
O_NDELAY() (在 os 模块中), 629
O_NOATIME() (在 os 模块中), 629
O_NOCTTY() (在 os 模块中), 629
O_NOFOLLOW() (在 os 模块中), 629
O_NOFOLLOW_ANY() (在 os 模块中), 629
O_NOINHERIT() (在 os 模块中), 629
O_NONBLOCK() (在 os 模块中), 629
O_PATH() (在 os 模块中), 629
O_RANDOM() (在 os 模块中), 629
O_RDONLY() (在 os 模块中), 629
O_RDWR() (在 os 模块中), 629
O_RSYNC() (在 os 模块中), 629
O_SEQUENTIAL() (在 os 模块中), 629
O_SHLOCK() (在 os 模块中), 629
O_SHORT_LIVED() (在 os 模块中), 629
O_SYMLINK() (在 os 模块中), 629
O_SYNC() (在 os 模块中), 629
O_TEMPORARY() (在 os 模块中), 629
O_TEXT() (在 os 模块中), 629
O_TMPFILE() (在 os 模块中), 629
O_TRUNC() (在 os 模块中), 629
O_WRONLY() (在 os 模块中), 629
object -- 对象
    bytearray, 45, 60, 61
    code -- 代码, 95, 487
    complex number -- 复数, 36
    dictionary -- 字典, 83
    GenericAlias, 89
    integer, 36
    io.StringIO, 48

    list, 45
    mapping -- 映射, 83
    memoryview, 60
    method -- 方法, 95
    range, 47
    sequence, 43
    set, 81
    socket, 1076
    string, 48
    traceback -- 回溯, 1792, 1861
    type, 29
    Union, 92
    元组, 45, 46
    字节串, 60
    布尔值, 36
    数字, 36
    浮点数, 36
object -- 对象, **2073**
objects
    flattening, 469
    marshalling, 469
    persistent, 469
    pickling, 469
    serializing, 469
    比较, 36
object (UnicodeError 属性), 105
object (内置类), 21
obj (memoryview 属性), 79
oct()
    built-in function, 21
octdigits() (在 string 模块中), 111
OCTOBER() (在 calendar 模块中), 236
offset_data (tarfile.TarInfo 属性), 560
offset (SyntaxError 属性), 104
offset (tarfile.TarInfo 属性), 560
offset (traceback.TracebackException 属性), 1865
offset (xml.parsers.expat.ExpatError 属性), 1305
OK() (在 curses 模块中), 866
OK() (在 tkinter.messagebox 模块中), 1519
ok_command() (tkinter.filedialog.LoadFileDialog 方法), 1517
ok_command() (tkinter.filedialog.SaveFileDialog 方法), 1517
ok_event() (tkinter.filedialog.FileDialog 方法), 1516
OKCANCEL() (在 tkinter.messagebox 模块中), 1519
old_value (contextvars.Token 属性), 975
OleDLL (ctypes 中的类), 771
on_motion() (tkinter.dnd.DndHandler 方法), 1520
on_release() (tkinter.dnd.DndHandler 方法), 1520
onclick() (在 turtle 模块中), 1482
ondrag() (在 turtle 模块中), 1477
onecmd() (cmd.Cmd 方法), 1492
onkey() (在 turtle 模块中), 1482
onkeypress() (在 turtle 模块中), 1482
onkeyrelease() (在 turtle 模块中), 1482
onrelease() (在 turtle 模块中), 1477
onscreenclick() (在 turtle 模块中), 1482
ontimer() (在 turtle 模块中), 1483
OP() (在 token 模块中), 1980
OP_ALL() (在 ssl 模块中), 1109
OP_CIPHER_SERVER_PREFERENCE() (在 ssl 模块中), 1110
OP_ENABLE_KTLS() (在 ssl 模块中), 1110
OP_ENABLE_MIDDLEBOX_COMPAT() (在 ssl 模块中), 1110
OP_IGNORE_UNEXPECTED_EOF() (在 ssl 模块中), 1110
OP_LEGACY_SERVER_CONNECT() (在 ssl 模块中), 1110
OP_NO_COMPRESSION() (在 ssl 模块中), 1110
OP_NO_RENEGOTIATION() (在 ssl 模块中), 1109
OP_NO_SSLv2() (在 ssl 模块中), 1109
OP_NO_SSLv3() (在 ssl 模块中), 1109
OP_NO_TICKET() (在 ssl 模块中), 1110
OP_NO_TLSv1() (在 ssl 模块中), 1109
OP_NO_TLSv1_1() (在 ssl 模块中), 1109
OP_NO_TLSv1_2() (在 ssl 模块中), 1109
OP_NO_TLSv1_3() (在 ssl 模块中), 1109
OP_SINGLE_DH_USE() (在 ssl 模块中), 1110
OP_SINGLE_ECDH_USE() (在 ssl 模块中), 1110
open()
    built-in function, 22
open() (imaplib.IMAP4 方法), 1371
open() (importlib.abc.Traversable 方法), 1917
open() (importlib.resources.abc.Traversable 方法), 1932
open() (pathlib.Path 方法), 426
open() (urllib.request.OpenerDirector 方法), 1328
open() (urllib.request.URLopener 方法), 1336
open() (webbrowser.controller 方法), 1311
open() (zipfile.Path 方法), 547
open() (zipfile.ZipFile 方法), 544
open() (tarfile.TarFile 类方法), 556
open() (在 bz2 模块中), 532
open() (在 codecs 模块中), 175
open() (在 dbm 模块中), 489
open() (在 dbm.dumb 模块中), 493
open() (在 dbm.gnu 模块中), 491
open() (在 dbm.ndbm 模块中), 492
open() (在 dbm.sqlite3 模块中), 490
open() (在 gzip 模块中), 528
open() (在 io 模块中), 679
open() (在 lzma 模块中), 536
open() (在 os 模块中), 628
open() (在 shelve 模块中), 484
open() (在 tarfile 模块中), 552
open() (在 tokenize 模块中), 1986

open() (在 wave 模块中), 1437
open() (在 webbrowser 模块中), 1310
open_binary() (在 importlib.resources 模块中), 1929
open_code() (在 io 模块中), 679
open_connection() (在 asyncio 模块中), 1003
open_flags() (在 dbm.gnu 模块中), 491
open_new() (webbrowser.controller 方法), 1312
open_new() (在 webbrowser 模块中), 1310
open_new_tab() (webbrowser.controller 方法), 1312
open_new_tab() (在 webbrowser 模块中), 1310
open_osfhandle() (在 msvcrt 模块中), 2020
open_resource() (importlib.abc.ResourceReader 方法), 1916
open_resource() (importlib.resources.abc.ResourceReader 方法), 1931
open_text() (在 importlib.resources 模块中), 1929
open_unix_connection() (在 asyncio 模块中), 1004
open_unknown() (urllib.request.URLopener 方法), 1336
open_urlresource() (在 test.support 模块中), 1716
OpenerDirector (urllib.request 中的类), 1324
OpenKey() (在 winreg 模块中), 2024
OpenKeyEx() (在 winreg 模块中), 2024
openlog() (在 syslog 模块中), 2046
openpty() (在 os 模块中), 630
openpty() (在 pty 模块中), 2038
OpenSSL
    (在 hashlib 模块中使用), 599
    (在 ssl 模块中使用), 1101
OPENSSL_VERSION() (在 ssl 模块中), 1112
OPENSSL_VERSION_INFO() (在 ssl 模块中), 1112
OPENSSL_VERSION_NUMBER() (在 ssl 模块中), 1112
Open (tkinter.filedialog 中的类), 1516
operation
    slice -- 切片, 43
    下标, 43
    拼接, 43
    重复, 43
OperationalError, 514
operator
    - (减号), 36
    % (百分号), 36
    & (和号), 37
    * (星号), 36
    ** , 36
    + (加号), 36
    / (斜杠), 36
    // , 36
    < (小于号), 36
    << , 37
    <= , 36
    != , 36
    == , 36
    > (大于号), 36
    >= , 36
    >> , 37
    ^ (脱字号), 37
    | (竖线), 37
    ~ (波浪号), 37
    and, 35, 36
    in, 36, 43
    is, 36
    is not, 36
    module, 402
    not, 36
    not in, 36, 43
    or, 35, 36
    比较, 36
opmap() (在 dis 模块中), 2016
opname() (在 dis 模块中), 2016
optim_args_from_interpreter_flags() (在 test.support 模块中), 1713
optimize() (在 pickletools 模块中), 2018
optimized scope -- 已优化的作用域, **2073**
OPTIMIZED_BYTECODE_SUFFIXES() (在 importlib.machinery 模块中), 1918
optimize (sys.flags 属性), 1793
--option
    doctest 命令行选项, 1604
Optional() (在 typing 模块中), 1564
OptionConflictError, 850
OptionError, 850
OptionGroup (optparse 中的类), 833
OptionParser (optparse 中的类), 836
options() (configparser.ConfigParser 方法), 589
options (doctest.Example 属性), 1616
Options (ssl 中的类), 1110
options (ssl.SSLContext 属性), 1123
OptionValueError, 850
optionxform() (configparser.ConfigParser 方法), 591
Option (optparse 中的类), 838
optparse
    module, 825
or
    operator, 35, 36
or_() (在 operator 模块中), 404
ord()
    built-in function, 24
ordered_attributes (xml.parsers.expat.xmlparser 属性), 1302
OrderedDict (collections 中的类), 251
OrderedDict (typing 中的类), 1593
orig_argv() (在 sys 模块中), 1802
origin_req_host (urllib.request.Request 属性), 1326

origin_server(wsgiref.handlers.BaseHandler 属性), 1319
origin (importlib.machinery.ModuleSpec 属性), 1922
Or (ast 中的类), 1950
os
    module, 615, 2033
os_environ (wsgiref.handlers.BaseHandler 属性), 1318
OSError, 102
os.path
    module, 434
ossaudiodev
    module, 2058
OUT_TO_DEFAULT() (在 msvcrt 模块中), 2020
OUT_TO_MSGBOX() (在 msvcrt 模块中), 2021
OUT_TO_STDERR() (在 msvcrt 模块中), 2020
outfile
    json.tool 命令行选项, 1220
--output
    pickletools 命令行选项, 2017
    zipapp 命令行选项, 1782
output() (http.cookies.BaseCookie 方法), 1399
output() (http.cookies.Morsel 方法), 1400
output_charset (email.charset.Charset 属性), 1206
output_codec (email.charset.Charset 属性), 1206
output_difference() (doctest.OutputChecker 方法), 1619
OutputChecker (doctest 中的类), 1619
OutputString() (http.cookies.Morsel 方法), 1400
output (subprocess.CalledProcessError 属性), 953
output (subprocess.TimeoutExpired 属性), 953
output (unittest.TestCase 属性), 1636
OutsideDestinationError, 554
OverflowError, 103
Overflow (decimal 中的类), 343
overlap() (statistics.NormalDist 方法), 374
overlaps() (ipaddress.IPv4Network 方法), 1429
overlaps() (ipaddress.IPv6Network 方法), 1431
overlay() (curses.window 方法), 865
overload() (在 typing 模块中), 1587
override() (在 typing 模块中), 1588
overwrite() (curses.window 方法), 865
owner() (pathlib.Path 方法), 431

P

-p
    compileall 命令行选项, 1993
    http.server 命令行选项, 1397
    pickletools 命令行选项, 2017
    timeit 命令行选项, 1756
    unittest-discover 命令行选项, 1626
    zipapp 命令行选项, 1782
p (*pdb command*), 1743
P_ALL() (在 os 模块中), 670
P_DETACH() (在 os 模块中), 668
P_NOWAIT() (在 os 模块中), 668
P_NOWAITO() (在 os 模块中), 668
P_OVERLAY() (在 os 模块中), 668
P_PGID() (在 os 模块中), 670
P_PID() (在 os 模块中), 670
P_PIDFD() (在 os 模块中), 670
P_WAIT() (在 os 模块中), 668
pack() (mailbox.MH 方法), 1228
pack() (struct.Struct 方法), 173
pack() (在 struct 模块中), 167
pack_into() (struct.Struct 方法), 173
pack_into() (在 struct 模块中), 168
package -- 包, **2073**
PackageMetadata(importlib.metadata 中的类), 1935
PackageNotFoundError, 1933
PackagePath (importlib.metadata 中的类), 1935
packages_distributions() (在 importlib.metadata 模块中), 1936
packed (ipaddress.IPv4Address 属性), 1424
packed (ipaddress.IPv6Address 属性), 1426
packing (部件), 1508
PAGER, 1598
pair_content() (在 curses 模块中), 858
pair_number() (在 curses 模块中), 858
pairwise() (在 itertools 模块中), 384
parameter -- 形参, **2073**
ParameterizedMIMEHeader (email.headerregistry 中的类), 1183
parameters (inspect.Signature 属性), 1882
Parameter (inspect 中的类), 1883
ParamSpecArgs() (在 typing 模块中), 1576
ParamSpecKwargs() (在 typing 模块中), 1576
ParamSpec (ast 中的类), 1967
ParamSpec (typing 中的类), 1575
paramstyle() (在 sqlite3 模块中), 499
params(email.headerregistry.ParameterizedMIMEHeader 属性), 1183
pardir() (在 os 模块中), 675
parent() (tkinter.ttk.Treeview 方法), 1532
parent_process() (在 multiprocessing 模块中), 910
parentNode (xml.dom.Node 属性), 1275
parents (collections.ChainMap 属性), 239
parents (pathlib.PurePath 属性), 416
parent (importlib.machinery.ModuleSpec 属性), 1922
parent (logging.Logger 属性), 703

parent (pathlib.PurePath 属性), 416
parent (pyclbr.Class 属性), 1990
parent (pyclbr.Function 属性), 1990
parent (urllib.request.BaseHandler 属性), 1329
paretovariate() (在 random 模块中), 358
parse() (doctest.DocTestParser 方法), 1617
parse() (email.parser.BytesParser 方法), 1169
parse() (email.parser.Parser 方法), 1169
parse() (string.Formatter 方法), 112
parse() (urllib.robotparser.RobotFileParser 方法), 1348
parse() (xml.etree.ElementTree.ElementTree 方法), 1269
Parse() (xml.parsers.expat.xmlparser 方法), 1301
parse() (xml.sax.xmlreader.XMLReader 方法), 1297
parse() (在 ast 模块中), 1971
parse() (在 xml.dom.minidom 模块中), 1283
parse() (在 xml.dom.pulldom 模块中), 1287
parse() (在 xml.etree.ElementTree 模块中), 1264
parse() (在 xml.sax 模块中), 1289
parse_and_bind() (在 readline 模块中), 161
parse_args() (argparse.ArgumentParser 方法), 801
parse_args() (optparse.OptionParser 方法), 842
PARSE_COLNAMES() (在 sqlite3 模块中), 498
parse_config_h() (在 sysconfig 模块中), 1819
PARSE_DECLTYPES() (在 sqlite3 模块中), 498
parse_headers() (在 http.client 模块中), 1353
parse_intermixed_args() (argparse.ArgumentParser 方法), 811
parse_known_args() (argparse.ArgumentParser 方法), 810
parse_known_intermixed_args() (argparse.ArgumentParser 方法), 811
parse_qs() (在 urllib.parse 模块中), 1340
parse_qsl() (在 urllib.parse 模块中), 1341
parseaddr() (在 email.utils 模块中), 1209
parsebytes() (email.parser.BytesParser 方法), 1169
parsedate() (在 email.utils 模块中), 1209
parsedate_to_datetime() (在 email.utils 模块中), 1210
parsedate_tz() (在 email.utils 模块中), 1210
ParseError (xml.etree.ElementTree 中的类), 1273
ParseFile() (xml.parsers.expat.xmlparser 方法), 1301
ParseFlags() (在 imaplib 模块中), 1369
ParserCreate() (在 xml.parsers.expat 模块中), 1300
ParseResultBytes (urllib.parse 中的类), 1344
ParseResult (urllib.parse 中的类), 1344
Parser (email.parser 中的类), 1169
parser (pathlib.PurePath 属性), 415
parsestr() (email.parser.Parser 方法), 1169
parseString() (在 xml.dom.minidom 模块中), 1283
parseString() (在 xml.dom.pulldom 模块中), 1288
parseString() (在 xml.sax 模块中), 1289
ParsingError, 592
partial() (imaplib.IMAP4 方法), 1371
partial() (在 functools 模块中), 397
partialmethod (functools 中的类), 397
partial (asyncio.IncompleteReadError 属性), 1022
parties (asyncio.Barrier 属性), 1014
parties (threading.Barrier 属性), 897
partition() (bytearray 方法), 64
partition() (bytes 方法), 64
partition() (str 方法), 53
parts (pathlib.PurePath 属性), 415
pass_() (poplib.POP3 方法), 1366
Pass (ast 中的类), 1956
patch() (在 test.support 模块中), 1717
patch() (在 unittest.mock 模块中), 1669
patch.dict() (在 unittest.mock 模块中), 1672
patch.multiple() (在 unittest.mock 模块中), 1674
patch.object() (在 unittest.mock 模块中), 1672
patch.stopall() (在 unittest.mock 模块中), 1675
PATH, 433, 462, 661, 666, 667, 676, 955, 1309, 1775, 1894
path
    module 搜索, 457, 1802, 1893
    操作, 411, 434
    配置 文件, 1894
path based finder -- 基于路径的查找器, **2074**
path entry -- 路径入口, **2073**
path entry finder -- 路径入口查找器, **2073**
path entry hook -- 路径入口钩子, **2074**
path() (在 importlib.resources 模块中), 1930
path() (在 sys 模块中), 1802
path-like object -- 路径类对象, **2074**
path_hook() (importlib.machinery.FileFinder 类方法), 1919
path_hooks() (在 sys 模块中), 1802
path_importer_cache() (在 sys 模块中), 1802
path_mtime() (importlib.abc.SourceLoader 方法), 1915
path_return_ok() (http.cookiejar.CookiePolicy 方法), 1406

path_stats() (importlib.abc.SourceLoader 方法), 1915
path_stats() (importlib.machinery.SourceFileLoader 方法), 1920
pathconf() (在 os 模块中), 643
pathconf_names() (在 os 模块中), 643
PathEntryFinder(importlib.abc 中的类), 1912
PATHEXT, 462
PathFinder (importlib.machinery 中的类), 1919
pathlib
    module, 411
PathLike (os 中的类), 618
pathname2url() (在 urllib.request 模块中), 1323
pathsep() (在 os 模块中), 676
Path.stem() (在 zipfile 模块中), 548
Path.suffix() (在 zipfile 模块中), 548
Path.suffixes() (在 zipfile 模块中), 548
path (http.cookiejar.Cookie 属性), 1408
path (http.cookies.Morsel 属性), 1399
path (http.server.BaseHTTPRequestHandler 属性), 1393
path (ImportError 属性), 101
path (importlib.abc.FileLoader 属性), 1915
path(importlib.machinery.AppleFrameworkLoader 属性), 1923
path(importlib.machinery.ExtensionFileLoader 属性), 1921
path (importlib.machinery.FileFinder 属性), 1919
path(importlib.machinery.SourceFileLoader 属性), 1920
path(importlib.machinery.SourcelessFileLoader 属性), 1920
path (os.DirEntry 属性), 646
Path (pathlib 中的类), 421
Path (zipfile 中的类), 547
--pattern
    unittest-discover 命令行选项, 1626
PatternError, 133
Pattern (re 中的类), 134
pattern (re.Pattern 属性), 135
pattern (re.PatternError 属性), 133
Pattern (typing 中的类), 1593
pause() (在 signal 模块中), 1147
pause_reading() (asyncio.ReadTransport 方法), 1050
pause_writing() (asyncio.BaseProtocol 方法), 1053
PAX_FORMAT() (在 tarfile 模块中), 555
pax_headers (tarfile.TarFile 属性), 559
pax_headers (tarfile.TarInfo 属性), 560
pbkdf2_hmac() (在 hashlib 模块中), 603
pd() (在 turtle 模块中), 1470
pdb
    module, 1738
Pdb (*pdb* 中的类), 1738
.pdbrc
    文件, 1741
pdb 命令行选项
    -c, 1739
    --command, 1739
    -m, 1739
Pdb (pdb 中的类), 1740
pdf() (statistics.NormalDist 方法), 374
peek() (bz2.BZ2File 方法), 532
peek() (gzip.GzipFile 方法), 529
peek() (io.BufferedReader 方法), 685
peek() (lzma.LZMAFile 方法), 537
peek() (weakref.finalize 方法), 272
PEM_cert_to_DER_cert()(在 ssl 模块中), 1106
pen() (在 turtle 模块中), 1471
pencolor() (在 turtle 模块中), 1472
pending() (ssl.SSLSocket 方法), 1116
PendingDeprecationWarning, 107
pending (ssl.MemoryBIO 属性), 1131
pendown() (在 turtle 模块中), 1470
pensize() (在 turtle 模块中), 1470
penup() (在 turtle 模块中), 1470
PEP, **2074**
PERCENT() (在 token 模块中), 1982
PERCENTEQUAL() (在 token 模块中), 1983
perf_counter() (在 time 模块中), 693
perf_counter_ns() (在 time 模块中), 693
perm() (在 math 模块中), 314
PermissionError, 106
permutations() (在 itertools 模块中), 384
persistence, 469
persistent
    objects, 469
persistent_id (*pickle* 协议), 476
persistent_id() (pickle.Pickler 方法), 472
persistent_load (*pickle* 协议), 476
persistent_load() (pickle.Unpickler 方法), 473
PF_CAN() (在 socket 模块中), 1081
PF_DIVERT() (在 socket 模块中), 1081
PF_PACKET() (在 socket 模块中), 1082
PF_RDS() (在 socket 模块中), 1082
pformat() (pprint.PrettyPrinter 方法), 285
pformat() (在 pprint 模块中), 284
pgettext() (gettext.GNUTranslations 方法), 1445
pgettext() (gettext.NullTranslations 方法), 1443
pgettext() (在 gettext 模块中), 1442
PGO() (在 test.support 模块中), 1711
phase() (在 cmath 模块中), 322
pi() (xml.etree.ElementTree.TreeBuilder 方法), 1270
pi() (在 cmath 模块中), 324
pi() (在 math 模块中), 320
pickle
    module, 282, 469, 484, 487
pickle() (在 copyreg 模块中), 484

PickleBuffer (pickle 中的类), 474
PickleError, 472
Pickler (pickle 中的类), 472
pickletools
    module, 2017
pickletools 命令行选项
    -a, 2017
    --annotate, 2017
    --indentlevel, 2017
    -l, 2017
    -m, 2017
    --memo, 2017
    -o, 2017
    --output, 2017
    -p, 2017
    --preamble, 2017
pickling
    objects, 469
PicklingError, 472
PIDFD_NONBLOCK() (在 os 模块中), 665
pidfd_open() (在 os 模块中), 665
pidfd_send_signal()(在 signal 模块中), 1148
PidfdChildWatcher (asyncio 中的类), 1063
pid (asyncio.subprocess.Process 属性), 1018
pid (multiprocessing.Process 属性), 905
pid (subprocess.Popen 属性), 961
Pipe() (在 multiprocessing 模块中), 908
pipe() (在 os 模块中), 630
PIPE() (在 subprocess 模块中), 953
pipe2() (在 os 模块中), 630
PIPE_BUF() (在 select 模块中), 1135
pipe_connection_lost()
        (asyncio.SubprocessProtocol 方法), 1055
pipe_data_received()
        (asyncio.SubprocessProtocol 方法), 1055
PIPE_MAX_SIZE() (在 test.support 模块中), 1711
pipes
    module, 2058
pkgutil
    module, 1903
placeholder (textwrap.TextWrapper 属性), 157
platform
    module, 742
platform() (在 platform 模块中), 742
platform() (在 sys 模块中), 1802
platlibdir() (在 sys 模块中), 1803
PlaySound() (在 winsound 模块中), 2030
plist
    文件, 595
plistlib
    module, 595
plock() (在 os 模块中), 665
plus() (decimal.Context 方法), 340
PLUS() (在 token 模块中), 1982
PLUSEQUAL() (在 token 模块中), 1983
pm() (在 pdb 模块中), 1740
POINTER() (在 ctypes 模块中), 778
pointer() (在 ctypes 模块中), 778
polar() (在 cmath 模块中), 322
Policy (email.policy 中的类), 1174
poll() (multiprocessing.connection.Connection 方法), 912
poll() (select.devpoll 方法), 1136
poll() (select.epoll 方法), 1137
poll() (select.poll 方法), 1138
poll() (subprocess.Popen 方法), 959
poll() (在 select 模块中), 1135
PollSelector (selectors 中的类), 1143
Pool (multiprocessing.pool 中的类), 924
pop() (序列方法), 45
pop() (array.array 方法), 268
pop() (collections.deque 方法), 244
pop() (dict 方法), 85
pop() (frozenset 方法), 83
pop() (mailbox.Mailbox 方法), 1223
POP3
    协议, 1365
POP3_SSL (poplib 中的类), 1365
POP3 (poplib 中的类), 1365
pop_all() (contextlib.ExitStack 方法), 1850
POP_BLOCK (*opcode*), 2015
POP_EXCEPT (*opcode*), 2005
POP_JUMP_IF_FALSE (*opcode*), 2010
POP_JUMP_IF_NONE (*opcode*), 2010
POP_JUMP_IF_NOT_NONE (*opcode*), 2010
POP_JUMP_IF_TRUE (*opcode*), 2009
pop_source() (shlex.shlex 方法), 1498
POP_TOP (*opcode*), 2001
popen() (在 *os* 模块中), 1135
popen() (在 os 模块中), 665
Popen (subprocess 中的类), 955
popitem() (collections.OrderedDict 方法), 251
popitem() (dict 方法), 85
popitem() (mailbox.Mailbox 方法), 1223
popleft() (collections.deque 方法), 244
poplib
    module, 1365
port
    http.server 命令行选项, 1397
port_specified (http.cookiejar.Cookie 属性), 1408
portion -- 部分, **2074**
port (http.cookiejar.Cookie 属性), 1408
pos() (在 operator 模块中), 404
pos() (在 turtle 模块中), 1469
position() (在 turtle 模块中), 1469
positional argument -- 位置参数, **2074**
Positions.col_offset()(在 dis 模块中), 2001
Positions.end_col_offset() (在 dis 模块中), 2001

