

O'REILLY®

TURING

图灵程序设计丛书

第2版 上

流畅的 Python

Fluent Python, Second Edition

深入理解Python语言核心特性及底层逻辑



[巴西] 卢西亚诺·拉马略 (Luciano Ramalho) 著

安道 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：流畅的 Python（第2版）

作者：[巴西] 卢西亚诺·拉马略（Luciano Ramalho）

译者：安道

ISBN：978-7-115-61236-6

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

版权声明

O'Reilly Media, Inc.介绍

献辞

专业推荐

前言

- 目标读者

- 非目标读者

- 五本书合而为一

 - 内容编排

- 以实践为基础

- 杂谈：一点个人看法

- 排版约定

- 使用代码示例

- O'Reilly 在线学习平台 (O'Reilly Online Learning)

- 联系我们

- 致谢

 - 第 1 版致谢

- 电子书

第一部分 数据结构

第 1 章 Python 数据模型

- 1.1 本章新增内容

- 1.2 一掬 Python 风格的纸牌

- 1.3 特殊方法是如何使用的

 - 1.3.1 模拟数值类型

 - 1.3.2 字符串表示形式

 - 1.3.3 自定义类型的布尔值

 - 1.3.4 容器 API

- 1.4 特殊方法概述

1.5 `len` 为什么不是方法

1.6 本章小结

1.7 延伸阅读

第 2 章 丰富的序列

2.1 本章新增内容

2.2 内置序列类型概览

2.3 列表推导式和生成器表达式

2.3.1 列表推导式对可读性的影响

2.3.2 列表推导式与 `map` 和 `filter` 比较

2.3.3 笛卡儿积

2.3.4 生成器表达式

2.4 元组不仅仅是不可变列表

2.4.1 用作记录

2.4.2 用作不可变列表

2.4.3 列表和元组方法的比较

2.5 序列和可迭代对象拆包

2.5.1 使用 `*` 获取余下的项

2.5.2 在函数调用和序列字面量中使用 `*` 拆包

2.5.3 嵌套拆包

2.6 序列模式匹配

使用模式匹配序列实现一个解释器

2.7 切片

2.7.1 为什么切片和区间排除最后一项

2.7.2 切片对象

2.7.3 多维切片和省略号

2.7.4 为切片赋值

2.8 使用 `+` 和 `*` 处理序列

2.8.1 构建嵌套列表

- 2.8.2 使用增量赋值运算符处理序列
 - 2.8.3 一个 += 运算符赋值谜题
- 2.9 list.sort 与内置函数 sorted
- 2.10 当列表不适用时
 - 2.10.1 数组
 - 2.10.2 memoryview
 - 2.10.3 NumPy
 - 2.10.4 双端队列和其他队列
- 2.11 本章小结
- 2.12 延伸阅读
- 第 3 章 字典和集合
 - 3.1 本章新增内容
 - 3.2 字典的现代句法
 - 3.2.1 字典推导式
 - 3.2.2 映射拆包
 - 3.2.3 使用 | 合并映射
 - 3.3 使用模式匹配处理映射
 - 3.4 映射类型的标准 API
 - 3.4.1 “可哈希”指什么
 - 3.4.2 常用映射方法概述
 - 3.4.3 插入或更新可变的值
 - 3.5 自动处理缺失的键
 - 3.5.1 defaultdict: 处理缺失键的另一种选择
 - 3.5.2 __missing__ 方法
 - 3.5.3 标准库对 __missing__ 方法的使用不一致
 - 3.6 dict 的变体
 - 3.6.1 collections.OrderedDict
 - 3.6.2 collections.ChainMap

- 3.6.3 `collections.Counter`
 - 3.6.4 `shelve.Shelf`
 - 3.6.5 子类应继承 `UserDict` 而不是 `dict`
 - 3.7 不可变映射
 - 3.8 字典视图
 - 3.9 `dict` 的实现方式对实践的影响
 - 3.10 集合论
 - 3.10.1 `set` 字面量
 - 3.10.2 集合推导式
 - 3.11 集合的实现方式对实践的影响
 - 集合运算
 - 3.12 字典视图的集合运算
 - 3.13 本章小结
 - 3.14 延伸阅读
- 第 4 章 Unicode 文本和字节序列
- 4.1 本章新增内容
 - 4.2 字符问题
 - 4.3 字节概要
 - 4.4 基本的编码解码器
 - 4.5 处理编码和解码问题
 - 4.5.1 处理 `UnicodeEncodeError`
 - 4.5.2 处理 `UnicodeDecodeError`
 - 4.5.3 加载模块时编码不符合预期抛出的 `SyntaxError`
 - 4.5.4 如何找出字节序列的编码
 - 4.5.5 BOM: 有用的鬼符
 - 4.6 处理文本文件
 - 了解默认编码
 - 4.7 为了正确比较而规范化 Unicode 字符串

- 4.7.1 大小写同一化
 - 4.7.2 规范化文本匹配的实用函数
 - 4.7.3 极端“规范化”：去掉变音符
 - 4.8 Unicode 文本排序
 - 使用 Unicode 排序算法排序
 - 4.9 Unicode 数据库
 - 4.9.1 按名称查找字符
 - 4.9.2 字符的数值意义
 - 4.10 支持 `str` 和 `bytes` 的双模式 API
 - 4.10.1 正则表达式中的 `str` 和 `bytes`
 - 4.10.2 `os` 函数中的 `str` 和 `bytes`
 - 4.11 本章小结
 - 4.12 延伸阅读
- 第 5 章 数据类构建器
- 5.1 本章新增内容
 - 5.2 数据类构建器概述
 - 主要功能
 - 5.3 典型的具名元组
 - 5.4 带类型的具名元组
 - 5.5 类型提示入门
 - 5.5.1 运行时没有作用
 - 5.5.2 变量注解句法
 - 5.5.3 变量注解的意义
 - 5.6 `@dataclass` 详解
 - 5.6.1 字段选项
 - 5.6.2 初始化后处理
 - 5.6.3 带类型的类属性
 - 5.6.4 初始化不作为字段的变量

- 5.6.5 @dataclass 示例：都柏林核心模式
- 5.7 数据类导致代码异味
 - 5.7.1 把数据类用作脚手架
 - 5.7.2 把数据类用作中间表述
- 5.8 模式匹配类实例
 - 5.8.1 简单类模式
 - 5.8.2 关键字类模式
 - 5.8.3 位置类模式
- 5.9 本章小结
- 5.10 延伸阅读
- 第 6 章 对象引用、可变性和垃圾回收
 - 6.1 本章新增内容
 - 6.2 变量不是盒子
 - 6.3 同一性、相等性和别名
 - 6.3.1 在 == 和 is 之间选择
 - 6.3.2 元组的相对不可变性
 - 6.4 默认做浅拷贝
 - 为任意对象做浅拷贝和深拷贝
 - 6.5 函数的参数是引用时
 - 6.5.1 不要使用可变类型作为参数的默认值
 - 6.5.2 防御可变参数
 - 6.6 del 和垃圾回收
 - 6.7 Python 对不可变类型施加的把戏
 - 6.8 本章小结
 - 6.9 延伸阅读
- 第二部分 函数即对象
- 第 7 章 函数是一等对象
 - 7.1 本章新增内容

7.2 把函数视为对象

7.3 高阶函数

`map`、`filter` 和 `reduce` 的现代替代品

7.4 匿名函数

7.5 9 种可调用对象

7.6 用户定义的可调用类型

7.7 从位置参数到仅限关键字参数

仅限位置参数

7.8 支持函数式编程的包

7.8.1 `operator` 模块

7.8.2 使用 `functools.partial` 冻结参数

7.9 本章小结

7.10 延伸阅读

第 8 章 函数中的类型提示

8.1 本章新增内容

8.2 关于渐进式类型

8.3 渐进式类型实践

8.3.1 Mypy 初体验

8.3.2 让 Mypy 严格要求

8.3.3 参数的默认值

8.3.4 使用 `None` 表示默认值

8.4 类型由受支持的操作定义

8.5 注解中可用的类型

8.5.1 `Any` 类型

8.5.2 简单的类型和类

8.5.3 `Optional` 类型和 `Union` 类型

8.5.4 泛化容器

8.5.5 元组类型

- 8.5.6 泛化映射
 - 8.5.7 抽象基类
 - 8.5.8 Iterable
 - 8.5.9 参数化泛型和 TypeVar
 - 8.5.10 静态协议
 - 8.5.11 Callable
 - 8.5.12 NoReturn
- 8.6 注解仅限位置参数和变长参数
- 8.7 类型不完美，测试须全面
- 8.8 本章小结
- 8.9 延伸阅读
- 第 9 章 装饰器和闭包
 - 9.1 本章新增内容
 - 9.2 装饰器基础知识
 - 9.3 Python 何时执行装饰器
 - 9.4 注册装饰器
 - 9.5 变量作用域规则
 - 9.6 闭包
 - 9.7 nonlocal 声明
 - 变量查找逻辑
 - 9.8 实现一个简单的装饰器
 - 工作原理
 - 9.9 标准库中的装饰器
 - 9.9.1 使用 `functools.cache` 做备忘
 - 9.9.2 使用 `lru_cache`
 - 9.9.3 单分派泛化函数
 - 9.10 参数化装饰器
 - 9.10.1 一个参数化注册装饰器

- 9.10.2 参数化 `clock` 装饰器
 - 9.10.3 基于类的 `clock` 装饰器
- 9.11 本章小结
- 9.12 延伸阅读
- 第 10 章 使用一等函数实现设计模式
 - 10.1 本章新增内容
 - 10.2 案例分析：重构策略模式
 - 10.2.1 经典的策略模式
 - 10.2.2 使用函数实现策略模式
 - 10.2.3 选择最佳策略的简单方式
 - 10.2.4 找出一个模块中的全部策略
 - 10.3 使用装饰器改进策略模式
 - 10.4 命令模式
 - 10.5 本章小结
 - 10.6 延伸阅读
- 第三部分 类和协议
- 第 11 章 符合 Python 风格的对象
 - 11.1 本章新增内容
 - 11.2 对象表示形式
 - 11.3 再谈向量类
 - 11.4 备选构造函数
 - 11.5 `classmethod` 与 `staticmethod`
 - 11.6 格式化显示
 - 11.7 可哈希的 `Vector2d`
 - 11.8 支持位置模式匹配
 - 11.9 第 3 版 `Vector2d` 的完整代码
 - 11.10 Python 私有属性和“受保护”的属性
 - 11.11 使用 `__slots__` 节省空间

- 11.11.1 简单衡量 `__slots__` 节省的内存
 - 11.11.2 总结 `__slots__` 的问题
- 11.12 覆盖类属性
- 11.13 本章小结
- 11.14 延伸阅读
- 第 12 章 序列的特殊方法
 - 12.1 本章新增内容
 - 12.2 `Vector` 类：用户定义的序列类型
 - 12.3 `Vector` 类第 1 版：与 `Vector2d` 类兼容
 - 12.4 协议和鸭子类型
 - 12.5 `Vector` 类第 2 版：可切片的序列
 - 12.5.1 切片原理
 - 12.5.2 能处理切片的 `__getitem__` 方法
 - 12.6 `Vector` 类第 3 版：动态存取属性
 - 12.7 `Vector` 类第 4 版：哈希和快速等值测试
 - 12.8 `Vector` 类第 5 版：格式化
 - 12.9 本章小结
 - 12.10 延伸阅读
- 第 13 章 接口、协议和抽象基类
 - 13.1 类型图
 - 13.2 本章新增内容
 - 13.3 两种协议
 - 13.4 利用鸭子类型编程
 - 13.4.1 Python 喜欢序列
 - 13.4.2 使用猴子补丁在运行时实现协议
 - 13.4.3 防御性编程和“快速失败”
 - 13.5 大鹅类型
 - 13.5.1 子类化一个抽象基类

- 13.5.2 标准库中的抽象基类
- 13.5.3 定义并使用一个抽象基类
- 13.5.4 抽象基类句法详解
- 13.5.5 子类化抽象基类 `Tombola`
- 13.5.6 抽象基类的虚拟子类
- 13.5.7 `register` 的实际使用
- 13.5.8 使用抽象基类实现结构类型
- 13.6 静态协议
 - 13.6.1 为 `double` 函数添加类型提示
 - 13.6.2 运行时可检查的静态协议
 - 13.6.3 运行时协议检查的局限性
 - 13.6.4 支持静态协议
 - 13.6.5 设计一个静态协议
 - 13.6.6 协议设计最佳实践
 - 13.6.7 扩展一个协议
 - 13.6.8 `numbers` 模块中的抽象基类和 `Numeric` 协议
- 13.7 本章小结
- 13.8 延伸阅读
- 第 14 章 继承：瑕瑜互见
 - 14.1 本章新增内容
 - 14.2 `super()` 函数
 - 14.3 子类化内置类型很麻烦
 - 14.4 多重继承和方法解析顺序
 - 14.5 混入类
 - 不区分大小写的映射
 - 14.6 多重继承的实际运用
 - 14.6.1 抽象基类也是混入类
 - 14.6.2 `ThreadingMixin` 和 `ForkingMixin`

14.6.3 Django 泛化视图混入类

14.6.4 Tkinter 中的多重继承

14.7 应对多重继承

14.7.1 优先使用对象组合，而不是类继承

14.7.2 理解不同情况下使用继承的原因

14.7.3 使用抽象基类显式表示接口

14.7.4 通过混入明确重用代码

14.7.5 为用户提供聚合类

14.7.6 仅子类化为子类化设计的类

14.7.7 避免子类化具体类

14.7.8 Tkinter 的好、不好以及令人厌恶的方面

14.9 本章小结

14.10 延伸阅读

第 15 章 类型提示进阶

15.1 本章新增内容

15.2 重载的签名

15.2.1 重载 `max` 函数

15.2.2 重载 `max` 函数的启示

15.3 TypedDict

15.4 类型校正

15.5 在运行时读取类型提示

15.5.1 注解在运行时的问题

15.5.2 解决这个问题

15.6 实现一个泛化类

泛型基本术语

15.7 型变

15.7.1 一个不变的自动售货机

15.7.2 一个协变的自动售货机

- 15.7.3 一个逆变的垃圾桶
 - 15.7.4 型变总结
- 15.8 实现泛化静态协议
- 15.9 本章小结
- 15.10 延伸阅读
- 第 16 章 运算符重载
 - 16.1 本章新增内容
 - 16.2 运算符重载入门
 - 16.3 一元运算符
 - 16.4 重载向量加法运算符 $+$
 - 16.5 重载标量乘法运算符 $*$
 - 16.6 把 `@` 当作中缀运算符使用
 - 16.7 算术运算符总结
 - 16.8 众多比较运算符
 - 16.9 增量赋值运算符
 - 16.10 本章小结
 - 16.11 延伸阅读
- 第四部分 控制流
- 第 17 章 迭代器、生成器和经典协程
 - 17.1 本章新增内容
 - 17.2 单词序列
 - 17.3 序列可以迭代的原因： `iter` 函数
 - 使用 `iter` 处理可调用对象
 - 17.4 可迭代对象与迭代器
 - 17.5 为 `Sentence` 类实现 `__iter__` 方法
 - 17.5.1 `Sentence` 类第 2 版：经典迭代器
 - 17.5.2 不要把可迭代对象变成迭代器
 - 17.5.3 `Sentence` 类第 3 版：生成器函数

- 17.5.4 生成器的工作原理
- 17.6 惰性实现版本
 - 17.6.1 `Sentence` 类第 4 版：惰性生成器
 - 17.6.2 `Sentence` 类第 5 版：惰性生成器表达式
- 17.7 何时使用生成器表达式
- 17.8 一个等差数列生成器
 - 使用 `itertools` 模块生成等差数列
- 17.9 标准库中的生成器函数
- 17.10 可迭代的归约函数
- 17.11 `yield from`：从子生成器中产出
 - 17.11.1 重新实现 `chain`
 - 17.11.2 遍历树状结构
- 17.12 泛化可迭代类型
- 17.13 经典协程
 - 17.13.1 示例：使用协程计算累计平均值
 - 17.13.2 让协程返回一个值
 - 17.13.3 经典协程的泛化类型提示
- 17.14 本章小结
- 17.15 延伸阅读
- 第 18 章 `with`、`match` 和 `else` 块
 - 18.1 本章新增内容
 - 18.2 上下文管理器和 `with` 块
 - 18.2.1 `contextlib` 包中的实用工具
 - 18.2.2 使用 `@contextmanager`
 - 18.3 案例分析：`lis.py` 中的模式匹配
 - 18.3.1 `Scheme` 句法
 - 18.3.2 导入和类型
 - 18.3.3 解析器

- 18.3.4 环境
 - 18.3.5 REPL
 - 18.3.6 求值函数
 - 18.3.7 实现闭包的 Procedure 类
 - 18.3.8 使用 OR 模式
- 18.4 先做这个，再做那个：if 语句之外的 else 块
- 18.5 本章小结
- 18.6 延伸阅读
- 第 19 章 Python 并发模型
 - 19.1 本章新增内容
 - 19.2 全景概览
 - 19.3 术语定义
 - 进程、线程和声名狼藉的 Python GIL
 - 19.4 一个演示并发的“Hello World”示例
 - 19.4.1 使用线程实现旋转指针
 - 19.4.2 使用进程实现旋转指针
 - 19.4.3 使用协程实现旋转指针
 - 19.4.4 对比几版 supervisor 函数
 - 19.5 GIL 真正的影响
 - 快速问答
 - 19.6 自建进程池
 - 19.6.1 基于进程的方案
 - 19.6.2 理解用时
 - 19.6.3 利用多核进行素数检测的程序代码
 - 19.6.4 实验：进程数多一些或少一些
 - 19.6.5 基于线程的方案并不可靠
 - 19.7 多核世界中的 Python
 - 19.7.1 系统管理

- 19.7.2 数据科学
 - 19.7.3 服务器端 Web 和移动开发
 - 19.7.4 WSGI 应用程序服务器
 - 19.7.5 分布式任务队列
- 19.8 本章小结
- 19.9 延伸阅读
 - 19.9.1 使用线程和进程实现并发
 - 19.9.2 GIL
 - 19.9.3 标准库之外的并发世界
 - 19.9.4 Python 之外的并发和伸缩世界
- 第 20 章 并发执行器
 - 20.1 本章新增内容
 - 20.2 并发网络下载
 - 20.2.1 依序下载脚本
 - 20.2.2 使用 `concurrent.futures` 模块下载
 - 20.2.3 `future` 对象在哪里
 - 20.3 使用 `concurrent.futures` 启动进程
 - 重写多核版素数检测程序
 - 20.4 实验 `Executor.map` 方法
 - 20.5 显示下载进度并处理错误
 - 20.5.1 `flags2` 系列示例处理错误的方式
 - 20.5.2 使用 `futures.as_completed` 函数
 - 20.6 本章小结
 - 20.7 延伸阅读
- 第 21 章 异步编程
 - 21.1 本章新增内容
 - 21.2 一些定义
 - 21.3 一个 `asyncio` 示例：探测域名

Guido 阅读异步代码的技巧

21.4 新概念：可异步调用对象

21.5 使用 `asyncio` 和 HTTPX 下载

21.5.1 原生协程的秘密：默默无闻的生成器

21.5.2 “不成功便成仁”问题

21.6 异步上下文管理器

21.7 增强 `asyncio` 版下载脚本的功能

21.7.1 使用 `asyncio.as_completed` 和一个线程

21.7.2 使用信号量限制请求

21.7.3 每次下载发起多个请求

21.8 把任务委托给执行器

21.9 使用 `asyncio` 编写服务器

21.9.1 一个 FastAPI Web 服务

21.9.2 一个使用 `asyncio` 编写的 TCP 服务器

21.10 异步迭代和异步可迭代对象

21.10.1 异步生成器函数

21.10.2 异步生成器表达式和异步推导式

21.11 `asyncio` 之外的异步世界：Curio

21.12 异步对象的类型提示

21.13 异步原理与陷阱

21.13.1 阻塞型调用导致漫长等待

21.13.2 I/O 密集型系统的误区

21.13.3 绕开 CPU 密集型陷阱

21.14 本章小结

21.15 延伸阅读

第五部分 元编程

第 22 章 动态属性和特性

22.1 本章新增内容

- 22.2 使用动态属性转换数据
 - 22.2.1 使用动态属性访问 JSON 类数据
 - 22.2.2 处理无效属性名
 - 22.2.3 使用 `__new__` 方法灵活创建对象
- 22.3 计算特性
 - 22.3.1 第 1 步：数据驱动属性创建
 - 22.3.2 第 2 步：通过特性获取链接的记录
 - 22.3.3 第 3 步：用特性覆盖现有属性
 - 22.3.4 第 4 步：自己实现特性缓存
 - 22.3.5 第 5 步：使用 `functools` 缓存特性
- 22.4 使用特性验证属性
 - 22.4.1 `LineItem` 类第 1 版：表示订单中商品的类
 - 22.4.2 `LineItem` 类第 2 版：能验证值的特性
- 22.5 特性全解析
 - 22.5.1 特性覆盖实例属性
 - 22.5.2 特性的文档
- 22.6 定义一个特性工厂函数
- 22.7 处理属性删除操作
- 22.8 处理属性的重要属性和函数
 - 22.8.1 影响属性处理方式的特殊属性
 - 22.8.2 处理属性的内置函数
 - 22.8.3 处理属性的特殊方法
- 22.9 本章小结
- 22.10 延伸阅读
- 第 23 章 属性描述符
 - 23.1 本章新增内容
 - 23.2 描述符示例：属性验证
 - 23.2.1 `LineItem` 类第 3 版：一个简单的描述符

- 23.2.2 `LineItem` 类第 4 版：为储存属性自动命名
 - 23.2.3 `LineItem` 类第 5 版：一种新型描述符
- 23.3 覆盖型描述符与非覆盖型描述符对比
 - 23.3.1 覆盖型描述符
 - 23.3.2 没有 `__get__` 方法的覆盖型描述符
 - 23.3.3 非覆盖型描述符
 - 23.3.4 覆盖类中的描述符
- 23.4 方法是描述符
- 23.5 描述符用法建议
- 23.6 描述符的文档字符串和覆盖删除操作
- 23.7 本章小结
- 23.8 延伸阅读
- 第 24 章 类元编程
 - 24.1 本章新增内容
 - 24.2 身为对象的类
 - 24.3 `type`：内置的类工厂函数
 - 24.4 类工厂函数
 - 24.5 引出 `__init_subclass__`
 - 为什么 `__init_subclass__` 不能配置 `__slots__`
 - 24.6 使用类装饰器增强类的功能
 - 24.7 导入时和运行时比较
 - 求解时间实验
 - 24.8 元类入门
 - 24.8.1 元类如何定制类
 - 24.8.2 一个友好的元类示例
 - 24.8.3 元类求解时间实验
 - 24.9 使用元类实现 `Checked` 类
 - 24.10 元类的实际运用

- 24.10.1 可简化或代替元类的现代功能
- 24.10.2 元类是稳定的语言功能
- 24.10.3 一个类只能有一个元类
- 24.10.4 元类应作为实现细节
- 24.11 使用元类的 `__prepare__` 方法实现新颖的构思
- 24.12 小结
- 24.13 本章小结
- 24.14 延伸阅读

结语

延伸阅读

作者简介

关于封面

版权声明

Copyright © 2022 Luciano Ramalho. All rights reserved.

Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2023. Authorized translation of the English edition, 2023 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2022。

简体中文版由人民邮电出版社有限公司出版，2023。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc.介绍

O'Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、互动学习、认证体验、图书及视频等，使客户更容易获取业务成功所需的专业知识。几十年来 O'Reilly 图书一直被视为学习开创未来之技术的权威资料。我们所做的一切是为了帮助各领域的专业人士学习最佳实践，发现并塑造科技行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

献辞

献给爱妻 Marta。

专业推荐

“在我刚学习 Python 时，这本书对我帮助非常大，我从书中学习到了 Python 的各种优秀特性。如今这本书出第 2 版了，内容也做了全面升级，依旧干货满满，每个学习 Python 的朋友都不容错过！”

——崔庆才，《Python3 网络爬虫开放实战》作者
微软（中国）软件工程师

“《流畅的 Python》是我心中最棒的编程书之一，它教会了我许多。从技术角度，书中对于 Python 数据模型、面向对象和元编程等知识的透彻讲述，让我的编程技术得到了很大提升。另外，作者精心打磨的遍布全书的各类细节，更是对我影响深远。书中每个示例都经过精心设计，每行疑难代码都有批注说明。当我自己开始技术写作时，我从这本书中获益良多。”

“第 2 版在第 1 版的基础上，将 Python 版本更新到了 Python 3.10，同时追加了异步编程部分，是助你精通 Python 语言的不二之选。”

——朱雷 (@piglei)，腾讯公司高级工程师
《Python 工匠：案例、技巧与工程实践》作者

前言

要不这样吧，如果编程语言里有个地方你弄不明白，而正好又有个人用了这个功能，那就请他出去。这比学习新特性要容易些，然后过不了多久，剩下的程序员就会开始用 Python 0.9.6 版，而且他们只需要使用这个版本中易于理解的那一小部分就好了（眨眼）。¹

——Tim Peters

传奇的核心开发者，《Python 之禅》作者

¹发布在 Usenet 小组 comp.lang.python 中的消息，2002 年 12 月 23 日，“Acrimony in c.l.p”。

Python 3.10 官方教程的开头是这样写的：“Python 是一门既容易上手又强大的编程语言。”这句话本身并无大碍，但需要注意的是，正因为它既好学又好用，所以很多 Python 程序员只用到了其强大功能的一小部分。

只需要几个小时，经验丰富的程序员就能学会用 Python 写出实用的程序。然而随着这最初高产的几个小时变成数周甚至数月，在那些先入为主的编程语言的影响下，开发人员将慢慢地写出带着“口音”的 Python 代码。即便 Python 是你的初恋，也难逃此命运。这是因为在学校里，抑或是那些入门书上，教授者往往有意避免介绍这门语言所独有的功能。

另外，向那些已在其他语言领域里拥有丰富经验的程序员介绍 Python 的时候，我还发现一个问题：人们总是倾向于寻求自己熟悉的东西。受到其他语言的影响，你大概能猜到 Python 支持正则表达式，然后就会去查阅文档。但是，如果你从未接触过元组拆包（tuple unpacking），也没听说过描述符（descriptor），那么估计你不会特

地去搜索，然后就永远失去了使用这些 Python 所独有的功能的机会。

这不是一本全面介绍 Python 的手册，而是着重讲解 Python 语言所独有的功能。Python 语言核心以及它的一些库也是本书的重点。尽管 Python 的包索引现在已经有 6 万多个库，而且很多十分实用，但是我几乎不会提到 Python 标准库以外的包。

目标读者

本书的目标读者是那些正在使用 Python，同时想熟练掌握 Python 3 的程序员。书中的示例使用 Python 3.10 做了测试，其中大多数还用 Python 3.9 和 Python 3.8 做了测试。如果某个示例必须使用 Python 3.10，我会标出来。

如果你尚不清楚自己对 Python 的熟练程度能否跟得上本书的内容，建议你回头看看 Python 官方教程。注意，除了一些新功能之外，官方教程涵盖的内容在本书中不做说明。

非目标读者

如果你才刚刚开始学 Python，本书的内容可能会显得有些“超纲”。比难懂更糟的是，如果在学习 Python 的过程中过早接触本书内容，你可能会误以为所有 Python 脚本都需要使用特殊方法和元编程（metaprogramming）技巧。不成熟的抽象和过早的优化一样，都会坏事。

五本书合而为一

建议你不要跳过第 1 章。如果你是本书的目标读者，直接阅读第 1 章后面的任何一章都可以，不过我通常假设你按顺序阅读每一部分中的

各章。本书第一部分到第五部分可以分别看作五本独立的书。

在介绍让你自己实现某些功能的方法之前，我通常会先把现成可用的工具讲清楚。比如说，第一部分的第 2 章涵盖现成可用的序列类型（sequence type），包括 `collections.deque` 这种不太受关注的序列。直到第三部分，我才会讲解如何自己定义序列，同时说明如何利用 `collections.abc` 包提供的抽象基类（abstract base class, ABC）。自己创建抽象基类更是延后到第三部分后半段，因为我相信，熟练使用抽象基类之后才能自己动手创建。

这样做有几个好处。第一，知道有什么现成的工具可用，能避免重新发明轮子，毕竟我们使用现有容器类的概率要远大于自己动手写一套新的。第二，如此一来，在讨论如何创建新基类之前，我们有更多的机会了解现成基类的高级用法。第三，比起从零开始，继承现有的抽象基类应该会更简单一些。最后，我认为见过一些实际案例之后会更容易理解抽象。

当然，这样也会带来一些不便之处，比如前面的内容遍布对后文的引用。当你明白这样做的理由之后，这一点不便之处希望是可以容忍的。

内容编排

下面是本书每一部分的主题。

第一部分 数据结构

第 1 章介绍 Python 数据模型，说明为什么特殊方法（例如 `__repr__`）是所有类型的对象在行为上保持一致的关键。特殊方法在书中各章有深入的讲解。接下来的几章介绍各种容器类型，包括序列（sequence）、映射（mapping）和集合（set），另外还涉及字符串（`str`）和字节序列（`bytes`）的区别。说起来，这是让亲者（Python 3 用户）快、仇者（Python 2 用户）痛的一个关键，因其使代码迁移难度陡增。另外，这一部分还讲到标准库中的高级类构建

器：具名元组工厂和 `@dataclass` 装饰器。Python 3.10 新引入的模式匹配在第 2 章、第 3 章和第 5 章的几节中有讨论，涉及序列模式、映射模式和类模式。第 6 章关注对象的生命周期：引用、可变性和垃圾回收。

第二部分 函数即对象

第二部分探讨作为 Python 语言一等对象的函数，说明这意味着什么、对一些流行的设计模式有何影响，以及如何利用闭包

(closure) 实现函数装饰器。这一部分还涉及 Python 中可调用对象的概念、函数属性、内省 (introspection)、参数注解，以及 Python 3 新引入的 `nonlocal` 声明。第 8 章介绍重要的新功能类型提示在函数签名中的使用。

第三部分 类和协议

第三部分的重点转移到自己动手构建类上，这与第 5 章介绍的类构建器截然不同。与其他面向对象语言一样，Python 有自己的一套功能，可能并不会出现在你我学习过的其他基于类的编程语言中。这一部分的各章说明如何自己动手构建容器、抽象基类和协议，以及如何处理多重继承、如何在必要时实现运算符重载。第 15 章继续讲解类型提示。

第四部分 控制流

第四部分涵盖 Python 中除条件判断、循环和子程序 (subroutine) 等传统的控制流以外的语言结构和库。先讲生成器 (generator)，再讲上下文管理器 (context manager) 和协程 (coroutine)，还涉及新增的功能强大但不容易理解的 `yield from` 句法。第 18 章有一个重要的示例：在一个简单的函数式语言的解释器中使用模式匹配。新增的第 19 章概述 Python 并发和并行处理的各种方案及其局限性，以及使 Python Web 应用程序具有良好伸缩性能的软件架构。为了强调语言核心功能，例如 `await`、`async def`、

`async for` 和 `async with`，我重写了讲**异步编程**的章节，说明如何在 `asyncio` 和其他框架中使用这些功能。

第五部分 元编程

第五部分开头先讲如何动态创建带属性的类，用以处理诸如 JSON 一类的半结构化数据。然后，从业已熟悉的特性（property）机制入手，借助描述符从底层解释 Python 对象属性的存取。接着，讲解函数、方法和描述符之间的关系。整个第五部分会一步步实现一个字段验证库，在此过程中会遇到一些不易察觉的问题，由此自然引出最后一章所讨论的高级工具：类装饰器和元类（metaclass）。

以实践为基础

我们经常使用 Python 交互式控制台来探索语言本身和各种库。我觉得有必要强调一下，这个工具非常适合学习，尤其是对更为熟悉没有 REPL（read-eval-print loop，“读取-求值-输出”循环）的静态编译型语言的读者来说。

`doctest` 是 Python 标准库提供的测试包，通过模拟控制台会话确认表达式求值是否正确。本书中的多数代码，包括控制台输出清单，使用 `doctest` 做了检查。在阅读本书的过程中，你不需要使用，甚至不需要知道 `doctest`。`doctest` 看起来就像是 Python 交互式控制台的转录，方便你自行试验。

有时，我会先给出一个 `doctest`，说明想实现什么效果，然后再编写代码，让测试通过。在考虑如何实现一个功能之前，先明确要实现什么功能，这样在编程时才能有的放矢。先写测试是测试驱动开发（test-driven development, TDD）的基本要求，我发现这在教学中也大有裨益。如果你对 `doctest` 不熟悉，请花点时间阅读文档和本书示例代码。

我还使用 `pytest` 为一些大型示例编写了单元测试。我发现，`pytest` 比标准库中的 `unittest` 模块用起来更简单，也更强大。你可以在自己操作系统的命令窗口中输入 `python3 -m doctest example_script.py` 或 `pytest`，确认本书中的多数代码是否正确。示例代码中的 `pytest.ini` 文件中有相关配置，确保 `pytest` 命令能够收集并执行 `doctest`。

杂谈：一点个人看法

我从 1998 年开始就使用和教授 Python，并一直参与相关的讨论。我喜欢学习和比较不同的编程语言，研究语言的设计和背后的理论。部分章节的末尾有一个“杂谈”栏目，我在其中谈了对 Python 和其他语言的看法。如果你不感兴趣，跳过即可。“杂谈”中的内容完全选读。

排版约定

本书使用下列排版约定。

黑体

表示新术语。

等宽字体 (`constant width`)

表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

加粗等宽字体 (**`constant width bold`**)

表示应该由用户输入的命令或其他文本。

等宽斜体 (*`constant width italic`*)

表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

与本书相关的技术问题，或者在使用代码示例上有疑问，请发电子邮件到 arrata@oreilly.com.cn。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例集则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN，例如：“*Fluent Python*, 2nd ed., by Luciano Ramalho (O'Reilly). Copyright 2022 Luciano Ramalho, 978-1-492-05635-5”。

如果你觉得自己对示例代码的使用超出了上述许可范围，请通过 permissions@oreilly.com 与我们联系。

O'Reilly 在线学习平台 (O'Reilly Online Learning)



40 多年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独特的由专家和创新者组成的庞大网络，他们通过图书、文章和我们的在线学习平台分享他们的知识和经验。O'Reilly 在线学习平台让你能够按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本资源和视频资源。有关的更多信息，请访问 <https://www.oreilly.com>。

联系我们

如有与本书有关的评价或问题，请联系出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室
(100035)

奥莱利技术咨询（北京）有限公司

勘误、示例和其他信息可访问

<https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/> 获取。²

²本书中文版勘误提交和随书代码获取，请访问 ituring.cn/book/2893。——编者注

对于本书的评论和技术性问题，请发送电子邮件到：
bookquestions@oreilly.com。

要了解更多 O'Reilly 图书和培训的信息，请访问
<https://www.oreilly.com/>。

我们在 Facebook 的地址如下：<https://www.facebook.com/OReilly/>。

请关注我们的 Twitter 动态：<https://twitter.com/oreillymedia>。

我们的 YouTube 视频地址如下：
<https://www.youtube.com/oreillymedia>。

致谢

没想到，五年后我会升级一本 Python 图书，而且工作量这么大。我深爱的妻子 Marta Mello 一直伴我左右，不离不弃。我亲爱的朋友 Leonardo Rochael 在初期写作到后期技术审校的过程中始终给我帮助，还负责整理和复核其他技术审校、读者和编辑的反馈。老实说，没有 Marta 和 Leonardo 的支持，我肯定坚持不下来。真诚地感谢你们！

第 2 版优秀的技术审校团队包括 Jürgen Gmach、Caleb Hattingh、Jess Males、Leonardo Rochael 和 Miroslav Šedivý。他们审校了整本书。Bill Behrman、Bruce Eckel、Renato Oliveira 和 Rodrigo Bernardo Pimentel 审校了部分章节。他们从不同角度提出的许多建议为本书增色不少。

很多读者对抢读版提交了勘误，或者做出了其他贡献，包括：Guilherme Alves、Christiano Anderson、Konstantin Baikov、K. Alex Birch、Michael Boesl、Lucas Brunialti、Sergio Cortez、Gino Crecco、Chukwuerika Dike、Juan Esteras、Federico Fissore、Will

Frey、Tim Gates、Alexander Hagerman、Chen Hanxiao、Sam Hyeong、Simon Ilincev、Parag Kalra、Tim King、David Kwast、Tina Lapine、Wanpeng Li、Guto Maia、Scott Martindale、Mark Meyer、Andy McFarland、Chad McIntire、Diego Rabatone Oliveira、Francesco Piccoli、Meredith Rawls、Michael Robinson、Federico Tula Rovaletti、Tushar Sadhwani、Arthur Constantino Scardua、Randal L. Schwartz、Avichai Sefati、Guannan Shen、William Simpson、Vivek Vashist、Jerry Zhang、Paul Zuradzki。还有些人不愿具名，或者在我提交草稿之后才发来勘误，也有可能是我没能记下名字——请原谅我。

在调研过程中，我与 Michael Albert、Pablo Aguilar、Kaleb Barrett、David Beazley、J. S. O. Bueno、Bruce Eckel、Martin Fowler、Ivan Levkivskyi、Alex Martelli、Peter Norvig、Sebastian Rittau、Guido van Rossum、Carol Willing 和 Jelle Zijlstra 有过深入交流，加深了对类型、并发、模式匹配和元编程的认识。

O'Reilly 的编辑 Jeff Bleiel、Jill Leonard 和 Amelia Blevins 提出了很多建议，增强了本书的连贯性。Jeff Bleiel 和执行编辑 Danny Elfanbaum 鼓励我坚持完成了这场漫长的马拉松。

以上每个人的真知灼见和真诚建议让本书变得更好、更准确。尽管如此，由于我个人的疏漏，成书难免存在缺憾之处，请允许我先在此致歉。

最后，我要衷心感谢 Thoughtworks 巴西分公司的同事，特别是资助我的 Alexey Bôas，他一直以多种方式支持这个项目。

当然，每个帮助我理解 Python 的人、帮助我撰写第 1 版的人，现在都值得再次感谢。没有第 1 版的成功，就不会有第 2 版。

第 1 版致谢

Josef Hartwig 设计的包豪斯国际象棋套装体现了最佳的设计理念：美观、简洁、清晰。在建筑师父亲和字体设计大师兄弟的影响下，Guido van Rossum 设计出了一门经典的编程语言。我之所以热衷于教授 Python，也正是因为它美观、简洁、清晰。

Alex Martelli 和 Anna Ravenscroft 是最先看到本书大纲的人，他们鼓励我把大纲提交给 O'Reilly 出版社。他们写的书不但向我展示了地道的 Python 代码，还让我见识了什么才称得上是清晰、准确和有深度的技术写作。Alex 在 Stack Overflow 上回答了 6200 多个问题，可见他对 Python 语言及其用法有深入的见解。

Martelli 和 Ravenscroft 还是本书的技术审校。优秀的技术审校团队还有 Lennart Regebro 和 Leonardo Rochaël，4 人都至少有 15 年 Python 经验，为许许多多具有广泛影响力的 Python 项目贡献过代码，并且跟社区里的其他开发者联系紧密。4 位审校一共提出了数百个修订、建议、问题和观点，为本书增色颇多。另外，`asyncio` 维护者 Victor Stinner 审校了第 21 章，提升了技术审校团队的权威性。在过去的几个月里能够跟他们合作，我倍感荣幸。

本书编辑 Meghan Blanchette 是一位出色的导师。她不但帮助我梳理整本书的结构、增强内容的连贯性，还为我指出哪里写得不够有趣，时刻督促我及时交稿。Brian MacDonald 在 Meghan 休假的时候帮忙编辑了第二部分中的章节。跟他们以及 O'Reilly 的所有人打交道的过程都十分愉快。另外，Atlas 系统的开发和支持团队也很棒（Atlas 是 O'Reilly 的图书出版平台，本书有幸使用这个平台撰写）。

Mario Domenech Goulart 在看过本书首个抢读版之后，提出了无数的详细建议。另外我还从 Dave Pawson、Elias Dorneles、Leonardo Alexandre Ferreira Leite、Bruce Eckel、J. S. Bueno、Rafael Gonçalves、Alex Chiaranda、Guto Maia、Lucas Vido 和 Lucas Brunialti 那里获得了宝贵的反馈。

过去几年，很多人劝我写书，Rubens Prates、Aurelio Jargas、Rudá Moura 和 Rubens Altimari 算是最有说服力的。Mauricio Bussab 为我

打开了很多机会之门，包括第一次真正尝试撰写一本书。Renzo Nuccitelli 自始至终大力支持这个写作项目，即使这样做拖慢了 Dev Pro 的建设步伐，也毫无怨言。

Python 巴西社区是一个集思广益、乐于分享且充满乐趣的地方。Python 巴西小组成员数以千计，每次全国范围的会议都会把成百上千人聚集在一起。在我的 Python 成长之路上，对我影响最大的人有：Leonardo Rochaël、Adriano Petrich、Daniel Vainsencher、Rodrigo RBP Pimentel、Bruno Gola、Leonardo Santagada、Jean Ferri、Rodrigo Senra、J. S. Bueno、David Kwast、Luiz Irber、Osvaldo Santana、Fernando Masanori、Henrique Bastos、Gustavo Niemayer、Pedro Werneck、Gustavo Barbieri、Lalo Martins、Danilo Bellini 和 Pedro Kroger。

Dorneles Tremea 是非常棒的朋友（他很愿意花时间分享知识），不但是很厉害的黑客，而且还是巴西 Python 协会最鼓舞人心的领导者。可惜他过早离开了我们。

我的学生们同时也是我的老师，他们的问题、见解、反馈和那些富有创造性的解答教会了我很多。Érico Andrei 和 Simples Consultoria 让我第一次感受到，我可以在 Python 教师这条道路上继续走下去。

Martijn Faassen 是我的 Grok 导师，同我分享了很多关于 Python 和尼安德特人的想法。Martijn 所做的事情，还有来自 Zope、Plone 和 Pyramid 文章聚合平台的 Paul Everitt、Chris McDonough、Tres Seaver、Jim Fulton、Shane Hathaway、Lennart Regebro、Alan Runyan、Alexander Limi、Martijn Pieters 和 Godefroid Chapelle 等人所做的事情，在我事业发展的过程中起到了决定性的作用。多亏了 Zope 和第一波互联网浪潮，让我在 1998 年就开始从事 Python 相关的工作并以此为生。José Octavio Castro Neves 是我的搭档，我们在巴西开了第一家以 Python 业务为主的软件公司。

Python 社区高手如云，实在没办法一一列出。除了前面提到的之外，我还要感谢 Steve Holden、Raymond Hettinger、A.M. Kuchling、

David Beazley、Fredrik Lundh、Doug Hellmann、Nick Coghlan、Mark Pilgrim、Martijn Pieters、Bruce Eckel、Michele Simionato、Wesley Chun、Brandon Craig Rhodes、Philip Guo、Daniel Greenfeld、Audrey Roy 和 Brett Slatkin，感谢他们让我见识到更新更好的 Python 教学方法。

这本书基本上是在我的书房和两个公共场所写成的。CoffeeLab 位于巴西圣保罗 Vila Madalena 区，是一个咖啡极客大本营。Garoa Hacker Clube 是一个开放的黑客空间，任何人都可以在这里实验他们的新点子。

Garoa 社区还为我提供了灵感、基础设施和放松的环境。我想 Aleph 会喜欢这本书的。

我的母亲 Maria Lucia 和父亲 Jairo 一直以各种方式支持我。真希望我的父亲还健在，能够看到本书出版。可喜的是，我的母亲能见证这一刻。

在写这本书的 15 个月里，身为丈夫的我几乎一直在工作，我的妻子 Marta Mello 陪我一起熬过了这段日子。在这如同马拉松一样漫长的写作过程中，她不但一直支持我，而且在我想要放弃的时候陪我一起渡过难关。

感谢每一个人，感谢你们所做的一切。

电子书

扫描如下二维码，即可购买本书中文版电子书。



第一部分 数据结构

- 第 1 章 Python 数据模型
- 第 2 章 丰富的序列
- 第 3 章 字典和集合
- 第 4 章 Unicode 文本和字节序列
- 第 5 章 数据类构建器
- 第 6 章 对象引用、可变性和垃圾回收

第 1 章 Python 数据模型

Guido 对语言设计美学的深入理解让人震惊。我认识不少不错的编程语言设计者，他们设计出来的东西理论上确实精彩，却无人问津。Guido 知道如何在理论上做出一定妥协，设计出来的语言让使用者觉得如沐春风，这真是不可多得。

——Jim Hugunin

Jython 作者，AspectJ 作者之一，.NET DLR 架构师¹

¹摘自“Story of Jython”，*Jython Essentials*（Samuele Pedroni 和 Noel Rappin 著）一书的前序。

Python 的质量保障得益于一致性。使用 Python 一段时间之后，便可以根据自己掌握的知识，正确地猜出新功能的作用。

然而，如果你在接触 Python 之前有其他面向对象语言的经验，就会觉得奇怪：为什么获取容器大小不使用 `collection.len()`，而是使用 `len(collection)`？这一点表面上看确实奇怪，而且只是众多奇怪行为的冰山一角，不过知道背后的原因之后，你会发现这才真正符合“Python 风格”。一切的一切都埋藏在 Python 数据模型中。我们平常自己创建对象时就要使用这个 API，确保使用最地道的语言功能。

可以把 Python 视为一个框架，而数据模型就是对框架的描述，规范语言自身各个组成部分的接口，确立序列、函数、迭代器、协程、类、上下文管理等部分的行为。

使用框架要花大量时间编写方法，交给框架调用。利用 Python 数据模型构建新类也是如此。Python 解释器调用特殊方法来执行基本对象操作，通常由特殊句法触发。特殊方法的名称前后两端都有双下划线。例如，在 `obj[key]` 句法背后提供支持的是特殊方法

`__getitem__`。为了求解 `my_collection[key]`，Python 解释器要调用 `my_collection.__getitem__(key)`。

如果想让对象支持以下基本的语言结构并与其交互，就需要实现特殊方法：

- 容器；
- 属性存取；
- 迭代（包括使用 `async for` 的异步迭代）；
- 运算符重载；
- 函数和方法调用；
- 字符串表示形式和格式化；
- 使用 `await` 的异步编程；
- 对象创建和析构；
- 使用 `with` 或 `async with` 语句管理上下文。



魔法方法和双下划线

特殊方法用行话说叫作**魔术方法**（magic method）。需要把一个特殊方法（例如 `__getitem__`）说出来时，应该怎么表达呢？

我一般说“dunder-getitem”，这是跟著名作家和教师 Steve Holden 学的。“dunder”表示“前后双下划线”。因此，特殊方法也叫“双下划线方法”。《Python 语言参考手册》中的第 2 章“词法分析”警告道：“**任何时候**，若不遵守文档明确说明的方式使用 `__*__` 名称，一切后果自负。”

1.1 本章新增内容

相较于第 1 版，本章内容改动较少，毕竟 Python 数据模型相当稳定。本章主要改动如下。

- 1.4 节中的表格增加了支持异步编程和其他新功能的特殊方法。

- 新增 1.3.4 节，在图 1-2 中给出容器相关的特殊方法，包括 Python 3.6 引入的 `collections.abc.Collection` 抽象基类。

另外，本章和第 2 版其他章节统一采用 Python 3.6 引入的 **f 字符串** 句法。这种句法可读性更好，而且通常比 `str.format()` 方法和 `%` 运算符等旧的字符串格式化表示法更方便。



如果 `my_fmt` 的定义与格式化操作在代码的不同位置，那就可以继续使用 `my_fmt.format()`。比如说，`my_fmt` 的内容占据多行、更适合定义为常量，或者必须从配置文件或数据库中读取。这些情况确实存在，但很少见。

1.2 一摞 Python 风格的纸牌

示例 1-1 虽然简单，却展示了实现 `__getitem__` 和 `__len__` 两个特殊方法之后得到的强大功能。

示例 1-1 一摞有序的纸牌

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])
class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

首先用 `collections.namedtuple` 构建了一个简单的类，表示单张纸牌。使用 `namedtuple` 构建只有属性而没有自定义方法的类对象，例如数据库中的一条记录。这个示例中使用这个类表示一摞牌中的各张纸牌，如以下控制台会话所示。

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

但是，这个示例的重点是简短精炼的 `FrenchDeck` 类。首先，与标准的 Python 容器一样，一摞牌响应 `len()` 函数，返回一摞牌有多少张。

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

得益于 `__getitem__` 方法，我们可以轻松地从此摞牌中抽取某一张，比如说第一张或最后一张。

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

如果想随机选一张牌，需要定义一个方法吗？不需要，因为 Python 已经提供了从序列中随机获取一项的函数，即 `random.choice`。我们可以在一摞牌上使用这个函数。

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

可以看到，通过特殊方法利用 Python 数据模型，这样做有两个优点。

- 类的用户不需要记住标准操作的方法名称（“怎样获取项数？使用 `.size()`、`.length()`，还是其他方法？”）。
- 可以充分利用 Python 标准库，例如 `random.choice` 函数，无须重新发明轮子。

好戏还在后面。

由于 `__getitem__` 方法把操作委托给 `self._cards` 的 `[]` 运算符，一摞牌自动支持切片（slicing）。下面展示如何从一摞新牌中抽取最上面三张，再从索引 12 位开始，跳过 13 张牌，只抽取 4 张 A。

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

实现特殊方法 `__getitem__` 之后，这摞纸牌还可以迭代。

```
>>> for card in deck: # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

另外，也可以反向迭代这摞纸牌。

```
>>> for card in reversed(deck): # doctest: +ELLIPSIS
...     print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```



doctest 中的省略号

本书中的 Python 控制台会话内容从 doctest 中摘录，力求准确无误。如果输出太长，则内容有所节略。节略的部分使用省略号 (...) 标记，例如前一段代码中的最后一行。这种情况下，为了让 doctest 通过，我加上了 # doctest: +ELLIPSIS。在交互式控制台中试验这些示例时，可以把 doctest 注释全部去掉。

迭代往往是隐式的。如果一个容器没有实现 `__contains__` 方法，那么 `in` 运算符就会做一次顺序扫描。本例就是这样，`FrenchDeck` 类支持 `in` 运算符，因为该类可迭代。下面来试试。

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

那么排序呢？按照常规，牌面大小按点数（A 最大），以及黑桃（最大）、红心、方块、梅花（最小）的顺序排列。下面按照这个规则定义扑克牌排序函数，梅花 2 返回 0，黑桃 A 返回 51。

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)
def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

定义好 `spades_high` 函数后，现在按照牌面大小升序列出一副牌。

```
>>> for card in sorted(deck, key=spades_high): # doctest:
+ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards omitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```


虽然 `FrenchDeck` 类隐式继承 `object` 类，但是前者的多数功能不是继承而来的，而是源自数据模型和组合模式。通过前面使用 `random.choice`、`reversed` 和 `sorted` 的示例可以看出，实现 `__len__` 和 `__getitem__` 两个特殊方法后，`FrenchDeck` 的行为就像标准的 Python 序列一样，受益于语言核心特性（例如迭代和切片）和标准库。`__len__` 和 `__getitem__` 的实现利用组合模式，把所有工作委托给一个 `list` 对象，即 `self._cards`。



如何洗牌？

按照目前的设计，`FrenchDeck` 对象不能洗牌，因为它是**不可变**的：纸牌自身及其位置不能变化，除非违背封装原则，直接处理 `_cards` 属性。第 13 章将添加只有一行代码的 `__setitem__` 方法，解决这个问题。

1.3 特殊方法是如何使用的

首先要明确一点，特殊方法供 Python 解释器调用，而不是你自己。也就是说，没有 `my_object.__len__()` 这种写法，正确的写法是 `len(my_object)`。如果 `my_object` 是用户定义的类的实例，Python 将调用你实现的 `__len__` 方法。

然而，处理内置类型时，例如 `list`、`str`、`bytearray` 或 `NumPy` 数组等扩展，Python 解释器会抄个近路。Python 中可变长度容器的底层 C 语言实现中有一个结构体，² 名为 `PyVarObject`。在这个结构体中，`ob_size` 字段保存着容器中的项数。如果 `my_object` 是某个内置类型的实例，则 `len(my_object)` 直接读取 `ob_size` 字段的值，这比调用方法快很多。

²C 语言结构体是一种使用具名字段的记录类型。

很多时候，特殊方法是隐式调用的。例如，`for i in x:` 语句其实在背后调用 `iter(x)`，接着又调用 `x.__iter__()`（前提是有该方法）或 `x.__getitem__()`。在 `FrenchDeck` 示例中，调用的是后者。

我们在编写代码时一般不直接调用特殊方法，除非涉及大量元编程。即便如此，大部分时间也是实现特殊方法，很少显式调用。唯一例外的是 `__init__` 方法，为自定义的类实现 `__init__` 方法时经常直接调用它调取超类的初始化方法。

如果需要调用特殊方法，则最好调用相应的内置函数，例如 `len`、`iter`、`str` 等。这些内置函数不仅调用对应的特殊方法，通常还提供额外服务，而且对于内置类型来说，速度比调用方法更快。17.3 节有一个示例。

接下来几节会说明特殊方法最重要的用途：

- 模拟数值类型；
- 对象的字符串表示形式；
- 对象的布尔值；
- 实现容器。

1.3.1 模拟数值类型

有几个特殊方法可以让用户对象响应 `+` 等运算符。第 16 章对此有详细探讨，这里只是借此再举一个简单的例子，说明特殊方法的用途。

接下来将实现一个二维向量类，即数学和物理中使用的欧几里得向量（见图 1-1）。



内置的 `complex` 类型可用于表示二维向量，不过我们实现的类经过扩展可以表示 n 维向量，详见第 17 章。

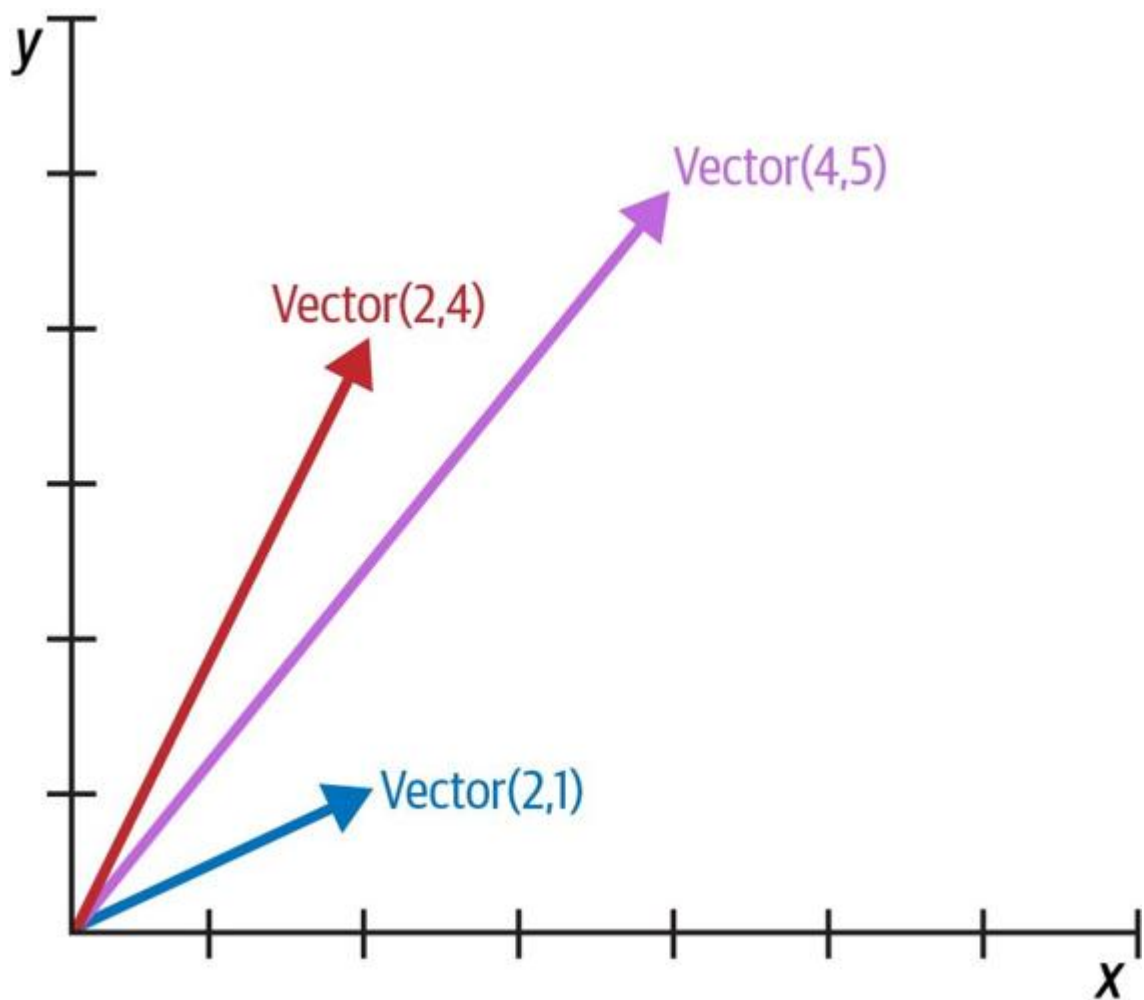


图 1-1：二维向量加法图示：Vector(2, 4) + Vector(2, 1) = Vector(4, 5)

为了给这个类设计 API，先写出模拟的控制台会话，作为 doctest。以下代码片段测试图 1-1 中的向量加法。

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

注意，+ 运算符的结果是一个新 Vector 对象，在控制台中以友好的格式显示。

内置函数 `abs` 返回整数和浮点数的绝对值，以及复数的模。为了保持一致性，我们的 API 也使用 `abs` 函数计算向量的模。

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

还可以实现 `*` 运算符，计算向量的标量积（即一个向量乘以一个数，得到一个方向相同、模为一定倍数的新向量）。

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

示例 1-2 使用 `__repr__`、`__abs__`、`__add__` 和 `__mul__` 等特殊方法为 `Vector` 类实现这几种运算。

示例 1-2 一个简单的二维向量类

```
"""
vector2d.py: 一个简单的类，演示一些特殊方法

只是演示，一些问题做简化处理。缺少错误处理，尤其是__add__和__mul__方法。

本书后文还会扩充这个示例。

加法::

    >>> v1 = Vector(2, 4)
    >>> v2 = Vector(2, 1)
    >>> v1 + v2
    Vector(4, 5)

绝对值::

    >>> v = Vector(3, 4)
    >>> abs(v)
    5.0

标量积::
```

```

>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0

"""

import math

class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

```

除了我们熟悉的 `__init__` 方法，这个类还实现了另外 5 个特殊方法。注意，这些方法在类内部，或者在前面的 `doctest` 中都没有直接调用。正如前文所说，多数特殊方法最常被 Python 解释器调用。

示例 1-2 实现了 `+` 和 `*` 两个运算符，展示了 `__add__` 和 `__mul__` 方法的基本用法。这两个方法创建并返回一个新 `Vector` 实例，没有修改运算对象，只是读取 `self` 或 `other`。这是中缀运算符的预期行为，即创建新对象，不修改运算对象。这一点会在第 16 章详谈。



按照示例 1-2 中的实现，一个 `Vector` 对象可以乘以一个数，但是一个数不能乘以一个 `Vector` 对象，这违背了标量积的

交换律。这个问题在第 16 章会使用特殊方法 `__rmul__` 解决。

接下来的几节讨论 `Vector` 类的其他特殊方法。

1.3.2 字符串表示形式

特殊方法 `__repr__` 供内置函数 `repr` 调用，获取对象的字符串表示形式。如未定义 `__repr__` 方法，`Vector` 实例在 Python 控制台中显示为 `<Vector object at 0x10e100070>` 形式。

交互式控制台和调试器在表达式求值结果上调用 `repr` 函数，处理方式与使用 `%` 运算符处理经典格式化方式中的 `%r` 占位符，以及使用 `str.format` 方法处理新字符串格式化句法中的 `!r` 转换字段一样。

注意，`Vector` 类 `__repr__` 方法中的 `f` 字符串使用 `!r` 以标准的表示形式显示属性。这样做比较好，因为 `Vector(1, 2)` 和 `Vector('1', '2')` 之间是有区别的，后者在这个示例中不可用，因为构造函数接受的参数是数值而不是字符串。

`__repr__` 方法返回的字符串应当没有歧义，如果可能，最好与源码保持一致，方便重新创建所表示的对象。鉴于此，我们才以类似构造函数的形式（例如 `Vector(3, 4)`）返回 `Vector` 对象的字符串表示形式。

与此形成对照的是，`__str__` 方法由内置函数 `str()` 调用，在背后供 `print` 函数使用，返回对终端用户友好的字符串。

有时，`__repr__` 方法返回的字符串足够友好，无须再定义 `__str__` 方法，因为继承自 `object` 类的实现最终会调用 `__repr__` 方法。本书中有几个示例定义了 `__str__` 方法，例如示例 5-2。



如果你熟悉的编程语言使用 `toString` 方法，那么你可能习惯实现 `__str__` 方法而不是 `__repr__` 方法。在 Python 中，如果必须二选一的话，请选择 `__repr__` 方法。

Stack Overflow 网站中有一个问题，“What is the difference between `__str__` and `__repr__`?”，Python 专家 Alex Martelli 和 Martijn Pieters 对此做出了详尽解答。

1.3.3 自定义类型的布尔值

Python 有一个 `bool` 类型，在需要布尔值的地方处理对象，例如 `if` 或 `while` 语句的条件表达式，或者 `and`、`or` 和 `not` 的运算对象。为了确定 `x` 表示的值为真或为假，Python 调用 `bool(x)`，返回 `True` 或 `False`。

默认情况下，用户定义类的实例都是真值，除非实现了 `__bool__` 或 `__len__` 方法。简单来说，`bool(x)` 调用 `x.__bool__()`，以后者返回的结果为准。如果没有实现 `__bool__` 方法，则 Python 尝试调用 `x.__len__()`；如果该方法返回零值，则 `bool` 函数返回 `False`，否则返回 `True`。

我们实现的 `__bool__` 方法没用到什么高深的理论，如果向量的模为零，则返回 `False`，否则返回 `True`。我们使用 `bool(abs(self))` 把向量的模转换成布尔值，因为 `__bool__` 方法必须返回一个布尔值。在 `__bool__` 方法外部，很少需要显式调用 `bool()`，因为任何对象都可在布尔值上下文中使用。

注意，这个 `__bool__` 特殊方法遵守 Python 标准库文档中“Built-in Types”一章里定义的真值测试规则。



`Vector.__bool__` 方法也可以像下面这样简单定义。

```
def __bool__(self):
    return bool(self.x or self.y)
```

这样定义虽然不易读懂，但是不用经过 `abs` 和 `__abs__` 处理，也无须计算平方和平方根。使用 `bool` 显式转换是有必要的，因为 `__bool__` 方法必须返回一个布尔值。`or` 返回两个操作数的其中一个，而且原封不动：如果 `x` 是真值，则 `x or y` 的求值结果为 `x`，否则为 `y`，无论 `y` 是真是假。

1.3.4 容器 API

Python 语言中基本容器类型的接口如图 1-2 所示。图中所有的类都是**抽象基类**。抽象基类和 `collections.abc` 模块将在第 13 章讨论。本节简要说明 Python 中最重要的容器接口，纵览容器类型对特殊方法的使用情况。

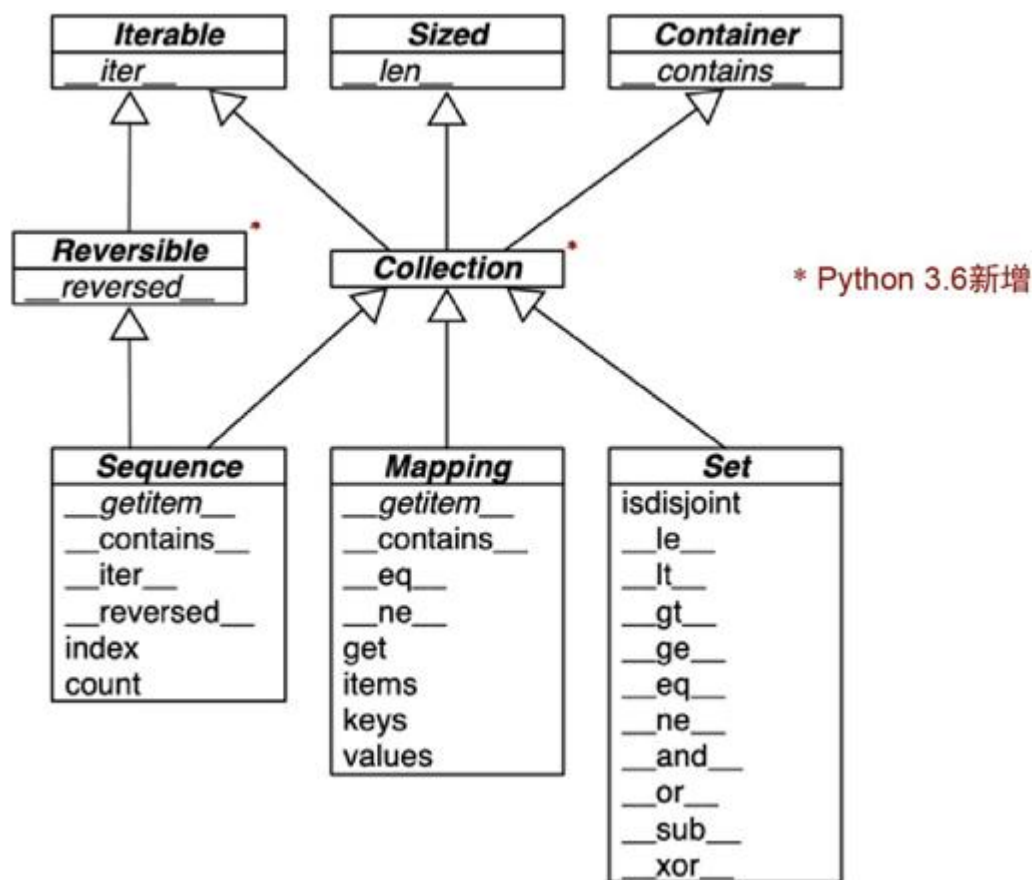


图 1-2：基本容器类型的 UML 类图。以斜体显示的方法名称表示抽象方法，必须由具体子类（例如 `list` 和 `dict`）实现。其他方法有具体实现，子类可以直接继承

顶部 3 个抽象基类均只有一个特殊方法。抽象基类 `Collection`（Python 3.6 新增）统一了这 3 个基本接口，每一个容器类型均应实现如下事项：

- `Iterable` 要支持 `for`、拆包和其他迭代方式；
- `Sized` 要支持内置函数 `len`；
- `Container` 要支持 `in` 运算符。

Python 不强制要求具体类继承这些抽象基类中的任何一个。只要实现了 `__len__` 方法，就说明那个类满足 `Sized` 接口。

`Collection` 有 3 个十分重要的专用接口：

- `Sequence` 规范 `list` 和 `str` 等内置类型的接口；
- `Mapping` 被 `dict`、`collections.defaultdict` 等实现；
- `Set` 是 `set` 和 `frozenset` 两个内置类型的接口。

只有 `Sequence` 实现了 `Reversible`，因为序列要支持以任意顺序排列内容，而 `Mapping` 和 `Set` 不需要。



自 Python 3.7 开始，`dict` 类型正式“有顺序”了，不过只是保留键的插入顺序。你不能随意重新排列 `dict` 中的键。

`Set` 抽象基类中的所有特殊方法实现的都是中缀运算符。例如，`a & b` 计算集合 `a` 和 `b` 的交集，该运算符由 `__and__` 特殊方法实现。

接下来的两章会详细说明标准库中的序列、映射和集合。

接下来按大类介绍 Python 数据模型定义的特殊方法。

1.4 特殊方法概述

《Python 语言参考手册》中的第 3 章列出了 80 多个特殊方法名称，其中一半以上用于实现算术运算符、按位运算符和比较运算符。下面几张表格概述了这些可用的特殊方法。

表 1-1 不包含实现中缀运算符和核心数学函数（例如 `abs`）的特殊方法。这里列出的多数特殊方法本书中有所讨论，包括新增的几个，比如 `__anext__`（Python 3.5 新增）等异步特殊方法，以及为类定义钩子的 `__init_subclass__`（Python 3.6 新增）。

表 1-1：特殊方法名称（不含运算符）

分类	方法名称
字符串（字节）表示形式	<code>__repr__</code> <code>__str__</code> <code>__format__</code> <code>__bytes__</code> <code>__fspath__</code>
转换为数值	<code>__bool__</code> <code>__complex__</code> <code>__int__</code> <code>__float__</code> <code>__hash__</code> <code>__index__</code>
模拟容器	<code>__len__</code> <code>__getitem__</code> <code>__setitem__</code> <code>__delitem__</code> <code>__contains__</code>
迭代	<code>__iter__</code> <code>__aiter__</code> <code>__next__</code> <code>__anext__</code> <code>__reversed__</code>
可调用对象或执行协程	<code>__call__</code> <code>__await__</code>
上下文管理	<code>__enter__</code> <code>__exit__</code> <code>__aexit__</code> <code>__aenter__</code>

分类	方法名称
构造和析构实例	<code>__new__</code> <code>__init__</code> <code>__del__</code>
属性管理	<code>__getattr__</code> <code>__getattribute__</code> <code>__setattr__</code> <code>__delattr__</code> <code>__dir__</code>
属性描述符	<code>__get__</code> <code>__set__</code> <code>__delete__</code> <code>__set_name__</code>
抽象基类	<code>__instancecheck__</code> <code>__subclasscheck__</code>
类元编程	<code>__prepare__</code> <code>__init_subclass__</code> <code>__class_getitem__</code> <code>__mro_entries__</code>

中缀运算符和数字运算符由表 1-2 中的特殊方法提供支持。其中，`__matmul__`、`__rmatmul__` 和 `__imatmul__` 是 Python 3.5 新增的，用于实现矩阵乘法中缀运算符 `@`（见第 16 章）。

表 1-2：运算符的符号和背后的特殊方法

运算符分类	符号	方法名称
一元数值运算符	<code>-</code> <code>+</code> <code>abs()</code>	<code>__neg__</code> <code>__pos__</code> <code>__abs__</code>
各种比较运算符	<code><</code> <code><=</code> <code>==</code> <code>!=</code> <code>></code> <code>>=</code>	<code>__lt__</code> <code>__le__</code> <code>__eq__</code> <code>__ne__</code> <code>__gt__</code> <code>__ge__</code>

运算符分类	符号	方法名称
算术运算符	+ - * / // % @ divmod() round() ** pow()	__add__ __sub__ __mul__ __truediv__ __floordiv__ __mod__ __matmul__ __divmod__ __round__ __pow__
反向算术运算符	(交换算术运算符的操作数)	__radd__ __rsub__ __rmul__ __rtruediv__ __rfloordiv__ __rmod__ __rmatmul__ __rdivmod__ __rpow__
增量赋值算术运算符	+= -= *= /= //= %= @= *=	__iadd__ __isub__ __imul__ __itruediv__ __ifloordiv__ __imod__ __imatmul__ __ipow__
按位运算符	& ^ << >> ~	__and__ __or__ __xor__ __lshift__ __rshift__ __invert__
反向按位运算符	(交换按位运算符的操作数)	__rand__ __ror__ __rxor__ __rlshift__ __rrshift__
增量赋值按位运算符	&= = ^= <<= >>=	__iand__ __ior__ __ixor__ __ilshift__ __irshift__



如果第一个操作数对应的特殊方法不可用，则 Python 在第二个操作数上调用反向运算符对应的特殊方法。增量赋值是结合了变量赋值功能的中缀运算符的简写形式，例如 `a += b`。

第 16 章会详细说明反向运算符和增量赋值。

1.5 len 为什么不是方法

这个问题我在 2013 年问过核心开发人员 Raymond Hettinger，他在回答时引用了《Python 之禅》中的一句话，道出了玄机：“实用胜过纯粹。”1.3 节说过，当 `x` 是内置类型的实例时，`len(x)` 运行速度非常快。计算 CPython 内置对象的长度时不调用任何方法，而是直接读取 C 语言结构体中的字段。获取容器中的项数是一项常见操作，`str`、`list`、`memoryview` 等各种基本的容器类型必须高效完成这项工作。

换句话说，`len` 之所以不作为方法调用，是因为它经过了特殊处理，被当作 Python 数据模型的一部分，就像 `abs` 函数一样。但是，借助特殊方法 `__len__`，也可让 `len` 适用于自定义对象。这是一种相对公平的折中方案，既满足了内置对象对速度的要求，又保证了语言的一致性。这也体现了《Python 之禅》中的另一句话：“特殊情况不是打破规则的理由。”



忘掉面向对象语言中的方法调用句法，把 `abs` 和 `len` 看作一元运算符，说不定你更能接受它们表面上看似对函数的调用。Python 源自 ABC 语言，很多特性继承自 ABC。ABC 中的 `#` 运算符与 Python 中的 `len` 作用相同，写作 `#s`。`#` 也可作为中缀运算符使用，写作 `x#s`，计算 `s` 中有多少个 `x`。对应到 Python 中，写作 `s.count(x)` (`s` 是某种序列)。

1.6 本章小结

借助特殊方法，自定义对象的行为可以像内置类型一样，让我们写出更具表现力的代码，符合社区所认可的 Python 风格。

Python 对象基本上都需要提供一个有用的字符串表示形式，在调试、登记日志和向终端用户展示时使用。鉴于此，数据模型中才有

`__repr__` 和 `__str__` 两个特殊方法。

模拟序列的行为（例如 `FrenchDeck` 示例）是特殊方法最常见的用途之一。比如说，数据库代码库返回的查询结果往往就是一个类似序列的容器。第 2 章会具体说明如何充分利用现有的序列类型。自己实现序列类型的方式在第 12 章讲解，届时我们将在 `Vector` 类的基础上创建一个多维向量类。

得益于运算符重载，Python 提供了丰富的数值类型，除内置的数值类型之外，还有 `decimal.Decimal` 和 `fractions.Fraction`，全都支持中缀算术运算符。数据科学库 NumPy 还提供了矩阵和张量的中缀运算符。第 16 章增强 `Vector` 类的示例将实现运算符，包括反向运算符和增量赋值运算符。

Python 数据模型中余下的特殊方法如何使用和实现，本书大部分有所涵盖。

1.7 延伸阅读

本章及本书大部分内容参考了《Python 语言参考手册》中的第 3 章“数据模型”，这是最权威的资料。

Python in a Nutshell, 3rd ed. (Alex Martelli、Anna Ravenscroft 和 Steve Holden 著) 对数据模型的讲解很精彩。书中对属性访问机制的解说是我见过除 CPython 源码之外最权威的。Martelli 还经常在 Stack Overflow 中回答问题，目前已回答 6200 多个问题。不信的话，可以看看他在 Stack Overflow 中的个人资料。

David Beazley 著有两本基于 Python 3 的书，对数据模型做了详尽的介绍。一本是《Python 参考手册（第 4 版）》，另一本是与 Brian K. Jones 合著的《Python Cookbook 中文版（第 3 版）》。

The Art of the Metaobject Protocol (Gregor Kiczales、Jim des Rivieres 和 Daniel G. Bobrow 著) 解读了元对象协议的概念，Python 数据模型就是其中一个例子。

杂谈

数据模型还是对象模型

Python 文档中的说法是“Python 数据模型”，而多数作者采用的说法是“Python 对象模型”。*Python in a Nutshell, 3rd ed.* 以及《Python 参考手册（第 4 版）》都对 Python 数据模型进行了深入解读，不过这几位作者用的均是“对象模型”。维基百科给对象模型的第一个定义是“一门计算机编程语言中对象的一般特性”。这正是“Python 数据模型”所要描述的概念。本书采用“数据模型”这一说法，因为 Python 文档始终使用这个词指代 Python 对象模型，而且还因为《Python 语言参考手册》中与本书关系最大的那一章使用的标题就是“数据模型”。

麻瓜方法

按照 *The Original Hacker's Dictionary* 的定义，“魔法”的意思是“神秘莫测，或复杂到说不清”，或者“鲜为人知的功能，让不可能成为可能”。

Ruby 中也有类似“特殊方法”的概念，Ruby 社区称之为“魔法方法”。Python 社区也有不少人采用这种说法。而我认为，“特殊方法”与“魔法方法”是对立的。Python 和 Ruby 都利用这个概念丰富元对象协议，即便是你我这种不会“魔法”的麻瓜，凭借完善的文档，也能模拟核心开发人员编写语言解释器时用到的很多功能。

Go 语言就不一样了。在这门语言中，一些对象的功能确实像魔法，因为用户自己定义的类型无法模拟。例如，Go 语言中的数组、字符串和映射支持使用方括号存取项，写作 `a[i]`。但是，我们自己定义的容器类型无法使用 `[]` 表示法。更糟的是，Go 语

言没有可迭代接口或迭代器对象之类的概念，因此 `for/range` 句法仅支持 5 种“魔法”内置类型，包括数组、字符串和映射。

以后 Go 语言的设计人员说不定会增强元对象协议。但是目前来看，与 Python 或 Ruby 相比，它的功能十分有限。

元对象

The Art of the Metaobject Protocol（以下简称 AMOP）是我最喜欢的一本计算机图书。我提到这本书是因为“元对象协议”对理解 Python 数据模型有帮助，而且其他语言中也有类似的功能。“元对象”指构成语言自身的基本对象。在这个语境下，“协议”等同于“接口”。所以，“元对象协议”就是对象模型的高级说法，指语言核心构件的 API。

一套丰富的元对象协议能让我们扩展语言，支持新的编程范式。AMOP 的第一作者 Gregor Kiczales 后来成为面向方面的程序设计（aspect-oriented programming）的先驱，也是 AspectJ（实现该范式的 Java 扩展）的最初作者。面向方面的程序设计的 Python 这样的动态语言中实现起来更简单，一些框架就提供了这个范式，Plone 内容管理系统中的 `zope.interface` 就是一例。

第 2 章 丰富的序列

你可能已经注意到了，前面几种操作对文本、列表和表格的处理方式没有差异。文本、列表和表格统称“行列”。.....FOR 命令同样可处理行列。

——Leo Geurts、Lambert Meertens 和 Steven Pemberton
ABC Programmer's Handbook, 第 8 页

在 Python 诞生以前，Guido 为 ABC 语言贡献过代码。ABC 语言是一个历时 10 年的研究项目，旨在设计一种适合初学者的编程环境。现在看来，ABC 语言中的很多理念比较符合 Python 风格，例如对各种序列一视同仁、内置元组和映射类型、源码结构通过缩进实现、无须声明变量的强类型等。可见，Python 对用户友好的特点是有渊源的。

Python 从 ABC 语言继承了对序列的统一处理方式。字符串、列表、字节序列、数组、XML 元素和数据库查询结果，这些序列在操作上有很多共通之处，都可以迭代、切片、排序和拼接。

深入理解 Python 中不同的序列类型，不但能避免重新发明轮子，还可以从它们共通的接口上受到启发，在自己实现 API 时合理支持及利用现有和将来可能添加的序列类型。

本章讨论的内容大多适用于一般意义上的序列，从我们熟悉的 `list`，到 Python 3 新增的 `str` 和 `bytes`。本章重点讲解列表、元组、数组和队列，Unicode 字符串和字节序列相关的话题在第 4 章探讨。另外，本章关注的是 Python 中现成可用的序列类型，自己创建序列类型是第 12 章的话题。

本章主要涵盖以下内容：

- 列表推导式和生成器表达式基础知识；
- 元组的两种用法——记录和不可变列表；
- 序列拆包和序列模式；
- 读写切片；
- 专门的序列类型，例如数组和队列。

2.1 本章新增内容

本章最大的变化是新增了 2.6 节。这是 Python 3.10 新引入的模式匹配首次出现在第 2 版中。

与第 1 版相比，其他改动不涉及重大更新，只做了一些改进：

- 增加有关序列内部机制的图表和说明，增加容器和扁平序列的对比；
- 简要比较 `list` 和 `tuple` 的性能和存储特点；
- 包含可变元素的元组需要注意的事项和应对方法。

具名元组相关的话题移到了 5.3 节，与 `typing.NamedTuple` 和 `@dataclass` 放在一起讲。

2.2 内置序列类型概览

Python 标准库用 C 语言实现了丰富的序列类型，列举如下。

容器序列

可存放不同类型的项，其中包括嵌套容器。示例：`list`、`tuple` 和 `collections.deque`。

扁平序列

可存放一种简单类型的项。示例：`str`、`bytes` 和 `array.array`。

容器序列存放的是所包含对象的引用，对象可以是任何类型。**扁平序列**在自己的内存空间中存储所含内容的值，而不是各自不同的 Python 对象。详见图 2-1。

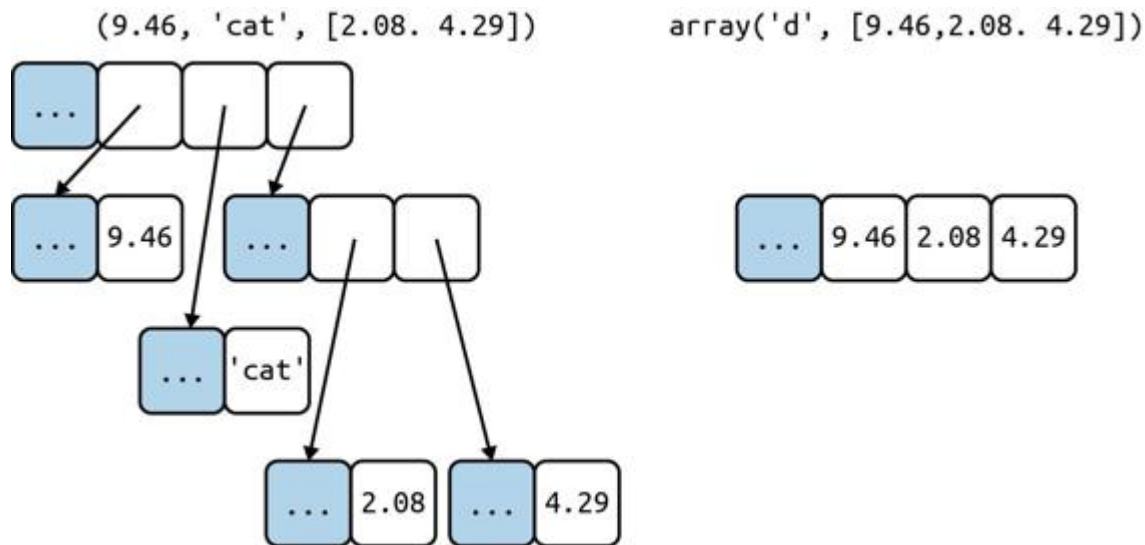


图 2-1：图中展示的是一个元组和一个数组的内存简图，它们各有 3 项。灰色方块（未按比例绘制）表示各个 Python 对象的内存标头。元组中的每一项都是引用，引用的是不同的 Python 对象，对象中还可以存放其他 Python 对象的引用，例如那个包含两个项的列表。相比之下，Python 中的数组整体是一个对象，存放一个 C 语言数组，包含 3 个双精度数

因此，扁平序列更加紧凑，但是只能存放原始机器值，例如字节、整数和浮点数。



任何 Python 对象在内存中都有一个包含元数据的标头。最简单的 Python 对象，例如一个 `float`，内存标头中有一个值字段和两个元数据字段。

- `ob_refcnt`：对象的引用计数。

- `ob_type`: 指向对象类型的指针。
- `ob_fval`: 一个 C 语言 `double` 类型值, 存放 `float` 的值。

在 64 位设备中, 每个字段占 8 字节。假如有一个浮点数数组和一个浮点数元组, 显然前者比后者更紧凑, 因为数组整体是一个对象, 存放各个浮点数的原始值, 而元组由多个对象构成: 元组自身和存放的各个 `float` 对象。

另外, 还可按可变性对序列类型分类。

可变序列

例如 `list`、`bytearray`、`array.array` 和 `collections.deque`。

不可变序列

例如 `tuple`、`str` 和 `bytes`。

可变序列继承不可变序列的所有方法, 另外还多实现了几个方法, 如图 2-2 所示。内置的具体序列类型其实不是 `Sequence` 和 `MutableSequence` 抽象基类的子类, 而是一种**虚拟子类** (virtual subclass), 使用这两个抽象基类注册 (详见第 13 章)。因为是虚拟子类, 所以 `tuple` 和 `list` 可以通过以下测试。

```
>>> from collections import abc
>>> issubclass(tuple, abc.Sequence)
True
>>> issubclass(list, abc.MutableSequence)
True
```

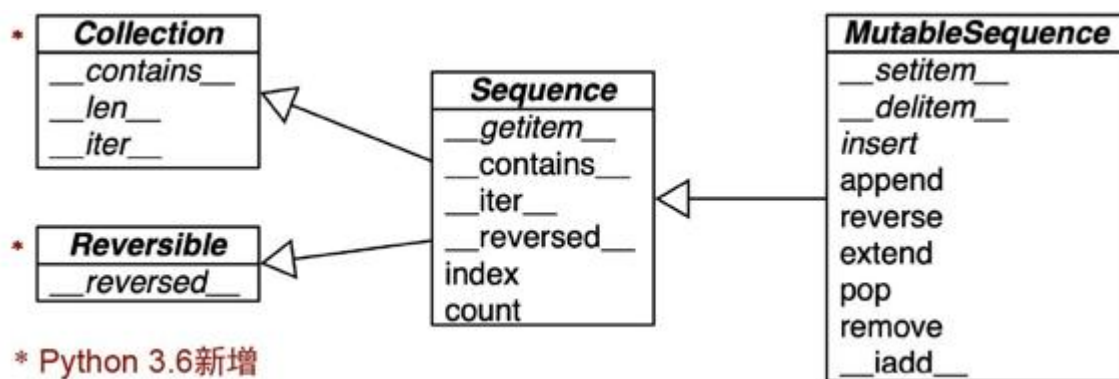


图 2-2: `collections.abc` 中部分类的简化 UML 类图（左边是超类；箭头从子类指向超类，表示继承；以斜体显示的名称是抽象类和抽象方法）

记住不同序列类型的共同点：有些是可变的，有些是不可变的；有些是容器，有些是扁平的。这有助于你把相关概念延伸到不太熟悉的序列类型上。

`list` 是最基本的序列类型，是一种可变容器。我假定你已经非常熟悉列表，因此接下来我会跳过基础，直接讲**列表推导式**（list comprehension）。这是构建列表的有力方式，句法乍一看晦涩难懂，因此往往被我们忽视。掌握列表推导式之后，我们便打开了**生成器表达式**（generator expression）的大门。生成器表达式功能强大，例如可以生成元素，填充任何类型的序列。这两个话题将在 2.3 节展开。

2.3 列表推导式和生成器表达式

使用列表推导式（目标是列表）或生成器表达式（目标是其他序列类型）可以快速构建一个序列。使用这两种句法写出的代码更易于理解，而且速度通常更快。如果你很少使用，那可真是遗憾。

这些结构体真的能让代码“更易于理解”吗？感到怀疑很正常，请继续往下读，我会尝试说服你。



很多 Python 程序员把列表推导式简称为 listcomps，把生成器表达式简称为 genexps。我有时也会这么做。¹

¹中文版不采用这种简写形式。——译者注

2.3.1 列表推导式对可读性的影响

请你切身体验一下，示例 2-1 和示例 2-2 哪一个更易于理解？

示例 2-1 基于一个字符串构建一个 Unicode 码点列表

```
>>> symbols = '$¢£¥€α'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

示例2-2 使用列表推导式基于一个字符串构建一个 Unicode 码点列表

```
>>> symbols = '$¢£¥€α'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

稍微懂一点 Python 或许就能看懂示例 2-1，但是学会列表推导式后，我觉得示例 2-2 更易于理解，因其意图是明确的。

`for` 循环能胜任许多任务，可以遍历一个序列，统计项数或挑选部分项，可以计算总数和平均数，等等。示例 2-1 使用 `for` 循环构建一个列表。

当然，如果滥用列表推导式，写出的代码也不一定易于理解。我曾经见过一段 Python 代码，仅仅为了得到副作用，就用列表推导式重复执行代码段。如果你不打算使用生成的列表，那就不要使用列表推导

式句法。另外，列表推导式应保持简短。如果超过两行，那么最好把语句拆开，或者使用传统的 `for` 循环重写。写 Python 代码就跟写文章一样，没有什么硬性规则，这个尺度你得自己把握。



句法提示

Python 会忽略 `[]`、`{}` 和 `()` 内部的换行。因此，列表、列表推导式、元组、字典等结构完全可以分成几行来写，无须使用续行转义符 `\`。如果不小心在续行转义符后面多输入一个空格，那反而不起作用。另外，使用这 3 种括号定义字面量时，项与项之间使用逗号分隔，末尾的逗号将被忽略。因此，跨多行定义列表字面量时，最好在最后一项后面添加一个逗号。这样不仅能方便其他程序员为列表添加更多项，还可以减少代码差异给阅读带来的干扰。

列表推导式和生成器表达式的局部作用域

Python 3 中的列表推导式、生成器表达式，以及类似的集合推导式和字典推导式，`for` 子句中赋值的变量在局部作用域内。

然而，使用“海象运算符”`:=` 赋值的变量在推导式或生成器表达式返回后依然可以访问，这与函数内的局部变量行为不同。根据“PEP 572—Assignment Expressions”，`:=` 运算符赋值的变量，其作用域限定在函数内，除非目标变量使用 `global` 或 `nonlocal` 声明。²

```
>>> x = 'ABC'
>>> codes = [ord(x) for x in x]
>>> x ❶
'ABC'
>>> codes
[65, 66, 67]
>>> codes = [last := ord(c) for c in x]
>>> last ❷
```

```
67
>>> c ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

- ❶ x 毫发无损，绑定的值仍是 'ABC'。
- ❷ last 依然可以访问。
- ❸ c 消失了，因为它只存在于列表推导式内部。

²感谢读者 Tina Lapine 指出这一点。

列表推导式筛选和转换序列或其他可迭代类型中的项，以此构建列表。综合使用 `filter` 和 `map` 两个内置函数也能达到这样的效果，但是接下来你会发现，写出的代码不易理解。

2.3.2 列表推导式与 `map` 和 `filter` 比较

列表推导式涵盖 `map` 和 `filter` 两个函数的功能，写出的代码不像 Python 的 `lambda` 表达式那样晦涩难懂。请看示例 2-3。

示例 2-3 使用列表推导式和 `map/filter` 组合构建同一个列表

```
>>> symbols = '$¢£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord,
symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

我原以为使用 `map` 和 `filter` 速度更快，但是 Alex Martelli 告诉我事实并非如此，至少前面的示例不是这样。本书代码中的 `02-array-seq/listcomp_speed.py` 脚本对列表推导式和 `filter/map` 的速度进行了简单比较。

`map` 和 `filter` 将在第 7 章深入讨论。下面来看如何使用列表推导式计算笛卡儿积，即构建一个列表，其中包含两个或更多列表中的所有项组成的元组。

2.3.3 笛卡儿积

列表推导式可以根据两个或更多可迭代对象的笛卡儿积构建列表。笛卡儿积的每一项是一个元组，由输入的各个可迭代对象中的项构成。得到的列表长度等于输入的各个可迭代对象长度的乘积。详见图 2-3。

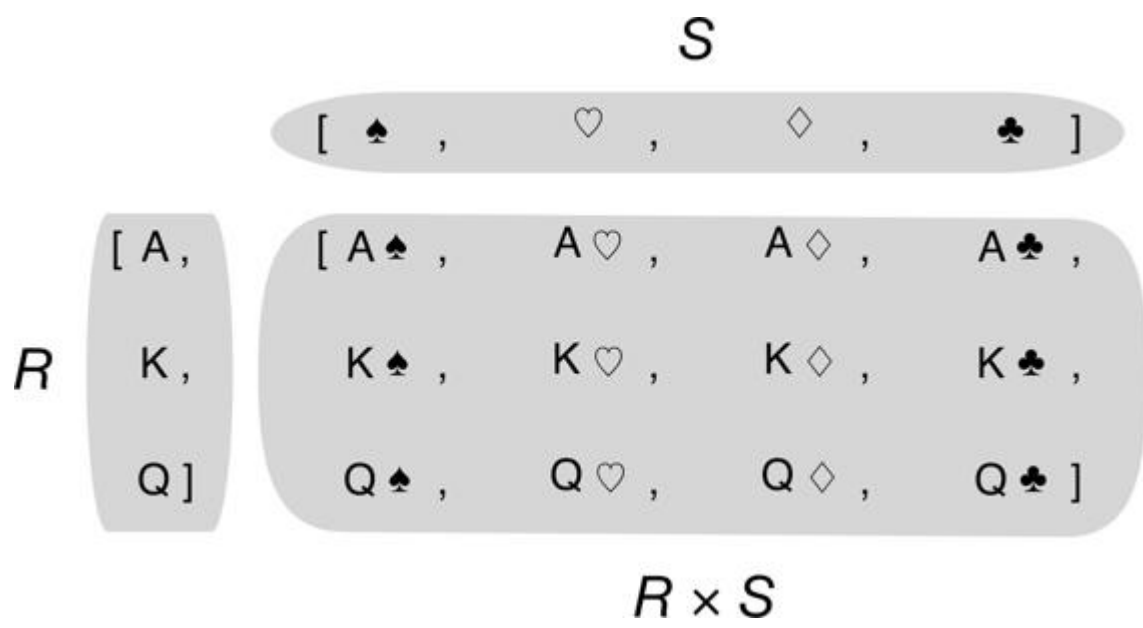


图 2-3：扑克牌的 3 种点数和 4 种花色的笛卡儿积是一个包含 12 种组合的序列

举个例子。假设你想生成一个列表，其中包含 2 种颜色和 3 种尺寸的 T 恤衫。示例 2-4 使用列表推导式生成这个列表，得到的列表有 6 项。

示例 2-4 使用列表推导式计算笛卡儿积

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
```

```

>>> tshirts = [(color, size) for color in colors for size in
sizes] ❶
>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors: ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes ❸
...             for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]

```

❶ 先按颜色再按尺寸排列，生成一个元组列表。

❷ 注意，这里两个循环的嵌套方式与列表推导式中 `for` 子句的先后顺序一样。

❸ 如果想先按尺寸再按颜色排列，则只需要调整 `for` 子句的顺序。我在这里插入了一个换行符，这样排列顺序就更明显了。

在第 1 章的示例 1-1 中，我使用以下表达式初始化一摞纸牌，这摞纸牌有 52 张，分为 13 种点数和 4 种花色，先按花色再按点数排列。

```

self._cards = [Card(rank, suit) for suit in self.suits
                for rank in self.ranks]

```

列表推导式的作用很单一，就是构建列表。如果想生成其他类型的序列，则应当使用生成器表达式。2.3.4 节会简要说明如何使用生成器表达式构建列表以外的序列。

2.3.4 生成器表达式

虽然列表推导式也可以生成元组、数组或其他类型的序列，但是生成器表达式占用的内存更少，因为生成器表达式使用迭代器协议逐个产出项，而不是构建整个列表提供给其他构造函数。

生成器表达式的句法跟列表推导式几乎一样，只不过把方括号换成圆括号而已。

示例 2-5 使用生成器表达式构建一个元组和一个数组。

示例 2-5 使用生成器表达式构建一个元组和一个数组

```
>>> symbols = '$¢£¥€α'
>>> tuple(ord(symbol) for symbol in symbols) ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols)) ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

❶ 如果生成器表达式是函数唯一的参数，则不需要额外再使用圆括号括起来。

❷ `array` 构造函数接受两个参数，因此必须在生成器表达式两侧加上圆括号。`array` 构造函数的第一个参数指定数组中数值的存储类型，详见 2.10.1 节。

示例 2-6 使用生成器表达式计算笛卡儿积，打印 2 种颜色 3 种尺寸的 T 恤衫组合。与示例 2-4 不同，这一次 6 种 T 恤衫组成的列表不在内存中构建，生成器表达式一次产出一项，提供给 `for` 循环。如果是两个各有 1000 项的列表，则使用生成器表达式计算笛卡儿积可以节省大量内存，因为不用先构建一个包含 100 万项的列表提供给 `for` 循环。

示例 2-6 使用生成器表达式计算笛卡儿积

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in (f'{c} {s}' for c in colors for s in sizes): ❶
```

```
...     print(tshirt)
...
black S
black M
black L
white S
white M
white L
```

❶ 生成器表达式逐个产出项。这个示例不会构建包含 6 种 T 恤衫组合的列表。



第 17 章会深入探讨生成器。本节只是简单展示如何使用生成器表达式构建列表以外的序列，以及如何输出不需要占用内存的结果。

接下来讨论 Python 中的另一种基本序列类型：元组。

2.4 元组不仅仅是不可变列表

有些 Python 入门教程把元组称为“不可变列表”，然而这没有完全概括元组的特点。元组有两个作用，除了可以作为不可变列表使用之外，还可用作没有字段名称的记录。后一种用法往往被忽略，那就先从这一点开始讲吧。

2.4.1 用作记录

用元组存放记录，元组中的一项对应一个字段的数据，项的位置决定数据的意义。

如果只把元组当作不可变列表，那么项数和项的顺序就变得可有可无。但是如果把元组当作字段的容器使用，那么项数通常是固定的，顺序也变得十分重要。

示例 2-7 把元组当作记录使用。注意，表达式中的元组都不能排序，否则元组携带的信息就会遭到破坏，因为字段的含义取决于字段在元组中的位置。

示例 2-7 把元组当作记录使用

```
>>> lax_coordinates = (33.9425, -118.408056) ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32_450, 0.66,
8014) ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ❸
...                 ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids): ❹
...     print('%s/%s' % passport) ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids: ❻
...     print(country)
...
USA
BRA
ESP
```

- ❶ 洛杉矶国际机场的经纬度。
- ❷ 东京市的一些信息：市名、年份、人口（单位：千）、人口变化（单位：百分比）和面积（单位：平方千米）。
- ❸ 一个元组列表，元组的形式为 (country_code, passport_number)。
- ❹ 在迭代列表的过程中，passport 变量被绑定到每个元组上。
- ❺ % 格式化运算符理解元组结构，把每一项当作不同的字段。
- ❻ for 循环知道如何获取元组中单独的每一项，这叫“拆包”。这里我们对第二项不感兴趣，因此把它赋值给虚拟变量 _。



一般习惯使用 `_` 表示虚拟变量。这个变量名称看似奇怪，却是有效的。然而，在 `match/case` 语句中，`_` 是通配符，匹配值但不绑定值（见 2.6 节）。另外，在 Python 控制台中，前一个命令的结果如果不是 `None`，则赋值给 `_`。

看到“记录”这个名称，我们往往认为它是一种带有具名字段的数据结构。第 5 章会提供两种创建具名字段元组的方式。

但是，通常没必要为了给字段指定一个名称而创建一个类。如果不使用索引访问字段，而只使用拆包，那就更没有必要了。在示例 2-7 中，我们仅使用一个语句就把 `('Tokyo', 2003, 32_450, 0.66, 8014)` 赋值给了 `city, year, pop, chg, area`。随后，在 `print` 函数的参数中，使用 `%` 运算符把 `passport` 元组中的各项赋值给格式化字符串中对应的占位符。这两处用到的都是**元组拆包**（tuple unpacking）。



Python 程序员大都使用“元组拆包”这种说法，不过**可迭代对象拆包**（iterable unpacking）出现的频率也在增加，例如“PEP 3132—Extended Iterable Unpacking”。

2.5 节会深入讲解拆包，内容不局限于元组，也涉及序列和一般意义上的可迭代对象。

现在，我们把 `tuple` 类当作 `list` 类的一种不可变的变体。

2.4.2 用作不可变列表

Python 解释器和标准库经常把元组当作不可变列表使用，你也可以。这么做主要有两个好处。

意图清晰

只要在源码中见到元组，你就知道它的长度永不可变。

性能优越

长度相同的元组和列表，元组占用的内存更少，而且 Python 可对元组做些优化。

然而，你要知道，元组的不可变性仅针对元组中的引用而言。元组中的引用不可删除、不可替换。倘若引用的是可变对象，改动对象之后，元组的值也会随之变化。下面的代码段通过创建两个元组（a 和 b）来演示这一点。一开始，a 和 b 是相等的，此时 b 在内存中的初始布局如图 2-4 所示。

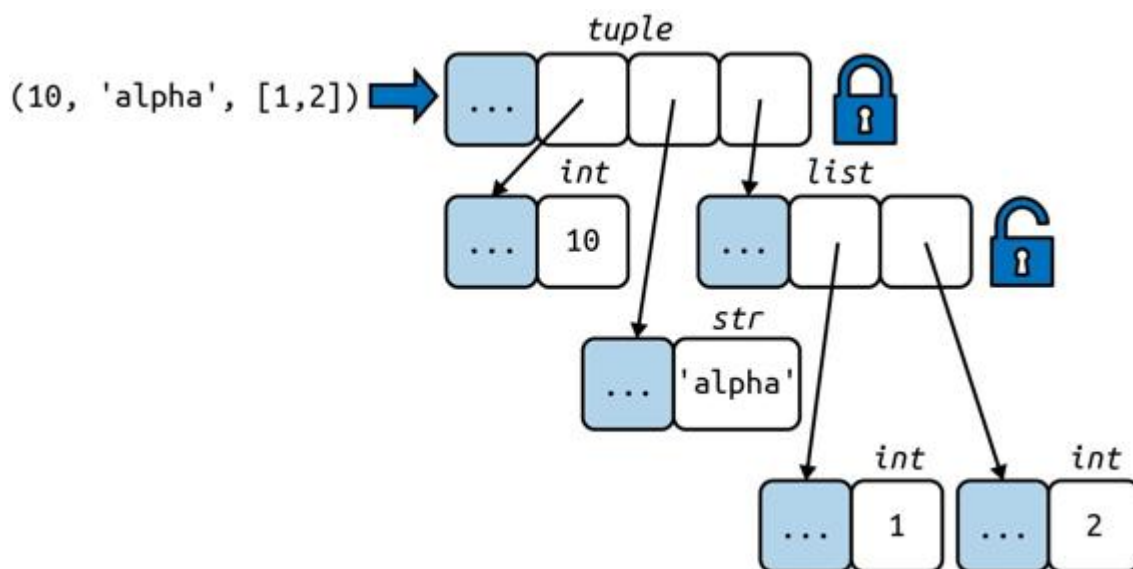


图 2-4：元组的内容自身是不可变的，但是这仅仅表明元组中存放的引用始终指向同一批对象。倘若引用的是可变对象，例如一个列表，那么元组的值就可能发生变化

我们修改 b 中的最后一项，现在 b 和 a 不相等了。

```
>>> a = (10, 'alpha', [1, 2])
>>> b = (10, 'alpha', [1, 2])
>>> a == b
True
>>> b[-1].append(99)
>>> a == b
False
```

```
>>> b  
(10, 'alpha', [1, 2, 99])
```

存放可变项的元组可能导致 bug。读到 3.4.1 节，你会发现，只有值永不可变的对象才是可哈希的。不可哈希的元组不能作为字典的键，也不能作为集合的元素。

如果你想显式判断一个元组（或其他对象）的值是否固定，则可以使用内置函数 `hash` 定义如下所示的 `fixed` 函数。

```
>>> def fixed(o):  
...     try:  
...         hash(o)  
...     except TypeError:  
...         return False  
...     return True  
...  
>>> tf = (10, 'alpha', (1, 2))  
>>> tm = (10, 'alpha', [1, 2])  
>>> fixed(tf)  
True  
>>> fixed(tm)  
False
```

6.3.2 节会进一步探讨这个问题。

尽管如此，元组依然广泛用作不可变列表。Stack Overflow 网站中有一个问题，题为“Are tuples more efficient than lists in Python?”。Python 核心开发人员 Raymond Hettinger 在回答这个问题时指出了元组在性能上的一些优势，总结如下。

- Python 编译器求解元组字面量时，经过一次操作即可生成元组常量的字节码。求解列表字面量时，生成的字节码将每个元素当作独立的常量推入数据栈，然后构建列表。
- 给定一个元组 `t`，`tuple(t)` 直接返回 `t` 的引用，不涉及复制。相比之下，给定一个列表 `l`，`list(l)` 创建 `l` 的副本。
- `tuple` 实例长度固定，分配的内存空间正好够用。而 `list` 实例的内存空间要富余一些，时刻准备追加元素。

- 对元组中项的引用存储在元组结构体内的一个数组中，而列表把引用数组的指针存储在别处。二者不存储在同一个地方的原因是列表可以变长，一旦超出当前分配的空间，Python 就需要重新分配引用数组来腾出空间，而这会导致 CPU 缓存效率较低。

2.4.3 列表和元组方法的比较

把元组当作列表的不可变变体使用时，有必要了解二者 API 之间的异同。从表 2-1 可以看出，元组支持所有不涉及增删项的列表方法，而且元组没有 `__reversed__` 方法。其实，这正是一种优化措施。没有 `__reversed__` 方法，`reversed(my_tuple)` 也能正常工作。

表 2-1：列表和元组的方法及属性（简单起见，省略了 object 实现的方法）

	列表	元组	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> ：拼接
<code>s.__iadd__(s2)</code>	•		<code>s += s2</code> ：就地拼接
<code>s.append(e)</code>	•		在最后一个元素后面追加一个元素
<code>s.clear()</code>	•		删除所有项
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		浅拷贝列表

	列表	元组	
<code>s.count(e)</code>	•	•	计算元素出现的次数
<code>s.__delitem__(p)</code>	•		删除位置 <code>p</code> 上的项
<code>s.extend(it)</code>	•		追加可迭代对象 <code>it</code> 中的项
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> : 获取指定位置上的项
<code>s.__getnewargs__()</code>		•	支持使用 <code>pickle</code> 优化序列化
<code>s.index(e)</code>	•	•	找出 <code>e</code> 首次出现的位置
<code>s.insert(p, e)</code>	•		在位置 <code>p</code> 上的项之前插入元素 <code>e</code>
<code>s.__iter__()</code>	•	•	获取迭代器
<code>s.__len__()</code>	•	•	<code>len(s)</code> : 项数
<code>s.__mul__(n)</code>	•	•	<code>s * n</code> : 重复拼接
<code>s.__imul__(n)</code>	•		<code>s *= n</code> : 就地重复拼接
<code>s.__rmul__(n)</code>	•	•	<code>n * s</code> : 反向重复拼接 ^a
<code>s.pop([p])</code>	•		移除并返回最后一项或可选的位置 <code>p</code> 上的项

	列表	元组	
<code>s.remove(e)</code>	•		把 <code>e</code> 的值从首次出现的位置上移除
<code>s.reverse()</code>	•		就地反转项的顺序
<code>s.__reversed__()</code>	•		获取从后向前遍历项的迭代器
<code>s.__setitem__(p, e)</code>	•		<code>s[p] = e</code> : 把 <code>e</code> 放在位置 <code>p</code> 上, 覆盖现有的项 ^b
<code>s.sort([key], [reverse])</code>	•		就地对项排序, <code>key</code> 和 <code>reverse</code> 是可选的关键字参数

a 反向运算符详见第 16 章。

b 也可覆盖一个子序列, 详见 2.7.4 节。

下面换个话题, 讲解 Python 编程中一种重要的惯用法, 即元组、列表和可迭代对象拆包。

2.5 序列和可迭代对象拆包

拆包的特点是不需要我们自己动手通过索引从序列中提取元素, 这样就减少了出错的可能。拆包的目标可以是任何可迭代对象, 包括不支持索引表示法 (`[]`) 的迭代器。拆包对可迭代对象的唯一要求是, 一次只能产出一项, 提供给接收端变量。不过也有例外, 可以使用星号 (`*`) 捕获余下的项, 详见 2.5.1 节。

最明显的拆包形式是**并行赋值**（parallel assignment），即把可迭代对象中的项赋值给变量元组，如以下示例所示。

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates # 拆包
>>> latitude
33.9425
>>> longitude
-118.408056
```

利用拆包还可以轻松对调两个变量的值，省掉中间的临时变量。

```
>>> b, a = a, b
```

调用函数时在参数前面加上一个 `*`，利用的也是拆包。

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

上述代码还展示了拆包的另一个用途：为函数返回多个值提供一种便于调用方使用的方式。再举一个例子：`os.path.split()` 函数根据传入的文件系统路径构建元组 `(path, last_part)`。

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/id_rsa.pub')
>>> filename
'id_rsa.pub'
```

如果只需要拆包得到的部分项，那么还可以使用 2.5.1 节介绍的 `*` 句法。

2.5.1 使用 `*` 获取余下的项

定义函数时可以使用 `*args` 捕获余下的任意数量的参数，这是 Python 的一个经典特性。

Python 3 把这一思想延伸到了并行赋值上。

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

并行赋值时，`*` 前缀只能应用到一个变量上，不过可以是任何位置上的变量。

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

2.5.2 在函数调用和序列字面量中使用 `*` 拆包

“PEP 448—Additional Unpacking Generalizations”为可迭代对象拆包引入了更灵活的句法，“What's New In Python 3.5”对此做了很好的概括。

在函数调用中可以多次使用 `*`。

```
>>> def fun(a, b, c, d, *rest):
...     return a, b, c, d, rest
...
>>> fun(*[1, 2], 3, *range(4, 7))
(1, 2, 3, 4, (5, 6))
```

定义列表、元组或集合字面量时，也可以使用 `*`，“What's New In Python 3.5”给出了一些示例。

```
>>> *range(4), 4
(0, 1, 2, 3, 4)
>>> [*range(4), 4]
[0, 1, 2, 3, 4]
>>> {*range(4), 4, *(5, 6, 7)}
{0, 1, 2, 3, 4, 5, 6, 7}
```

PEP 448 为 `**` 引入了类似的新句法，详见 3.2.2 节。

最后，元组拆包的一个强大功能是可以处理嵌套结构。

2.5.3 嵌套拆包

拆包的对象可以嵌套，例如 `(a, b, (c, d))`。如果值的嵌套结构是相同的，则 Python 能正确处理。示例 2-8 演示了嵌套拆包的具体用法。

示例 2-8 拆包嵌套元组，获取经度

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)), ❶
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

def main():
    print(f'{"":15} | {"latitude":>9} | {"longitude":>9}')
    for name, _, _, (lat, lon) in metro_areas: ❷
        if lon <= 0: ❸
            print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')

if __name__ == '__main__':
    main()
```

❶ 每个元组是一个四字段记录，最后一个字段是坐标对。

❷ 把最后一个字段赋值给一个嵌套元组，拆包坐标对。

❸ `lon <= 0`：测试条件只选取西半球的城市。

示例 2-8 的输出如下所示。

	latitude	longitude
Mexico City	19.4333	-99.1333
New York-Newark	40.8086	-74.0204
São Paulo	-23.5478	-46.6358

拆包赋值的对象也可以是一个列表，不过用途不大。据我所知，只有一种情况用得到：一个数据库查询只返回一个记录（比如，SQL 语句中有 `LIMIT 1` 子句），利用拆包确保只返回一个结果。

```
>>> [record] = query_returning_single_row()
```

如果记录中只有一个字段，则可以像下面这样直接获取。

```
>>> [[field]] = query_returning_single_row_with_single_field()
```

这两种情况也都可以使用元组，但是句法有点怪异，元组中唯一的项后面要加一个逗号。因此，第一个拆包的对象要写成 `(record,)`，第二个拆包的对象要写成 `((field,),)`。如果忘记末尾的逗号，将埋下不易察觉的 bug。³

³感谢技术审校 Leonardo Rochael 提供这个示例。

接下来我们学习更强大的序列拆包方式：模式匹配。

2.6 序列模式匹配

Python 3.10 最引人注目的新功能是“PEP 634—Structural Pattern Matching: Specification”提出的 `match/case` 语句实现的模式匹配。



在“[What's New In Python 3.10](#)”中的“[Structural Pattern Matching](#)”一节，Python 核心开发人员 Carol Willing 对模式匹配的介绍精彩纷呈，建议你读一下。根据不同的模式类型，模式匹配出现在本书中不同的地方：3.3 节和 5.8 节，还有 18.3 节的一个延伸示例。

下面是使用 `match/case` 处理序列的第一个示例。假设你在设计一个机器人，它接受以文字和数值序列形式发送的命令，例如 `BEEPER 440 3`。经过拆分和解析之后，得到消息 `['BEEPER', 440, 3]`。可以使用类似示例 2-9 中的方法处理这样的消息。

示例 2-9 假想的 Robot 类中的方法

```
def handle_command(self, message):
    match message: ❶
        case ['BEEPER', frequency, times]: ❷
            self.beep(times, frequency)
        case ['NECK', angle]: ❸
            self.rotate_neck(angle)
        case ['LED', ident, intensity]: ❹
            self.leds[ident].set_brightness(ident, intensity)
        case ['LED', ident, red, green, blue]: ❺
            self.leds[ident].set_color(ident, red, green,
blue)
        case _: ❻
            raise InvalidCommand(message)
```

❶ `match` 关键字后面的表达式是**匹配对象**（subject），即各个 `case` 子句中的模式尝试匹配的数据。

❷ 这个模式匹配一个含有 3 项的序列。第一项必须是字符串 `'BEEPER'`。第二项和第三项任意，依次绑定到变量 `frequency` 和 `times` 上。

❸ 这个模式匹配任何含有两项，而且第一项为 `'NECK'` 的序列。

④ 这个模式匹配第一项为 'LED'，共有 3 项的序列。如果项数不匹配，则 Python 继续执行下一个 case 子句。

⑤ 这个模式也匹配第一项为 'LED' 的序列，不过一共有 5 项。

⑥ 这是默认的 case 子句，前面所有模式都不匹配时执行。_ 是特殊的变量，稍后讲解。

表面上看，match/case 与 C 语言中的 switch/case 语句很像，但这只是表象。⁴ 与 switch 相比，match 的一大改进是支持析构，这是一种高级拆包形式。析构对 Python 世界来说是一个新词，不过在支持模式匹配的语言（例如 Scala 和 Elixir）文档中经常出现。

⁴在我看来，处理序列的 switch/case 语句完全可以替换成 if/elif/elif/.../else 代码块。这样做可以避免“落空”（fallthrough）和“else 垂悬”问题。这两个问题的起因是一些语言设计者一味照搬 C 语言，而数十年后人们才意识到这是导致无数 bug 的原因。

示例 2-10 在本书中首次使用析构，使用 match/case 重写示例 2-8 的一部分代码。

示例 2-10 析构嵌套元组（要求 Python 3.10 及以上版本）

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

def main():
    print(f'{"":15} | {"latitude":>9} | {"longitude":>9}')
    for record in metro_areas:
        match record: ❶
            case [name, _, _, (lat, lon)] if lon <= 0: ❷
                print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
```

❶ 这个 match 的匹配对象是 record，即 metro_areas 中的各个元组。

② 一个 `case` 子句由两部分组成：一部分是模式，另一部分是使用 `if` 关键字指定的卫语句（guard clause，可选）。

一般来说，匹配对象同时满足以下条件方能匹配序列模式。

01. 匹配对象是序列。
02. 匹配对象和模式的项数相等。
03. 对应的项相互匹配，包括嵌套的项。

例如，示例 2-10 中的模式 `[name, _, _, (lat, lon)]` 匹配一个含有 4 项的序列，而且最后一项必须是一个含有两项的序列。

序列模式可以写成元组或列表，或者任意形式的嵌套元组和列表，使用哪种句法都没有区别，因为在序列模式中，方括号和圆括号的意思是一样的。示例 2-10 中的模式写成列表形式，其中嵌套的序列则写成元组形式，这样做只是为了避免重复使用方括号或圆括号。

序列模式可以匹配 `collections.abc.Sequence` 的多数实际子类或虚拟子类的实例，但 `str`、`bytes` 和 `bytearray` 除外。



在 `match/case` 上下文中，`str`、`bytes` 和 `bytearray` 实例不作为序列处理。`match` 把这些类型视为“原子”值，就像整数 987 整体被视为一个值，而不是数字序列。倘若把这三种类型视为序列，就可能会由于意外匹配而导致 bug。如果想把这些类型的对象视为序列，则要在 `match` 子句中转换，例如以下示例中的 `tuple(phone)`。

```
match tuple(phone):
    case ['1', *rest]: # 北美洲和加勒比地区
        ...
    case ['2', *rest]: # 非洲
        ...
    case ['3' | '4', *rest]: # 欧洲
        ...
```

标准库中的以下类型与序列模式兼容。

<code>list</code>	<code>memoryview</code>	<code>array.array</code>
<code>tuple</code>	<code>range</code>	<code>collections.deque</code>

与拆包不同，模式不析构序列以外的可迭代对象（例如迭代器）。

`_` 符号在模式中有特殊意义：匹配相应位置上的任何一项，但不绑定匹配项的值。另外，`_` 是唯一可在模式中多次出现的变量。

模式中的任何一部分均可使用 `as` 关键字绑定到变量上。

<code>case [name, _, _, (lat, lon) as coord]:</code>

上述模式可以匹配 `['Shanghai', 'CN', 24.9, (31.1, 121.3)]`，并设定以下变量。

变量	设定的值
<code>name</code>	<code>'Shanghai'</code>
<code>lat</code>	<code>31.1</code>
<code>lon</code>	<code>121.3</code>
<code>coord</code>	<code>(31.1, 121.3)</code>

添加类型信息可以让模式更具体。例如，下面的模式与前面的示例匹配相同的嵌套序列结构，不过第一项必须是 `str` 实例，而且二元组中的两项必须是 `float` 实例。

<code>case [str(name), _, _, (float(lat), float(lon))]:</code>



表达式 `str(name)` 和 `float(lat)` 看起来像是构造函数调用——前者把 `name` 转换成 `str`，后者把 `lat` 转换成 `float`。其实不然，在模式上下文中，这种句法的作用是在运行时检查类型。前面的模式将匹配一个 4 项序列，其中第一项必须是一个字符串，第四项必须是一对浮点数。而且，第一项中的字符串将绑定到 `name` 变量上，第四项中的一对浮点数将分别绑定到 `lat` 和 `lon` 两个变量上。因此，尽管 `str(name)` 借用了构造函数调用句法，但是在模式上下文中，语义是完全不同的。5.8 节会说明如何在模式中使用任意的类。

此外，如果想要匹配任何以字符串开头、以嵌套两个浮点数的序列结尾的序列，则可以使用如下模式。

```
case [str(name), *_ , (float(lat), float(lon))]:
```

`*_` 匹配任意数量的项，而且不绑定变量。如果把 `*_` 换成 `*extra`，匹配的零项或多项将作为列表绑定到 `extra` 变量上。

以 `if` 开头的卫语句是可选的，仅当匹配模式时才运行。卫语句可以像示例 2-10 那样引用模式中绑定的变量。

```
match record:
    case [name, _, _, (lat, lon)] if lon <= 0:
        print(f'{name:15} | {lat:9.4f} | {lon:9.4f}')
```

嵌套块中的 `print` 语句仅当匹配模式且卫语句**为真**时才运行。



模式析构表现力十足，即使只有一个 `case` 子句，`match` 语句也可以让代码变得更简单。Guido van Rossum 列举了一些使用 `match/case` 的示例，其中一例非常有趣，题为“A very deep iterable and type match with extraction”。

示例 2-10 不是示例 2-8 的改进版本，只是做同一件事的另一种方式。下一个示例说明如何利用模式匹配写出清晰、简洁、高效的代码。

使用模式匹配序列实现一个解释器

斯坦福大学的 Peter Norvig 编写的 `lis.py` 是一个解释器，用于解释 Lisp 编程语言 Scheme 方言的一个子集。这个解释器仅用了 132 行 Python 代码，优雅且易于理解。我把 Norvig 以 MIT 许可证发布的源码更新到了 Python 3.10，展示模式匹配的用法。这一节比较一下 Norvig 使用 `if/elif` 和拆包实现的代码和我使用 `match/case` 重写的代码。

`lis.py` 中的两个主要函数是 `parse` 和 `evaluate`。⁵ `parse` 函数接受的参数为包含在圆括号内的 Scheme 风格表达式，返回值是 Python 列表。下面举两个例子。

⁵在 Norvig 的代码中，后一个函数名为 `eval`。重写时，我把名称换掉了，免得与 Python 内置函数 `eval` 混淆。

```
>>> parse('(gcd 18 45)')
['gcd', 18, 45]
>>> parse('''
... (define double
...     (lambda (n)
...         (* n 2)))
... ''')
['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

`evaluate` 函数的作用就是执行通过参数传入的这种列表。在第一个例子中，`evaluate` 函数调用 `gcd` 函数，计算传入的参数 18 和 45 的最大公约数，即 9。第二个例子定义一个名为 `double` 的函数，参数为 `n`，主体为表达式 `(* n 2)`。在 Scheme 中，调用函数返回的结果是主体中最后一个表达式的值。

本节关注的重点是析构序列，求值操作不做深入探讨。lis.py 的进一步说明见 18.3 节。

示例 2-11 是 Norvig 实现的 `evaluate` 函数，有所节略，只给出了序列模式部分。

示例 2-11 不使用 `match/case` 匹配模式

```
def evaluate(exp: Expression, env: Environment) -> Any:
    "根据所处的环境求解表达式。"
    if isinstance(exp, Symbol):          # 变量引用
        return env[exp]
    # ... 省略几行
    elif exp[0] == 'quote':              # (quote exp)
        (_, x) = exp
        return x
    elif exp[0] == 'if':                  # (if test consequ alt)
        (_, test, consequence, alternative) = exp
        if evaluate(test, env):
            return evaluate(consequence, env)
        else:
            return evaluate(alternative, env)
    elif exp[0] == 'lambda':              # (lambda (parm...) body...)
        (_, parms, *body) = exp
        return Procedure(parms, body, env)
    elif exp[0] == 'define':
        (_, name, value_exp) = exp
        env[name] = evaluate(value_exp, env)
    # ... 省略余下的代码
```

注意，每个 `elif` 子句检查列表的第一项，然后拆包列表，忽略第一项。可以看到，Norvig 大量使用拆包，这表明他很喜欢模式匹配。可惜，他的代码是用 Python 2 写的（也能使用 Python 3 运行），那时还没有模式匹配。

使用 Python 3.10 及以上版本中的 `match/case` 重构后的 `evaluate` 函数如示例 2-12 所示。

示例 2-12 使用 `match/case` 匹配模式（要求 Python 3.10 及以上版本）

```
def evaluate(exp: Expression, env: Environment) -> Any:
    "Evaluate an expression in an environment."
    match exp:
        # ... 省略几行
        case ['quote', x]: ❶
            return x
        case ['if', test, consequence, alternative]: ❷
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
        case ['lambda', [*parms], *body] if body: ❸
            return Procedure(parms, body, env)
        case ['define', Symbol() as name, value_exp]: ❹
            env[name] = evaluate(value_exp, env)
        # ... 又省略几行
        case _: ❺
            raise SyntaxError(lispstr(exp))
```

❶ 匹配以 'quote' 开头的 2 项序列。

❷ 匹配以 'if' 开头的 4 项序列。

❸ 匹配以 'lambda' 开头的 3 项或更多项序列。卫语句确保 body 不为空。

❹ 匹配以 'define' 开头、后跟一个 Symbol 实例的 3 项序列。

❺ 提供一个兜底 case 子句是个好习惯。在这个示例中，如果 exp 没有匹配任何模式，说明表达式格式有误，那就抛出 SyntaxError。

如果没有兜底 case 子句，那么当匹配对象不符合任何情况时，match 语句就什么也不做，悄无声息，有问题也不提醒我们。

为了让 lis.py 的代码更易于理解，Norvig 故意没有写错误检查。而使用模式匹配，就可以添加很多错误检查，同时不影响代码的可读性。例如，原版代码没有确保 'define' 模式中的 name 必须是 Symbol 实例。如果需要这项检查，那就要添加一个 if 块、一个

`isinstance` 函数调用，以及一些其他代码。示例 2-12 比示例 2-11 更简短、更安全。

01. 替换 `lambda` 表达式的模式

在 Scheme 中，`lambda` 表达式的句法如下所示。句法中的后缀 `...` 是一种约定，表示元素可以出现零次或多次。