Positions.end_lineno()(在 dis 模块中), 2001
Positions.lineno() (在 dis 模块中), 2001
Positions (dis 中的类), 2001
positions (inspect.FrameInfo 属性), 1888
positions (inspect.Traceback 属性), 1888
position(xml.etree.ElementTree.ParseError 属性), 1273
POSIX
    I/O 控制, 2035
    threads, 978
posix
    module, 2033
POSIX 共享内存, 939
POSIX_FADV_DONTNEED() (在 os 模块中), 630
POSIX_FADV_NOREUSE() (在 os 模块中), 630
POSIX_FADV_NORMAL() (在 os 模块中), 630
POSIX_FADV_RANDOM() (在 os 模块中), 630
POSIX_FADV_SEQUENTIAL() (在 os 模块中), 630
POSIX_FADV_WILLNEED() (在 os 模块中), 630
posix_fadvise() (在 os 模块中), 630
posix_fallocate() (在 os 模块中), 630
posix_openpt() (在 os 模块中), 631
posix_spawn() (在 os 模块中), 665
POSIX_SPAWN_CLOSE() (在 os 模块中), 666
POSIX_SPAWN_CLOSEFROM() (在 os 模块中), 666
POSIX_SPAWN_DUP2() (在 os 模块中), 666
POSIX_SPAWN_OPEN() (在 os 模块中), 666
posix_spawnp() (在 os 模块中), 666
PosixPath (pathlib 中的类), 422
post_handshake_auth (ssl.SSLContext 属性), 1123
post_mortem() (在 pdb 模块中), 1740
post_setup() (venv.EnvBuilder 方法), 1778
postcmd() (cmd.Cmd 方法), 1492
postloop() (cmd.Cmd 方法), 1493
pos (json.JSONDecodeError 属性), 1218
pos (re.Match 属性), 137
pos (re.PatternError 属性), 133
pow()
    built-in function, 24
pow() (在 math 模块中), 317
pow() (在 operator 模块中), 404
power() (decimal.Context 方法), 340
Pow (ast 中的类), 1949
pp (*pdb command*), 1743
pp() (在 pprint 模块中), 283
pprint
    module, 283
pprint() (pprint.PrettyPrinter 方法), 285
pprint() (在 pprint 模块中), 284
prcal() (在 calendar 模块中), 235
pread() (在 os 模块中), 630
preadv() (在 os 模块中), 631
--preamble
    pickletools 命令行选项, 2017
preamble (email.message.EmailMessage 属性), 1167
preamble (email.message.Message 属性), 1201
precmd() (cmd.Cmd 方法), 1492
prefix() (在 sys 模块中), 1803
PREFIXES() (在 site 模块中), 1895
prefixlen (ipaddress.IPv4Network 属性), 1429
prefixlen (ipaddress.IPv6Network 属性), 1431
prefix (xml.dom.Attr 属性), 1279
prefix (xml.dom.Node 属性), 1276
prefix (zipimport.zipimporter 属性), 1903
preloop() (cmd.Cmd 方法), 1492
prepare() (graphlib.TopologicalSorter 方法), 306
prepare() (logging.handlers.QueueHandler 方法), 740
prepare() (logging.handlers.QueueListener 方法), 741
prepare_class() (在 types 模块中), 276
prepare_input_source() (在 xml.sax.saxutils 模块中), 1295
PrepareProtocol (sqlite3 中的类), 514
PrettyPrinter (pprint 中的类), 284
prev() (tkinter.ttk.Treeview 方法), 1532
previousSibling (xml.dom.Node 属性), 1276
print()
    built-in function, 24
print() (traceback.TracebackException 方法), 1865
print_callees() (pstats.Stats 方法), 1752
print_callers() (pstats.Stats 方法), 1751
print_exc() (timeit.Timer 方法), 1755
print_exc() (在 traceback 模块中), 1862
print_exception() (在 traceback 模块中), 1862
print_help() (argparse.ArgumentParser 方法), 810
print_last() (在 traceback 模块中), 1862
print_list() (在 traceback 模块中), 1863
print_stack() (asyncio.Task 方法), 1000
print_stack() (在 traceback 模块中), 1863
print_stats() (profile.Profile 方法), 1749
print_stats() (pstats.Stats 方法), 1751
print_tb() (在 traceback 模块中), 1862
print_usage() (argparse.ArgumentParser 方法), 810
print_usage() (optparse.OptionParser 方法), 844
print_version() (optparse.OptionParser 方法), 834
print_warning() (在 test.support 模块中), 1714
printable() (在 string 模块中), 112
printdir() (zipfile.ZipFile 方法), 545
printf 风格的格式化, 58, 72
PRIO_DARWIN_BG() (在 os 模块中), 620
PRIO_DARWIN_NONUI() (在 os 模块中), 620
PRIO_DARWIN_PROCESS() (在 os 模块中), 620
PRIO_DARWIN_THREAD() (在 os 模块中), 620

PRIO_PGRP() (在 os 模块中), 620
PRIO_PROCESS() (在 os 模块中), 620
PRIO_USER() (在 os 模块中), 620
PriorityQueue (asyncio 中的类), 1020
PriorityQueue (queue 中的类), 971
prlimit() (在 resource 模块中), 2043
prmonth() (calendar.TextCalendar 方法), 232
prmonth() (在 calendar 模块中), 234
ProactorEventLoop (asyncio 中的类), 1042
process
    group, 619
    id, 619
    killing, 664
    signalling, 664
    父 id, 620
    调度优先级, 620, 622
--process
    timeit 命令行选项, 1756
process() (logging.LoggerAdapter 方法), 712
process_cpu_count() (在 os 模块中), 675
process_exited() (asyncio.SubprocessProtocol 方法), 1055
process_request() (socketserver.BaseServer 方法), 1387
process_time() (在 time 模块中), 693
process_time_ns() (在 time 模块中), 693
process_tokens() (在 tabnanny 模块中), 1989
ProcessError, 907
processingInstruction() (xml.sax.handler.ContentHandler 方法), 1293
ProcessingInstruction() (在 xml.etree.ElementTree 模块中), 1264
ProcessingInstructionHandler() (xml.parsers.expat.xmlparser 方法), 1303
ProcessLookupError, 106
processor time, 693, 698
processor() (在 platform 模块中), 742
ProcessPoolExecutor (concurrent.futures 中的类), 947
Process (multiprocessing 中的类), 904
prod() (在 math 模块中), 318
product() (在 itertools 模块中), 385
profile
    module, 1749
Profile (profile 中的类), 1749
ProgrammingError, 514
Progressbar (tkinter.ttk 中的类), 1527
prompt_user_passwd() (urllib.request.FancyURLopener 方法), 1337
prompt (cmd.Cmd 属性), 1493
propagate (logging.Logger 属性), 703
property list, 595
property() (在 enum 模块中), 304
property_declaration_handler() (在 xml.sax.handler 模块中), 1291
property_dom_node() (在 xml.sax.handler 模块中), 1291
property_lexical_handler() (在 xml.sax.handler 模块中), 1291
property_xml_string() (在 xml.sax.handler 模块中), 1291
property.deleter()
    built-in function, 25
property.getter()
    built-in function, 25
PropertyMock (unittest.mock 中的类), 1661
property.setter()
    built-in function, 25
property (内置类), 25
prot_c() (ftplib.FTP_TLS 方法), 1364
prot_p() (ftplib.FTP_TLS 方法), 1364
--protocol
    http.server 命令行选项, 1397
PROTOCOL_SSLv3() (在 ssl 模块中), 1108
PROTOCOL_SSLv23() (在 ssl 模块中), 1108
PROTOCOL_TLS() (在 ssl 模块中), 1108
PROTOCOL_TLS_CLIENT() (在 ssl 模块中), 1108
PROTOCOL_TLS_SERVER() (在 ssl 模块中), 1108
PROTOCOL_TLSv1() (在 ssl 模块中), 1108
PROTOCOL_TLSv1_1() (在 ssl 模块中), 1108
PROTOCOL_TLSv1_2() (在 ssl 模块中), 1108
protocol_version(http.server.BaseHTTPRequestHandler 属性), 1394
PROTOCOL_VERSION (imaplib.IMAP4 属性), 1373
ProtocolError (xmlrpc.client 中的类), 1414
Protocol (asyncio 中的类), 1052
protocol (ssl.SSLContext 属性), 1123
Protocol (typing 中的类), 1579
proto (socket.socket 属性), 1097
provisional API -- 暂定 API, **2074**
provisional package -- 暂定包, **2074**
proxy() (在 weakref 模块中), 270
proxyauth() (imaplib.IMAP4 方法), 1371
ProxyBasicAuthHandler (urllib.request 中的类), 1325
ProxyDigestAuthHandler (urllib.request 中的类), 1326
ProxyHandler (urllib.request 中的类), 1324
ProxyType() (在 weakref 模块中), 272
ProxyTypes() (在 weakref 模块中), 272
pryear() (calendar.TextCalendar 方法), 232
ps1() (在 sys 模块中), 1804
ps2() (在 sys 模块中), 1804
pstats
    module, 1750
pstdev() (在 statistics 模块中), 369
pthread_getcpuclockid() (在 time 模块中), 691
pthread_kill() (在 signal 模块中), 1148

pthread_sigmask() (在 signal 模块中), 1148
pthreads, 978
pthreads (sys._emscripten_info 属性), 1791
ptsname() (在 os 模块中), 631
pty
    module, 630, 2038
pu() (在 turtle 模块中), 1470
publicId (xml.dom.DocumentType 属性), 1277
PullDom (xml.dom.pulldom 中的类), 1287
punctuation() (在 string 模块中), 112
punctuation_chars (shlex.shlex 属性), 1499
PurePath (pathlib 中的类), 413
PurePosixPath (pathlib 中的类), 413
PureWindowsPath (pathlib 中的类), 414
purge() (在 re 模块中), 133
Purpose.CLIENT_AUTH() (在 ssl 模块中), 1112
Purpose.SERVER_AUTH() (在 ssl 模块中), 1112
push() (code.InteractiveConsole 方法), 1899
push() (contextlib.ExitStack 方法), 1849
push_async_callback() (contextlib.AsyncExitStack 方法), 1850
push_async_exit() (contextlib.AsyncExitStack 方法), 1850
PUSH_EXC_INFO (*opcode*), 2005
PUSH_NULL (*opcode*), 2012
push_source() (shlex.shlex 方法), 1498
push_token() (shlex.shlex 方法), 1497
put() (asyncio.Queue 方法), 1020
put() (multiprocessing.Queue 方法), 908
put() (multiprocessing.SimpleQueue 方法), 909
put() (queue.Queue 方法), 972
put() (queue.SimpleQueue 方法), 974
put_nowait() (asyncio.Queue 方法), 1020
put_nowait() (multiprocessing.Queue 方法), 908
put_nowait() (queue.Queue 方法), 972
put_nowait() (queue.SimpleQueue 方法), 974
putch() (在 msvcrt 模块中), 2020
putenv() (在 os 模块中), 620
putheader() (http.client.HTTPConnection 方法), 1356
putp() (在 curses 模块中), 858
putrequest() (http.client.HTTPConnection 方法), 1356
putwch() (在 msvcrt 模块中), 2020
putwin() (curses.window 方法), 865
pvariance() (在 statistics 模块中), 369
pwd
    module, 436, 2034
pwd() (ftplib.FTP 方法), 1363
pwrite() (在 os 模块中), 632
pwritev() (在 os 模块中), 632
py_compile
    module, 1991
Py_DEBUG() (在 test.support 模块中), 1711
py_object (ctypes 中的类), 781
PY_RESUME (*monitoring event*), 1812
PY_RETURN (*monitoring event*), 1812
PY_START (*monitoring event*), 1812
PY_THROW (*monitoring event*), 1812
PY_UNWIND (*monitoring event*), 1812
PY_YIELD (*monitoring event*), 1812
pycache_prefix() (在 sys 模块中), 1791
PyCF_ALLOW_TOP_LEVEL_AWAIT() (在 ast 模块中), 1975
PyCF_ONLY_AST() (在 ast 模块中), 1975
PyCF_OPTIMIZED_AST() (在 ast 模块中), 1975
PyCF_TYPE_COMMENTS() (在 ast 模块中), 1975
PycInvalidationMode (py_compile 中的类), 1991
pyclbr
    module, 1989
PyCompileError, 1991
PyDLL (ctypes 中的类), 771
pydoc
    module, 1597
pyexpat
    module, 1300
PYFUNCTYPE() (在 ctypes 模块中), 774
--python
    zipapp 命令行选项, 1782
Python 3000, **2074**
Python 增强建议; PEP 1, 2074
Python 增强建议; PEP 8, 28
Python 增强建议; PEP 205, 272
Python 增强建议; PEP 227, 1871
Python 增强建议; PEP 235, 1910
Python 增强建议; PEP 236, 1872
Python 增强建议; PEP 237, 60, 74
Python 增强建议; PEP 238, 1871, 2068
Python 增强建议; PEP 246, 514
Python 增强建议; PEP 249, 494, 497, 509, 514, 516, 523
Python 增强建议; PEP 0249#threadsafety, 499
Python 增强建议; PEP 255, 1871
Python 增强建议; PEP 263, 1910, 1985, 1986
Python 增强建议; PEP 273, 1901
Python 增强建议; PEP 278, 2077
Python 增强建议; PEP 282, 464, 718
Python 增强建议; PEP 292, 120
Python 增强建议; PEP 302, 31, 457, 1802, 1901, 1904, 1905, 1907, 1910, 1913, 1914, 2071
Python 增强建议; PEP 307, 470
Python 增强建议; PEP 324, 951
Python 增强建议; PEP 328, 31, 1871, 1910
Python 增强建议; PEP 338, 1909
Python 增强建议; PEP 342, 257
Python 增强建议; PEP 343, 1853, 1871, 2066
Python 增强建议; PEP 362, 1885, 2064, 2073
Python 增强建议; PEP 366, 1909, 1910
Python 增强建议; PEP 370, 1896
Python 增强建议; PEP 378, 115

Python 增强建议; PEP 380#use-of-stopiteration-to-return-values, 1813
Python 增强建议; PEP 383, 177, 1076
Python 增强建议; PEP 387, 107
Python 增强建议; PEP 393, 182, 1801
Python 增强建议; PEP 405, 1773
Python 增强建议; PEP 411, 1799, 1806, 2074
Python 增强建议; PEP 412, 394
Python 增强建议; PEP 420, 1910, 2072, 2074
Python 增强建议; PEP 421, 1800
Python 增强建议; PEP 428, 412
Python 增强建议; PEP 434, 1548
Python 增强建议; PEP 442, 1874
Python 增强建议; PEP 443, 2069
Python 增强建议; PEP 451, 1802, 1904, 19081910
Python 增强建议; PEP 453, 1771
Python 增强建议; PEP 461, 74
Python 增强建议; PEP 468, 252
Python 增强建议; PEP 475, 24, 106, 629, 633, 635, 670, 694, 10911096, 11351139, 1142, 1150
Python 增强建议; PEP 479, 103, 104, 1871
Python 增强建议; PEP 483, 2069
Python 增强建议; PEP 484, 92, 1551, 1560, 1572, 1587, 1946, 1971, 1975, 2063, 2068, 2069, 2077
Python 增强建议; PEP 485, 316, 324
Python 增强建议; PEP 488, 1724, 1910, 1923, 1991
Python 增强建议; PEP 489, 1910, 1918, 1921, 1925
Python 增强建议; PEP 492, 258, 1892, 2064, 2066
Python 增强建议; PEP 495, 226
Python 增强建议; PEP 498, 2067
Python 增强建议; PEP 506, 612
Python 增强建议; PEP 515, 115, 352
Python 增强建议; PEP 519, 2074
Python 增强建议; PEP 524, 676
Python 增强建议; PEP 525, 258, 1799, 1806, 1892, 2064
Python 增强建议; PEP 526, 1565, 1578, 1832, 1839, 1971, 1975, 2063, 2077
Python 增强建议; PEP 529, 639, 1796, 1807
Python 增强建议; PEP 538, 1453
Python 增强建议; PEP 540, 616, 1453
Python 增强建议; PEP 544, 1560, 1579
Python 增强建议; PEP 552, 1910, 1991
Python 增强建议; PEP 557, 1832
Python 增强建议; PEP 560, 277
Python 增强建议; PEP 563, 1592, 1871
Python 增强建议; PEP 565, 107
Python 增强建议; PEP 566, 1935
Python 增强建议; PEP 567, 974, 1026, 1047
Python 增强建议; PEP 574, 471, 482
Python 增强建议; PEP 578, 1727, 1787
Python 增强建议; PEP 584, 239, 247, 252, 271, 280, 617
Python 增强建议; PEP 585, 92, 255, 279, 15911597, 2069
Python 增强建议; PEP 586, 1565
Python 增强建议; PEP 589, 1583
Python 增强建议; PEP 591, 1566, 1588
Python 增强建议; PEP 593, 1568, 1589
Python 增强建议; PEP 594, 2051, 20552059
Python 增强建议; PEP 597, 679
Python 增强建议; PEP 604, 94
Python 增强建议; PEP 610, 1937
Python 增强建议; PEP 612, 1553, 1558, 1565, 1576, 1596
Python 增强建议; PEP 613, 1563
Python 增强建议; PEP 615, 226
Python 增强建议; PEP 626, 1999
Python 增强建议; PEP 632, 2056
Python 增强建议; PEP 644, 1102
Python 增强建议; PEP 646, 1575
Python 增强建议; PEP 647, 1569
Python 增强建议; PEP 649, 1871
Python 增强建议; PEP 655, 1566, 1583
Python 增强建议; PEP 667, 20, 1739
Python 增强建议; PEP 673, 1563
Python 增强建议; PEP 675, 1561
Python 增强建议; PEP 681, 1587
Python 增强建议; PEP 682, 115
Python 增强建议; PEP 683, 2069
Python 增强建议; PEP 686, 617, 679
Python 增强建议; PEP 688, 259
Python 增强建议; PEP 692, 1570
Python 增强建议; PEP 695, 1561, 1572, 1573, 1575, 1576, 1597
Python 增强建议; PEP 698, 1589
Python 增强建议; PEP 702, 1831
Python 增强建议; PEP 703, 2068, 2069
Python 增强建议; PEP 705, 1566
Python 增强建议; PEP 706, 561
Python 增强建议; PEP 709, 20
Python 增强建议; PEP 742, 1569
Python 增强建议; PEP 3101, 112
Python 增强建议; PEP 3105, 1871
Python 增强建议; PEP 3112, 1871
Python 增强建议; PEP 3115, 276, 1970
Python 增强建议; PEP 3116, 2077
Python 增强建议; PEP 3118, 76
Python 增强建议; PEP 3119, 259, 1856
Python 增强建议; PEP 3120, 1910
Python 增强建议; PEP 3134, 100
Python 增强建议; PEP 3141, 309, 1856
Python 增强建议; PEP 3147, 1724, 1908, 1910, 1923, 1991, 19931995
Python 增强建议; PEP 3148, 950
Python 增强建议; PEP 3149, 1787
Python 增强建议; PEP 3151, 107, 1079, 1134, 2042
Python 增强建议; PEP 3154, 470
Python 增强建议; PEP 3155, 2075
Python 增强建议; PEP 3333, 13121316, 1319, 1320
Python 增强建议; PEP 3333#input-and-error-streams, 1320
Python 增强建议; PEP

3333#optional-platform-specific-file-system-extensions), 1320
Python 增强建议; PEP 3333#the-start-response-callable, 1320
Python 编辑器, 1537
python_branch() (在 platform 模块中), 743
python_build() (在 platform 模块中), 742
python_compiler() (在 platform 模块中), 742
PYTHON_CPU_COUNT, 675, 910
PYTHON_DOM, 1274
PYTHON_GIL, 2069
python_implementation() (在 platform 模块中), 743
python_is_optimized() (在 test.support 模块中), 1712
python_revision() (在 platform 模块中), 743
python_version() (在 platform 模块中), 743
python_version_tuple() (在 platform 模块中), 743
PYTHONASYNCIODEBUG, 1038, 1073, 1599
PYTHONBREAKPOINT, 9, 1789, 1790
PYTHONCASEOK, 32
PYTHONCOERCECLOCALE, 617
PYTHONDEVMODE, 1598
PYTHONDONTWRITEBYTECODE, 1790
PYTHONFAULTHANDLER, 1599, 1736
PythonFinalizationError, 103
PYTHONHOME, 1719, 1939, 1940
Pythonic, **2074**
PYTHONINTMAXSTRDIGITS, 97, 1800
PYTHONIOENCODING, 616, 1808
PYTHONLEGACYWINDOWSFSENCODING, 1807
PYTHONLEGACYWINDOWSSTDIO, 1808
PYTHONMALLOC, 1599
python--m-py_compile 命令行选项
  -, 1992
  <file>, 1992
  -q, 1992
  --quiet, 1992
python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 命令行选项
  -h, 516
  --help, 516
  -v, 516
  --version, 516
PYTHONNOUSERSITE, 1895
PYTHONPATH, 1719, 1802, 1939
PYTHONPLATLIBDIR, 1939
PYTHONPYCACHEPREFIX, 1791
PYTHONSAFEPATH, 1802, 2061
PYTHONSTARTUP, 163, 982, 1544, 1800, 1895
PYTHONTRACEMALLOC, 1761, 1766
PYTHONTZPATH, 230
PYTHONUNBUFFERED, 1808
PYTHONUSERBASE, 1895
PYTHONUSERSITE, 1719
PYTHONUTF8, 616, 617, 1808
PYTHONWARNDEFAULTENCODING, 679
PYTHONWARNINGS, 1599, 1827, 1828
PyZipFile (zipfile 中的类), 548

Q

-q
  compileall 命令行选项, 1993
  python--m-py_compile 命令行选项, 1992
qiflush() (在 curses 模块中), 858
QName (xml.etree.ElementTree 中的类), 1270
qsize() (asyncio.Queue 方法), 1020
qsize() (multiprocessing.Queue 方法), 908
qsize() (queue.Queue 方法), 972
qsize() (queue.SimpleQueue 方法), 974
qualified name -- 限定名称, **2075**
quantiles() (statistics.NormalDist 方法), 374
quantiles() (在 statistics 模块中), 371
quantize() (decimal.Context 方法), 340
quantize() (decimal.Decimal 方法), 333
QueryInfoKey() (在 winreg 模块中), 2025
QueryReflectionKey() (在 winreg 模块中), 2027
QueryValue() (在 winreg 模块中), 2025
QueryValueEx() (在 winreg 模块中), 2025
QUESTION() (在 tkinter.messagebox 模块中), 1519
queue
  module, 971
Queue() (multiprocessing.managers.SyncManager 方法), 920
QueueEmpty, 1021
QueueFull, 1021
QueueHandler (logging.handlers 中的类), 739
QueueListener (logging.handlers 中的类), 740
QueueShutDown, 1021
Queue (asyncio 中的类), 1019
Queue (multiprocessing 中的类), 908
Queue (queue 中的类), 971
queue (sched.scheduler 属性), 971
quick_ratio() (difflib.SequenceMatcher 方法), 149
--quiet
  python--m-py_compile 命令行选项, 1992
quiet (sys.flags 属性), 1793
quit (*pdb command*), 1745
quit() (ftplib.FTP 方法), 1363
quit() (poplib.POP3 方法), 1367
quit() (smtplib.SMTP 方法), 1379
quit() (tkinter.filedialog.FileDialog 方法), 1516
quitting (*bdb.Bdb* 属性), 1734
quit (内置变量), 34
quopri
  module, 1246
quote() (在 email.utils 模块中), 1209
quote() (在 shlex 模块中), 1496
quote() (在 urllib.parse 模块中), 1345

QUOTE_ALL() (在 csv 模块中), 572
quote_from_bytes() (在 urllib.parse 模块中), 1345
QUOTE_MINIMAL() (在 csv 模块中), 572
QUOTE_NONE() (在 csv 模块中), 572
QUOTE_NONNUMERIC() (在 csv 模块中), 572
QUOTE_NOTNULL() (在 csv 模块中), 573
quote_plus() (在 urllib.parse 模块中), 1345
QUOTE_STRINGS() (在 csv 模块中), 573
quoteattr() (在 xml.sax.saxutils 模块中), 1295
quotechar (csv.Dialect 属性), 573
quoted-printable
    encoding, 1246
quotes (shlex.shlex 属性), 1498
quoting (csv.Dialect 属性), 574

R

-R
    trace 命令行选项, 1759
-r
    compileall 命令行选项, 1993
    idle 命令行选项, 1544
    timeit 命令行选项, 1756
    trace 命令行选项, 1759
R_OK() (在 os 模块中), 637
radians() (在 math 模块中), 318
radians() (在 turtle 模块中), 1470
radix() (decimal.Context 方法), 340
radix() (decimal.Decimal 方法), 334
RADIXCHAR() (在 locale 模块中), 1451
radix (sys.float_info 属性), 1795
raise
    statement -- 语句, 99
RAISE (*monitoring event*), 1812
raise_on_defect (email.policy.Policy 属性), 1175
raise_signal() (在 signal 模块中), 1148
RAISE_VARARGS (*opcode*), 2011
raiseExceptions() (在 logging 模块中), 717
Raise (ast 中的类), 1955
RAND_add() (在 ssl 模块中), 1105
RAND_bytes() (在 ssl 模块中), 1105
RAND_status() (在 ssl 模块中), 1105
randbelow() (在 secrets 模块中), 612
randbits() (在 secrets 模块中), 612
randbytes() (在 random 模块中), 355
randint() (在 random 模块中), 356
random
    module, 354
random() (random.Random 方法), 358
random() (在 random 模块中), 357
random 命令行选项
    -c, 362
    --choice, 362
    -f, 362
    --float, 362
    -h, 362
    --help, 362
    -i, 362
    --integer, 362
Random (random 中的类), 358
randrange() (在 random 模块中), 355
range
    object -- 对象, 47
range (内置类), 47
RARROW() (在 token 模块中), 1984
ratio() (difflib.SequenceMatcher 方法), 148
Rational (numbers 中的类), 310
raw() (pickle.PickleBuffer 方法), 474
raw() (在 curses 模块中), 858
raw_data_manager() (在 email.contentmanager 模块中), 1186
raw_decode() (json.JSONDecoder 方法), 1217
raw_input() (code.InteractiveConsole 方法), 1899
RawArray() (在 multiprocessing.sharedctypes 模块中), 916
RawConfigParser (configparser 中的类), 591
RawDescriptionHelpFormatter (argparse 中的类), 788
RawIOBase (io 中的类), 682
RawPen (turtle 中的类), 1486
RawTextHelpFormatter (argparse 中的类), 788
RawTurtle (turtle 中的类), 1486
RawValue() (在 multiprocessing.sharedctypes 模块中), 916
raw (io.BufferedIOBase 属性), 683
RBRACE() (在 token 模块中), 1982
re
    module, 49, 122, 456
read() (asyncio.StreamReader 方法), 1004
read() (codecs.StreamReader 方法), 181
read() (configparser.ConfigParser 方法), 589
read() (http.client.HTTPResponse 方法), 1356
read() (imaplib.IMAP4 方法), 1371
read() (io.BufferedIOBase 方法), 683
read() (io.BufferedReader 方法), 685
read() (io.RawIOBase 方法), 682
read() (io.TextIOBase 方法), 687
read() (mimetypes.MimeTypes 方法), 1241
read() (mmap.mmap 方法), 1155
read() (sqlite3.Blob 方法), 513
read() (ssl.MemoryBIO 方法), 1132
read() (ssl.SSLSocket 方法), 1114
read() (urllib.robotparser.RobotFileParser 方法), 1347
read() (zipfile.ZipFile 方法), 545
read() (在 os 模块中), 632
read1() (bz2.BZ2File 方法), 533
read1() (io.BufferedIOBase 方法), 683
read1() (io.BufferedReader 方法), 685

read1() (io.BytesIO 方法), 685
read_binary() (在 importlib.resources 模块中), 1929
read_byte() (mmap.mmap 方法), 1155
read_bytes() (importlib.abc.Traversable 方法), 1917
read_bytes() (importlib.resources.abc.Traversable 方法), 1932
read_bytes() (pathlib.Path 方法), 426
read_bytes() (zipfile.Path 方法), 548
read_dict() (configparser.ConfigParser 方法), 590
read_environ() (在 wsgiref.handlers 模块中), 1319
read_events() (xml.etree.ElementTree.XMLPullParser 方法), 1272
read_file() (configparser.ConfigParser 方法), 589
read_history_file() (在 readline 模块中), 162
read_init_file() (在 readline 模块中), 161
read_mime_types() (在 mimetypes 模块中), 1240
read_string() (configparser.ConfigParser 方法), 589
read_text() (importlib.abc.Traversable 方法), 1917
read_text() (importlib.resources.abc.Traversable 方法), 1932
read_text() (pathlib.Path 方法), 426
read_text() (zipfile.Path 方法), 548
read_text() (在 importlib.resources 模块中), 1930
read_token() (shlex.shlex 方法), 1497
read_windows_registry() (mimetypes.MimeTypes 方法), 1241
readable() (bz2.BZ2File 方法), 533
readable() (io.IOBase 方法), 681
READABLE() (在 _tkinter 模块中), 1513
readall() (io.RawIOBase 方法), 682
reader() (在 csv 模块中), 569
ReadError, 554
readexactly() (asyncio.StreamReader 方法), 1005
readfp() (mimetypes.MimeTypes 方法), 1241
readframes() (wave.Wave_read 方法), 1438
readinto() (bz2.BZ2File 方法), 533
readinto() (http.client.HTTPResponse 方法), 1356
readinto() (io.BufferedIOBase 方法), 683
readinto() (io.RawIOBase 方法), 682
readinto1() (io.BufferedIOBase 方法), 683
readinto1() (io.BytesIO 方法), 685
readline
    module, 161
readline() (asyncio.StreamReader 方法), 1005
readline() (codecs.StreamReader 方法), 181
readline() (imaplib.IMAP4 方法), 1371
readline() (io.IOBase 方法), 681
readline() (io.TextIOBase 方法), 687
readline() (mmap.mmap 方法), 1155
readlines() (codecs.StreamReader 方法), 181
readlines() (io.IOBase 方法), 681
readlink() (pathlib.Path 方法), 424
readlink() (在 os 模块中), 643
readmodule() (在 pyclbr 模块中), 1989
readmodule_ex() (在 pyclbr 模块中), 1989
ReadOnly() (在 typing 模块中), 1566
readonly (memoryview 属性), 80
ReadTransport (asyncio 中的类), 1049
readuntil() (asyncio.StreamReader 方法), 1005
readv() (在 os 模块中), 634
ready() (multiprocessing.pool.AsyncResult 方法), 926
READ (inspect.BufferFlags 属性), 1893
real_max_memuse() (在 test.support 模块中), 1711
real_quick_ratio() (difflib.SequenceMatcher 方法), 149
realpath() (在 os.path 模块中), 438
REALTIME_PRIORITY_CLASS() (在 subprocess 模块中), 963
Real (numbers 中的类), 309
real (numbers.Complex 属性), 309
reap_children() (在 test.support 模块中), 1716
reap_threads() (在 test.support.threading_helper 模块中), 1720
reason (http.client.HTTPResponse 属性), 1357
reason (ssl.SSLError 属性), 1104
reason (UnicodeError 属性), 105
reason (urllib.error.HTTPError 属性), 1347
reason (urllib.error.URLError 属性), 1347
reattach() (tkinter.ttk.Treeview 方法), 1532
recent() (imaplib.IMAP4 方法), 1371
reconfigure() (io.TextIOWrapper 方法), 688
record_original_stdout() (在 test.support 模块中), 1713
RECORDS_RO (inspect.BufferFlags 属性), 1893
RECORDS (inspect.BufferFlags 属性), 1893
records (unittest.TestCase 属性), 1636
rect() (在 cmath 模块中), 322
rectangle() (在 curses.textpad 模块中), 878
RecursionError, 103
recursive_repr() (在 reprlib 模块中), 289
recv() (multiprocessing.connection.Connection 方法), 912
recv() (socket.socket 方法), 1093
recv_bytes() (multiprocessing.connection.Connection

方法），912
recv_bytes_into()（multiprocessing.connection.Connection 方法），912
recv_fds()（在 socket 模块中），1090
recv_into()（socket.socket 方法），1095
recvfrom()（socket.socket 方法），1093
recvfrom_into()（socket.socket 方法），1095
recvmsg()（socket.socket 方法），1093
recvmsg_into()（socket.socket 方法），1094
redirect_request()（urllib.request.HTTPRedirectHandler 方法），1330
redirect_stderr()（在 contextlib 模块中），1847
redirect_stdout()（在 contextlib 模块中），1846
redisplay()（在 readline 模块中），161
redrawln()（curses.window 方法），865
redrawwin()（curses.window 方法），865
reduce()（在 functools 模块中），398
reducer_override()（pickle.Pickler 方法），473
refcount_test()（在 test.support 模块中），1716
reference count -- 引用计数，**2075**
ReferenceError，103
ReferenceType()（在 weakref 模块中），272
refold_source（email.policy.EmailPolicy 属性），1177
refresh()（curses.window 方法），865
ref（weakref 中的类），270
REG_BINARY()（在 winreg 模块中），2028
REG_DWORD()（在 winreg 模块中），2028
REG_DWORD_BIG_ENDIAN()（在 winreg 模块中），2028
REG_DWORD_LITTLE_ENDIAN()（在 winreg 模块中），2028
REG_EXPAND_SZ()（在 winreg 模块中），2028
REG_FULL_RESOURCE_DESCRIPTOR()（在 winreg 模块中），2029
REG_LINK()（在 winreg 模块中），2028
REG_MULTI_SZ()（在 winreg 模块中），2028
REG_NONE()（在 winreg 模块中），2028
REG_QWORD()（在 winreg 模块中），2028
REG_QWORD_LITTLE_ENDIAN()（在 winreg 模块中），2028
REG_RESOURCE_LIST()（在 winreg 模块中），2029
REG_RESOURCE_REQUIREMENTS_LIST()（在 winreg 模块中），2029
REG_SZ()（在 winreg 模块中），2029
RegexFlag（re 中的类），128
register()（abc.ABCMeta 方法），1856
register()（argparse.ArgumentParser 方法），812
register()（multiprocessing.managers.BaseManager 方法），919
register()（select.devpoll 方法），1136
register()（select.epoll 方法），1137
register()（selectors.BaseSelector 方法），1141
register()（select.poll 方法），1137
register()（在 atexit 模块中），1860
register()（在 codecs 模块中），175
register()（在 faulthandler 模块中），1737
register()（在 webbrowser 模块中），1310
register_adapter()（在 sqlite3 模块中），498
register_archive_format()（在 shutil 模块中），464
register_at_fork()（在 os 模块中），667
register_callback()（在 sys.monitoring 模块中），1814
register_converter()（在 sqlite3 模块中），498
register_defect()（email.policy.Policy 方法），1175
register_dialect()（在 csv 模块中），570
register_error()（在 codecs 模块中），177
register_function()（xmlrpc.server.CGIXMLRPCRequestHandler 方法），1421
register_function()（xmlrpc.server.SimpleXMLRPCServer 方法），1418
register_instance()（xmlrpc.server.CGIXMLRPCRequestHandler 方法），1421
register_instance()（xmlrpc.server.SimpleXMLRPCServer 方法），1418
register_introspection_functions()（xmlrpc.server.CGIXMLRPCRequestHandler 方法），1421
register_introspection_functions()（xmlrpc.server.SimpleXMLRPCServer 方法），1418
register_multicall_functions()（xmlrpc.server.CGIXMLRPCRequestHandler 方法），1421
register_multicall_functions()（xmlrpc.server.SimpleXMLRPCServer 方法），1418
register_namespace()（在 xml.etree.ElementTree 模块中），1264
register_optionflag()（在 doctest 模块中），1609
register_shape()（在 turtle 模块中），1484
register_unpack_format()（在 shutil 模块中），465
registerDOMImplementation()（在 xml.dom 模块中），1274
registerResult()（在 unittest 模块中），1650
REGTYPE()（在 tarfile 模块中），554
regular package -- 常规包，**2075**
relative_to()（pathlib.PurePath 方法），420

release() (_thread.lock 方法), 980
release() (asyncio.Condition 方法), 1012
release() (asyncio.Lock 方法), 1010
release() (asyncio.Semaphore 方法), 1013
release() (logging.Handler 方法), 707
release() (memoryview 方法), 77
release() (multiprocessing.Lock 方法), 914
release() (multiprocessing.RLock 方法), 915
release() (pickle.PickleBuffer 方法), 474
release() (threading.Condition 方法), 893
release() (threading.Lock 方法), 891
release() (threading.RLock 方法), 892
release() (threading.Semaphore 方法), 895
release() (在 platform 模块中), 743
reload() (在 importlib 模块中), 1911
relpath() (在 os.path 模块中), 439
remainder() (decimal.Context 方法), 341
remainder() (在 math 模块中), 315
remainder_near() (decimal.Context 方法), 341
remainder_near() (decimal.Decimal 方法), 334
RemoteDisconnected, 1354
remove() (序列方法), 45
remove() (array.array 方法), 268
remove() (collections.deque 方法), 244
remove() (frozenset 方法), 83
remove() (mailbox.Mailbox 方法), 1222
remove() (mailbox.MH 方法), 1228
remove() (xml.etree.ElementTree.Element 方法), 1268
remove() (在 os 模块中), 643
remove_child_handler() (asyncio.AbstractChildWatcher 方法), 1062
remove_done_callback() (asyncio.Future 方法), 1047
remove_done_callback() (asyncio.Task 方法), 999
remove_flag() (mailbox.Maildir 方法), 1225
remove_flag() (mailbox.MaildirMessage 方法), 1231
remove_flag() (mailbox.mboxMessage 方法), 1233
remove_flag() (mailbox.MMDFMessage 方法), 1236
remove_folder() (mailbox.Maildir 方法), 1225
remove_folder() (mailbox.MH 方法), 1228
remove_header() (urllib.request.Request 方法), 1327
remove_history_item() (在 readline 模块中), 162
remove_label() (mailbox.BabylMessage 方法), 1235
remove_option() (configparser.ConfigParser 方法), 590
remove_option() (optparse.OptionParser 方法), 843
remove_reader() (asyncio.loop 方法), 1033
remove_section() (configparser.ConfigParser 方法), 591
remove_sequence() (mailbox.MHMessage 方法), 1234
remove_signal_handler() (asyncio.loop 方法), 1036
remove_writer() (asyncio.loop 方法), 1033
removeAttribute() (xml.dom.Element 方法), 1279
removeAttributeNode() (xml.dom.Element 方法), 1279
removeAttributeNS() (xml.dom.Element 方法), 1279
removeChild() (xml.dom.Node 方法), 1277
removedirs() (在 os 模块中), 644
removeFilter() (logging.Handler 方法), 707
removeFilter() (logging.Logger 方法), 706
removeHandler() (logging.Logger 方法), 706
removeHandler() (在 unittest 模块中), 1650
removeprefix() (bytearray 方法), 63
removeprefix() (bytes 方法), 63
removeprefix() (str 方法), 53
removeResult() (在 unittest 模块中), 1650
removesuffix() (bytearray 方法), 63
removesuffix() (bytes 方法), 63
removesuffix() (str 方法), 53
removexattr() (在 os 模块中), 660
rename() (ftplib.FTP 方法), 1362
rename() (imaplib.IMAP4 方法), 1371
rename() (pathlib.Path 方法), 431
rename() (在 os 模块中), 644
renames() (在 os 模块中), 644
reopenIfNeeded() (logging.handlers.WatchedFileHandler 方法), 731
reorganize() (dbm.gnu.gdbm 方法), 491
--repeat
    timeit 命令行选项, 1756
repeat() (timeit.Timer 方法), 1755
repeat() (在 itertools 模块中), 385
repeat() (在 timeit 模块中), 1754
REPL, **2075**
replace
    错误处理器名称, 176
replace() (bytearray 方法), 65
replace() (bytes 方法), 65
replace() (curses.panel.Panel 方法), 883
replace() (datetime.date 方法), 198
replace() (datetime.datetime 方法), 206
replace() (datetime.time 方法), 213
replace() (inspect.Parameter 方法), 1884
replace() (inspect.Signature 方法), 1882
replace() (pathlib.Path 方法), 431
replace() (str 方法), 53
replace() (tarfile.TarInfo 方法), 560
replace() (在 copy 模块中), 282

replace() (在 dataclasses 模块中), 1837
replace() (在 os 模块中), 644
replace_errors() (在 codecs 模块中), 178
replace_header() (email.message.EmailMessage 方法), 1163
replace_header() (email.message.Message 方法), 1198
replace_history_item() (在 readline 模块中), 162
replace_whitespace (textwrap.TextWrapper 属性), 156
replaceChild() (xml.dom.Node 方法), 1277
ReplacePackage() (在 modulefinder 模块中), 1906
--report
    trace 命令行选项, 1759
report() (filecmp.dircmp 方法), 447
report() (modulefinder.ModuleFinder 方法), 1906
REPORT_CDIFF() (在 doctest 模块中), 1609
REPORT_ERRMODE() (在 msvcrt 模块中), 2021
report_failure() (doctest.DocTestRunner 方法), 1618
report_full_closure() (filecmp.dircmp 方法), 447
REPORT_NDIFF() (在 doctest 模块中), 1609
REPORT_ONLY_FIRST_FAILURE() (在 doctest 模块中), 1609
report_partial_closure() (filecmp.dircmp 方法), 447
report_start() (doctest.DocTestRunner 方法), 1618
report_success() (doctest.DocTestRunner 方法), 1618
REPORT_UDIFF() (在 doctest 模块中), 1609
report_unexpected_exception() (doctest.DocTestRunner 方法), 1618
REPORTING_FLAGS() (在 doctest 模块中), 1609
repr()
    built-in function, 26
repr() (reprlib.Repr 方法), 290
repr() (在 reprlib 模块中), 288
repr1() (reprlib.Repr 方法), 290
ReprEnum (enum 中的类), 301
reprlib
    module, 288
Repr (reprlib 中的类), 288
request() (http.client.HTTPConnection 方法), 1354
request_queue_size (socketserver.BaseServer 属性), 1387
request_rate() (urllib.robotparser.RobotFileParser 方法), 1348
request_uri() (在 wsgiref.util 模块中), 1312
request_version (http.server.BaseHTTPRequestHandler 属性), 1393
RequestHandlerClass (socketserver.BaseServer 属性), 1387
requestline (http.server.BaseHTTPRequestHandler 属性), 1393
request (socketserver.BaseRequestHandler 属性), 1388
Request (urllib.request 中的类), 1323
Required() (在 typing 模块中), 1566
requires() (在 importlib.metadata 模块中), 1936
requires() (在 test.support 模块中), 1712
requires_bz2() (在 test.support 模块中), 1715
requires_docstrings() (在 test.support 模块中), 1715
requires_freebsd_version() (在 test.support 模块中), 1715
requires_gil_enabled() (在 test.support 模块中), 1715
requires_gzip() (在 test.support 模块中), 1715
requires_IEEE_754() (在 test.support 模块中), 1715
requires_limited_api() (在 test.support 模块中), 1715
requires_linux_version() (在 test.support 模块中), 1715
requires_lzma() (在 test.support 模块中), 1715
requires_mac_version() (在 test.support 模块中), 1715
requires_resource() (在 test.support 模块中), 1715
requires_zlib() (在 test.support 模块中), 1715
RERAISE (*monitoring event*), 1812
RERAISE (*opcode*), 2005
reschedule() (asyncio.Timeout 方法), 994
RESERVED_FUTURE() (在 uuid 模块中), 1383
RESERVED_MICROSOFT() (在 uuid 模块中), 1382
RESERVED_NCS() (在 uuid 模块中), 1382
reserved (zipfile.ZipInfo 属性), 550
reset() (asyncio.Barrier 方法), 1014
reset() (bdb.Bdb 方法), 1732
reset() (codecs.IncrementalDecoder 方法), 180
reset() (codecs.IncrementalEncoder 方法), 179
reset() (codecs.StreamReader 方法), 181
reset() (codecs.StreamWriter 方法), 180
reset() (contextvars.ContextVar 方法), 975
reset() (html.parser.HTMLParser 方法), 1250
reset() (threading.Barrier 方法), 897
reset() (xml.dom.pulldom.DOMEventStream 方法), 1288
reset() (xml.sax.xmlreader.IncrementalParser

方法), 1298
reset() (在 turtle 模块中), 1474
reset_mock() (unittest.mock.AsyncMock 方法), 1664
reset_mock() (unittest.mock.Mock 方法), 1655
reset_peak() (在 tracemalloc 模块中), 1765
reset_prog_mode() (在 curses 模块中), 858
reset_shell_mode() (在 curses 模块中), 858
reset_tzpath() (在 zoneinfo 模块中), 230
resetbuffer() (code.InteractiveConsole 方法), 1899
resetscreen() (在 turtle 模块中), 1480
resetty() (在 curses 模块中), 858
resetwarnings() (在 warnings 模块中), 1831
resize() (curses.window 方法), 865
resize() (mmap.mmap 方法), 1155
resize() (在 ctypes 模块中), 778
resize_term() (在 curses 模块中), 858
resizemode() (在 turtle 模块中), 1475
resizeterm() (在 curses 模块中), 858
resolution (datetime.date 属性), 197
resolution (datetime.datetime 属性), 204
resolution (datetime.time 属性), 212
resolution (datetime.timedelta 属性), 194
resolve() (pathlib.Path 方法), 424
resolve_bases() (在 types 模块中), 276
resolve_name() (在 importlib.util 模块中), 1923
resolve_name() (在 pkgutil 模块中), 1905
resolveEntity() (xml.sax.handler.EntityResolver 方法), 1294
resource
    module, 2042
resource_path() (importlib.abc.ResourceReader 方法), 1916
resource_path() (importlib.resources.abc.ResourceReader 方法), 1931
ResourceDenied, 1710
ResourceLoader (importlib.abc 中的类), 1913
ResourceReader (importlib.abc 中的类), 1916
ResourceReader (importlib.resources.abc 中的类), 1931
ResourceWarning, 108
response() (imaplib.IMAP4 方法), 1371
ResponseNotReady, 1353
responses() (在 http.client 模块中), 1354
responses (http.server.BaseHTTPRequestHandler 属性), 1394
restart (*pdb command*), 1745
restart_events() (在 sys.monitoring 模块中), 1814
restore() (test.support.SaveSignals 方法), 1718
restore() (在 difflib 模块中), 145
restype (ctypes._CFuncPtr 属性), 773
result() (asyncio.Future 方法), 1046
result() (asyncio.Task 方法), 999
result() (concurrent.futures.Future 方法), 949
results() (trace.Trace 方法), 1760
RESUME (*opcode*), 2013
resume_reading() (asyncio.ReadTransport 方法), 1050
resume_writing() (asyncio.BaseProtocol 方法), 1053
retr() (poplib.POP3 方法), 1366
retrbinary() (ftplib.FTP 方法), 1361
retrieve() (urllib.request.URLopener 方法), 1336
retrlines() (ftplib.FTP 方法), 1361
RETRY() (在 tkinter.messagebox 模块中), 1518
RETRYCANCEL() (在 tkinter.messagebox 模块中), 1519
return (*pdb command*), 1742
return_annotation (inspect.Signature 属性), 1882
RETURN_CONST (*opcode*), 2005
RETURN_GENERATOR (*opcode*), 2014
return_ok() (http.cookiejar.CookiePolicy 方法), 1405
RETURN_VALUE (*opcode*), 2005
return_value (unittest.mock.Mock 属性), 1657
returncode (asyncio.subprocess.Process 属性), 1018
returncode (subprocess.CalledProcessError 属性), 953
returncode (subprocess.CompletedProcess 属性), 952
returncode (subprocess.Popen 属性), 961
Return (ast 中的类), 1969
retval (*pdb command*), 1745
reveal_type() (在 typing 模块中), 1585
reverse() (*序列方法*), 45
reverse() (array.array 方法), 268
reverse() (collections.deque 方法), 244
reverse_order() (pstats.Stats 方法), 1751
reverse_pointer (ipaddress.IPv4Address 属性), 1424
reverse_pointer (ipaddress.IPv6Address 属性), 1426
reversed()
    built-in function, 26
Reversible (collections.abc 中的类), 257
Reversible (typing 中的类), 1596
revert() (http.cookiejar.FileCookieJar 方法), 1404
rewind() (wave.Wave_read 方法), 1438
re (re.Match 属性), 137
RFC
    RFC 821, 1374, 1376
    RFC 822, 696, 1188, 1204, 1356, 13771379, 1444
    RFC 959, 1359
    RFC 1123, 696
    RFC 1321, 599

RFC 1422, 1126, 1133
RFC 1521, 1244, 1247
RFC 1522, 1245, 1247
RFC 1730, 1368
RFC 1738, 1346
RFC 1750, 1105
RFC 1766, 1452
RFC 1808, 1339, 1346
RFC 1869, 1374, 1376
RFC 1939, 1365
RFC 2045, 1159, 1163, 1182, 1183, 1198, 1199, 1204, 1242, 1244
RFC 2045 Section 6.8, 1413
RFC 2046, 1159, 1187, 1204
RFC 2047, 1159, 1176, 1181, 1204, 1205, 1209
RFC 2060, 1368, 1372
RFC 2068, 1398
RFC 2104, 610
RFC 2109, 13981403, 1407, 1408
RFC 2183, 1159, 1164, 1200
RFC 2231, 1159, 1162, 1163, 1197, 1199, 1204, 1210, 1211
RFC 2295, 1350
RFC 2324, 1350
RFC 2342, 1371
RFC 2368, 1346
RFC 2373, 1424, 1425
RFC 2396, 1341, 1345, 1346
RFC 2397, 1332
RFC 2449, 1366
RFC 2595, 1365, 1367
RFC 2616, 1313, 1316, 1330, 1337, 1347
RFC 2616 Section 5.1.2, 1354
RFC 2616 Section 14.23, 1354
RFC 2640, 1359, 1360, 1364
RFC 2732, 1346
RFC 2821, 1159
RFC 2822, 696, 1196, 1204, 1205, 1209, 1210, 1230, 1353, 1393
RFC 2964, 1403
RFC 2965, 1324, 1327, 14011403, 14051409
RFC 3171, 1424
RFC 3280, 1114
RFC 3330, 1425
RFC 3454, 159, 160
RFC 3490, 187, 188
RFC 3490 Section 3.1, 188
RFC 3492, 187, 188
RFC 3493, 1101
RFC 3501, 1373
RFC 3542, 1089
RFC 3548, 1245
RFC 3659, 1362
RFC 3879, 1426
RFC 3927, 1425
RFC 3986, 1339, 1340, 1343, 1345, 1346, 1393
RFC 4007, 1425, 1426
RFC 4086, 1134
RFC 4122, 13801383
RFC 4180, 569
RFC 4193, 1426
RFC 4217, 1363
RFC 4291, 1425
RFC 4380, 1426
RFC 4627, 1212, 1220
RFC 4648, 12411244, 2061
RFC 4954, 1377
RFC 5161, 1370
RFC 5246, 1112, 1134
RFC 5280, 1103, 1105, 1134
RFC 5321, 1185
RFC 5322, 1159, 1160, 1169, 1171, 1172, 1174, 1176, 1177, 11791182, 1185, 1194, 1379
RFC 5424, 736
RFC 5735, 1425
RFC 5789, 1351
RFC 5891, 188
RFC 5895, 188
RFC 5929, 1116
RFC 6066, 1111, 1120, 1134
RFC 6531, 1161, 1176, 1374
RFC 6532, 1159, 1160, 1169, 1176
RFC 6585, 1350
RFC 6855, 1370
RFC 6856, 1367
RFC 7159, 1212, 1219, 1220
RFC 7230, 1324, 1356
RFC 7301, 1111, 1120
RFC 7525, 1134
RFC 7693, 603
RFC 7725, 1350
RFC 7914, 603
RFC 8089, 422
RFC 8297, 1349
RFC 8305, 1028
RFC 8470, 1350
RFC 9110, 13491351
rfc2109_as_netscape (http.cookiejar.DefaultCookiePolicy 属性), 1407
rfc2109 (http.cookiejar.Cookie 属性), 1408
rfc2965 (http.cookiejar.CookiePolicy 属性), 1406
RFC_4122() (在 uuid 模块中), 1382
rfile (http.server.BaseHTTPRequestHandler 属性), 1393
rfile(socketserver.DatagramRequestHandler 属性), 1388
rfind() (bytearray 方法), 65
rfind() (bytes 方法), 65
rfind() (mmap.mmap 方法), 1155
rfind() (str 方法), 53
rgb_to_hls() (在 colorsys 模块中), 1440
rgb_to_hsv() (在 colorsys 模块中), 1440
rgb_to_yiq() (在 colorsys 模块中), 1440
rglob() (pathlib.Path 方法), 428