```
(lambda (parms...) body1 body2...)
```

考虑简单一些，`'lambda'` 模式可能会写成下面这样。

```
case ['lambda', parms, *body] if body:
```

然而，这个模式在 `parms` 位置上能匹配任何值，包括下面这个无效的匹配对象中的第一个 `'x'`。

```
['lambda', 'x', ['*', 'x', 2]]
```

在 Scheme 中，`lambda` 关键字后面的嵌套列表是函数的形式参数名称，即使只有一个元素，也要写成列表。如果函数不接受参数，那就是空列表，就像 Python 中的 `random.random()` 函数一样。

在示例 2-12 中，为了确保 `'lambda'` 模式更安全，我使用的是一个嵌套序列模式。

```
case ['lambda', [*parms], *body] if body:
    return Procedure(parms, body, env)
```

在序列模式中，一个序列中只能有一个 `*`。这里用到两个序列，外层一个，内层一个。

在 `parms` 两侧加上 `[*]` 之后，模式看起来更符合 Scheme 句法，而且额外多了一项结构检查。

02. 定义函数的快捷句法

Scheme 中还有一种 `define` 句法，不使用嵌套的 `lambda` 定义具名函数，如下所示。

```
(define (name parm...) body1 body2...)
```

`define` 关键字后跟一个列表，`name` 是函数的名称，`parm...` 是零个或多个参数名称。这个列表之后是函数主体，主体有一个或多个表达式。

在 `match` 语句中添加以下两行可以实现这种句法。

```
case ['define', [Symbol() as name, *parms], *body] if
body:
    env[name] = Procedure(parms, body, env)
```

我选择把这个 `case` 子句放在示例 2-12 中匹配其他 `define` 的 `case` 子句后面。在这个示例中，`define` 的不同情况以何种顺序放置其实无关紧要，因为没有哪个匹配对象能同时满足所有模式：现有的 `define` 的 `case` 子句，第二个元素必须是 `Symbol` 实例；新增的快捷句法用来定义函数，第二个元素必须是一个以 `Symbol` 实例开头的序列。

可以想象一下，如果按照示例 2-11 那样不使用模式匹配，为了支持新 `define` 句法需要多少工作量。`match` 语句做的事情比类 C 语言中的 `switch` 语句多出不少。

模式匹配是一种声明式编程风格，即描述你想匹配“什么”，而不是“如何”匹配，这样写出的代码结构与数据结构是一致的，如表 2-2 所示。

表 2-2：一些 Scheme 句法形式和处理句法的 `case` 模式

Scheme 句法	序列模式
<code>(quote exp)</code>	<code>['quote', exp]</code>
<code>(if test conseq alt)</code>	<code>['if', test, conseq, alt]</code>
<code>(lambda (parms...) body1 body2...)</code>	<code>['lambda', [*parms], *body] if body</code>
<code>(define name exp)</code>	<code>['define', Symbol() as name, exp]</code>
<code>(define (name parms...) body1 body2...)</code>	<code>['define', [Symbol() as name, *parms], *body] if body</code>

相信通过本节对 Norvig 原先编写的 `evaluate` 函数进行重构之后，你能发现使用 `match/ case` 写出的代码更具可读性，也更安全。



18.3 节还会进一步分析 `lis.py`，届时将全面研究 `evaluate` 函数中的 `match/case` 语句。如果想要深入了解 `lis.py`，请阅读 Norvig 写的文章：“(How to Write a (Lisp) Interpreter (in Python))”。

对拆包、析构和模式匹配的体验之旅到此结束。其他模式类型在后续章节再讲。

每个 Python 程序员都知道，序列可以使用 `s[a:b]` 句法切片。2.7 节会探索一些鲜为人知的切片功能。

2.7 切片

在 Python 中，列表、元组、字符串等所有序列类型都支持切片操作。切片比多数人认为的要强大很多。

本节讨论切片的高级用法。本书这一部分关注的是现成可用的类，在用户定义的类中实现切片的方法在第 12 章说明，自定义类则延后到第三部分。

2.7.1 为什么切片和区间排除最后一项

切片和区间排除最后一项是一种 Python 风格约定，这与 Python、C 和很多其他语言中从零开始的索引相匹配。排除最后一项可以带来以下好处。

- 在仅指定停止位置时，容易判断切片或区间的长度。例如，`range(3)` 和 `my_list[:3]` 都只产生 3 项。
- 同时指定起始和停止位置时，容易计算切片或区间的长度，做个减法即可：`stop - start`。
- 方便在索引 `x` 处把一个序列拆分成两部分而不产生重叠，直接使用 `my_list[:x]` 和 `my_list[x:]` 即可。例如：

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2] # 在索引位2处拆分
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3] # 在索引位3处拆分
[10, 20, 30]
>>> l[3:]
[40, 50, 60]
```

荷兰计算机科学家 Edsger W. Dijkstra 写过一篇文章，对这一风格的解释应该是最好的（见 2.12 节）。

接下来深入了解 Python 是如何解释切片表示法的。

2.7.2 切片对象

一个众所周知的秘密是，我们还可以使用 `s[a:b:c]` 句法指定步距 `c`，让切片操作跳过部分项。步距也可以是负数，反向返回项。下面举 3 个具体的例子。

```
>>> s = 'bicycle'
>>> s[::-3]
```

```
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

第 1 章也有这样的例子，当时我们使用 `deck[12::13]` 从一擦没洗过的牌中抽取所有 A。

```
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

`a:b:c` 表示法只在 `[]` 内部有效，表示索引或下标运算符，得到的结果是一个切片对象：`slice(a, b, c)`。读到 12.5.1 节你会发现，为了求解表达式 `seq[start:stop:step]`，Python 调用 `seq.__getitem__(slice(start, stop, step))`。就算你不想自己实现序列类型，了解切片对象也有一定的好处。例如，你可以给切片命名，就像在电子表格软件里给单元格区域命名一样。

假设你需要处理类似示例 2-13 中的发票那种纯文本数据。与其让代码充斥大量晦涩的切片，不如为切片命名。不信的话，可以看看这个示例末尾的 `for` 循环，你会发现可读性非常高。

示例 2-13 从纯文本形式的发票中提取商品信息

```
>>> invoice = """
...
0.....6.....40.....52...55.....
... 1909  Pimoroni PiBrella                $17.50    3
$52.50
... 1489  6mm Tactile Switch x20            $4.95     2
$9.90
... 1510  Panavise Jr. - PV-201             $28.00    1
$28.00
... 1601  PiTFT Mini Kit 320x240           $34.95    1
$34.95
... """
>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
```

```
>>> QUANTITY = slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
...     print(item[UNIT_PRICE], item[DESCRIPTION])
...
$17.50    Pimoroni PiBrella
$4.95     6mm Tactile Switch x20
$28.00    Panavise Jr. - PV-201
$34.95    PiTFT Mini Kit 320x240
```

12.5 节讲解自定义容器类型时还会再次讨论切片对象。从用户的角度出发，切片还有两个额外的功能：多维切片和省略号 (...) 表示法。请继续往下读。

2.7.3 多维切片和省略号

[] 运算符还可以接受多个索引或切片，以逗号分隔。负责处理 [] 运算符的特殊方法 `__getitem__` 和 `__setitem__` 把接收到的 `a[i, j]` 中的索引当作元组。也就是说，为了求解 `a[i, j]`，Python 调用 `a.__getitem__((i, j))`。

例如，在外部包 NumPy 中，`numpy.ndarray` 表示的二维数组可以使用 `a[i, j]` 句法获取数组中的元素，还可以使用表达式 `a[m:n, k:l]` 获取二维切片。本章稍后的示例 2-22 会展示这一用法。

除了 `memoryview` 之外，Python 内置的序列类型都是一维的，因此只支持一个索引或切片，不支持索引或切片元组。⁶

⁶读到 2.10.2 节你会发现，特殊结构的内存视图可以有多个维度。

省略号写作 3 个句点 (...)，而不是 ... (Unicode U+2026)，Python 解析器把它识别为一个记号。省略号是 `Ellipsis` 对象的别名，而 `Ellipsis` 对象是 `ellipsis` 类的单例。⁷ 因此，你可以把省略号作为参数传给函数，也可以写在切片规范中，例如 `f(a, ..., z)` 或 `a[i:...]`。NumPy 在处理多维数组切片时把 ... 解释为一种

快捷句法。例如，对四维数组 `x`，`x[i, ...]` 是 `x[i, :, :, :, ]` 的快捷句法。如果想进一步了解，请阅读 NumPy 官网的“NumPy quickstart”。

⁷是的，我没有写反，`ellipsis` 这个类名的确是全小写形式，而内置的 `Ellipsis` 对象首字母是大写的。这与 `bool` 是一样的，类名全小写，而实例 `True` 和 `False` 首字母大写。

写作本书时，我还没有发现 Python 标准库采用 `Ellipsis` 或多维索引和多维切片。如果你发现了，请告诉我。存在这种句法是为了给用户定义的类型和扩展（例如 NumPy）提供支持。

切片不仅可从序列中提取信息，还可以就地更改可变序列，即不重新构建序列。

2.7.4 为切片赋值

在赋值语句的左侧使用切片表示法，或者作为 `del` 语句的目标，可以就地移植、切除或以其他方式修改可变序列。下面举几个例子演示这种表示法的强大功能。

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100 ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

❶ 如果赋值目标是一个切片，则右边必须是一个可迭代对象，即使只有一项。

每个程序员都知道序列经常需要做拼接操作。很多 Python 入门教程会介绍如何使用 `+` 和 `*` 执行拼接操作，不过这两个运算符还有一些细节需要注意，详见 2.8 节。

2.8 使用 `+` 和 `*` 处理序列

Python 程序员预期序列支持 `+` 和 `*`。通常，`+` 的两个运算对象必须是同一种序列，而且都不可修改，拼接的结果是一个同类型的新序列。

如果想多次拼接同一个序列，可以乘以一个整数。同样，结果是一个新创建的序列。

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

`+` 和 `*` 始终创建一个新对象，绝不更改操作数。



注意 `a * n` 这种表达式，如果序列 `a` 中包含可变项，则结果可能出乎意料。例如，使用 `my_list = [[]] * 3` 初始化一个嵌套列表，得到的结果是一个列表没错，但是嵌套的 3 个引用指向同一个列表，而你或许并不希望如此。

2.8.1 节说明使用 `*` 初始化嵌套列表的陷阱。

2.8.1 构建嵌套列表

有时，我们需要初始化内部嵌套一定数量列表的列表，例如把学生分配到团队列表中，或者表示棋盘游戏中的方块。为了解决这一问题，

最佳方法是使用列表推导式，如示例 2-14 所示。

示例 2-14 一个列表中嵌套 3 个长度为 3 的列表，可以表示井字游戏棋盘

```
>>> board = [['_' * 3 for i in range(3)]] ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X' ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

❶ 创建一个列表，嵌套 3 个 3 项列表。检查列表结构。

❷ 在第 2 行第 3 列放一个叉号，检查结果。

示例 2-15 中的做法看似省事，却是错的。

示例 2-15 不能在一个列表中 3 次引用同一个列表

```
>>> weird_board = [['_' * 3] * 3] ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = 'O' ❷
>>> weird_board
[['_', '_', 'O'], ['_', '_', 'O'], ['_', '_', 'O']]
```

❶ 外层列表内部的 3 个引用指向同一个列表。这一步的结果跟前面一样，看似一切正常。

❷ 在第 2 行第 3 列放一个圈，这时你就会发现所有行都引用同一个对象。

示例 2-15 的问题在于，本质上，它的行为与以下代码类似。

```
row = ['_' * 3]
board = []
for i in range(3):
    board.append(row) ❶
```


❶ 同一个 row 向 board 中追加 3 次。

此外，示例 2-14 中的列表推导式等价于以下代码。

```
>>> board = []
>>> for i in range(3):
...     row = ['_'] * 3 ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

❶ 每次迭代构建一个新 row，追加到 board 中。

❷ 符合预期，只有第 3 行变了。



如果你不理解本节所讲的问题或解决方案，没关系。第 6 章就是为了阐明引用和可变对象的机制和陷阱而编写的。

目前，我们讨论的是如何使用普通的 + 和 * 运算符处理序列，除此之外还有 += 和 *= 运算符。根据目标序列的可变性而定，后两个运算符产生的效果完全不同，详见 2.8.2 节。

2.8.2 使用增量赋值运算符处理序列

根据第一个操作数而定，增量赋值运算符 += 和 *= 的行为差异较大。为了简化讨论，本节重点关注增量加法运算符（+=），不过相关概念也适用于 *= 和其他增量赋值运算符。

背后支持 += 运算符的是特殊方法 `__iadd__`（就地相加）。

但是，如果没有实现 `__iadd__`，那么 Python 转而调用 `__add__`。以下面这个简单的表达式为例。

```
>>> a += b
```

如果 `a` 实现了 `__iadd__`，那就调用它。如果 `a` 是可变序列（例如，`list`、`bytearray`、`array.array`），则就地修改 `a`（行为类似于 `a.extend(b)`）。倘若 `a` 没有实现 `__iadd__` 方法，表达式 `a += b` 的作用等同于 `a = a + b`，先求解表达式 `a + b`，再把得到的新对象绑定到 `a` 上。也就是说，`a` 绑定的对象身份可能变了，也可能没变，这取决于有没有实现 `__iadd__` 方法。

通常，对于可变序列，最好实现 `__iadd__` 方法，而且 `+=` 运算符就地修改。对于不可变序列，显然不能就地修改。

以上内容也适用于通过 `__imul__` 方法实现的 `*=` 运算符。`__iadd__` 和 `__imul__` 这两个特殊方法将在第 16 章讨论。下面使用 `*=` 运算符分别处理一个可变序列和一个不可变序列。

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800 ❶
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800 ❷
>>> t = (1, 2, 3)
>>> id(t)
4312681568 ❸
>>> t *= 2
>>> id(t)
4301348296 ❹
```

❶ 列表最初的 ID。

❷ 乘法运算之后，列表还是同一个对象，只是追加了几项。

❸ 元组最初的 ID。

❹ 乘法运算之后，创建了一个新元组。

对不可变序列重复拼接效率低下，因为解释器必须复制整个目标序列，创建一个新序列，包含要拼接的项，而不是简单追加新项。⁸

⁸字符串例外。由于现实中的代码基经常在循环中使用 `+=` 运算符构建字符串，因此 CPython 对这种情况做了优化。内存为 `str` 实例分配空间时会留些富余，因此拼接字符串时无须每次都复制整个字符串。

本节讲的是 `+=` 运算符的常规用法。2.8.3 节将展示一个有趣的极端情况，以元组为例说明“不可变”的真正含义。

2.8.3 一个 `+=` 运算符赋值谜题

不要使用控制台，请尝试答出示例 2-16 中两个表达式的求解结果。⁹

⁹感谢 Leonardo Rochael 和 Cesar Kawakami 在 2013 年 Python Brasil 技术大会上分享这个谜题。

示例 2-16 一个谜题

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

结果如何？请从以下 4 个选项中选出最佳答案。

- A. `t` 变成 `(1, 2, [30, 40, 50, 60])`。
- B. 抛出 `TypeError`，错误消息为 `'tuple' object does not support item assignment`。
- C. A 和 B 都不对。
- D. A 和 B 都对。

看到这个问题时，我非常确定答案是 B，然而正确答案是 D，“A 和 B 都对”。在 Python 3.9 的控制台中，以上两个表达式的输出如示例 2-17 所示。¹⁰

¹⁰一些读者指出，本例中的操作可以使用 `t[2].extend([50, 60])` 执行，不报错。我知道，但是我的本意是展示 `+=` 运算符的这种奇怪行为。

示例 2-17 结果让人意外，`t[2]` 项有变动，同时抛出异常

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

Python Tutor 是一个优秀的在线工具，以图解方式阐释 Python 的运作机制。图 2-5 包含两幅截图，分别展示示例 2-17 中元组 `t` 的初始和最终状态。

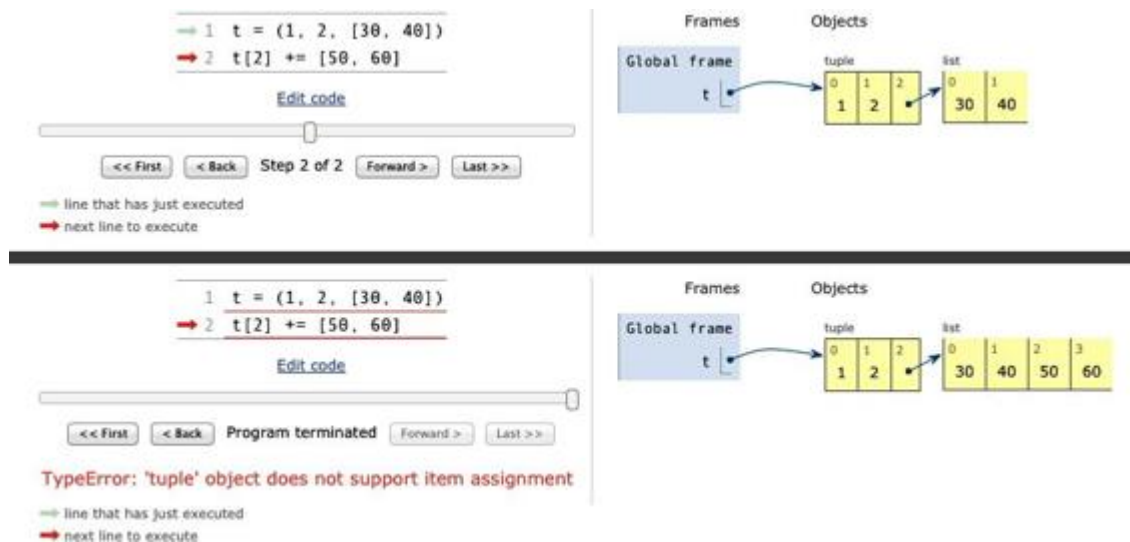


图 2-5：元组赋值谜题的初始和最终状态（图片由 Python Tutor 网站生成）

看一下 Python 为表达式 `s[a] += b` 生成的字节码（见示例 2-18），你一定会恍然大悟。

示例 2-18 `s[a] += b` 表达式的字节码

```

>>> dis.dis('s[a] += b')
 1          0 LOAD_NAME          0 (s)
          3 LOAD_NAME          1 (a)
          6 DUP_TOP_TWO
          7 BINARY_SUBSCR          ❶
          8 LOAD_NAME          2 (b)
         11 INPLACE_ADD          ❷
         12 ROT_THREE
         13 STORE_SUBSCR          ❸
         14 LOAD_CONST          0 (None)
         17 RETURN_VALUE

```

- ❶ 把 `s[a]` 的值放在栈顶 (TOS)。
- ❷ 执行 `TOS += b`。如果 `TOS` 引用一个可变对象 (在示例 2-17 中是一个列表)，则操作成功。
- ❸ 赋值 `s[a] = TOS`。如果 `s` 是不可变对象 (在示例 2-17 中是 `t` 元组)，则操作失败。

这种情况相当极端，在我使用 Python 的 20 年间，从未见过有人受到这个奇怪行为的影响。

我从这个谜题中吸取了 3 个教训。

- 不要在元组中存放可变的项。
- 增量赋值不是原子操作。这一点刚刚见识到，部分操作执行完毕后又抛出了异常。
- 检查 Python 字节码并不太难，从中可以看出 Python 在背后做了什么。

揭露使用 `+` 和 `*` 拼接序列的微妙细节之后，我们换个话题，讲解序列的另一个基本操作——排序。

2.9 `list.sort` 与内置函数 `sorted`

`list.sort` 方法就地排序列表，即不创建副本。返回值为 `None`，目的就是提醒我们，它更改了接收者，¹¹ 没有创建新列表。这是 Python API 的一个重要约定：就地更改对象的函数或方法应该返回 `None`，让调用方清楚地知道接收者已被更改，没有创建新对象。例如，`random.shuffle(s)` 函数也有类似的行为：就地混洗可变序列 `s`，返回 `None`。

¹¹接收者是方法调用的目标，即方法定义体中绑定到 `self` 上的对象。



返回 `None` 表示就地更改的约定有一个缺点：这种方法不能级联调用。相反，返回新对象的方法（例如，`str` 的所有方法）可以在流式接口（fluent interface）风格中级联调用。

与之相反，内置函数 `sorted` 返回创建的新列表。该函数接受任何可迭代对象作为参数，包括不可变序列和生成器（见第 17 章）。无论传入什么类型的可迭代对象，`sorted` 函数始终返回新创建的列表。

`list.sort` 和 `sorted` 均接受两个可选的关键字参数。

`reverse`

值为 `True` 时，降序返回项（即反向比较各项）。默认值为 `False`。

`key`

一个只接受一个参数的函数，应用到每一项上，作为排序依据。例如，排序字符串时，`key=str.lower` 执行不区分大小写排序，而 `key=len` 按字符长度排序各个字符串。默认值是恒等函数（即比较项本身）。



`min()` 和 `max()` 等内置函数，以及标准库中的其他函数（例如 `itertools.groupby()` 和 `heapq.nlargest()`）也

有可选的关键字参数 `key`。

下面举几个例子说明如何使用这个函数及其关键字参数。这些例子还表明 Python 的排序算法是稳定的（即能够保留比较时相等的两项的相对顺序）。¹²

¹²Python 主要使用的排序算法是 Timsort，以创建人 Tim Peters 命名。Timsort 的一些趣闻见本章末尾的“杂谈”。

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry'] ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple'] ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry'] ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple'] ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❻
>>> fruits.sort() ❼
>>> fruits
['apple', 'banana', 'grape', 'raspberry'] ❸
```

❶ 得到一个按字母表顺序排列的新字符串列表。¹³

¹³这个例子中的各个词按照字母表顺序排列，因为每个词中的字母全都是小写的 ASCII 字符。详见这个例子后面的警告栏。

❷ 查看原来的列表，可以看到没有改动。

❸ 也按字母表顺序排列，只不过是反向排序。

❹ 一个新字符串列表，这一次按长度排序。排序算法很稳定，尽管“grape”和“apple”这两个词的长度都是 5，但仍按原顺序排列。

- ⑤ 按字符串长度降序排列。这不是前一个结果的反转，因为排序算法很稳定，所以“grape”依然出现在“apple”前面。
- ⑥ 目前为止，`fruits` 列表的原始顺序没有改变。
- ⑦ 就地排序列表，返回 `None`（被控制台省略了）。
- ⑧ 现在 `fruits` 有顺序了。



默认情况下，Python 按字符代码的字典顺序排序字符串。这意味着，ASCII 大写字母排在小写字母前面，而非 ASCII 字符不太可能以合理的方式排序。4.8 节会讲解排序文本的正确方式，即一种符合人类预期的方式。

排序后的序列，搜索效率非常高。Python 标准库中的 `bisect` 模块提供了一种二进制搜索算法，随取随用。该模块中的 `bisect.insort` 函数还可以确保排序后的序列始终保持有序。

本章目前所讲的内容大都适用于一般意义上的序列，不仅限于列表和元组。Python 程序员有时过度使用 `list` 类型，因为列表太方便了，我也不例外。然而，处理大型数值列表，应考虑使用数组。本章余下的内容就讲一讲列表和元组之外的选择。

2.10 当列表不适用时

`list` 类型简单灵活，不过，针对具体的需求，或许还有更好的选择。例如，使用数组处理上百万个浮点值可以节省大量内存。另外，如果经常需要在列表的两端添加和删除项，使用 `deque`（double-ended queue，双端队列）更合适，这是一种更高效的 FIFO¹⁴ 数据结构。

¹⁴FIFO 是“First in, first out”的简称，即先进先出。这是队列的默认行为。



如果你在代码中经常检查容器中是否存在某一项（例如 `item in my_collection`），应考虑使用 `set` 类型存储 `my_collection`，尤其是项数较多的情况。Python 对 `set` 成员检查做了优化，速度更快。`set` 也是可迭代对象，但不是序列，因为 `set` 中的项是无序的，详见第 3 章。

本章余下的内容将讨论很多情况下可以取代列表的可变序列类型，接下来先从数组开始。

2.10.1 数组

如果一个列表只包含数值，那么使用 `array.array` 会更高效。数组支持所有可变序列操作（包括 `.pop`、`.insert` 和 `.extend`），此外还有快速加载项和保存项的方法，例如 `.frombytes` 和 `.tofile`。

Python 数组像 C 语言数组一样精简。如图 2-1 所示，一个由 `float` 值构成的数组，存放的并不是完整的 `float` 实例，而是表示相应机器值的压缩字节，与 C 语言中由 `double` 值构成的数组如出一辙。创建 `array` 对象时要提供类型代码，它是一个字母，用来确定底层使用什么 C 类型存储数组中各项。例如，类型代码 `b` 对应 C 语言中的 `signed char` 类型，即取值范围是 -128~127 的整数。如果使用 `array('b')` 创建一个数组，那么这个数组中的每一项都使用一个字节存储，而且均被解释为整数。对于大型数值序列，这样做可以节省大量内存。另外，Python 不允许向数组中添加与指定类型不同的值。

示例 2-19 创建一个含有 1000 万个随机浮点数的数组，把这些浮点数存入文件，再从文件中读取出来。

示例 2-19 创建、保存和加载一个大型浮点数数组

```
>>> from array import array ❶
>>> from random import random
```

```
>>> floats = array('d', (random() for i in range(10**7))) ❷
>>> floats[-1] ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp) ❹
>>> fp.close()
>>> floats2 = array('d') ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7) ❻
>>> fp.close()
>>> floats2[-1] ❼
0.07802343889111107
>>> floats2 == floats ❽
True
```

❶ 导入 `array` 类型。

❷ 从一个可迭代对象（本例使用的是一个生成器表达式）中创建一个双精度浮点数数组（类型代码为 `'d'`）。

❸ 查看数组中的最后一个数。

❹ 把数组存入一个二进制文件。

❺ 创建一个存放双精度浮点数的空数组。

❻ 从二进制文件中读取 1000 万个数。

❼ 查看数组中的最后一个数。

❽ 确认两个数组的内容是一致的。

可以看到，`array.tofile` 和 `array.fromfile` 使用起来并不难。你可以试试，二者的运行速度非常快。经我试验，使用 `array.fromfile` 从 `array.tofile` 创建的二进制文件中加载 1000 万个双精度浮点数，用时大约 0.1 秒。这比从文本文件中读取快了近 60 倍，而且无须使用内置函数 `float` 解析每一行。`array.tofile` 保存文件的速度约比一行一个浮点数写入文本文件快

7 倍。此外，保存 1000 万个双精度浮点数的二进制文件占 80 000 000 字节（一个双精度浮点数占 8 字节，零开销），而保存相同数据量的文本文件占 181 515 739 字节。

如果想使用数值数组表示二进制数据，例如光栅图像，Python 有专门的类型：`bytes` 和 `bytearray`（详见第 4 章）。

最后，比较一下 `list` 和 `array` 的功能，见表 2-3。

表 2-3: `list` 和 `array` 的方法及属性（简单起见，省略了弃用的数组方法和由 `object` 实现的方法）

	<code>list</code>	<code>array</code>	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> ：拼接
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> ：就地拼接
<code>s.append(e)</code>	•	•	在最后一个元素后面追加一个元素
<code>s.byteswap()</code>		•	交换数组中所有项的字节，转换字节序
<code>s.clear()</code>	•		删除所有项
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		浅拷贝列表
<code>s.__copy__()</code>		•	为 <code>copy.copy</code> 提供支持
<code>s.count(e)</code>	•	•	计算一个元素出现的次数

	list	array	
<code>s.__deepcopy__()</code>		•	为优化 <code>copy.deepcopy</code> 提供支持
<code>s.__delitem__(p)</code>	•	•	删除位置 <code>p</code> 上的项
<code>s.extend(it)</code>	•	•	追加可迭代对象 <code>it</code> 中的项
<code>s.frombytes(b)</code>		•	追加字节序列中的项（解释为压缩机器值）
<code>s.fromfile(f, n)</code>		•	追加二进制文件 <code>f</code> 中的 <code>n</code> 项（解释为压缩机器值）
<code>s.fromlist(l)</code>		•	追加列表中的项；一旦抛出 <code>TypeError</code> ，一项也不追加
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> ：获取指定位置上的项或切片
<code>s.index(e)</code>	•	•	查找 <code>e</code> 首次出现的位置
<code>s.insert(p, e)</code>	•	•	在 <code>p</code> 位置上的项之前插入元素 <code>e</code>
<code>s.itemsize</code>		•	数组中每一项的字节长度
<code>s.__iter__()</code>	•	•	获取迭代器
<code>s.__len__()</code>	•	•	<code>len(s)</code> ：项数
<code>s.__mul__(n)</code>	•	•	<code>s * n</code> ：重复拼接
<code>s.__imul__(n)</code>	•	•	<code>s *= n</code> ：就地重复拼接

	<code>list</code>	<code>array</code>	
<code>s.__rmul__(n)</code>	•	•	$n * s$: 反向重复拼接 ^a
<code>s.pop([p])</code>	•	•	删除并返回位置 <code>p</code> 上的项（默认为最后一项）
<code>s.remove(e)</code>	•	•	把 <code>e</code> 的值从首次出现的位置上移除
<code>s.reverse()</code>	•	•	就地反转项的顺序
<code>s.__reversed__()</code>	•		获取从后向前遍历项的迭代器
<code>s.__setitem__(p, e)</code>	•	•	<code>s[p] = e</code> : 把 <code>e</code> 放在位置 <code>p</code> 上, 覆盖现有的项或切片
<code>s.sort([key], [reverse])</code>	•		就地对项排序, <code>key</code> 和 <code>reverse</code> 是可选的关键字参数
<code>s.tobytes()</code>		•	返回项的压缩机器值, 结果为一个 <code>bytes</code> 对象
<code>s.tofile(f)</code>		•	把项的压缩机器值存入二进制文件 <code>f</code>
<code>s.tolist()</code>		•	返回项的数值对象, 结果为一个 <code>list</code> 对象
<code>s.typecode</code>		•	单字符串, 即项的 C 语言类型

^a 反向运算符详见第 16 章。



从 Python3.10 开始, `array` 类型没有列表那种就地排序方法 `sort`。如果需要对数组进行排序, 请使用内置函数 `sorted`

重新构建数组。

```
a = array.array(a.typecode, sorted(a))
```

为了保证增加新项后数组仍然是有序的，请使用 `bisect.insort` 函数。

经常使用数组的人不知道 `memoryview`，可谓遗憾终生。详见接下来的话题。

2.10.2 `memoryview`

内置的 `memoryview` 类是一种共享内存的序列类型，可在不复制字节的情况下处理数组的切片。`memoryview` 类的灵感来自 NumPy 库（见 2.10.3 节）。NumPy 的主要作者 Travis Oliphant 在回答“什么时候适合使用 `memoryview`”问题时说道：

`memoryview` 是 NumPy 中一种普遍使用的结构，本质上就是 Python 中的数组（除去数学功能）。`memoryview` 在数据结构（例如 PIL 图像、SQLite 数据库、NumPy 数组等）之间共享内存，而不是事先复制。这对大型数据集来说非常重要。

`memoryview.cast` 方法使用的表示法与 `array` 模块类似，作用是改变读写多字节单元的方式，无须移动位。`memoryview.cast` 方法返回另一个 `memoryview` 对象，而且始终共享内存。

示例 2-20 展示了如何将同一个 6 字节数组处理为不同的视图，先是一个 2×3 矩阵，后是一个 3×2 矩阵。

示例 2-20 分别以 1×6、2×3 和 3×2 矩阵的视图处理 6 字节内存

```
>>> from array import array
>>> octets = array('B', range(6)) ❶
>>> m1 = memoryview(octets) ❷
```

```

>>> m1.tolist()
[0, 1, 2, 3, 4, 5]
>>> m2 = m1.cast('B', [2, 3]) ❸
>>> m2.tolist()
[[0, 1, 2], [3, 4, 5]]
>>> m3 = m1.cast('B', [3, 2]) ❹
>>> m3.tolist()
[[0, 1], [2, 3], [4, 5]]
>>> m2[1,1] = 22 ❺
>>> m3[1,1] = 33 ❻
>>> octets ❼
array('B', [0, 1, 2, 33, 22, 5])

```

- ❶ 创建一个 6 字节数组（类型代码为 'B'）。
- ❷ 根据这个数组创建一个 `memoryview` 对象，然后导出为一个列表。
- ❸ 根据前一个 `memoryview` 对象构建一个新 `memoryview` 对象，不过是 2 行 3 列。
- ❹ 再构建一个 `memoryview` 对象，这一次是 3 行 2 列。
- ❺ 使用 22 覆盖 `m2` 中行 1 列 1 上的字节。
- ❻ 使用 33 覆盖 `m3` 中行 1 列 1 上的字节。
- ❼ 显示原数组，证明 `octets`、`m1`、`m2` 和 `m3` 之间的内存是共享的。

`memoryview` 的强大功能也可用于搞破坏。示例 2-21 展示如何修改一个 16 位整数数组中某一项的一个字节。

示例 2-21 修改一个 16 位整数数组中某一项的字节，改变该项的值

```

>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers) ❶
>>> len(memv)
5

```

```
>>> memv[0] ❷  
-2  
>>> memv_oct = memv.cast('B') ❸  
>>> memv_oct.tolist() ❹  
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]  
>>> memv_oct[5] = 4 ❺  
>>> numbers  
array('h', [-2, -1, 1024, 1, 2]) ❻
```

❶ 根据一个包含 5 个 16 位带符号整数的数组（类型代码为 'h'）创建一个 `memoryview` 对象。

❷ `memv` 中的 5 项与原数组一样。

❸ 把 `memv` 中的元素转换成字节（类型代码为 'B'），创建 `memv_oct` 对象。

❹ 导出 `memv_oct` 中的元素，显示为一个包含 10 个字节的列表，方便查看。

❺ 把偏移位 5 上的字节设为 4。

❻ 注意 `numbers` 的变化：对于一个 2 字节无符号整数，最高有效字节为 4，对应的十进制值是 1024。

另外，如果你想对数组做一些高级数值处理，应该使用 NumPy 库。时不我待，现在就开始吧。

2.10.3 NumPy

本书重点关注的是 Python 标准库中已有的功能，着重讲解如何充分利用它们。但是，NumPy 太优秀了，值得单列出来讲一讲。

科学计算经常需要做一些高级数组和矩阵运算，得益于 NumPy，Python 成为这一领域的主流语言。NumPy 实现了多维同构数组和矩

阵类型，除了存放数值之外，还可以存放用户定义的记录，而且提供了高效的元素层面操作。

在 NumPy 基础之上编写的 SciPy 库提供了许多科学计算算法，从线性代数到数值微积分和统计学，不一而足。SciPy 速度快、运算可靠，因为它大量沿用了 Netlib Repository 的 C 语言和 Fortran 基准代码。换句话说，SciPy 为科学家提供了两全其美的工具，既有交互式提示符，又有高级 Python API，另外还通过 C 语言和 Fortran 优化了工业级数值运算函数。

示例 2-22 简单演示 NumPy 的用法，对二维数组做了些基本操作。

示例 2-22 numpy.ndarray 中行和列的基本操作

```
>>> import numpy as np ❶
>>> a = np.arange(12) ❷
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape ❸
(12,)
>>> a.shape = 3, 4 ❹
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[2] ❺
array([ 8,  9, 10, 11])
>>> a[2, 1] ❻
9
>>> a[:, 1] ❼
array([1, 5, 9])
>>> a.transpose() ❽
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

❶ 导入安装好的 NumPy（NumPy 不在 Python 标准库中）。习惯上把 numpy 导入为 np。

- ❷ 构建一个从整数 0 到 11 的 `numpy.ndarray` 对象，然后查看该对象的内容。
- ❸ 查看数组的维度：这是一个一维数组，含有 12 个元素。
- ❹ 改变数组的维度：增加一个维度，然后查看结果。
- ❺ 获取索引位 2 上的行。
- ❻ 获取索引位 2, 1 上的元素。
- ❼ 获取索引位 1 上的列。
- ❽ 转置数组（行列交换），创建一个新数组。

NumPy 还支持一些高级操作，例如加载、保存和操作 `numpy.ndarray` 对象的所有元素。

```
>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt') ❶
>>> floats[-3:] ❷
array([ 3016362.69195522,   535281.10514262,   4566560.44373946])
>>> floats *= .5 ❸
>>> floats[-3:]
array([ 1508181.34597761,   267640.55257131,   2283280.22186973])
>>> from time import perf_counter as pc ❹
>>> t0 = pc(); floats /= 3; pc() - t0 ❺
0.03690556302899495
>>> numpy.save('floats-10M', floats) ❻
>>> floats2 = numpy.load('floats-10M.npy', 'r+') ❼
>>> floats2 *= 6
>>> floats2[-3:] ❽
memmap([ 3016362.69195522,   535281.10514262,   4566560.44373946])
```

- ❶ 从一个文本文件中加载 1000 万个浮点数。
- ❷ 使用序列切片表示法查看最后 3 个数。

- ③ 把 `floats` 数组中的每个元素都乘以 `.5`，然后再次查看最后 3 个元素。
- ④ 导入高分辨率性能衡量计时器（自 Python 3.3 起可用）。
- ⑤ 把每个元素除以 3。1000 万个浮点数耗时不超过 40 毫秒。
- ⑥ 把数组保存到一个 `.npy` 二进制文件中。
- ⑦ 以内存映射文件的形式把数组加载到另一个数组中。即使文件较大，内存放不下，这样做也能高效处理数组切片。
- ⑧ 每个元素乘以 6 之后，查看最后 3 个元素。

这仅仅是开始。

NumPy 和 SciPy 这两个库的功能异常强大，为很多优秀的工具提供了坚实的基础，例如 Pandas 和 scikit-learn。Pandas 实现的高效数组类型可以保存非数值数据，此外还支持导入和导出多种格式，包括 `.csv`、`.xls`、SQL 转储、HDF5 等。scikit-learn 是目前最广泛使用的机器学习工具集。NumPy 和 SciPy 这两个库中的函数大多是用 C 或 C++ 实现的，可以利用所有 CPU 核，因为它们释放了 Python 的全局解释器锁（Global Interpreter Lock, GIL）。Dask 项目支持跨设备集群并行处理 NumPy、Pandas 和 scikit-learn 操作。这些包各自都可以写一本书，不是本书关注的重点。然而，此处又不得不提，因为如果不简单介绍一下 NumPy 数组，你就无法看到 Python 序列的全貌。

我们已经讲了两种扁平序列：标准数组和 NumPy 数组。下面换个话题，讲一种完全不同的列表替代结构——队列。

2.10.4 双端队列和其他队列

借助 `.append` 和 `.pop` 方法，列表可以当作栈或队列使用

（`.append` 和 `.pop(0)` 实现的是先进先出行为）。但是，在列表头

部（索引位为 0）插入和删除项有一定开销，因为整个列表必须在内存中移动。

`collections.deque` 类实现一种线程安全的双端队列，旨在快速在两端插入和删除项。如果需要保留“最后几项”，或者实现类似的行为，则双端队列是唯一选择，因为 `deque` 对象可以有界，即长度固定。有界的 `deque` 对象填满之后，从一端添加新项，将从另一端丢弃一项。示例 2-23 展示了可对 `deque` 对象执行的一些典型操作。

示例 2-23 处理一个 `deque` 对象

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3) ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1) ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33]) ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40]) ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

❶ 可选的 `maxlen` 参数设定 `deque` 实例中最多允许存放多少项。
`maxlen` 也是 `deque` 实例的一个只读属性。

❷ 轮转，当 $n > 0$ ，从右端取几项放到左端；当 $n < 0$ ，从左端取几项放到右端。

❸ 向已满 (`len(d) == d.maxlen`) 的 `deque` 对象中的一端追加几项，则另一端要丢弃几项。注意下一行，可以看到 0 没有了。

④ 在右端添加 3 项，把左端前 3 项 -1、1 和 2 挤出队列。

⑤ 注意，`extendleft(iter)` 依次把 `iter` 参数中的各项追加到 `deque` 对象的左端，因此项之间的位置顺序得到保留。

`list` 和 `deque` 之间的方法（同样不含 `object` 实现的方法）对比见表 2-4。

注意，`deque` 实现了多数 `list` 方法，另外增加了一些专用方法，例如 `popleft` 和 `rotate`。不过，这里隐藏一个不太高效的操作：从 `deque` 对象中部删除项的速度不快。双端队列优化的是在两端增减项的操作。

`append` 和 `popleft` 是原子操作，因此你可以放心地在多线程应用中把 `deque` 作为先进先出队列使用，无须加锁。

表 2-4：list 和 deque 实现的方法（简单起见，省略了 object 实现的方法）

	list	deque	
<code>s.__add__(s2)</code>	•		<code>s + s2</code> ：拼接
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> ：就地拼接
<code>s.append(e)</code>	•	•	把一个元素追加到右端（放在最后一个元素后面）
<code>s.appendleft(e)</code>		•	把一个元素追加到左端（放在第一个元素前面）
<code>s.clear()</code>	•	•	删除所有项

	list	deque	
<code>s.__contains__(e)</code>	•		<code>e in s</code>
<code>s.copy()</code>	•		浅拷贝列表
<code>s.__copy__()</code>		•	为 <code>copy.copy</code> （浅拷贝）提供支持
<code>s.count(e)</code>	•	•	计算一个元素出现的次数
<code>s.__delitem__(p)</code>	•	•	删除位置 <code>p</code> 上的项
<code>s.extend(i)</code>	•	•	把可迭代对象 <code>i</code> 中的项追加到右端
<code>s.extendleft(i)</code>		•	把可迭代对象 <code>i</code> 中的项追加到左端
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> ：获取位置 <code>p</code> 上的项或切片
<code>s.index(e)</code>	•		查找 <code>e</code> 首次出现的位置
<code>s.insert(p, e)</code>	•		把元素 <code>e</code> 插入位置 <code>p</code> 上的项前面
<code>s.__iter__()</code>	•	•	获取迭代器
<code>s.__len__()</code>	•	•	<code>len(s)</code> ：项数
<code>s.__mul__(n)</code>	•		<code>s * n</code> ：重复拼接

	list	deque	
<code>s.__imul__(n)</code>	•		<code>s *= n</code> : 就地重复拼接
<code>s.__rmul__(n)</code>	•		<code>n * s</code> : 反向重复拼接 ^a
<code>s.pop()</code>	•	•	删除并返回最后一项 ^b
<code>s.popleft()</code>		•	删除并返回第一项
<code>s.remove(e)</code>	•	•	把 <code>e</code> 的值从首次出现的位置上移除
<code>s.reverse()</code>	•	•	就地反转项的顺序
<code>s.__reversed__()</code>	•	•	获取从后向前遍历项的迭代器
<code>s.rotate(n)</code>		•	把 <code>n</code> 项从一端移到另一端
<code>s.__setitem__(p, e)</code>	•	•	<code>s[p] = e</code> : 把 <code>e</code> 放在位置 <code>p</code> 上, 覆盖现有的项或切片
<code>s.sort([key], [reverse])</code>	•		就地对项排序, <code>key</code> 和 <code>reverse</code> 是可选的关键字参数

^a 反向运算符详见第 16 章。

^b 使用 `a_list.pop(p)` 可以删除列表中位置 `p` 上的项, 但是 `deque` 不支持这么做。

除了 `deque` 之外, Python 标准库中的其他包还实现了以下队列。

`queue`

提供几个同步（即线程安全）队列类：`SimpleQueue`、`Queue`、`LifoQueue` 和 `PriorityQueue`。这些类可在线程之间安全通信。除 `SimpleQueue` 之外，其他几个类都可以有界——为构造函数提供 `maxsize` 参数，设为大于 0 的值。但是，它们不像 `deque` 那样为了腾出空间而把项丢弃，而是在队列填满后阻塞插入新项，等待其他线程从队列中取出一项。利用这种行为可以限制活动线程的数量。

`multiprocessing`

单独实现了无界的 `SimpleQueue` 和有界的 `Queue`。这与 `queue` 包中的队列类非常相似，只不过专门针对进程间通信。它还为任务管理提供了专用的 `multiprocessing.JoinableQueue`。

`asyncio`

提供了 `Queue`、`LifoQueue`、`PriorityQueue` 和 `JoinableQueue`，API 源自 `queue` 和 `multiprocessing` 模块中的类，不过为管理异步编程任务而做了修改。

`heapq`

与前三个模块相比，`heapq` 没有实现任何队列类，而是提供了 `heappush` 和 `heappop` 等函数，可把可变序列当作堆队列或优先级队列使用。

我们对 `list` 类型替代方案的介绍到此结束，对一般序列类型的探讨也暂告一段落，剩下的 `str` 和二进制序列将在第 4 章解读。

2.11 本章小结

若想写出简洁、有效和地道的 Python 代码，势必要掌握标准库中的各种序列类型。

Python 序列通常按可变性分类，不过也可以换个角度，分成扁平序列和容器序列。前者结构更紧凑、处理速度更快，也更易于使用，但是仅限于存储原子数据，例如数值、字符和字节。容器序列更加灵活，不过存储可变对象的行为或许会让你感到惊讶，因此遇到嵌套数据结构时务必小心谨慎。

可惜，Python 中的不可变容器序列类型也不能保证万无一失。即便是“不可变的”元组，如果存有可变的项，例如列表或用户定义的对象，则其值也是可变的。

列表推导式和生成器表达式是构建和初始化序列的强大表示法。如果你还不习惯使用，请花点时间掌握基本用法，这并不难，你很快就会顺手。

元组在 Python 中扮演两个角色，一是不具名字段记录，二是不可变列表。把元组当作不可变列表使用时请记住，仅当元组中的所有项也都是不可变对象时，才能保证元组值是固定的。在元组上调用 `hash(t)` 函数可以快速判断元组的值是否固定。如果 `t` 包含可变的项，则 `hash(t)` 抛出 `TypeError`。

把元组当作记录使用时，元组拆包是提取元组字段最安全、可读性最高的方法。除了元组之外，`*` 在许多上下文中还适用于列表和可迭代对象。`*` 的一些用途由“PEP 448—Additional Unpacking Generalizations”引入，出现在 Python 3.5 中。Python 3.10 引入以 `match/case` 句法表示的模式匹配，开始支持更强大的拆包功能，即所谓的析构。

序列切片是最受欢迎的 Python 句法特性之一，其功能比许多人所想的还要强大。用户定义的序列甚至可以支持 NumPy 那种多维切片和省略号 (`...`) 表示法。通过切片赋值修改可变序列是极具表现力的操作。

重复拼接（例如 `seq * n`）很方便，若使用得当，可用于初始化内含不可变项的嵌套列表。`+=` 和 `*=` 两个增量赋值运算符的行为因序列的可变性而异。对于不可变序列，必须构建新序列。如果目标序列是可变的，则往往就地修改，但也不尽然，具体要看序列的实现。

`sort` 方法和内置的 `sorted` 函数易于使用，十分灵活，这要归功于可选的 `key` 参数，其值为一个计算排序标准的函数。顺便说一下，内置函数 `min` 和 `max` 也有 `key` 参数。

除了列表和元组，Python 标准库还提供了 `array.array`。尽管 NumPy 和 SciPy 不在标准库中，但是如果需要对大型数据集做任何数值处理，哪怕只学习一点也能受用很长时间。

最后，我们窥探了用途多样、对线程安全的 `collections.deque`，在表 2-4 中与 `list` 的 API 做了对比，还提到了标准库实现的其他队列。

2.12 延伸阅读

《Python Cookbook 中文版（第 3 版）》（David Beazley 和 Brian K. Jones 著）第 1 章有很多序列相关的经典实例，比如“1.11 对切片命名”，我从中学会了把切片赋值给变量（就像示例 2-13 那样），以此提升代码可读性。

《Python Cookbook 中文版（第 2 版）》针对 Python 2.4，不过大部分内容也适用于 Python 3。这一版中的第 5 章和第 6 章有许多关于序列的经典实例。第 2 版由 Alex Martelli、Anna Ravenscroft 和 David Ascher 编辑，融汇了多位 Python 专家的智慧。第 3 版推倒重来，着重关注 Python 语言的语义，尤其是 Python 3 的变化，而第 2 版更强调语用，即如何使用 Python 语言实际问题。尽管第 2 版中的一些解决方案已经过时，不过我还是认为两版都值得拥有。

Python 官方文档中的“Sorting HOW TO”一文举了几个例子，说明 `sorted` 和 `list.sort` 的高级技巧。

如果你想了解并行赋值语句左手边可用的新句法 `*extra`，“PEP 3132—Extended Iterable Unpacking”是最权威的资料。如果你想了解 Python 的发展过程，可以查看 bug 跟踪程序中的“Missing *-unpacking generalizations”工单，这里提议增强可迭代对象的拆包表示法。这个工单的讨论结果汇集为“PEP 448—Additional Unpacking Generalizations”。

2.6 节提过，在“What's New In Python 3.10”一文中，Carol Willing 写的“Structural Pattern Matching”一节大约 1400 字（在 Firefox 中把 HTML 打印为 PDF，不到 5 页），很好地介绍了这个新增功能。“PEP 636—Structural Pattern Matching: Tutorial”也不错，只是内容较长。PEP 636 中的“Appendix A—Quick Intro”一节比 Willing 写的介绍还短，因为这里没有综述模式匹配的优势。如果你需要更多的论据来说服自己或他人，Python 需要模式匹配，请阅读长达 22 页的“PEP 635—Structural Pattern Matching: Motivation and Rationale”。

Eli Bendersky 写的博客文章“Less copies in Python with the buffer protocol and memoryviews”是一篇简短的 `memoryview` 教程。

讲 NumPy 的书很多，有些书名甚至没有提及“NumPy”。推荐两本书：《Python 数据科学手册》¹⁵ 和《利用 Python 进行数据分析》（第 2 版）。

¹⁵该书中文版已由人民邮电出版社图灵公司出版：it-ebook.com/1937。——编者注

“NumPy 建筑在向量之上。”这是 *From Python to NumPy* (Nicolas P. Rougier 著) 一书的开首语。向量化运算把数学函数应用到数组的所有元素上，无须显式编写 Python 循环。相关操作可以并行执行，使用现代 CPU 中的特殊向量指令，利用多个核或者委托给 GPU（取决于具体的库）。这本书中的第一个示例把一个使用生成器的 Python

类重构为一个精简的函数，调用几个 NumPy 函数便把速度提升了 500 倍。

如果想学习如何使用 deque（以及其他容器），可以阅读 Python 文档中“Container datatypes”一文内的示例和实践诀窍。

Edsger W. Dijkstra 写的一份简短备忘录对 Python 排除区间和切片中最后一项的方法做了最好的辩护，题为“Why Numbering Should Start at Zero”。这篇备忘录的主题是数学符号，不过与 Python 也有关系。Dijkstra 行文严谨、幽默，解释了像 2, 3, ..., 12 这样的序列为什么始终应该通过 $2 \leq i < 13$ 表达，除此之外也有其他约定写法，可能合理，但是均被否定，因为不能让每一个用户都各行其是。虽然这篇备忘录的标题指的是基于零的索引，但是文内讨论的其实是为什么 `'ABCDE'[1:3]` 得到的是 `'BC'` 而不是 `'BCD'`，以及为什么通过 `range(2, 13)` 生成 2, 3, 4, ..., 12 最符合常理。顺便说一下，这篇备忘是手写笔记，不过字迹清晰可辨。甚至还有人根据这篇笔记设计了一种字体。

杂谈

元组的本质

2012 年，我在 PyCon US 上展示了一张关于 ABC 语言的海报。创造 Python 之前，Guido van Rossum 曾开发过 ABC 语言的解释器，所以他过来看了我的海报。交谈中我们讲到了 ABC 语言的 compound 类型，这显然是 Python 元组的前身。compound 类型还支持并行赋值，也可用作字典（或 ABC 语言中的 Table 类型）的组合键。然而，compound 类型不是序列，不可迭代，不能通过索引获取字段，更别提切片了。compound 类型只能作为一个整体处理，或者通过并行赋值提取各个字段，仅此而已。

我跟 Guido 讲，这些限制凸显了 compound 类型的主要用途，即没有具名字段的记录。他回应道：“为了让元组显现序列行为，可是费了一番功夫。”

这是一种务实的做法，正是因为 Python 更务实，所以才比 ABC 语言成功。从语言实现者的角度来看，让元组表现为序列的成本很低。这样做虽然把记录这种用途弱化了，但得到了一种不可变列表——把元组类型命名为 `frozenset` 或许更明显。

扁平序列与容器序列

为了突出不同序列类型采用的内存模型，我发明了术语“容器序列”和“扁平序列”。“容器”这个词来取自“Data Model”文档。

一些对象包含对其他对象的引用，这种对象称为容器。

为了表达准确，我把“容器”和“序列”连在一起，因为 Python 中有些容器不是序列，例如 `dict` 和 `set`。容器类型可以嵌套，包含任何类型的对象，甚至是容器类型自身。

另一方面，扁平序列是不可嵌套的序列类型，只能包含简单的原子类型，例如整数、浮点数或字符。

采用“扁平序列”这个术语是为了与“容器序列”区分开。

在官方文档中，除了前面提到的“容器”之外，`collections.abc` 中还有一个抽象类名为 `Container`。这个抽象基类只有一个方法，即 `in` 运算符背后的特殊方法 `__contains__`。字符串和数组不是传统意义上的容器，却是 `Container` 的虚拟子类，因为它们实现了 `__contains__` 方法。这表明人类经常使用同一个词指代不同的事物。¹⁶ 本书使用“容器”一词指代可包含其他对象引用的对象，使用 `Container` 指代 `collections.abc.Container` 类。

大杂烩列表

Python 入门教程往往强调列表可以包含不同类型的对象，但是在实践中，这样做没有什么用。我们把一些项放在列表中，是为了后续处理，这就隐含了一层意思，即所有项至少都应该支持同一

种操作（也就是说，无论在基因上是否 100% 是鸭子，但是都应该“嘎嘎叫”）。例如，在 Python 3 中，如果列表中的项不可比较，那就不能排序。

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

与列表不同，元组通常包含不同类型的项。这也符合常理：如果元组中的每一项都是一个字段，那么每个字段就可以具有不同的类型。

聪明的 key

`list.sort`、`sorted`、`max` 和 `min` 都有 `key` 参数，这真是一个好想法。其他语言要求提供带两个参数的比较函数，例如 Python 2 中的 `cmp(a, b)` 函数（现已弃用）。使用 `key` 既简单又高效。说简单是因为，你只需要定义一个单参数函数，通过它获取或计算想用于排序对象的标准，这比编写返回 `-1`、`0` 或 `1` 的双参数函数更容易。说高效是因为，`key` 指定的函数只在处理各项时调用一次，而双参数比较函数在每次需要通过排序算法比较两项时都要调用一次。当然，Python 在排序时也要比较键，但是这种比较交给优化的 C 语言代码完成，而不是你自己编写的 Python 函数。

顺便说一下，使用 `key` 参数，哪怕掺杂数值和类似数值的字符串，也可以排序。我们只需要决定把所有项全都视为整数还是字符串。

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
>>> sorted(l, key=str)
[0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

甲骨文、谷歌和 Timbot 阴谋论

`Oracle` 和 `list.sort` 使用的排序算法是 Timsort。这是一种自适应算法，可根据数据的排序方式在插入排序法和归并排序法之间切换。这样做效率高，因为现实中的数据，某些项往往具有一定顺序。

Timsort 于 2002 年首次在 CPython 中使用。自 2009 年起，标准的 Java 和 Android 也使用 Timsort 排序数组。后来发生的事广为人知，甲骨文以 Timsort 相关的代码为证据，状告谷歌侵犯 Sun 公司的知识产权。2021 年，美国最高法院裁定，谷歌对 Java 代码的使用属于“合理使用”范围。

Timsort 由 Tim Peters 发明。Tim 是 Python 核心开发人员，作品车载斗量，生产力就像 AI 一样，人称 Timbot。你可以在“Python Humor”页面找到这个阴谋论。《Python 之禅》（`import this`）也是 Tim 写的。

¹⁶中文版把“collection”一词翻译为“容器”，而不是常见的“集合”，目的是与“set”（数学中的集合）区分开。用词不同，但道理相同。请读者根据上下文区分具体指代的事物。——译者注

第 3 章 字典和集合

说起来，Python 就是包裹在一堆语法糖中的字典。

——Lalo Martins

早期数字游民，Python 专家

没有 Python 程序不使用字典，即使不直接出现在我们自己编写的代码中，我们也间接用到了，因为 `dict` 类型是实现 Python 的基石。一些 Python 核心结构在内存中以字典的形式存在，比如说类和实例属性、模块命名空间，以及函数的关键字参数。另外，`__builtins__.__dict__` 存储着所有内置类型、对象和函数。

由于字典的关键作用，Python 对字典做了高度优化，而且一直在改进。Python 字典能如此高效，要归功于**哈希表**。

除了字典之外，内置类型中 `set` 和 `frozenset` 也基于哈希表。这两种类型的 API 和运算符比其他流行语言中的集合更丰富。具体而言，Python 集合实现了集合论中的所有基本运算，包括并集、交集、子集测试等。有了这些运算，就能以更具描述性的方式表达算法，避免了一层层的嵌套循环和测试条件。

本章涵盖以下内容：

- 构建及处理 `dict` 和映射的现代句法，包括增强的拆包和模式匹配；
- 映射类型的常用方法；
- 特别处理缺失键的情况；
- 标准库中的 `dict` 变体；
- `set` 和 `frozenset` 类型；
- 哈希表对集合和字典行为的影响。

3.1 本章新增内容

第 2 版的改动大多是对映射类型新功能的介绍。

- 3.2 节介绍增强的拆包句法和合并映射的不同方式，包括自 Python 3.9 开始受 `dict` 支持的 `|` 和 `|=` 运算符。
- 3.3 节说明如何使用 Python 3.10 引入的 `match/case` 句法处理映射。
- 3.6.1 节现在关注的是 `dict` 和 `OrderedDict` 之间微小却不可忽略的差异，毕竟自 Python 3.6 起，`dict` 也能保留键的插入顺序。
- 新增 3.8 节和 3.12 节，讲解 `dict.keys`、`dict.items` 和 `dict.values` 返回的视图对象。

`dict` 和 `set` 的底层实现仍然依赖于哈希表，不过 `dict` 的代码有两项重要的优化，节省了内存，还能保留键的插入顺序。3.9 节和 3.11 节综述如何利用这两种类型。

3.2 字典的现代句法

接下来几节介绍用于构建、拆包和处理映射的高级语法功能。其中一些功能在 Python 中早已出现，不过你可能还不知道。还有一些功能相对较新，例如 Python 3.9 引入的 `|` 运算符和 Python 3.10 引入的 `match/case` 句法。先我们从熟知的功能讲起。

3.2.1 字典推导式

自 Python 2.7 开始，列表推导式和生成器表达式经过改造，以适用于字典推导式（以及后文要讲的集合推导式）。**字典推导式**从任何可迭代对象中获取键值对，构建 `dict` 实例。示例 3-1 使用字典推导式根据同一个元组列表构建两个字典。

示例 3-1 字典推导式示例

```
>>> dial_codes = [
❶ ...     (880, 'Bangladesh'),
...     (55, 'Brazil'),
...     (86, 'China'),
...     (91, 'India'),
...     (62, 'Indonesia'),
...     (81, 'Japan'),
...     (234, 'Nigeria'),
...     (92, 'Pakistan'),
...     (7, 'Russia'),
...     (1, 'United States'),
... ]
>>> country_dial = {country: code for code, country in dial_codes}
❷ >>> country_dial
{'Bangladesh': 880, 'Brazil': 55, 'China': 86, 'India': 91,
 'Indonesia': 62,
 'Japan': 81, 'Nigeria': 234, 'Pakistan': 92, 'Russia': 7, 'United
States': 1}
>>> {code: country.upper()
❸ ...     for country, code in sorted(country_dial.items())
...     if code < 70}
{55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA', 1: 'UNITED STATES'}
```

❶ 像 `dial_codes` 这种包含键值对的可迭代对象可以直接传给 `dict` 构造函数，但是.....

❷ 这里，我们对调了键和值的位置，以 `country` 为键，以 `code` 为值。

❸ 按国家名称排序 `country_dial`，再次对调键和值，把值转换成全大写形式，并通过 `code < 70` 条件筛选项。

习惯列表推导式之后，自然能理解字典推导式。即使现在不理解也没关系，推导式句法已经被人们广泛使用，熟练掌握是迟早的事。

3.2.2 映射拆包

从 Python3.5 开始, “PEP 448—Additional Unpacking Generalizations”在两方面增强了映射拆包功能。

首先, 调用函数时, 不止一个参数可以使用 `**`。但是, 所有键都要是字符串, 而且在所有参数中是唯一的 (因为关键字参数不可重复)。

```
>>> def dump(**kwargs):
...     return kwargs
...
>>> dump(**{'x': 1}, y=2, **{'z': 3})
{'x': 1, 'y': 2, 'z': 3}
```

其次, `**` 可在 `dict` 字面量中使用, 同样可以多次使用。

```
>>> {'a': 0, **{'x': 1}, 'y': 2, **{'z': 3, 'x': 4}}
{'a': 0, 'x': 4, 'y': 2, 'z': 3}
```

这种情况下允许键重复, 后面的键覆盖前面的键, 比如本例中 `x` 映射的值。

这种句法也可用于合并映射, 但是合并映射还有其他方式。请继续往下读。

3.2.3 使用 `|` 合并映射

Python 3.9 支持使用 `|` 和 `|=` 合并映射。这不难理解, 因为二者也是并集运算符。

`|` 运算符创建一个新映射。

```
>>> d1 = {'a': 1, 'b': 3}
>>> d2 = {'a': 2, 'b': 4, 'c': 6}
>>> d1 | d2
{'a': 2, 'b': 4, 'c': 6}
```

通常, 新映射的类型与左操作数 (本例中的 `d1`) 的类型相同。不过, 涉及用户定义的类型时, 也可能与第二个操作数的类型相同。这背后

涉及运算符重载规则，详见第 16 章。

如果想就地更新现有映射，则使用 `|=`。续前例，当时 `d1` 没有变化，但是现在变了。

```
>>> d1
{'a': 1, 'b': 3}
>>> d1 |= d2
>>> d1
{'a': 2, 'b': 4, 'c': 6}
```



如果你需要维护使用 Python 3.8 或更早版本编写的代码，请看“PEP 584— Add Union Operators To dict”中的“Motivation”一节，那里简单总结了合并映射的其他方式。

下面来看如何使用模式匹配处理映射。

3.3 使用模式匹配处理映射

`match/case` 语句的匹配对象可以是映射。映射的模式看似 `dict` 字面量，其实能够匹配 `collections.abc.Mapping` 的任何具体子类或虚拟子类。¹

¹虚拟子类是调用抽象基类的 `.register()` 方法注册的类，详见 13.5.6 节。通过 Python 或 C 语言 API 实现的类型，如果设定了 `Py_TPFLAGS_MAPPING` 标记位，那么也能匹配。

第 2 章只重点讲了序列模式，其实不同类型的模式可以组合和嵌套。模式匹配是一种强大的工具，借助析构可以处理嵌套的映射和序列等结构化记录。我们经常需要从 JSON API 和具有半结构化模式的数据库（例如 MongoDB、EdgeDB 或 PostgreSQL）中读取这类记录，示例 3-2 就是一例。`get_creators` 函数有一些简单的类型注解，作用是明确表明参数为一个 `dict`，返回值是一个 `list`。

示例 3-2 creator.py: get_creators() 函数从出版物记录中提取创作者的名字

```
def get_creators(record: dict) -> list:
    match record:
        case {'type': 'book', 'api': 2, 'authors': [*names]}: ❶
            return names
        case {'type': 'book', 'api': 1, 'author': name}: ❷
            return [name]
        case {'type': 'book'}: ❸
            raise ValueError(f"Invalid 'book' record: {record!r}")
        case {'type': 'movie', 'director': name}: ❹
            return [name]
        case _: ❺
            raise ValueError(f'Invalid record: {record!r}')
```

❶ 匹配含有 'type': 'book', 'api': 2, 而且 'authors' 键映射一个序列的映射对象。以列表形式返回序列中的项。

❷ 匹配含有 'type': 'book', 'api': 1, 而且 'author' 键映射任何对象的映射对象。以列表形式返回匹配的对象。

❸ 其他含有 'type': 'book' 的映射均无效, 抛出 ValueError。

❹ 匹配含有 'type': 'movie', 而且 'director' 映射单个对象的映射对象。以列表形式返回匹配的对象。

❺ 其他匹配对象均无效, 抛出 ValueError。

通过示例 3-2 可以看出处理半结构化数据 (例如 JSON 记录) 的几点注意事项:

- 包含一个描述记录种类的字段 (例如 'type': 'movie') ;
- 包含一个标识模式版本的字段 (例如 'api': 2) , 方便公开 API 版本更迭;
- 包含处理特定无效记录 (例如 'book') 的 case 子句, 以及兜底 case 子句。

下面在 doctest 中测试一下 get_creators 函数。

```
>>> b1 = dict(api=1, author='Douglas Hofstadter',
...           type='book', title='Gödel, Escher, Bach')
>>> get_creators(b1)
['Douglas Hofstadter']
>>> from collections import OrderedDict
>>> b2 = OrderedDict(api=2, type='book',
...                   title='Python in a Nutshell',
...                   authors='Martelli Ravenscroft Holden'.split())
>>> get_creators(b2)
['Martelli', 'Ravenscroft', 'Holden']
>>> get_creators({'type': 'book', 'pages': 770})
Traceback (most recent call last):
...
ValueError: Invalid 'book' record: {'type': 'book', 'pages': 770}
>>> get_creators('Spam, spam, spam')
Traceback (most recent call last):
...
ValueError: Invalid record: 'Spam, spam, spam'
```

注意，模式中键的顺序无关紧要。即使 b2 是一个 OrderedDict，也能作为匹配对象。

与序列模式不同，就算只有部分匹配，映射模式也算成功匹配。在上述 doctest 中，b1 和 b2 两个匹配对象中都有 'title' 键，尽管任何 'book' 模式中都没有这个键，但依然可以匹配。

没有必要使用 **extra 匹配多出的键值对，倘若你想把多出的键值对捕获到一个 dict 中，可以在一个变量前面加上 **，不过必须放在模式最后。**_ 是无效的，纯属画蛇添足。下面是一个简单的例子。

```
>>> food = dict(category='ice cream', flavor='vanilla', cost=199)
>>> match food:
...     case {'category': 'ice cream', **details}:
...         print(f'Ice cream details: {details}')
...
Ice cream details: {'flavor': 'vanilla', 'cost': 199}
```

3.5 节讲解的 defaultdict 等映射，通过 __getitem__ 查找键（即 d[key]）始终成功。这是因为倘若缺失某一项，则自动创建。

对模式匹配而言，仅当匹配对象在运行 `match` 语句之前已经含有所需的键才能成功匹配。



模式匹配不自动处理缺失的键，因为模式匹配始终使用 `d.get(key, sentinel)` 方法。其中，`sentinel` 是特殊的标记值，不会出现在用户数据中。

讲完句法和结构之后，接下来研究映射的 API。

3.4 映射类型的标准 API

`collections.abc` 模块中的抽象基类 `Mapping` 和 `MutableMapping` 描述 `dict` 和类似类型的接口，见图 3-1。

这两个抽象基类的主要作用是确立映射对象的标准接口，并在需要广义上的映射对象时为 `isinstance` 提供测试标准。

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
>>> isinstance(my_dict, abc.MutableMapping)
True
```



提示

使用 `isinstance` 测试是否满足抽象基类的接口要求，往往比检查一个函数的参数是不是具体的 `dict` 类型要好，因为除了 `dict` 之外还有其他映射类型可用。这个问题将在第 13 章详谈。

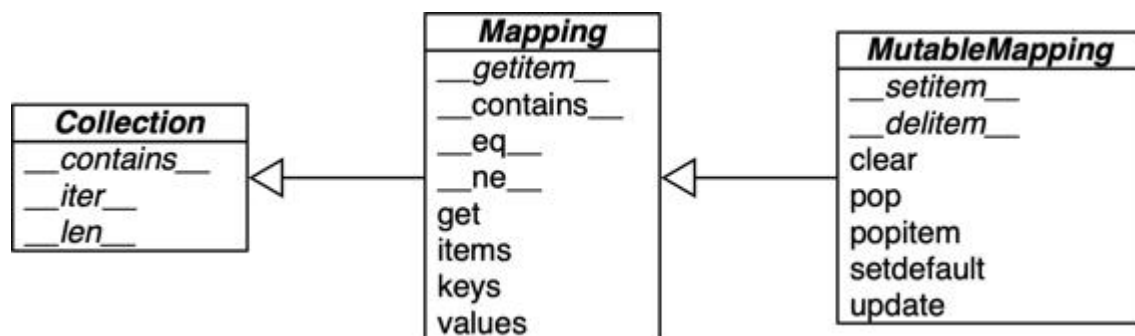


图 3-1: MutableMapping 及其在 collections.abc 中的超类的简化 UML 类图

如果想自定义映射类型，扩展 `collections.UserDict` 或通过组合模式包装 `dict` 更简单，那就不要定义这些抽象基类的子类。`collections.UserDict` 类和标准库中的所有具体映射类都在实现中封装了基本的 `dict`，而 `dict` 又建立在哈希表之上。因此，这些类有一个共同的限制，即键必须**可哈希**（值不要求可哈希，只有键需要）。3.4.1 节会讲一讲相关概念，帮你回顾一下。

3.4.1 “可哈希”指什么

“可哈希”在 Python 术语表中有定义，摘录（有部分改动）如下：

如果一个对象的哈希码在整个生命周期内永不改变（依托 `__hash__()` 方法），而且可与其他对象比较（依托 `__eq__()` 方法），那么这个对象就是可哈希的。两个可哈希对象仅当哈希码相同时相等。²

²Python 术语表中的“可哈希”词条使用的是“哈希值”，而不是“哈希码”。我倾向于使用“哈希码”，因为这个概念出现在映射上下文中，而映射对象中的项由键和值构成，使用“哈希值”容易与键对应的“值”混淆。本书始终使用“哈希码”一说。

数值类型以及不可变的扁平类型 `str` 和 `bytes` 均是可哈希的。如果容器类型是不可变的，而且所含的对象全是可哈希的，那么容器类型自身也是可哈希的。`frozenset` 对象全部是可哈希的，因为按照定

义，每一个元素都必须是可哈希的。仅当所有项均可哈希，tuple 对象才是可哈希的。请考察以下示例中的 tt、tl 和 tf。

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```

一个对象的哈希码根据所用的 Python 版本和设备架构有所不同。如果出于安全考量而在哈希计算过程中**加盐**，那么哈希码也会发生变化。³ 正确实现的对象，其哈希码在一个 Python 进程内保持不变。

³一些安全隐患和应对方案参见“PEP 456—Secure and interchangeable hash algorithm”。

默认情况下，用户定义的类型是可哈希的，因为自定义类型的哈希码取自 id()，而且继承自 object 类的 __eq__() 方法只不过是比較对象 ID。如果自己实现了 __eq__() 方法，根据对象的内部状态进行比较，那么仅当 __hash__() 方法始终返回同一个哈希码时，对象才是可哈希的。实践中，这要求 __eq__() 和 __hash__() 只考虑在对象的生命周期内始终不变的实例属性。

接下来概览 Python 中最常用的几种映射类型的 API，包括 dict、defaultdict 和 OrderedDict。

3.4.2 常用映射方法概述

映射的基本 API 非常丰富。表 3-1 列出了 dict，以及 collections 模块中两个常用变体 defaultdict 和 OrderedDict 实现的方法。

表 3-1: 映射类型 dict、collections.defaultdict 和 collections.OrderedDict 实现的方法（简单起见，省略了 object 实现的方法），[...] 表示可选参数

	dict	defaultdict	OrderedDict	
d.clear()	•	•	•	删除所有项
d.__contains__(k)	•	•	•	k in d
d.copy()	•	•	•	浅拷贝
d.__copy__()		•		为 copy.copy(d) 提供支持
d.default_factory		•		由 __missing__ 调用的可调用对象，用于设置缺失的值 ^a
d.__delitem__(k)	•	•	•	del d[k]: 删除键 k 对应的项
d.fromkeys(it, [initial])	•	•	•	根据可迭代对象中的键构建一个映射，可选参数 initial 指定初始值（默认为 None）
d.get(k, [default])	•	•	•	获取键 k 对应的项，不存在时返回 default 或 None
d.__getitem__(k)	•	•	•	d[k]: 获取键 k 对应的项

	<code>dict</code>	<code>defaultdict</code>	<code>OrderedDict</code>	
<code>d.items()</code>	•	•	•	获取项视图, 即 <code>(key, value)</code> 对
<code>d.__iter__()</code>	•	•	•	获取遍历键的迭代器
<code>d.keys()</code>	•	•	•	获取键视图
<code>d.__len__()</code>	•	•	•	<code>len(d)</code> : 项数
<code>d.__missing__(k)</code>		•		当 <code>__getitem__</code> 找不到相应的键时调用
<code>d.move_to_end(k, [last])</code>			•	把 <code>k</code> 移到开头或结尾 (<code>last</code> 默认为 <code>True</code>)
<code>d.__or__(other)</code>	•	•	•	<code>d1 d2</code> : 合并 <code>d1</code> 和 <code>d2</code> , 新建一个 <code>dict</code> 对象 (Python 3.9 及以上版本)
<code>d.__ior__(other)</code>	•	•	•	<code>d1 = d2</code> : 使用 <code>d2</code> 中的项更新 <code>d1</code> (Python 3.9 及以上版本)
<code>d.pop(k, [default])</code>	•	•	•	删除并返回 <code>k</code> 对应的项, 如果没有键 <code>k</code> , 则返回 <code>default</code> 或 <code>None</code>

	<code>dict</code>	<code>defaultdict</code>	<code>OrderedDict</code>	
<code>d.popitem()</code>	•	•	•	删除并返回 (<code>(key, value)</code> 形式) 最后插入的项 ^b
<code>d.__reversed__()</code>	•	•	•	<code>reverse(d)</code> : 按插入顺序从后向前返回遍历键的迭代器
<code>d.__ror__(other)</code>	•	•	•	<code>other dd</code> : 反向合并运算符 (Python 3.9 及以上版本) ^c
<code>d.setdefault(k, [default])</code>	•	•	•	如果 <code>d</code> 中有键 <code>k</code> , 则返回 <code>d[k]</code> ; 否则, 把 <code>d[k]</code> 设为 <code>default</code> , 并返回 <code>default</code>
<code>d.__setitem__(k, v)</code>	•	•	•	<code>d[k] = v</code> : 把键 <code>k</code> 对应的值设为 <code>v</code>
<code>d.update(m, **kwargs)</code>	•	•	•	使用映射或可迭代对象中的键值对更新 <code>d</code>
<code>d.values()</code>	•	•	•	获取值视图

^a `default_factory` 不是方法, 而是可调用的属性, 由终端用户在实例化 `defaultdict` 对象时设置。

^b `OrderedDict.popitem(last=False)` 删除最先插入的项 (即先进先出)。截至 Python 3.10b3, `dict` 和 `defaultdict` 不支持关键字参数 `last`。

c 反向运算符详见第 16 章。

`d.update(m)` 处理第一个参数 `m` 的方式是“鸭子类型”的典型应用：先检查 `m` 有没有 `keys` 方法，如果有，则假定 `m` 是映射；否则，退而求其次，迭代 `m`，假定项是键值对形式 `((key, value))`。多数 Python 映射的构造函数，内部逻辑与 `update()` 方法一样，也就是说，可以从其他映射或生成键值对的可迭代对象初始化。

`setdefault()` 方法不容忽视。如果想就地更新一个项的值，使用该方法就不用再去查找键。3.4.3 节会说明具体用法。

3.4.3 插入或更新可变的值

根据 Python 的“快速失败”原则，当键 `k` 不存在时，`d[k]` 抛出错误。深谙 Python 的人知道，如果觉得默认值比抛出 `KeyError` 更好，那么可以把 `d[k]` 换成 `d.get(k, default)`。然而，如果你想更新得到的可变值，那么还有更好的方法。

举个例子。假设你想编写一个脚本分析文本，编制索引，生成一个映射，以词为键，值为一个列表，表示词出现的位置，如示例 3-3 所示。

示例 3-3 使用示例 3-4 处理《Python 之禅》得到的部分输出，每一行对应一个词及其出现的位置列表（第一个数是行号，第二个数是列号）

```
$ python3 index0.py zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
```

```
Beautiful [(3, 1)]
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11), (17,
8), (18, 25)]
...
```

示例 3-4 中的脚本不太完美，目的是告诉你 `dict.get` 不是处理缺失键的最佳方式。这个脚本根据 Alex Martelli 的一个示例修改而来。⁴

⁴原脚本出自 Martelli 的演讲“Re-learning Python”，第 41 张幻灯片。他的脚本其实是演示 `dict.setdefault`，如示例 3-5 所示。

示例 3-4 index0.py：使用 `dict.get` 获取并更新词出现的位置列表，编制索引（更好的方案见示例 3-5）

```
"""构建一个索引映射，列出词出现的位置"""

import re
import sys

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            # 这样写不完美，仅作演示
            occurrences = index.get(word, []) ❶
            occurrences.append(location)      ❷
            index[word] = occurrences        ❸

# 按字母表顺序显示
for word in sorted(index, key=str.upper): ❹
    print(word, index[word])
```

❶ 获取 `word` 出现的位置列表，如未找到，则返回 `[]`。

❷ 把新找到的位置追加到 `occurrences` 中。

❸ 把更新后的 occurrences 放入 index 字典。这里蕴涵一次对 index 的搜索操作。

❹ sorted 函数的 key 参数不是调用 str.upper 方法，而是传入那个方法的引用，供 sorted 函数排序各个词，规范化输出。⁵

⁵这里把方法当作一等对象使用，详见第 7 章。

示例 3-4 处理 occurrences 用了 3 行代码，换成 dict.setdefault 则只需一行代码。示例 3-5 与 Alex Martelli 最初写的代码类似。

示例 3-5 index.py: 使用 dict.setdefault 获取并更新词出现的位置列表，编制索引。与示例 3-4 相比，这部分只用了一行代码

```
"""构建一个索引映射，列出词出现的位置"""

import re
import sys

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location) ❶

# 按字母表顺序显示
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ 获取 word 出现的位置列表，如未找到，则设为 []; setdefault 返回该列表，可以直接更新，不用再搜索一次。

也就是说，下面这一行：

```
my_dict.setdefault(key, []).append(new_value)
```

作用与下面 3 行一样：

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

不过，后一种写法至少要搜索 `key` 两次，如未找到则搜索 3 次，而 `setdefault` 只搜索一次。

3.5 节将探讨一个相关话题：处理任何操作（而不仅限于插入）缺失键的情况。

3.5 自动处理缺失的键

有时搜索的键不一定存在，为了以防万一，可以人为设置一个值，以方便某些情况的处理。人为设置的值主要有两种方法：第一种是把普通的 `dict` 换成 `defaultdict`；第二种是定义 `dict` 或其他映射类型的子类，实现 `__missing__` 方法。下面分别介绍这两种情况。

3.5.1 defaultdict：处理缺失键的另一种选择

对于 `collections.defaultdict`，`d[k]` 句法找不到搜索的键时，使用指定的默认值创建对应的项。示例 3-6 使用 `defaultdict` 重新实现示例 3-5，以另一种优雅的方式编制索引。

这次实现的原理是，实例化 `defaultdict` 对象时提供一个可调用对象，当 `__getitem__` 遇到不存在的键时，调用那个可调用对象生成一个默认值。

举个例子。假设使用 `dd = defaultdict(list)` 创建一个 `defaultdict` 对象，而且 `dd` 中没有 `'new-key'` 键，那么 `dd['new-key']` 表达式按以下几步处理。

01. 调用 `list()` 创建一个新列表。
02. 把该列表插入 `dd`, 对应到 `'new-key'` 键上。
03. 返回该列表的引用。

生成默认值的可调用对象存放在实例属性 `default_factory` 中。

示例 3-6 index_default.py: 使用 `defaultdict` 代替 `setdefault` 方法

```
"""构建一个索引映射，列出词出现的位置"""

import collections
import re
import sys

WORD_RE = re.compile(r'\w+')

index = collections.defaultdict(list) ❶
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start() + 1
            location = (line_no, column_no)
            index[word].append(location) ❷

# 按字母表顺序显示
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ 创建一个 `defaultdict` 对象，把 `default_factory` 设为 `list` 构造函数。

❷ 如果 `index` 中当前没有 `word` 键，调用 `default_factory` 生成缺失的值。这里生成一个空列表，赋值给 `index[word]`，然后返回空列表。因此，`.append(location)` 操作始终成功。

如未提供 `default_factory`，遇到缺失的键时，则像往常一样抛出 `KeyError`。



`defaultdict` 的 `default_factory` 仅为 `__getitem__` 提供默认值，其他方法用不到。例如，`dd` 是一个 `defaultdict` 对象，如果没有键 `k`，那么 `dd[k]` 将调用 `default_factory` 创建默认值，但是 `dd.get(k)` 依然返回 `None`，而且 `k in dd` 也返回 `False`。

`defaultdict` 之所以调用 `default_factory`，背后的机制是接下来要讨论的特殊方法 `__missing__`。

3.5.2 `__missing__` 方法

映射处理缺失键的底层逻辑在 `__missing__` 方法中。`dict` 基类本身没有定义这个方法，但是如果 `dict` 的子类定义了这个方法，那么 `dict.__getitem__` 找不到键时将调用 `__missing__` 方法，不抛出 `KeyError`。

假设你希望查找映射的键时把键转换成 `str` 类型。IoT 设备的代码库就需要这么做。⁶ 可编程板（例如 Raspberry Pi 或 Arduino）上有一些通用 I/O 引脚，在代码库中通过 `Board` 类的 `my_board.pins` 属性表示。这个属性把物理引脚标识符映射到引脚对象上。物理引脚标识符可以是纯数值，也可以是 "A0" 或 "P9_12" 之类的字符串。为了保持一致，我们希望 `board.pins` 中的所有键都是字符串，可是有些时候也想通过数值来查找引脚，例如 `my_arduino.pin[13]`，这是为了免得初学者不知道如何点亮 Arduino 板 13 号引脚上的 LED。这种映射要求如示例 3-7 所示。

⁶例如开发已经停滞的 Pingo.io 库。

示例 3-7 搜索非字符串键时，`StrKeyDict0` 把未找到的键转换成字符串

测试使用 ``d[key]`` 表示法检索项::

```
>>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
>>> d['2']
'two'
>>> d[4]
'four'
>>> d[1]
Traceback (most recent call last):
...
KeyError: '1'
```

测试使用 `d.get(key)` 表示法检索项::

```
>>> d.get('2')
'two'
>>> d.get(4)
'four'
>>> d.get(1, 'N/A')
'N/A'
```

测试 ``in`` 运算符::

```
>>> 2 in d
True
>>> 1 in d
False
```

示例 3-8 实现了可通过以上 doctest 的 `StrKeyDict0` 类。



用户自定义映射类型最好像示例 3-9 那样继承 `collections.UserDict` 类, 而不要继承 `dict` 类。这里继承 `dict` 类是为了说明内置的 `dict.__getitem__` 方法支持 `__missing__`。

示例 3-8 `StrKeyDict0` 在查找键时把非字符串键转换成字符串 (见示例 3-7 中的测试)

```
class StrKeyDict0(dict): ❶

    def __missing__(self, key):
        if isinstance(key, str): ❷
            raise KeyError(key)
        return self[str(key)] ❸
```

```
def get(self, key, default=None):
    try:
        return self[key] ❷
    except KeyError:
        return default ❸

def __contains__(self, key):
    return key in self.keys() or str(key) in self.keys() ❹
```

❶ StrKeyDict0 继承 dict。

❷ 检查 key 是不是 str 类型。如果是，并且找不到这个键，则抛出 KeyError。

❸ 把 key 转换成 str 类型，查找字符串键。

❹ get 方法委托 __getitem__，通过 self[key] 表示法查找键，让 __missing__ 发挥作用。

❺ 如果抛出 KeyError，说明 __missing__ 也找不到键，那就返回 default。

❻ 先搜索未经修改的键（实例可能有非字符串键），再搜索键的字符串形式。

请想一想，为什么实现 __missing__ 方法时要做 isinstance(key, str) 测试。

如果不做这个测试，则 __missing__ 方法可以处理任何类型的键，是不是字符串都无所谓，但前提是 str(k) 得到的是现有键。然而，倘若 str(k) 得到的键不存在，那就进入了无限递归：

__missing__ 方法的最后一行 self[str(key)] 将调用 __getitem__ 方法，转而又调用 __missing__ 方法。

在本例中，出于对行为一致性的考虑，也需要 __contains__ 方法，因为 k in d 操作调用它，而从 dict 继承的 __contains__

方法不回落到 `__missing__` 方法。`__contains__` 方法的实现方式有一个微妙的细节，我们没有按照常规的 Python 风格检查键 (`k in my_dict`)，因为 `str(key) in self` 将递归调用 `__contains__` 方法。为了避免这个问题，我们显式地在 `self.keys()` 中查找键。

在 Python 3 中，即便是特别大型的映射，`k in my_dict.keys()` 这样的搜索效率也不低，因为 `dict.keys()` 返回的是类似集合的视图，详见 3.12 节。尽管如此，`k in my_dict` 的作用是相同的，而且速度更快，因为无须通过属性查找 `.keys` 方法。

示例 3-8 中，`__contains__` 方法使用 `self.keys()` 还有一个特定的原因。为了保证结果正确，必须使用 `key in self.keys()` 检查未经修改的键，因为 `StrKeyDict0` 不强制要求字典中的所有键都是 `str` 类型。这个简单的示例只有一个目标，即提高搜索的“友好度”，而不是限定键的类型。



用户自己定义的类，如果继承标准库中的映射，在实现 `__getitem__`、`get` 或 `__contains__` 方法时不一定要回落到 `__missing__` 方法，具体原因见 3.5.3 节。

3.5.3 标准库对 `__missing__` 方法的使用不一致

下面分几种情况演示查找缺失键的不同行为。

`dict` 子类

定义一个 `dict` 子类，只实现 `__missing__` 方法，其他方法均不实现。这种情况下，可能只有 `d[k]`（使用继承自 `dict` 的 `__getitem__` 方法）会调用 `__missing__` 方法。

`collections.UserDict` 子类

定义一个 `UserDict` 子类，同样只实现 `__missing__` 方法，其他方法均不实现。继承自 `UserDict` 的 `get` 方法调用 `__getitem__`。这意味着，`d[k]` 和 `d.get(k)` 在查找键时可能会调用 `__missing__` 方法。

`abc.Mapping` 子类，以最简单的方式实现 `__getitem__` 方法

定义一个精简的 `abc.Mapping` 子类，实现 `__missing__` 方法和必要的抽象方法，其中 `__getitem__` 方法不调用 `__missing__`。这个类永不触发 `__missing__` 方法。

`abc.Mapping` 子类，让 `__getitem__` 调用 `__missing__`

定义一个精简的 `abc.Mapping` 子类，实现 `__missing__` 方法和必要的抽象方法，其中 `__getitem__` 方法调用 `__missing__`。`d[k]`、`d.get(k)` 和 `k in d` 遇到缺失的键将触发 `__missing__` 方法。

这几种情况的代码见示例代码中的 `missing.py` 文件。

以上 4 种情况的具体实现都做了简化处理。你在自己定义的子类中实现 `__getitem__`、`get` 和 `__contains__` 方法时，可以根据实际需求决定是否使用 `__missing__` 方法。本节的目的是指出，继承标准库中的映射时要小心谨慎，不同的基类对 `__missing__` 的使用方式不一样。

不要忘了，`setdefault` 和 `update` 的行为也受键查找的影响。最后，`__missing__` 的逻辑容易出错，有时还要以特殊的逻辑实现 `__setitem__`，以免行为不一致或出乎意料。3.6.5 节有一个示例。

目前我们讨论了 `dict` 和 `defaultdict` 两种映射类型，除此之外，标准库中还有其他映射类型，下面将逐一讨论。

3.6 dict 的变体

本节概述标准库中除 `defaultdict` (3.5.1 节已经介绍) 之外的映射类型。

3.6.1 `collections.OrderedDict`

自 Python 3.6 起, 内置的 `dict` 也保留键的顺序。使用 `OrderedDict` 最主要的原因是编写与早期 Python 版本兼容的代码。不过, `dict` 和 `OrderedDict` 之间还有一些差异, Python 文档中有说明, 摘录如下 (根据日常使用频率, 顺序有调整)。

- `OrderedDict` 的等值检查考虑顺序。
- `OrderedDict` 的 `popitem()` 方法签名不同, 可通过一个可选参数指定移除哪一项。
- `OrderedDict` 多了一个 `move_to_end()` 方法, 便于把元素的位置移到某一端。
- 常规的 `dict` 主要用于执行映射操作, 插入顺序是次要的。
- `OrderedDict` 的目的是方便执行重新排序操作, 空间利用率、迭代速度和更新操作的性能是次要的。
- 从算法上看, `OrderedDict` 处理频繁重新排序操作的效果比 `dict` 好, 因此适合用于跟踪近期存取情况 (例如在 LRU 缓存中)。

3.6.2 `collections.ChainMap`

`ChainMap` 实例存放一组映射, 可作为一个整体来搜索。查找操作按照输入映射在构造函数调用中出现的顺序执行, 一旦在某个映射中找到指定的键, 旋即结束。例如:

```
>>> d1 = dict(a=1, b=3)
>>> d2 = dict(a=2, b=4, c=6)
>>> from collections import ChainMap
```

```
>>> chain = ChainMap(d1, d2)
>>> chain['a']
1
>>> chain['c']
6
```

ChainMap 实例不复制输入映射，而是存放映射的引用。ChainMap 的更新或插入操作只影响第一个输入映射。续前例：

```
>>> chain['c'] = -1
>>> d1
{'a': 1, 'b': 3, 'c': -1}
>>> d2
{'a': 2, 'b': 4, 'c': 6}
```

ChainMap 可用于实现支持嵌套作用域的语言解释器，按嵌套层级从内到外，一个映射表示一个作用域上下文。collections 文档中的“ChainMap objects”一节举了几个 ChainMap 用法示例，其中一个就是模仿 Python 查找变量的基本规则，如下所示。

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

第 18 章中的示例 18-14 使用一个 ChainMap 子类为 Scheme 编程语言的子集实现解释器。

3.6.3 collections.Counter

这是一种对键计数的映射。更新现有的键，计数随之增加。可用于统计可哈希对象的实例数量，或者作为多重集（multiset，本节后文讨论）使用。Counter 实现了组合计数的 + 和 - 运算符，以及其他一些有用的方法，例如 `most_common([n])`。该方法返回一个有序元组列表，对应前 n 个计数值最大的项及其数量。下面使用 Counter 统计词中的字母数量。

```
>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
```



```
>>> ct.update('aaaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(3)
[('a', 10), ('z', 3), ('b', 2)]
```

注意，'b' 和 'r' 两个键并列第三，但是 `ct.most_common(3)` 只显示 3 项。

若想把 `collections.Counter` 当作多重集使用，假设各个键是集合中的元素，计数值则是元素在集合中出现的次数。

3.6.4 `shelve.Shelf`

标准库中的 `shelve` 模块持久存储字符串键与（以 `pickle` 二进制格式序列化的）Python 对象之间的映射。你可能觉得 `shelve` 这个名称有点奇怪，不过想到泡菜坛（pickle jar）是放在货架（shelve）上的，还是有一定道理的。

模块级函数 `shelve.open` 返回一个 `shelve.Shelf` 实例，这是一个简单的键值 DBM 数据库，背后是 `dbm` 模块。`shelve.Shelf` 具有以下特征。

- `shelve.Shelf` 是 `abc.MutableMapping` 的子类，提供了我们预期的映射类型基本方法。
- 此外，`shelve.Shelf` 还提供了一些其他 I/O 管理方法，例如 `sync` 和 `close`。
- `Shelf` 实例是上下文管理器，因此可以使用 `with` 块确保在使用后关闭。
- 为键分配新值后即保存键和值。
- 键必须是字符串。
- 值必须是 `pickle` 模块可以序列化的对象。

详细说明和一些注意事项见 `shelve` 模块、`dbm` 模块和 `pickle` 模块的文档。



在极为简单的情况下，Python 的 `pickle` 模块用着比较顺手，但是隐藏的陷阱也不少。使用 `pickle` 解决问题之前，请务必阅读 Ned Batchelder 写的“Pickle's nine flaws”一文。Ned 在这篇文章中给出了可供选择的其它序列化格式。

`OrderedDict`、`ChainMap`、`Counter` 和 `Shelf` 本身就可使用，此外也可以通过子类定制。相反，`UserDict` 只应作为基类，在此基础上扩展。

3.6.5 子类应继承 `UserDict` 而不是 `dict`

创建新的映射类型，最好扩展 `collections.UserDict`，而不是 `dict`。为了确保以 `str` 类型存储添加到映射中的键，我们在示例 3-8 中定义了 `StrKeyDict0` 类。那时我们就意识到了这一点。

子类最好继承 `UserDict` 的主要原因是，内置的 `dict` 在实现上走了一些捷径，如果继承 `dict`，那就不得不覆盖一些方法，而继承 `UserDict` 则没有这些问题。⁷

⁷子类继承 `dict` 和其他内置类型的具体问题将在 14.3 节讨论。

注意，`UserDict` 没有继承 `dict`，使用的是组合模式：内部有一个 `dict` 实例，名为 `data`，存放具体的项。与示例 3-8 相比，这样做可以避免 `__setitem__` 等特殊方法意外递归，还能简化 `__contains__` 的实现。

示例 3-9 中的 `StrKeyDict` 继承 `UserDict`，实现过程比 `StrKeyDict0`（示例 3-8）更简洁，而且功能更丰富：所有键都以 `str` 类型存储，使用包含非字符串键的数据构建或更新实例不会发生意外情况。

示例 3-9 StrKeyDict 在插入、更新和查找时，始终把非字符串键转换成 str 类型

```
import collections

class StrKeyDict(collections.UserDict): ❶

    def __missing__(self, key): ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]

    def __contains__(self, key):
        return str(key) in self.data ❸

    def __setitem__(self, key, item):
        self.data[str(key)] = item ❹
```

❶ StrKeyDict 扩展 UserDict。

❷ __missing__ 与示例 3-8 完全相同。

❸ __contains__ 更简单，因为存储的所有键都可假定为 str 类型，不用像 StrKeyDict0 那样调用 self.keys()，检查 self.data 即可。

❹ __setitem__ 把 key 转换成 str 类型。委托给 self.data 属性之后，这个方法更易于重写。

由于 UserDict 扩展 abc.MutableMapping，因此使 StrKeyDict 成为一种功能完整的映射的方法，是从 UserDict、MutableMapping 或 Mapping 继承的方法。后两个类虽然是抽象基类，但是也有一些有用的具体方法。下面两个方法值得关注。

MutableMapping.update

这个强大的方法可以直接调用，__init__ 也使用它从其他映射、键值对可迭代对象和关键字参数中加载实例。该方法使用

`self[key] = value` 句法添加项，最终会调用子类实现的 `__setitem__`。

`Mapping.get`

在 `StrKeyDict0` 中（参见示例 3-8），我们必须自己实现 `get` 方法，返回与 `__getitem__` 一样的结果。而在示例 3-9 中，我们继承了 `Mapping.get`，它的实现与 `StrKeyDict0.get` 完全相同。



Antoine Pitrou 编写的“PEP 455—Adding a key-transforming dictionary to collections”提议为 `collections` 模块增加 `TransformDict`，这比 `StrKeyDict` 更通用，在转换之前保留键的原本类型。PEP 455 在 2015 年 5 月被否决，拒绝原因见 Raymond Hettinger 的回应。为了试验 `TransformDict`，我把 Pitrou 的补丁从 18986 号工单中提取出来了，制成了独立的模块（03-dict-set/transformdict.py）。

我们知道 Python 有不可变序列类型，那有没有不可变映射呢？其实，标准库中没有，不过有变通方案。详见 3.7 节。

3.7 不可变映射

标准库提供的映射类型都是可变的，不过有时也需要防止用户意外更改映射。3.5.2 节提到的硬件编程库 Pingo 就有一例：`board.pins` 映射表示设备上的 GPIO 物理引脚，因此需要防止被人无意中更新，毕竟软件不能更改硬件，不然就与设备的实际物理结构不一致了。

`types` 模块提供的 `MappingProxyType` 是一个包装类，把传入的映射包装成一个 `mappingproxy` 实例，这是原映射的动态代理，只可读取。这意味着，对原映射的更新将体现在 `mappingproxy` 实例身上，但是不能通过 `mappingproxy` 实例更改映射。示例 3-10 简单演示 `MappingProxyType` 的用途。

示例 3-10 MappingProxyType 根据 dict 对象构建只读的 mappingproxy 实例

```
>>> from types import MappingProxyType
>>> d = {1: 'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1] ❶
'A'
>>> d_proxy[2] = 'x' ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy ❸
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

- ❶ d_proxy 可以访问 d 中的项。
- ❷ 不能通过 d_proxy 更改映射。
- ❸ d_proxy 是动态的，能够反映 d 的变化。

在硬件编程场景中可以这样使用：定义 Board 的具体子类，在构造方法中声明一个私有映射，存放引脚对象，再使用 mappingproxy 实现一个公开的 .pin 属性，通过 API 开放给客户端使用。这样，客户端就不会意外地添加、删除或更改引脚。

接下来探讨视图。通过视图可对字典执行一些高性能操作，免去了复制数据的麻烦。

3.8 字典视图

`dict` 的实例方法 `.keys()`、`.values()` 和 `.items()` 分别返回 `dict_keys`、`dict_values` 和 `dict_items` 类的实例。这些字典视图是 `dict` 内部实现使用的数据结构的只读投影。Python 2 中对应的方法返回列表，重复 `dict` 中已有的数据，有一定的内存开销。另外，视图还取代了返回迭代器的旧方法。

示例 3-11 展示所有字典视图均支持的一些基本操作。

示例 3-11 `.values()` 方法返回 `dict` 对象的值视图

```
>>> d = dict(a=10, b=20, c=30)
>>> values = d.values()
>>> values
dict_values([10, 20, 30]) ❶
>>> len(values) ❷
3
>>> list(values) ❸
[10, 20, 30]
>>> reversed(values) ❹
<dict_reversevalueiterator object at 0x10e9e7310>
>>> values[0] ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_values' object is not subscriptable
```

- ❶ 通过视图对象的字符串表示形式查看视图的内容。
- ❷ 可以查询视图的长度。
- ❸ 视图是可迭代对象，方便构建列表。
- ❹ 视图实现了 `__reversed__` 方法，返回一个自定义迭代器。
- ❺ 不能使用 `[]` 获取视图中的项。

视图对象是动态代理。更新原 `dict` 对象后，现有视图立即就能看到变化。续示例 3-11：

```
>>> d['z'] = 99
>>> d
{'a': 10, 'b': 20, 'c': 30, 'z': 99}
>>> values
dict_values([10, 20, 30, 99])
```

`dict_keys`、`dict_values` 和 `dict_items` 是内部类，不能通过 `__builtins__` 或标准库中的任何模块获取，尽管可以得到实例，但是在 Python 代码中不能自己动手创建。

```
>>> values_class = type({}.values())
>>> v = values_class()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'dict_values' instances
```

`dict_values` 类是最简单的字典视图，只实现了 `__len__`、`__iter__` 和 `__reversed__` 这 3 个特殊方法。除此之外，`dict_keys` 和 `dict_items` 还实现了多个集合方法，基本与 `frozenset` 类相当。讲解集合之后再深入探讨 `dict_keys` 和 `dict_items`，详见 3.12 节。

接下来讲一讲 `dict` 的内部实现产生的一些制约，并给出一些实践小窍门。

3.9 dict 的实现方式对实践的影响

Python 使用哈希表实现 `dict`，因此字典的效率非常高，不过这种设计对实践也有一些影响，不容忽视。

- 键必须是可哈希的对象。就像 3.4.1 节所说的那样，必须正确实现 `__hash__` 和 `__eq__` 方法。
- 通过键访问项速度非常快。对于一个包含数百万个键的 `dict` 对象，Python 通过计算键的哈希码就可以直接定位键，然后找出索

引在哈希表中的偏移量，稍微尝试几次就能找到匹配的条目，因此开销不大。

- 在 CPython 3.6 中，`dict` 的内存布局更为紧凑，顺带的一个副作用是键的顺序得以保留。Python 3.7 正式支持保留顺序。
- 尽管采用了新的紧凑布局，但是字典仍然占用大量内存，这是不可避免的。对容器来说，最紧凑的内部数据结构是指向项的指针的数组。⁸ 与之相比，哈希表中的条目存储的数据更多，而且为了保证效率，Python 至少需要把哈希表中三分之一的行留空。
- 为了节省内存，不要在 `__init__` 方法之外创建实例属性。

⁸元组就是这样存储的。

最后一点背后的原因是，Python 默认在特殊的 `__dict__` 属性中存储实例属性，而这个属性的值是一个字典，依附在各个实例上。⁹ 自从 Python 3.3 实现“PEP 412—Key-Sharing Dictionary”之后，类的实例可以共用一个哈希表，随类一起存储。如果新实例与 `__init__` 返回的第一个实例拥有相同的属性名称，那么新实例的 `__dict__` 属性就共享这个哈希表，仅以指针数组的形式存储新实例的属性值。

`__init__` 方法执行完毕后再添加实例属性，Python 就不得不为这个实例的 `__dict__` 属性创建一个新哈希表（在 Python 3.3 之前，这是所有实例的默认行为）。根据 PEP 412，这种优化可将面向对象程序的内存使用量减少 10%~20%。

⁹除非类有 `__slots__` 属性，详见 11.11 节。

下面开始研究集合。

3.10 集合论

集合对 Python 来说并不是新鲜事物，但目前尚未得到充分利用。`set` 类型及其不可变形式 `frozenset` 首先作为模块出现在 Python 2.3 标准库中，随后在 Python 2.6 中被提升为内置类型。



本书使用“集合”一词指代 `set` 和 `frozenset`。如果讨论内容仅涉及 `set` 类，则使用等宽字体，写作 `set`。

集合是一组唯一的对象。集合的基本作用是去除重复项。

```
>>> l = ['spam', 'spam', 'eggs', 'spam', 'bacon', 'eggs']
>>> set(l)
{'eggs', 'spam', 'bacon'}
>>> list(set(l))
['eggs', 'spam', 'bacon']
```



如果想去除重复项，同时保留每一项首次出现位置的顺序，那么现在使用普通的 `dict` 即可，如下所示。

```
>>> dict.fromkeys(l).keys()
dict_keys(['spam', 'eggs', 'bacon'])
>>> list(dict.fromkeys(l).keys())
['spam', 'eggs', 'bacon']
```

集合元素必须是可哈希的对象。`set` 类型不可哈希，因此不能构建嵌套 `set` 实例的 `set` 对象。但是 `frozenset` 可以哈希，所以 `set` 对象可以包含 `frozenset` 元素。

除了强制唯一性之外，集合类型通过中缀运算符实现了许多集合运算。给定两个集合 `a` 和 `b`，`a | b` 计算并集，`a & b` 计算交集，`a - b` 计算差集，`a ^ b` 计算对称差集。巧妙使用集合运算既可以减少代码行数，也能缩减 Python 程序的运行时间，同时还能少编写一些循环和条件逻辑，从而让代码更易于阅读和理解。

举个例子，假设有一个集合存储大量电子邮件地址（`haystack`），还有一个集合存储少量电子邮件地址（`needles`），你想统计 `needles` 中有多少邮件地址出现在 `haystack` 中。借助集合交集运算（`&` 运算符），用一行代码即可实现（见示例 3-12）。

示例 3-12 统计 `needles` 在 `haystack` 中出现的次数（二者均为集合类型）

```
found = len(needles & haystack)
```

如果不使用交集运算符，则要像示例 3-13 那样编写代码，才能实现与示例 3-12 一样的效果。

示例 3-13 统计 `needles` 在 `haystack` 中出现的次数（效果与示例 3-12 一样）

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

示例 3-12 的运行速度比示例 3-13 稍快。不过，示例 3-13 处理的 `needles` 和 `haystack` 可以是任何可迭代对象，而示例 3-12 要求二者均为集合。然而，就算一开始不是集合，也可以快速构建，如示例 3-14 所示。

示例 3-14 统计 `needles` 在 `haystack` 中出现的次数，支持任何可迭代类型

```
found = len(set(needles) & set(haystack))

# 另一种方式
found = len(set(needles).intersection(haystack))
```

当然，像示例 3-14 那样构建集合有一定的额外开销。但是，如果 `needles` 和 `haystack` 中有一个本身就是集合，那么示例 3-14 中的第二种方式比示例 3-13 开销更小。

假如 `needles` 中有 1000 个元素，`haystack` 中有 10 000 000 项，那么前面几例的运行时间都在 0.3 毫秒左右，平摊到每个元素上约为 0.3 微秒。

除了成员测试速度极快（归功于底层哈希表）之外，内置类型 `set` 和 `frozenset` 还提供了丰富的 API，有的用于创建新集合，有的用于更改 `set` 对象。在讨论这些操作之前，先讲一讲句法。

3.10.1 `set` 字面量

`set` 字面量的句法与集合的数学表示法几乎一样，例如 `{1}`、`{1, 2}` 等。唯有一点例外：空 `set` 没有字面量表示法，必须写作 `set()`。



句法陷阱

创建空 `set`，务必使用不带参数的构造函数，即 `set()`。倘若写成 `{}`，则创建的是空 `dict`——这一点在 Python 3 中没有变化。

在 Python 3 中，集合的标准字符串表示形式始终使用 `{...}` 表示法，唯有空集例外。

```
>>> s = {1}
>>> type(s)
<class 'set'>
>>> s
{1}
>>> s.pop()
1
>>> s
set()
```

与调用构造函数（例如 `set([1, 2, 3])`）相比，使用 `set` 的字面量句法（例如 `{1, 2, 3}`）不仅速度快，而且更具可读性。调用构造函数速度慢的原因是，Python 要查找 `set` 名称，找出构造函数，然后构建一个列表，最后再把列表传给构造函数。相比之下，Python 处理字面量只需要运行一个专门的 `BUILD_SET` 字节码。¹⁰

¹⁰这可能很有趣，但不是特别重要。只有求解一个集合字面量时才有加速效果，而且每个 Python 进程最多发生一次，即首次编译模块时。如果你好奇，可以从 `dis` 模块中导入 `dis` 函数，反汇编 `set` 字面量（例如 `dis('{1}')`）和 `set` 调用（例如 `dis('set([1])')`）的字节码。

`frozenset` 没有字面量句法，必须调用构造函数创建。在 Python 3 中，`frozenset` 的字符串表示形式类似于构造函数调用。下面是控制台会话的输出。

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

说到句法，列表推导式的思想也可用于构建集合。

3.10.2 集合推导式

集合推导式早在 Python 2.7 就已出现，与 3.2.1 节讲到的字典推导式同时引入。示例 3-15 演示集合推导式的用法。

示例 3-15 构建一个集合，元素为 Unicode 名称中带有“SIGN”一词的 Latin-1 字符

```
>>> from unicodedata import name ❶
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')}
❷
{'$', '=', 'ç', '#', 'α', '<', '¥', 'μ', '×', '$', 'ℓ', '£', '©',
'°', '+', '÷', '±', '>', '¬', '®', '%'}
```

❶ 从 `unicodedata` 中导入 `name` 函数，获取字符名称。

❷ 将代码在 32~255 的范围内，而且名称中带有 `'SIGN'` 一词的字符放入集合。

不同的 Python 进程得到的输出顺序不一样，原因与 3.4.1 节提到的加盐哈希有关。

句法讲完了，接下来讨论集合的行为。

3.11 集合的实现方式对实践的影响

`set` 和 `frozenset` 类型都使用哈希表实现。这种设计带来了以下影响。

- 集合元素必须是可哈希对象，必须像 3.4.1 节所说的那样正确实现 `__hash__` 和 `__eq__` 方法。
- 成员测试效率非常高。对于一个包含数百万个元素的集合，计算元素的哈希码就可以直接定位元素，找出元素的索引偏移量。稍微搜索几次就能找到匹配的元素，即使穷尽搜索开销也不大。
- 与存放元素指针的低层数组相比，集合占用大量内存。尽管集合的结构更紧凑，但是一旦要搜索的元素数量变多，搜索速度将显著下降。
- 元素的顺序取决于插入顺序，但是顺序对集合没有什么意义，也得不到保障。如果两个元素具有相同的哈希码，则顺序取决于哪个元素先被添加到集合中。
- 向集合中添加元素后，现有元素的顺序可能发生变化。这是因为哈希表使用率超过三分之二后，算法效率会有所下降，Python 可能需要移动和调整哈希表的大小，然后重新插入元素，导致元素的相对顺序发生变化。

接下来讲解丰富的集合运算。

集合运算

图 3-2 概括了可变和不可变集合可以使用的方法。其中很多方法是重载运算符（例如 `&` 和 `>=`）的特殊方法。表 3-2 列出数学上的集合运算符和对应的 Python 运算符或方法。注意，一些运算符和方法就地更改目标集合，例如 `&=`、`difference_update` 等。这样的运算对理想世界中的数学集合没有意义，而且 `frozenset` 没有实现。



表 3-2 列出的中缀运算符要求两个操作数均为集合，其他方法则接受一个或多个可迭代对象参数。比如说，为了计算 4 个容器 *a*、*b*、*c* 和 *d* 的并集，可以调用 `a.union(b, c, d)`，其中 *a* 必须是集合，而 *b*、*c* 和 *d* 可以是任何类型的可迭代对象，只要项是可哈希的对象即可。Python 3.5 实现“PEP 448—Additional Unpacking Generalizations”之后，如果你想使用 4 个可迭代对象合并后的结果创建一个集合，那么无须在现有的集合之上更新，就可以使用 `{*a, *b, *c, *d}` 句法。

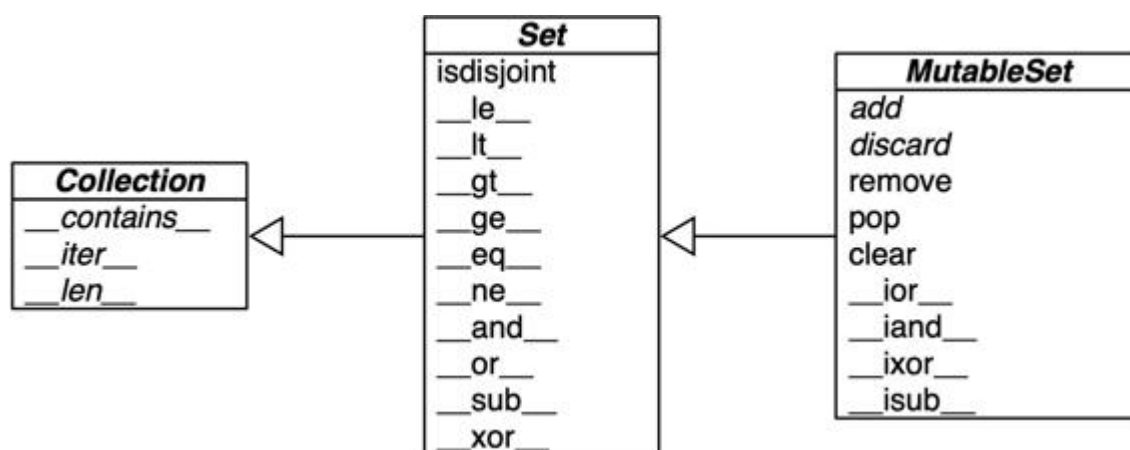


图 3-2: `MutableSet` 及其在 `collections.abc` 中的超类的简化 UML 类图（以斜体显示的名称是抽象类和抽象方法；简单起见，省略了反向运算符方法）

表 3-2: 集合数学运算（要么生成一个新集合，要么就地更新可变集合）

数学符号	Python 运算符	方法	说明
$S \cap Z$	<code>s & z</code>	<code>s.__and__(z)</code>	<i>s</i> 和 <i>z</i> 的交集

数学符号	Python 运算符	方法	说明
	$z \ \& \ s$	<code>s.__rand__(z)</code>	反向 $\&$ 运算符
		<code>s.intersection(it, ...)</code>	s 和根据可迭代对象 it 等构建的集合的交集
	$s \ \&= \ z$	<code>s.__iand__(z)</code>	使用 s 和 z 的交集更新 s
		<code>s.intersection_update(it, ...)</code>	使用 s 和根据可迭代对象 it 等构建的集合的交集更新 s
$s \cup z$	$s \ \ z$	<code>s.__or__(z)</code>	s 和 z 的并集
	$z \ \ s$	<code>s.__ror__(z)</code>	反向 $ $ 运算符
		<code>s.union(it, ...)</code>	s 和根据可迭代对象 it 等构建的集合的并集
	$s \ = \ z$	<code>s.__ior__(z)</code>	使用 s 和 z 的并集更新 s
		<code>s.update(it, ...)</code>	使用 s 和根据可迭代对象 it 等构建的集合的并集更新 s
$s \setminus z$	$s \ - \ z$	<code>s.__sub__(z)</code>	s 和 z 的相对补集（或差集）

数学符号	Python 运算符	方法	说明
	$z - s$	<code>s.__rsub__(z)</code>	反向 - 运算符
		<code>s.difference(it, ...)</code>	<code>s</code> 和根据可迭代对象 <code>it</code> 等构建的集合的差集
	$s -= z$	<code>s.__isub__(z)</code>	使用 <code>s</code> 和 <code>z</code> 的差集更新 <code>s</code>
		<code>s.difference_update(it, ...)</code>	使用 <code>s</code> 和根据可迭代对象 <code>it</code> 等构建的集合的差集更新 <code>s</code>
$S \Delta Z$	$s \wedge z$	<code>s.__xor__(z)</code>	对称差集 (<code>s</code> & <code>z</code> 的补集)
	$z \wedge s$	<code>s.__rxor__(z)</code>	反向 $\wedge$ 运算符
		<code>s.symmetric_difference(it)</code>	<code>s</code> & <code>set(it)</code> 的补集
	$s \wedge= z$	<code>s.__ixor__(z)</code>	使用 <code>s</code> 和 <code>z</code> 的对称差集更新 <code>s</code>
		<code>s.symmetric_difference_update(it, ...)</code>	使用 <code>s</code> 和根据可迭代对象 <code>it</code> 等构建的集合的对称差集更新 <code>s</code>

表 3-3 列出了集合谓词，即返回 `True` 或 `False` 的运算符和方法。

表 3-3：返回布尔值的集合比较运算符和方法

数学符号	Python 运算符	方法	说明
$S \cap Z = \emptyset$		<code>s.isdisjoint(z)</code>	s 和 z 不相交（没有共同元素）
$e \in S$	<code>e in s</code>	<code>s.__contains__(e)</code>	元素 e 是 s 的成员
$S \subseteq Z$	<code>s <= z</code>	<code>s.__le__(z)</code>	s 是 z 的子集
		<code>s.issubset(it)</code>	s 是由可迭代对象 it 构建的集合的子集
$S \subset Z$	<code>s < z</code>	<code>s.__lt__(z)</code>	s 是 z 的真子集
$S \supseteq Z$	<code>s >= z</code>	<code>s.__ge__(z)</code>	s 是 z 的超集
		<code>s.issuperset(it)</code>	s 是由可迭代对象 it 构建的集合的超集
$S \supset Z$	<code>s > z</code>	<code>s.__gt__(z)</code>	s 是 z 的真超集

除了源自数学集合论的运算符和方法之外，集合类型还实现了一些其他实用方法，如表 3-4 所示。

表 3-4：集合的其他方法

	<code>set</code>	<code>frozenset</code>	

	set	frozenset	
<code>s.add(e)</code>	•		把元素 <code>e</code> 添加到 <code>s</code> 中
<code>s.clear()</code>	•		删除 <code>s</code> 中的全部元素
<code>s.copy()</code>	•	•	浅拷贝 <code>s</code>
<code>s.discard(e)</code>	•		从 <code>s</code> 中删除元素 <code>e</code> (如果存在 <code>e</code>)
<code>s.__iter__()</code>	•	•	获取遍历 <code>s</code> 的迭代器
<code>s.__len__()</code>	•	•	<code>len(s)</code>
<code>s.pop()</code>	•		从 <code>s</code> 中删除并返回一个元素, 如果 <code>s</code> 为空, 则抛出 <code>KeyError</code>
<code>s.remove(e)</code>	•		从 <code>s</code> 中删除元素 <code>e</code> , 如果 <code>e</code> 不在 <code>s</code> 中, 则抛出 <code>KeyError</code>

对集合功能的概述到此结束。3.12 节将兑现 3.8 节的承诺, 探讨与 `frozenset` 行为非常相似的两种字典视图。

3.12 字典视图的集合运算

`.keys()` 和 `.items()` 这两个 `dict` 方法返回的视图对象与 `frozenset` 极为相似, 如表 3-5 所示。

表 3-5: `frozenset`、`dict_keys` 和 `dict_items` 实现的方法

	frozenset	dict_keys	dict_items	说明
<code>s.__and__(z)</code>	•	•	•	$s \& z$ (s 和 z 的交集)
<code>s.__rand__(z)</code>	•	•	•	反向 $\&$ 运算符
<code>s.__contains__()</code>	•	•	•	$e \text{ in } s$
<code>s.copy()</code>	•			浅拷贝 s
<code>s.difference(it, ...)</code>	•			s 和可迭代对象 it 等的差集
<code>s.intersection(it, ...)</code>	•			s 和可迭代对象 it 等的交集
<code>s.isdisjoint(z)</code>	•	•	•	s 和 z 不相交 (没有共同元素)
<code>s.issubset(it)</code>	•			s 是可迭代对象 it 的子集
<code>s.issuperset(it)</code>	•			s 是可迭代对象 it 的超集
<code>s.__iter__()</code>	•	•	•	获取遍历 s 的迭代器

	frozenset	dict_keys	dict_items	说明
<code>s.__len__()</code>	•	•	•	<code>len(s)</code>
<code>s.__or__(z)</code>	•	•	•	$s \mid z$ (s 和 z 的并集)
<code>s.__ror__()</code>	•	•	•	反向 $\mid$ 运算符
<code>s.__reversed__()</code>		•	•	获取逆序遍历 s 的迭代器
<code>s.__rsub__(z)</code>	•	•	•	反向 $-$ 运算符
<code>s.__sub__(z)</code>	•	•	•	$s - z$ (s 和 z 的差集)
<code>s.symmetric_difference(it)</code>	•			$s \oplus \text{set}(it)$ 的补集
<code>s.union(it, ...)</code>	•			s 和可迭代对象 it 等的并集
<code>s.__xor__()</code>	•	•	•	$s \wedge z$ (s 和 z 的对称差集)
<code>s.__rxor__()</code>	•	•	•	反向 $\wedge$ 运算符

需要特别注意的是，`dict_keys` 和 `dict_items` 实现了一些特殊方法，支持强大的集合运算符，包括 `&`（交集）、`|`（并集）、`-`（差集）和 `^`（对称差集）。

例如，使用 `&` 运算符可以轻易获取两个字典都有的键。

```
>>> d1 = dict(a=1, b=2, c=3, d=4)
>>> d2 = dict(b=20, d=40, e=50)
>>> d1.keys() & d2.keys()
{'b', 'd'}
```

注意，`&` 运算符返回一个 `set`。更方便的是，字典视图的集合运算符均兼容 `set` 实例，如下所示。