right() (在 turtle 模块中), 1464
right_list (filecmp.dircmp 属性), 447
right_only (filecmp.dircmp 属性), 447
RIGHTSHIFT() (在 token 模块中), 1983
RIGHTSHIFTEQUAL() (在 token 模块中), 1983
right (filecmp.dircmp 属性), 447
rindex() (bytearray 方法), 65
rindex() (bytes 方法), 65
rindex() (str 方法), 53
rjust() (bytearray 方法), 66
rjust() (bytes 方法), 66
rjust() (str 方法), 54
rlcompleter
    module, 165
RLIM_INFINITY() (在 resource 模块中), 2042
RLIMIT_AS() (在 resource 模块中), 2043
RLIMIT_CORE() (在 resource 模块中), 2043
RLIMIT_CPU() (在 resource 模块中), 2043
RLIMIT_DATA() (在 resource 模块中), 2043
RLIMIT_FSIZE() (在 resource 模块中), 2043
RLIMIT_KQUEUES() (在 resource 模块中), 2044
RLIMIT_MEMLOCK() (在 resource 模块中), 2043
RLIMIT_MSGQUEUE() (在 resource 模块中), 2044
RLIMIT_NICE() (在 resource 模块中), 2044
RLIMIT_NOFILE() (在 resource 模块中), 2043
RLIMIT_NPROC() (在 resource 模块中), 2043
RLIMIT_NPTS() (在 resource 模块中), 2044
RLIMIT_OFILE() (在 resource 模块中), 2043
RLIMIT_RSS() (在 resource 模块中), 2043
RLIMIT_RTPRIO() (在 resource 模块中), 2044
RLIMIT_RTTIME() (在 resource 模块中), 2044
RLIMIT_SBSIZE() (在 resource 模块中), 2044
RLIMIT_SIGPENDING() (在 resource 模块中), 2044
RLIMIT_STACK() (在 resource 模块中), 2043
RLIMIT_SWAP() (在 resource 模块中), 2044
RLIMIT_VMEM() (在 resource 模块中), 2043
RLock() (multiprocessing.managers.SyncManager 方法), 920
RLock (multiprocessing 中的类), 914
RLock (threading 中的类), 891
rmd() (ftplib.FTP 方法), 1363
rmdir() (pathlib.Path 方法), 431
rmdir() (在 os 模块中), 644
rmdir() (在 test.support.os_helper 模块中), 1722
rmtree() (在 shutil 模块中), 460
rmtree() (在 test.support.os_helper 模块中), 1722
RobotFileParser (urllib.robotparser 中的类), 1347
robots.txt, 1347
rollback() (sqlite3.Connection 方法), 501
rollover() (tempfile.SpooledTemporaryFile 方法), 450
ROMAN() (在 tkinter.font 模块中), 1513
--root
    ensurepip 命令行选项, 1772
root (pathlib.PurePath 属性), 416
rotate() (collections.deque 方法), 244
rotate() (decimal.Context 方法), 341
rotate() (decimal.Decimal 方法), 334
rotate() (logging.handlers.BaseRotatingHandler 方法), 732
RotatingFileHandler (logging.handlers 中的类), 732
rotation_filename() (logging.handlers.BaseRotatingHandler 方法), 731
rotator (logging.handlers.BaseRotatingHandler 属性), 731
round()
    built-in function, 26
ROUND_05UP() (在 decimal 模块中), 342
ROUND_CEILING() (在 decimal 模块中), 342
ROUND_DOWN() (在 decimal 模块中), 342
ROUND_FLOOR() (在 decimal 模块中), 342
ROUND_HALF_DOWN() (在 decimal 模块中), 342
ROUND_HALF_EVEN() (在 decimal 模块中), 342
ROUND_HALF_UP() (在 decimal 模块中), 342
ROUND_UP() (在 decimal 模块中), 342
Rounded (decimal 中的类), 343
rounds (sys.float_info 属性), 1795
row_factory (sqlite3.Connection 属性), 510
row_factory (sqlite3.Cursor 属性), 512
rowcount (sqlite3.Cursor 属性), 512
Row (sqlite3 中的类), 513
RPAR() (在 token 模块中), 1981
rpartition() (bytearray 方法), 65
rpartition() (bytes 方法), 65
rpartition() (str 方法), 54
rpc_paths (xmlrpc.server.SimpleXMLRPCRequestHandler 属性), 1418
rpop() (poplib.POP3 方法), 1366
RS() (在 curses.ascii 模块中), 881
rset() (poplib.POP3 方法), 1367
rshift() (在 operator 模块中), 404
RShift (ast 中的类), 1949
rsplit() (bytearray 方法), 67
rsplit() (bytes 方法), 67
rsplit() (str 方法), 54
RSQB() (在 token 模块中), 1981
rstrip() (bytearray 方法), 67
rstrip() (bytes 方法), 67
rstrip() (str 方法), 54
rt() (在 turtle 模块中), 1464
RTLD_DEEPBIND() (在 os 模块中), 676
RTLD_GLOBAL() (在 os 模块中), 676
RTLD_LAZY() (在 os 模块中), 676
RTLD_LOCAL() (在 os 模块中), 676
RTLD_NODELETE() (在 os 模块中), 676
RTLD_NOLOAD() (在 os 模块中), 676
RTLD_NOW() (在 os 模块中), 676
ruler (cmd.Cmd 属性), 1493
run (*pdb command*), 1745
run() (asyncio.Runner 方法), 983

run() (bdb.Bdb 方法), 1735
run() (contextvars.Context 方法), 976
run() (doctest.DocTestRunner 方法), 1618
run() (multiprocessing.Process 方法), 905
run() (pdb.Pdb 方法), 1740
run() (profile.Profile 方法), 1750
run() (sched.scheduler 方法), 971
run() (threading.Thread 方法), 889
run() (trace.Trace 方法), 1759
run() (unittest.IsolatedAsyncioTestCase 方法), 1640
run() (unittest.TestCase 方法), 1632
run() (unittest.TestSuite 方法), 1641
run() (unittest.TextTestRunner 方法), 1647
run() (wsgiref.handlers.BaseHandler 方法), 1317
run() (在 asyncio 模块中), 982
run() (在 pdb 模块中), 1739
run() (在 profile 模块中), 1749
run() (在 subprocess 模块中), 951
run_coroutine_threadsafe() (在 asyncio 模块中), 998
run_docstring_examples() (在 doctest 模块中), 1612
run_forever() (asyncio.loop 方法), 1024
run_in_executor() (asyncio.loop 方法), 1036
run_in_subinterp() (在 test.support 模块中), 1717
run_module() (在 runpy 模块中), 1907
run_path() (在 runpy 模块中), 1908
run_python_until_end() (在 test.support.script_helper 模块中), 1719
run_script() (modulefinder.ModuleFinder 方法), 1906
run_until_complete() (asyncio.loop 方法), 1024
run_with_locale() (在 test.support 模块中), 1715
run_with_tz() (在 test.support 模块中), 1715
runcall() (bdb.Bdb 方法), 1735
runcall() (pdb.Pdb 方法), 1740
runcall() (profile.Profile 方法), 1750
runcall() (在 pdb 模块中), 1739
runcode() (code.InteractiveInterpreter 方法), 1898
runctx() (bdb.Bdb 方法), 1735
runctx() (profile.Profile 方法), 1750
runctx() (trace.Trace 方法), 1760
runctx() (在 profile 模块中), 1749
runeval() (bdb.Bdb 方法), 1735
runeval() (pdb.Pdb 方法), 1740
runeval() (在 pdb 模块中), 1739
runfunc() (trace.Trace 方法), 1760
Runner (asyncio 中的类), 983
running() (concurrent.futures.Future 方法), 949
runpy
    module, 1907
runsource() (code.InteractiveInterpreter 方法), 1898
runtime_checkable() (在 typing 模块中), 1579
RuntimeError, 103
RuntimeWarning, 107
runtime (sys._emscripten_info 属性), 1791
RUSAGE_BOTH() (在 resource 模块中), 2046
RUSAGE_CHILDREN() (在 resource 模块中), 2045
RUSAGE_SELF() (在 resource 模块中), 2045
RUSAGE_THREAD() (在 resource 模块中), 2046
RWF_APPEND() (在 os 模块中), 632
RWF_DSYNC() (在 os 模块中), 632
RWF_HIPRI() (在 os 模块中), 631
RWF_NOWAIT() (在 os 模块中), 631
RWF_SYNC() (在 os 模块中), 632

S

-s
    calendar 命令行选项, 238
    compileall 命令行选项, 1993
    cProfile 命令行选项, 1747
    idle 命令行选项, 1544
    timeit 命令行选项, 1756
    trace 命令行选项, 1759
    unittest-discover 命令行选项, 1626
S() (在 re 模块中), 130
S_ENFMT() (在 stat 模块中), 444
S_IEXEC() (在 stat 模块中), 444
S_IFBLK() (在 stat 模块中), 443
S_IFCHR() (在 stat 模块中), 443
S_IFDIR() (在 stat 模块中), 443
S_IFDOOR() (在 stat 模块中), 443
S_IFIFO() (在 stat 模块中), 443
S_IFLNK() (在 stat 模块中), 443
S_IFMT() (在 stat 模块中), 441
S_IFPORT() (在 stat 模块中), 443
S_IFREG() (在 stat 模块中), 443
S_IFSOCK() (在 stat 模块中), 443
S_IFWHT() (在 stat 模块中), 443
S_IMODE() (在 stat 模块中), 441
S_IREAD() (在 stat 模块中), 444
S_IRGRP() (在 stat 模块中), 444
S_IROTH() (在 stat 模块中), 444
S_IRUSR() (在 stat 模块中), 444
S_IRWXG() (在 stat 模块中), 444
S_IRWXO() (在 stat 模块中), 444
S_IRWXU() (在 stat 模块中), 443
S_ISBLK() (在 stat 模块中), 441
S_ISCHR() (在 stat 模块中), 441
S_ISDIR() (在 stat 模块中), 441
S_ISDOOR() (在 stat 模块中), 441
S_ISFIFO() (在 stat 模块中), 441
S_ISGID() (在 stat 模块中), 443
S_ISLNK() (在 stat 模块中), 441
S_ISPORT() (在 stat 模块中), 441
S_ISREG() (在 stat 模块中), 441

S_ISSOCK() (在 stat 模块中), 441
S_ISUID() (在 stat 模块中), 443
S_ISVTX() (在 stat 模块中), 443
S_ISWHT() (在 stat 模块中), 441
S_IWGRP() (在 stat 模块中), 444
S_IWOTH() (在 stat 模块中), 444
S_IWRITE() (在 stat 模块中), 444
S_IWUSR() (在 stat 模块中), 444
S_IXGRP() (在 stat 模块中), 444
S_IXOTH() (在 stat 模块中), 444
S_IXUSR() (在 stat 模块中), 444
safe_path (sys.flags 属性), 1793
safe_substitute() (string.Template 方法), 120
SafeChildWatcher (asyncio 中的类), 1063
saferepr() (在 pprint 模块中), 284
SafeUUID (uuid 中的类), 1380
safe (uuid.SafeUUID 属性), 1380
same_files (filecmp.dircmp 属性), 448
same_quantum() (decimal.Context 方法), 341
same_quantum() (decimal.Decimal 方法), 334
samefile() (pathlib.Path 方法), 426
samefile() (在 os.path 模块中), 439
SameFileError, 458
sameopenfile() (在 os.path 模块中), 439
samesite (http.cookies.Morsel 属性), 1399
samestat() (在 os.path 模块中), 439
sample() (在 random 模块中), 356
samples() (statistics.NormalDist 方法), 373
SATURDAY() (在 calendar 模块中), 235
save() (http.cookiejar.FileCookieJar 方法), 1404
save() (test.support.SaveSignals 方法), 1718
SaveAs (tkinter.filedialog 中的类), 1516
SAVEDCWD() (在 test.support.os_helper 模块中), 1721
SaveFileDialog (tkinter.filedialog 中的类), 1517
SaveKey() (在 winreg 模块中), 2025
SaveSignals (test.support 中的类), 1718
savetty() (在 curses 模块中), 858
SAX2DOM (xml.dom.pulldom 中的类), 1287
SAXException, 1289
SAXNotRecognizedException, 1289
SAXNotSupportedException, 1289
SAXParseException, 1289
scaleb() (decimal.Context 方法), 341
scaleb() (decimal.Decimal 方法), 334
scandir() (在 os 模块中), 645
scanf (*C* 函数), 139
sched
    module, 969
SCHED_BATCH() (在 os 模块中), 673
SCHED_FIFO() (在 os 模块中), 673
sched_get_priority_max() (在 os 模块中), 674
sched_get_priority_min() (在 os 模块中), 674
sched_getaffinity() (在 os 模块中), 674
sched_getparam() (在 os 模块中), 674
sched_getscheduler() (在 os 模块中), 674
SCHED_IDLE() (在 os 模块中), 673
SCHED_OTHER() (在 os 模块中), 673
sched_param (os 中的类), 674
sched_priority (os.sched_param 属性), 674
SCHED_RESET_ON_FORK() (在 os 模块中), 673
SCHED_RR() (在 os 模块中), 673
sched_rr_get_interval() (在 os 模块中), 674
sched_setaffinity() (在 os 模块中), 674
sched_setparam() (在 os 模块中), 674
sched_setscheduler() (在 os 模块中), 674
SCHED_SPORADIC() (在 os 模块中), 673
sched_yield() (在 os 模块中), 674
scheduler (sched 中的类), 969
SCM_CREDS2() (在 socket 模块中), 1083
scope_id (ipaddress.IPv6Address 属性), 1426
screensize() (在 turtle 模块中), 1480
Screen (turtle 中的类), 1486
script_from_examples() (在 doctest 模块中), 1620
scroll() (curses.window 方法), 865
ScrolledCanvas (turtle 中的类), 1486
ScrolledText (tkinter.scrolledtext 中的类), 1519
scrollok() (curses.window 方法), 865
scrypt() (在 hashlib 模块中), 603
seal() (在 unittest.mock 模块中), 1687
search() (imaplib.IMAP4 方法), 1371
search() (re.Pattern 方法), 134
search() (在 re 模块中), 130
seconds (datetime.timedelta 属性), 194
second (datetime.datetime 属性), 205
second (datetime.time 属性), 212
secrets
    module, 611
SECTCRE (configparser.ConfigParser 属性), 585
sections() (configparser.ConfigParser 方法), 589
Secure Sockets Layer, 1101
secure (http.cookiejar.Cookie 属性), 1408
secure (http.cookies.Morsel 属性), 1399
security_level (ssl.SSLContext 属性), 1123
see() (tkinter.ttk.Treeview 方法), 1532
seed() (random.Random 方法), 358
seed() (在 random 模块中), 355
seed_bits (sys.hash_info 属性), 1799
seek() (io.IOBase 方法), 681
seek() (io.TextIOBase 方法), 687
seek() (io.TextIOWrapper 方法), 688
seek() (mmap.mmap 方法), 1155
seek() (sqlite3.Blob 方法), 513
SEEK_CUR() (在 os 模块中), 628
SEEK_DATA() (在 os 模块中), 628
SEEK_END() (在 os 模块中), 628
SEEK_HOLE() (在 os 模块中), 628

SEEK_SET() (在 os 模块中), 628
seekable() (bz2.BZ2File 方法), 533
seekable() (io.IOBase 方法), 682
seekable() (mmap.mmap 方法), 1155
select
    module, 1134
select() (imaplib.IMAP4 方法), 1372
select() (selectors.BaseSelector 方法), 1142
select() (tkinter.ttk.Notebook 方法), 1527
select() (在 select 模块中), 1135
selected_alpn_protocol() (ssl.SSLSocket 方法), 1116
selected_npn_protocol() (ssl.SSLSocket 方法), 1116
selection() (tkinter.ttk.Treeview 方法), 1532
selection_add() (tkinter.ttk.Treeview 方法), 1532
selection_remove() (tkinter.ttk.Treeview 方法), 1532
selection_set() (tkinter.ttk.Treeview 方法), 1532
selection_toggle() (tkinter.ttk.Treeview 方法), 1532
SelectorEventLoop (asyncio 中的类), 1042
SelectorKey (selectors 中的类), 1141
selectors
    module, 1140
selector (urllib.request.Request 属性), 1326
SelectSelector (selectors 中的类), 1142
Self() (在 typing 模块中), 1562
Semaphore() (multiprocessing.managers.SyncManager 方法), 920
Semaphore (asyncio 中的类), 1012
Semaphore (multiprocessing 中的类), 915
Semaphore (threading 中的类), 894
SEMI() (在 token 模块中), 1981
SEND (*opcode*), 2014
send() (http.client.HTTPConnection 方法), 1356
send() (imaplib.IMAP4 方法), 1372
send() (logging.handlers.DatagramHandler 方法), 735
send() (logging.handlers.SocketHandler 方法), 734
send() (multiprocessing.connection.Connection 方法), 912
send() (socket.socket 方法), 1095
send_bytes() (multiprocessing.connection.Connection 方法), 912
send_error() (http.server.BaseHTTPRequestHandler 方法), 1394
send_fds() (在 socket 模块中), 1090
send_header() (http.server.BaseHTTPRequestHandler 方法), 1394
send_message() (smtplib.SMTP 方法), 1379
send_response() (http.server.BaseHTTPRequestHandler 方法), 1394
send_response_only() (http.server.BaseHTTPRequestHandler 方法), 1394
send_signal() (asyncio.subprocess.Process 方法), 1017
send_signal() (asyncio.SubprocessTransport 方法), 1052
send_signal() (subprocess.Popen 方法), 960
sendall() (socket.socket 方法), 1095
sendcmd() (ftplib.FTP 方法), 1361
sendfile() (asyncio.loop 方法), 1032
sendfile() (socket.socket 方法), 1096
sendfile() (wsgiref.handlers.BaseHandler 方法), 1319
sendfile() (在 os 模块中), 633
SendfileNotAvailableError, 1022
sendmail() (smtplib.SMTP 方法), 1378
sendmsg() (socket.socket 方法), 1095
sendmsg_afalg() (socket.socket 方法), 1096
sendto() (asyncio.DatagramTransport 方法), 1051
sendto() (socket.socket 方法), 1095
sentinel() (在 unittest.mock 模块中), 1680
sentinel (multiprocessing.Process 属性), 906
sep() (在 os 模块中), 675
SEPTEMBER() (在 calendar 模块中), 236
sequence
    object -- 对象, 43
    types, immutable -- 不可变对象, 45
    types, mutable -- 可变对象, 45
    types, 运算目标, 43, 45
    迭代, 42
sequence -- 序列, **2075**
SequenceMatcher (difflib 中的类), 147
Sequence (collections.abc 中的类), 257
Sequence (typing 中的类), 1595
serialize() (sqlite3.Connection 方法), 508
serializing
    objects, 469
serve_forever() (asyncio.Server 方法), 1041
serve_forever() (socketserver.BaseServer 方法), 1386
server_activate() (socketserver.BaseServer 方法), 1388
server_address (socketserver.BaseServer 属性), 1387
server_bind() (socketserver.BaseServer 方法), 1388
server_close() (socketserver.BaseServer 方法), 1387
server_hostname (ssl.SSLSocket 属性), 1117
server_side (ssl.SSLSocket 属性), 1116
server_software (wsgiref.handlers.BaseHandler

属性), 1318
server_version(http.server.BaseHTTPRequestHandler 属性), 1393
server_version(http.server.SimpleHTTPRequestHandler 属性), 1395
ServerProxy (xmlrpc.client 中的类), 1410
Server (asyncio 中的类), 1040
server(http.server.BaseHTTPRequestHandler 属性), 1393
server(socketserver.BaseRequestHandler 属性), 1388
service_actions() (socketserver.BaseServer 方法), 1386
session_reused (ssl.SSLSocket 属性), 1117
session_stats() (ssl.SSLContext 方法), 1122
session (ssl.SSLSocket 属性), 1117
set
    object -- 对象, 81
set comprehension -- 集合推导式, 2075
set() (asyncio.Event 方法), 1011
set() (configparser.ConfigParser 方法), 590
set() (configparser.RawConfigParser 方法), 591
set() (contextvars.ContextVar 方法), 975
set() (http.cookies.Morsel 方法), 1400
set() (test.support.os_helper.EnvironmentVarGuard 方法), 1722
set() (threading.Event 方法), 895
set() (tkinter.ttk.Combobox 方法), 1524
set() (tkinter.ttk.Spinbox 方法), 1525
set() (tkinter.ttk.Treeview 方法), 1532
set() (xml.etree.ElementTree.Element 方法), 1267
SET_ADD (*opcode*), 2004
set_allowed_domains() (http.cookiejar.DefaultCookiePolicy 方法), 1407
set_alpn_protocols() (ssl.SSLContext 方法), 1120
set_app() (wsgiref.simple_server.WSGIServer 方法), 1315
set_asyncgen_hooks() (在 sys 模块中), 1806
set_authorizer() (sqlite3.Connection 方法), 504
set_auto_history() (在 readline 模块中), 162
set_blocked_domains() (http.cookiejar.DefaultCookiePolicy 方法), 1406
set_blocking() (在 os 模块中), 633
set_boundary() (email.message.EmailMessage 方法), 1164
set_boundary() (email.message.Message 方法), 1199
set_break() (bdb.Bdb 方法), 1734
set_charset() (email.message.Message 方法), 1196
set_child_watcher() (asyncio.AbstractEventLoopPolicy 方法), 1061
set_child_watcher() (在 asyncio 模块中), 1062
set_children() (tkinter.ttk.Treeview 方法), 1530
set_ciphers() (ssl.SSLContext 方法), 1120
set_completer() (在 readline 模块中), 163
set_completer_delims() (在 readline 模块中), 163
set_completion_display_matches_hook() (在 readline 模块中), 163
set_content() (email.contentmanager.ContentManager 方法), 1186
set_content() (email.message.EmailMessage 方法), 1166
set_content() (在 email.contentmanager 模块中), 1187
set_continue() (bdb.Bdb 方法), 1734
set_cookie() (http.cookiejar.CookieJar 方法), 1404
set_cookie_if_ok() (http.cookiejar.CookieJar 方法), 1404
set_coroutine_origin_tracking_depth() (在 sys 模块中), 1806
set_data() (importlib.abc.SourceLoader 方法), 1916
set_data() (importlib.machinery.SourceFileLoader 方法), 1920
set_date() (mailbox.MaildirMessage 方法), 1231
set_debug() (asyncio.loop 方法), 1038
set_debug() (在 gc 模块中), 1872
set_debuglevel() (ftplib.FTP 方法), 1360
set_debuglevel() (http.client.HTTPConnection 方法), 1355
set_debuglevel() (poplib.POP3 方法), 1366
set_debuglevel() (smtplib.SMTP 方法), 1376
set_default_executor() (asyncio.loop 方法), 1037
set_default_type() (email.message.EmailMessage 方法), 1163
set_default_type() (email.message.Message 方法), 1198
set_default_verify_paths() (ssl.SSLContext 方法), 1119
set_defaults() (argparse.ArgumentParser 方法), 809
set_defaults() (optparse.OptionParser 方法), 844
set_ecdh_curve() (ssl.SSLContext 方法), 1121
set_errno() (在 ctypes 模块中), 778
set_error_mode() (在 msvcrt 模块中), 2020

set_escdelay() (在 curses 模块中), 859
set_event_loop() (asyncio.AbstractEventLoopPolicy 方法), 1061
set_event_loop() (在 asyncio 模块中), 1023
set_event_loop_policy() (在 asyncio 模块中), 1061
set_events() (在 sys.monitoring 模块中), 1813
set_exception() (asyncio.Future 方法), 1046
set_exception() (concurrent.futures.Future 方法), 949
set_exception_handler() (asyncio.loop 方法), 1037
set_executable() (在 multiprocessing 模块中), 911
set_filter() (tkinter.filedialog.FileDialog 方法), 1516
set_flags() (mailbox.Maildir 方法), 1225
set_flags() (mailbox.MaildirMessage 方法), 1231
set_flags() (mailbox.mboxMessage 方法), 1233
set_flags() (mailbox.MMDFMessage 方法), 1236
set_forkserver_preload() (在 multiprocessing 模块中), 911
set_from() (mailbox.mboxMessage 方法), 1232
set_from() (mailbox.MMDFMessage 方法), 1236
SET_FUNCTION_ATTRIBUTE (*opcode*), 2012
set_handle_inheritable() (在 os 模块中), 636
set_history_length() (在 readline 模块中), 162
set_info() (mailbox.Maildir 方法), 1226
set_info() (mailbox.MaildirMessage 方法), 1231
set_inheritable() (socket.socket 方法), 1096
set_inheritable() (在 os 模块中), 636
set_int_max_str_digits() (在 sys 模块中), 1804
set_labels() (mailbox.BabylMessage 方法), 1235
set_last_error() (在 ctypes 模块中), 778
set_local_events() (在 sys.monitoring 模块中), 1813
set_memlimit() (在 test.support 模块中), 1713
set_name() (asyncio.Task 方法), 1000
set_next() (bdb.Bdb 方法), 1734
set_nonstandard_attr() (http.cookiejar.Cookie 方法), 1409
set_npn_protocols() (ssl.SSLContext 方法), 1120
set_ok() (http.cookiejar.CookiePolicy 方法), 1405
set_param() (email.message.EmailMessage 方法), 1163
set_param() (email.message.Message 方法), 1199
set_pasv() (ftplib.FTP 方法), 1361
set_payload() (email.message.Message 方法), 1196
set_policy() (http.cookiejar.CookieJar 方法), 1403
set_pre_input_hook() (在 readline 模块中), 163
set_progress_handler() (sqlite3.Connection 方法), 505
set_protocol() (asyncio.BaseTransport 方法), 1050
set_proxy() (urllib.request.Request 方法), 1327
set_psk_client_callback() (ssl.SSLContext 方法), 1124
set_psk_server_callback() (ssl.SSLContext 方法), 1125
set_quit() (bdb.Bdb 方法), 1734
set_result() (asyncio.Future 方法), 1046
set_result() (concurrent.futures.Future 方法), 949
set_return() (bdb.Bdb 方法), 1734
set_running_or_notify_cancel() (concurrent.futures.Future 方法), 949
set_selection() (tkinter.filedialog.FileDialog 方法), 1517
set_seq1() (difflib.SequenceMatcher 方法), 147
set_seq2() (difflib.SequenceMatcher 方法), 147
set_seqs() (difflib.SequenceMatcher 方法), 147
set_sequences() (mailbox.MH 方法), 1228
set_sequences() (mailbox.MHMessage 方法), 1234
set_server_documentation() (xmlrpc.server.DocCGIXMLRPCRequestHandler 方法), 1422
set_server_documentation() (xmlrpc.server.DocXMLRPCServer 方法), 1422
set_server_name() (xmlrpc.server.DocCGIXMLRPCRequestHandler 方法), 1422
set_server_name() (xmlrpc.server.DocXMLRPCServer 方法), 1422
set_server_title() (xmlrpc.server.DocCGIXMLRPCRequestHandler 方法), 1422
set_server_title() (xmlrpc.server.DocXMLRPCServer