```
>>> s = {'a', 'e', 'i'}
>>> d1.keys() & s
{'a'}
>>> d1.keys() | s
{'a', 'c', 'b', 'd', 'i', 'e'}
```



仅当 `dict` 中的所有值均可哈希时，`dict_items` 视图才可当作集合使用。倘若 `dict` 中有不可哈希的值，对 `dict_items` 视图做集合运算将抛出 `TypeError: unhashable type 'T'`，其中 `T` 是不可哈希的类型。

相反，`dict_keys` 视图始终可当作集合使用，因为按照其设计，所有键均可哈希。

使用集合运算符处理视图可以省去大量循环和条件判断。繁重工作都可交给 C 语言实现的 Python 高效完成。

本章到此告一段落。

3.13 本章小结

字典是 Python 的基石。近些年，我们熟悉的字面量句法 `{k1: v1, k2: v2}` 有所增强，现在支持使用 `**` 拆包、模式匹配和字典推导式。

除了基本的 `dict` 之外，标准库中的 `collections` 模块还提供了随取随用的专门映射，例如 `defaultdict`、`ChainMap` 和 `Counter`。重新实现 `dict` 之后，`OrderedDict` 不像以前那样有用了，但仍然应该留在标准库中，一方面是为了向后兼容，另一方面 `OrderedDict` 还有一些特性是 `dict` 所不具备的，例如比较运算符 `==` 把键的顺序纳入考虑范围。`collections` 模块中的 `UserDict` 是一个基类，供用户自定义映射。

多数映射具有两个强大的方法：`setdefault` 和 `update`。`setdefault` 方法可以更新存放可变值的项（例如 `list` 值），从而避免再次搜索相同的键。`update` 方法可以批量插入或覆盖项，项可以来自提供键值对的可迭代对象，也可以来自关键字参数。映射构造函数在内部也使用 `update`，以便你根据映射、可迭代对象或关键字参数初始化实例。从 Python 3.9 开始，还可以使用 `|=` 运算符更新映射，以及使用 `|` 运算符根据两个映射的合集创建新映射。

映射 API 中的 `__missing__` 方法是一个巧妙的钩子，利用它可以自定义 `d[k]` 句法（调用 `__getitem__`）找不到键时的行为。

`collections.abc` 模块中的抽象基类 `Mapping` 和 `MutableMapping` 定义标准接口，可在运行时检查类型。`types` 模块中的 `MappingProxyType` 为映射包装一层不可变外壳，防止意外更改映射。`Set` 和 `MutableSet` 也有抽象基类。

Python 3 引入的字典视图是一个很好的特性，消除了 Python 2 中 `.keys()`、`.values()` 和 `.items()` 方法开销的内存，不用再构建列表，重复目标 `dict` 实例中的数据。此外，`dict_keys` 和 `dict_items` 类还支持 `frozenset` 的一些最有用的运算符。

3.14 延伸阅读

Python 标准库文档中的“collections—Container datatypes”有一些示例和实践技巧，涵盖多种映射类型。如果你想创建新映射类型，或者了解现有映射的逻辑，`collections` 模块的 Python 源码

(`Lib/collections/__init__.py`) 是不可错过的参考。《Python Cookbook 中文版（第 3 版）》第 1 章有 20 个实用的经典实例，深入挖掘数据结构的用法，尤其是 `dict` 的使用技巧。

Greg Ganderberger 在“Python Dictionaries Are Now Ordered. Keep Using OrderedDict”一文中提倡继续使用 `collections.OrderedDict`，理由有三：一是“明确胜于模糊”，二是向后兼容，三是一些工具和库假定 `dict` 键的顺序无关紧要。

Guido van Rossum 在“PEP 3106—Revamping `dict.keys()`, `.values()` and `.items()`”中提出为 Python 3 增加字典视图功能。摘要指出，这个想法源自 Java Collections Framework。

在几个 Python 解释器中，PyPy 第一个实现了 Raymond Hettinger 提议的紧凑字典，详见博客文章“Faster, more memory efficient and more ordered dictionaries on PyPy”。文中承认，PHP 7 也采用了类似的布局，详见“PHP's new hashtable implementation”一文。指明出处，值得肯定。

在 PyCon 2017，Brandon Rhodes 做了题为“The Dictionary Even Mightier”的演讲，这是经典动画演讲“The Mighty Dictionary”的续集，用动画演示了哈希碰撞。Raymond Hettinger 的演讲“Modern Dictionaries”对 `dict` 的内部机制剖析更为深入。在演讲中他提到，最初他先向 CPython 核心开发团队建议实现紧凑字典，但无果而终，后来成功游说 PyPy 团队，引起 CPython 团队的关注，而后由 INADA Naoki 在 CPython 3.6 中实现。详细信息参阅 CPython 源码 `Objects/dictobject.c` 中的大量注释和设计文档 `Objects/dictnotes.txt`。

为 Python 添加集合的根本原因见“PEP 218—Adding a Built-In Set Object Type”。PEP 218 被批准时，尚未为集合提供专门的字面量句法。`set` 字面量，连同字典和集合推导式，在 Python 3 中才实现，并向后移植到 Python 2.7。在 PyCon 2019，我做了题为“Set Practice: learning from Python's set types”的演讲，通过具体的程序说明了集合的用途，介绍了集合 API 的设计思路和 `uintset` 的实现方式。`uintset` 是一个存放整数元素的集合类，使用的是位向量，而不是哈希表，灵感来自《Go 程序设计语言》第 6 章的一个示例。

IEEE 出版的杂志 *Spectrum* 有一篇关于 Hans Peter Luhn 的报道，他是一位多产的发明家，曾获得一种穿孔卡片组专利，根据可用的成分选择鸡尾酒配方。在他众多的发明中就包括哈希表。详见“Hans Peter Luhn and the Birth of the Hashing Algorithm”一文。

杂谈

语法糖

我的朋友 Geraldo Cohen 说过，Python“简单又正确”。

编程语言纯粹主义者认为句法并不重要。

语法糖诱发分号癌。

——Alan Perlis

句法是编程语言的用户界面，在实践中很重要。

发现 Python 之前，我使用 Perl 和 PHP 做过一些 Web 编程。这些语言的映射句法非常好用，在不得不使用 Java 或 C 语言的日子里，总是令我怀念。

好的映射字面量句法用着十分便利，方便配置，方便实现表驱动设计，还方便存放原型设计和测试数据。Go 语言的设计人员就

从动态语言中学到了这一点。由于缺乏在代码中表达结构化数据的好方法，Java 社区不得不采用烦琐的 XML 作为数据格式。

JSON 是 XML 的一种替代方案，去繁从简，取得了巨大成功，在很多情况下可以取代 XML。列表和字典的句法简洁明了，使 JSON 成为一种出色的数据交换格式。

PHP 和 Ruby 模仿 Perl 的哈希句法，使用 `=>` 链接键和值。JavaScript 与 Python 一样，使用 `:` 一个字符就能表明意图，为什么要用两个呢？¹¹

JSON 源自 JavaScript，无巧不成书，它几乎就是 Python 句法的子集。除了 `true`、`false` 和 `null` 等值的拼写不一样之外，JSON 与 Python 几乎完全兼容。

Armin Ronacher 在一篇推文中说，他喜欢在 Python 的全局命名空间中为 Python 的 `True`、`False` 和 `None` 添加兼容 JSON 的别名，这样就可以直接把 JSON 粘贴到 Python 控制台中了，就像下面这样。

```
>>> true, false, null = True, False, None
>>> fruit = {
...     "type": "banana",
...     "avg_weight": 123.2,
...     "edible_peel": false,
...     "species": ["acuminata", "balbisisiana", "paradisiaca"],
...     "issues": null,
... }
>>> fruit
{'type': 'banana', 'avg_weight': 123.2, 'edible_peel': False,
'species': ['acuminata', 'balbisisiana', 'paradisiaca'],
'issues': None}
```

现在，人人都使用 Python 的 `dict` 和 `list` 句法交换数据。如今，句法越来越便利，还能保留插入顺序。

真是简单又正确。

¹¹从 Ruby 1.9 开始，如果哈希的键是 Symbol 类型，也可以使用：字符。——译者注

第 4 章 Unicode 文本和字节序列

文本给人类阅读，字节序列供计算机处理。

——Esther Nam 和 Travis Fischer
“Character Encoding and Unicode in Python”¹

¹PyCon 2014, “Character Encoding and Unicode in Python”演讲第 12 张幻灯片。

Python 3 明确区分了人类可读的文本字符串和原始的字节序列。把字节序列隐式转换成 Unicode 文本已成过去。本章讨论 Unicode 字符串、二进制序列，以及在二者之间转换时使用的编码。

深入理解 Unicode 对你可能十分重要，也可能无关紧要，这取决于 Python 编程场景。不过，现在用不到，并不代表以后用不到，至少总有需要区分 `str` 和 `byte` 的时候。此外，你会发现专门的二进制序列类型所提供的功能，有些是 Python 2 中“全功能”的 `str` 类型不具备的。

本章涵盖以下内容：

- 字符、码点和字节表述；
- `bytes`、`bytearray` 和 `memoryview` 等二进制序列的特性；
- 全部 Unicode 和陈旧字符集的编码；
- 避免和处理编码错误；
- 处理文本文件的最佳实践；
- 默认编码的陷阱和标准 I/O 的问题；
- 规范化 Unicode 文本，进行安全比较；
- 规范化、大小写同一化和暴力移除变音符的实用函数；
- 使用 `locale` 模块和 `pyuca` 库正确排序 Unicode 文本；

- Unicode 数据库中的字符元数据；
- 能处理 `str` 和 `bytes` 的双模式 API。

4.1 本章新增内容

Python 3 对 Unicode 的支持业已完善、稳定，本章最大的变化是新增了 4.9.1 节，介绍了一个用于搜索 Unicode 数据库的实用函数，方便在命令行中查找带圈数字和笑脸猫等。

另外，6.1 节的“了解默认编码”还有一处小变化，讲到自 Python 3.6 起，Windows 对 Unicode 的支持更好、更简单。

下面先从字符、码点和字节序列讲起，这些概念并不新奇，却是根基。

4.2 字符问题

“字符串”是个相当简单的概念：一个字符串就是一个字符序列。问题出在“字符”的定义上。

在 2021 年，“字符”的最佳定义是 Unicode 字符。因此，从 Python 3 的 `str` 对象中获取的项是 Unicode 字符，从 Python 2 的 `unicode` 对象中获取的项也是 Unicode 字符，而从 Python 2 的 `str` 对象中获取的项是原始字节序列。

Unicode 标准明确区分字符的标识和具体的字节表述。

- 字符的标识，即**码点**，是 0~1 114 111 范围内的数（十进制），在 Unicode 标准中以 4~6 个十六进制数表示，前加“U+”，取值范围是 U+0000~U+10FFFF。例如，字母 A 的码点是 U+0041，欧元符号的码点是 U+20AC，音乐中高音谱号的码点是 U+1D11E。在 Python 3.10.0b4 中使用的 Unicode 13.0.0 标准中，约 13% 的有效码点对应着字符。

- 字符的具体表述取决于所用的**编码**。编码是在码点和字节序列之间转换时使用的算法。例如，字母 A (U+0041) 在 UTF-8 编码中使用单个字节 `\x41` 表述，而在 UTF-16LE 编码中使用字节序列 `\x41\x00` 表述。再比如，欧元符号 (U+20AC) 在 UTF-8 编码中需要 3 个字节，即 `\xe2\x82\xac`，而在 UTF-16LE 中，同一个码点编码成两个字节，即 `\xac\x20`。

把码点转换成字节序列的过程叫**编码**，把字节序列转换成码点的过程叫**解码**。示例 4-1 演示了二者之间的区别。

示例 4-1 编码和解码

```
>>> s = 'café'
>>> len(s) ❶
4
>>> b = s.encode('utf8') ❷
>>> b
b'caf\xc3\xa9' ❸
>>> len(b) ❹
5
>>> b.decode('utf8') ❺
'café'
```

- ❶ 字符串 `'café'` 有 4 个 Unicode 字符。
- ❷ 使用 UTF-8 把 `str` 对象编码成 `bytes` 对象。
- ❸ `bytes` 字面量以 `b` 开头。
- ❹ 字节序列 `b` 有 5 个字节（在 UTF-8 中，“é”的码点编码成两个字节）。
- ❺ 使用 UTF-8 把 `bytes` 对象解码成 `str` 对象。



如果你记不住 `.decode()` 和 `.encode()` 的区别，可以把字节序列当成晦涩难懂的机器核心转储，把 Unicode 字符串当

成“人类可读”的文本。这样你就知道，把字节序列变成人类可读的字符串是**解码**，而把字符串变成用于存储或传输的字节序列是**编码**。

虽然 Python 3 的 `str` 类型基本相当于 Python 2 的 `unicode` 类型，只不过换了个新名称而已，但是 Python 3 的 `bytes` 类型并不是把 `str` 类型换个名称那么简单，而且还有关系紧密的 `bytearray` 类型。因此，在讨论编码和解码问题之前，有必要先介绍一下二进制序列类型。

4.3 字节概要

新的二进制序列类型在很多方面与 Python 2 的 `str` 类型不同。首先要知道，Python 内置两种基本的二进制序列类型：Python 3 引入的不可变类型 `bytes` 和 Python 2.6 添加的可变类型 `bytearray`。

²Python 文档有时把 `bytes` 和 `bytearray` 统称为“字节字符串”。这个术语有歧义，我一般不用。

²Python 2.6 和 2.7 还有 `bytes` 类型，不过只是 `str` 类型的别名。

`bytes` 和 `bytearray` 中的项是 0~255（含）的整数，而不像 Python 2 的 `str` 对象那样是单个字符。然而，二进制序列的切片始终是同一类型的二进制序列，包括长度为 1 的切片，如示例 4-2 所示。

示例 4-2 包含 5 个字节的 `bytes` 和 `bytearray` 对象

```
>>> cafe = bytes('café', encoding='utf_8') ❶
>>> cafe
b'caf\xc3\xa9'
>>> cafe[0] ❷
99
>>> cafe[:1] ❸
b'c'
>>> cafe_arr = bytearray(cafe)
```

```
>>> cafe_arr ❹  
bytearray(b'caf\xc3\xa9')  
>>> cafe_arr[-1:] ❺  
bytearray(b'\xa9')
```

- ❶ 指定编码，可以根据 `str` 对象构建 `bytes` 对象。
- ❷ 各项是 `range(256)` 内的整数。
- ❸ `bytes` 对象的切片还是 `bytes` 对象，即使是只有一个字节的切片。
- ❹ `bytearray` 对象没有字面量句法，显示为 `bytearray()` 调用形式，参数是一个 `bytes` 字面量。
- ❺ `bytearray` 对象的切片也还是 `bytearray` 对象。



`my_bytes[0]` 获取的是一个整数，而 `my_bytes[:1]` 返回的是一个长度为 1 的字节序列。如果你不理解这种行为，只能说明你习惯了 Python 的 `str` 类型。对 `str` 类型来说，`s[0] == s[:1]`。除此之外，对于 Python 中的其他所有序列类型，一个项不能等同于长度为 1 的切片。

虽然二进制序列其实是整数序列，但是它们的字面量表示法表明其中含有 ASCII 文本。因此，字节的值可能会使用以下 4 种不同方式显示。

- 十进制代码在 32~126 范围内的字节（从空格到波浪号 ~），使用 ASCII 字符本身。
- 制表符、换行符、回车符和 \ 对应的字节，使用转义序列 `\t`、`\n`、`\r` 和 `\\`。
- 如果字节序列同时包含两种字符串分隔符 ' 和 "，整个序列使用 ' 区隔，序列内的 ' 转义为 `\'`。³

- 其他字节的值，使用十六进制转义序列（例如，`\x00` 是空字节）。

³小知识：Python 默认用作字符串分隔符的 ASCII“单引号”字符，在 Unicode 标准中的名称实际上是 apostrophe（撇号）。真正的单引号是不对称的，左边是 U+2018，右边是 U+2019。

因此，在示例 4-2 中，我们看到的是 `b'caf\xc3\xa9'`：前 3 个字节 `b'caf'` 在可打印的 ASCII 范围内，后两个字节则不然。

在 `str` 类型的一众方法中，除了格式化方法 `format` 和 `format_map`，以及处理 Unicode 数据的方法 `casefold`、`isdecimal`、`isidentifier`、`isnumeric`、`isprintable` 和 `encode` 之外，其他方法均受 `bytes` 和 `bytearray` 类型的支持。这意味着可以使用熟悉的字符串方法处理二进制序列，例如 `endswith`、`replace`、`strip`、`translate`、`upper` 等，只不过把 `str` 类型的参数换成 `bytes`。另外，如果正则表达式编译自二进制序列而不是字符串，那么 `re` 模块中的正则表达式函数也能处理二进制序列。从 Python 3.5 开始，`%` 运算符又能处理二进制序列了。⁴

⁴在 Python 3.0~3.4 中，`%` 运算符不能处理二进制序列，为需要处理二进制数据的开发人员带来很多痛苦。这次逆转的说明见“PEP 461—Adding `%` formatting to `bytes` and `bytearray`”。

二进制序列有一个类方法是 `str` 没有的，名为 `fromhex`，它的作用是解析十六进制数字对（数字对之间的空格是可选的），构建二进制序列。

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

构建 `bytes` 或 `bytearray` 实例还可以调用各自的构造函数，传入以下参数。

- 一个 `str` 对象和 `encoding` 关键字参数。

- 一个可迭代对象，项为 0~255 范围内的数。
- 一个实现了缓冲协议的对象（例如 `bytes`、`bytearray`、`memoryview`、`array.array`）。构造函数把源对象中的字节序列复制到新创建的二进制序列中。



在 Python 3.5 之前，`bytes` 和 `bytearray` 构造函数的参数还可以是一个整数，使用空字节创建对应长度的二进制序列。这种签名在 Python 3.5 中弃用，在 Python 3.6 中正式移除。详见“PEP 467—Minor API improvements for binary sequences”。

使用缓冲类对象构建二进制序列是一种底层操作，可能涉及类型转换，如示例 4-3 所示。

示例 4-3 使用数组中的原始数据初始化 `bytes` 对象

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
>>> octets = bytes(numbers) ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

- ❶ 指定类型代码 `h`，创建一个短整数（16 位）数组。
- ❷ 把组成 `numbers` 的字节序列赋值给 `octets`，存储一份副本。
- ❸ 这些是表示那 5 个短整数的 10 个字节。

使用缓冲类对象创建 `bytes` 或 `bytearray` 对象，始终复制源对象中的字节序列。与之相反，`memoryview` 对象在二进制数据结构之间共享内存，详见 2.10.2 节。

简要探讨 Python 的二进制序列类型之后，下面说明如何在它们和字符串之间转换。

4.4 基本的编码解码器

Python 自带超过 100 种**编码解码器**（codec, encoder/decoder），用于在文本和字节之间相互转换。每种编码解码器都有一个名称，例如 'utf_8'，而且经常有几个别名，例如 'utf8'、'utf-8' 和 'U8'。这些名称可以传给 open()、str.encode()、bytes.decode() 等函数的 encoding 参数。示例 4-4 使用 3 种编码解码器把相同的文本编码成不同的字节序列。

示例 4-4 使用 3 种编码解码器编码字符串“El Niño”，得到的字节序列差异很大

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8   b'El Ni\xc3\xb1o'
utf_16  b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xfl\x00o\x00'
```

图 4-1 展示了不同编码解码器对字母“A”和高音谱号等字符编码后得到的字节序列。注意，后 3 种是可变长度多字节编码。

字符	码点	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
é	U+006F	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
Г	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
¿	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

图 4-1：12 个字符，它们的码点及不同编码的字节表述（十六进制；星号表示字符不能使用那种编码表述）

图 4-1 中的星号表明，某些编码（例如 ASCII 和多字节的 GB2312）不能表示所有 Unicode 字符。然而，UTF 编码的设计目的就是处理每一个 Unicode 码点。

图 4-1 中展示的是一些典型编码，介绍如下。

`latin1` (即 `iso8859_1`)

一种重要的编码，是其他编码的基础，例如 `cp1252` 和 Unicode（注意，`latin1` 与 `cp1252` 的字节值是一样的，甚至连码点也相同）。

`cp1252`

Microsoft 制定的 `latin1` 超集，添加了有用的符号，例如弯引号和 €（欧元符号）。有些 Windows 应用程序将其称为“ANSI”，但它并不是 ANSI 标准。

`cp437`

IBM PC 最初的字符集，包含框图符号。与后来出现的 `latin1` 不兼容。

`gb2312`

用于编码简体中文的陈旧标准。亚洲语言使用较广泛的多字节编码之一。

`utf-8`

目前 Web 最常用的 8 位编码。据 W3Techs 发布的“Usage statistics of character encodings for websites”报告，截至 2021 年 7 月，97% 的网站使用 UTF-8。2014 年 9 月，本书英文版第 1 版出版时，这一比例是 81.4%。

`utf-16le`

UTF 16 位编码方案的一种形式。所有 UTF-16 支持通过转义序列（称为“代理对”，surrogate pair）表示超过 U+FFFF 的码点。



UTF-16 取代了 1996 年发布的 Unicode 1.0 编码（UCS-2）。UCS-2 编码支持的码点最大只到 U+FFFF，现已弃用，不过在很多系统中仍有使用。截至 2021 年，已分配的码点中超过 57% 大于 U+FFFF，其中包括十分重要的表情符号（emoji）。

概述常用的编码之后，下面探讨编码和解码操作涉及的问题。

4.5 处理编码和解码问题

`UnicodeError` 是一般性的异常，Python 在报告错误时通常更具体，抛出 `UnicodeEncodeError`（把 `str` 转换成二进制序列时出错）或 `UnicodeDecodeError`（把二进制序列转换成 `str` 时出错）。如果源码的编码与预期不符，那么加载 Python 模块时还可能抛出 `SyntaxError`。接下来的几节将说明如何处理这些错误。



出现与 Unicode 有关的错误时，首先要明确异常的类型。要知道导致编码问题的究竟是 `UnicodeEncodeError`、`UnicodeDecodeError`，还是其他错误（例如 `SyntaxError`）。解决问题之前必须清楚这一点。

4.5.1 处理 `UnicodeEncodeError`

多数非 UTF 编码解码器只能处理 Unicode 字符的一小分子集。把文本转换成字节序列时，如果目标编码没有定义某个字符，则会抛出 `UnicodeEncodeError`，除非把 `errors` 参数传给编码方法或函数，做特殊处理。处理错误的方式如示例 4-5 所示。

示例 4-5 把 str 编码成字节序列，有些成功，有些需要处理错误

```
>>> city = 'São Paulo'
>>> city.encode('utf_8') ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1') ❷
b'S\xe3o Paulo'
>>> city.encode('cp437') ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../lib/python3.4/encodings/cp437.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3'
in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore') ❹
b'So Paulo'
>>> city.encode('cp437', errors='replace') ❺
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace') ❻
b'S&#227;o Paulo'
```

- ❶ UTF 编码能处理任何 str 对象。
- ❷ 'iso8859_1' 编码也能处理字符串 'São Paulo'。
- ❸ 'cp437' 无法编码 'ã'（带波浪号的“a”）。默认的错误处理方式是 'strict'，即抛出 UnicodeEncodeError。
- ❹ error='ignore' 处理方式跳过无法编码的字符。这样做通常很不妥，会导致数据悄无声息地丢失。
- ❺ 编码时指定 error='replace'，无法编码的字符使用 '?' 代替。也有数据丢失，不过能提醒用户出了问题。
- ❻ 'xmlcharrefreplace' 把无法编码的字符替换成 XML 实体。如果你无法使用 UTF，而且不想丢失数据，那么这就是唯一的选择。



编码解码器的错误处理方式可以扩展。可以为 `errors` 参数注册额外的字符串，方法是把一个名称和一个错误处理函数传给 `codecs.register_error` 函数。详见 `codecs.register_error` 函数的文档。

据我所知，ASCII 是所有编码的共同子集，因此，只要文本全是 ASCII 字符，编码就一定能成功。Python 3.7 新增了布尔值方法 `str.isascii()`，用于检查 Unicode 文本是不是全部由 ASCII 字符构成。如果是，那就可以放心使用任何编码把文本转换成字节序列，而且肯定不会抛出 `UnicodeEncodeError`。

4.5.2 处理 `UnicodeDecodeError`

并非所有字节都包含有效的 ASCII 字符，也并非所有字节序列都是有效的 UTF-8 或 UTF-16 码点。因此，把二进制序列转换成文本时，如果假定使用的是这两个编码中的一个，则遇到无法转换的字节序列时将抛出 `UnicodeDecodeError`。

另一方面，`'cp1252'`、`'iso8859_1'` 和 `'koi8_r'` 等陈旧的 8 位编码能解码任何字节序列流（包括随机噪声），而不抛出错误。因此，如果程序使用错误的 8 位编码，则可能生成乱码，也不会报错。



乱码字符称为鬼符（gremlin）或 mojibake（文字化け，“变形文本”的日文）。

示例 4-6 演示使用错误的编码解码器可能得到鬼符，或抛出 `UnicodeDecodeError`。

示例 4-6 把字节序列解码成 `str`，有些成功，有些需要处理错误

```
>>> octets = b'Montr\xe9al' ❶
>>> octets.decode('cp1252') ❷
```

```

'Montréal'
>>> octets.decode('iso8859_7') ❸
'Montrial'
>>> octets.decode('koi8_r') ❹
'MontrMal'
>>> octets.decode('utf_8') ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in
position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace') ❻
'Montr al'

```

- ❶ 使用 `latin1` 编码单词“Montréal”，`'\xe9'` 是“é”的字节。
- ❷ 可以使用 `'cp1252'` 解码，因为它是 `latin1` 的超集。
- ❸ `'iso8859_7'` 针对希腊语，因此无法正确解释 `'\xe9'` 字节，而且没有抛出错误。
- ❹ `'koi8_r'` 针对俄文。在这种编码中，`'\xe9'` 表示西里尔字母“И”。
- ❺ `'utf_8'` 编码解码器检测到 `octets` 不是有效的 UTF-8 二进制序列，抛出 `UnicodeDecodeError`。
- ❻ 使用 `'replace'` 错误处理方式，`\xe9` 被替换成“”（码点是 U+FFFD）。这是 Unicode 标准中的 REPLACEMENT CHARACTER，表示未知字符。

4.5.3 加载模块时编码不符合预期抛出的 `SyntaxError`

Python 3 默认使用 UTF-8 编码源码，Python 2 则默认使用 ASCII。如果加载的 `.py` 模块中包含 UTF-8 之外的数据，而且没有声明编码，那么将看到类似下面的消息。

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on
line
    1, but no encoding declared; see
https://python.org/dev/peps/pep-0263/
    for details
```

GNU/Linux 和 macOS 系统大都使用 UTF-8，因此打开在 Windows 系统中使用 cp1252 编码的 .py 文件时就可能遇到这种情况。注意，这个错误在 Windows 版 Python 中也可能会发生，因为 Python 3 源码在所有平台中均默认使用 UTF-8 编码。

为了解决这个问题，可以在文件顶部添加一个神奇的 coding 注释，如示例 4-7 所示。

示例 4-7 ola.py: 打印葡萄牙语“你好，世界！”

```
# coding: cp1252

print('Olá, Mundo!')
```



现在，Python 3 源码不再限于使用 ASCII，而是默认使用优秀的 UTF-8 编码，因此要修正源码的陈旧编码（例如 'cp1252'）问题，最好将其转换成 UTF-8，别去麻烦 coding 注释。如果你用的编辑器不支持 UTF-8，那么是时候换一个了。

假如有一个文本文件，里面保存的是源码或诗句，但是你不知道它的编码。那么，如何查明具体使用的是什么编码呢？4.5.4 节将揭晓答案。

4.5.4 如何找出字节序列的编码

如何自己找出字节序列的编码呢？简单来说，不能。这只能由别人来告诉你。

有些通信协议和文件格式，例如 HTTP 和 XML，通过首部明确指明内容编码。如果字节流中包含大于 127 的字节值，则可以肯定，用的不

是 ASCII 编码。另外，按照 UTF-8 和 UTF-16 的设计方式，可用的字节序列也受到限制。

Leo 猜测 UTF-8 解码的技巧

(以下几段摘自技术审校 Leonardo Rochael 在本书草稿中留下的评语。)

按照 UTF-8 的设计方式，一段随机字节序列，甚至是使用其他编码得到的非随机字节序列，解码结果几乎不可能是乱码，更不会抛出 `UnicodeDecodeError`。

原因在于，UTF-8 转义序列绝不使用 ASCII 字符，而且转义序列具有位模式，即使是随机数据也很难产生无效的 UTF-8 字符。

因此，如果你可以把十进制代码大于 127 的字节解码为 UTF-8，那么它使用的编码可能就是 UTF-8。

处理巴西在线服务（其中一些附加到传统后端），我有时迫不得已，只能先尝试按照 UTF-8 解码，万一遇到 `UnicodeDecodeError`，再使用 `cp1252` 解码。这样做不太优雅，但是行之有效。

然而，就像人类语言也有规则和限制一样，只要假定字节流是人类可读的纯文本，就可能通过试探和分析找出编码。例如，如果 `b'\x00'` 字节经常出现，那就可能是 16 位或 32 位编码，而不是 8 位编码方案，因为纯文本中不能包含空字符；如果字节序列 `b'\x20\x00'` 经常出现，那就可能是 UTF-16LE 编码中的空格字符（U+0020），而不是鲜为人知的 U+2000 EN QUAD 字符——谁知道这是什么呢！

统一字符编码侦测包 Chardet 就是这样工作的，它能识别所支持的 30 种编码。Chardet 是一个 Python 库，可以在程序中使用，不过它也提供了命令行实用工具 `chardetect`。下面使用它侦测本章书稿文件的编码。

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

二进制序列编码的文本通常不包含明确的编码线索，而 UTF 格式可以在文本内容的开头添加一个字节序标记，详见 4.5.5 节。

4.5.5 BOM：有用的鬼符

在示例 4-4 中，你可能注意到了，UTF-16 编码的序列开头有几个额外的字节，如下所示。

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

我指的是 `b'\xff\xfe'`。这是 BOM，即**字节序标记**（byte-order mark），指明编码时使用 Intel CPU 的小端序。

在小端序设备中，各个码点的最低有效字节在前面。例如，字母 'E' 的码点是 U+0045（十进制数 69），在字节偏移的第 2 位和第 3 位编码为 69 和 0。

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

在大端序 CPU 中，编码顺序反过来，'E' 被编码为 0 和 69。

为了避免混淆，UTF-16 编码在要编码的文本前面加上特殊的不可见字符 ZERO WIDTH NO-BREAK SPACE (U+FEFF)。在小端序系统中，这个字符编码为 `b'\xff\xfe'`（十进制数 255, 254）。因为按照设计，Unicode 标准没有 U+FFFE 字符，在小端序编码中，字节序列 `b'\xff\xfe'` 必定是 ZERO WIDTH NO-BREAK SPACE，所以编码解码器知道该用哪个字节序。

UTF-16 有两个变种：UTF-16LE，显式指明使用小端序；UTF-16BE，显式指明使用大端序。如果直接使用这两个变种，则不生成

BOM。

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

如果有 BOM，那么 UTF-16 编码解码器应把开头的 ZERO WIDTH NO-BREAK SPACE 字符去掉，只提供文件中真正的文本内容。根据 Unicode 标准，如果文件使用 UTF-16 编码，而且没有 BOM，那么应该假定使用的是 UTF-16BE（大端序）。然而，Intel x86 架构用的是小端序，因此也有很多文件用的是不带 BOM 的小端序 UTF-16 编码。

字节序只对一个字（word）占多个字节的编码（例如 UTF-16 和 UTF-32）有影响。UTF-8 的一大优势是，不管设备使用哪种字节序，生成的字节序列始终一致，因此不需要 BOM。尽管如此，某些 Windows 应用程序（比如 Notepad）依然会在 UTF-8 编码的文件中添加 BOM。而且，Excel 会根据有没有 BOM 来确定文件是不是 UTF-8 编码，不然它就假设内容使用 Windows 代码页（code page）编码。带有 BOM 的 UTF-8 编码，在 Python 注册的编码解码器中叫作 UTF-8-SIG。UTF-8-SIG 编码的 U+FEFF 字符是一个三字节序列：b'\xef\xbb\xbf'。因此，如果文件以这三个字节开头，可能就是带有 BOM 的 UTF-8 文件。

Caleb 建议使用 UTF-8-SIG

本书技术审校 Caleb Hattingh 建议始终使用 UTF-8-SIG 编码解码器读取 UTF-8 文件。这样做没有任何问题，因为不管文件带不带 BOM，UTF-8-SIG 都能正确读取，而且不返回 BOM 本身。不过，我在书中建议使用 UTF-8，方便互操作。举个例子，在 Unix 系统中，如果 Python 脚本以注释 `#!/usr/bin/env python3` 开头，则可以用作可执行文件。为此，文件的前两个字

节必须是 `b'#!'`，而有了 BOM，这个约定就被打破了。如果你需要导出带有 BOM 的数据，供其他应用使用，应使用 UTF-8-SIG。但是，请记住 Python 编码解码器文档中的一句话：“不鼓励也不建议为 UTF-8 添加 BOM。”

下面换个话题，讨论 Python 3 处理文本文件的方式。

4.6 处理文本文件

目前处理文本的最佳实践是“Unicode 三明治”原则（见图 4-2）。⁵ 根据这个原则，我们应当尽早把输入的 `bytes`（例如读取文件得到）解码成 `str`。在 Unicode 三明治中，“肉饼”是程序的业务逻辑，在这里只能处理 `str` 对象。在其他处理过程中，一定不能编码或解码。对输出来说，则要尽量晚地把 `str` 编码成 `bytes`。多数 Web 框架是这样做的，在使用框架的过程中，我们很少接触到 `bytes`。例如，在 Django 中，视图应该输出 Unicode 字符串，至于如何把响应编码成 `bytes`（默认使用 UTF-8），是 Django 的事。

⁵Ned Batchelder 在 PyCon US 2012 做了精彩演讲“Pragmatic Unicode”，这是我第一次听说“Unicode 三明治”。



图 4-2：Unicode 三明治——目前处理文本的最佳实践

在 Python 3 中，我们可以轻松地采纳“Unicode 三明治”的建议，因为内置函数 `open()` 在读取文件时会做必要的解码，以文本模式写入文件时还会做必要的编码，所以调用 `my_file.read()` 方法得到的以及传给 `my_file.write(text)` 方法的都是 `str` 对象。

可见，处理文本文件很简单。但是，如果依赖默认编码，你就会遇到麻烦。

看一下示例 4-8 中的控制台会话。你能发现问题吗？

示例 4-8 一个平台的编码问题（在你自己的设备中不一定遇到这个问题）

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

示例 4-8 的问题是，写入文件时指定了 UTF-8 编码，读取文件时却没有这么做。Python 假定使用 Windows 系统的默认编码（代码页 1252），导致文件的最后一个字节被解码成字符 'Ã©'，而不是 'é'。

我在 64 位 Windows 10 (build 18363) 中使用 Python 3.8.1 运行示例 4-8。在新版 GNU/Linux 或 macOS 中运行同样的语句则没有问题，因为这几个操作系统的默认编码是 UTF-8，让人误以为一切正常。如果打开文件是为了写入，但是没有指定编码参数，Python 则会使用区域设置中的默认编码，再使用同样的编码也能正确读取文件。然而，在不同的平台中，脚本生成的字节内容不一样，即使是在同一个平台中，由于区域设置不同，也会导致兼容问题。



需要在多台设备中或多种场合下运行的代码，一定不能依赖默认编码。打开文件时始终应该明确传入 `encoding=` 参数，因

为不同的设备使用的默认编码可能不同，有时隔一天也会发生变化。

示例 4-8 有一个奇怪的细节：第一个语句中的 `write` 函数报告写入了 4 个字符，但是下一行读取时得到了 5 个字符。示例 4-9 在示例 4-8 的基础上增加了一些代码，对这个问题以及其他细节做了说明。

示例 4-9 仔细分析在 Windows 中运行的示例 4-8，找出问题并修正

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café') ❷
4
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size ❸
5
>>> fp2 = open('cafe.txt')
>>> fp2 ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ❺
'cp1252'
>>> fp2.read() ❻
'cafÃ©'
>>> fp3 = open('cafe.txt', encoding='utf_8') ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read() ❸
'café'
>>> fp4 = open('cafe.txt', 'rb') ❾
>>> fp4 ❿
<_io.BufferedReader name='cafe.txt'>
>>> fp4.read() ⓫
b'caf\xc3\xa9'
```

❶ 默认情况下，`open` 函数采用文本模式，返回一个使用指定方式编码的 `TextIOWrapper` 对象。

- ❷ 在 `TextIOWrapper` 对象上调用 `write` 方法，返回写入的 Unicode 字符数。
- ❸ `os.stat` 报告文件中有 5 个字节，因为 UTF-8 编码的 'é' 占两个字节，`0xc3` 和 `0xa9`。
- ❹ 打开文本文件，不显式指定编码，返回一个 `TextIOWrapper` 对象，使用区域设置中的默认编码。
- ❺ `TextIOWrapper` 对象有个 `encoding` 属性，查看该属性的值，发现这里的编码是 `cp1252`。
- ❻ 在 Windows `cp1252` 编码中，`0xc3` 字节是“Ã”（带波浪号的 A），`0xa9` 字节是版权符号。
- ❼ 使用正确的编码打开那个文件。
- ❽ 结果符合预期，得到 4 个 Unicode 字符，即 'café'。
- ❾ 'rb' 标志指明以二进制模式读取文件。
- ❿ 返回一个 `BufferedReader` 对象，而不是 `TextIOWrapper` 对象。
- ⓫ 读取返回的字节序列，结果与预期相符。



除非想判断编码，否则不要以二进制模式打开文本文件。即使你真的想判断编码，也应该使用 `Chardet`，而不要重新发明轮子（参见 4.5.4 节）。一般来说，二进制模式只能用于打开二进制文件，例如光栅图像。

示例 4-9 的问题是，打开文本文件时依赖了默认设置。默认设置有许多来源，接下来会说明。

了解默认编码

在 Python 中，I/O 默认使用的编码受到几个设置的影响，如示例 4-10 中的 `default_encodings.py` 脚本所示。

示例 4-10 探索默认编码

```
import locale
import sys

expressions = """
    locale.getpreferredencoding()
    type(my_file)
    my_file.encoding
    sys.stdout.isatty()
    sys.stdout.encoding
    sys.stdin.isatty()
    sys.stdin.encoding
    sys.stderr.isatty()
    sys.stderr.encoding
    sys.getdefaultencoding()
    sys.getfilesystemencoding()
    """

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(f'{expression:>30} -> {value!r}')
```

示例 4-10 在 GNU/Linux (Ubuntu 14.04~19.10) 和 macOS (10.9~10.14) 中的输出一样，表明这些系统始终使用 UTF-8。

```
$ python3 default_encodings.py
locale.getpreferredencoding() -> 'UTF-8'
    type(my_file) -> <class '_io.TextIOWrapper'>
    my_file.encoding -> 'UTF-8'
    sys.stdout.isatty() -> True
    sys.stdout.encoding -> 'utf-8'
    sys.stdin.isatty() -> True
    sys.stdin.encoding -> 'utf-8'
    sys.stderr.isatty() -> True
    sys.stderr.encoding -> 'utf-8'
    sys.getdefaultencoding() -> 'utf-8'
    sys.getfilesystemencoding() -> 'utf-8'
```


然而，在 Windows 中的输出有所不同，如示例 4-11 所示。

示例 4-11 在 Windows 10 PowerShell 中运行，查看默认编码
(在 cmd.exe 中的输出相同)

```
> chcp ❶
Active code page: 437
> python default_encodings.py ❷
locale.getpreferredencoding() -> 'cp1252' ❸
    type(my_file) -> <class '_io.TextIOWrapper'>
    my_file.encoding -> 'cp1252' ❹
    sys.stdout.isatty() -> True ❺
    sys.stdout.encoding -> 'utf-8' ❻
    sys.stdin.isatty() -> True
    sys.stdin.encoding -> 'utf-8'
    sys.stderr.isatty() -> True
    sys.stderr.encoding -> 'utf-8'
    sys.getdefaultencoding() -> 'utf-8'
    sys.getfilesystemencoding() -> 'utf-8'
```

- ❶ chcp 输出控制台当前的活动代码页，这里是 437。
- ❷ 运行 default_encodings.py，结果输出到控制台。
- ❸ locale.getpreferredencoding() 是最重要的设置。
- ❹ 文本文件默认使用 locale.getpreferredencoding()。
- ❺ 结果输出到控制台中，因此 sys.stdout.isatty() 返回 True。
- ❻ 现在，sys.stdout.encoding 与 chcp 报告的控制台代码页不一样。

自本书第 1 版出版后，Windows 自身和 Windows 版 Python 对 Unicode 的支持有所改善。以前，在 Windows 7 中使用 Python 3.4 运行示例 4-11，报告 4 种编码。之前，stdout、stdin 和 stderr 使用的编码与 chcp 命令报告的代码页相同，现在全都使用 utf-8。

这要归功于 Python 3.6 实现的“PEP 528—Change Windows console encoding to UTF-8”，以及（从 2018 年 10 月发布的 Windows 1809 之后的）PowerShell 和 cmd.exe 对 Unicode 的支持。⁶ 奇怪的是，把 `stdout` 写入控制台时，`chcp` 和 `sys.stdout.encoding` 报告的编码不一样。但是，现在我们可以 Windows 中打印 Unicode 字符串了，不会出现编码错误，这是一大进步。然而，倘若把输出重定向到文件中，情况并不乐观，详见后文。不要太过乐观，你心心念念的表情符号仍然不一定能显示在控制台中，这取决于控制台使用的字体。

⁶来源：“Windows Command-Line: Unicode and UTF-8 Output Text Buffer”。

另一个变化是同在 Python 3.6 中实现的“PEP 529—Change Windows filesystem encoding to UTF-8”，把文件系统的编码由 Microsoft 专属的 MBCS 改成了 UTF-8。

然而，如果把示例 4-10 的输出重定向到文件中，结果如下所示。

```
Z:\>python default_encodings.py > encodings.log
```

那么，`sys.stdout.isatty()` 的值变成 `False`，`sys.stdout.encoding` 由 `locale.getpreferredencoding()` 设置（在那台设备中是 `'cp1252'`），而 `sys.stdin.encoding` 和 `sys.stderr.encoding` 仍是 `utf-8`。



在示例 4-12 中，我使用 `'\N{...}'` 转义 Unicode 字面量，内部是字符的官方名称。这样做是比较麻烦，不过安全，如果字符名称不存在，则 Python 抛出 `SyntaxError`。这比编写一个十六进制数好多了——它可能出错，而且不易察觉。而且，有时你会在注释中说明字符代码的意思，那么直接使用 `\N{...}` 岂不是更好。

这意味着，示例 4-12 把输出打印到控制台中时一切正常，而把输出重定向到文件中时就可能遇到问题。

示例 4-12 stdout_check.py

```
import sys
from unicodedata import name

print(sys.version)
print()
print('sys.stdout.isatty():', sys.stdout.isatty())
print('sys.stdout.encoding:', sys.stdout.encoding)
print()

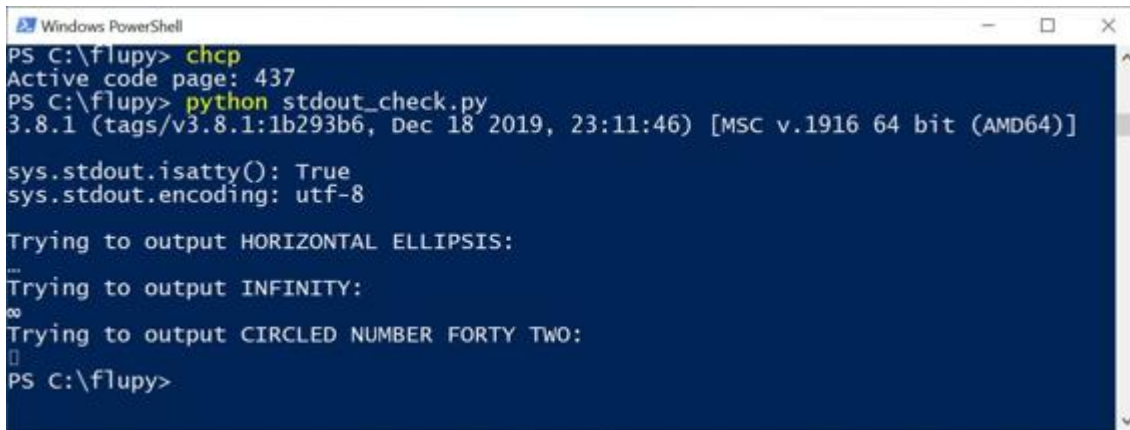
test_chars = [
    '\N{HORIZONTAL ELLIPSIS}',      # cp1252中存在, cp437中不存在
    '\N{INFINITY}',                  # cp437中存在, cp1252中不存在
    '\N{CIRCLED NUMBER FORTY TWO}', # cp437和cp1252中都不存在
]

for char in test_chars:
    print(f'Trying to output {name(char)}:')
    print(char)
```

示例 4-12 显示 `sys.stdout.isatty()` 和 `sys.stdout.encoding` 的值，以及以下 3 个字符。

- '...' (HORIZONTAL ELLIPSIS) : CP 1252 中存在, CP 437 中不存在。
- '∞' (INFINITY) : CP 437 中存在, CP 1252 中不存在。
- ④₂ (CIRCLED NUMBER FORTY TWO) : CP 437 和 CP 1252 中都不存在。

在 PowerShell 或 cmd.exe 中运行 `stdout_check.py`，结果如图 4-3 所示。

A screenshot of a Windows PowerShell window. The title bar says "Windows PowerShell". The command prompt shows the user running 'chcp' which reports 'Active code page: 437'. Then they run 'python stdout_check.py'. The output shows the Python version '3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]'. It then checks 'sys.stdout.isatty(): True' and 'sys.stdout.encoding: utf-8'. It attempts to output 'HORIZONTAL ELLIPSIS:' (which appears as three dots), 'INFINITY:' (which appears as an infinity symbol), and 'CIRCLED NUMBER FORTY TWO:' (which appears as a square box). The prompt returns to 'PS C:\flupy>'.

```
PS C:\flupy> chcp
Active code page: 437
PS C:\flupy> python stdout_check.py
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

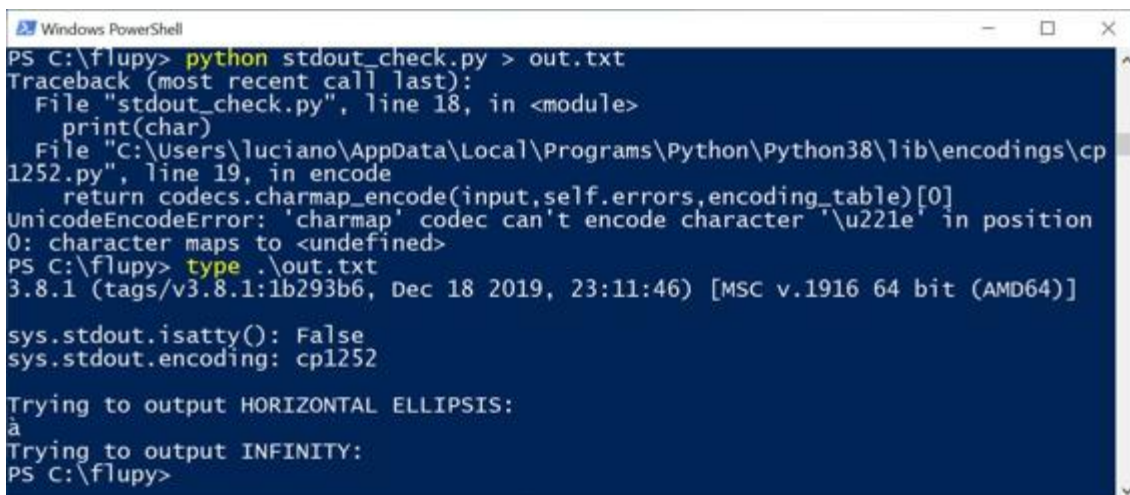
sys.stdout.isatty(): True
sys.stdout.encoding: utf-8

Trying to output HORIZONTAL ELLIPSIS:
...
Trying to output INFINITY:
∞
Trying to output CIRCLED NUMBER FORTY TWO:
□
PS C:\flupy>
```

图 4-3: 在 PowerShell 中运行 stdout_check.py

尽管 chcp 报告当前的活动代码页是 437，但是 sys.stdout.encoding 的值是 UTF-8，因此 HORIZONTAL ELLIPSIS 和 INFINITY 都能正确输出。CIRCLED NUMBER FORTY TWO 显示为一个方框，没有抛出错误。这可能表明，PowerShell 能识别这个字符，只是控制台使用的字体没有显示它的字形。

然而，把 stdout_check.py 的输出重定向到文件中，我得到的结果如图 4-4 所示。

A screenshot of a Windows PowerShell window. The user runs 'python stdout_check.py > out.txt'. This results in a 'UnicodeEncodeError: 'charmap' codec can't encode character '\u221e' in position 0: character maps to <undefined>'. The error message shows the traceback from stdout_check.py line 18 to the Python encoding module. After the error, the user runs 'type .\out.txt', which shows the output of the script up to the point of the error: 'Trying to output HORIZONTAL ELLIPSIS:' followed by three dots, and 'Trying to output INFINITY:' followed by an infinity symbol. The prompt returns to 'PS C:\flupy>'.

```
PS C:\flupy> python stdout_check.py > out.txt
Traceback (most recent call last):
  File "stdout_check.py", line 18, in <module>
    print(char)
  File "C:\Users\luciano\AppData\Local\Programs\Python\Python38\lib\encodings\cp1252.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u221e' in position 0: character maps to <undefined>
PS C:\flupy> type .\out.txt
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

sys.stdout.isatty(): False
sys.stdout.encoding: cp1252

Trying to output HORIZONTAL ELLIPSIS:
...
Trying to output INFINITY:
∞
PS C:\flupy>
```

图 4-4: 在 PowerShell 中运行 stdout_check.py，重定向输出

图 4-4 中的第一个问题是提到 `'\u221e'` 字符的 `UnicodeEncodeError`，因为 `sys.stdout.encoding` 的值是 `'cp1252'`，这个代码页没有 `INFINITY` 字符。

使用 `type` 命令读取 `out.txt` 文件（也可以使用 Windows 平台中的编辑器，例如 VS Code 或 Sublime Text），你会发现 `HORIZONTAL ELLIPSIS` 显示为 `'à'`（LATIN SMALL LETTER A WITH GRAVE）。这是因为，字节值 `0x85` 在 CP 1252 中表示 `'...'`，而在 CP 437 中表示 `'à'`。可见，活动代码页还是有影响的，只是不那么合理，用处也不大，对 Unicode 体验有一定的干扰。



以上实验使用一台为美国市场配置的笔记本电脑，在 Windows 10 OEM 中运行。为其他国家及地区本地化的 Windows 版本可能使用不同的编码配置。例如，在巴西，Windows 控制台默认使用代码页 850，而不是 437。

根据示例 4-11 使用的不同编码，我们总结一下令人抓狂的默认编码。

- 打开文件时如果没有指定 `encoding` 参数，则默认编码由 `locale.getpreferredencoding()` 指定（示例 4-11 中是 `'cp1252'`）。
- 在 Python 3.6 之前，`sys.stdout/stdin/stderr` 编码由环境变量 `PYTHONIOENCODING` 设置。现在，Python 忽略这个变量，除非把 `PYTHONLEGACYWINDOWSSTDIO` 设为一个非空字符串。否则，交互式环境下的标准 I/O 使用 UTF-8 编码，重定向到文件中的 I/O 则使用 `locale.getpreferredencoding()` 定义的编码。
- 在二进制数据和 `str` 之间转换时，Python 内部使用 `sys.getdefaultencoding()`。该设置不可更改。
- 编码和解码文件名（不是文件内容）使用 `sys.getfilesystemencoding()`。对于 `open()` 函数，如果传入的文件名参数是 `str` 类型，则使用

`sys.getfilesystemencoding()`；如果传入的文件名参数是 `bytes` 类型，则不经改动，直接传给操作系统 API。



在 GNU/Linux 和 macOS 中，这些编码的默认值都是 UTF-8，而且多年以来均是如此，因此 I/O 能处理所有 Unicode 字符。在 Windows 中，不仅同一个系统中使用不同的编码，而且一些代码页（例如 `'cp850'` 和 `'cp1252'`）往往只支持 ASCII，而且不同的代码页之间增加的 127 个字符也有所不同。因此，若不多加小心，Windows 用户就会更容易遇到编码问题。

综上，`locale.getpreferredencoding()` 返回的编码是最重要的，既是打开文本文件时默认使用的编码，也是把 `sys.stdout/stdin/stderr` 重定向到文件时默认使用的编码。然而，文档是这样说的（摘录部分）：

```
locale.getpreferredencoding(do_setlocale=True)
```

根据用户偏好设置，返回文本数据的编码。用户偏好设置在不同的系统中以不同的方式设置，而且在某些系统中可能无法通过编程方式设置，因此这个函数返回的只是猜测的编码。.....

因此，关于默认编码的最佳建议：别依赖默认编码。

遵从“Unicode 三明治”的建议，而且始终在程序中显式指定编码，你将避免很多问题。可惜，即使把 `bytes` 正确转换成 `str`，Unicode 仍有不尽如人意的地方。4.7 节和 4.8 节讨论的话题对 ASCII 世界来说很简单，但在 Unicode 领域就变得相当复杂：文本规范化（即为了比较而把文本转换成统一的表述）和排序。

4.7 为了正确比较而规范化 Unicode 字符串

因为 Unicode 有组合字符（变音符和附加到前一个字符上的其他记号，打印时作为一个整体），所以字符串比较起来很复杂。

例如，“café”这个词可以使用两种方式构成，分别有 4 个和 5 个码点，但是结果看起来完全一样。

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

把 COMBINING ACUTE ACCENT (U+0301) 放在“e”后面，得到的字符是“é”。按 Unicode 标准规定，'é' 和 'e\u0301' 是标准等价物（canonical equivalent），应用程序应把它们视作相同的字符。但是，Python 看到的是不同的码点序列，因此判定二者不相等。

这个问题的解决方案是使用 `unicodedata.normalize()` 函数。该函数的第一个参数是 'NFC'、'NFD'、'NFKC' 和 'NFKD' 这 4 个字符串中的一个。下面先说明前两个。

NFC（Normalization Form C）使用最少的码点构成等价的字符串，而 NFD 把合成字符分解成基字符和单独的组合字符。这两种规范化方式都能让比较行为符合预期，如下所示。

```
>>> from unicodedata import normalize
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

键盘驱动通常能输出合成字符，因此用户输入的文本默认是 NFC 形式。不过，安全起见，保存文本之前，最好使用 `normalize('NFC', user_text)` 规范化字符串。NFC 也是 W3C 推荐的规范化形式，详见“Character Model for the World Wide Web: String Matching and Searching”。

使用 NFC 时，有些单体字符会被规范化成另一个单体字符。例如，电阻的单位欧姆 (Ω) 会被规范化成希腊字母大写的奥米伽。二者在视觉上是一样的，但是比较时并不相等，因此要规范化，以防止出现意外。

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

另外两种规范化形式 NFKC 和 NFKD，字母 K 表示“compatibility”（兼容性）。这两种规范化形式较为严格，对所谓的“兼容字符”有影响。虽然 Unicode 的目标是为每一个字符提供“标准”的码点，但为了兼容现有标准，有些字符会出现多次。例如，尽管希腊字母表中有“ μ ”这个字母（码点是 U+03BC, GREEK SMALL LETTER MU），但 Unicode 还是添加了“微”符号 μ (U+00B5, MICRO SIGN)，以便与 latin1 相互转换。因此，“微”符号是一个“兼容字符”。

在 NFKC 和 NFKD 形式中，兼容字符经兼容性分解，被替换成一个或多个字符。即便这样有些格式损失，但仍是“首选”表述——理想情况下，格式化是外部标记的职责，不应该由 Unicode 处理。举两个例子，二分之一 ' $\frac{1}{2}$ ' (U+00BD) 经过兼容性分解，得到 3 个字符序

列，即 '1/2'；“微”符号 'μ' (U+00B5) 经过兼容性分解，得到小写字母 'µ' (U+03BC)。⁷

⁷“微”符号是兼容字符，而欧姆符号不是，这真是奇怪。鉴于此，NFC 不改动“微”符号，但会把欧姆符号变成大写的奥米伽。而 NFKC 和 NFKD 则把欧姆符号和“微”符号都变成其他字符。

下面是 NFKC 规范化的具体效果。

```
>>> from unicodedata import normalize, name
>>> half = '\N{VULGAR FRACTION ONE HALF}'
>>> print(half)
½
>>> normalize('NFKC', half)
'1/2'
>>> for char in normalize('NFKC', half):
...     print(char, name(char), sep='\t')
...
1    DIGIT ONE
/    FRACTION SLASH
2    DIGIT TWO
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'μ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('μ', 'µ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

使用 '1 /2' 替代 '½' 可以接受，“微”符号也确实是小写希腊字母 'μ'，但是把 '4²' 转换成 '42' 就改变原意了。应用程序可以把 '4²' 保存为 '4²'，可是 normalize 函数对格式一无所知。因此，NFKC 或 NFKD 可能会损失或曲解信息，不过可以为搜索和索引提供便利的中间表述。

然而，Unicode 出现之后，看似简单的事情往往变得更加复杂。

VULGAR FRACTION ONE HALF 经 NFKC 规范化，1 和 2 之间用的

是 FRACTION SLASH，而不是 SOLIDUS，即我们熟悉的 ASCII 字符“斜线”（十进制代码为 47）。因此，如果用户搜索由 3 个 ASCII 字符序列构成的 '1/2'，则不会找到规范化之后的 Unicode 序列。



NFKC 和 NFKD 规范化形式会导致数据损失，应只在特殊情况下使用，例如搜索和索引，而不能用于持久存储文本。

为搜索或索引准备文本时，还有一个有用的操作，即 4.7.1 节讨论的大小写同一化。

4.7.1 大小写同一化

大小写同一化其实就是把所有文本变成小写，再做些其他转换。这个操作由 `str.casefold()` 方法实现。

对于只包含 `latin1` 字符的字符串 `s`，`s.casefold()` 得到的结果与 `s.lower()` 一样，唯有两个例外：“微”符号 'μ' 变成小写希腊字母“μ”（在多数字体中二者看起来一样）；德语 Eszett (“sharp s”，ß) 变成“ss”。

```
>>> micro = 'μ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('μ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

`str.casefold()` 和 `str.lower()` 得到不同结果的码点有近 300 个。

与 Unicode 相关的其他所有问题一样，大小写同一化也很复杂，有许多语言层面上的特殊情况，但是 Python 核心团队尽心尽力，提供了一种能满足多数用户需求的方案。

接下来的内容使用这些规范化知识开发几个实用函数。

4.7.2 规范化文本匹配的实用函数

由前文可知，我们可以放心使用 NFC 和 NFD 比较 Unicode 字符串，结果是合理的。对多数应用程序来说，NFC 是最好的规范化形式。不区分大小写的比较应该使用 `str.casefold()`。

如果需要处理多语言文本，你的工具箱应该增加示例 4-13 中的 `nfc_equal` 和 `fold_equal` 函数。

示例 4-13 normeq.py: 规范化 Unicode 字符串，准确比较

```
"""
规范化Unicode字符串的实用函数，确保准确比较。
```

```
使用NFC规范化形式，区分大小写：
```

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1 == s2
False
>>> nfc_equal(s1, s2)
True
>>> nfc_equal('A', 'a')
False
```

```
使用NFC规范化形式，大小写同一化：
```

```
>>> s3 = 'Straße'
>>> s4 = 'strasse'
>>> s3 == s4
False
>>> nfc_equal(s3, s4)
False
>>> fold_equal(s3, s4)
True
>>> fold_equal(s1, s2)
```

```
True
>>> fold_equal('A', 'a')
True

"""

from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

除了 Unicode 规范化和大小写同一化（均由 Unicode 标准规定）之外，有时需要进行更为深入的转换，例如把 'café' 转换成 'cafe'。4.7.3 节将说明何时以及如何进行这种转换。

4.7.3 极端“规范化”：去掉变音符

Google 搜索有很多技巧，其中一个显然是忽略变音符（例如重音符、下加符等），至少在某些情况下要这么做。去掉变音符不是正确的规范化方式，因为这往往会改变词的意思，而且可能让人误判搜索结果。但这对现实生活有帮助：人们有时很懒，或者不知道怎么正确使用变音符，而且拼写规则会随时间变化，因此实际语言中的重音经常变来变去。

除了搜索，去掉变音符还能让 URL 更易于阅读，至少对拉丁语系语言来说如此。如果想把字符串中的所有变音符都去掉，可以使用示例 4-14 中的函数。

示例 4-14 simplify.py：去掉全部组合记号的函数

```
import unicodedata
import string

def shave_marks(txt):
    """删除所有变音符"""
```

```
norm_txt = unicodedata.normalize('NFD', txt) ❶
shaved = ''.join(c for c in norm_txt
                  if not unicodedata.combining(c)) ❷
return unicodedata.normalize('NFC', shaved) ❸
```

❶ 把所有字符分解成基字符和组合记号。

❷ 过滤所有组合记号。

❸ 重组所有字符。

示例 4-15 演示了 `shave_marks` 函数的效果。

示例 4-15 示例 4-14 中 `shave_marks` 函数的两个使用示例

```
>>> order = '"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of  
açai."'
>>> shave_marks(order)
'"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of acai."' ❶
>>> Greek = 'Ζέφυρος, Zéfiro'
>>> shave_marks(Greek)
'Ζεφυρος, Zefiro' ❷
```

❶ 只替换字母“è”“ç”和“í”。

❷ “ξ”和“é”都被替换了。

示例 4-14 定义的 `shave_marks` 函数使用起来没有问题，不过有点极端。通常，去掉变音符是为了把拉丁文本变成纯粹的 ASCII，但是 `shave_marks` 函数还会修改非拉丁字符（例如希腊字母），而只去掉重音符并不能把它们变成 ASCII 字符。因此，我们应该分析各个基字符，仅当字符在拉丁字母表中时才删除附加的记号，如示例 4-16 所示。

示例 4-16 删除拉丁字母中组合记号的函数（`import` 语句省略了，因为这个函数也在 示例 4-14 定义的 `simplify.py` 模块中）

```

def shave_marks_latin(txt):
    """删除所有拉丁基字符上的变音符"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    latin_base = False
    preserve = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base: ❷
            continue # 忽略拉丁基字符的变音符
        preserve.append(c) ❸
        # 如果不是组合字符，那就是新的基字符
        if not unicodedata.combining(c): ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(preserve)
    return unicodedata.normalize('NFC', shaved) ❺

```

- ❶ 把所有字符分解成基字符和组合记号。
- ❷ 基字符为拉丁字符时，跳过组合记号。
- ❸ 否则，保存当前字符。
- ❹ 检测新的基字符，判断是不是拉丁字符。
- ❺ 重组所有字符。

规范化步骤还可以更彻底，把西文文本中的常见符号（例如弯引号、长破折号、项目符号等）替换成 ASCII 中的对等字符。示例 4-17 中的 `asciize` 函数就是这么做的。

示例 4-17 把一些西文印刷字符转换成 ASCII 字符（这个代码片段也是示例 4-14 中 `sanitize.py` 模块的一部分）

```

single_map = str.maketrans(" ", f„^<'!""•--~""", ❶
                             """"'f"^^<'!""---~>""")

multi_map = str.maketrans({ ❷
    '€': 'EUR',
    '...': '...',
    'Æ': 'AE',
    'æ': 'ae',
    'OE': 'OE',

```

```

    'oe': 'oe',
    'TM': ' (TM) ',
    '%': '<per mille>',
    '†': '**',
    '‡': '***',
})

multi_map.update(single_map) ❸

def dewinize(txt):
    """把cp1252符号替换为ASCII字符或字符序列"""
    return txt.translate(multi_map) ❹

def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt)) ❺
    no_marks = no_marks.replace('ß', 'ss') ❻
    return unicodedata.normalize('NFKC', no_marks) ❼

```

❶ 构建“字符到字符”替换映射表。

❷ 构建“字符到字符串”替换映射表。

❸ 合并两个映射表。

❹ `dewinize` 函数不影响 ASCII 或 `latin1` 文本，只替换 Microsoft 在 `cp1252` 中为 `latin1` 额外添加的字符。

❺ 调用 `dewinize` 函数，再去掉变音符。

❻ 把德语 Eszett ('ß') 替换成“ss”（这里没有做大小写同一化，因为我们想保留大小写）。

❼ 使用 NFKC 规范化形式把字符和与之兼容的码点组合起来。

示例 4-18 演示了 `asciize` 函数的效果。

示例 4-18 示例 4-17 中 `asciize` 函数的使用示例

```

>>> order = '"Herr Voß: • ½ cup of OEtker™ caffè latte • bowl of
açai."'

```

```
>>> dewinize(order)
'"Herr Voß: - ½ cup of OEtker(TM) caffè latte - bowl of açaí."' ❶
>>> asciize(order)
'"Herr Voss: - 1/2 cup of OEtker(TM) caffè latte - bowl of acai."'
❷
```

❶ dewinize 函数替换弯引号、项目符号和™（商标符号）。

❷ asciize 函数调用 dewinize 函数，不仅去掉变音符，还替换了 'ß'。



不同语言删除变音符的规则不一样。例如，德语把 'ü' 变成 'ue'。我们定义的 asciize 函数没有考虑这么多，可能适合你的语言，也可能不适合。不过，对葡萄牙语的处理是可以接受的。

综上，simplify.py 中的函数做的事情超出了标准的规范化，而且会对文本做进一步处理，很有可能改变原意。只有知道目标语言、目标用户群和转换后的用途，才能确定要不要做这么深入的规范化。

我们对 Unicode 文本规范化的讨论到此结束。

接下来探讨 Unicode 文本排序问题。

4.8 Unicode 文本排序

给任何类型的序列排序，Python 都会逐一比较序列中的每一项。对字符串来说，比较的是码点。可是，一旦遇到非 ASCII 字符，结果就往往不尽如人意。

下面对一个巴西产水果的列表进行排序。

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```


不同区域采用的排序规则有所不同，葡萄牙语等许多语言按照拉丁字母表排序，重音符和下加符对排序几乎没什么影响。⁸ 因此，排序时，“cajá”视作“caja”，必然排在“caju”前面。

⁸变音符对排序有影响的情况很少发生，只有两个词之间唯有变音符不同时才有影响。此时，带变音符的词排在常规词的后面。

排序后的 `fruits` 列表应该是下面这样。

```
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

在 Python 中，非 ASCII 文本的标准排序方式是使用 `locale.strxfrm` 函数。根据 `locale` 模块的文档，这个函数“把字符串转换成适合所在区域进行比较的形式”。

使用 `locale.strxfrm` 函数之前，必须先为应用设定合适的区域设置，还要祈祷操作系统支持这项设置。示例 4-19 中的命令也许可以做到这一点。

示例 4-19 `locale_sort.py`：把排序键设为 `locale.strxfrm` 函数

```
import locale
my_locale = locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
print(my_locale)
fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
sorted_fruits = sorted(fruits, key=locale.strxfrm)
print(sorted_fruits)
```

在区域设置为 `pt_BR.UTF-8` 的 GNU/Linux (Ubuntu 19.10) 中运行示例 4-19，得到的结果是正确的。

```
'pt_BR.UTF-8'
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

因此，使用 `locale.strxfrm` 函数做排序键之前，要调用 `setlocale(LC_COLLATE, «your_locale»)`。

不过，有几个问题需要注意。

- 区域设置全局生效，因此不建议在库中调用 `setlocale` 函数。应用程序或框架应该在启动进程时设定区域，而且此后不要再修改。
- 操作系统必须支持你设定的区域，否则 `setlocale` 函数会抛出 `locale.Error: unsupported locale setting` 异常。
- 你必须知道如何拼写区域名称。
- 操作系统制造商必须正确实现你设定的区域。我在 Ubuntu 19.10 中成功了，但是在 macOS 10.14 中失败了。在 macOS 中，`setlocale(LC_COLLATE, 'pt_BR.UTF-8')` 调用返回字符串 `'pt_BR.UTF-8'`，没有报错。但是，`sorted(fruits, key=locale.strxfrm)` 的结果与 `sorted(fruits)` 一样，也是错的。我在 macOS 中也试过 `fr_FR`、`es_ES` 和 `de_DE` 等区域，`locale.strxfrm` 均未生效。⁹

⁹同样，我没找到解决方案，不过我发现其他人也报告了同样的问题。本书技术审校之一 Alex Martelli 在他装有 macOS 10.9 的 Macintosh 中使用 `setlocale` 和 `locale.strxfrm` 没有任何问题。可见，结果因人而异。

因此，标准库提供的国际化排序方案有一定效果，但是似乎只对 GNU/Linux 有良好支持（可能也支持 Windows，但你得是专家）。即便如此，还是要依赖区域设置，而这会为部署带来麻烦。

幸好，有一个较为简单的方案可用，即 PyPI 中的 `pyuca` 库。

使用 Unicode 排序算法排序

James Tauber，一位高产的 Django 贡献者，他一定是感受到了这个痛点，因此开发了 `pyuca` 库，单纯使用 Python 实现了 Unicode 排序算法（Unicode Collation Algorithm, UCA）。通过示例 4-20 可以看到这个库的用法是多么简单。

示例 4-20 使用 `pyuca.Collator.sort_key` 方法

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açai', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

这样做简单多了，而且在 GNU/Linux、macOS 和 Windows 中都能正确排序——至少这个简短的示例是正确的。

pyuca 不考虑区域设置。如果你想自定义排序方式，那么可以把自定义的排序表路径传给 `Collator()` 构造函数。pyuca 默认使用项目自带的 `allkeys.txt`，这是 Unicode 网站中默认 Unicode 排序元素表的副本。



Miro 建议使用 PyICU 排序 Unicode 文本

(技术审校 Miroslav Šedivý 是 Unicode 专家，通晓多种语言。下面是他对 pyuca 的评价。)

pyuca 有一种排序算法不考虑个别语言的字母顺序。例如，德语中的 Ä 位于 A 和 B 之间，在瑞典语中 Ä 却位于 Z 之后。这种情况下建议使用 PyICU，这个库像区域设置一样牢靠，但不更改进程使用的区域设置。如果你想更改土耳其语中 il/II 的大小写，也需要使用 PyICU。PyICU 中有一个扩展必须编译，因此在某些系统中，可能比单纯使用 Python 的 pyuca 难安装。

顺便说一下，那个排序表是 Unicode 数据库中众多数据文件中的一个。4.9 节将讨论 Unicode 数据库。

4.9 Unicode 数据库

Unicode 标准提供了一个完整的数据库（许多结构化文本文件），不仅包括码点与字符名称之间的映射表，还包括各个字符的元数据，以

及字符之间的关系。例如，Unicode 数据库记录了字符是否可以打印、是不是字母、是不是数字，或者是不是其他数值符号。`str` 的 `isalpha`、`isprintable`、`isdecimal` 和 `isnumeric` 等方法就是靠这些信息来判断的。`str.casefold` 方法也使用一个 Unicode 表中的信息。



`unicodedata.category(char)` 函数返回 `char` 在 Unicode 数据库中的类别（以两个字母表示）。判断类别使用高层级的 `str` 方法更简单。例如，如果 `label` 中的每个字符都属于 `Lm`、`Lt`、`Lu`、`Ll` 或 `Lo` 类别，则 `label.isalpha()` 返回 `True`。

4.9.1 按名称查找字符

`unicodedata` 模块中有几个函数用于获取字符的元数据。例如，`unicodedata.name()` 返回一个字符在标准中的官方名称，如图 4-5 所示。¹⁰

¹⁰这是一个图像，而不是代码清单，因为我写这本书时，O'Reilly 的数字出版工具链对表情符号的支持尚不完善。

```
>>> from unicodedata import name
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> name('ã')
'LATIN SMALL LETTER A WITH TILDE'
>>> name('♚')
'BLACK CHESS QUEEN'
>>> name('😺')
'GRINNING CAT FACE WITH SMILING EYES'
```

图 4-5：在 Python 控制台中探索 `unicodedata.name()`

你可以利用 `name()` 函数构建一个应用程序，让用户通过名称搜索字符。图 4-6 展示了命令行脚本 `cf.py` 的效果。该脚本的参数为一个或

多个单词，列出 Unicode 官方名称中带有这些单词的字符。cf.py 脚本的源码在示例 4-21 中。

```
$ ./cf.py cat smiling
U+1F638 🐱 GRINNING CAT FACE WITH SMILING EYES
U+1F63A 🐱 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 🐱 SMILING CAT FACE WITH HEART-SHAPED EYES
```

图 4-6：使用 cf.py 脚本查找微笑猫表情



不同操作系统和不同应用程序对表情符号的支持差异很大。近几年，macOS 终端对表情符号的支持最好，其次是 GNU/Linux 的现代化图形终端。在 Windows 中，cmd.exe 和 PowerShell 现在支持 Unicode 输出，但我在 2020 年 1 月写这一节时，仍然不能显示表情符号——至少不是“开箱即用”。本书技术审校 Leonardo Rochael 告诉我，Windows 发布了全新开源的 Windows Terminal，对 Unicode 的支持可能比陈旧的 Microsoft 控制台要更好。我还没来得及试试。

注意，在示例 4-21 中，find 函数内的 if 语句使用 .issubset() 方法快速测试 query 集合中的所有单词是否出现在根据字符名称构建的单词列表中。得益于 Python 丰富的集合 API，我们不用嵌套 for 循环，再加上一个 if 语句检查。

示例 4-21 cf.py：字符查找实用脚本

```
#!/usr/bin/env python3
import sys
import unicodedata

START, END = ord(' '), sys.maxunicode + 1 ❶

def find(*query_words, start=START, end=END): ❷
    query = {w.upper() for w in query_words} ❸
    for code in range(start, end):
        char = chr(code) ❹
        name = unicodedata.name(char, None) ❺
```

```
        if name and query.issubset(name.split()): ❸
            print(f'U+{code:04X}\t{char}\t{name}') ❹

def main(words):
    if words:
        find(*words)
    else:
        print('Please provide words to find.')

if __name__ == '__main__':
    main(sys.argv[1:])
```

❶ 设置默认搜索的码点区间。

❷ `find` 函数接受多个查询词 (`query_words`)，以及两个可选的关键字参数，限制搜索区间，便于测试。

❸ 把 `query_words` 转换成一个全是大写字母的字符串集合。

❹ 获取 `code` 对应的 Unicode 字符。

❺ 获取字符的名称。如果码点不对应任何字符，则返回 `None`。

❻ 如果有名称，则把名称拆分为单词列表，然后检查 `query` 集合是不是该列表的子集。

❼ 打印 `U+9999` 格式的码点、字符和字符名称。

`unicodedata` 模块还有一些有趣的函数。4.9.2 节会介绍其中的几个函数，用于从具有数值意义的字符中获取信息。

4.9.2 字符的数值意义

`unicodedata` 模块中有几个函数可以检查 Unicode 字符是不是表示数值，如果是的话，还能确定人类可读的具体数值，而不是码点数。

示例 4-22 演示了 `unicodedata.name()` 和

unicodedata.numeric() 函数, 以及 str 的 .isdecimal() 和 .isnumeric() 方法。

示例 4-22 Unicode 数据库中数值字符的元数据示例 (各个标号说明输出中的各列)

```
import unicodedata
import re

re_digit = re.compile(r'\d')
sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print(f'U+{ord(char):04x}',          ❶
          char.center(6),                ❷
          're_digit' if re_digit.match(char) else '-', ❸
          'isdigit' if char.isdigit() else '-',        ❹
          'isnum' if char.isnumeric() else '-',        ❺
          f'{unicodedata.numeric(char):5.2f}',          ❻
          unicodedata.name(char),              ❼
          sep='\t')
```

- ❶ U+0000 格式的码点。
- ❷ 在长度为 6 的字符串中居中显示字符。
- ❸ 如果字符匹配正则表达式 `r'\d'`, 则显示 `re_digit`。
- ❹ 如果 `char.isdigit()` 返回 `True`, 则显示 `isdigit`。
- ❺ 如果 `char.isnumeric()` 返回 `True`, 则显示 `isnum`。
- ❻ 使用长度为 5、小数点后保留 2 位的浮点数显示数值。
- ❼ Unicode 标准中字符的名称。

如果你的终端使用的字体支持所有这些字符的字形, 则运行示例 4-22 得到的结果如图 4-7 所示。

```
$ python3 numerics_demo.py
U+0031 1 re_dig isdig isnum 1.00 DIGIT ONE
U+00bc ¼ - - isnum 0.25 VULGAR FRACTION ONE QUARTER
U+00b2 ² - isdig isnum 2.00 SUPERSCRIPT TWO
U+0969 ३ re_dig isdig isnum 3.00 DEVANAGARI DIGIT THREE
U+136b ፫ - isdig isnum 3.00 ETHIOPIC DIGIT THREE
U+216b XII - - isnum 12.00 ROMAN NUMERAL TWELVE
U+2466 ⑦ - isdig isnum 7.00 CIRCLED DIGIT SEVEN
U+2480 ⑬ - - isnum 13.00 PARENTHEZIZED NUMBER THIRTEEN
U+3285 ㊼ - - isnum 6.00 CIRCLED IDEOGRAPH SIX
$
```

图 4-7: macOS 终端中显示的数值字符及其元数据; `re_dig` 表示字符匹配正则表达式 `r'\d'`

图 4-7 中的第 6 列是在字符上调用 `unicodedata.numeric(char)` 函数得到的结果。这表明, Unicode 知道表示数字的符号的数值。因此, 如果你想创建一个支持泰米尔数字和罗马数字的电子表格应用程序, 那就放心去做吧!

图 4-7 表明, 正则表达式 `r'\d'` 能匹配阿拉伯数字 1 和梵文数字 3, 但是不能匹配 `isdigit` 方法判断为数字的其他字符。可见, `re` 模块对 Unicode 的支持并不充分。PyPI 中有个新开发的 `regex` 模块, 它的目标是最终取代 `re` 模块, 提供更好的 Unicode 支持。¹¹4.10 节将回过头来讨论 `re` 模块。

¹¹不过对这个示例来说, `regex` 模块在识别数字方面的表现并不比 `re` 模块更好。

本章用到了 `unicodedata` 模块中的几个函数, 但是还有很多没有涵盖。请阅读 `unicodedata` 模块的文档进一步学习。

接下来的内容会简要说明双模式 API。这种 API 提供的函数, 接受的参数既可以是 `str`, 也可以是 `bytes`, 具体如何处理则根据参数的类型而定。

4.10 支持 `str` 和 `bytes` 的双模式 API

Python 标准库中的一些函数能接受 `str` 或 `bytes` 为参数，根据其具体类型展现不同的行为。`re` 和 `os` 模块中就有这样的函数。

4.10.1 正则表达式中的 `str` 和 `bytes`

如果使用 `bytes` 构建正则表达式，则 `\d` 和 `\w` 等模式只能匹配 ASCII 字符；相比之下，如果是 `str` 模式，那就能匹配 ASCII 之外的 Unicode 数字或字母。示例 4-23 和图 4-8 展示了 `str` 和 `bytes` 模式对字母、ASCII 数字、上标和泰米尔数字的匹配情况。

示例 4-23 `ramanujan.py`: 比较简单的 `str` 和 `bytes` 正则表达式的行为

```
import re

re_numbers_str = re.compile(r'\d+')           ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+')       ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef"  ❸
            " as 1729 = 13 + 123 = 93 + 103." )      ❹

text_bytes = text_str.encode('utf_8')         ❺

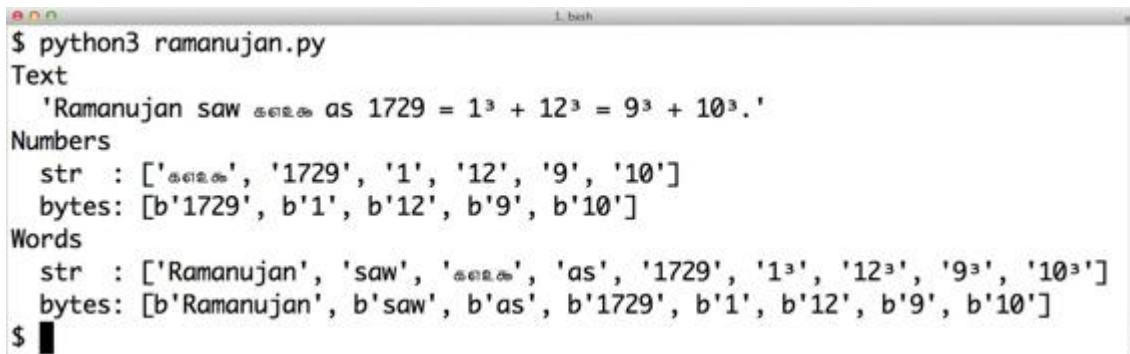
print(f'Text\n {text_str!r}')
print('Numbers')
print('  str  :', re_numbers_str.findall(text_str))      ❻
print('  bytes:', re_numbers_bytes.findall(text_bytes))  ❼
print('Words')
print('  str  :', re_words_str.findall(text_str))        ❽
print('  bytes:', re_words_bytes.findall(text_bytes))    ❾
```

❶ 前两个正则表达式是 `str` 类型。

❷ 后两个正则表达式是 `bytes` 类型。

❸ 要搜索的 Unicode 文本，包括 1729 的泰米尔数字（逻辑行直到右括号才结束）。

- ④ 这个字符串在编译时与前一个拼接起来（见《Python 语言参考手册》中的“2.4.2. 字符串字面值合并”）。
- ⑤ `bytes` 正则表达式只能搜索 `bytes` 字符串。
- ⑥ `str` 模式 `r'\d+'` 能匹配泰米尔数字和 ASCII 数字。
- ⑦ `bytes` 模式 `rb'\d+'` 只能匹配 ASCII 字节中的数字。
- ⑧ `str` 模式 `r'\w+'` 能匹配字母、上标、泰米尔数字和 ASCII 数字。
- ⑨ `bytes` 模式 `rb'\w+'` 只能匹配 ASCII 字节中的字母和数字。



```
$ python3 ramanujan.py
Text
'Ramanujan saw க௭௭௭௭ as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
str : ['க௭௭௭௭', '1729', '1', '12', '9', '10']
bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
str : ['Ramanujan', 'saw', 'க௭௭௭௭', 'as', '1729', '1³', '12³', '9³', '10³']
bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$
```

图 4-8：运行示例 4-23 中的 `ramanujan.py` 脚本得到的结果截图

示例 4-23 是随便举的例子，目的是说明一个问题：使用正则表达式可以搜索 `str` 和 `bytes`，但是在后一种情况下，ASCII 范围外的字节序列不会被当成数字和组成单词的字符。

`str` 正则表达式有个 `re.ASCII` 标志，能让 `\w`、`\W`、`\b`、`\B`、`\d`、`\D`、`\s` 和 `\S` 只匹配 ASCII 字符。详见 `re` 模块的文档。

另一个重要的双模式模块是 `os`。

4.10.2 `os` 函数中的 `str` 和 `bytes`

GNU/Linux 内核不理解 Unicode，因此你可能会遇到一些文件名，其中的字节序列对任何合理的编码方案来说都是无效的，不能解码成 `str`。如果你使用客户端连接不同操作系统中的文件服务器，那就尤其要注意这个问题。

为了规避这个问题，`os` 模块中所有接受文件名或路径名的函数，既可以传入 `str` 参数，也可以传入 `bytes` 参数。传入 `str` 参数时，使用 `sys.getfilesystemencoding()` 获得的编码解码器自动转换参数，操作系统回显时也使用该编码解码器解码。这几乎就是我们想要的行为，与 Unicode 三明治最佳实践一致。

但是，如果必须处理（可能是为了修正）那些无法使用上述方式自动处理的文件名，则可以把 `bytes` 参数传给 `os` 模块中的函数，得到 `bytes` 类型的返回值。如此一来，便可以处理任何文件名或路径名，不管里面有多少鬼符，如示例 4-24 所示。

示例 4-24 分别把 `str` 和 `bytes` 参数传给 `listdir` 函数，看看得到的结果

```
>>> os.listdir('.') ❶
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.') ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

❶ 第 2 个文件名是“digits-of-π.txt”（有一个希腊字母π）。

❷ 参数是 `byte` 类型，`listdir` 函数返回的文件名是字节序列，其中 `b'\xcf\x80'` 是希腊字母π的 UTF-8 编码。

为了便于手动处理 `str` 或 `bytes` 类型的文件名或路径名，`os` 模块提供了特殊的编码解码函数 `os.fsencode(name_or_path)` 和 `os.fsdecode(name_or_path)`。这两个函数接受的参数可以是 `str` 或 `bytes` 类型，自 Python 3.6 起，还可以是实现了 `os.PathLike` 接口的对象。

Unicode 话题深似海，我们对 `str` 和 `bytes` 的探索暂且告一段落。

4.11 本章小结

本章首先澄清了人们对一个字符等于一个字节的误解。随着 Unicode 的广泛使用，我们必须把文本字符串与它们在文件中的二进制序列表述区分开，而且这是 Python 3 强制要求区分的。

对 `bytes`、`bytearray` 和 `memoryview` 等二进制序列数据类型做了简要概述之后，我们转到了编码和解码话题，通过示例展示了重要的编码解码器，随后讨论了如何避免和处理臭名昭著的 `UnicodeEncodeError` 和 `UnicodeDecodeError`，以及由于 Python 源码文件编码错误导致的 `SyntaxError`。

然后，我们说明了在没有元数据的情况下检测编码的理论和实际情况：理论上，做不到这一点；但实际上，`Chardet` 包能够正确处理一些流行的编码。随后介绍了字节序标记，这是 UTF-16 和 UTF-32 文件中常见的编码提示，某些 UTF-8 文件中也有。

接下来的 4.6 节演示了如何打开文本文件，这是一项简单的任务，不过有个陷阱：打开文本文件时，`encoding=` 关键字参数不是必需的，但是应该指定。如果没有指定编码，那么程序会想方设法生成“纯文本”，如此一来，不一致的默认编码就会导致跨平台不兼容性。然后，我们说明了 Python 使用的几个默认编码设置，以及检测方法。对 Windows 用户来说，现实不容乐观：这些设置在同一台设备中往往有不同的值，而且各个设置相互不兼容。而对 GNU/Linux 和 macOS 用户来说，情况就好多了，几乎所有地方使用的默认编码都是 UTF-8。

Unicode 为某些字符提供了不同的表示，匹配文本之前一定要先规范化。说明规范化和大小写同一化之后，我给出了几个实用函数，你可以根据自己的需求改编。其中有个函数所做的转换较为极端，比如会去掉所有重音符。随后，我们说明了如何使用标准库中的 `locale` 模

块正确排序 Unicode 文本（有一些注意事项）。此外，还可以使用外部包 `pyuca`，由此摆脱对捉摸不定的区域配置的依赖。

最后，我们利用 Unicode 数据库编写了一个命令行实用脚本，按名称搜索字符。得益于 Python 强大的功能，这个脚本只有 28 行代码。我们还简单介绍了 Unicode 元数据，简要说明了双模式 API。双模式 API 提供的函数，根据传入的参数是 `str` 还是 `bytes` 类型，会产生不同的结果。

4.12 延伸阅读

Ned Batchelder 在 2012 年 PyCon US 上所做的演讲非常出色，题为“Pragmatic Unicode, or, How Do I Stop the Pain?”。Ned 很专业，除了幻灯片和视频之外，他还提供了完整的文字记录。

Esther Nam 和 Travis Fischer 在 PyCon 2014 上做了一场精彩的演讲，题为“Character encoding and Unicode in Python: How to (√ ° □ °) √ ——— with dignity”。本章开头那句简短有力的话就是出自这次演讲：“文本给人类阅读，字节序列供计算机处理。”

本书第 1 版技术审校之一 Lennart Regebro 在“Unconfusing Unicode: What Is Unicode?”这篇短文中提出了“Useful Mental Model of Unicode (UMMU)”这一概念。Unicode 是个复杂的标准，Lennart 提出的 UMMU 是个很好的切入点。

Python 文档中的“Unicode HOWTO”一文从几个不同的角度对本章涉及的话题做了讨论，涵盖历史简介、句法细节、编码解码器、正则表达式、文件名和 Unicode 的 I/O 最佳实践（即 Unicode 三明治），而且每一节都给出了大量参考资料链接。*Dive into Python 3* (Mark Pilgrim 著) 是一本非常优秀的书，第 4 章讲到了 Python 3 对 Unicode 的支持，内容翔实。此外，该书第 15 章说明了 Chardet 库从 Python 2 移植到 Python 3 的过程。这是一个宝贵的案例分析，从

中可以看出，从旧的 `str` 类型转到新的 `bytes` 类型是造成迁移如此痛苦的主要原因，也是检测编码的库应该关注的重点。

如果你用过 Python 2，但是刚接触 Python 3，可以阅读 Guido van Rossum 写的“[What's New in Python 3.0](#)”。这篇文章简要列出了新版的 15 点变化。Guido 开门见山地说道：“你自以为知道的二进制数据和 Unicode 知识全都变了。”Armin Ronacher 的博客文章“[The Updated Guide to Unicode on Python](#)”深入分析了 Python 3 中 Unicode 的一些陷阱（Armin 不太喜欢 Python 3）。

《Python Cookbook 中文版（第 3 版）》的第 2 章“字符串和文本”中有几个经典实例谈到了 Unicode 规范化、文本清洗，以及在字节序列上执行面向文本的操作。第 5 章涵盖文件和 I/O，“5.17 将字节数据写入文本文件”指出，任何文本文件的底层都有一个二进制流，如果需要可以直接访问。之后，“6.11 读写二进制结构的数组”用到了 `struct` 模块。

Nick Coghlan 的“[Python Notes](#)”博客中有两篇文章与本章的话题联系紧密：“[Python 3 and ASCII Compatible Binary Protocols](#)”和“[Processing Text Files in Python 3](#)”。强烈推荐阅读。

Python 支持的编码列表参见 `codecs` 模块文档中的“[Standard Encodings](#)”一节。如果需要通过编程方式获得那个列表，请看 CPython 源码中的 `/Tools/unicode/listcodecs.py` 脚本是怎么做的。

Unicode Explained（Jukka K. Korpela 著）和 *Unicode Demystified*（Richard Gillam 著）这两本书不是针对 Python 的，但是在我学习 Unicode 相关概念时给了我很大的帮助。*Programming with Unicode*（Victor Stinner 著）是一本自出版图书，可免费阅读（Creative Commons BY-SA），其中讨论了 Unicode 一般性话题，还介绍了主流操作系统和几门编程语言（包括 Python）相关的工具和 API。

W3C 网站中的“[Case Folding: An Introduction](#)”和“[Character Model for the World Wide Web: String Matching](#)”页面讨论了规范化相关的概

念，前一篇是介绍性文章，后一篇则是以枯燥的标准用语写就的工作草案——“Unicode Standard Annex #15—Unicode Normalization Forms”也是这种风格。Unicode 网站中的“Frequently Asked Questions, Normalization”更容易理解，Mark Davis 写的“NFC FAQ”也不错。Mark 是多个 Unicode 算法的作者，写作本书时，他还担任 Unicode 联盟主席。

2016 年，纽约现代艺术博物馆（Museum of Modern Art, MoMA）收藏了最初的 176 个表情符号。这些表情符号是栗田穰崇在 1999 年为日本移动运营商 NTT DOCOMO 设计的。根据 Emojipedia 网站中的“Correcting the Record on the First Emoji Set”一文，表情符号的历史还可以追溯到更早的时期——1997 年日本 SoftBank 公司最先在手机中部署了一套表情包。SoftBank 的那套表情包有 90 个表情符号，现已纳入 Unicode，例如 U+1F4A9 (PILE OF POO)。Matthew Rothenberg 创建的 emoji tracker 网站实时更新 Twitter 上表情的使用量。在我写下这段话时，Twitter 最流行的表情符号是 FACE WITH TEARS OF JOY (U+1F602)，使用量超过 3 313 667 315 次。

杂谈

在源码中应该使用非 ASCII 名称吗

Python 3 允许在源码中使用非 ASCII 标识符。

```
>>> ação = 'PBR' # ação = stock
>>> ε = 10**-6   # ε = epsilon
```

有些人不喜欢这样做。坚持使用 ASCII 标识符的最常见理由是，让每个人都能轻松地阅读和编辑代码。这种观点没有抓住要点——观点的持有者希望的是源码对于目标群体是可读的和可编辑的，而不是“所有人”。在一家跨国企业中，或者一个开源项目中，如果希望世界各地的人都能共享代码，那么标识符推荐使用英语，因此也要使用 ASCII 字符。

然而，如果你是巴西的一名教师，你的学生更喜欢阅读用葡萄牙语命名的变量和函数（当然，拼写要正确），那么使用本地化键盘可以让他们轻松地输入变音符和重读元音。

既然 Python 可以解析 Unicode 名称，而且现在源码默认使用 UTF-8 编码，那么我认为没有必要像过去在 Python 2 中那样，用不带重音符的葡萄牙语编写标识符，除非你也需要使用 Python 2 运行代码。如果使用葡萄牙语命名，却省略重音符，那么对任何人来说，代码都不可能更易于阅读。

这是我作为一个说葡萄牙语的巴西人的观点，不过我相信这个道理是无国界的：任何人都应该选择能让团队成员更容易理解代码的语言，并使用正确的字符拼写。

“纯文本”是什么

不经常处理英语文本的人，往往误认为“纯文本”指的是“ASCII”。Unicode 词汇表是这样定义纯文本的：

只由特定标准的码点序列组成的计算机编码文本，不含其他格式化或结构化信息。

这个定义的前半句说得很好，但是后半句我不认同。HTML 就包含格式化和结构化信息，但它依然是纯文本，因为 HTML 文件中的每个字节都表示一个文本字符（通常使用 UTF-8 编码），没有任何字节表示文本之外的信息。.png 或 .xsl 文档则不同，其中多数字节表示打包的二进制值，例如 RGB 值和浮点数。在纯文本中，数字使用数字符号序列表示。

本书英文版使用一种名为 AsciiDoc 的纯文本格式撰写

（把“Ascii”和“Doc”放在一起，有点讽刺），它是 O'Reilly 优秀的图书出版平台 Atlas 工具链中的一部分。AsciiDoc 源文件是纯文本，但用的是 UTF-8 编码，而不是 ASCII。不然，撰写本章必定痛苦不堪。姑且不论名称，AsciiDoc 是一个很棒的工具。¹²

Unicode 世界正在不断扩张，但是有些边缘场景缺少支持工具。比如，我想使用的字符在本书使用的字体中就不一定有。因此，本章有好几个代码示例用图像代替了。不过，Ubuntu 和 macOS 的终端能正确显示多数 Unicode 文本，包括“mojibake”（文字化け）这个日语词。

str 的码点在 RAM 中如何表示

Python 官方文档对 `str` 的码点在内存中如何存储避而不谈。毕竟，这是实现细节。理论上，怎么存储都没关系，不管内部表述如何，输出时每个 `str` 都要编码成 `bytes`。

在内存中，Python 3 使用固定数量的字节存储 `str` 中的各个码点，以便高效访问任何字符或切片。

从 Python 3.3 起，创建 `str` 对象时，解释器会检查里面的字符，选择最经济的内存布局：如果字符都在 `latin1` 字符集中，则使用 1 个字节存储一个码点；否则，根据字符串中的具体字符，选择 2 个或 4 个字节存储一个码点。这是简要说明，完整细节请参阅“PEP 393—Flexible String Representation”。

Python 3 对 `int` 类型的处理方式也像字符串表述一样灵活：如果一个整数在一个机器字中放得下，那就存储在一个机器字中；否则，解释器采用变长表述，类似于 Python 2 中的 `long` 类型那样。这种聪明的做法得到推广，真是让人欣喜！

然而，对于 Python 3，Armin Ronacher 有话要说。他向我解释了这样做在实践中为什么不好：在一个原本全是 ASCII 字符的文本中添加一个 `RAT` 字符（U+1F400），内存中存储各个字符的数组会立刻变大。原来，每个字符只占 1 字节，而现在全占 4 字节。此外，由于 Unicode 字符能以各种方式组合，按位置检索字符就没那么容易了，从 Unicode 文本中提取切片也没有想象中那么简单，而且结果往往是错的，会产生乱码。随着表情符号的流行，这些问题只会越来越严重。

¹²本书译稿也用 AsciiDoc 撰写，然后转换成图灵社区使用的 Markdown 格式。Markdown 源文件也是纯文本。——译者注

第 5 章 数据类构建器

数据类就像小孩子。作为一个起点很好，但若要让它们像成熟的对象那样参与整个系统的工作，它们就必须承担一定责任。

——Martin Fowler 和 Kent Beck¹

¹《重构：改善既有代码的设计》（后面简称《重构》），第 3 章，3.20 节。

Python 提供了几种构建简单类的方式，这些类只是字段的容器，几乎没有额外功能。这种模式称为“数据类”（data class），`dataclasses` 包就支持该模式。本章介绍以下 3 个可简化数据类构建过程的类构建器。

```
collections.namedtuple
```

最简单的构建方式，从 Python 2.6 开始提供。

```
typing.NamedTuple
```

另一种构建方式，需要为字段添加类型提示，从 Python 3.5 开始提供。`class` 句法在 Python 3.6 中增加。

```
@dataclasses.dataclass
```

一个类装饰器，与前两种方式相比，可定制的内容更多，增加了大量选项，可实现更复杂的功能，从 Python 3.7 开始提供。

介绍这些类构建器之后，接下来将讨论为什么**数据类**模式也是一种代码异味，它的出现可能意味着面向对象设计欠佳。



`typing.TypedDict` 与 `typing.NamedTuple` 句法相似，而且在 Python 3.9 的 `typing` 模块文档中，二者紧挨在一

起。这让人感觉，`typing.TypedDict` 也是一种数据类构建器。

然而，`TypedDict` 不能构建可以实例化的具体类。`TypedDict` 只提供编写类型提示的句法，为把映射值用作记录（键是字段名称）的函数参数和变量注解类型。详见 15.3 节。

5.1 本章新增内容

本章是第 2 版新增的。5.3 节原来在第 1 版第 2 章中，除此之外，本章都是新内容。

首先概述 3 个类构建器。

5.2 数据类构建器概述

示例 5-1 是一个简单的类，表示地理位置的经纬度。

示例 5-1 class/coordinates.py

```
class Coordinate:

    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon
```

`Coordinate` 类的作用是保存经纬度属性。为 `__init__` 方法编写样板代码容易让人感到枯燥，尤其是属性较多的时候。想想看，每一个属性都要写 3 次。更糟的是，样板代码并没有给我们提供 Python 对象都有的基本功能。

```
>>> from coordinates import Coordinate
>>> moscow = Coordinate(55.76, 37.62)
>>> moscow
<coordinates.Coordinate object at 0x107142f10> ❶
>>> location = Coordinate(55.76, 37.62)
```

```
>>> location == moscow ❷
False
>>> (location.lat, location.lon) == (moscow.lat, moscow.lon) ❸
True
```

❶ 继承自 `object` 的 `__repr__` 作用不大。

❷ `==` 没有意义，因为继承自 `object` 的 `__eq__` 方法比较对象的 ID。

❸ 想比较两个地理位置的经纬度，只能——比较各个属性。

本章要讲的数据类构建器自动提供必要的 `__init__`、`__repr__` 和 `__eq__` 等方法，此外还有其他有用的功能。



本章讨论的类构建器都不依赖继承。

`collections.namedtuple` 和 `typing.NamedTuple` 构建的类都是 `tuple` 的子类。`@dataclass` 是类装饰器，不影响类层次结构。这 3 个类构建器使用不同的元编程技术把方法和数据属性注入要构建的类。

下面使用 `namedtuple` 构建 `Coordinate` 类。`namedtuple` 是一个工厂方法，使用指定的名称和字段构建 `tuple` 的子类。

```
>>> from collections import namedtuple
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
>>> issubclass(Coordinate, tuple)
True
>>> moscow = Coordinate(55.756, 37.617)
>>> moscow
Coordinate(lat=55.756, lon=37.617) ❶
>>> moscow == Coordinate(lat=55.756, lon=37.617) ❷
True
```

❶ 有用的 `__repr__`。

❷ 有意义的 `__eq__`。

新出现的 `typing.NamedTuple` 具有一样的功能，不过可为各个字段添加类型注解。

```
>>> import typing
>>> Coordinate = typing.NamedTuple('Coordinate',
...     [('lat', float), ('lon', float)])
>>> isinstance(Coordinate, tuple)
True
>>> typing.get_type_hints(Coordinate)
{'lat': <class 'float'>, 'lon': <class 'float'>}
```



构建带类型的具名元组，也可以通过关键字参数指定字段，如下所示。

```
Coordinate = typing.NamedTuple('Coordinate', lat=float,
lon=float)
```

这种方式可读性高，而且可以通过映射指定字段及其类型，再使用 `**fields_and_types` 拆包。

从 Python 3.6 开始，`typing.NamedTuple` 也可以在 `class` 语句中使用，类型注解按“PEP 526—Syntax for Variable Annotations”标准编写。这样写出的代码可读性更高，而且方便覆盖方法或添加新方法。示例 5-2 再次定义了 `Coordinate` 类，经纬度属性均为 `float` 类型，同时自定义了 `__str__` 方法，以 `55.8°N, 37.6°E` 的格式显示坐标经纬度。

示例 5-2 `typing_namedtuple/coordinates.py`

```
from typing import NamedTuple

class Coordinate(NamedTuple):
    lat: float
    lon: float

    def __str__(self):
        ns = 'N' if self.lat >= 0 else 'S'
        we = 'E' if self.lon >= 0 else 'W'
```

```
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°  
{we}'
```



在 `class` 语句中，虽然 `NamedTuple` 出现在超类的位置上，但其实它不是超类。`typing.NamedTuple` 使用元类²这一高级功能创建用户类。不信，请看下面的代码片段。

```
>>> issubclass(Coordinate, typing.NamedTuple)  
False  
>>> issubclass(Coordinate, tuple)  
True
```

²元类将在第 24 章探讨。

在 `typing.NamedTuple` 生成的 `__init__` 方法中，字段参数的顺序与在 `class` 语句中出现的顺序相同。

与 `typing.NamedTuple` 一样，`dataclass` 装饰器也支持使用 PEP 526 句法来声明实例属性。`dataclass` 装饰器读取变量注解，自动为构建的类生成方法。示例 5-3 使用 `dataclass` 装饰器再次定义 `Coordinate` 类，你可以比较一下。

示例 5-3 dataclass/coordinates.py

```
from dataclasses import dataclass  
  
@dataclass(frozen=True)  
class Coordinate:  
    lat: float  
    lon: float  
  
    def __str__(self):  
        ns = 'N' if self.lat >= 0 else 'S'  
        we = 'E' if self.lon >= 0 else 'W'  
        return f'{abs(self.lat):.1f}°{ns}, {abs(self.lon):.1f}°  
{we}'
```

注意，示例 5-2 和示例 5-3 中类的主体完全一样，区别在 `class` 语句上。`@dataclass` 装饰器不依赖继承或元类，如果你想使用这些机

制，则不受影响。³ 示例 5-3 中的 `Coordinate` 类是 `object` 的子类。

³类装饰器和元类都将在第 24 章探讨。这两种机制提供了超越继承的功能，方便你定制类的行为。

主要功能

3 个数据类构建器有许多共同点，如表 5-1 所示。

表 5-1：比较 3 个数据类构建器的部分功能（x 表示此数据类的实例）

	<code>namedtuple</code>	<code>NamedTuple</code>	<code>dataclass</code>
可变实例	否	否	是
<code>class</code> 语句句法	否	是	是
构造字典	<code>x._asdict()</code>	<code>x._asdict()</code>	<code>dataclasses.asdict(x)</code>
获取字段名称	<code>x._fields</code>	<code>x._fields</code>	<code>[f.name for f in dataclasses.fields(x)]</code>
获取默认值	<code>x._field_defaults</code>	<code>x._field_defaults</code>	<code>[f.default for f in dataclasses.fields(x)]</code>

	<code>namedtuple</code>	<code>NamedTuple</code>	<code>dataclass</code>
获取 字段 类型	N/A	<code>x.__annotations__</code>	<code>x.__annotations__</code>
更改 之后 创建 新实例	<code>x._replace(...)</code>	<code>x._replace(...)</code>	<code>dataclasses.replace(x, ...)</code>
运行 时定 义新 类	<code>namedtuple(...)</code>	<code>NamedTuple(...)</code>	<code>dataclasses.make_dataclass(...)</code>



`typing.NamedTuple` 和 `@dataclass` 构建的类有一个 `__annotations__` 属性，存放字段的类型提示。然而，不建议直接读取 `__annotations__` 属性。推荐使用 `inspect.get_annotations(MyClass)` (Python 3.10 新增) 或 `typing.get_type_hints(MyClass)` (Python 3.5~3.9) 获取类型信息，因为这两个函数提供了额外的服务，例如可以解析类型提示中的向前引用。15.5.1 节将详谈这个问题。

下面分别讨论这些主要功能。

01. 可变实例

3 个数据类构建器之间主要的区别在于，`collections.namedtuple` 和 `typing.NamedTuple` 构建的类是 `tuple` 的子类，因此实例是不可变的。`@dataclass` 默认

构建可变的类。不过，`@dataclass` 装饰器接受一个关键字参数 `frozen`，如示例 5-3 所示。指定 `frozen=True`，初始化实例之后，如果为字段赋值，则抛出异常。

02. `class` 语句句法

只有 `typing.NamedTuple` 和 `dataclass` 支持常规的 `class` 语句句法，方便为构建的类添加方法和文档字符串。

03. 构造字典

两种具名元组都提供了构造 `dict` 对象的实例方法 (`._asdict`)，可根据数据类实例的字段构造字典。`dataclasses` 模块也提供了构造字典的函数，即 `dataclasses.asdict`。

04. 获取字段名称和默认值

3 个类构建器都支持获取字段名称和可能配置的默认值。对于具名元组类，这些元数据在类属性 `._fields` 和 `._fields_defaults` 中。对于使用 `dataclass` 装饰器构建的类，这些元数据使用 `dataclasses` 模块中的 `fields` 函数获取。`fields` 函数返回一个由 `Field` 对象构成的元组，`Field` 对象有几个属性，包括 `name` 和 `default`。

05. 获取字段类型

`typing.NamedTuple` 和 `@dataclass` 定义的类有一个 `__annotations__` 类属性，值为字段名称到类型的映射。前面说过，不要直接读取 `__annotations__` 属性，而要使用 `typing.get_type_hints` 函数。

06. 更改之后创建新实例

对于具名元组实例 `x`，`x._replace(**kwargs)` 根据指定的关键字参数替换某些属性的值，返回一个新实例。模块级函数 `dataclasses.replace(x, **kwargs)` 与 `@dataclass` 装饰的类具有相同的作用。

07. 运行时定义新类

`class` 句法虽然可读性更高，但毕竟还是硬编码的。框架可能需要在运行时动态构建数据类。为此，可以使用默认的函数调用句法，`collections.namedtuple` 和 `typing.NamedTuple` 都支持。`dataclasses` 模块提供的 `make_dataclass` 函数也是出于这个目的。

大致介绍数据类构建器的主要功能之后，下面逐一讨论这 3 个类构建器，先从最简单的开始。

5.3 典型的具名元组

`collections.namedtuple` 是一个工厂函数，用于构建增强的 `tuple` 子类，具有字段名称、类名和提供有用信息的 `__repr__` 方法。`namedtuple` 构建的类可在任何需要元组的地方使用。其实，为了方便，以前 Python 标准库中返回元组的很多函数，现在都返回具名元组，这对用户的代码没有任何影响。



`namedtuple` 构建的类，其实例占用的内存量与元组相同，因为字段名称存储在类中。

示例 5-4 定义一个具名元组，存储一个城市的信息。

示例 5-4 定义并使用一个具名元组类型

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population
coordinates') ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=
(35.689722,
139.691667))
>>> tokyo.population ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

❶ 创建具名元组需要指定两个参数：一个类名和一个字段名称列表。后一个参数可以是产生字符串的可迭代对象，也可以是一整个以空格分隔的字符串。

❷ 字段的值必须以单个位置参数传给构造函数（而 `tuple` 构造函数接受单个可迭代对象）。

❸ 可以通过名称或位置访问字段。

作为 `tuple` 的子类，`City` 继承了一些有用的方法，例如 `__eq__`，以及比较运算符背后的特殊方法 `__lt__` 等，可用于排序 `City` 实例构成的列表。

除了从 `tuple` 继承，具名元组还有几个额外的属性和方法。示例 5-5 演示了几个最有用的属性和方法：类属性 `_fields`、类方法

`_make(iterable)` 和实例方法 `_asdict()`。

示例 5-5 具名元组的属性和方法（续前例）

```
>>> City._fields ❶
('name', 'country', 'population', 'location')
>>> Coordinate = namedtuple('Coordinate', 'lat lon')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, Coordinate(28.613889,
77.208889))
>>> delhi = City._make(delhi_data) ❷
>>> delhi._asdict() ❸
{'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935,
'location': Coordinate(lat=28.613889, lon=77.208889)}
>>> import json
>>> json.dumps(delhi._asdict()) ❹
'{"name": "Delhi NCR", "country": "IN", "population": 21.935,
"location": [28.613889, 77.208889]}'
```

- ❶ `._fields` 属性的值是一个元组，存储类的字段名称。
- ❷ `._make()` 方法根据可迭代对象构建 `City` 实例，与 `City(*delhi_data)` 作用相同。
- ❸ `._asdict()` 方法返回根据具名元组实例构建的 `dict` 对象。
- ❹ `._asdict()` 方法可把数据序列化成 JSON 格式。



在 Python 3.7 之前，`_asdict` 方法返回一个 `OrderedDict` 对象。从 Python 3.8 开始，它返回一个简单的 `dict` 对象，因为现在键的插入顺序得以保留，所以影响不大。如果你还想得到 `OrderedDict` 对象，`_asdict` 文档建议根据返回结果自行构建，即 `OrderedDict(x._asdict())`。

从 Python 3.7 开始，`namedtuple` 接受 `defaults` 关键字参数，值为一个产生 N 项的可迭代对象，为从右数的 N 个字段指定默认值。示例 5-6 定义具名元组 `Coordinate`，为 `reference` 字段指定默认值。

示例 5-6 构建一个具名元组，为字段指定默认值

```
>>> Coordinate = namedtuple('Coordinate', 'lat lon reference',
defaults=['WGS84'])
>>> Coordinate(0, 0)
Coordinate(lat=0, lon=0, reference='WGS84')
>>> Coordinate._field_defaults
{'reference': 'WGS84'}
```

5.2 节的“class 语句句法”中提到，使用 `typing.NamedTuple` 和 `@dataclass` 支持的 `class` 句法方便我们增加方法。具名元组也能用于增加方法，只是过程有点曲折。如果你不想这么麻烦，请跳过下面的附注栏。

为具名元组注入方法

回顾一下第 1 章中示例 1-1 是如何构建 `Card` 类的。

```
Card = collections.namedtuple('Card', ['rank', 'suit'])
```

随后定义了 `spades_high` 函数，用来给扑克牌排序。如果把这部分逻辑封装成 `Card` 的方法就好了。然而，`Card` 不是使用 `class` 语句定义的，添加 `spades_high` 方法的过程有点曲折，要先定义函数，再把函数赋值给一个类属性，如示例 5-7 所示。

示例 5-7 frenchdeck.doctest: 为 `Card` (1.2 节构建的具名元组) 添加一个类属性和一个方法

```
>>> Card.suit_values = dict(spades=3, hearts=2, diamonds=1,
clubs=0) ❶
>>> def spades_high(card):
❷
...     rank_value = FrenchDeck.ranks.index(card.rank)
...     suit_value = card.suit_values[card.suit]
...     return rank_value * len(card.suit_values) + suit_value
...
>>> Card.overall_rank = spades_high
❸
>>> lowest_card = Card('2', 'clubs')
>>> highest_card = Card('A', 'spades')
```

```
>>> lowest_card.overall_rank()
④
0
>>> highest_card.overall_rank()
51
```

❶ 添加一个类属性，值为各个花色。

❷ `spades_high` 函数将变成方法。第一个参数不必命名为 `self`，但是调用时指代的的就是接收方。

❸ 把 `spades_high` 函数依附到 `Card` 类型，变成一个方法，名为 `overall_rank`。

❹ 成功！

在 `class` 语句中定义方法可读性好，也方便后期维护。但你也要知道，刚刚介绍的这种方式是可行的，或许有时用得到。⁴

我们稍微偏离了主线，目的是展示动态语言的强大。

⁴不知你是否了解 Ruby，Ruby 程序员经常注入方法，不过这种技术也有一些争议。在 Python 中，这种做法不常见，因为 `str`、`list` 等内置类型不支持。我觉得这种限制是一件好事。

接下来讲讲具名元组的变体——`typing.NamedTuple`。

5.4 带类型的具名元组

示例 5-6 中有一个默认字段的 `Coordinate` 类还可以使用 `typing.NamedTuple` 定义，如示例 5-8 所示。

示例 5-8 `typing_namedtuple/coordinates2.py`

```
from typing import NamedTuple
```

```
class Coordinate(NamedTuple):  
    lat: float ❶  
    lon: float  
    reference: str = 'WGS84' ❷
```

❶ 每个实例字段都要注解类型。

❷ 实例字段 `reference` 注解了类型，还指定了默认值。

使用 `typing.NamedTuple` 构建的类，拥有的方法并不比 `collections.namedtuple` 生成的更多，而且同样也从 `tuple` 继承方法。唯一的区别是多了类属性 `__annotations__`，而在运行时，Python 完全忽略该属性。

鉴于 `typing.NamedTuple` 的主要功能是类型注解，下面就花点儿时间介绍一下，然后再继续探讨数据类构建器。

5.5 类型提示入门

类型提示（也叫类型注解）声明函数参数、返回值、变量和属性的预期类型。

关于类型提示，首先你要知道，Python 字节码编译器和解释器根本不强制你提供类型信息。



本节只简略介绍类型提示，让你对 `typing.NamedTuple` 和 `@dataclass` 声明使用的句法和注解的意义有一点感性认识。函数签名的类型注解将在第 8 章讲解，更高级的注解将在第 15 章探讨。本节主要涉及 `str`、`int` 和 `float` 等简单内置类型的注解。数据类中的字段最常使用这些类型。

5.5.1 运行时没有作用

Python 类型提示可以看作“供 IDE 和类型检查工具验证类型的文档”。

这是因为，类型提示对 Python 程序的运行时行为没有影响。请看示例 5-9。

示例 5-9 Python 在运行时不考虑类型提示

```
>>> import typing
>>> class Coordinate(typing.NamedTuple):
...     lat: float
...     lon: float
...
>>> trash = Coordinate('Ni!', None)
>>> print(trash)
Coordinate(lat='Ni!', lon=None) ❶
```

❶ 就像我说过的，运行时不检查类型。

在一个 Python 模块中输入示例 5-9 中的代码，运行后，你会看到一个没什么意义的 `Coordinate`，不报错也不发出警告。

```
$ python3 nocheck_demo.py
Coordinate(lat='Ni!', lon=None)
```

类型提示主要为第三方类型检查工具提供支持，例如 `Mypy` 和 `PyCharm IDE` 内置的类型检查器。这些是静态分析工具，在“静止”状态下检查 Python 源码，不运行代码。

为了看到类型提示的效果，必须使用相关工具（例如 `linter`）检查代码。使用 `Mypy` 检查之前的示例，看到的输出如下所示。

```
$ mypy nocheck_demo.py
nocheck_demo.py:8: error: Argument 1 to "Coordinate" has
incompatible type "str"; expected "float"
nocheck_demo.py:8: error: Argument 2 to "Coordinate" has
incompatible type "None"; expected "float"
```

可以看到，根据 `Coordinate` 的定义，`Mypy` 知道创建实例时传入的两个参数必须都是 `float` 类型，但是创建 `trash` 时传入的是 `str` 对象和 `None`。⁵

⁵在类型提示上下文中，`None` 不是 `NoneType` 单例，而是 `NoneType` 自身的别名。仔细想一想，这样做有点奇怪，但也符合直觉，而且对于返回 `None` 的函数，使用 `None` 注解返回值更容易理解。

接下来讲一讲类型提示的句法和意义。

5.5.2 变量注解句法

`typing.NamedTuple` 和 `@dataclass` 使用 PEP 526 定义的句法注解变量。本节简要介绍在 `class` 语句中定义属性的注解句法。

变量注解的基本句法如下所示。

```
var_name: some_type
```

允许使用的类型在 PEP 484 中的“Acceptable type hints”一节规定，不过定义数据类时，最常使用以下类型。

- 一个具体类，例如 `str` 或 `FrenchDeck`。
- 一个参数化容器类型，例如 `list[int]`、`tuple[str, float]` 等。
- `typing.Optional`，例如 `Optional[str]`，声明一个字段的类型可以是 `str` 或 `None`。

另外，还可以为变量指定初始值。在 `typing.NamedTuple` 和 `@dataclass` 声明中，指定的初始值作为属性的默认值，防止调用构造函数时没有提供对应的参数。

```
var_name: some_type = a_value
```

5.5.3 变量注解的意义

5.5.1 节说过，类型提示在运行时没有作用。然而，Python 在导入时（加载模块时）会读取类型提示，构建 `__annotations__` 字典，

供 `typing.NamedTuple` 和 `@dataclass` 使用，增强类的功能。

接下来先分析示例 5-10 定义的一个简单类，然后再讨论 `typing.NamedTuple` 和 `@dataclass` 增加的额外功能。

示例 5-10 `meaning/demo_plain.py`: 一个简单的类，带有类型提示

```
class DemoPlainClass:
    a: int           ❶
    b: float = 1.1   ❷
    c = 'spam'       ❸
```

❶ `a` 出现在 `__annotations__` 中，但被抛弃了，因为该类没有名为 `a` 的属性。

❷ `b` 作为注解记录在案，而且是一个类属性，值为 `1.1`。

❸ `c` 是普通的类属性，没有注解。

我们可以在控制台中验证一下，首先读取 `DemoPlainClass` 的 `__annotations__`，然后尝试获取属性 `a`、`b` 和 `c`。

```
>>> from demo_plain import DemoPlainClass
>>> DemoPlainClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoPlainClass.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'DemoPlainClass' has no attribute 'a'
>>> DemoPlainClass.b
1.1
>>> DemoPlainClass.c
'spam'
```

注意，特殊属性 `__annotations__` 由解释器创建，记录源码中出现的类型提示，即使是普通的类。

a 只作为注解存在，不是类属性，因为没有绑定值。⁶ b 和 c 存储为类属性，因为它们绑定了值。

⁶Python 没有“未定义” (undefined) 概念。“未定义”是 JavaScript 设计最大的败笔之一。感谢 Guido!