方法), 1422
set_servername_callback(ssl.SSLContext 属性), 1121
set_start_method() (在 multiprocessing 模块中), 911
set_startup_hook()(在 readline 模块中), 163
set_step() (bdb.Bdb 方法), 1734
set_subdir() (mailbox.MaildirMessage 方法), 1231
set_tabsize() (在 curses 模块中), 859
set_task_factory() (asyncio.loop 方法), 1027
set_threshold() (在 gc 模块中), 1873
set_trace() (bdb.Bdb 方法), 1734
set_trace() (pdb.Pdb 方法), 1740
set_trace() (在 bdb 模块中), 1735
set_trace() (在 pdb 模块中), 1740
set_trace_callback() (sqlite3.Connection 方法), 505
set_tunnel() (http.client.HTTPConnection 方法), 1355
set_type() (email.message.Message 方法), 1199
set_unittest_reportflags() (在 doctest 模块中), 1614
set_unixfrom() (email.message.EmailMessage 方法), 1161
set_unixfrom() (email.message.Message 方法), 1195
set_until() (bdb.Bdb 方法), 1734
SET_UPDATE (*opcode*), 2008
set_url() (urllib.robotparser.RobotFileParser 方法), 1347
set_usage() (optparse.OptionParser 方法), 844
set_userptr() (curses.panel.Panel 方法), 883
set_visible() (mailbox.BabylMessage 方法), 1235
set_wakeup_fd() (在 signal 模块中), 1149
set_write_buffer_limits() (asyncio.WriteTransport 方法), 1051
setacl() (imaplib.IMAP4 方法), 1372
setannotation() (imaplib.IMAP4 方法), 1372
setattr()
    built-in function, 27
setAttribute() (xml.dom.Element 方法), 1279
setAttributeNode() (xml.dom.Element 方法), 1279
setAttributeNodeNS() (xml.dom.Element 方法), 1279
setAttributeNS() (xml.dom.Element 方法), 1279
SetBase() (xml.parsers.expat.xmlparser 方法), 1301
setblocking() (socket.socket 方法), 1096
setByteStream() (xml.sax.xmlreader.InputSource 方法), 1298
setcbreak() (在 tty 模块中), 2037
setCharacterStream() (xml.sax.xmlreader.InputSource 方法), 1299
setcomptype() (wave.Wave_write 方法), 1439
SetComp (ast 中的类), 1952
setconfig() (sqlite3.Connection 方法), 508
setContentHandler() (xml.sax.xmlreader.XMLReader 方法), 1297
setcontext() (在 decimal 模块中), 336
setDaemon() (threading.Thread 方法), 890
setdefault() (dict 方法), 85
setdefault() (http.cookies.Morsel 方法), 1400
setdefaulttimeout()(在 socket 模块中), 1089
setdlopenflags() (在 sys 模块中), 1804
setDocumentLocator() (xml.sax.handler.ContentHandler 方法), 1292
setDTDHandler() (xml.sax.xmlreader.XMLReader 方法), 1297
setegid() (在 os 模块中), 621
setEncoding() (xml.sax.xmlreader.InputSource 方法), 1298
setEntityResolver() (xml.sax.xmlreader.XMLReader 方法), 1297
setErrorHandler() (xml.sax.xmlreader.XMLReader 方法), 1297
seteuid() (在 os 模块中), 621
setFeature() (xml.sax.xmlreader.XMLReader 方法), 1297
setfirstweekday() (calendar.Calendar 方法), 231
setfirstweekday()(在 calendar 模块中), 234
setFormatter() (logging.Handler 方法), 707
setframerate() (wave.Wave_write 方法), 1439
setgid() (在 os 模块中), 621
setgroups() (在 os 模块中), 621
seth() (在 turtle 模块中), 1466
setheading() (在 turtle 模块中), 1466
sethostname() (在 socket 模块中), 1089
setinputsizes() (sqlite3.Cursor 方法), 512
setitem() (在 operator 模块中), 405
setitimer() (在 signal 模块中), 1149
setLevel() (logging.Handler 方法), 707
setLevel() (logging.Logger 方法), 704
setlimit() (sqlite3.Connection 方法), 507
setLocale() (xml.sax.xmlreader.XMLReader 方法), 1297
setlocale() (在 locale 模块中), 1448
setLoggerClass() (在 logging 模块中), 716
setlogmask() (在 syslog 模块中), 2047

setLogRecordFactory() (在 logging 模块中), 716
setMaxConns() (urllib.request.CacheFTPHandler 方法), 1332
setmode() (在 msvcrt 模块中), 2019
setName() (threading.Thread 方法), 889
setnchannels() (wave.Wave_write 方法), 1439
setnframes() (wave.Wave_write 方法), 1439
setns() (在 os 模块中), 621
setoutputsize() (sqlite3.Cursor 方法), 512
SetParamEntityParsing() (xml.parsers.expat.xmlparser 方法), 1301
setparams() (wave.Wave_write 方法), 1439
setpassword() (zipfile.ZipFile 方法), 545
setpgid() (在 os 模块中), 622
setpgrp() (在 os 模块中), 622
setpos() (wave.Wave_read 方法), 1438
setpos() (在 turtle 模块中), 1464
setposition() (在 turtle 模块中), 1464
setpriority() (在 os 模块中), 622
setprofile() (在 sys 模块中), 1804
setprofile() (在 threading 模块中), 887
setprofile_all_threads() (在 threading 模块中), 887
setProperty() (xml.sax.xmlreader.XMLReader 方法), 1297
setPublicId() (xml.sax.xmlreader.InputSource 方法), 1298
setquota() (imaplib.IMAP4 方法), 1372
setraw() (在 tty 模块中), 2037
setrecursionlimit() (在 sys 模块中), 1804
setregid() (在 os 模块中), 622
SetReparseDeferralEnabled() (xml.parsers.expat.xmlparser 方法), 1301
setresgid() (在 os 模块中), 622
setresuid() (在 os 模块中), 622
setreuid() (在 os 模块中), 622
setrlimit() (在 resource 模块中), 2042
setsampwidth() (wave.Wave_write 方法), 1439
setscrreg() (curses.window 方法), 865
setsid() (在 os 模块中), 622
setsockopt() (socket.socket 方法), 1096
setstate() (codecs.IncrementalDecoder 方法), 180
setstate() (codecs.IncrementalEncoder 方法), 179
setstate() (random.Random 方法), 358
setstate() (在 random 模块中), 355
setStream() (logging.StreamHandler 方法), 729
setswitchinterval() (在 sys 模块中), 1805
setswitchinterval() (在 test.support 模块中), 1713
setSystemId() (xml.sax.xmlreader.InputSource 方法), 1298
setsyx() (在 curses 模块中), 859
setTarget() (logging.handlers.MemoryHandler 方法), 739
settimeout() (socket.socket 方法), 1096
setTimeout() (urllib.request.CacheFTPHandler 方法), 1332
settrace() (在 sys 模块中), 1805
settrace() (在 threading 模块中), 887
settrace_all_threads() (在 threading 模块中), 887
setuid() (在 os 模块中), 622
setundobuffer() (在 turtle 模块中), 1478
--setup
    timeit 命令行选项, 1756
setup() (socketserver.BaseRequestHandler 方法), 1388
setUp() (unittest.TestCase 方法), 1631
setup() (在 turtle 模块中), 1485
SETUP_ANNOTATIONS (*opcode*), 2005
SETUP_CLEANUP (*opcode*), 2015
setup_environ() (wsgiref.handlers.BaseHandler 方法), 1318
SETUP_FINALLY (*opcode*), 2015
setup_python() (venv.EnvBuilder 方法), 1777
setup_scripts() (venv.EnvBuilder 方法), 1777
setup_testing_defaults() (在 wsgiref.util 模块中), 1313
SETUP_WITH (*opcode*), 2015
setUpClass() (unittest.TestCase 方法), 1631
setupterm() (在 curses 模块中), 859
SetValue() (在 winreg 模块中), 2026
SetValueEx() (在 winreg 模块中), 2026
setworldcoordinates() (在 turtle 模块中), 1480
setx() (在 turtle 模块中), 1465
setxattr() (在 os 模块中), 660
sety() (在 turtle 模块中), 1466
Set (ast 中的类), 1947
Set (collections.abc 中的类), 257
Set (typing 中的类), 1592
set (内置类), 81
SF_APPEND() (在 stat 模块中), 445
SF_ARCHIVED() (在 stat 模块中), 445
SF_DATALESS() (在 stat 模块中), 445
SF_FIRMLINK() (在 stat 模块中), 445
SF_IMMUTABLE() (在 stat 模块中), 445
SF_MNOWAIT() (在 os 模块中), 633
SF_NOCACHE() (在 os 模块中), 633
SF_NODISKIO() (在 os 模块中), 633
SF_NOUNLINK() (在 stat 模块中), 445
SF_RESTRICTED() (在 stat 模块中), 445
SF_SETTABLE() (在 stat 模块中), 445
SF_SNAPSHOT() (在 stat 模块中), 445
SF_SUPPORTED() (在 stat 模块中), 445

SF_SYNC() (在 os 模块中), 633
SF_SYNTHETIC() (在 stat 模块中), 445
sha1() (在 hashlib 模块中), 600
sha3_224() (在 hashlib 模块中), 600
sha3_256() (在 hashlib 模块中), 601
sha3_384() (在 hashlib 模块中), 601
sha3_512() (在 hashlib 模块中), 601
sha224() (在 hashlib 模块中), 600
sha256() (在 hashlib 模块中), 600
sha384() (在 hashlib 模块中), 600
sha512() (在 hashlib 模块中), 600
shake_128() (在 hashlib 模块中), 602
shake_256() (在 hashlib 模块中), 602
shape() (在 turtle 模块中), 1475
shapesize() (在 turtle 模块中), 1475
shapetransform() (在 turtle 模块中), 1476
shape (memoryview 属性), 80
Shape (turtle 中的类), 1486
share() (socket.socket 方法), 1097
ShareableList() (multiprocessing.managers.SharedMemoryManager 方法), 941
ShareableList(multiprocessing.shared_memory 中的类), 942
shared_ciphers() (ssl.SSLSocket 方法), 1115
shared_memory (sys._emscripten_info 属性), 1791
SharedMemory() (multiprocessing.managers.SharedMemoryManager 方法), 941
SharedMemoryManager (multiprocessing.managers 中的类), 941
SharedMemory(multiprocessing.shared_memory 中的类), 939
shearfactor() (在 turtle 模块中), 1475
Shelf (shelve 中的类), 486
shelve
    module, 484, 487
shield() (在 asyncio 模块中), 992
shift() (decimal.Context 方法), 341
shift() (decimal.Decimal 方法), 334
shift_path_info() (在 wsgiref.util 模块中), 1312
shlex
    module, 1496
shlex (shlex 中的类), 1497
shm(multiprocessing.shared_memory.ShareableList 属性), 943
SHORT_TIMEOUT() (在 test.support 模块中), 1710
shortDescription() (unittest.TestCase 方法), 1639
shorten() (在 textwrap 模块中), 155
shouldFlush() (logging.handlers.BufferingHandler 方法), 738
shouldFlush() (logging.handlers.MemoryHandler 方法), 739
shouldStop (unittest.TestResult 属性), 1644
show() (curses.panel.Panel 方法), 883
show() (tkinter.commondialog.Dialog 方法), 1517
show() (tkinter.messagebox.Message 方法), 1518
show_code() (在 dis 模块中), 1998
show_flag_values() (在 enum 模块中), 305
--show-caches
    dis 命令行选项, 1997
showerror() (在 tkinter.messagebox 模块中), 1518
showinfo() (在 tkinter.messagebox 模块中), 1518
--show-offsets
    dis 命令行选项, 1997
showsyntaxerror() (code.InteractiveInterpreter 方法), 1898
showtraceback() (code.InteractiveInterpreter 方法), 1898
showturtle() (在 turtle 模块中), 1474
showwarning() (在 tkinter.messagebox 模块中), 1518
showwarning() (在 warnings 模块中), 1830
shuffle() (在 random 模块中), 356
SHUT_RD() (在 socket 模块中), 1084
SHUT_RDWR() (在 socket 模块中), 1084
SHUT_WR() (在 socket 模块中), 1084
ShutDown, 972
shutdown() (asyncio.Queue 方法), 1020
shutdown() (concurrent.futures.Executor 方法), 945
shutdown() (imaplib.IMAP4 方法), 1372
shutdown() (multiprocessing.managers.BaseManager 方法), 919
shutdown() (queue.Queue 方法), 973
shutdown() (socketserver.BaseServer 方法), 1386
shutdown() (socket.socket 方法), 1097
shutdown() (在 logging 模块中), 716
shutdown_asyncgens() (asyncio.loop 方法), 1025
shutdown_default_executor() (asyncio.loop 方法), 1025
shutil
    module, 458
SI() (在 curses.ascii 模块中), 880
side_effect (unittest.mock.Mock 属性), 1657
SIG_BLOCK() (在 signal 模块中), 1147
SIG_DFL() (在 signal 模块中), 1145
SIG_IGN() (在 signal 模块中), 1145
SIG_SETMASK() (在 signal 模块中), 1147
SIG_UNBLOCK() (在 signal 模块中), 1147
SIGABRT() (在 signal 模块中), 1145
SIGALRM() (在 signal 模块中), 1145
SIGBREAK() (在 signal 模块中), 1145
SIGBUS() (在 signal 模块中), 1145
SIGCHLD() (在 signal 模块中), 1145

SIGCLD() (在 signal 模块中), 1145
SIGCONT() (在 signal 模块中), 1145
SIGFPE() (在 signal 模块中), 1145
SIGHUP() (在 signal 模块中), 1145
SIGILL() (在 signal 模块中), 1145
SIGINT() (在 signal 模块中), 1145
siginterrupt() (在 signal 模块中), 1149
SIGKILL() (在 signal 模块中), 1146
Sigmasks (signal 中的类), 1144
signal
    module, 980, 1144
signal() (在 signal 模块中), 1149
Signals (signal 中的类), 1144
signature() (在 inspect 模块中), 1881
Signature (inspect 中的类), 1882
signature (inspect.BoundArguments 属性), 1885
sigpending() (在 signal 模块中), 1150
SIGPIPE() (在 signal 模块中), 1146
SIGSEGV() (在 signal 模块中), 1146
SIGSTKFLT() (在 signal 模块中), 1146
SIGTERM() (在 signal 模块中), 1146
sigtimedwait() (在 signal 模块中), 1150
SIGUSR1() (在 signal 模块中), 1146
SIGUSR2() (在 signal 模块中), 1146
sigwait() (在 signal 模块中), 1150
sigwaitinfo() (在 signal 模块中), 1150
SIGWINCH() (在 signal 模块中), 1146
SimpleCookie (http.cookies 中的类), 1398
simplefilter() (在 warnings 模块中), 1831
SimpleHandler (wsgiref.handlers 中的类), 1317
SimpleHTTPRequestHandler (http.server 中的类), 1395
SimpleNamespace (types 中的类), 281
SimpleQueue (multiprocessing 中的类), 909
SimpleQueue (queue 中的类), 972
SimpleXMLRPCRequestHandler (xmlrpc.server 中的类), 1417
SimpleXMLRPCServer (xmlrpc.server 中的类), 1417
SIMPLE (inspect.BufferFlags 属性), 1892
sin() (在 cmath 模块中), 323
sin() (在 math 模块中), 319
single dispatch -- 单分派, **2075**
SingleAddressHeader (email.headerregistry 中的类), 1182
singledispatch() (在 functools 模块中), 398
singledispatchmethod (functools 中的类), 400
sinh() (在 cmath 模块中), 323
sinh() (在 math 模块中), 319
SIO_KEEPALIVE_VALS() (在 socket 模块中), 1082
SIO_LOOPBACK_FAST_PATH() (在 socket 模块中), 1082
SIO_RCVALL() (在 socket 模块中), 1082
site
    module, 1893
site_maps() (urllib.robotparser.RobotFileParser 方法), 1348
sitecustomize
    module, 1894
site-packages
    directory, 1893
site 命令行选项
    --user-base, 1896
    --user-site, 1896
sixtofour (ipaddress.IPv6Address 属性), 1426
size() (ftplib.FTP 方法), 1363
size() (mmap.mmap 方法), 1156
size_diff (tracemalloc.StatisticDiff 属性), 1769
Sized (collections.abc 中的类), 257
Sized (typing 中的类), 1596
sizeof() (在 ctypes 模块中), 778
sizeof_digit (sys.int_info 属性), 1800
size(multiprocessing.shared_memory.SharedMemory 属性), 940
size (struct.Struct 属性), 173
size (tarfile.TarInfo 属性), 559
size (tracemalloc.Statistic 属性), 1768
size (tracemalloc.StatisticDiff 属性), 1769
size (tracemalloc.Trace 属性), 1769
SKIP() (在 doctest 模块中), 1609
skip() (在 unittest 模块中), 1629
skip_if_broken_multiprocessing_synchronize() (在 test.support 模块中), 1717
skip_unless_bind_unix_socket() (在 test.support.socket_helper 模块中), 1719
skip_unless_symlink() (在 test.support.os_helper 模块中), 1723
skip_unless_xattr() (在 test.support.os_helper 模块中), 1723
skipIf() (在 unittest 模块中), 1630
skipinitialspace (csv.Dialect 属性), 574
skippedEntity() (xml.sax.handler.ContentHandler 方法), 1293
skipped (doctest.TestResults 属性), 1617
skipped (unittest.TestResult 属性), 1644
skips (doctest.DocTestRunner 属性), 1618
SkipTest, 1630
skipTest() (unittest.TestCase 方法), 1632
skipUnless() (在 unittest 模块中), 1630
SLASH() (在 token 模块中), 1982
SLASHEQUAL() (在 token 模块中), 1983
sleep() (在 asyncio 模块中), 990
sleep() (在 time 模块中), 693
sleeping_retry() (在 test.support 模块中), 1712
slice -- 切片
    operation, 43
    内置函数, 2012

    赋值, 45
slice -- 切片, **2075**
Slice (ast 中的类), 1952
slice (内置类), 27
slow_callback_duration (asyncio.loop 属性), 1038
SMALLEST() (在 test.support 模块中), 1712
SMTP
    协议, 1374
SMTP() (在 email.policy 模块中), 1178
SMTP_SSL (smtplib 中的类), 1374
SMTPAuthenticationError, 1376
SMTPConnectError, 1375
smtpd
    module, 2058
SMTPDataError, 1375
SMTPException, 1375
SMTPHandler (logging.handlers 中的类), 738
SMTPHeloError, 1375
smtplib
    module, 1374
SMTPNotSupportedError, 1375
SMTPRecipientsRefused, 1375
SMTPResponseException, 1375
SMTPSenderRefused, 1375
SMTPServerDisconnected, 1375
SMTPUTF8() (在 email.policy 模块中), 1178
SMTP (smtplib 中的类), 1374
Snapshot (tracemalloc 中的类), 1767
SND_ALIAS() (在 winsound 模块中), 2030
SND_APPLICATION() (在 winsound 模块中), 2031
SND_ASYNC() (在 winsound 模块中), 2031
SND_FILENAME() (在 winsound 模块中), 2030
SND_LOOP() (在 winsound 模块中), 2030
SND_MEMORY() (在 winsound 模块中), 2030
SND_NODEFAULT() (在 winsound 模块中), 2031
SND_NOSTOP() (在 winsound 模块中), 2031
SND_NOWAIT() (在 winsound 模块中), 2031
SND_PURGE() (在 winsound 模块中), 2031
sndhdr
    module, 2058
sni_callback (ssl.SSLContext 属性), 1120
sniff() (csv.Sniffer 方法), 572
Sniffer (csv 中的类), 572
SO() (在 curses.ascii 模块中), 880
SO_INCOMING_CPU() (在 socket 模块中), 1083
sock_accept() (asyncio.loop 方法), 1034
SOCK_CLOEXEC() (在 socket 模块中), 1080
sock_connect() (asyncio.loop 方法), 1034
SOCK_DGRAM() (在 socket 模块中), 1079
SOCK_MAX_SIZE() (在 test.support 模块中), 1711
SOCK_NONBLOCK() (在 socket 模块中), 1080
SOCK_RAW() (在 socket 模块中), 1079
SOCK_RDM() (在 socket 模块中), 1079
sock_recv() (asyncio.loop 方法), 1033
sock_recv_into() (asyncio.loop 方法), 1033
sock_recvfrom() (asyncio.loop 方法), 1033
sock_recvfrom_into() (asyncio.loop 方法), 1033
sock_sendall() (asyncio.loop 方法), 1034
sock_sendfile() (asyncio.loop 方法), 1034
sock_sendto() (asyncio.loop 方法), 1034
SOCK_SEQPACKET() (在 socket 模块中), 1079
SOCK_STREAM() (在 socket 模块中), 1079
socket
    module, 1075, 1309
    object -- 对象, 1076
socket() (在 *socket* 模块中), 1135
socket() (imaplib.IMAP4 方法), 1372
socket_type (socketserver.BaseServer 属性), 1387
SocketHandler (logging.handlers 中的类), 734
socketpair() (在 socket 模块中), 1084
socketserver
    module, 1384
sockets (asyncio.Server 属性), 1042
SocketType() (在 socket 模块中), 1086
socket (socket 中的类), 1084
socket (socketserver.BaseServer 属性), 1387
soft deprecated -- 软弃用, **2076**
SOFT_KEYWORD() (在 token 模块中), 1981
softkwlist() (在 keyword 模块中), 1984
SOH() (在 curses.ascii 模块中), 879
SOL_ALG() (在 socket 模块中), 1082
SOL_RDS() (在 socket 模块中), 1082
SOMAXCONN() (在 socket 模块中), 1080
sort() (imaplib.IMAP4 方法), 1372
sort() (list 方法), 46
sort_stats() (pstats.Stats 方法), 1750
sortdict() (在 test.support 模块中), 1712
sorted()
    built-in function, 27
--sort-keys
    json.tool 命令行选项, 1221
sortTestMethodsUsing (unittest.TestLoader 属性), 1643
source (*pdb command*), 1743
SOURCE_DATE_EPOCH, 1991, 1993
source_from_cache() (在 importlib.util 模块中), 1923
source_hash() (在 importlib.util 模块中), 1924
SOURCE_SUFFIXES() (在 importlib.machinery 模块中), 1918
source_to_code() (importlib.abc.InspectLoader 静态方法), 1914
SourceFileLoader (importlib.machinery 中的类), 1920
sourcehook() (shlex.shlex 方法), 1497
SourcelessFileLoader (importlib.machinery 中的类), 1920
SourceLoader (importlib.abc 中的类), 1915
source (doctest.Example 属性), 1615
source (shlex.shlex 属性), 1499
SP() (在 curses.ascii 模块中), 881

space
    使用 printf 风格的格式化, 59, 73
    在字符串格式化中, 115
--spacing
    calendar 命令行选项, 238
span() (re.Match 方法), 137
sparse (tarfile.TarInfo 属性), 560
spawn() (在 pty 模块中), 2038
spawn_python() (在 test.support.script_helper 模块中), 1719
spawnl() (在 os 模块中), 667
spawnle() (在 os 模块中), 667
spawnlp() (在 os 模块中), 667
spawnlpe() (在 os 模块中), 667
spawnv() (在 os 模块中), 667
spawnve() (在 os 模块中), 667
spawnvp() (在 os 模块中), 667
spawnvpe() (在 os 模块中), 667
spec_from_file_location() (在 importlib.util 模块中), 1924
spec_from_loader() (在 importlib.util 模块中), 1924
special method -- 特殊方法, **2076**
SpecialFileError, 554
specified_attributes (xml.parsers.expat.xmlparser 属性), 1302
speed() (在 turtle 模块中), 1468
Spinbox (tkinter.ttk 中的类), 1525
splice() (在 os 模块中), 634
SPLICE_F_MORE() (在 os 模块中), 634
SPLICE_F_MOVE() (在 os 模块中), 634
SPLICE_F_NONBLOCK() (在 os 模块中), 634
split() (BaseExceptionGroup 方法), 108
split() (bytearray 方法), 67
split() (bytes 方法), 67
split() (re.Pattern 方法), 134
split() (str 方法), 54
split() (在 os.path 模块中), 439
split() (在 re 模块中), 131
split() (在 shlex 模块中), 1496
splitdrive() (在 os.path 模块中), 439
splitext() (在 os.path 模块中), 440
splitlines() (bytearray 方法), 70
splitlines() (bytes 方法), 70
splitlines() (str 方法), 55
SplitResultBytes(urllib.parse 中的类), 1345
SplitResult (urllib.parse 中的类), 1344
splitroot() (在 os.path 模块中), 439
SpooledTemporaryFile (tempfile 中的类), 450
sprintf 风格的格式化, 58, 72
spwd
    module, 2058
sqlite3
    module, 494
SQLITE_DBCONFIG_DEFENSIVE() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_DQS_DDL() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_DQS_DML() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_ENABLE_FKEY() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_ENABLE_QPSG() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_ENABLE_TRIGGER() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_ENABLE_VIEW() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_LEGACY_ALTER_TABLE() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_LEGACY_FILE_FORMAT() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_RESET_DATABASE() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_TRIGGER_EQP() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_TRUSTED_SCHEMA() (在 sqlite3 模块中), 500
SQLITE_DBCONFIG_WRITABLE_SCHEMA() (在 sqlite3 模块中), 500
SQLITE_DENY() (在 sqlite3 模块中), 499
sqlite_errorcode (sqlite3.Error 属性), 514
sqlite_errorname (sqlite3.Error 属性), 514
SQLITE_IGNORE() (在 sqlite3 模块中), 499
SQLITE_OK() (在 sqlite3 模块中), 499
sqlite_version() (在 sqlite3 模块中), 499
sqlite_version_info() (在 sqlite3 模块中), 499
sqrt() (decimal.Context 方法), 341
sqrt() (decimal.Decimal 方法), 334
sqrt() (在 cmath 模块中), 323
sqrt() (在 math 模块中), 317
SSL, 1101
ssl
    module, 1101
ssl_version (ftplib.FTP_TLS 属性), 1364
SSLCertVerificationError, 1104
SSLContext (ssl 中的类), 1117
SSLEOFError, 1104
SSLError, 1104
SSLErrorNumber (ssl 中的类), 1112
SSLKEYLOGFILE, 1103
sslobject_class (ssl.SSLContext 属性), 1122
SSLObject (ssl 中的类), 1130
SSLSession (ssl 中的类), 1132
sslsocket_class (ssl.SSLContext 属性), 1122
SSLSocket (ssl 中的类), 1113
SSLSyscallError, 1104