这 3 个属性都不出现在 DemoPlainClass 的实例中。使用 o = DemoPlainClass() 创建一个对象，o.a 抛出 AttributeError，而 o.b 和 o.c 检索类属性，值分别为 1.1 和 'spam'，行为与常规的 Python 对象相同。

01. 研究一个 typing.NamedTuple 类

现在来研究一个使用 typing.NamedTuple 构建的类（见示例 5-11）。这个类的属性和注解与示例 5-10 中的 DemoPlainClass 类一样。

示例 5-11 meaning/demo_nt.py: 使用 typing.NamedTuple 构建一个类

```
import typing

class DemoNTClass(typing.NamedTuple):
    a: int          ❶
    b: float = 1.1   ❷
    c = 'spam'       ❸
```

❶ a 是注解，也是实例属性。

❷ 同样，b 是注解，也是实例属性，默认值为 1.1。

❸ c 是普通的类属性，没有注解。

研究一下 DemoNTClass，结果如下。

```
>>> from demo_nt import DemoNTClass
>>> DemoNTClass.__annotations__
```

```
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoNTClass.a
<_collections._tuplegetter object at 0x101f0f940>
>>> DemoNTClass.b
<_collections._tuplegetter object at 0x101f0f8b0>
>>> DemoNTClass.c
'spam'
```

可以看到，a 和 b 的注解与示例 5-10 一样。但是，`typing.NamedTuple` 创建了类属性 a 和 b。c 是普通的类属性，值为 'spam'。

类属性 a 和 b 是**描述符**。这是高级功能，将在第 23 章探讨。现在可以把描述符理解为特性（property）读值（getter）方法，即不带调用运算符（`()`）的方法，用于读取实例属性。实际上，这意味着 a 和 b 是只读实例属性。这一点不难理解，因为 `DemoNTClass` 实例是某种高级的元组，而元组是不可变的。

`DemoNTClass` 还有定制文档字符串。

```
>>> DemoNTClass.__doc__
'DemoNTClass(a, b)'
```

下面研究一个 `DemoNTClass` 实例。

```
>>> nt = DemoNTClass(8)
>>> nt.a
8
>>> nt.b
1.1
>>> nt.c
'spam'
```

构造 `nt` 对象时，至少要为 `DemoNTClass` 提供 a 参数。b 也是构造函数的参数，不过它有默认值 1.1，因此可以不提供。nt 对象有 a 和 b 两个属性，这在预期之中。但是，没有 c 属性，像往常一样，Python 从类中检索该属性。

为 `nt.a`、`nt.b`、`nt.c` 甚至 `nt.z` 赋值，抛出 `AttributeError` 异常，这几个错误消息稍有区别。请你自己试一下，分析错误消息的内容。

02. 研究一个使用 `dataclass` 装饰的类

现在研究一下示例 5-12。

示例 5-12 `meaning/demo_dc.py`: 一个使用 `@dataclass` 装饰的类

```
from dataclasses import dataclass

@dataclass
class DemoDataClass:
    a: int          ❶
    b: float = 1.1   ❷
    c = 'spam'       ❸
```

❶ `a` 是注解，也是受描述符控制的实例属性。

❷ 同样，`b` 是注解，也是受描述符控制的实例属性，默认值为 `1.1`。

❸ `c` 是普通的类属性，没有注解。

现在查看 `DemoDataClass` 类的 `__annotations__`、`__doc__`，以及 `a`、`b` 和 `c` 属性。

```
>>> from demo_dc import DemoDataClass
>>> DemoDataClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoDataClass.__doc__
'DemoDataClass(a: int, b: float = 1.1)'
>>> DemoDataClass.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: type object 'DemoDataClass' has no attribute 'a'
>>> DemoDataClass.b
1.1
>>> DemoDataClass.c
'spam'
```

`__annotations__` 和 `__doc__` 没什么让人意外的。然而，`DemoDataClass` 没有名为 `a` 的属性。相比之下，示例 5-11 中的 `DemoNTClass` 有可从实例中获取只读属性 `a` 的描述符（那个神秘的 `<_collections._tuplegetter>`）。这是因为，`a` 属性只在 `DemoDataClass` 实例中存在。如果冻结 `DemoDataClass` 类，那么 `a` 就变成可获取和设定的公开属性。但是，`b` 和 `c` 作为类属性存在，`b` 存储实例属性 `b` 的默认值，而 `c` 本身就是类属性，不绑定到实例上。

下面来看 `DemoDataClass` 实例的情况。

```
>>> dc = DemoDataClass(9)
>>> dc.a
9
>>> dc.b
1.1
>>> dc.c
'spam'
```

同样，`a` 和 `b` 是实例属性，而 `c` 是通过实例获取的类属性。

前文说过，`DemoDataClass` 实例是可变的，而且运行时不检查类型。

```
>>> dc.a = 10
>>> dc.b = 'oops'
```

甚至还可以为不存在的属性赋值。

```
>>> dc.c = 'whatever'
>>> dc.z = 'secret stash'
```

现在，dc 实例有 c 属性，这对类属性 c 没有影响。我们还可以新增一个 z 属性。这是 Python 正常的行为：常规实例自身可以有未出现在类中的属性。⁷

⁷执行 `__init__` 之后设置属性，有悖于 3.9 节讲过的字典键共享内存优化措施。

5.6 @dataclass 详解

我们目前见过的 @dataclass 示例都比较简单。这个装饰器接受多个关键字参数，完整签名如下。

```
@dataclass(*, init=True, repr=True, eq=True, order=False,
            unsafe_hash=False, frozen=False)
```

第一个参数位置上的 * 表示后面都是关键字参数。表 5-2 简要说明这些关键字参数。

表 5-2: @dataclass 装饰器接受的关键字参数

参数	作用	默认值	备注
init	生成 <code>__init__</code>	True	如果用户自己实现了 <code>__init__</code> ，则忽略该参数
repr	生成 <code>__repr__</code>	True	如果用户自己实现了 <code>__repr__</code> ，则忽略该参数
eq	生成 <code>__eq__</code>	True	如果用户自己实现了 <code>__eq__</code> ，则忽略该参数

参数	作用	默认值	备注
<code>order</code>	生成 <code>__lt__</code> 、 <code>__le__</code> 、 <code>__gt__</code> 、 <code>__ge__</code>	<code>False</code>	设为 <code>True</code> 时，如果 <code>eq=False</code> ，或者自行定义或继承其他用于比较的方法，则抛出异常
<code>unsafe_hash</code>	生成 <code>__hash__</code>	<code>False</code>	语义复杂，有多问题需要注意，详见 <code>dataclass</code> 函数的文档
<code>frozen</code>	让实例不可变	<code>False</code>	防止意外更改实例，相对安全，但不是绝对不可变 ^a

^a `@dataclass` 生成 `__setattr__` 和 `__delattr__`，在用户尝试设置或删除字段时抛出 `dataclass.FrozenInstanceError` (`AttributeError` 的子类)，以此模拟不可变性。

以上参数的默认值适用于多数情况。不过，你可能会更改以下参数的值，不使用默认值。

```
frozen=True
```

防止意外更改类的实例。

```
order=True
```

允许排序数据类的实例。

Python 对象是动态的，只要愿意，程序员还是可以绕过 `frozen=True` 这道防线。不过，在代码评审阶段很容易发现这种小伎俩。

如果 `eq` 和 `frozen` 参数的值都是 `True`，那么 `@dataclass` 将生成一个合适的 `__hash__` 方法，确保实例是可哈希的。生成的 `__hash__` 方法使用所有字段的数据，通过字段选项（见 5.6.1 节）也不能排除。对于 `frozen=False`（默认值），`@dataclass` 把 `__hash__` 设为 `None`，覆盖从任何超类继承的 `__hash__` 方法，表明实例不可哈希。

关于 `unsafe_hash`，“PEP 557—Data Classes”是这样说的：

可以设置 `unsafe_hash=True`，强制数据类创建 `__hash__` 方法，但是不建议这么做。如果一个类在逻辑上是不可变的，但事实上是可变的，则可以这么做。然而，这是特殊情况，务必小心谨慎。

这也是我对 `unsafe_hash` 的观点。如果你认为必须使用这个选项，请认真阅读 `dataclasses.dataclass` 文档。

生成的数据类还可以在字段层面进一步定制。

5.6.1 字段选项

我们已经见过最基本的字段选项，即在提供类型提示的同时设定默认值。声明的字段将作为参数传给生成的 `__init__` 方法。Python 规定，带默认值的参数后面不能有不带默认值的参数。因此，为一个字段声明默认值之后，余下的字段都要有默认值。

对初级 Python 开发人员来说，可变的默认值往往导致 bug。如果在函数定义中使用可变默认值，调用函数时很容易破坏默认值，则导致后续调用的行为发生变化（这个问题将在 6.5.1 节详谈）。类属性通常用作实例属性的默认值，数据类也是如此。`@dataclass` 使用类型提示中的默认值生成传给 `__init__` 方法的参数默认值。为了避免 bug，`@dataclass` 拒绝像示例 5-13 那样定义类。

示例 5-13 `dataclass/club_wrong.py`：这个类抛出 `ValueError`

```
@dataclass
class ClubMember:
    name: str
    guests: list = []
```

加载 ClubMember 所在的模块，看到的结果如下所示。

```
$ python3 club_wrong.py
Traceback (most recent call last):
  File "club_wrong.py", line 4, in <module>
    class ClubMember:
    ...省略多行...
ValueError: mutable default <class 'list'> for field guests is not
allowed:
use default_factory
```

ValueError 消息指出了问题所在，还提供了一个解决方案：使用 default_factory。示例 5-14 给出了纠正 ClubMember 的方法。

示例 5-14 dataclasses/club.py：这里定义的 ClubMember 没有问题

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
```

在示例 5-14 中，guests 字段的默认值不是一个列表字面量，而是调用 dataclasses.field 函数，把参数设为 default_factory=list，以此设定默认值。

default_factory 参数的值可以是一个函数、一个类，或者其他可调用对象，在每次创建数据类的实例时调用（不带参数），构建默认值。这样，每个 ClubMember 实例都有自己的一个 list，而不是所有实例共用同一个 list。共用往往导致 bug，而且我们很少希望共用。



如果类中有默认值为 `list` 的字段，则 `@dataclass` 拒绝定义，这一点很好。然而你要知道，这种方案只适用于部分情况，只对 `list`、`dict` 和 `set` 有效。除此之外，其他可变的值不会引起 `@dataclass` 的注意。遇到这样的问题，你要自己处理，为可变的默认值设置默认工厂。

浏览 `dataclasses` 模块文档，你会发现有一个 `list` 字段使用的句法比较新奇，如示例 5-15 所示。

示例 5-15 `dataclass/club_generic.py`：这里定义的 `ClubMember` 更精确

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:
    name: str
    guests: list[str] = field(default_factory=list) ❶
```

❶ `list[str]` 的意思是由字符串构成的列表。

新句法 `list[str]` 是一种参数化泛型。从 Python 3.9 开始，内置类型 `list` 可以使用方括号表示法指定列表中项的类型。



在 Python 3.9 之前，内置容器类型不支持泛型表示法。为了临时解决这一问题，`typing` 模块提供了对应的容器类型。在 Python 3.8 或之前的版本中，如果需要参数化 `list` 类型提示，则必须使用从 `typing` 模块中导入的 `List` 类型，写作 `List[str]`。这个问题详见 8.5.4 节中的附注栏。

泛型将在第 8 章探讨。现在只需要知道，示例 5-14 和示例 5-15 中的做法都是对的，而且 `Mypy` 在检查这两种定义的类型时均不报错。

这两种声明方式是有区别的，`guests: list` 表示 `guests` 列表可以由任何类型的对象构成，而 `guests: list[str]` 的意思是 `guests` 列表中的每一项都必须是字符串。因此，如果在列表中存储无效的项，或者读取到无效的项，则类型检查工具将报错。

`default_factory` 应该是 `field` 函数最常使用的参数，不过除此之外还有其他参数可用，如表 5-3 所示。

表 5-3: `field` 函数接受的关键字参数

参数	作用	默认值
<code>default</code>	字段的默认值	<code>_MISSING_TYPE</code> ^a
<code>default_factory</code>	不接受参数的函数，用于产生默认值	<code>_MISSING_TYPE</code>
<code>init</code>	把字段作为参数传给 <code>__init__</code> 方法	<code>True</code>
<code>repr</code>	在 <code>__repr__</code> 方法中使用字段	<code>True</code>
<code>compare</code>	在 <code>__eq__</code> 、 <code>__lt__</code> 等比较方法中使用字段	<code>True</code>
<code>hash</code>	在 <code>__hash__</code> 方法中使用字段计算哈希值	<code>None</code> ^b
<code>metadata</code>	用户定义的数据映射； <code>@dataclass</code> 忽略该参数	<code>None</code>

^a `dataclass._MISSING_TYPE` 是一个哨符值，表示未提供该参数。这样就可以把默认值设为经常需要使用的 `None`。

`b hash=None` 表示仅当 `compare=True` 时才在 `__hash__` 方法中使用字段。

之所以有 `default` 参数，是因为在字段注解中设置默认值的位置被 `field` 函数调用占据了。假如我们想创建一个 `athlete` 字段，把默认值设为 `False`，而且不提供给 `__repr__` 方法使用，那么要像下面这样编写。

```
@dataclass
class ClubMember:
    name: str
    guests: list = field(default_factory=list)
    athlete: bool = field(default=False, repr=False)
```

5.6.2 初始化后处理

`@dataclass` 生成的 `__init__` 方法只做一件事：把传入的参数及其默认值（如未指定值）赋值给实例属性，变成实例字段。可是，有些时候初始化实例要做的不只是这些。为此，可以提供一个 `__post_init__` 方法。如果存在这个方法，则 `@dataclass` 将在生成的 `__init__` 方法最后调用 `__post_init__` 方法。

`__post_init__` 经常用于执行验证，以及根据其他字段计算一个字段的值。下面举个例子说明这两种用途。

我们将定义一个 `ClubMember` 子类，名为 `HackerClubMember`。首先，通过 `doctest` 说明 `HackerClubMember` 的预期行为，如示例 5-16 所示。

示例 5-16 `dataclass/hackerclub.py`: `HackerClubMember` 的 `doctest`

```
"""
`HackerClubMember` 构造函数接受一个可选的 `handle` 参数::

    >>> anna = HackerClubMember('Anna Ravenscroft',
handle='AnnaRaven')
    >>> anna
```

```
HackerClubMember(name='Anna Ravenscroft', guests=[],
handle='AnnaRaven')
```

如果没有指定``handle``, 则设为会员姓名的第一部分::

```
>>> leo = HackerClubMember('Leo Rochael')
>>> leo
HackerClubMember(name='Leo Rochael', guests=[], handle='Leo')
```

会员的昵称必须是唯一的。下面的``leo2``无法创建, 因为``handle``的值是'Leo', 而这个昵称已经被``leo``占用了::

```
>>> leo2 = HackerClubMember('Leo DaVinci')
Traceback (most recent call last):
...
ValueError: handle 'Leo' already exists.
```

因此, 创建``leo2``时必须明确指定``handle``::

```
>>> leo2 = HackerClubMember('Leo DaVinci', handle='Neo')
>>> leo2
HackerClubMember(name='Leo DaVinci', guests=[], handle='Neo')
"""
```

注意, `handle` 必须是关键字参数, 因为 `HackerClubMember` 从 `ClubMember` 继承了 `name` 和 `guests`, `handle` 字段是额外增加的。从为 `HackerClubMember` 生成的文档字符串可以看出构造函数调用中各字段的顺序。

```
>>> HackerClubMember.__doc__
"HackerClubMember(name: str, guests: list = <factory>, handle: str
= '')"
```

这里, `<factory>` 是一种简略表示, 意思是 `guests` 字段的默认值由某个可调用对象产生 (这里使用的工厂是 `list` 类)。上述文档字符串的重点是, 如果想提供 `handle`, 但不提供 `guests`, 那么必须利用关键字参数传入 `handle`。

`dataclasses` 模块文档中的“Inheritance”一节详细说明涉及多层继承时如何给字段排序。



第 14 章将讨论继承的错误用法，尤其是超类不是抽象类的情况。创建具有层次结构的数据类往往不是个好主意，示例 5-17 这么做是为了减少 `HackerClubMember` 类的代码，把精力集中在 `handle` 字段声明和 `__post_init__` 方法中的验证逻辑上。

`HackerClubMember` 类的实现如示例 5-17 所示。

示例 5-17 `dataclass/hackerclub.py`: 构建 `HackerClubMember` 类

```
from dataclasses import dataclass
from club import ClubMember

@dataclass
class HackerClubMember(ClubMember):
    all_handles = set()
    handle: str = ''

    def __post_init__(self):
        cls = self.__class__
        if self.handle == '':
            self.handle = self.name.split()[0]
        if self.handle in cls.all_handles:
            msg = f'handle {self.handle!r} already exists.'
            raise ValueError(msg)
        cls.all_handles.add(self.handle)
```

- ❶ `HackerClubMember` 扩展 `ClubMember`。
- ❷ `all_handles` 是一个类属性。
- ❸ `handle` 是一个实例字段，类型为 `str`，默认值为空字符串（相当于是可选的）。
- ❹ 获取实例所属的类。
- ❺ 如果 `self.handle` 是空字符串，则把它设为 `name` 的第一部分。

❹ 如果 `self.handle` 在 `cls.all_handles` 中，则抛出 `ValueError`。

❺ 把新 `handle` 添加到 `cls.all_handles` 中。

示例 5-17 能实现我们的需求，但不能让静态类型检查工具满意。5.6.3 节会说明原因，并给出修正方法。

5.6.3 带类型的类属性

使用 Mypy 检查示例 5-17，报错信息如下。

```
$ mypy hackerclub.py
hackerclub.py:37: error: Need type annotation for "all_handles"
(hint: "all_handles: Set[<type>] = ...")
Found 1 error in 1 file (checked 1 source file)
```

可惜，Mypy（我用的是 0.910 版）提供的提示对使用 `@dataclass` 构建的类没有什么用处。

Mypy 建议使用 `Set`，而我用的是 Python 3.9，可以使用 `set`——免得再从 `typing` 模块中导入 `Set`。更重要的是，如果为 `all_handles` 添加类型提示，例如 `set[...]`，那么 `@dataclass` 将把 `all_handles` 变成实例字段。5.5.3 节的“研究一个使用 `dataclass` 装饰的类”中就出现过这种情况。

“PEP 526—Syntax for Variable Annotations”中定义的变通方法不太优雅。若想为类变量添加类型提示，则要使用一个名为 `typing.ClassVar` 的伪类型，借助泛型表示法 `[]` 设定变量的类型，同时声明为类属性。

为了让类型检查工具和 `@dataclass` 满意，在示例 5-17 中应当像下面这样声明 `all_handles`。

```
all_handles: ClassVar[set[str]] = set()
```

这里，类型提示的意思如下：

`all_handles` 是一个类属性，类型为字符串构成的集合，默认值是一个空集合。

编写这个注解之前，必须从 `typing` 模块中导入 `ClassVar`。

`@dataclass` 装饰器不关心注解中的类型，但有两种例外情况，这是其中之一，即类型为 `ClassVar` 时，不为属性生成实例字段。

另外一种情况是声明“仅作初始化的变量”，详见 5.6.4 节。

5.6.4 初始化不作为字段的变量

有时，我们需要把不作为实例字段的参数传给 `__init__` 方法。按照 `dataclasses` 文档的说法，这种参数叫“仅作初始化的变量”（`init-only variable`）。为了声明这种参数，`dataclasses` 模块提供了伪类型 `InitVar`，句法与 `typing.ClassVar` 一样。文档中给出的例子定义一个数据类，包含一个使用数据库初始化的字段，因此必须把数据库对象传给构造方法。

下面重现文档中的那个例子，如示例 5-18 所示。

示例 5-18 `dataclasses` 模块文档中的例子

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

注意 `database` 属性的声明方式。 `InitVar` 阻止 `@dataclass` 把 `database` 视为常规的字段。 `database` 不会被设为实例属性，也不会出现在 `dataclasses.fields` 函数返回的列表中。然而，对于生成的 `__init__` 方法， `database` 是参数之一，同时也传给 `__post_init__` 方法。如果你想自己编写 `__post_init__` 方法，那就必须像示例 5-18 那样，在方法签名中增加相应的参数。

本书对 `@dataclass` 的讲解占据了很长篇幅，涵盖了多数有用的功能。有些功能在前面提到过，比如，表 5-1 把 3 个数据类构建器放在一起做了比较。如果想深入了解，请阅读 `dataclasses` 文档和“PEP 526—Syntax for Variable Annotations”。

5.6.5 节会再举一个示例，使用 `@dataclass` 构建一个内容较长的类。

5.6.5 @dataclass 示例：都柏林核心模式

目前所见的示例中，字段数量不多，实际使用中通常需要更多的字段。本节根据都柏林核心（Dublin Core）模式，使用 `@dataclass` 构建一个更复杂的类。

都柏林核心模式是一小组术语，可用于描述数字资源（视频、图像、网页等），也可用于描述物理资源，例如图书、CD 和艺术作品等对象。

该模式定义了 15 个可选字段，示例 5-19 中的 `Resource` 类用到了其中 8 个。

示例 5-19 `dataclass/resource.py`：基于都柏林核心模式构建 `Resource` 类

```
from dataclasses import dataclass, field
from typing import Optional
from enum import Enum, auto
from datetime import date
```

```

class ResourceType(Enum): ❶
    BOOK = auto()
    EBOOK = auto()
    VIDEO = auto()

@dataclass
class Resource:
    """描述媒体资源。"""
    identifier: str ❷
    title: str = '<untitled>' ❸
    creators: list[str] = field(default_factory=list)
    date: Optional[date] = None ❹
    type: ResourceType = ResourceType.BOOK ❺
    description: str = ''
    language: str = ''
    subjects: list[str] = field(default_factory=list)

```

- ❶ Enum 为 `Resource.type` 字段提供类型安全的值。
- ❷ `identifier` 是唯一必需的字段。
- ❸ `title` 是第一个有默认值的字段。因此，后续字段都要提供默认值。
- ❹ `date` 的值可以是一个 `datetime.date` 实例或 `None`。
- ❺ `type` 字段的默认值是 `ResourceType.BOOK`。

示例 5-20 中的 `doctest` 演示了如何在代码中使用 `Resource` 记录。

示例 5-20 dataclass/resource.py: 使用 `Resource` 类

```

>>> description = 'Improving the design of existing code'
>>> book = Resource('978-0-13-475759-9', 'Refactoring, 2nd
Edition',
...     ['Martin Fowler', 'Kent Beck'], date(2018, 11, 19),
...     ResourceType.BOOK, description, 'EN',
...     ['computer programming', 'OOP'])

```

```
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(identifier='978-0-13-475759-9', title='Refactoring,
2nd Edition',
  creators=['Martin Fowler', 'Kent Beck'],
  date=datetime.date(2018, 11, 19),
  type=<ResourceType.BOOK: 1>, description='Improving the design
of existing code',
  language='EN', subjects=['computer programming', 'OOP'])
```

@dataclass 生成的 `__repr__` 方法效果还行，不过还可以进一步定制，以提高其可读性。我们希望 `repr(book)` 返回以下格式。

```
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(
    identifier = '978-0-13-475759-9',
    title = 'Refactoring, 2nd Edition',
    creators = ['Martin Fowler', 'Kent Beck'],
    date = datetime.date(2018, 11, 19),
    type = <ResourceType.BOOK: 1>,
    description = 'Improving the design of existing code',
    language = 'EN',
    subjects = ['computer programming', 'OOP'],
)
```

示例 5-21 给出 `__repr__` 方法的代码，输出这种格式。这个示例使用 `dataclass.fields` 获取数据类字段的名称。

示例 5-21 `dataclass/resource_repr.py`: 为示例 5-19 中的 `Resource` 类实现 `__repr__` 方法

```
def __repr__(self):
    cls = self.__class__
    cls_name = cls.__name__
    indent = ' ' * 4
    res = [f'{cls_name}(']
    for f in fields(cls):
        value = getattr(self, f.name)
        res.append(f'{indent}{f.name} = {value!r},')
    res.append(')')
    return '\n'.join(res)
```

❶ 声明构建输出字符串的 `res` 列表，把第一项设为类名和左圆括号。

- ❷ 遍历类中的各个字段 `f`。
- ❸ 从实例上获取属性的名称。
- ❹ 追加缩进的行，显示字段的名称和 `repr(value)` (`!r` 起到的作用)。
- ❺ 追加右圆括号。
- ❻ 返回根据 `res` 构建的多行字符串。

这个示例的灵感来自在美国俄亥俄州都柏林举办的一次会议。我们对 Python 数据类构建器的探讨到此结束。

数据类的确很方便，但是如果过度使用，也会为你的项目带来不好的影响。5.7 节将详谈。

5.7 数据类导致代码异味

无论是自己编写所有代码实现数据类，还是利用本章介绍的某个类构建器实现数据类，都要注意一点：这可能表示你的设计存在问题。

在《重构（第 2 版）》中，Martin Fowler 和 Kent Beck 提出了“代码异味”这一概念。一旦代码出现异味，可能就意味着需要重构。讲数据类那一节，开头是这样说的：

所谓数据类是指，它们拥有一些字段，以及用于访问（读写）这些字段的函数，除此之外一无长物。这样的类只是一种不会说话的数据容器，它们几乎一定被其他类过分烦琐地操控着。

在 Martin Fowler 的个人网站中，有一篇很有启发性的文章，题为“Code Smell”。这篇文章以本章讨论的数据类为例说明代码异味，并给出了解决建议。下面完整转载了该文。⁸

⁸很荣幸，我与 Martin Fowler 是 Thoughtworks 同事，只用 20 分钟就得到了授权。

代码异味

Martin Fowler

代码异味是一种迹象，通常表明系统存在深层问题。这个说法最初由 Kent Beck 提出，当时我们在商讨《重构》一书的写作。

上面的定义比较简单，但是隐含了几层意思。首先，异味能迅速引起我们的注意。我最近喜欢说，异味能被“嗅到”。内容较长的方法就有异味，只要看到十几行 Java 代码，我的鼻子就会不自觉地抽动。

其次，有异味并不一定代表有问题。有些方法的内容就是长。你必须深入观察，判断有没有潜在的问题。不是说有异味就不好，异味通常是问题的表征，但不是说一定有问题。

我们都喜欢容易察觉的异味，多数时候这能让我们找到问题的根源。数据类（只包含数据而没有行为的类）就是很好的例子。遇到数据类，请问自己一个问题：这个类需要什么行为？然后，开始重构，加入需要的行为。通常，简单的思考和基本的重构就可以把空洞的对象抽象为真正的类。

异味的好处之一是，没有经验的人也很容易发现，即使他们既没有足够的知识来评判是否真的有问题，也不知道如何修正问题。我听说，有一些首席开发人员会提出“一周异味之星”，让团队成员寻找代码异味问题，找到问题后交给高级开发人员解决。一次解决一个异味问题是不错的做法，可以让团队成员循序渐进，引导他们成为更好的程序员。

面向对象编程的主要思想是把行为和数据放在同一个代码单元（一个类）中。如果一个类使用广泛，但是自身没有什么重要的行为，那么整个系统中可能遍布处理实例的代码，并出现在很多方法和函数中

(有些甚至是重复的)。这样的系统对维护来说简直就是噩梦。鉴于此，Martin Fowler 提出的重构方案才建议把职责放回数据类中。

尽管如此，仍然有几种情况适合使用没什么行为或者没有任何行为的数据类。

5.7.1 把数据类用作脚手架

这种情况是指，刚开始创建一个项目或者编写一个模块时，先用数据类简单实现一个类。随着时间的推移，类应该拥有自己的方法，而不是依赖其他类的方法操作该类的实例。脚手架是临时的，最终，你自定义的类或许应当完全独立，不依赖一开始使用的类构建器。

Python 也可用于快速解决问题和实验，用完之后把脚手架留在原地完全没有问题。

5.7.2 把数据类用作中间表述

数据类可用于构建将要导出为 JSON 或其他交换格式的记录，也可用于存储刚刚从其他系统导入的数据。Python 中的数据类构建器都提供了把实例转换为普通字典的方法或函数，而且构造函数全部支持通过关键字参数提供一个字典（非常接近 JSON 记录），再使用 `**` 展开。

在这种情况下，应把数据类实例当作不可变对象处理，即便字段是可变的，也不应在处于中间形式时更改。倘若更改，把数据和行为结合在一起的巨大优势就没有了。假如导入或导出时需要更改值，应该自己实现构建器方法，而不是使用数据类构建器提供的“用作字典”方法或常规的构造函数。

2.6 节和 3.3 节讲过如何通过模式匹配序列和映射，现在换个话题，说明如何使用模式匹配任意类的实例。

5.8 模式匹配类实例

类模式通过类型和属性（可选）匹配类实例。类模式的匹配对象可以是任何类的实例，而不仅仅是数据类的实例。⁹

⁹我之所以把这部分内容放在这里，是因为这一章在本书中最早讲到用户定义的类，而我认为类的模式匹配非常重要，等到第二部分再讲就晚了。我的理念是，知道如何使用类比知道如何定义类更重要。

类模式有 3 种变体：简单类模式、关键字类模式和位置类模式。下面按顺序依次研究。

5.8.1 简单类模式

其实，2.6 节有一个示例已经用到了类模式，那时是作为子模式使用的。

```
case [str(name), _, _, (float(lat), float(lon))]:
```

那个模式匹配项数为 4 的序列，第一项必须是 `str` 实例，最后一项必须是二元组，两项均为 `float` 实例。

类模式的句法看起来与构造函数调用差不多。下面的类模式匹配 `float` 值，未绑定变量（在 `case` 主体中，如果需要可以直接引用 `x`）。

```
match x:
  case float():
    do_something_with(x)
```

但是，像下面这样做可能导致 bug。

```
match x:
  case float: # 危险!!!
    do_something_with(x)
```

这里, `case float:` 可以匹配任何对象, 因为 Python 把 `float` 看作匹配对象绑定的变量。

`float(x)` 这种简单模式句法只适用于 9 种内置类型 (在“PEP 634—Structural Pattern Matching: Specification”中“Class Patterns”一节的末尾列出)。

```
bytes    dict    float    frozenset    int    list    set    str    tuple
```

对这些类来说, 看上去像构造函数的参数的那个变量, 例如 `float(x)` 中的 `x`, 绑定整个匹配的实例。如果是子模式, 则绑定匹配对象的一部分, 例如前例中序列模式内的 `str(name)`。

```
case [str(name), _, _, (float(lat), float(lon))]:
```

除 9 种内置类型之外, 看上去像参数的那个变量表示模式匹配的类实例的属性。

5.8.2 关键字类模式

为了说明如何使用关键字类模式, 下面定义一个 `City` 类, 再创建 5 个实例, 如示例 5-22 所示。

示例 5-22 `City` 类和几个实例

```
import typing

class City(typing.NamedTuple):
    continent: str
    name: str
    country: str

cities = [
    City('Asia', 'Tokyo', 'JP'),
    City('Asia', 'Delhi', 'IN'),
    City('North America', 'Mexico City', 'MX'),
    City('North America', 'New York', 'US'),
    City('South America', 'São Paulo', 'BR'),
]
```

那么，以下函数返回的列表中都是位于亚洲的城市。

```
def match_asian_cities():
    results = []
    for city in cities:
        match city:
            case City(continent='Asia'):
                results.append(city)
    return results
```

`City(continent='Asia')` 匹配的 `City` 实例，`continent` 属性的值等于 `'Asia'`，其他属性的值不考虑。

如果你想收集 `country` 属性的值，可以像下面这样写。

```
def match_asian_countries():
    results = []
    for city in cities:
        match city:
            case City(continent='Asia', country=cc):
                results.append(cc)
    return results
```

与前面一样，`City(continent='Asia', country=cc)` 也匹配位于亚洲的城市，不过现在把变量 `cc` 绑定到了实例的 `country` 属性上。模式变量叫 `country` 也没关系。

```
match city:
    case City(continent='Asia', country=country):
        results.append(country)
```

关键字类模式的可读性非常高，适用于任何有公开的实例属性的类，不过有点烦琐。

有时候，使用位置类模式更方便，不过匹配对象所属的类要显式支持，详见 5.8.3 节。

5.8.3 位置类模式

对于示例 5-22 中的定义，以下函数使用位置类模式获取亚洲城市列表。

```
def match_asian_cities_pos():
    results = []
    for city in cities:
        match city:
            case City('Asia'):
                results.append(city)
    return results
```

`City('Asia')` 匹配的 `City` 实例，第一个属性的值是 `'Asia'`，其他属性的值不考虑。

如果你想收集 `country` 属性的值，可以像下面这样写。

```
def match_asian_countries_pos():
    results = []
    for city in cities:
        match city:
            case City('Asia', _, country):
                results.append(country)
    return results
```

与前面一样，`City('Asia', _, country)` 也匹配位于亚洲的城市，不过现在把变量 `country` 绑定到了实例的第三个属性上。

可是，“第一个属性”和“第三个属性”是什么意思呢？

`City` 或其他类若想使用位置模式，要有一个名为 `__match_args__` 的特殊类属性。本章讲到的类构建器会自动创建这个属性。对于 `City` 类，`__match_args__` 属性的值如下所示。

```
>>> City.__match_args__
('continent', 'name', 'country')
```

可以看到，位置模式中属性的顺序就是 `__match_args__` 声明的顺序。

11.8 节将说明如何为没有使用类构建器创建的类定义

`__match_args__` 属性。



一个模式可以同时使用关键字参数和位置参数。

`__match_args__` 列出的是可供匹配的实例属性，不是全部属性。因此，有时候除了位置参数之外可能还需要使用关键字参数。

小结时间到。

5.9 本章小结

本章主要讲解了 3 个数据类构建器：`collections.namedtuple`、`typing.NamedTuple` 和 `dataclasses.dataclass`。我们知道，每个构建器都可以根据传给工厂函数的参数生成数据类，后两个构建器还可以通过 `class` 语句提供类型提示。两种具名元组变体生成的是 `tuple` 子类，与普通的元组相比，增加了通过名称访问字段的功能，另外还提供一个类属性 `_fields`，以字符串元组的形式列出字段名称。

我们把 3 个类构建器并排放在一起，研究了它们的主要功能，包括如何提取实例数据，返回一个 `dict`，如何获取字段的名称和默认值，以及如何根据现有实例创建新实例。

借此机会，我们第一次讲到类型提示，尤其是如何使用 Python 3.6 引入的表示法（“PEP 526—Syntax for Variable Annotations”）在 `class` 语句中注解变量。类型提示最让人惊讶的一方面应该是，它在运行时根本没有作用。毕竟，Python 是动态语言。如果想利用类型信息检测错误，则需要使用外部工具，例如 `Mypy`，对源码做静态分析。基本了解 PEP 526 引入的句法之后，我们研究了注解在普通类和通过 `typing.NamedTuple` 和 `@dataclass` 构建的类中起到什么效果。

接下来，我们探讨了 `@dataclass` 提供的常用功能，以及 `dataclasses.field` 函数的 `default_factory` 选项。我们还介绍了对数据类很重要的两个特殊的伪类型提示——`typing.ClassVar` 和 `dataclasses.InitVar`。随后，根据都柏林核心模式举了一个例子，说明如何在自定义的 `__repr__` 方法中使用 `dataclasses.fields` 迭代 `Resource` 实例的属性。

然后，告诫大家不要滥用数据类，以免违背面向对象编程的一个基本原则，即数据和处理数据的函数应放在同一个类中。不含逻辑的类可能表明你把逻辑放错位置了。

5.10 节将讲解如何使用模式匹配任意类的实例，而且不限于只匹配本章涵盖的类构建器构建的类。

5.10 延伸阅读

Python 标准库文档对数据类构建器的讲解很全面，还有许多小例子。

提议增加 `@dataclass` 的“PEP 557—Data Classes”，多数内容复制到了 `dataclasses` 模块的文档中。不过，PEP 557 中有几节信息丰富，却没有复制到文档中，包括“Why not just use `namedtuple`?”“Why not just use `typing.NamedTuple`?”和“Rationale”。“Rationale”一节最后提出了一个问题，并给出了解答。

什么时候不适合使用数据类？

需要兼容元组或字典的 API。需要的类型验证超出 PEP 484 和 PEP 526 定义的范围，或者需要验证值或做转换。

——Eric V. Smith
PEP 557, “Rationale”

Geir Arne Hjelle 写了一篇非常全面的文章，题为“Data Classes in Python 3.7+ (Guide)”。

PyCon US 2018, Raymond Hettinger 所做的演讲，“Dataclasses: The code generator to end all code generators”（视频）。

如果你需要更多更高级的功能，例如验证，则可以研究一下 Hynek Schlawack 创建的 `attrs` 项目。这个项目比 `dataclasses` 早几年，功能更多，承诺“把你从实现对象协议（即双下划线方法）的苦差事中解放出来，让你重拾编写类的乐趣”。Eric V. Smith 在 PEP 557 中特别感谢了 `attrs` 对 `@dataclass` 的影响。Smith 所指的影响可能包括最重要的 API 决策：使用类装饰器实现目的，而不使用基类和（或）元类。

Twisted 项目的创始人 Glyph 写的一篇文章对 `attrs` 做了精彩的介绍，题为“The One Python Library Everyone Needs”。`attrs` 的文档对替代方案也做了讨论。

图书作者、讲师和狂热的计算机科学家 Dave Beazley 编写的 `cluegen` 也是一个数据类生成器。如果你听过 Dave 的演讲，不难发现他是一位 Python 元编程专家。从 `cluegen` 项目的 README.md 文件给出的具体用例可以看出，尽管 Python 已经有 `@dataclass` 了，但他还要实现一种替代方案，以及他为什么提供一种解决方案，而不是一个工具。工具一开始是用着方便，但是方案更灵活，并适用于各种情况。

至于**数据类**是一种代码异味，我能找到的最好的佐证资料是 Martin Fowler 写的《重构（第 2 版）》一书。这一版去掉了本章开头引用的那句话，即“数据类就像小孩子……”，但仍不失为该书最好的版本。对 Python 程序员来说尤其如此，因为书中的示例用的是现代的 JavaScript，与 Python 接近，而不像第 1 版用的是 Java。

杂谈

“The Jargon File”中的“Guido”词条讲的是 Guido van Rossum。其中有一部分是这样说的：

你可能想象不到，除了 Python 之外，Guido 还有一个代表性作品——一台时间机器。人们声称 Guido 有这样一台设备，因为急躁的用户经常请求增加新功能，而得到的答复往往是“我昨晚刚刚实现了……”

很长一段时间以来，Python 缺少为类声明实例属性的标准句法，显得不便。而这在很多面向对象语言中有。下面是使用 Smalltalk 定义的 Point 类的一部分。

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'Kernel-BasicObjects'
```

第二行列出实例属性的名称，即 `x` 和 `y`。如果是类属性的话，则放在第三行。

Python 一直都有声明类属性的简便方式，如果类属性有初始值的话。然而，实例属性更常用，Python 程序员不得不在 `__init__` 方法中寻找有没有实例属性，而且总是担心类中的其他地方，甚至外部函数或其他类的方法也创建了实例属性。

现在好了，我们有了 `@dataclass`。

但是，问题也随之而来。

首先，使用 `@dataclass` 时不能省略类型提示。过去 7 年间，“PEP 484—Type Hints”给我们的承诺是，类型提示始终是可选的。而现在，这个重要的语言功能却要求必须提供类型提示。如果你不喜欢静态类型趋势，可以选择使用 `attrs`。

其次，PEP 526 提出的实例属性和类属性注解句法与我们在 `class` 语句中养成的习惯相反。以前，`class` 块顶层声明的全

是类属性（方法也是类属性）。而对于 PEP 526 和 `@dataclass`，在顶层声明的带有类型提示的属性变成了实例属性。

```
@dataclass
class Spam:
    repeat: int # 实例属性
```

下面的 `repeat` 也是实例属性。

```
@dataclass
class Spam:
    repeat: int = 99 # 实例属性
```

但是，如果没有类型提示，我们一下子就回到了从前，在类顶层声明的属性只属于类自身。

```
@dataclass
class Spam:
    repeat = 99 # 类属性!
```

最后，如果你想为类属性注解类型，则不能使用常规的类型，否则就变成实例属性了。正确的做法是使用伪类型 `ClassVar` 注解。

```
@dataclass
class Spam:
    repeat: ClassVar[int] = 99 # 真乱!
```

这是例外中的例外。我觉得这不太符合 Python 风格。

我没有参与 PEP 526 和“PEP 557—Data Classes”的讨论，我希望实现的是下面这种句法。

```
@dataclass
class HackerClubMember:
    .name: str
    .guests: list = field(default_factory=list)
```

```
.handle: str = ''  
  
all_handles = set()
```

❷

❶ 声明实例属性时必须在前面加上 `.`。

❷ 前面没有 `.` 的属性是类属性（像以前一样）。

为此，语法必须做出改变。我觉得这种句法的可读性非常高，而且没有例外中的例外。

真希望我能把 Guido 的时间机器借来一用，回到 2017 年，让核心团队采纳我的想法。

第 6 章 对象引用、可变性和垃圾回收

“你不开心，”白骑士用一种忧虑的声调说，“让我给你唱一首歌安慰你吧。……这首歌的曲名叫作《黑线鳕的眼睛》。”

“哦，那是一首歌的曲名，是吗？”爱丽丝问道，她试着使自己感到有兴趣。

“不，你不明白，”白骑士说，看来有些心烦的样子，“那是**人家这么叫的**曲名。真正的曲名是《老而又老的老头儿》。”

——Lewis Carroll

《爱丽丝镜中奇遇记》（有改动）

爱丽丝和白骑士为本章要讨论的内容定下了基调。本章的主题是对象与对象名称之间的区别。名称不是对象，名称就是名称。

本章先以一个比喻说明 Python 变量：变量是标注，而不是盒子。如果你不知道引用式变量是什么，可以像这样对别人解释别名。

然后，本章讨论对象标识、值和别名等概念。随后，本章揭露元组的一个神奇特征：元组虽不可变，但是其中的值可以改变。随后，引出浅拷贝和深拷贝。接下来的话题是引用和函数参数：可变的参数默认值导致的问题，以及如何安全地处理函数的调用方传入的可变参数。

本章最后几节讨论垃圾回收、`del` 命令，以及 Python 处理不可变对象的一些技巧。

本章内容有点儿枯燥，但这些话题是解决 Python 程序中很多不易察觉的 bug 的关键。

6.1 本章新增内容

本章涵盖的内容非常基础、非常稳定。第 2 版没有什么大变化。

6.3.1 节增加了一个示例，使用 `is` 测试哨符对象，另外还增加了一个警告框，告诫你不要错误使用 `is` 运算符。

本章原本在第四部分开头，第 2 版移到了前面，因为我觉得这一章更适合作为第二部分的结尾。

首先，我们要抛弃变量是存储数据的盒子这一错误观念。

6.2 变量不是盒子

1997 年夏天，我在 MIT 学了一门 Java 课程。Lynn Stein 教授¹指出，人们经常使用“变量是盒子”这样的比喻，但是这有碍于理解面向对象语言中的引用式变量。Python 变量类似于 Java 中的引用式变量，因此最好把它们理解为附加在对象上的标注。

¹Lynn Andrea Stein 是一位获奖的计算机科学教育工作者，目前在欧林工程学院任教。

在示例 6-1 所示的交互式控制台中，无法使用“变量是盒子”来解释。图 6-1 说明了在 Python 中为什么不能使用盒子来比喻变量，而便利贴才是变量的真正用途。

示例 6-1 变量 `a` 和 `b` 引用同一个列表，而不是那个列表的副本

```
>>> a = [1, 2, 3] ❶
>>> b = a         ❷
>>> a.append(4)    ❸
>>> b              ❹
[1, 2, 3, 4]
```

❶ 创建列表 `[1, 2, 3]`，绑定变量 `a`。

- ❷ 变量 `b` 绑定 `a` 引用的值。
- ❸ 修改 `a` 引用的列表，追加一项。
- ❹ 通过变量 `b` 可以看出效果。如果你认为 `b` 是一个盒子，存储盒子 `a` 中 `[1, 2, 3]` 的副本，那么这个行为就说不通了。

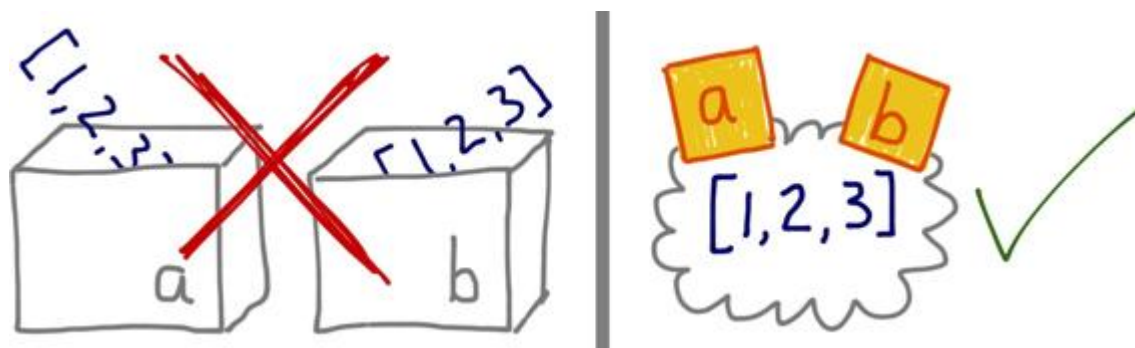


图 6-1：把变量想象为盒子，无法解释 Python 中的赋值。应该把变量视作便利贴，这样就好解释示例 6-1 中的行为了

因此，`b = a` 语句不是把 `a` 盒子中的内容复制到 `b` 盒子中，而是在标注为 `a` 的对象上再贴一个标注 `b`。

Stein 教授还反复讲解了赋值方式。例如讲到 `seesaw` 对象时，她会说“把变量 `s` 分配给 `seesaw`”，绝不会说“把 `seesaw` 分配给变量 `s`”。对引用式变量来说，说把变量分配给对象更合理，反过来说就有问题。毕竟，对象在赋值之前就创建了。示例 6-2 证明赋值语句的右边先执行。

动词“分配”自相矛盾，经常使用“绑定”代替。在 Python 中，赋值语句 `x = ...` 把名称 `x` 绑定到右边创建或引用的对象上。在绑定名称之前，对象必须存在，示例 6-2 证明了这一点。

示例 6-2 创建对象之后才能绑定变量

```
>>> class Gizmo:
...     def __init__(self):
```

```

...         print(f'Gizmo id: {id(self)}')
...
>>> x = Gizmo()
Gizmo id: 4301489152 ❶
>>> y = Gizmo() * 10 ❷
Gizmo id: 4301489432 ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
>>>
>>> dir() ❹
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'x']

```

- ❶ 输出的 `Gizmo id: ...` 是创建 `Gizmo` 实例的副作用。
- ❷ 在乘法运算中使用 `Gizmo` 实例会抛出异常。
- ❸ 这里表明，在尝试求积之前其实会创建一个新的 `Gizmo` 实例。
- ❹ 但是，肯定不会创建变量 `y`，因为在求解赋值语句的右边时抛出了异常。



为了理解 Python 中的赋值语句，应该始终先读右边。对象先在右边创建或获取，然后左边的变量才会绑定到对象上，就像给对象贴上标签一样。忘掉盒子吧！

因为变量只不过是标注，所以即使为对象贴上多个标注也没关系。多出来的标注就是**别名**，详见 6.3 节。

6.3 同一性、相等性和别名

Lewis Carroll 是 Charles Lutwidge Dodgson 教授的笔名。Carroll 先生指的就是 Dodgson 教授，是同一个人。示例 6-3 用 Python 表达这个概念。

示例 6-3 `charles` 和 `lewis` 指代同一个对象

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles ❶
>>> lewis is charles
True
>>> id(charles), id(lewis) ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950 ❸
>>> charles
{'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}
```

❶ lewis 是 charles 的别名。

❷ is 运算符和 id 函数确认了这一点。

❸ 向 lewis 中添加一项相当于向 charles 中添加一项。

然而，假如有一个冒充者，姑且叫他 Alexander Pedachenko 博士。Alexander 生于 1832 年，他声称自己是 Charles L. Dodgson。这个冒充者的证件可能是一样的，但是 Pedachenko 博士不是 Dodgson 教授。这种情况如图 6-2 所示。

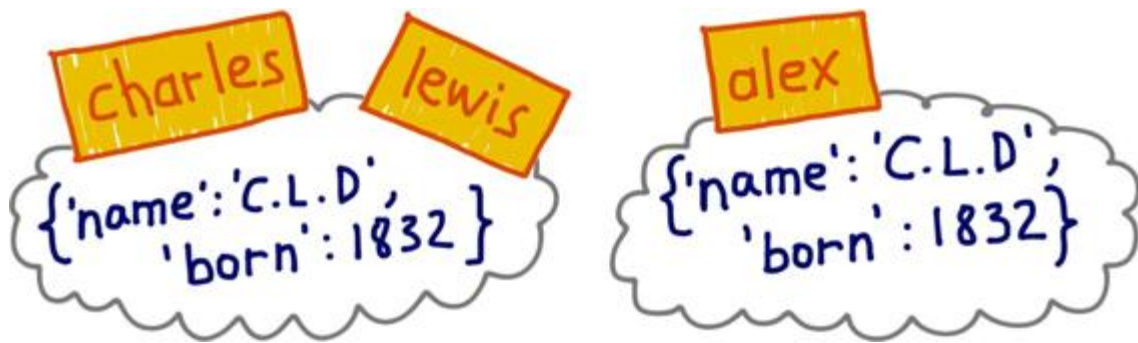


图 6-2: charles 和 lewis 绑定同一个对象，alex 绑定另一个具有相同内容的对象

示例 6-4 实现并测试图 6-2 中那个 alex 对象。

示例 6-4 经比较，alex 与 charles 相等，但 alex 绝对不是 charles

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance':  
950} ❶  
>>> alex == charles ❷  
True  
>>> alex is not charles ❸  
True
```

❶ alex 指代的对象与分配给 charles 的对象内容一样。

❷ 比较两个对象，结果相等，这是因为 dict 类的 `__eq__` 方法就是这样实现的。

❸ 但它们是不同的对象。在 Python 中，使用 `a is not b` 判断两个对象的标识是否不同。

示例 6-3 用到了**别名**。在那段代码中，`lewis` 和 `charles` 是别名，即两个变量绑定同一个对象。而 `alex` 不是 `charles` 的别名，因为二者绑定的是不同的对象。`alex` 和 `charles` 绑定的对象具有相同的值（`==` 比较的就是值），但是它们的标识不同。

《Python 语言参考手册》中的 3.1 节说道：

对象一旦创建，标识始终不变。可以把标识理解为对象在内存中的地址。`is` 运算符比较两个对象的标识，`id()` 函数返回对象标识的整数表示。

对象 ID 的真正意义取决于具体实现。在 CPython 中，`id()` 返回对象的内存地址，但是在其他 Python 解释器中可能是别的值。关键是，ID 一定是唯一的整数标注，而且在对象的生命周期内绝不会变。

实际编程中很少使用 `id()` 函数。对象的标识最常使用 `is` 运算符比较，无须直接调用 `id()` 函数。接下来讨论 `is` 和 `==` 的异同。



本书技术审校 Leonardo Rochael 指出，`id()` 最常用于调试，因为两个对象的字符串表示形式（`repr()`）可能是相似

的，而你需要判断二者究竟是别名还是指向不同的对象。在不同的上下文中，例如不同的栈帧中，不能使用 `is` 运算符比较两个引用。

6.3.1 在 `==` 和 `is` 之间选择

`==` 运算符比较两个对象的值（对象存储的数据），而 `is` 比较对象的标识。

编程时，我们关注的通常是值，而不是标识，因此在 Python 代码中 `==` 出现的频率比 `is` 高。

然而，比较一个变量和一个单例时，应该使用 `is`。目前，最常使用 `is` 检查变量绑定的值是不是 `None`。下面是推荐的写法：

```
x is None
```

否定的正确写法：

```
x is not None
```

`None` 是最常使用 `is` 测试的单例。哨符对象也是单例，同样使用 `is` 测试。下面是创建和测试哨符对象的一种方式。

```
END_OF_DATA = object()
# 省略很多行
def traverse(...):
    # 又省略很多行
    if node is END_OF_DATA:
        return
    # 等等.....
```

`is` 运算符比 `==` 速度快，因为它不能重载，所以 Python 不用寻找要调用的特殊方法，而是直接比较两个整数 ID。其实，`a == b` 是语法糖，等同于 `a.__eq__(b)`。继承自 `object` 的 `__eq__` 方法比较两个对象的 ID，结果与 `is` 一样。但是，多数内置类型使用更有意义的

方式覆盖了 `__eq__` 方法，把对象的属性值纳入考虑范围。相等性测试可能涉及大量处理工作，例如，比较大型集合或嵌套层级较深的结构时。



通常，我们更关注对象的相等性，而不是同一性。一般来说，`is` 运算符只用于测试 `None`。根据我审查代码的经验，`is` 的用法大多数是错的。如果你不确定，那就使用 `==`。这通常正是你想要的行为，而且也适用于 `None`，尽管速度没那么快。

在结束对同一性和相等性的讨论之前，我们来看看著名的不可变类型 `tuple`，它没有你所想象的那么一成不变。

6.3.2 元组的相对不可变性

元组与多数 Python 容器（列表、字典、集合等）一样，存储的是对象的引用。² 如果引用的项是可变的，即便元组本身不可变，项依然可以更改。也就是说，元组的不可变性其实是指 `tuple` 数据结构的物理内容（即存储的引用）不可变，与引用的对象无关。

²相比之下，`str`、`bytes` 和 `array.array` 等扁平序列存储的不是引用，而是在连续的内存中存储内容本身（字符、字节序列和数值）。

示例 6-5 表明，元组的值会随着引用的可变对象而变。元组中永远不变的是项的标识。

示例 6-5 一开始，`t1` 和 `t2` 相等，但是修改 `t1` 中的一个可变项后，二者不相等了

```
>>> t1 = (1, 2, [30, 40]) ❶
>>> t2 = (1, 2, [30, 40]) ❷
>>> t1 == t2 ❸
True
>>> id(t1[-1]) ❹
4302515784
>>> t1[-1].append(99) ❺
```

```
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1]) ❹
4302515784
>>> t1 == t2 ❶
False
```

- ❶ `t1` 不可变，但是 `t1[-1]` 可变。
- ❷ 构建元组 `t2`，所含的项与 `t1` 一样。
- ❸ 虽然 `t1` 和 `t2` 是不同的对象，但是二者相等——与预期相符。
- ❹ 查看 `t1[-1]` 列表的标识。
- ❺ 就地修改 `t1[-1]` 列表。
- ❻ `t1[-1]` 的标识没变，只是值变了。
- ❼ 现在，`t1` 和 `t2` 不相等。

元组的相对不可变性解释了 2.8.3 节的谜题。这也是有些元组不可哈希（见 3.4.1 节）的原因。

复制对象时，相等性和同一性之间的区别有更深层的影响。副本与源对象相等，但是 ID 不同。可是，如果对象中包含其他对象，那么应该复制内部对象吗？可以共享内部对象吗？这些问题没有唯一的答案。详见以下讨论。

6.4 默认做浅拷贝

复制列表（或多数内置的可变容器）最简单的方式是使用内置的类型构造函数。例如：

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1) ❶
```

```
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1 ❷
True
>>> l2 is l1 ❸
False
```

❶ `list(l1)` 创建 `l1` 的副本。

❷ 副本与源列表相等。

❸ 但是二者指代不同的对象。

对列表和其他可变序列来说，还可以使用简洁的 `l2 = l1[:]` 语句创建副本。

然而，构造函数或 `[:]` 做的是**浅拷贝**（即复制最外层容器，副本中的项是源容器中项的引用）。如果所有项都是不可变的，那么这种行为没有问题，而且还能节省内存。但是，如果有可变的项，可能会导致意想不到的问题。

在示例 6-6 中，我们对一个包含另一个列表和一个元组的列表做了浅拷贝，然后做些修改，看看对引用的对象有什么影响。



如果你手头有联网的计算机，强烈建议你在 Python Tutor 网站中查看示例 6-6 的交互式动画。写作本书时，我还无法直接提供 Python Tutor 网站中准备好的示例链接，不过这个工具很出色，值得花点时间复制粘贴代码。

示例 6-6 对一个包含另一个列表的列表做浅拷贝；建议把这段代码复制粘贴到 Python Tutor 网站中，看看动画效果

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1) ❶
l1.append(100) ❷
l1[1].remove(55) ❸
print('l1:', l1)
```

```
print('l2:', l2)
l2[1] += [33, 22] ④
l2[2] += (10, 11) ⑤
print('l1:', l1)
print('l2:', l2)
```

❶ l2 是 l1 的浅拷贝。此时的状态如图 6-3 所示。

❷ 把 100 追加到 l1 中，l2 不受影响。

❸ 把内部列表 l1[1] 中的 55 删除。这对 l2 有影响，因为 l2[1] 绑定的列表与 l1[1] 是同一个。

❹ 对可变的对象来说，例如 l2[1] 引用的列表，+= 运算符就地修改列表。这次修改在 l1[1] 中也有体现，因为它是 l2[1] 的别名。

❺ 对元组来说，+= 运算符创建一个新元组，然后重新绑定给变量 l2[2]。这等同于 $l2[2] = l2[2] + (10, 11)$ 。现在，l1 和 l2 中最后位置上的元组不是同一个对象，如图 6-4 所示。

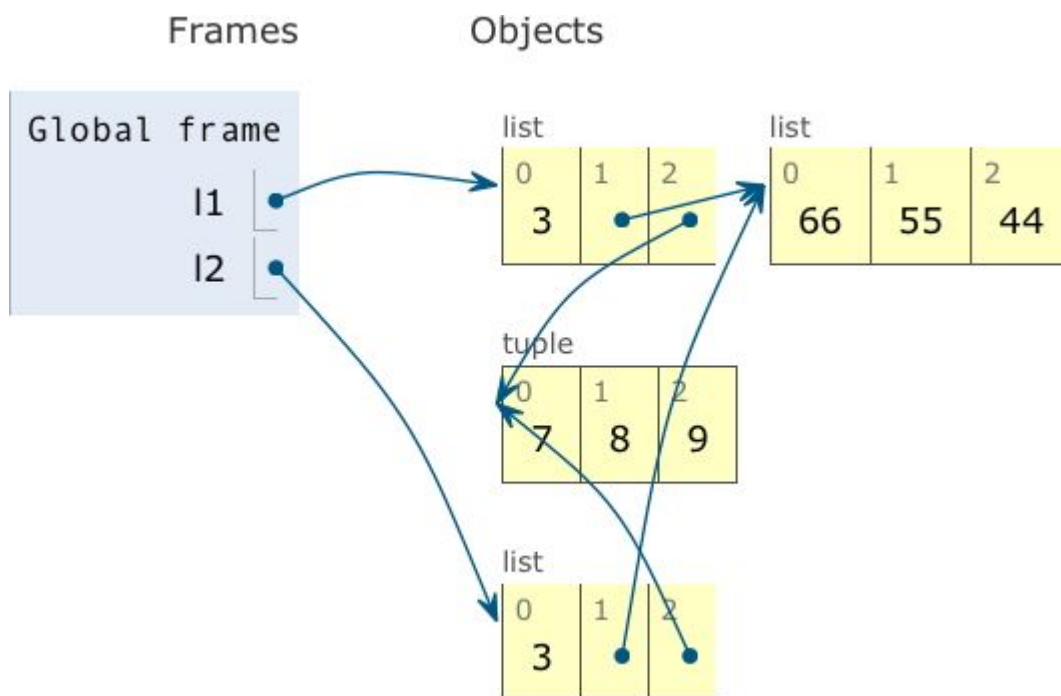


图 6-3: 示例 6-6 执行 `l2 = list(l1)` 赋值后的程序状态。l1 和 l2 引用不同的列表，但是这两个列表内的 `[66, 55, 44]` 列表和 `(7, 8, 9)` 元组引用的是相同的对象（截图由 Python Tutor 网站生成）

示例 6-6 的输出在示例 6-7 中，对象的最终状态如图 6-4 所示。

示例 6-7 示例 6-6 的输出

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

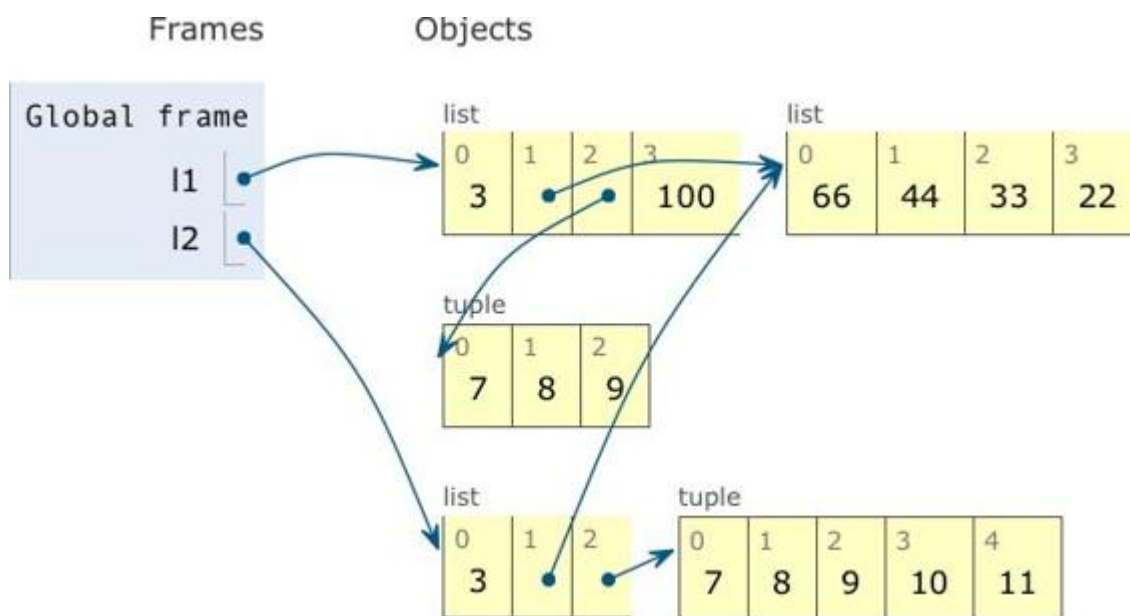


图 6-4: l1 和 l2 的最终状态：二者依然引用同一个列表对象，现在列表的值是 `[66, 44, 33, 22]`，不过 `l2[2] += (10, 11)` 操作创建一个新元组，内容是 `(7, 8, 9, 10, 11)`，它与 `l1[2]` 引用的元组 `(7, 8, 9)` 无关（截图由 Python Tutor 网站生成）

现在你应该明白了，浅拷贝操作简单，但是得到的结果可能并不是你想要的。接下来说明如何做深拷贝。

为任意对象做浅拷贝和深拷贝

浅拷贝通常来说没什么问题，但有时我们需要的是**深拷贝**（即副本不共享内部对象的引用）。`copy` 模块提供的 `copy` 和 `deepcopy` 函数分别对任意对象做浅拷贝和深拷贝。

为了演示 `copy()` 和 `deepcopy()` 的用法，示例 6-8 定义了一个简单的类，名为 `Bus`。这个类表示运载乘客的校车，乘客在途中有上有下。

示例 6-8 校车乘客在途中有上有下

```
class Bus:

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)
```

接下来，在示例 6-9 中的交互式控制台中，我们创建一个 `Bus` 对象（`bus1`）和两个副本，一个是浅拷贝副本（`bus2`），另一个是深拷贝副本（`bus3`），看看 `bus1` 有学生下车后会发生什么。

示例 6-9 `copy` 和 `deepcopy` 产生的不同效果

```
>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752) ❶
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David'] ❷
```

```
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
(4302658568, 4302658568, 4302657800) ❸
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David'] ❹
```

❶ 使用 `copy` 和 `deepcopy` 创建 3 个不同的 `Bus` 实例。

❷ `bus1` 中的 'Bill' 下车后, `bus2` 中也没有他了。

❸ 查看 `passengers` 属性后发现, `bus1` 和 `bus2` 共享同一个列表对象, 因为 `bus2` 是 `bus1` 的浅拷贝副本。

❹ `bus3` 是 `bus1` 的深拷贝副本, 因此它的 `passengers` 属性引用另一个列表。

注意, 一般来说, 深拷贝不是一件简单的事。如果对象有循环引用, 那么简单的算法会进入无限循环。`deepcopy` 函数会记住已经复制的对象, 因此能优雅地处理循环引用, 如示例 6-10 所示。

示例 6-10 循环引用: `b` 引用 `a`, 又把 `b` 追加到 `a` 中;
`deepcopy` 会想办法复制 `a`

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```

另外, 深拷贝有时可能太深了。例如, 对象可能会引用不该复制的外部资源或单例。可以实现特殊方法 `__copy__()` 和 `__deepcopy__()`, 来控制 `copy` 和 `deepcopy` 的行为, 详见 `copy` 模块的文档。

通过别名共享对象还能解释 Python 中传递参数的方式，以及使用可变类型作为参数默认值引起的问题。接下来讨论这些问题。

6.5 函数的参数是引用时

Python 唯一支持的参数传递模式是**共享传参**（call by sharing）。多数面向对象语言采用这一模式，包括 JavaScript、Ruby 和 Java（Java 的引用类型是这样，原始类型按值传参）。共享传参指函数的形参获得实参引用的副本。也就是说，函数内部的形参是实参的别名。

这种模式的结果是，函数可能会修改作为参数传入的可变对象，但是无法修改那些对象的标识（也就是说，不能把一个对象彻底替换成另一个对象）。示例 6-11 中有个简单的函数，它在形参上调用 += 运算符。分别把数值、列表和元组传给该函数，传入的实参会以不同的方式受到影响。

示例 6-11 函数可能会修改接收到的任何可变对象

```
>>> def f(a, b):
...     a += b
...     return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y ❶
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b ❷
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u) ❸
(10, 20, 30, 40)
```

```
>>> t, u
((10, 20), (30, 40))
```

❶ 数值 `x` 没变。

❷ 列表 `a` 变了。

❸ 元组 `t` 没变。

与函数参数相关的另一个问题是使用可变的值作为默认值，6.5.1 节将讨论这一点。

6.5.1 不要使用可变类型作为参数的默认值

可选参数可以有默认值，这是 Python 函数定义的一个很棒的特性，这样我们的 API 在演进的同时能保证向后兼容。然而，应该避免使用可变的对象作为参数的默认值。

下面以示例 6-12 说明这个问题。在示例 6-8 中 `Bus` 类的基础之上，改动一下 `__init__` 方法，定义 `HauntedBus` 类。这里，`passengers` 的默认值不是 `None`，而是 `[]`，免得在 `__init__` 方法中使用 `if` 语句检查。这个“聪明的举动”会让我们陷入麻烦。

示例 6-12 一个简单的类，说明可变默认值的危险

```
class HauntedBus:
    """一个受幽灵乘客折磨的校车模型"""

    def __init__(self, passengers=[]): ❶
        self.passengers = passengers ❷

    def pick(self, name):
        self.passengers.append(name) ❸

    def drop(self, name):
        self.passengers.remove(name)
```

❶ 如果没有传入 `passengers` 参数，则绑定默认列表对象（一开始是空列表）。

❷ 这个赋值语句把 `self.passengers` 变成 `passengers` 的别名。没有提供 `passengers` 参数时，`passengers` 是默认列表的别名。

❸ 在 `self.passengers` 上调用 `.remove()` 和 `.append()` 方法，修改的其实是默认列表，它是函数对象的一个属性。

`HauntedBus` 的诡异行为如示例 6-13 所示。

示例 6-13 备受“幽灵乘客”折磨的校车

```
>>> bus1 = HauntedBus(['Alice', 'Bill']) ❶
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers ❷
['Bill', 'Charlie']
>>> bus2 = HauntedBus() ❸
>>> bus2.pick('Carrie')
>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus() ❹
>>> bus3.passengers ❺
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers ❻
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers ❼
True
>>> bus1.passengers ❽
['Bill', 'Charlie']
```

❶ 一开始，`bus1` 有两位乘客。

❷ 目前没什么问题，`bus1` 没有异常。

- ③ 一开始, bus2 是空的, 因此把默认的空列表分配给 `self.passengers`。
- ④ bus3 一开始也是空的, 同样还是分配默认列表。
- ⑤ 但是默认列表不为空!
- ⑥ 登上 bus3 的 Dave 出现在 bus2 中。
- ⑦ 问题是, bus2.passengers 和 bus3.passengers 指代同一个列表。
- ⑧ 但 bus1.passengers 是不同的列表。

问题在于, 没有指定初始乘客的 HauntedBus 实例共享同一个乘客列表。

这种问题很难发现。如示例 6-13 所示, 实例化 HauntedBus 时, 如果传入乘客, 则一切正常。但是, 不为 HauntedBus 指定乘客的话, 奇怪的事就会发生, 这是因为 `self.passengers` 变成了 `passengers` 参数默认值的别名。出现这个问题的根源是, 默认值在定义函数时求解 (通常在加载模块时), 因此默认值变成了函数对象的属性。所以, 如果默认值是可变对象, 而且修改了它的值, 那么后续的函数调用都会受到影响。

运行示例 6-13 中的代码之后, 可以查看 `HauntedBus.__init__` 对象, 看看它的 `__defaults__` 属性中的那些“幽灵乘客”。