SSLv3 (ssl.TLSVersion 属性), 1113
SSLWantReadError, 1104
SSLWantWriteError, 1104
SSLZeroReturnError, 1104
st() (在 turtle 模块中), 1474
ST_ATIME() (在 stat 模块中), 442
st_atime_ns (os.stat_result 属性), 649
st_atime (os.stat_result 属性), 648
st_birthtime_ns (os.stat_result 属性), 649
st_birthtime (os.stat_result 属性), 649
st_blksize (os.stat_result 属性), 649
st_blocks (os.stat_result 属性), 649
st_creator (os.stat_result 属性), 650
ST_CTIME() (在 stat 模块中), 442
st_ctime_ns (os.stat_result 属性), 649
st_ctime (os.stat_result 属性), 648
ST_DEV() (在 stat 模块中), 442
st_dev (os.stat_result 属性), 648
st_file_attributes (os.stat_result 属性), 650
st_flags (os.stat_result 属性), 649
st_fstype (os.stat_result 属性), 650
st_gen (os.stat_result 属性), 649
ST_GID() (在 stat 模块中), 442
st_gid (os.stat_result 属性), 648
ST_INO() (在 stat 模块中), 442
st_ino (os.stat_result 属性), 648
ST_MODE() (在 stat 模块中), 442
st_mode (os.stat_result 属性), 648
ST_MTIME() (在 stat 模块中), 442
st_mtime_ns (os.stat_result 属性), 649
st_mtime (os.stat_result 属性), 648
ST_NLINK() (在 stat 模块中), 442
st_nlink (os.stat_result 属性), 648
st_rdev (os.stat_result 属性), 649
st_reparse_tag (os.stat_result 属性), 650
st_rsize (os.stat_result 属性), 650
ST_SIZE() (在 stat 模块中), 442
st_size (os.stat_result 属性), 648
st_type (os.stat_result 属性), 650
ST_UID() (在 stat 模块中), 442
st_uid (os.stat_result 属性), 648
stack() (在 inspect 模块中), 1889
stack_effect() (在 dis 模块中), 2000
stack_size() (在 _thread 模块中), 979
stack_size() (在 threading 模块中), 887
stackable
    streams, 173
StackSummary (traceback 中的类), 1866
stack (traceback.TracebackException 属性), 1865
stamp() (在 turtle 模块中), 1467
standard_b64decode() (在 base64 模块中), 1242
standard_b64encode() (在 base64 模块中), 1242
standend() (curses.window 方法), 865
standout() (curses.window 方法), 866
STAR() (在 token 模块中), 1982
STAREQUAL() (在 token 模块中), 1983
starmap() (multiprocessing.pool.Pool 方法), 925
starmap() (在 itertools 模块中), 386
starmap_async() (multiprocessing.pool.Pool 方法), 926
Starred (ast 中的类), 1948
start() (logging.handlers.QueueListener 方法), 741
start() (multiprocessing.managers.BaseManager 方法), 918
start() (multiprocessing.Process 方法), 905
start() (re.Match 方法), 137
start() (threading.Thread 方法), 889
start() (tkinter.ttk.Progressbar 方法), 1527
start() (xml.etree.ElementTree.TreeBuilder 方法), 1270
start() (在 tracemalloc 模块中), 1765
start_color() (在 curses 模块中), 859
start_new_thread() (在 _thread 模块中), 978
start_ns() (xml.etree.ElementTree.TreeBuilder 方法), 1271
start_server() (在 asyncio 模块中), 1003
start_serving() (asyncio.Server 方法), 1041
start_threads() (在 test.support.threading_helper 模块中), 1720
start_tls() (asyncio.loop 方法), 1032
start_tls() (asyncio.StreamWriter 方法), 1006
start_unix_server() (在 asyncio 模块中), 1004
StartBoundaryNotFoundDefect, 1179
startCDATA() (xml.sax.handler.LexicalHandler 方法), 1294
StartCdataSectionHandler() (xml.parsers.expat.xmlparser 方法), 1304
--start-directory
    unittest-discover 命令行选项, 1626
StartDoctypeDeclHandler() (xml.parsers.expat.xmlparser 方法), 1303
startDocument() (xml.sax.handler.ContentHandler 方法), 1292
startDTD() (xml.sax.handler.LexicalHandler 方法), 1294
startElement() (xml.sax.handler.ContentHandler 方法), 1292
StartElementHandler() (xml.parsers.expat.xmlparser 方法), 1303
startElementNS() (xml.sax.handler.ContentHandler 方法), 1293

STARTF_FORCEOFFFEEDBACK() (在 subprocess 模块中), 962
STARTF_FORCEONFEEDBACK() (在 subprocess 模块中), 962
STARTF_USESHOWWINDOW() (在 subprocess 模块中), 962
STARTF_USESTDHANDLES() (在 subprocess 模块中), 962
startfile() (在 os 模块中), 668
StartNamespaceDeclHandler() (xml.parsers.expat.xmlparser 方法), 1304
startPrefixMapping() (xml.sax.handler.ContentHandler 方法), 1292
StartResponse (wsgiref.types 中的类), 1320
startswith() (bytearray 方法), 65
startswith() (bytes 方法), 65
startswith() (str 方法), 55
startTest() (unittest.TestResult 方法), 1645
startTestRun() (unittest.TestResult 方法), 1645
starttls() (imaplib.IMAP4 方法), 1372
starttls() (smtplib.SMTP 方法), 1378
STARTUPINFO (subprocess 中的类), 961
start (range 属性), 47
start (slice 属性), 27
start (UnicodeError 属性), 105
stat
    module, 440, 647
stat() (os.DirEntry 方法), 647
stat() (pathlib.Path 方法), 424
stat() (poplib.POP3 方法), 1366
stat() (在 os 模块中), 647
stat_result (os 中的类), 648
state() (tkinter.ttk.Widget 方法), 1523
statement -- 语句
    assert, 101
    del, 45, 83
    except, 99
    if, 35
    import, 31, 1893
    raise, 99
    try, 99
    while, 35
statement -- 语句, **2076**
static type checker -- 静态类型检查器, **2076**
static_order() (graphlib.TopologicalSorter 方法), 307
staticmethod()
    built-in function, 28
StatisticDiff (tracemalloc 中的类), 1768
statistics
    module, 363
statistics() (tracemalloc.Snapshot 方法), 1768
StatisticsError, 373
Statistic (tracemalloc 中的类), 1768
Stats (pstats 中的类), 1750
status() (imaplib.IMAP4 方法), 1372
status (http.client.HTTPResponse 属性), 1357
status (urllib.response.addinfourl 属性), 1338
statvfs() (在 os 模块中), 650
STD_ERROR_HANDLE() (在 subprocess 模块中), 962
STD_INPUT_HANDLE() (在 subprocess 模块中), 962
STD_OUTPUT_HANDLE() (在 subprocess 模块中), 962
stderr() (在 sys 模块中), 1807
stderr (asyncio.subprocess.Process 属性), 1018
stderr (subprocess.CalledProcessError 属性), 953
stderr (subprocess.CompletedProcess 属性), 952
stderr (subprocess.Popen 属性), 961
stderr (subprocess.TimeoutExpired 属性), 953
stdev() (在 statistics 模块中), 370
stdev (statistics.NormalDist 属性), 373
stdin() (在 sys 模块中), 1807
stdin (asyncio.subprocess.Process 属性), 1018
stdin (subprocess.Popen 属性), 961
stdlib_module_names() (在 sys 模块中), 1808
STDOUT() (在 subprocess 模块中), 953
stdout() (在 sys 模块中), 1807
stdout (asyncio.subprocess.Process 属性), 1018
stdout (subprocess.CalledProcessError 属性), 953
stdout (subprocess.CompletedProcess 属性), 952
stdout (subprocess.Popen 属性), 961
stdout (subprocess.TimeoutExpired 属性), 953
stem (pathlib.PurePath 属性), 418
step (*pdb command*), 1742
step() (tkinter.ttk.Progressbar 方法), 1527
step (range 属性), 47
step (slice 属性), 27
stls() (poplib.POP3 方法), 1367
stop() (asyncio.loop 方法), 1024
stop() (logging.handlers.QueueListener 方法), 741
stop() (tkinter.ttk.Progressbar 方法), 1527
stop() (unittest.TestResult 方法), 1645
stop() (在 tracemalloc 模块中), 1766
stop_here() (bdb.Bdb 方法), 1733
STOP_ITERATION (*monitoring event*), 1812

StopAsyncIteration, 104
StopIteration, 103
stopListening() (在 logging.config 模块中), 720
stopTest() (unittest.TestResult 方法), 1645
stopTestRun() (unittest.TestResult 方法), 1645
stop (range 属性), 47
stop (slice 属性), 27
storbinary() (ftplib.FTP 方法), 1361
store() (imaplib.IMAP4 方法), 1372
STORE_ACTIONS (optparse.Option 属性), 849
STORE_ATTR (*opcode*), 2007
STORE_DEREF (*opcode*), 2011
STORE_FAST (*opcode*), 2010
STORE_FAST_LOAD_FAST (*opcode*), 2010
STORE_FAST_STORE_FAST (*opcode*), 2010
STORE_GLOBAL (*opcode*), 2007
STORE_NAME (*opcode*), 2006
STORE_SLICE (*opcode*), 2003
STORE_SUBSCR (*opcode*), 2003
Store (ast 中的类), 1948
storlines() (ftplib.FTP 方法), 1362
str (内置类)
    (另请参阅字符串), 48
str() (在 locale 模块中), 1453
str_digits_check_threshold (sys.int_info 属性), 1800
strcoll() (在 locale 模块中), 1452
StreamError, 554
StreamHandler (logging 中的类), 729
StreamReaderWriter (codecs 中的类), 182
StreamReader (asyncio 中的类), 1004
StreamReader (codecs 中的类), 181
streamreader (codecs.CodecInfo 属性), 174
StreamRecoder (codecs 中的类), 182
StreamRequestHandler (socketserver 中的类), 1388
streams, 173
    stackable, 173
StreamWriter (asyncio 中的类), 1005
StreamWriter (codecs 中的类), 180
streamwriter (codecs.CodecInfo 属性), 174
StrEnum (enum 中的类), 298
strerror() (在 os 模块中), 623
strerror (OSError 属性), 102
strftime() (datetime.date 方法), 200
strftime() (datetime.datetime 方法), 209
strftime() (datetime.time 方法), 214
strftime() (在 time 模块中), 694
strict
    错误处理器名称, 176
strict() (在 email.policy 模块中), 1178
strict_domain(http.cookiejar.DefaultCookiePolicy 属性), 1407
strict_errors() (在 codecs 模块中), 178
strict_ns_domain(http.cookiejar.DefaultCookiePolicy 属性), 1407
strict_ns_set_initial_dollar (http.cookiejar.DefaultCookiePolicy 属性), 1407
strict_ns_set_path (http.cookiejar.DefaultCookiePolicy 属性), 1407
strict_ns_unverifiable (http.cookiejar.DefaultCookiePolicy 属性), 1407
strict_rfc2965_unverifiable (http.cookiejar.DefaultCookiePolicy 属性), 1407
strict (csv.Dialect 属性), 574
STRICT (enum.FlagBoundary 属性), 302
STRIDED_RO (inspect.BufferFlags 属性), 1893
STRIDED (inspect.BufferFlags 属性), 1893
STRIDES (inspect.BufferFlags 属性), 1892
strides (memoryview 属性), 81
string
    format() (内置函数), 16
    module, 111
    object -- 对象, 48
    str (内置类), 48
    str() (内置函数), 28
    插值, printf, 58
    插值字面值, 57
    文本序列类型, 48
    方法, 49
    格式化, printf, 58
    格式化字面值, 57
STRING() (在 token 模块中), 1980
string_at() (在 ctypes 模块中), 778
StringIO (io 中的类), 688
stringprep
    module, 159
string (re.Match 属性), 138
strip() (bytearray 方法), 68
strip() (bytes 方法), 68
strip() (str 方法), 55
strip_dirs() (pstats.Stats 方法), 1750
stripspaces (curses.textpad.Textbox 属性), 879
strong reference -- 强引用, **2076**
strptime() (datetime.datetime 类方法), 204
strptime() (在 time 模块中), 696
strsignal() (在 signal 模块中), 1147
struct
    module, 167, 1097
struct_time (time 中的类), 696
Structure (ctypes 中的类), 782
Struct (struct 中的类), 173
strxfrm() (在 locale 模块中), 1452
sty (内置类), 48
STX() (在 curses.ascii 模块中), 880
Style (tkinter.ttk 中的类), 1533
sub() (re.Pattern 方法), 135

SUB() (在 curses.ascii 模块中), 881
sub() (在 operator 模块中), 404
sub() (在 re 模块中), 132
subdirs (filecmp.dircmp 属性), 448
SubElement() (在 xml.etree.ElementTree 模块中), 1264
subgroup() (BaseExceptionGroup 方法), 108
submit() (concurrent.futures.Executor 方法), 945
submodule_search_locations (importlib.machinery.ModuleSpec 属性), 1922
subn() (re.Pattern 方法), 135
subn() (在 re 模块中), 133
subnet_of() (ipaddress.IPv4Network 方法), 1430
subnet_of() (ipaddress.IPv6Network 方法), 1431
subnets() (ipaddress.IPv4Network 方法), 1429
subnets() (ipaddress.IPv6Network 方法), 1431
Subnormal (decimal 中的类), 343
suboffsets (memoryview 属性), 81
subpad() (curses.window 方法), 866
subprocess
    module, 951
subprocess_exec() (asyncio.loop 方法), 1039
subprocess_shell() (asyncio.loop 方法), 1040
SubprocessError, 953
SubprocessProtocol (asyncio 中的类), 1053
SubprocessTransport (asyncio 中的类), 1049
subscribe() (imaplib.IMAP4 方法), 1373
Subscript (ast 中的类), 1951
subsequent_indent (textwrap.TextWrapper 属性), 157
substitute() (string.Template 方法), 120
subTest() (unittest.TestCase 方法), 1632
subtract() (collections.Counter 方法), 242
subtract() (decimal.Context 方法), 341
subtype (email.headerregistry.ContentTypeHeader 属性), 1183
subwin() (curses.window 方法), 866
Sub (ast 中的类), 1949
successful() (multiprocessing.pool.AsyncResult 方法), 926
suffix_map() (在 mimetypes 模块中), 1240
suffix_map (mimetypes.MimeTypes 属性), 1240
suffixes (pathlib.PurePath 属性), 417
suffix (pathlib.PurePath 属性), 417
suiteClass (unittest.TestLoader 属性), 1644
sum()
    built-in function, 28
summarize() (doctest.DocTestRunner 方法), 1618
summarize_address_range() (在 ipaddress 模块中), 1434
--summary
    trace 命令行选项, 1759
sumprod() (在 math 模块中), 318
sunau
    module, 2058
SUNDAY() (在 calendar 模块中), 235
supernet() (ipaddress.IPv4Network 方法), 1430
supernet() (ipaddress.IPv6Network 方法), 1431
supernet_of() (ipaddress.IPv4Network 方法), 1430
supernet_of() (ipaddress.IPv6Network 方法), 1432
super (pyclbr.Class 属性), 1990
super (内置类), 28
supports_bytes_environ() (在 os 模块中), 623
supports_dir_fd() (在 os 模块中), 651
supports_effective_ids() (在 os 模块中), 651
supports_fd() (在 os 模块中), 651
supports_follow_symlinks() (在 os 模块中), 651
supports_unicode_filenames() (在 os.path 模块中), 440
SupportsAbs (typing 中的类), 1583
SupportsBytes (typing 中的类), 1583
SupportsComplex (typing 中的类), 1584
SupportsFloat (typing 中的类), 1584
SupportsIndex (typing 中的类), 1584
SupportsInt (typing 中的类), 1584
SupportsRound (typing 中的类), 1584
suppress() (在 contextlib 模块中), 1846
SuppressCrashReport (test.support 中的类), 1718
surrogateescape
    错误处理器名称, 176
surrogatepass
    错误处理器名称, 177
SW_HIDE() (在 subprocess 模块中), 962
SWAP (*opcode*), 2002
swap_attr() (在 test.support 模块中), 1714
swap_item() (在 test.support 模块中), 1714
swapcase() (bytearray 方法), 71
swapcase() (bytes 方法), 71
swapcase() (str 方法), 56
SymbolTableType (symtable 中的类), 1976
SymbolTable (symtable 中的类), 1977
Symbol (symtable 中的类), 1978
symlink() (在 os 模块中), 652
symlink_to() (pathlib.Path 方法), 430
symmetric_difference() (frozenset 方法), 82
symmetric_difference_update() (frozenset 方法), 83
symtable
    module, 1976
symtable() (在 symtable 模块中), 1976

SYMTYPE() (在 tarfile 模块中), 554
SYN() (在 curses.ascii 模块中), 881
sync() (dbm.dumb.dumbdbm 方法), 494
sync() (dbm.gnu.gdbm 方法), 492
sync() (shelve.Shelf 方法), 485
sync() (在 os 模块中), 652
syncdown() (curses.window 方法), 866
synchronized() (在 multiprocessing.sharedctypes 模块中), 917
SyncManager (multiprocessing.managers 中的类), 919
syncok() (curses.window 方法), 866
syncup() (curses.window 方法), 866
SyntaxErr, 1281
SyntaxError, 104
SyntaxWarning, 107
sys
    module, 23, 1787
sys_version (http.server.BaseHTTPRequestHandler 属性), 1393
sysconf() (在 os 模块中), 675
sysconf_names() (在 os 模块中), 675
sysconfig
    module, 1815
syslog
    module, 2046
syslog() (在 syslog 模块中), 2046
SysLogHandler (logging.handlers 中的类), 735
sys.monitoring
    module, 1810
system() (在 os 模块中), 669
system() (在 platform 模块中), 743
system_alias() (在 platform 模块中), 743
system_must_validate_cert() (在 test.support 模块中), 1715
SystemError, 104
SystemExit, 104
systemId (xml.dom.DocumentType 属性), 1277
SystemRandom (random 中的类), 358
SystemRandom (secrets 中的类), 612
SystemRoot, 958

T

-T
    trace 命令行选项, 1759
-t
    calendar 命令行选项, 237
    idle 命令行选项, 1544
    tarfile 命令行选项, 565
    trace 命令行选项, 1759
    unittest-discover 命令行选项, 1626
    webbrowser 命令行选项, 1309
    zipfile 命令行选项, 551
T_FMT() (在 locale 模块中), 1450
T_FMT_AMPM() (在 locale 模块中), 1450
--tab
    json.tool 命令行选项, 1221
tab() (tkinter.ttk.Notebook 方法), 1527
TAB() (在 curses.ascii 模块中), 880
TabError, 104
tabnanny
    module, 1988
tabs() (tkinter.ttk.Notebook 方法), 1527
tabsize (textwrap.TextWrapper 属性), 156
tabular
    数据, 569
tag_bind() (tkinter.ttk.Treeview 方法), 1533
tag_configure() (tkinter.ttk.Treeview 方法), 1533
tag_has() (tkinter.ttk.Treeview 方法), 1533
tagName (xml.dom.Element 属性), 1278
tag (xml.etree.ElementTree.Element 属性), 1266
tail (xml.etree.ElementTree.Element 属性), 1266
take_snapshot() (在 tracemalloc 模块中), 1766
takewhile() (在 itertools 模块中), 386
tan() (在 cmath 模块中), 323
tan() (在 math 模块中), 319
tanh() (在 cmath 模块中), 323
tanh() (在 math 模块中), 319
tar_filter() (在 tarfile 模块中), 562
TarError, 554
tarfile
    module, 552
tarfile 命令行选项
    -c, 565
    --create, 565
    -e, 565
    --extract, 565
    --filter, 565
    -l, 565
    --list, 565
    -t, 565
    --test, 565
    -v, 565
    --verbose, 565
TarFile (tarfile 中的类), 556
target (xml.dom.ProcessingInstruction 属性), 1280
TarInfo (tarfile 中的类), 559
tarinfo (tarfile.FilterError 属性), 554
task_done() (asyncio.Queue 方法), 1020
task_done() (multiprocessing.JoinableQueue 方法), 910
task_done() (queue.Queue 方法), 973
TaskGroup (asyncio 中的类), 989
Task (asyncio 中的类), 998
tau() (在 cmath 模块中), 324
tau() (在 math 模块中), 320
tb_locals (unittest.TestResult 属性), 1645
tbreak (*pdb command*), 1741

tcdrain() (在 termios 模块中), 2036
tcflow() (在 termios 模块中), 2036
tcflush() (在 termios 模块中), 2036
tcgetattr() (在 termios 模块中), 2036
tcgetpgrp() (在 os 模块中), 634
tcgetwinsize() (在 termios 模块中), 2036
Tcl() (在 tkinter 模块中), 1503
TCPServer (socketserver 中的类), 1384
TCSADRAIN() (在 termios 模块中), 2036
TCSAFLUSH() (在 termios 模块中), 2036
TCSANOW() (在 termios 模块中), 2036
tcsendbreak() (在 termios 模块中), 2036
tcsetattr() (在 termios 模块中), 2036
tcsetpgrp() (在 os 模块中), 634
tcsetwinsize() (在 termios 模块中), 2036
tearDown() (unittest.TestCase 方法), 1631
tearDownClass() (unittest.TestCase 方法), 1632
tee() (在 itertools 模块中), 386
teleport() (在 turtle 模块中), 1465
tell() (io.IOBase 方法), 682
tell() (io.TextIOBase 方法), 687
tell() (io.TextIOWrapper 方法), 688
tell() (mmap.mmap 方法), 1156
tell() (sqlite3.Blob 方法), 513
tell() (wave.Wave_read 方法), 1438
tell() (wave.Wave_write 方法), 1439
telnetlib
    module, 2059
TEMP, 451
temp_cwd() (在 test.support.os_helper 模块中), 1723
temp_dir() (在 test.support.os_helper 模块中), 1723
temp_umask() (在 test.support.os_helper 模块中), 1723
tempdir() (在 tempfile 模块中), 452
tempfile
    module, 448
Template (string 中的类), 120
template (string.Template 属性), 120
TemporaryDirectory (tempfile 中的类), 450
TemporaryFile() (在 tempfile 模块中), 448
temporary (bdb.Breakpoint 属性), 1732
teredo (ipaddress.IPv6Address 属性), 1426
TERM, 859
termattrs() (在 curses 模块中), 859
terminal_size (os 中的类), 635
terminate() (asyncio.subprocess.Process 方法), 1017
terminate() (asyncio.SubprocessTransport 方法), 1052
terminate() (multiprocessing.pool.Pool 方法), 926
terminate() (multiprocessing.Process 方法), 906
terminate() (subprocess.Popen 方法), 960
terminator (logging.StreamHandler 属性), 730
termios
    module, 2035
termname() (在 curses 模块中), 859
test
    module, 1707
--test
    tarfile 命令行选项, 565
    zipfile 命令行选项, 551
TEST_DATA_DIR() (在 test.support 模块中), 1711
TEST_HOME_DIR() (在 test.support 模块中), 1711
TEST_HTTP_URL() (在 test.support 模块中), 1711
TEST_SUPPORT_DIR() (在 test.support 模块中), 1711
TestCase (unittest 中的类), 1631
TestFailed, 1710
testfile() (在 doctest 模块中), 1611
TESTFN() (在 test.support.os_helper 模块中), 1721
TESTFN_NONASCII() (在 test.support.os_helper 模块中), 1721
TESTFN_UNDECODABLE() (在 test.support.os_helper 模块中), 1722
TESTFN_UNENCODABLE() (在 test.support.os_helper 模块中), 1722
TESTFN_UNICODE() (在 test.support.os_helper 模块中), 1722
TestLoader (unittest 中的类), 1642
testMethodPrefix (unittest.TestLoader 属性), 1643
testmod() (在 doctest 模块中), 1612
testNamePatterns (unittest.TestLoader 属性), 1644
test.regrtest
    module, 1709
TestResults (doctest 中的类), 1617
TestResult (unittest 中的类), 1644
testsource() (在 doctest 模块中), 1620
testsRun (unittest.TestResult 属性), 1644
TestSuite (unittest 中的类), 1641
test.support
    module, 1710
test.support.bytecode_helper
    module, 1720
test.support.import_helper
    module, 1723
test.support.os_helper
    module, 1721
test.support.script_helper
    module, 1719

test.support.socket_helper
    module, 1718
test.support.threading_helper
    module, 1720
test.support.warnings_helper
    module, 1724
test (doctest.DocTestFailure 属性), 1621
test (doctest.UnexpectedException 属性), 1621
testzip() (zipfile.ZipFile 方法), 545
text encoding -- 文本编码格式, 2076
text file -- 文本文件, 2076
text_encoding() (在 io 模块中), 679
text_factory (sqlite3.Connection 属性), 510
Textbox (curses.textpad 中的类), 878
TextCalendar (calendar 中的类), 232
textdomain() (在 gettext 模块中), 1441
textdomain() (在 locale 模块中), 1454
textinput() (在 turtle 模块中), 1483
TextIOBase (io 中的类), 686
TextIOWrapper (io 中的类), 687
TextIO (typing 中的类), 1584
TextTestResult (unittest 中的类), 1646
TextTestRunner (unittest 中的类), 1646
textwrap
    module, 154
TextWrapper (textwrap 中的类), 156
text (SyntaxError 属性), 104
text (traceback.TracebackException 属性), 1865
Text (typing 中的类), 1593
text (xml.etree.ElementTree.Element 属性), 1266
TFD_CLOEXEC() (在 os 模块中), 659
TFD_NONBLOCK() (在 os 模块中), 659
TFD_TIMER_ABSTIME() (在 os 模块中), 659
TFD_TIMER_CANCEL_ON_SET() (在 os 模块中), 659
theme_create() (tkinter.ttk.Style 方法), 1536
theme_names() (tkinter.ttk.Style 方法), 1536
theme_settings() (tkinter.ttk.Style 方法), 1536
theme_use() (tkinter.ttk.Style 方法), 1537
THOUSEP() (在 locale 模块中), 1451
thread() (imaplib.IMAP4 方法), 1373
thread_info() (在 sys 模块中), 1808
thread_time() (在 time 模块中), 698
thread_time_ns() (在 time 模块中), 698
ThreadedChildWatcher (asyncio 中的类), 1063
threading
    module, 885
threading_cleanup() (在 test.support.threading_helper 模块中), 1721
threading_setup() (在 test.support.threading_helper 模块中), 1721
ThreadingHTTPServer (http.server 中的类), 1392
ThreadingMixIn (socketserver 中的类), 1385
ThreadingMock (unittest.mock 中的类), 1665
ThreadingTCPServer (socketserver 中的类), 1385
ThreadingUDPServer (socketserver 中的类), 1385
ThreadingUnixDatagramServer (socketserver 中的类), 1385
ThreadingUnixStreamServer (socketserver 中的类), 1385
ThreadPoolExecutor (concurrent.futures 中的类), 946
ThreadPool (multiprocessing.pool 中的类), 930
threads
    POSIX, 978
threadsafety() (在 sqlite3 模块中), 499
Thread (threading 中的类), 888
THURSDAY() (在 calendar 模块中), 235
ticket_lifetime_hint (ssl.SSLSession 属性), 1132
tigetflag() (在 curses 模块中), 859
tigetnum() (在 curses 模块中), 859
tigetstr() (在 curses 模块中), 859
TILDE() (在 token 模块中), 1982
tilt() (在 turtle 模块中), 1476
tiltangle() (在 turtle 模块中), 1476
time
    module, 690
time() (asyncio.loop 方法), 1026
time() (datetime.datetime 方法), 206
time() (在 time 模块中), 697
Time2Internaldate() (在 imaplib 模块中), 1369
time_hi_version (uuid.UUID 属性), 1381
time_low (uuid.UUID 属性), 1381
time_mid (uuid.UUID 属性), 1381
time_ns() (在 time 模块中), 698
timedelta (datetime 中的类), 193
TimedRotatingFileHandler (logging.handlers 中的类), 733
timegm() (在 calendar 模块中), 235
timeit
    module, 1754
timeit() (timeit.Timer 方法), 1755
timeit() (在 timeit 模块中), 1754
timeit 命令行选项
    -h, 1756
    --help, 1756
    -n, 1756
    --number, 1756
    -p, 1756
    --process, 1756
    -r, 1756
    --repeat, 1756

-s, 1756
--setup, 1756
-u, 1756
--unit, 1756
-v, 1756
--verbose, 1756
timeout, 1079
timeout() (curses.window 方法), 866
timeout() (在 asyncio 模块中), 993
timeout_at() (在 asyncio 模块中), 994
TIMEOUT_MAX() (在 _thread 模块中), 979
TIMEOUT_MAX() (在 threading 模块中), 887
TimeoutError, 106, 907, 950, 1022
TimeoutExpired, 953
Timeout (asyncio 中的类), 994
timeout (socketserver.BaseServer 属性), 1387
timeout (ssl.SSLSession 属性), 1132
timeout (subprocess.TimeoutExpired 属性), 953
timerfd_create() (在 os 模块中), 657
timerfd_gettime() (在 os 模块中), 659
timerfd_gettime_ns() (在 os 模块中), 659
timerfd_settime() (在 os 模块中), 658
timerfd_settime_ns() (在 os 模块中), 658
TimerHandle (asyncio 中的类), 1040
Timer (threading 中的类), 896
Timer (timeit 中的类), 1754
times() (在 os 模块中), 669
timestamp() (datetime.datetime 方法), 207
TIMESTAMP (py_compile.PycInvalidationMode 属性), 1992
timetuple() (datetime.date 方法), 199
timetuple() (datetime.datetime 方法), 207
timetz() (datetime.datetime 方法), 206
time (datetime 中的类), 212
time (ssl.SSLSession 属性), 1132
time (uuid.UUID 属性), 1381
timezone() (在 time 模块中), 701
timezone (datetime 中的类), 221
--timing
    trace 命令行选项, 1759
title() (bytearray 方法), 71
title() (bytes 方法), 71
title() (str 方法), 56
title() (在 turtle 模块中), 1486
Tk, 1501
Tk 参数的数据类型, 1510
Tkinter, 1501
tkinter
    module, 1501
tkinter.colorchooser
    module, 1513
tkinter.commondialog
    module, 1517
tkinter.dnd
    module, 1520
tkinter.filedialog
    module, 1515
tkinter.font
    module, 1513
tkinter.messagebox
    module, 1517
tkinter.scrolledtext
    module, 1519
tkinter.simpledialog
    module, 1514
tkinter.ttk
    module, 1520
Tk (tkinter 中的类), 1503
tk (tkinter.Tk 属性), 1503
TLS, 1101
TLSv1_1 (ssl.TLSVersion 属性), 1113
TLSv1_2 (ssl.TLSVersion 属性), 1113
TLSv1_3 (ssl.TLSVersion 属性), 1113
TLSv1 (ssl.TLSVersion 属性), 1113
TLSVersion (ssl 中的类), 1112
tm_gmtoff (time.struct_time 属性), 697
tm_hour (time.struct_time 属性), 697
tm_isdst (time.struct_time 属性), 697
tm_mday (time.struct_time 属性), 697
tm_min (time.struct_time 属性), 697
tm_mon (time.struct_time 属性), 697
tm_sec (time.struct_time 属性), 697
tm_wday (time.struct_time 属性), 697
tm_yday (time.struct_time 属性), 697
tm_year (time.struct_time 属性), 697
tm_zone (time.struct_time 属性), 697
TMP, 451
TMPDIR, 451
TO_BOOL (*opcode*), 2002
to_bytes() (int 方法), 39
to_eng_string() (decimal.Context 方法), 341
to_eng_string() (decimal.Decimal 方法), 334
to_integral() (decimal.Decimal 方法), 335
to_integral_exact() (decimal.Context 方法), 341
to_integral_exact() (decimal.Decimal 方法), 335
to_integral_value() (decimal.Decimal 方法), 335
to_sci_string() (decimal.Context 方法), 341
to_thread() (在 asyncio 模块中), 997
ToASCII() (在 encodings.idna 模块中), 188
tobuf() (tarfile.TarInfo 方法), 559
tobytes() (array.array 方法), 268
tobytes() (memoryview 方法), 77
today() (datetime.date 类方法), 197
today() (datetime.datetime 类方法), 201
tofile() (array.array 方法), 268
tok_name() (在 token 模块中), 1980
token
    module, 1979

token_bytes() (在 secrets 模块中), 612
token_hex() (在 secrets 模块中), 612
token_urlsafe() (在 secrets 模块中), 612
TokenError, 1986
tokenize
    module, 1985
tokenize() (在 tokenize 模块中), 1985
tokenize 命令行选项
    -e, 1986
    --exact, 1986
    -h, 1986
    --help, 1986
Token (contextvars 中的类), 975
token (shlex.shlex 属性), 1499
tolist() (array.array 方法), 268
tolist() (memoryview 方法), 77
TOMLDecodeError, 593
tomllib
    module, 592
toordinal() (datetime.date 方法), 199
toordinal() (datetime.datetime 方法), 207
top() (curses.panel.Panel 方法), 884
top() (poplib.POP3 方法), 1367
top_panel() (在 curses.panel 模块中), 883
--top-level-directory
    unittest-discover 命令行选项, 1626
TopologicalSorter (graphlib 中的类), 305
toprettyxml() (xml.dom.minidom.Node 方法), 1285
toreadonly() (memoryview 方法), 77
tostring() (在 xml.etree.ElementTree 模块中), 1264
tostringlist() (在 xml.etree.ElementTree 模块中), 1265
total() (collections.Counter 方法), 242
total_changes (sqlite3.Connection 属性), 510
total_nframe (tracemalloc.Traceback 属性), 1769
total_ordering() (在 functools 模块中), 396
total_seconds() (datetime.timedelta 方法), 196
touch() (pathlib.Path 方法), 430
touchline() (curses.window 方法), 866
touchwin() (curses.window 方法), 866
tounicode() (array.array 方法), 268
ToUnicode() (在 encodings.idna 模块中), 188
towards() (在 turtle 模块中), 1469
toxml() (xml.dom.minidom.Node 方法), 1284
tparm() (在 curses 模块中), 859
trace
    module, 1758
--trace
    trace 命令行选项, 1759
trace() (在 inspect 模块中), 1889
trace_dispatch() (bdb.Bdb 方法), 1732
traceback
    module, 1861
traceback -- 回溯
    object -- 对象, 1792, 1861
traceback_limit (tracemalloc.Snapshot 属性), 1768
traceback_limit(wsgiref.handlers.BaseHandler 属性), 1318
TracebackException (traceback 中的类), 1864
tracebacklimit() (在 sys 模块中), 1809
TracebackType (types 中的类), 279
Traceback (inspect 中的类), 1888
Traceback (tracemalloc 中的类), 1769
traceback (tracemalloc.Statistic 属性), 1768
traceback (tracemalloc.StatisticDiff 属性), 1769
traceback (tracemalloc.Trace 属性), 1769
tracemalloc
    module, 1760
tracer() (在 turtle 模块中), 1481
traces (tracemalloc.Snapshot 属性), 1768
trace 命令行选项
    -C, 1759
    -c, 1759
    --count, 1759
    --coverdir, 1759
    -f, 1759
    --file, 1759
    -g, 1759
    --help, 1758
    --ignore-dir, 1759
    --ignore-module, 1759
    -l, 1759
    --listfuncs, 1759
    -m, 1759
    --missing, 1759
    --no-report, 1759
    -R, 1759
    -r, 1759
    --report, 1759
    -s, 1759
    --summary, 1759
    -T, 1759
    -t, 1759
    --timing, 1759
    --trace, 1759
    --trackcalls, 1759
    --version, 1758
Trace (trace 中的类), 1759
Trace (tracemalloc 中的类), 1769
--trackcalls
    trace 命令行选项, 1759
transfercmd() (ftplib.FTP 方法), 1362
transient_internet() (在 test.support.socket_helper 模块中), 1719
translate() (bytearray 方法), 65
translate() (bytes 方法), 65
translate() (str 方法), 56

translate() (在 fnmatch 模块中), 456
translate() (在 glob 模块中), 454
translation() (在 gettext 模块中), 1443
Transport Layer Security, 1101
Transport (asyncio 中的类), 1049
transport (asyncio.StreamWriter 属性), 1006
TraversableResources(importlib.abc 中的类), 1917
TraversableResources (importlib.resources.abc 中 的类), 1932
Traversable (importlib.abc 中的类), 1917
Traversable (importlib.resources.abc 中 的类), 1931
TreeBuilder (xml.etree.ElementTree 中的类), 1270
Treeview (tkinter.ttk 中的类), 1530
triangular() (在 random 模块中), 357
tries (doctest.DocTestRunner 属性), 1618
triple-quoted string -- 三引号字符串, **2076**
True, 35, 42
true, 35
truediv() (在 operator 模块中), 404
True (内置变量), 33
trunc() (在 *math* 模块中), 37
trunc() (在 math 模块中), 315
truncate() (io.IOBase 方法), 682
truncate() (在 os 模块中), 652
truth() (在 operator 模块中), 403
try
    statement -- 语句, 99
TryStar (ast 中的类), 1960
Try (ast 中的类), 1959
ttk, 1520
tty
    I/O 控制, 2035
    module, 2037
ttyname() (在 os 模块中), 634
TUESDAY() (在 calendar 模块中), 235
Tuple() (在 typing 模块中), 1592
Tuple (ast 中的类), 1947
tuple (内置类), 46
turtle
    module, 1457
turtledemo
    module, 1490
turtles() (在 turtle 模块中), 1485
TurtleScreen (turtle 中的类), 1486
turtlesize() (在 turtle 模块中), 1475
Turtle (turtle 中的类), 1486
type
    object -- 对象, 29
    union, 92
    内置函数, 95
    布尔值, 9
    运算目标 dictionary -- 字典, 83
    运算目标 list, 45
--type
    calendar 命令行选项, 237
type -- 类型, **2076**
type alias -- 类型别名, **2076**
type hint -- 类型注解, **2077**
TYPE_ALIAS (symtable.SymbolTableType 属性), 1977
type_check_only() (在 typing 模块中), 1589
TYPE_CHECKER (optparse.Option 属性), 848
TYPE_CHECKING() (在 typing 模块中), 1591
TYPE_COMMENT() (在 token 模块中), 1981
type_comment (ast.arg 属性), 1968
type_comment (ast.Assign 属性), 1954
type_comment (ast.For 属性), 1958
type_comment (ast.FunctionDef 属性), 1968
type_comment (ast.With 属性), 1960
TYPE_IGNORE() (在 token 模块中), 1981
TYPE_PARAMETERS (symtable.SymbolTableType 属性), 1977
TYPE_VARIABLE(symtable.SymbolTableType 属性), 1977
typeahead() (在 curses 模块中), 859
TypeAlias() (在 typing 模块中), 1563
TypeAliasType (typing 中的类), 1576
TypeAlias (ast 中的类), 1956
typecodes() (在 array 模块中), 267
typecode (array.array 属性), 267
TYPED_ACTIONS (optparse.Option 属性), 849
typed_subpart_iterator() (在 email.iterators 模块中), 1211
TypedDict (typing 中的类), 1580
TypeError, 105
TypeGuard() (在 typing 模块中), 1569
TypeIs() (在 typing 模块中), 1568
types
    immutable -- 不可变对象 sequence, 45
    module, 95, 276
    mutable -- 可变对象 sequence, 45
    内置, 35
    运算目标 integer, 37
    运算目标 mapping -- 映射, 83
    运算目标 sequence, 43, 45
    运算目标 数字, 37
types_map() (在 mimetypes 模块中), 1240
types_map_inv (mimetypes.MimeTypes 属性), 1241
types_map (mimetypes.MimeTypes 属性), 1241
TYPES (optparse.Option 属性), 848
TypeVarTuple (ast 中的类), 1967
TypeVarTuple (typing 中的类), 1573
TypeVar (ast 中的类), 1966
TypeVar (typing 中的类), 1571
type (optparse.Option 属性), 838
type (socket.socket 属性), 1097
type (tarfile.TarInfo 属性), 560
Type (typing 中的类), 1592
type (urllib.request.Request 属性), 1326
type (内置类), 29
typing

module, 1549
TZ, 698, 699
tzinfo (datetime 中的类), 215
tzinfo (datetime.datetime 属性), 205
tzinfo (datetime.time 属性), 212
tzname() (datetime.datetime 方法), 207
tzname() (datetime.time 方法), 214
tzname() (datetime.timezone 方法), 221
tzname() (datetime.tzinfo 方法), 216
tzname() (在 time 模块中), 701
TZPATH() (在 zoneinfo 模块中), 230
tzset() (在 time 模块中), 698

U

-u
    timeit 命令行选项, 1756
    uuid 命令行选项, 1383
U() (在 re 模块中), 130
UAdd (ast 中的类), 1949
ucd_3_2_0() (在 unicodedata 模块中), 159
udata (select.kevent 属性), 1140
UDPServer (socketserver 中的类), 1384
UF_APPEND() (在 stat 模块中), 444
UF_COMPRESSED() (在 stat 模块中), 445
UF_DATAVAULT() (在 stat 模块中), 445
UF_HIDDEN() (在 stat 模块中), 445
UF_IMMUTABLE() (在 stat 模块中), 444
UF_NODUMP() (在 stat 模块中), 444
UF_NOUNLINK() (在 stat 模块中), 445
UF_OPAQUE() (在 stat 模块中), 444
UF_SETTABLE() (在 stat 模块中), 444
UF_TRACKED() (在 stat 模块中), 445
uid() (imaplib.IMAP4 方法), 1373
uidl() (poplib.POP3 方法), 1367
UID (plistlib 中的类), 596
uid (tarfile.TarInfo 属性), 560
ulp() (在 math 模块中), 316
umask() (在 os 模块中), 623
unalias (*pdb command*), 1745
uname() (在 os 模块中), 623
uname() (在 platform 模块中), 743
uname (tarfile.TarInfo 属性), 560
UNARY_INVERT (*opcode*), 2002
UNARY_NEGATIVE (*opcode*), 2002
UNARY_NOT (*opcode*), 2002
UnaryOp (ast 中的类), 1949
UnboundLocalError, 105
UNC 路径
    和 os.makedirs(), 642
uncancel() (asyncio.Task 方法), 1001
UNCHECKED_HASH(py_compile.PycInvalidationMode 属性), 1992
unconsumed_tail (zlib.Decompress 属性), 527
unctrl() (在 curses 模块中), 860
unctrl() (在 curses.ascii 模块中), 882
Underflow (decimal 中的类), 343
undisplay (*pdb command*), 1744
undo() (在 turtle 模块中), 1468
undobufferentries()(在 turtle 模块中), 1479
undoc_header (cmd.Cmd 属性), 1493
unescape() (在 html 模块中), 1249
unescape() (在 xml.sax.saxutils 模块中), 1295
UnexpectedException, 1621
unexpectedSuccesses (unittest.TestResult 属性), 1644
unfreeze() (在 gc 模块中), 1874
unget_wch() (在 curses 模块中), 860
ungetch() (在 curses 模块中), 860
ungetch() (在 msvcrt 模块中), 2020
ungetmouse() (在 curses 模块中), 860
ungetwch() (在 msvcrt 模块中), 2020
unhexlify() (在 binascii 模块中), 1246
Unicode, 158, 173
    数据库, 158
UNICODE() (在 re 模块中), 130
unicodedata
    module, 158
UnicodeDecodeError, 105
UnicodeEncodeError, 105
UnicodeError, 105
UnicodeTranslateError, 105
UnicodeWarning, 107
unidata_version() (在 unicodedata 模块中), 159
unified_diff() (在 difflib 模块中), 146
uniform() (在 random 模块中), 357
UnimplementedFileMode, 1353
Union
    object -- 对象, 92
union
    type, 92
union() (frozenset 方法), 82
Union() (在 typing 模块中), 1563
UnionType (types 中的类), 279
Union (ctypes 中的类), 782
unique() (在 enum 模块中), 304
UNIQUE (enum.EnumCheck 属性), 301
--unit
    timeit 命令行选项, 1756
unittest
    module, 1622
unittest-discover 命令行选项
    -p, 1626
    --pattern, 1626
    -s, 1626
    --start-directory, 1626
    -t, 1626
    --top-level-directory, 1626
    -v, 1625
    --verbose, 1625
unittest.mock
    module, 1650
unittest 命令行选项
    -b, 1625
    --buffer, 1625

-c, 1625
--catch, 1625
--durations, 1625
-f, 1625
--failfast, 1625
-k, 1625
--locals, 1625
universal newlines -- 通用换行
bytearray.splitlines 方法, 70
bytes.splitlines 方法, 70
csv.reader 函数, 569
importlib.abc.InspectLoader.get_source 方法, 1914
io.IncrementalNewlineDecoder 类, 689
io.TextIOWrapper 类, 687
open() 内置函数, 23
str.splitlines 方法, 55
subprocess 模块, 954
universal newlines -- 通用换行, **2077**
UNIX
I/O 控制, 2039
文件控制, 2039
unix_dialect (csv 中的类), 572
unix_shell() (在 test.support 模块中), 1710
UnixDatagramServer (socketserver 中的类), 1384
UnixStreamServer(socketserver 中的类), 1384
unknown_decl() (html.parser.HTMLParser 方法), 1252
unknown_open() (urllib.request.BaseHandler 方法), 1329
unknown_open() (urllib.request.UnknownHandler 方法), 1332
UnknownHandler(urllib.request 中的类), 1326
UnknownProtocol, 1353
UnknownTransferEncoding, 1353
unknown (uuid.SafeUUID 属性), 1380
unlink() (multiprocessing.shared_memory.SharedMemory 方法), 940
unlink() (pathlib.Path 方法), 431
unlink() (xml.dom.minidom.Node 方法), 1284
unlink() (在 os 模块中), 653
unlink()(在 test.support.os_helper 模块中), 1723
unload()(在 test.support.import_helper 模块中), 1724
unlock() (mailbox.Babyl 方法), 1229
unlock() (mailbox.Mailbox 方法), 1224
unlock() (mailbox.Maildir 方法), 1226
unlock() (mailbox.mbox 方法), 1227
unlock() (mailbox.MH 方法), 1228
unlock() (mailbox.MMDF 方法), 1230
unlockpt() (在 os 模块中), 634
UNNAMED_SECTION() (在 configparser 模块中), 591
unpack() (struct.Struct 方法), 173
unpack() (在 struct 模块中), 168
Unpack() (在 typing 模块中), 1570
unpack_archive() (在 shutil 模块中), 465
UNPACK_EX (*opcode*), 2006
unpack_from() (struct.Struct 方法), 173
unpack_from() (在 struct 模块中), 168
UNPACK_SEQUENCE (*opcode*), 2006
unparse() (在 ast 模块中), 1972
unparsedEntityDecl() (xml.sax.handler.DTDHandler 方法), 1294
UnparsedEntityDeclHandler() (xml.parsers.expat.xmlparser 方法), 1303
Unpickler (pickle 中的类), 473
UnpicklingError, 472
unquote() (在 email.utils 模块中), 1209
unquote() (在 urllib.parse 模块中), 1345
unquote_plus() (在 urllib.parse 模块中), 1345
unquote_to_bytes()(在 urllib.parse 模块中), 1345
unraisablehook() (在 sys 模块中), 1809
unregister() (select.devpoll 方法), 1136
unregister() (select.epoll 方法), 1137
unregister() (selectors.BaseSelector 方法), 1142
unregister() (select.poll 方法), 1138
unregister() (在 atexit 模块中), 1860
unregister() (在 codecs 模块中), 175
unregister()(在 faulthandler 模块中), 1737
unregister_archive_format() (在 shutil 模块中), 465
unregister_dialect() (在 csv 模块中), 570
unregister_unpack_format()(在 shutil 模块中), 465
unsafe (uuid.SafeUUID 属性), 1380
unselect() (imaplib.IMAP4 方法), 1373
unset() (test.support.os_helper.EnvironmentVarGuard 方法), 1722
unsetenv() (在 os 模块中), 623
unshare() (在 os 模块中), 623
UnstructuredHeader (email.headerregistry 中的类), 1181
unsubscribe() (imaplib.IMAP4 方法), 1373
UnsupportedOperation, 413, 679
until (*pdb command*), 1742
untokenize() (在 tokenize 模块中), 1985
untouchwin() (curses.window 方法), 866
unused_data (bz2.BZ2Decompressor 属性), 534
unused_data (lzma.LZMADecompressor 属性), 539
unused_data (zlib.Decompress 属性), 527
unverifiable (urllib.request.Request 属性), 1326
unwrap() (ssl.SSLSocket 方法), 1116
unwrap() (在 inspect 模块中), 1887
unwrap() (在 urllib.parse 模块中), 1343
up (*pdb command*), 1741
up() (在 turtle 模块中), 1470

update() (collections.Counter 方法), 242
update() (dict 方法), 85
update() (frozenset 方法), 83
update() (hashlib.hash 方法), 601
update() (hmac.HMAC 方法), 610
update() (http.cookies.Morsel 方法), 1400
update() (mailbox.Mailbox 方法), 1223
update() (mailbox.Maildir 方法), 1226
update() (trace.CoverageResults 方法), 1760
update() (在 turtle 模块中), 1481
update_abstractmethods() (在 abc 模块中), 1860
update_authenticated() (urllib.request.HTTPPasswordMgrWithPriorAuth 方法), 1331
update_lines_cols() (在 curses 模块中), 860
update_panels() (在 curses.panel 模块中), 883
update_visible() (mailbox.BabylMessage 方法), 1235
update_wrapper() (在 functools 模块中), 401
upgrade_dependencies() (venv.EnvBuilder 方法), 1777
upper() (bytearray 方法), 72
upper() (bytes 方法), 72
upper() (str 方法), 56
urandom() (在 os 模块中), 676
url2pathname() (在 urllib.request 模块中), 1323
urlcleanup() (在 urllib.request 模块中), 1336
urldefrag() (在 urllib.parse 模块中), 1343
urlencode() (在 urllib.parse 模块中), 1346
URLError, 1347
urljoin() (在 urllib.parse 模块中), 1342
urllib
    module, 1321
urllib.error
    module, 1347
urllib.parse
    module, 1338
urllib.request
    module, 1321, 1352
urllib.response
    module, 1338
urllib.robotparser
    module, 1347
urlopen() (在 urllib.request 模块中), 1322
URLopener (urllib.request 中的类), 1336
urlparse() (在 urllib.parse 模块中), 1339
urlretrieve() (在 urllib.request 模块中), 1335
urlsafe_b64decode() (在 base64 模块中), 1242
urlsafe_b64encode() (在 base64 模块中), 1242
urlsplit() (在 urllib.parse 模块中), 1341
urlunparse() (在 urllib.parse 模块中), 1341
urlunsplit() (在 urllib.parse 模块中), 1342
url (http.client.HTTPResponse 属性), 1357
url (urllib.error.HTTPError 属性), 1347
url (urllib.response.addinfourl 属性), 1338
url (xmlrpc.client.ProtocolError 属性), 1414
urn (uuid.UUID 属性), 1381
US() (在 curses.ascii 模块中), 881
use_default_colors() (在 curses 模块中), 860
use_env() (在 curses 模块中), 860
use_rawinput (cmd.Cmd 属性), 1493
use_tool_id() (在 sys.monitoring 模块中), 1811
UseForeignDTD() (xml.parsers.expat.xmlparser 方法), 1301
USER, 851
user
    id, 620
    id, 设置, 623
    有效 id, 619
--user
    ensurepip 命令行选项, 1772
user() (poplib.POP3 方法), 1366
USER_BASE() (在 site 模块中), 1895
user_call() (bdb.Bdb 方法), 1733
user_exception() (bdb.Bdb 方法), 1734
user_line() (bdb.Bdb 方法), 1733
user_return() (bdb.Bdb 方法), 1733
USER_SITE() (在 site 模块中), 1895
--user-base
    site 命令行选项, 1896
usercustomize
    module, 1894
UserDict (collections 中的类), 253
UserList (collections 中的类), 253
USERNAME, 436, 619, 851
username (email.headerregistry.Address 属性), 1185
USERPROFILE, 436
userptr() (curses.panel.Panel 方法), 884
--user-site
    site 命令行选项, 1896
UserString (collections 中的类), 254
UserWarning, 107
USTAR_FORMAT() (在 tarfile 模块中), 555
USub (ast 中的类), 1949
UTC, 690
UTC() (在 datetime 模块中), 192
utcfromtimestamp() (datetime.datetime 类方法), 202
utcnow() (datetime.datetime 类方法), 202
utcoffset() (datetime.datetime 方法), 207
utcoffset() (datetime.time 方法), 214
utcoffset() (datetime.timezone 方法), 221
utcoffset() (datetime.tzinfo 方法), 215
utctimetuple() (datetime.datetime 方法), 207
utc (datetime.timezone 属性), 222
utf8() (poplib.POP3 方法), 1367

utf8_enabled (imaplib.IMAP4 属性), 1373
utf8_mode (sys.flags 属性), 1793
utf8 (email.policy.EmailPolicy 属性), 1176
utime() (在 os 模块中), 653
uu
    module, 2059
uuid
    module, 1380
--uuid
    uuid 命令行选项, 1383
uuid1() (在 uuid 模块中), 1382
uuid3() (在 uuid 模块中), 1382
uuid4() (在 uuid 模块中), 1382
uuid5() (在 uuid 模块中), 1382
uuid 命令行选项
    -h, 1383
    --help, 1383
    -N, 1383
    -n, 1383
    --name, 1383
    --namespace, 1383
    -u, 1383
    --uuid, 1383
UUID (uuid 中的类), 1380

V

-v
    doctest 命令行选项, 1604
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql 命令行选项, 516
    tarfile 命令行选项, 565
    timeit 命令行选项, 1756
    unittest-discover 命令行选项, 1625
v4_int_to_packed() (在 ipaddress 模块中), 1434
v6_int_to_packed() (在 ipaddress 模块中), 1434
valid_signals() (在 signal 模块中), 1147
validator() (在 wsgiref.validate 模块中), 1316
value
    真值, 35
Value() (multiprocessing.managers.SyncManager 方法), 920
Value() (在 multiprocessing 模块中), 915
Value() (在 multiprocessing.sharedctypes 模块中), 917
value_decode() (http.cookies.BaseCookie 方法), 1399
value_encode() (http.cookies.BaseCookie 方法), 1399
ValueError, 105
valuerefs() (weakref.WeakValueDictionary 方法), 271
values
    布尔值, 42
values() (contextvars.Context 方法), 977
values() (dict 方法), 85
values() (email.message.EmailMessage 方法), 1162
values() (email.message.Message 方法), 1197
values() (mailbox.Mailbox 方法), 1222
values() (types.MappingProxyType 方法), 280
ValuesView (collections.abc 中的类), 258
ValuesView (typing 中的类), 1595
Values (optparse 中的类), 837
value (ctypes._SimpleCData 属性), 779
value (enum.Enum 属性), 295
value (http.cookiejar.Cookie 属性), 1408
value (http.cookies.Morsel 属性), 1400
value (StopIteration 属性), 103
value (xml.dom.Attr 属性), 1280
variable annotation -- 变量标注, 2077
variance() (在 statistics 模块中), 370
variance (statistics.NormalDist 属性), 373
variant (uuid.UUID 属性), 1382
vars()
    built-in function, 30
var (contextvars.Token 属性), 975
VBAR() (在 token 模块中), 1982
VBAREQUAL() (在 token 模块中), 1983
vbar(tkinter.scrolledtext.ScrolledText 属性), 1519
元组
    object -- 对象, 45, 46
八进制
    字面值, 36
共享内存, 939
VC_ASSEMBLY_PUBLICKEYTOKEN() (在 msvcrt 模块中), 2021
内置
    types, 35
内置函数
    eval, 95, 284, 285
    exec, 14, 95
    float, 36
    hash, 45
    int, 36
    len, 43, 83
    max, 43
    min, 43
    slice -- 切片, 2012
    type, 95
    复数, 36
    编译, 95, 278
剪切, 1541
包, 1894
十六进制
    字面值, 36
协议
    copy, 476
    FTP, 1337, 1359
    HTTP, 1337, 1338, 1348, 1352, 1392
    IMAP4, 1368

IMAP4_SSL, 1368
IMAP4_stream, 1368
iterator -- 迭代器, 42
POP3, 1365
SMTP, 1374
上下文管理, 88
可变序列
循环, 44
可执行的 Zip 文件, 1782
Vec2D (turtle 中的类), 1487
venv
module, 1773
--verbose
doctest 命令行选项, 1604
tarfile 命令行选项, 565
timeit 命令行选项, 1756
unittest-discover 命令行选项, 1625
VERBOSE() (在 re 模块中), 130
verbose() (在 tabnanny 模块中), 1989
verbose() (在 test.support 模块中), 1710
verbose (sys.flags 属性), 1793
verify() (smtplib.SMTP 方法), 1377
verify() (在 enum 模块中), 305
VERIFY_ALLOW_PROXY_CERTS() (在 ssl 模块中), 1107
verify_client_post_handshake() (ssl.SSLSocket 方法), 1116
verify_code (ssl.SSLCertVerificationError 属性), 1104
VERIFY_CRL_CHECK_CHAIN() (在 ssl 模块中), 1107
VERIFY_CRL_CHECK_LEAF() (在 ssl 模块中), 1107
VERIFY_DEFAULT() (在 ssl 模块中), 1107
verify_flags (ssl.SSLContext 属性), 1124
verify_generated_headers (email.policy.Policy 属性), 1175
verify_message(ssl.SSLCertVerificationError 属性), 1104
verify_mode (ssl.SSLContext 属性), 1124
verify_request() (socketserver.BaseServer 方法), 1388
VERIFY_X509_PARTIAL_CHAIN()(在 ssl 模块中), 1107
VERIFY_X509_STRICT() (在 ssl 模块中), 1107
VERIFY_X509_TRUSTED_FIRST()(在 ssl 模块中), 1107
VerifyFlags (ssl 中的类), 1108
VerifyMode (ssl 中的类), 1107
--version
python--m-sqlite3-[-h]-[-v]-[filename]-[sql 命令行选项, 516
trace 命令行选项, 1758
version() (ssl.SSLSocket 方法), 1116
version() (在 curses 模块中), 866
version() (在 ensurepip 模块中), 1772
version() (在 importlib.metadata 模块中), 1935
version() (在 marshal 模块中), 488
version() (在 platform 模块中), 743
version() (在 sqlite3 模块中), 499
version() (在 sys 模块中), 1809
version_info() (在 sqlite3 模块中), 500
version_info() (在 sys 模块中), 1810
version_string() (http.server.BaseHTTPRequestHandler 方法), 1395
version(email.headerregistry.MIMEVersionHeader 属性), 1182
version (http.client.HTTPResponse 属性), 1357
version (http.cookiejar.Cookie 属性), 1408
version (http.cookies.Morsel 属性), 1399
version (ipaddress.IPv4Address 属性), 1423
version (ipaddress.IPv4Network 属性), 1428
version (ipaddress.IPv6Address 属性), 1426
version (ipaddress.IPv6Network 属性), 1431
version (sys.thread_info 属性), 1809
version (urllib.request.URLopener 属性), 1337
version (uuid.UUID 属性), 1382
图形用户界面, 1501
vformat() (string.Formatter 方法), 112
基准测试, 1754
复制, 1541
复数
内置函数, 36
大文件, 2033
virtual environment -- 虚拟环境, **2077**
virtual machine -- 虚拟机, **2077**
visit() (ast.NodeVisitor 方法), 1973
visit_Constant() (ast.NodeVisitor 方法), 1973
字符, 158
字节串
object -- 对象, 60
str (内置类), 49
插值, 72
方法, 62
格式化, 72
字节码
文件, 1991
字面值
complex number -- 复数, 36
integer, 36
二进制, 36
八进制, 36
十六进制, 36
数字, 36
浮点数, 36
安全
http.server, 1398
安全哈希算法, SHA1, SHA2, SHA224, SHA256, SHA384, SHA512, SHA3, Shake, Blake2, 599
安全考量, 2059
审计, 1788

审计事件, 1727
容器
    迭代目标, 42
密码, 599
vline() (curses.window 方法), 866
已命名共享内存, 939
布尔值
    object -- 对象, 36
    type, 9
    values, 42
    操作, 35
带行缓冲的 I/O, 23
异常
    chaining, 99
形符, **2076**
循环
    针对可变序列, 44
循环冗余检测, 526
voidcmd() (ftplib.FTP 方法), 1361
volume (zipfile.ZipInfo 属性), 550
vonmisesvariate() (在 random 模块中), 358
VT() (在 curses.ascii 模块中), 880

W

-w
    calendar 命令行选项, 238
W_OK() (在 os 模块中), 637
性能, 1754
性能分析函数, 887, 1798, 1804
性能分析器, 1798, 1804
性能分析, 确定性的, 1746
wait() (asyncio.Barrier 方法), 1014
wait() (asyncio.Condition 方法), 1012
wait() (asyncio.Event 方法), 1011
wait() (asyncio.subprocess.Process 方法), 1017
wait() (multiprocessing.pool.AsyncResult 方法), 926
wait() (subprocess.Popen 方法), 959
wait() (threading.Barrier 方法), 897
wait() (threading.Condition 方法), 893
wait() (threading.Event 方法), 896
wait() (在 asyncio 模块中), 995
wait() (在 concurrent.futures 模块中), 950
wait() (在 multiprocessing.connection 模块中), 928
wait() (在 os 模块中), 669
wait3() (在 os 模块中), 670
wait4() (在 os 模块中), 670
wait_closed() (asyncio.Server 方法), 1042
wait_closed() (asyncio.StreamWriter 方法), 1006
wait_for() (asyncio.Condition 方法), 1012
wait_for() (threading.Condition 方法), 893
wait_for() (在 asyncio 模块中), 995
wait_process() (在 test.support 模块中), 1714
wait_threads_exit() (在 test.support.threading_helper 模块中), 1721
wait_until_any_call_with() (unittest.mock.ThreadingMock 方法), 1665
wait_until_called() (unittest.mock.ThreadingMock 方法), 1665
waitid() (在 os 模块中), 669
waitpid() (在 os 模块中), 670
waitstatus_to_exitcode() (在 os 模块中), 672
walk() (email.message.EmailMessage 方法), 1164
walk() (email.message.Message 方法), 1200
walk() (pathlib.Path 方法), 428
walk() (在 ast 模块中), 1973
walk() (在 os 模块中), 653
walk_packages() (在 pkgutil 模块中), 1904
walk_stack() (在 traceback 模块中), 1864
walk_tb() (在 traceback 模块中), 1864
want (doctest.Example 属性), 1615
warn() (在 warnings 模块中), 1830
warn_default_encoding (sys.flags 属性), 1793
warn_explicit() (在 warnings 模块中), 1830
Warning, 107, 514
warning() (logging.Logger 方法), 705
warning() (xml.sax.handler.ErrorHandler 方法), 1294
WARNING() (在 logging 模块中), 707
warning() (在 logging 模块中), 714
WARNING() (在 tkinter.messagebox 模块中), 1519
warnings, 1825
    module, 1825
WarningsRecorder(test.support.warnings_helper 中的类), 1725
warnoptions() (在 sys 模块中), 1810
wasSuccessful() (unittest.TestResult 方法), 1645
WatchedFileHandler (logging.handlers 中的类), 731
wave
    module, 1437
Wave_read (wave 中的类), 1438
Wave_write (wave 中的类), 1439
WCONTINUED() (在 os 模块中), 671
WCOREDUMP() (在 os 模块中), 672
打包
    二进制 数据, 167
拷贝文件, 458
拼接
    operation, 43
掩码
    操作, 37
提示符, 解释器, 1804
插值

bytearray (%), 72
bytes (%), 72
插值，字符串 (%), 58
插值字符串字面值, 57
搜索
path, module, 457, 1802, 1893
操作
bitwise, 37
布尔值, 35
掩码, 37
移位, 37
数字
object -- 对象, 36
types, 运算目标, 37
字面值, 36
转换, 37
数据
tabular, 569
打包 二进制, 167
数据库, 493
Unicode, 158
数组, 266
文件
.ini, 576
mime.types, 1240
path 配置, 1894
.pdbrc, 1741
plist, 595
临时, 448
大文件, 2033
字节码, 1991
拷贝, 458
模式, 22
调试器 配置, 1741
配置, 576
文件名
临时, 448
路径名扩展, 453
通配符扩展, 456
文件控制
UNIX, 2039
文本模式, 23
文档
在线, 1597
生成, 1597
方法
bytearray, 62
string, 49
字节串, 62
WeakKeyDictionary (weakref 中的类), 270
WeakMethod (weakref 中的类), 271
weakref
module, 269
WeakSet (weakref 中的类), 271
WeakValueDictionary (weakref 中的类), 271
webbrowser
module, 1309
webbrowser 命令行选项
-n, 1309
--new-tab, 1309
--new-window, 1309
-t, 1309
WEDNESDAY() (在 calendar 模块中), 235
weekday() (datetime.date 方法), 199
weekday() (datetime.datetime 方法), 208
weekday() (在 calendar 模块中), 234
weekday (calendar.IllegalWeekdayError 属性), 236
weekheader() (在 calendar 模块中), 234
weibullvariate() (在 random 模块中), 358
WEXITED() (在 os 模块中), 671
WEXITSTATUS() (在 os 模块中), 673
服务器
WWW, 1392
wfile (http.server.BaseHTTPRequestHandler 属性), 1393
wfile(socketserver.DatagramRequestHandler 属性), 1388
标头
MIME, 1239
栈查看器, 1540
格式化
bytearray (%), 72
bytes (%), 72
格式化，字符串 (%), 58
格式化字符串字面值, 57
whatis (*pdb command*), 1743
when() (asyncio.Timeout 方法), 994
when() (asyncio.TimerHandle 方法), 1040
where (*pdb command*), 1741
which() (在 shutil 模块中), 462
whichdb() (在 dbm 模块中), 489
while
statement -- 语句, 35
While (ast 中的类), 1958
whitespace() (在 string 模块中), 112
whitespace_split (shlex.shlex 属性), 1498
whitespace (shlex.shlex 属性), 1498
Widget (tkinter.ttk 中的类), 1523
--width
calendar 命令行选项, 238
width() (在 turtle 模块中), 1470
width (sys.hash_info 属性), 1799
width (textwrap.TextWrapper 属性), 156
WIFCONTINUED() (在 os 模块中), 672
WIFEXITED() (在 os 模块中), 673
WIFSIGNALED() (在 os 模块中), 673
WIFSTOPPED() (在 os 模块中), 673
模块浏览器, 1538
模式
文件, 22
win32_edition() (在 platform 模块中), 744
win32_is_iot() (在 platform 模块中), 744
win32_ver() (在 platform 模块中), 744
WinDLL (ctypes 中的类), 771
window manager (部件), 1509

window() (curses.panel.Panel 方法), 884
window_height() (在 turtle 模块中), 1485
window_width() (在 turtle 模块中), 1485
Windows ini 文件, 576
WindowsError, 105
WindowsPath (pathlib 中的类), 422
WindowsProactorEventLoopPolicy (asyncio 中的类), 1062
WindowsRegistryFinder (importlib.machinery 中的类), 1918
WindowsSelectorEventLoopPolicy (asyncio 中的类), 1061
WinError() (在 ctypes 模块中), 778
winerror (OSError 属性), 102
WINFUNCTYPE() (在 ctypes 模块中), 774
winreg
    module, 2021
WinSock, 1135
winsound
    module, 2030
winver() (在 sys 模块中), 1810
WITH_EXCEPT_START (*opcode*), 2005
with_hostmask (ipaddress.IPv4Interface 属性), 1433
with_hostmask (ipaddress.IPv4Network 属性), 1429
with_hostmask (ipaddress.IPv6Interface 属性), 1433
with_hostmask (ipaddress.IPv6Network 属性), 1431
with_name() (pathlib.PurePath 方法), 420
with_netmask (ipaddress.IPv4Interface 属性), 1433
with_netmask (ipaddress.IPv4Network 属性), 1429
with_netmask (ipaddress.IPv6Interface 属性), 1433
with_netmask (ipaddress.IPv6Network 属性), 1431
with_prefixlen (ipaddress.IPv4Interface 属性), 1433
with_prefixlen (ipaddress.IPv4Network 属性), 1428
with_prefixlen (ipaddress.IPv6Interface 属性), 1433
with_prefixlen (ipaddress.IPv6Network 属性), 1431
with_pymalloc() (在 test.support 模块中), 1712
with_segments() (pathlib.PurePath 方法), 421
with_stem() (pathlib.PurePath 方法), 420
with_suffix() (pathlib.PurePath 方法), 421
with_traceback() (BaseException 方法), 100
withitem (ast 中的类), 1961
With (ast 中的类), 1960
比较
    chaining, 36
    objects, 36
    operator, 36
浮点数
    object -- 对象, 36
    字面值, 36
消息摘要, MD5, 599
清除断点, 1541
WNOHANG() (在 os 模块中), 671
WNOWAIT() (在 os 模块中), 671
wordchars (shlex.shlex 属性), 1498
World Wide Web, 1309, 1338, 1347
wrap() (textwrap.TextWrapper 方法), 157
wrap() (在 textwrap 模块中), 154
wrap_bio() (ssl.SSLContext 方法), 1122
wrap_future() (在 asyncio 模块中), 1046
wrap_socket() (ssl.SSLContext 方法), 1121
wrapper() (在 curses 模块中), 860
WrapperDescriptorType() (在 types 模块中), 278
wraps() (在 functools 模块中), 402
writable() (bz2.BZ2File 方法), 533
writable() (io.IOBase 方法), 682
WRITABLE() (在 _tkinter 模块中), 1513
WRITABLE (inspect.BufferFlags 属性), 1892
write() (asyncio.StreamWriter 方法), 1005
write() (asyncio.WriteTransport 方法), 1051
write() (codecs.StreamWriter 方法), 180
write() (code.InteractiveInterpreter 方法), 1898
write() (configparser.ConfigParser 方法), 590
write() (email.generator.BytesGenerator 方法), 1171
write() (email.generator.Generator 方法), 1172
write() (io.BufferedIOBase 方法), 683
write() (io.BufferedWriter 方法), 686
write() (io.RawIOBase 方法), 682
write() (io.TextIOBase 方法), 687
write() (mmap.mmap 方法), 1156
write() (sqlite3.Blob 方法), 513
write() (ssl.MemoryBIO 方法), 1132
write() (ssl.SSLSocket 方法), 1114
write() (xml.etree.ElementTree.ElementTree 方法), 1269
write() (zipfile.ZipFile 方法), 545
write() (在 os 模块中), 634
write() (在 turtle 模块中), 1474
write_byte() (mmap.mmap 方法), 1156
write_bytes() (pathlib.Path 方法), 427
write_docstringdict() (在 turtle 模块中), 1489
write_eof() (asyncio.StreamWriter 方法), 1006
write_eof() (asyncio.WriteTransport 方法), 1051
write_eof() (ssl.MemoryBIO 方法), 1132

write_history_file() (在 readline 模块中), 162
write_results() (trace.CoverageResults 方法), 1760
write_text() (pathlib.Path 方法), 427
write_through (io.TextIOWrapper 属性), 688
writeframes() (wave.Wave_write 方法), 1439
writeframesraw() (wave.Wave_write 方法), 1439
writeheader() (csv.DictWriter 方法), 574
writelines() (asyncio.StreamWriter 方法), 1005
writelines() (asyncio.WriteTransport 方法), 1051
writelines() (codecs.StreamWriter 方法), 180
writelines() (io.IOBase 方法), 682
writepy() (zipfile.PyZipFile 方法), 548
writer() (在 csv 模块中), 570
writerow() (csv.csvwriter 方法), 574
writerows() (csv.csvwriter 方法), 574
writestr() (zipfile.ZipFile 方法), 546
WriteTransport (asyncio 中的类), 1049
writev() (在 os 模块中), 635
writexml() (xml.dom.minidom.Node 方法), 1284
WRITE (inspect.BufferFlags 属性), 1893
WrongDocumentErr, 1281
wsgi_file_wrapper (wsgiref.handlers.BaseHandler 属性), 1319
wsgi_multiprocess (wsgiref.handlers.BaseHandler 属性), 1318
wsgi_multithread(wsgiref.handlers.BaseHandler 属性), 1318
wsgi_run_once(wsgiref.handlers.BaseHandler 属性), 1318
WSGIApplication()(在 wsgiref.types 模块中), 1320
WSGIEnvironment()(在 wsgiref.types 模块中), 1320
wsgiref
    module, 1312
wsgiref.handlers
    module, 1316
wsgiref.headers
    module, 1314
wsgiref.simple_server
    module, 1315
wsgiref.types
    module, 1320
wsgiref.util
    module, 1312
wsgiref.validate
    module, 1316
WSGIRequestHandler (wsgiref.simple_server 中的类), 1315
WSGIServer (wsgiref.simple_server 中的类), 1315
wShowWindow (subprocess.STARTUPINFO 属性), 962
WSTOPPED() (在 os 模块中), 671
WSTOPSIG() (在 os 模块中), 673
wstring_at() (在 ctypes 模块中), 778
WTERMSIG() (在 os 模块中), 673
WUNTRACED() (在 os 模块中), 671
WWW, 1309, 1338, 1347
    服务器, 1392

X

-x
    compileall 命令行选项, 1993
X() (在 re 模块中), 130
X509 证书, 1125
X_OK() (在 os 模块中), 637
xatom() (imaplib.IMAP4 方法), 1373
XATTR_CREATE() (在 os 模块中), 660
XATTR_REPLACE() (在 os 模块中), 660
XATTR_SIZE_MAX() (在 os 模块中), 660
xcor() (在 turtle 模块中), 1469
xdrlib
    module, 2059
XHTML, 1249
XHTML_NAMESPACE() (在 xml.dom 模块中), 1274
xml
    module, 1254
XML() (在 xml.etree.ElementTree 模块中), 1265
XML_ERROR_ABORTED() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_AMPLIFICATION_LIMIT_BREACH()(在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_ASYNC_ENTITY() (在 xml.parsers.expat.errors 模块中), 1306
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF() (在 xml.parsers.expat.errors 模块中), 1306
XML_ERROR_BAD_CHAR_REF() (在 xml.parsers.expat.errors 模块中), 1306
XML_ERROR_BINARY_ENTITY_REF() (在 xml.parsers.expat.errors 模块中), 1306
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_DUPLICATE_ATTRIBUTE() (在 xml.parsers.expat.errors 模块中), 1306
XML_ERROR_ENTITY_DECLARED_IN_PE() (在 xml.parsers.expat.errors 模块中), 1307

XML_ERROR_EXTERNAL_ENTITY_HANDLING() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_FEATURE_REQUIRES_XML_DTD() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_FINISHED() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_INCOMPLETE_PE() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_INCORRECT_ENCODING() (在 xml.parsers.expat.errors 模块中), 1306
XML_ERROR_INVALID_ARGUMENT() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_INVALID_TOKEN() (在 xml.parsers.expat.errors 模块中), 1306
XML_ERROR_JUNK_AFTER_DOC_ELEMENT() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_MISPLACED_XML_PI() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_NO_BUFFER() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_NO_ELEMENTS() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_NO_MEMORY() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_NOT_STANDALONE() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_NOT_SUSPENDED() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_PARAM_ENTITY_REF() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_PARTIAL_CHAR() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_PUBLICID() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_RECURSIVE_ENTITY_REF() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_RESERVED_NAMESPACE_URI() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_RESERVED_PREFIX_XML() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_RESERVED_PREFIX_XMLNS() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_SUSPEND_PE() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_SUSPENDED() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_SYNTAX() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_TAG_MISMATCH() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_TEXT_DECL() (在 xml.parsers.expat.errors 模块中), 1308
XML_ERROR_UNBOUND_PREFIX() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_UNCLOSED_CDATA_SECTION() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_UNCLOSED_TOKEN() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_UNDECLARING_PREFIX() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_UNDEFINED_ENTITY() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_UNEXPECTED_STATE() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_UNKNOWN_ENCODING() (在 xml.parsers.expat.errors 模块中), 1307
XML_ERROR_XML_DECL() (在 xml.parsers.expat.errors 模块中), 1308
XML_NAMESPACE() (在 xml.dom 模块中), 1274
xmlcharrefreplace
    错误处理器名称, 177
xmlcharrefreplace_errors() (在 codecs 模块中), 178
XmlDeclHandler() (xml.parsers.expat.xmlparser 方法), 1303
xml.dom
    module, 1273
xml.dom.minidom
    module, 1283
xml.dom.pulldom
    module, 1286
xml.etree.ElementInclude

module, 1266
xml.etree.ElementTree
module, 1256
XMLFilterBase (xml.sax.saxutils 中的类), 1295
XMLGenerator(xml.sax.saxutils 中的类), 1295
XMLID() (在 xml.etree.ElementTree 模块中), 1265
XMLNS_NAMESPACE() (在 xml.dom 模块中), 1274
xml.parsers.expat
module, 1299
xml.parsers.expat.errors
module, 1306
xml.parsers.expat.model
module, 1305
XMLParserType() (在 xml.parsers.expat 模块中), 1300
XMLParser (xml.etree.ElementTree 中的类), 1271
XMLPullParser (xml.etree.ElementTree 中的类), 1272
XMLReader (xml.sax.xmlreader 中的类), 1296
xmlrpc
module, 1409
xmlrpc.client
module, 1409
xmlrpc.server
module, 1417
xml.sax
module, 1288
xml.sax.handler
module, 1290
xml.sax.saxutils
module, 1295
xml.sax.xmlreader
module, 1296
xor() (在 operator 模块中), 404
xview() (tkinter.ttk.Treeview 方法), 1533

Y

ycor() (在 turtle 模块中), 1469
year
calendar 命令行选项, 237
yeardatescalendar() (calendar.Calendar 方法), 232
yeardays2calendar() (calendar.Calendar 方法), 232
yeardayscalendar() (calendar.Calendar 方法), 232
year (datetime.date 属性), 197
year (datetime.datetime 属性), 204
YES() (在 tkinter.messagebox 模块中), 1519
YESEXPR() (在 locale 模块中), 1451
YESNO()(在 tkinter.messagebox 模块中), 1519
YESNOCANCEL() (在 tkinter.messagebox 模块中), 1519
YIELD_VALUE (*opcode*), 2005
YieldFrom (ast 中的类), 1969
Yield (ast 中的类), 1969
yiq_to_rgb() (在 colorsys 模块中), 1440
yview() (tkinter.ttk.Treeview 方法), 1533

Z

z
在字符串格式化中, 115
z85decode() (在 base64 模块中), 1243
z85encode() (在 base64 模块中), 1243
Zen of Python -- Python 之禅, **2077**
ZeroDivisionError, 105
zfill() (bytearray 方法), 72
zfill() (bytes 方法), 72
zfill() (str 方法), 56
zip()
built-in function, 30
ZIP_BZIP2() (在 zipfile 模块中), 542
ZIP_DEFLATED() (在 zipfile 模块中), 542
zip_longest() (在 itertools 模块中), 387
ZIP_LZMA() (在 zipfile 模块中), 542
ZIP_STORED() (在 zipfile 模块中), 542
zipapp
module, 1782
zipapp 命令行选项
-c, 1783
--compress, 1783
-h, 1783
--help, 1783
--info, 1783
-m, 1782
--main, 1782
-o, 1782
--output, 1782
-p, 1782
--python, 1782
zipfile
module, 541
zipfile 命令行选项
-c, 551
--create, 551
-e, 551
--extract, 551
-l, 551
--list, 551
--metadata-encoding, 551
-t, 551
--test, 551
ZipFile (zipfile 中的类), 543
zipimport
module, 1901
ZipImportError, 1901
zipimporter (zipimport 中的类), 1902
ZipInfo (zipfile 中的类), 542
zlib
module, 525
ZLIB_RUNTIME_VERSION()(在 zlib 模块中), 528
ZLIB_VERSION() (在 zlib 模块中), 528
zoneinfo

    module, 226
ZoneInfoNotFoundError, 230
ZoneInfo (zoneinfo 中的类), 228
zscore() (statistics.NormalDist 方法), 374