```
>>> dir(HauntedBus.__init__) # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'],)
```

最后, 我们可以验证 `bus2.passengers` 是一个别名, 绑定 `HauntedBus.__init__.__defaults__` 属性的第一个元素。

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

可变默认值导致的这个问题说明了为什么通常使用 `None` 作为接收可变值的参数的默认值。在示例 6-8 中，`__init__` 方法检查 `passengers` 参数的值是不是 `None`。如果是，则为 `self.passengers` 绑定一个新的空列表。6.5.2 节将说明为什么复制实参才是正解。

6.5.2 防御可变参数

如果你定义的函数接收可变参数，那就应该谨慎考虑调用方是否期望修改传入的参数。

例如，如果函数接收一个字典，而且在处理的过程中要修改它，那么这个副作用要不要体现到函数外部？具体问题具体分析。这其实需要函数的编写者和调用方达成共识。

下面是本章最后一个有关校车的示例。在这个示例中，`TwilightBus` 实例与客户共享乘客列表，这会产生意料之外的结果。分析具体实现之前，我们先从客户的角度看看 `TwilightBus` 类是如何工作的，如示例 6-14 所示。

示例 6-14 从 `TwilightBus` 下车后，乘客消失了

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat'] ❶
>>> bus = TwilightBus(basketball_team) ❷
>>> bus.drop('Tina') ❸
>>> bus.drop('Pat')
>>> basketball_team ❹
['Sue', 'Maya', 'Diana']
```

❶ `basketball_team` 中有 5 个学生的名字。

❷ 使用这队学生实例化 `TwilightBus`。

③ 一个学生从 bus 下车了，接着又有一个学生下车了。

④ 下车的学生从篮球队中消失了！

TwilightBus 违反了设计接口的最佳实践，即“最少惊讶原则”（Principle of least astonishment）。离开校车后，学生的名字就从篮球队的名单中消失了，这确实让人惊讶。

示例 6-15 是 TwilightBus 的实现，随后解释出现这个问题的原因。

示例 6-15 一个简单的类，说明接受可变参数的风险

```
class TwilightBus:
    """让乘客销声匿迹的校车"""

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = [] ❶
        else:
            self.passengers = passengers ❷

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name) ❸
```

❶ 这里谨慎处理，当 passengers 为 None 时，创建一个新的空列表。

❷ 然而，这个赋值语句把 self.passengers 变成 passengers 的别名，而后者是传给 __init__ 方法的实参（即示例 6-14 中的 basketball_team）的别名。

❸ 在 self.passengers 上调用 .remove() 和 .append() 方法，其实会修改传给构造方法的列表。

这里的问题是，校车为传给构造方法的列表创建了别名。正确的做法是，校车自己维护乘客列表。修正的方法很简单：像示例 6-8 那样，在 `__init__` 中，把传入的 `passengers` 参数的副本赋值给 `self.passengers`。

```
def __init__(self, passengers=None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers) ❶
```

❶ 创建 `passengers` 列表的副本；如果不是列表，就把它转换成列表。

在内部像这样处理乘客列表，就不会影响初始化校车时传入的参数了。此外，这种处理方式还更灵活：现在，传给 `passengers` 参数的值可以是元组或任何其他可迭代对象，例如一个 `set`，甚至数据库查询结果，因为 `list` 构造函数接受任何可迭代对象。自己创建并管理列表可以确保 `.pick()` 和 `.drop()` 方法内的 `.remove()` 和 `.append()` 操作能正常执行。



除非方法确实想修改通过参数传入的对象，否则在类中直接把参数赋值给实例变量之前一定要三思，因为这样会为参数对象创建别名。如果不确定，那就创建副本，免得给客户添麻烦。当然，创建副本会消耗一定的 CPU 和内存。但是，与速度和资源相比，在 API 中埋下难以察觉的 bug 显然是更严重的问题。

接下来谈一谈 Python 中最容易被误解的一个语句：`del`。

6.6 `del` 和垃圾回收

对象绝不会自行销毁；然而，对象不可达时，可能会被当作垃圾回收。

首先，你可能觉得奇怪，`del` 不是函数而是语句，写作 `del x` 而不是 `del(x)`。后一种写法也能起到作用，但这仅仅是因为在 Python 中，`x` 和 `(x)` 这两个表达式往往是同一个意思。

其次，`del` 语句删除引用，而不是对象。`del` 可能导致对象被当作垃圾回收，但是仅当删除的变量保存的是对象的最后一个引用时。重新绑定也可能导致对象的引用数量归零，致使对象被销毁。

```
>>> a = [1, 2] ❶
>>> b = a      ❷
>>> del a      ❸
>>> b          ❹
[1, 2]
>>> b = [3]    ❺
```

- ❶ 创建对象 `[1, 2]`，绑定变量 `a`。
- ❷ 变量 `b` 也绑定 `[1, 2]` 对象。
- ❸ 删除引用 `a`。
- ❹ `[1, 2]` 不受影响，因为还有 `b` 指向它。
- ❺ 把 `b` 重新绑定另一个对象，`[1, 2]` 的最后一个引用随之删除。现在，垃圾回收程序可以销毁 `[1, 2]` 了。



你可能听说过特殊方法 `__del__`，但是它不负责销毁实例，而且不应该在代码中调用。即将销毁实例时，Python 解释器调用 `__del__` 方法，给实例最后的机会释放外部资源。自己编写的代码很少需要实现 `__del__` 方法，有些 Python 程序员会花时间实现，但吃力不讨好，因为 `__del__` 方法不那么容易实现。详见《Python 语言参考手册》第 3 章中对特殊方法 `__del__` 的说明。

在 CPython 中，垃圾回收使用的主要算法是引用计数。实际上，每个对象都会统计有多少引用指向自己。当**引用计数**归零时，对象立即被销毁：CPython 在对象上调用 `__del__` 方法（如果定义了），然后释放分配给对象的内存。CPython 2.0 增加了分代垃圾回收算法，用于检测引用循环中涉及的对象组——如果一组对象之间全是相互引用，那么即使再出色的引用方式也会导致组中的对象不可达。有些 Python 的实现，垃圾回收程序更复杂，不依赖引用计数，这意味着对象的引用计数为零时可能不会立即调用 `__del__` 方法。A. Jesse Jiryu Davis 写的“PyPy, Garbage Collection, and a Deadlock”一文对 `__del__` 方法的恰当用法和不当用法做了讨论。

为了演示对象生命结束时的情形，示例 6-16 使用 `weakref.finalize` 注册一个回调函数，在销毁对象时调用。

示例 6-16 没有指向对象的引用时，监控对象生命结束时的情形

```
>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1 ①
>>> def bye(): ②
...     print('...like tears in the rain.')
...
>>> ender = weakref.finalize(s1, bye) ③
>>> ender.alive ④
True
>>> del s1
>>> ender.alive ⑤
True
>>> s2 = 'spam' ⑥
...like tears in the rain.
>>> ender.alive
False
```

① `s1` 和 `s2` 是别名，指向同一个集合 `{1, 2, 3}`。

② 这个函数一定不能是要销毁的对象的绑定方法，否则会有一个指向对象的引用。

- ③ 在 `s1` 引用的对象上注册 `bye` 回调。
- ④ 调用 `finalize` 对象之前，`.alive` 属性的值为 `True`。
- ⑤ 如前所述，`del` 不删除对象，而是删除对象的引用 `s1`。
- ⑥ 重新绑定最后一个引用 `s2`，让 `{1, 2, 3}` 不可达。对象被销毁了，调用了 `bye` 回调，`ender.alive` 的值变成了 `False`。

示例 6-16 的目的是明确指出 `del` 不删除对象，但是执行 `del` 操作后可能会导致对象不可达，从而使得对象被删除。

你可能觉得奇怪，为什么示例 6-16 中的对象 `{1, 2, 3}` 被销毁了？毕竟，我们把 `s1` 引用传给 `finalize` 函数了，而为了监控对象和调用回调，必须要有引用。这是因为，`finalize` 持有 `{1, 2, 3}` 的**弱引用**（weak reference）。对象的弱引用不增加对象的引用计数。因此，弱引用不阻碍目标对象被当作垃圾而回收。弱引用在缓存应用中用得到，因为我们不希望由于存在对缓存的引用而导致缓存的对象无法被删除。

6.7 Python 对不可变类型施加的把戏



你可以放心地跳过本节。这里讨论的是 Python 的实现细节，对 Python **用户**来说没那么重要，可能也不适用于 Python 的其他实现甚至 CPython 的未来版本。尽管如此，但我见过有人遇到极端情况时错误地使用 `is` 运算符，因此还是有必要讲一下。

我惊讶地发现，对元组 `t` 来说，`t[:]` 不创建副本，而是返回同一个对象的引用。此外，`tuple(t)` 获得的也是同一个元组的引用。³ 示例 6-17 可以证明这一点。

³文档明确指出了这个行为。在 Python 控制台中输入 `help(tuple)`，你会看到这句话：“如果参数是一个元组，则返回值是同一个对象。”在撰写这本书之前，我还以为自己对

元组无所不知。

示例 6-17 使用一个元组构建一个元组，得到的其实是同一个元组

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1 ❶
True
>>> t3 = t1[:]
>>> t3 is t1 ❷
True
```

❶ t1 和 t2 绑定同一个对象。

❷ t3 也是。

str、bytes 和 frozenset 实例也有这种行为。注意，frozenset 对象不是序列，因此 fs[:]（fs 是一个 frozenset 对象）不起作用。但是，fs.copy() 具有相同的效果：它会欺骗你，返回同一个对象的引用，而不是创建一个副本，如示例 6-18 所示。⁴

⁴copy 方法不复制对象，这是一个善意的谎言，为的是接口的兼容性，尽量保证 frozenset 与 set 兼容。两个不可变对象究竟是同一个对象还是副本，反正对最终用户来说没有区别。

示例 6-18 字符串字面量可能会创建共享的对象

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3) ❶
>>> t3 is t1 ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC' ❸
>>> s2 is s1 ❹
True
```

❶ 新建一个元组。

- ❷ `t1` 和 `t3` 相等，但不是同一个对象。
- ❸ 再新建一个字符串。
- ❹ 奇怪的事发生了，`a` 和 `b` 指代同一个 `str` 对象。

共享字符串字面量是一种优化措施，称为**驻留**（interning）。CPython 还会在小的整数上使用这个优化措施，防止重复创建“热门”数值，例如 0、1、-1 等。注意，CPython 不会驻留所有字符串和整数，驻留的条件是实现细节，而且没有文档说明。



千万不要依赖字符串或整数的驻留行为！比较字符串或整数是否相等时，应该使用 `==`，而不是 `is`。驻留是 Python 解释器内部使用的功能。

本节讨论的把戏，包括 `frozenset.copy()` 的行为，是“善意的谎言”，能节省内存，提升解释器的速度。别担心，这种行为不会给你添任何麻烦，因为只有不可变类型受到影响。或许这些细枝末节的最佳用途是与其他 Python 程序员打赌，提高自己的胜算。⁵

⁵如果在面试或认证考试中根据这些信息提出问题，那就太可怕了。千万别这么做，因为有更多更重要的 Python 知识点可以考查。

6.8 本章小结

每个 Python 对象都有标识、类型和值。只有对象的值不时变化。⁶

⁶其实，对象的类型也可以变，方法只有一个：为 `__class__` 属性指定其他类。但这是在作恶，我后悔加上这个脚注了。

如果两个变量指代的不可变对象具有相同的值（`a == b` 为 `True`），那么实际上它们指代的是副本还是同一个对象的别名基本没什么关系，因为不可变对象的值不会变，但有一个例外。这个例外是

不可变的容器，例如元组：如果不可变容器存储的项是可变对象的引用，那么可变项的值发生变化后，不可变容器的值也会随之改变。实际上，这种情况不是很常见。不可变容器不变的是所含对象的标识。`frozenset` 类不受这个问题的影响，因为 `frozenset` 对象中的元素必须可哈希，而按照定义，可哈希对象的值绝不可变。

变量保存的是引用，这一点对 Python 编程有很多实质影响。

- 简单的赋值不创建副本。
- 对 `+=` 或 `*=` 所做的增量赋值来说，如果左边的变量绑定的是不可变对象，则创建新对象；如果是可变对象，则就地修改。
- 为现有的变量赋予新值，不修改之前绑定的变量。这叫重新绑定：现在变量绑定了其他对象。如果变量是之前那个对象的最后一个引用，则对象被当作垃圾回收。
- 函数的形参以别名的形式传递，这意味着函数可能会修改通过实参传入的可变对象。这一行为无法避免，除非在本地创建副本，或者使用不可变对象（例如，传入元组，而不传入列表）。
- 使用可变类型作为函数参数的默认值有危险，因为如果就地修改了参数，默认值也就变了，这会影响后续使用默认值的调用。

在 CPython 中，对象的引用数量归零后，对象立即被销毁。如果除了循环引用之外没有其他引用，则循环引用的两个对象都被销毁。

某些情况下，可能需要存储对象的引用，但不留存对象本身。例如，有一个类想要记录当前的所有实例。这个需求可以使用弱引用实现。这是一种底层机制，是 `weakref` 模块中 `WeakValueDictionary`、`WeakKeyDictionary` 和 `WeakSet` 等有用的容器类，以及 `finalize` 函数的底层支持。

6.9 延伸阅读

《Python 语言参考手册》中第 3 章的开头清楚地解释了对对象的标识和值。

“Python 核心”系列图书的作者 Wesley Chun 在 EuroPython 2011 会上所做的演讲“Understanding Python's Memory Model, Mutability, and Methods”不仅涵盖了本章的主题，还讨论了特殊方法的使用。

Doug Hellmann 所写的两篇文章涵盖了本章讨论的部分话题：“copy – Duplicate Objects”和“weakref—Garbage-Collectable References to Objects”。

关于 CPython 分代垃圾回收程序的更多信息，参阅 gc 模块的文档。文档开头的第一句话是，“这个模块为可选的垃圾回收程序提供接口”。“可选的”这个修饰词可能让人惊讶，不过《Python 语言参考手册》第 3 章也说：

垃圾回收可以延缓实现，或者完全不实现——如何实现垃圾回收是实现的质量问题，只要不把依然可达的对象给回收了就行。

在《Python 开发者指南》中，Pablo Galindo 写的“Design of CPython's Garbage Collector”一文深入探讨了 Python 的垃圾回收机制，方便不同层次的贡献者了解 CPython 的实现，无论是新手还是老手。

CPython 3.4 的垃圾回收程序改进了对有 `__del__` 方法的对象的处理方式，详见“PEP 442—Safe object finalization”。

杂谈

平等对待所有对象

在发现 Python 之前，我学过 Java。我一直觉得 Java 的 `==` 运算符用着不顺手。程序员关注的基本上是相等性，而不是同一性，然而 Java 的 `==` 运算符比较的是对象（不含原始类型）的引用，而不是对象的值。即便是比较字符串这样的基本操作，Java 也强制使用 `.equals` 方法。尽管如此，`.equals` 方法还有另一个问题：对于 `a.equals(b)`，如果 `a` 是 `null`，则抛出空指针异

常。如果 Java 设计者觉得有必要重载字符串的 + 运算符，那为什么不把 == 也重载了呢？

Python 采用的方式是正确的，== 运算符比较对象的值，而 is 比较引用。此外，Python 支持重载运算符，== 能正确处理标准库中的所有对象，包括 None——这是一个正常的对象，与 Java 的 null 不同。

当然，你可以在自己的类中定义 __eq__ 方法，决定 == 如何比较实例。如果未覆盖 __eq__ 方法，那么从 object 继承的 __eq__ 方法会比较对象的 ID，因此这种后备机制认为用户定义的类的每一个实例都是不同的。

1998 年 9 月的一个下午，读完 *The Python Tutorial* 后，考虑到这种行为，我立即就从 Java 转到 Python 了。

可变性

如果所有 Python 对象都是不可变的，那么本章就没有存在的必要了。处理不可变的对象时，变量保存的是真正的对象还是共享对象的引用无关紧要。如果 a == b 成立，而且两个对象都不会变，那么它们就可能是相同的对象。这就是为什么字符串可以安全使用驻留。仅当对象可变时，对象标识才重要。

在“纯”函数式编程中，所有数据都是不可变的。为容器追加项，其实会创建新容器。Elixir 是一门务实的函数式语言，简单易学。这门语言中所有内置类型——包括列表——都是不可变的。

然而，Python 不是函数式语言，更别提纯不纯了。在 Python 中，用户定义的类，其实例默认可变（多数面向对象语言是如此）。自己创建对象时，如果需要不可变的对象，那么一定要额外小心。此时，对象的每个属性都必须是不可变的，否则会出现类似元组那种行为：元组对象的 ID 始终不变，但是如果元组中存有可变的对象，那么元组的值可能会变。

可变对象还是导致多线程编程难以处理的主要原因，因为某个线程改动对象后，如果不正确同步，就会损坏数据，但是过度同步又会导致死锁。Erlang 语言和平台（包括 Elixir）旨在最大限度地延长电信交换机等高并发分布式应用的正常运行时间，自然选择了不可变数据。

对象析构和垃圾回收

Python 没有直接销毁对象的机制，这样做其实也有道理：如果随时可以销毁对象，那么指向对象的现存引用怎么办？

CPython 的垃圾回收主要依靠引用计数，这样方便实现，但是遇到引用循环容易导致内存泄漏，因此 CPython 2.0（2000 年 10 月发布）实现了分代垃圾回收程序，它能把引用循环中不可达的对象销毁。

但是，引用计数仍然作为一种基准存在，一旦引用数量归零，就立即销毁对象。这意味着，在 CPython 中，像下面这样写是安全的（至少目前如此）。

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

这行代码是安全的，因为文件对象的引用数量会在 `write` 方法返回后归零，Python 立即关闭文件，然后销毁文件对象在内存中的表示。然而，这行代码在 Jython 或 IronPython 中不安全，因为它们使用的是宿主运行时（Java VM 和 .NET CLR）中的垃圾回收程序，那些回收程序更复杂，而且不依靠引用计数，销毁对象和关闭文件的时间可能更长。在任何情况下，包括 CPython，最好显式关闭文件，而关闭文件的最可靠方式是使用 `with` 语句，它能保证文件一定被关闭，即使打开文件时抛出了异常也无妨。使用 `with` 语句，上述代码片段可改为：

```
with open('test.txt', 'wt', encoding='utf-8') as fp:  
    fp.write('1, 2, 3')
```


如果你对垃圾回收程序感兴趣，可以阅读 Thomas Perl 的论文，“Python Garbage Collector Implementations: CPython, PyPy and GaS”。我就是从这篇论文中得知，`open().write()` 在 CPython 中是安全的。

参数传递：共享传参

解释 Python 中参数传递的方式时，人们经常这样说：“参数按值传递，但是这里的值是引用。”这么说没错，但会引起误解，因为在早期的语言中，最常用的参数传递模式有**按值传递**（函数得到实参的副本）和**按引用传递**（函数得到实参的指针）。在 Python 中，函数得到实参的副本，但是实参始终是引用。因此，如果引用的是可变对象，那么对象的值可能会被修改，但是对象的标识不变。此外，因为函数得到的是实参引用的副本，所以重新绑定对函数外部没有影响。读过《程序设计语言：实践之路（第 3 版）》（Michael L. Scott 著），尤其是 8.3.1 节“参数模式”之后，我决定采用**共享传参**（call by sharing）这个说法。

第二部分 函数即对象

- 第 7 章 函数是一等对象
- 第 8 章 函数中的类型提示
- 第 9 章 装饰器和闭包
- 第 10 章 使用一等函数实现设计模式

第 7 章 函数是一等对象

不管别人怎么说或怎么想，我从未觉得 Python 受到函数式语言太多的影响。我非常熟悉像 C 和 Algol 68 这样的命令式语言。虽然我把函数定为一等对象，但是并不把 Python 当作函数式编程语言。

——Guido van Rossum
Python 仁慈的“独裁者”¹

¹摘自 Guido 的 *The History of Python* 博客，题为“Origins of Python's 'Functional' Features”。

在 Python 中，函数是一等对象。编程语言研究人员把“一等对象”定义为满足以下条件的程序实体：

- 在运行时创建；
- 能赋值给变量或数据结构中的元素；
- 能作为参数传给函数；
- 能作为函数的返回结果。

在 Python 中，整数、字符串和字典都是一等对象——没什么特别的。像 Clojure、Elixir 和 Haskell 这样的函数式语言均把函数当作一等对象。不过，由于一等函数是个非常有用的功能，因此像 JavaScript、Go 和 Java（自 JDK 8 起）这样的流行语言也采用了这种设计，但这些语言都不算“函数式语言”。

本章以及第三部分的大多数章会探讨把函数视为对象的现实意义。



人们经常将“把函数视为一等对象”简称为“一等函数”。这样说并不完美，似乎表明这是函数中的特殊群体。在 Python 中，所有函数都是一等对象。

7.1 本章新增内容

本书第 1 版中的 5.4 节，即“7 种可调用对象”，在第 2 版中变成了“9 种可调用对象”。新增的可调用对象是原生协程和异步生成器，分别由 Python 3.5 和 Python 3.6 引入。这两种可调用对象将在第 21 章探讨，但是为了保证信息完整，7.5 节会有所提及。

新增“仅限位置参数”一节，涵盖 Python 3.8 添加的一个功能。

我把运行时访问函数注解相关的讨论移到了 15.5 节。写作第 1 版时，“PEP 484—Type Hints”还在研究中，没有统一的注解方式。从 Python 3.5 开始，注解应该遵守 PEP 484。因此，最好在讨论类型提示的章节说明。

现在来说明为什么 Python 函数是完备的对象。

7.2 把函数视为对象

示例 7-1 中的控制台会话表明，Python 函数就是对象。这里我们创建一个函数，然后调用它，读取它的 `__doc__` 属性，再确认函数对象本身是 `function` 类的实例。

示例 7-1 创建并测试一个函数，读取函数的 `__doc__` 属性，再检查函数的类型

```
>>> def factorial(n): ❶
...     """返回n!"""
...     return 1 if n < 2 else n * factorial(n - 1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__ ❷
'returns n!'
>>> type(factorial) ❸
<class 'function'>
```

- ❶ 这是一个控制台会话，因此是在“运行时”创建了一个函数。
- ❷ `__doc__` 是函数对象众多属性中的一个。
- ❸ `factorial` 是 `function` 类的实例。

`__doc__` 属性用于生成对象的帮助文本。在 Python 交互式控制台中，`help(factorial)` 命令输出的内容如图 7-1 所示。

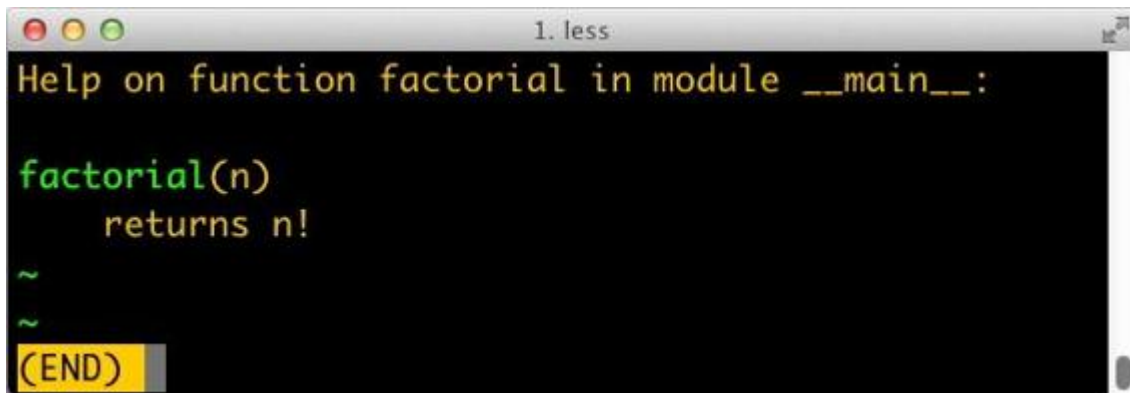


图 7-1: `factorial` 函数的帮助界面，输出的文本来自函数对象的 `__doc__` 属性

示例 7-2 展示了函数对象的“一等”本性。可以把 `factorial` 函数赋值给变量 `fact`，然后通过变量名调用。还可以把 `factorial` 函数作为参数传给 `map` 函数。`map(function, iterable)` 调用会返回一个可迭代对象，所含的项是把第一个参数（一个函数）应用到第二个参数（一个可迭代对象，这里是 `range(11)`）中各个元素上得到的结果。

示例 7-2 通过其他名称使用 `factorial` 函数，再把 `factorial` 函数作为参数传递

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
```

```
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(factorial, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

有了一等函数，便可以使用函数式风格编程。函数式编程的特色之一是高阶函数，详见 7.3 节。

7.3 高阶函数

接受函数为参数或者把函数作为结果返回的函数是**高阶函数**（higher-order function）。示例 7-2 中的 `map` 函数就是一例。此外，内置函数 `sorted` 也是：通过可选的 `key` 参数提供一个函数，应用到每一项上进行排序（参见 2.9 节）。如果想根据单词的长度排序，则只需把 `len` 函数传给 `key` 参数，如示例 7-3 所示。

示例 7-3 根据单词长度排序一个列表

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry',
'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

任何单参数函数都能作为 `key` 参数的值。例如，为了创建押韵词典，可以把各个单词反过来拼写，然后排序。注意，在示例 7-4 中，列表内的单词没有变，只是把反向拼写当作了排序条件，因此各种浆果都排在一起了。

示例 7-4 根据反向拼写排序一个单词列表

```
>>> def reverse(word):
...     return word[::-1]
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

在函数式编程范式中，最为人熟知的高阶函数有 `map`、`filter`、`reduce` 和 `apply`。`apply` 函数在 Python 2.3 中已弃用，在 Python 3 中已正式移除，因为用不到了。如果想使用不定量的参数调用函数，可以编写 `fn(*args, **kwargs)`，无须再编写 `apply(fn, args, kwargs)`。

`map`、`filter` 和 `reduce` 这 3 个高阶函数还能用到，不过在大多数使用场景中有更好的替代品，详见接下来的内容。

map、filter 和 reduce 的现代替代品

函数式语言通常会提供 `map`、`filter` 和 `reduce` 这 3 个高阶函数（有时使用不同的名称）。在 Python 3 中，`map` 和 `filter` 还是内置函数，但是由于引入了列表推导式和生成器表达式，因此二者就变得没那么重要了。列表推导式或生成器表达式兼具 `map` 和 `filter` 这两个函数的功能，而且代码可读性更高，如示例 7-5 所示。

示例 7-5 计算阶乘列表：map 和 filter 与列表推导式对比

```
>>> list(map(factorial, range(6))) ❶
[1, 1, 2, 6, 24, 120]
>>> [factorial(n) for n in range(6)] ❷
[1, 1, 2, 6, 24, 120]
>>> list(map(factorial, filter(lambda n: n % 2, range(6)))) ❸
[1, 6, 120]
>>> [factorial(n) for n in range(6) if n % 2] ❹
[1, 6, 120]
>>>
```

- ❶ 使用 $0! \sim 5!$ 构建一个阶乘列表。
- ❷ 使用列表推导式执行相同的操作。
- ❸ 使用 `map` 和 `filter` 计算直到 $5!$ 的奇数阶乘列表。

④ 使用列表推导式做相同的工作，换掉 `map` 和 `filter`，也无须使用 `lambda` 表达式。

在 Python 3 中，`map` 和 `filter` 返回生成器（一种迭代器），因此现在它们的直接替代品是生成器表达式。（在 Python 2 中，这两个函数返回列表，因此最接近的替代品是列表推导式。）

在 Python 2 中，`reduce` 是内置函数，但是 Python 3 把它放到 `functools` 模块里了。这个函数最常用于求和，但自 2003 年 Python 2.3 发布以来，内置函数 `sum` 在执行这项操作时效果更好。在可读性和性能方面，这是一项重大改善，如示例 7-6 所示。

示例 7-6 使用 `reduce` 和 `sum` 计算 0~99 的整数之和

```
>>> from functools import reduce ❶
>>> from operator import add ❷
>>> reduce(add, range(100)) ❸
4950
>>> sum(range(100)) ❹
4950
>>>
```

❶ 从 Python 3.0 开始，`reduce` 不再是内置函数。

❷ 导入 `add`，以免创建一个只求两数之和的函数。

❸ 计算 0~99 的整数之和。

❹ 使用 `sum` 执行同一个操作，无须导入并调用 `reduce` 和 `add`。



`sum` 和 `reduce` 的整体运作方式是一样的，即把某个操作连续应用到序列中的项上，累计前一个结果，把一系列值归约成一个值。

内置的归约函数还有 `all` 和 `any`。


```
all(iterable)
```

`iterable` 中没有表示假值的元素时返回 `True`。`all([])` 返回 `True`。

```
any(iterable)
```

只要 `iterable` 中有元素是真值就返回 `True`。`any([])` 返回 `False`。

12.7 节将详细说明 `reduce` 函数，届时我们会不断改进一个示例，为讨论提供有意义的上下文。17.10 节将重点讨论可迭代对象，届时会总结各个归约函数。

为了使用高阶函数，有时创建一次性的小型函数更便利。这便是匿名函数存在的原因，详见 7.4 节。

7.4 匿名函数

`lambda` 关键字使用 Python 表达式创建匿名函数。

然而，受 Python 简单的句法限制，`lambda` 函数的主体只能是纯粹的表达式。也就是说，`lambda` 函数的主体中不能有 `while`、`try` 等 Python 语句。使用 `=` 赋值也是一种语句，不能出现在 `lambda` 函数的主体中。可以有新出现的 `:=` 赋值表达式。不过，有这种赋值表达式的 `lambda` 函数可能太过复杂，可读性低，因此建议重构，改成使用 `def` 定义的常规函数。

在高阶函数的参数列表中最适合使用匿名函数。例如，示例 7-7 使用 `lambda` 表达式重写了示例 7-4 中排序押韵单词的示例，这样就省掉了 `reverse` 函数。

示例 7-7 使用 `lambda` 表达式反转拼写，然后依次给单词列表排序

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

除了作为参数传给高阶函数，Python 很少使用匿名函数。由于句法上的限制，非平凡的 `lambda` 表达式要么难以阅读，要么无法写出。如果发现 `lambda` 表达式的可读性低，强烈建议你遵照 Fredrik Lundh 的重构建议。

Fredrik Lundh 提出的 `lambda` 表达式重构秘诀

如果使用 `lambda` 表达式导致一段代码难以理解，Fredrik Lundh 建议像下面这样重构。

01. 编写注释，说明 `lambda` 表达式的作用。
02. 研究一会儿注释，找出一个名称来概括注释。
03. 把 `lambda` 表达式转换成 `def` 语句，使用那个名称来定义函数。
04. 删除注释。

以上步骤摘自“Functional Programming HOWTO”一文。这篇文章不可错过。

`lambda` 句法只是语法糖，`lambda` 表达式会像 `def` 语句一样创建函数对象。`lambda` 表达式只是 Python 中几种可调用对象的一种。7.5 节会说明所有可调用对象。

7.5 9 种可调用对象

除了函数，调用运算符 `()` 还可以应用到其他对象上。如果想判断对象能否调用，可以使用内置的 `callable()` 函数。数据模型文档列出了自 Python 3.9 起可用的 9 种可调用对象。

用户定义的函数

使用 `def` 语句或 `lambda` 表达式创建的函数。

内置函数

使用 C 语言（CPython）实现的函数，例如 `len` 或 `time.strftime`。

内置方法

使用 C 语言实现的方法，例如 `dict.get`。

方法

在类主体中定义的函数。

类

调用类时运行类的 `__new__` 方法创建一个实例，然后运行 `__init__` 方法，初始化实例，最后再把实例返回给调用方。Python 中没有 `new` 运算符，调用类就相当于调用函数。²

²通常，调用类会创建类的实例，不过，如果覆盖 `__new__` 方法，则也可能出现其他行为。22.2.3 节展示了一个例子。

类的实例

如果类定义了 `__call__` 方法，那么它的实例可以作为函数调用。详见 7.6 节。

生成器函数

主体中有 `yield` 关键字的函数或方法。调用生成器函数返回一个生成器对象。

原生协程函数

使用 `async def` 定义的函数或方法。调用原生协程函数返回一个协程对象。Python 3.5 新增。

异步生成器函数

使用 `async def` 定义，而且主体中有 `yield` 关键字的函数或方法。调用异步生成器函数返回一个异步生成器，供 `async for` 使用。Python 3.6 新增。

与其他可调用对象不同，生成器、原生协程和异步生成器函数的返回值不是应用程序数据，而是需要进一步处理的对象，要么产出应用程序数据，要么执行某种操作。生成器函数会返回迭代器（详见第 17 章）。原生协程函数和异步生成器函数返回的对象只能由异步编程框架（例如 `asyncio`）处理（详见第 21 章）。



Python 中有各种各样的可调用类型，因此判断对象能否调用，最安全的方法是使用内置函数 `callable()`。

```
>>> abs, str, 'Ni!'
(<built-in function abs>, <class 'str'>, 'Ni!')
>>> [callable(obj) for obj in (abs, str, 'Ni!')]
[True, True, False]
```

接下来阐述如何把类的实例变成可调用对象。

7.6 用户定义的可调用类型

不仅 Python 函数是真正的对象，而且任何 Python 对象都可以表现得像函数。为此，只需实现实例方法 `__call__`。

示例 7-8 实现的 `BingoCage` 类的实例可使用任何可迭代对象构建，内部存储一个随机排序的元素列表，调用实例从中取出一个元素。³

³有现成的 `random.choice` 可用，为什么还要定义 `BingoCage` 类呢？`choice` 函数可能会多次返回同一个元素，因为被选中的元素不从指定的容器中删除。而调用 `BingoCage` 绝不会返回重复结果，当然前提是填充实例的值各不相同。

示例 7-8 bingo.py: 调用 `BingoCage` 实例，从打乱顺序的列表中取出一个元素

```
import random

class BingoCage:

    def __init__(self, items):
        self._items = list(items) ❶
        random.shuffle(self._items) ❷

    def pick(self): ❸
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage') ❹

    def __call__(self): ❺
        return self.pick()
```

❶ `__init__` 接受任何可迭代对象。在本地构建一个副本，防止传入的列表参数有什么意外的副作用。

❷ `shuffle` 定能打乱顺序，因为 `self._items` 是列表。

❸ 起主要作用的方法。

❹ 如果 `self._items` 为空，就抛出异常，并设定错误消息。

❺ `bingo()` 是 `bingo.pick()` 的快捷方式。

下面的例子简单演示了如何使用示例 7-8 定义的行为。注意，`bingo` 实例可以作为函数调用，而且内置函数 `callable()` 判定它是可调对象。

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

实现 `__call__` 方法是创建类似函数的对象的简便方式，此时必须在内部维护一个状态，让它在多次调用之间存续，例如 `BingoCage` 中的剩余元素。`__call__` 的另一个用处是实现装饰器。装饰器必须可调用，而且有时要在多次调用之间“记住”某些事 [例如备忘 (memoization) ，即缓存消耗大的计算结果，供后面使用]，或者把复杂的操作分成几个方法实现。

在函数式编程中，创建保有内部状态的函数要使用闭包 (closure) 。闭包和装饰器将在第 9 章讨论。

下面探讨 Python 为声明函数形参和传入实参所提供的强大句法。

7.7 从位置参数到仅限关键字参数

Python 函数最好的功能之一是提供了极为灵活的参数处理机制。与之密切相关的是，调用函数时可以使用 `*` 和 `**` 拆包可迭代对象，映射各个参数。下面让我们通过示例 7-9 中的代码和示例 7-10 中的测试来展示这些功能。

示例 7-9 `tag` 函数用于生成 HTML 标签。可以使用名为 `class_` 的仅限关键字参数传入“class”属性，这是一种变通方法，因为“class”是 Python 中的关键字

```
def tag(name, *content, class_=None, **attrs):
    """生成一个或多个HTML标签"""
    if class_ is not None:
        attrs['class'] = class_
    attr_pairs = (f' {attr}="{value}"' for attr, value
                  in sorted(attrs.items()))
```

```

attr_str = ''.join(attr_pairs)
if content:
    elements = (f'<{name}{attr_str}>{c}</{name}>'
                for c in content)
    return '\n'.join(elements)
else:
    return f'<{name}{attr_str} />'

```

tag 函数的调用方式很多，如示例 7-10 所示。

示例 7-10 tag 函数（参见示例 7-9）众多调用方式中的几种

```

>>> tag('br') ❶
'<br />'
>>> tag('p', 'hello') ❷
'<p>hello</p>'
>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>
>>> tag('p', 'hello', id=33) ❸
'<p id="33">hello</p>'
>>> print(tag('p', 'hello', 'world', class_='sidebar')) ❹
<p class="sidebar">hello</p>
<p class="sidebar">world</p>
>>> tag(content='testing', name="img") ❺
'<img content="testing" />'
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'class': 'framed'}
>>> tag(**my_tag) ❻
''

```

- ❶ 传入单个位置参数，生成一个指定名称的空标签。
- ❷ 第一个参数后面的任意数量的参数被 `*content` 捕获，存入一个元组。
- ❸ tag 函数签名中没有明确指定名称的关键字参数被 `**attrs` 捕获，存入一个字典。
- ❹ `class_` 参数只能作为关键字参数传入。
- ❺ 第一个位置参数也能作为关键字参数传入。

⑥ 在 `my_tag` 前面加上 `**`，字典中的所有项作为参数依次传入，同名键绑定到对应的具名参数上，余下的则被 `**attrs` 捕获。在这个字典中可以使用 `'class'` 作为键，因为它是字符串，与保留字 `class` 不冲突。

仅限关键字参数是 Python 3 新增的功能。在示例 7-9 中，`class_` 参数只能通过关键字参数指定，它一定不会捕获无名位置参数。定义函数时，如果想指定仅限关键字参数，就要把它们放到前面有 `*` 的参数后面。如果不想支持数量不定的位置参数，但是想支持仅限关键字参数，则可以在签名中放一个 `*`，如下所示。

```
>>> def f(a, *, b):
...     return a, b
...
>>> f(1, b=2)
(1, 2)
>>> f(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 1 positional argument but 2 were given
```

注意，仅限关键字参数不一定要有默认值，可以像上例中的 `b` 那样，强制要求传入实参。

仅限位置参数

从 Python 3.8 开始，用户定义的函数签名可以指定仅限位置参数。内置函数都是如此，例如 `divmod(a, b)` 只能使用位置参数调用，不能写成 `divmod(a=10, b=4)`。

如果想定义只接受位置参数的函数，则可以在参数列表中使用 `/`。

下面的例子摘自“[What's New In Python 3.8](#)”，展示了如何模拟内置函数 `divmod` 的参数行为。

```
def divmod(a, b, /):
    return (a // b, a % b)
```


/ 左边均是仅限位置参数。在 / 后面，可以指定其他参数，处理方式一同往常。



在 Python 3.7 或之前的版本中，参数列表中的 / 将导致句法错误。

以示例 7-9 中的 `tag` 函数为例。如果希望 `name` 是仅限位置参数，可以在函数签名中添加一个 /，如下所示。

```
def tag(name, /, *content, class_=None, **attrs):  
    ...
```

“What's New In Python 3.8”和 PEP 570 中还有一些仅限位置参数的例子。

深入分析 Python 灵活的参数声明功能之后，本章余下的内容将介绍标准库为函数式编程提供支持的常用包。

7.8 支持函数式编程的包

虽然 Guido 明确表明，Python 的目标不是变成函数式编程语言，但是得益于一等函数、模式匹配，以及 `operator` 和 `functools` 等包的支持，其对函数式编程风格也可以“信手拈来”。接下来的 7.8.1 节和 7.8.2 节将分别介绍这两个包。

7.8.1 `operator` 模块

在函数式编程中，经常需要把算术运算符当作函数使用。例如，不使用递归计算阶乘。求和可以使用 `sum` 函数，求积则没有这样的函数。可以使用 `reduce` 函数（参见“`map`、`filter` 和 `reduce` 的现代替代品”一节），但是需要一个函数来计算序列中两项之积。示例 7-11 展示了如何使用 `lambda` 表达式解决这个问题。

示例 7-11 使用 reduce 函数和匿名函数计算阶乘

```
from functools import reduce

def factorial(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

`operator` 模块为多个算术运算符提供了对应的函数，无须再动手编写像 `lambda a, b: a*b` 这样的匿名函数。使用算术运算符函数，可以把示例 7-11 改写成示例 7-12 那样。

示例 7-12 使用 reduce 函数和 `operator.mul` 函数计算阶乘

```
from functools import reduce
from operator import mul

def factorial(n):
    return reduce(mul, range(1, n+1))
```

`operator` 模块中还有一类函数，即工厂函数 `itemgetter` 和 `attrgetter`，能替代从序列中取出项或读取对象属性的 `lambda` 表达式。

示例 7-13 展示了 `itemgetter` 的常见用途：根据元组的某个字段对元组列表进行排序。在这个示例中，我们按照国家代码（第 2 个字段）的顺序打印各个城市的信息。其实，`itemgetter(1)` 会创建一个接受容器的函数，返回索引位 1 上的项。与作用相同的 `lambda fields: fields[1]` 相比，使用 `itemgetter` 更容易写出代码，而且可读性更高。

示例 7-13 演示使用 `itemgetter` 排序一个元组列表（数据来自示例 2-8）

```
>>> metro_data = [
...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
...     ('New York-Newark', 'US', 20.104, (40.808611,
```

```

-74.020386)),
...     ('São Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('São Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))

```

如果传给 `itemgetter` 多个索引参数，那么 `itemgetter` 构建的函数就会返回提取的值构成的元组，以方便根据多个键排序。

```

>>> cc_name = itemgetter(1, 0)
>>> for city in metro_data:
...     print(cc_name(city))
...
('JP', 'Tokyo')
('IN', 'Delhi NCR')
('MX', 'Mexico City')
('US', 'New York-Newark')
('BR', 'São Paulo')
>>>

```

`itemgetter` 使用 `[]` 运算符，因此它不仅支持序列，还支持映射和任何实现 `__getitem__` 方法的类。

与 `itemgetter` 的作用类似，`attrgetter` 创建的函数会根据名称来提取对象的属性。如果传给 `attrgetter` 多个属性名，那么它也会返回由提取的值构成的元组。此外，如果参数名中包含 `.`（点号），那么 `attrgetter` 就会深入嵌套对象，检索属性。这些行为如示例 7-14 所示。这个控制台会话不短，因为我们要构建一个嵌套结构，以展示 `attrgetter` 如何处理包含点号的属性名。

示例 7-14 演示使用 `attrgetter` 处理前文定义的具名元组 `metro_data`（会用到示例 7-13 定义的列表）

```

>>> from collections import namedtuple
>>> LatLon = namedtuple('LatLon', 'lat lon') ❶
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord') ❷
>>> metro_areas = [Metropolis(name, cc, pop, LatLon(lat, lon)) ❸
...                 for name, cc, pop, (lat, lon) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933,
coord=LatLon(lat=35.689722,
lon=139.691667))
>>> metro_areas[0].coord.lat ❹
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat') ❺
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')):
❻
...     print(name_lat(city)) ❼
...
('São Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)

```

❶ 使用 `namedtuple` 定义 `LatLon`。

❷ 再定义 `Metropolis`。

❸ 使用 `Metropolis` 实例构建 `metro_areas` 列表。注意，这里会使用嵌套的元组拆包提取 `(lat, lon)`，然后使用它们构建 `LatLon`，作为 `Metropolis` 的 `coord` 属性。

❹ 深入 `metro_areas[0]`，获取它的纬度。

❺ 定义一个 `attrgetter`，获取 `name` 属性和嵌套的 `coord.lat` 属性。

❻ 再次使用 `attrgetter`，按照纬度排序城市列表。

❼ 使用 ❺ 中定义的 `attrgetter`，只显示城市名和纬度。

下面是 `operator` 模块中定义的部分函数列表（省略了以 `_` 开头的名称，因为它们基本上是实现细节）。

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imatmul',
'imod', 'imul', 'index', 'indexOf', 'inv', 'invert', 'ior',
'ipow', 'irshift', 'is_', 'is_not', 'isub', 'itemgetter',
'itruediv', 'ixor', 'le', 'length_hint', 'lshift', 'lt', 'matmul',
'methodcaller', 'mod', 'mul', 'ne', 'neg', 'not_', 'or_', 'pos',
'pow', 'rshift', 'setitem', 'sub', 'truediv', 'truth', 'xor']
```

这 54 个名称中大部分名称的作用不言而喻。以 `i` 开头、后面是另一个运算符的那些名称（例如 `iadd`、`iand` 等），对应的是增量赋值运算符（例如 `+=`、`&=` 等）。如果第一个参数是可变的，那么这些函数就会就地修改第一个参数；否则，作用与不带 `i` 的函数一样，直接返回运算结果。

在 `operator` 模块余下的函数中，最后介绍一下 `methodcaller`。它的作用与 `attrgetter` 和 `itemgetter` 类似，即创建函数。`methodcaller` 创建的函数会在对象上调用参数指定的方法，如示例 7-15 所示。

示例 7-15 `methodcaller` 使用示例：第二个测试会展示绑定额外参数的方式

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
'THE TIME HAS COME'
>>> hyphenate = methodcaller('replace', ' ', '-')
>>> hyphenate(s)
'The-time-has-come'
```

示例 7-15 中的第一个测试只是为了展示 `methodcaller` 的用法。如果想把 `str.upper` 作为函数使用，则只需在 `str` 类上调用，传入一个字符串参数即可，如下所示。

```
>>> str.upper(s)
'THE TIME HAS COME'
```

示例 7-15 中的第二个测试表明，`methodcaller` 还可以冻结某些参数，也就是部分应用程序（partial application），这与 `functools.partial` 函数的作用类似，详见 7.8.2 节。

7.8.2 使用 `functools.partial` 冻结参数

`functools` 模块提供了一系列高阶函数，比如 7.3 节“map、filter 和 reduce 的现代替代品”中用过的 `reduce`。另外一个值得关注的函数是 `partial`，它可以根据提供的可调用对象产生一个新可调用对象，为原可调用对象的某些参数绑定预定的值。使用这个函数可以把接受一个或多个参数的函数改造成需要更少参数的回调的 API。示例 7-16 是一个简单演示。

示例 7-16 使用 `partial` 把一个双参数函数改造成只需要一个参数的可调用对象

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3) ❶
>>> triple(7) ❷
21
>>> list(map(triple, range(1, 10))) ❸
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

- ❶ 使用 `mul` 创建 `triple` 函数，把第一个位置参数绑定为 3。
- ❷ 测试 `triple` 函数。
- ❸ 在 `map` 中使用 `triple`，在这个示例中不能使用 `mul`。

使用 4.7 节中讲过的 `unicode.normalize` 函数再举一个例子，这个示例更有实际意义。处理包含多国语言的文本时，你可能想在比较或排序之前使用 `unicode.normalize('NFC', s)` 规范化字符串

s。如果经常需要这么做，则可以定义一个 `nfc` 函数，如示例 7-17 所示。

示例 7-17 使用 `partial` 构建一个便利的 Unicode 规范化函数

```
>>> import unicodedata, functools
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

`partial` 的第一个参数是一个可调用对象，后面跟着任意个要绑定的位置参数和关键字参数。

示例 7-18 在示例 7-9 中定义的 `tag` 函数上使用 `partial` 冻结了一个位置参数和一个关键字参数。

示例 7-18 把 `partial` 应用到示例 7-9 中定义的 `tag` 函数上

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0> ❶
>>> from functools import partial
>>> picture = partial(tag, 'img', class_='pic-frame') ❷
>>> picture(src='wumpus.jpeg')
'' ❸
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img',
class_='pic-frame') ❹
>>> picture.func ❺
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
>>> picture.keywords
{'class_': 'pic-frame'}
```

❶ 从示例 7-9 中导入 `tag` 函数，查看它的 ID。

❷ 使用 `tag` 创建 `picture` 函数，把第一个位置参数固定为 `'img'`，把关键字参数 `class_` 固定为 `'pic-frame'`。

❸ `picture` 的行为符合预期。

❹ `partial()` 返回一个 `functools.partial` 对象。⁴

⁴`functools.py` 的源码表明，`functools.partial` 是使用 C 语言实现的，而且默认使用这个实现。如果这个实现不可用，则可以使用从 Python 3.4 起 `functools` 模块为 `partial` 提供的纯 Python 实现。

❺ `functools.partial` 对象提供了访问原函数和固定参数的属性。

`functools.partialmethod` 函数的作用与 `partial` 一样，不过其用于处理方法。

`functools` 模块中还有一些高阶函数可用作函数装饰器，例如 `cache`、`singledispatch` 等。第 9 章将介绍这些函数，同时还将探讨如何自定义装饰器。

7.9 本章小结

本章的目标是探讨 Python 函数的一等本性。这意味着，可以把函数赋值给变量、传给其他函数、存储在数据结构中，以及访问函数的属性，供框架和一些工具使用。

高阶函数是函数式编程的重要组成部分，Python 中也经常用到，比如内置函数 `sorted`、`min` 和 `max`，以及标准库中的 `functools.partial`。即使现在不像以前那样经常使用 `map`、`filter` 和 `reduce` 等函数了，但是还有列表推导式（以及类似的结构，例如生成器表达式），以及 `sum`、`all` 和 `any` 等内置的归约函数。

自 Python 3.6 起，Python 有 9 种可调用对象，从 `lambda` 表达式创建的简单函数，到实现 `__call__` 方法的类实例。生成器和协程也是可调用对象，不过行为与其他可调用对象差异较大。可调用对象都能通过内置函数 `callable()` 检测。可调用对象支持丰富的形参声明句法，包括仅限关键字参数、仅限位置参数和注解。

最后，本章介绍了 `operator` 模块中的一些函数以及 `functools.partial` 函数。有了这些函数，就不太需要使用功能有限的 `lambda` 表达式实现函数式编程了。

7.10 延伸阅读

接下来的 3 章将继续探讨编程中对函数对象的使用：第 8 章专门说明函数参数和返回值的类型注解；第 9 章深入讲解函数装饰器（一种特殊的高阶函数），以及背后用到的闭包机制；第 10 章介绍一等函数如何简化某些经典的面向对象设计模式。

《Python 语言参考手册》中的 3.2 节“标准类型层次结构”介绍了 9 种可调用类型和其他所有内置类型。

《Python Cookbook（第 3 版）中文版》的第 7 章采用不同的方式探讨了相关概念，对本章和第 9 章是不错的补充。

如果对仅限关键字参数的基本原理和使用场景感兴趣，请阅读“PEP 3102—Keyword-Only Arguments”。

A. M. Kuchling 写的文章“Python Functional Programming HOWTO”对 Python 函数式编程做了很好的介绍。不过，该文章的重点是使用迭代器和生成器（参见第 17 章）。

Stack Overflow 网站中的问题“Python: Why is `functools.partial` necessary?”有个翔实而有趣的回答，答主是 Alex Martelli，他是经典的 *Python in a Nutshell* 一书的作者。

“Python 是一门函数式语言吗？”我把对这个问题的思考汇集到一个演讲中——“Beyond Paradigms”。这个演讲在 PyCaribbean、PyBay 和 PyConDE 上都讲过，是我最得意的成就之一。在这次大会上我遇到了后来担任本书技术审校的 Miroslav Šedivý 和 Jürgen Gmach。

杂谈

Python 是一门函数式语言吗？

2000 年的某一天，我参加了 Zope 公司在美国组织的一场讲习班，期间 Guido van Rossum 到访了教室（他不是讲师）。在课后的问答环节，有人问他 Python 的哪些功能是从其他语言借鉴而来的。他答道：“Python 中一切好的功能都是从其他语言中借鉴来的。”

布朗大学计算机科学教授 Shriram Krishnamurthi 在其论文“Teaching Programming Languages in a Post-Linnaean Age”的开头这样写道：

编程语言“范式”已近末日，是旧时代的遗留物，令人厌烦。既然现代语言的设计者对范式不屑一顾，那么我们的课程为什么要像奴隶一样对其言听计从呢？

在该论文中，下面这一段点名提到了 Python：

对 Python、Ruby 或 Perl 这些语言还要了解什么呢？它们的设计者没有耐心去精确实现林奈层次结构，设计者按照自己的意愿从别处借鉴功能，创建出完全无视过往概念的大杂烩。

Krishnamurthi 指出，不要试图把语言归为某一类；相反，应把语言视作功能的聚合。7.10 节提到的“Beyond Paradigms”演讲就受到了他的影响。

为 Python 提供一等函数打开了函数式编程的大门，不过这并不是 Guido 的本意。他在“Origins of Python's Functional Features”一文中说，`map`、`filter` 和 `reduce` 的最初目的是为 Python 增加 `lambda` 表达式。这些功能都由 Amrit Prem 贡献，添加在 1994 年发布的 Python 1.0 中（参见 CPython 源码中的 Misc/HISTORY 文件）。

`map`、`filter` 和 `reduce` 等函数首次出现在 Lisp 中，这是最早的一门函数式语言。然而，Lisp 不限制在 `lambda` 表达式中能做什么，因为 Lisp 中的一切都是表达式。Python 使用的是面向语句的句法，表达式中不能包含语句，而很多语言结构是语句——包括 `try/catch`，我编写 `lambda` 表达式时最想念这个语句。Python 为了提高句法的可读性，必须付出这样的代价。⁵ Lisp 有很多优点，可读性一定不是其中之一。

讽刺的是，从另一门函数式语言（Haskell）中借用列表推导式之后，Python 对 `map`、`filter`，以及 `lambda` 表达式的需求极大地减少了。

除了匿名函数句法上的限制，影响函数式编程惯用法在 Python 中广泛使用的最大障碍是缺少尾调用消除（tail-call elimination）。这是一项优化措施，在函数的主体“末尾”递归调用，从而提高计算函数的内存使用效率。Guido 在另一篇博客文章（“Tail Recursion Elimination”）中解释了为什么这种优化措施不适合 Python。该文章详细讨论了技术论证，不过前 3 个（也是最重要的原因）与易用性有关。Python 作为一门易于使用、学习和教授的语言并非偶然，有 Guido 在为我们把关。

综上所述，从设计上看，不管函数式语言的定义如何，Python 都不是一门函数式语言。它只是从函数式语言中借鉴了一些好的想法。

匿名函数的问题

除了 Python 独有的句法上的局限，对任何一门语言来说，匿名函数都有一个严重的缺点：没有名称。

我是半开玩笑的。函数有名称，栈跟踪更易于阅读。匿名函数是一种便利的简洁方式，人们乐于使用它们，但是有时会忘乎所以，尤其是在鼓励深层嵌套匿名函数的语言和环境中的，例如 Node.js 之上的 JavaScript。匿名函数嵌套的层级太深，不利于调试和处理错误。Python 中的异步编程结构更好，或许就是因为受 `lambda` 句法的限制，想滥用都不可能，必须使用更明确的方式。现代异步 API 开始使用 `promise`、`future` 和 `deferred` 等概念。加上协程，我们终于逃脱了“回调地狱”。我保证，后面会进一步讨论异步编程，但是必须等到第 21 章。

⁵此外，还有一个问题：把代码粘贴到 Web 论坛时，缩进会丢失。当然，这是题外话。

第 8 章 函数中的类型提示

还应该强调的是，Python **仍是一门动态类型语言**，作者并不意图强制使用类型提示，这只是一种约定。

——Guido van Rossum、Jukka Lehtosalo 和 Łukasz Langa
“PEP 484—Type Hints”¹

¹“Rationale and Goals”，保留原文中的加粗强调。

自 2001 年发布的 Python 2.2 统一类型和类之后，类型提示是 Python 历史发展过程中最大的变化。但是，不是所有 Python 用户都能从类型提示中受益。因此，必须把这作为一种可选的功能。

“PEP 484—Type Hints”为函数参数、返回值和变量的显式类型声明规定了句法和语义，目标是协助开发者工具通过静态分析（例如，不通过测试真正运行代码）发现 Python 基准代码中的 bug。

类型提示的主要受益者是使用 IDE（Integrated Development Environment，集成开发环境）和 CI（Continuous Integration，持续集成）的专业软件工程师。这类人群看重的是类型提示的成本效益分析，不是所有 Python 用户都关注这一点。

Python 的用户群体分布广泛，包括科学家、交易员、记者、艺术家、创客、分析师和许多学科的学生等。对他们中的大多数人来说，学习类型提示的成本可能较高，除非以前用过具有静态类型、子类型和泛型的语言。考虑到他们使用 Python 的方式，以及基准代码和团队（通常是“一人团队”）的规模较小，这些用户的收益较低。Python 默认的动态类型更简单，也更具表现力，特别适合在数据科学、创造性计算和学习中编写探索数据和想法的代码。

本章重点讲解 Python 函数签名中的类型提示。第 15 章将探讨类中的类型提示以及 `typing` 模块的其他功能。

本章主要涵盖以下内容：

- 通过 Mypy，以实践的方式介绍渐进式类型；
- 鸭子类型和名义类型的互补作用；
- 概述可以出现在注解中的主要类型——约占本章 60% 的篇幅；
- 为变长参数 (`*args` 和 `**kwargs`) 添加类型提示；
- 类型提示和静态类型的局限及缺点。

8.1 本章新增内容

这是全新的一章。我写完本书第 1 版后发布的 Python 3.5 中才出现类型提示。

考虑到静态类型系统的局限性，PEP 484 只能引入一种**渐进式类型系统** (gradual type system)。首先为这个概念下一个定义。

8.2 关于渐进式类型

PEP 484 为 Python 引入了一种**渐进式类型系统**。其他语言也有使用渐进式类型系统的，例如 Microsoft 的 TypeScript、Dart (Flutter SDK 使用的语言，由谷歌创建) 和 Hack (PHP 的一种方言，由 Facebook 的 HHVM 虚拟机支持)。类型检查工具 Mypy 最初也是一门语言，是 Python 的一种方言，有自己的解释器，支持渐进式类型。后经 Guido van Rossum 的劝说，Mypy 的创建者 Jukka Lehtosalo 把它改造成成了一个检查 Python 代码注解的工具。

渐进式类型系统具有以下性质。

是可选的

默认情况下，类型检查工具不应对没有类型提示的代码发出警告。当类型检查工具无法确定对象的类型时，会假定其为 `Any` 类型。`Any` 类型与其他所有类型兼容。

不在运行时捕获类型错误

类型提示相关的问题由静态类型检查工具、lint 程序和 IDE 捕获。在运行时不能阻止把不一致的值传给函数或分配给变量。

不能改善性能

类型注解提供的数据在理论上可以优化生成的字节码，但是据我所知，截至 2021 年 7 月，任何 Python 运行时都没有实现这种优化。²

²PyPy 的即时编译器提供的数据比类型提示丰富很多，能在 Python 程序运行过程中对其进行监控，检测具体使用的类型，优化生成的机器码。

对渐进式类型来说，注解始终是可选的，这个性质最能体现可用性。

在静态类型系统中，大多数类型约束很容易表达，但是也有许多类型约束很难表达，有些是不易表达，有些则根本表达不出来。³ 你写的 Python 代码可能质量很高，也有较好的测试覆盖率，能顺利通过测试，但就是无法添加类型提示，让类型检测工具满意。这没什么关系，类型提示有瑕疵就随它去吧，不影响产品发布。

³例如，截至 2021 年 7 月，不支持递归类型，详见 `typing` 模块的 182 号工单“Define a JSON type”以及 `Mypy` 的 731 号工单“Support recursive types”。

类型提示在所有层面上均是可选的，一整个包都可以没有类型提示，即便有类型提示，导入模块时也可以让类型检查工具保持静默，另外还可以通过特殊的注释让类型检查工具忽略代码中指定的行。



100% 的类型提示覆盖率太过激进，只是一味追求指标，不现实，也有碍团队充分利用 Python 的强大功能和灵活性。应该坦然接受没有类型提示的代码，防止注解扰乱 API，增加实现难度。

8.3 渐进式类型实践

接下来，我们逐步为一个简单的函数添加类型提示，使用 Mypy 检查，实际体验一下渐进式类型系统。



兼容 PEP 484 的 Python 类型检查工具很多，比如谷歌的 pytype、微软的 Pyright、Facebook 的 Pyre，以及 PyCharm 等 IDE 内置的类型检查器。本节的示例选择 Mypy 是因为它最出名。可以根据具体的项目或团队的喜好从其他几个工具中做出选择。Pytype 就是不错的选择，对于没有类型提示的基准代码，Pytype 仍能提供有用的建议，而且容错能力比 Mypy 强，还能为代码生成注解。

我们要注解的是 `show_count` 函数。这个函数会返回一个字符串，根据数量多少做单复数变形。

```
>>> show_count(99, 'bird')
'99 birds'
>>> show_count(1, 'bird')
'1 bird'
>>> show_count(0, 'bird')
'no birds'
```

示例 8-1 是 `show_count` 函数不带注解的源码。

示例 8-1 messages.py: 没有注解的 `show_count` 函数

```
def show_count(count, word):
    if count == 1:
        return f'1 {word}'
```



```
count_str = str(count) if count else 'no'
return f'{count_str} {word}s'
```

8.3.1 Mypy 初体验

对 messages.py 模块运行 mypy 命令，开始做类型检查。

```
.../no_hints/ $ pip install mypy
[lots of messages omitted...]
.../no_hints/ $ mypy messages.py
Success: no issues found in 1 source file
```

使用默认设置的 Mypy 没有发现示例 8-1 存在问题。



我使用的是 Mypy 0.910，这是我在 2021 年 7 月审稿时 Mypy 的最新版。按照 Mypy 文档所说，Mypy“严格来说是测试版软件，偶有破坏向后兼容性的改动。”Mypy 回显的报告中至少有一条与我在 2020 年 4 月撰写本章时不一样。你阅读本章时看到的结果或许也有差异。

对于没有注解的函数签名，Mypy 默认忽略，除非另有配置。

示例 8-2 仍然使用 pytest 做单元测试。这部分代码被放在 messages_test.py 模块中。

示例 8-2 没有类型提示的 messages_test.py 模块

```
from pytest import mark

from messages import show_count

@mark.parametrize('qty, expected', [
    (1, '1 part'),
    (2, '2 parts'),
])
def test_show_count(qty, expected):
    got = show_count(qty, 'part')
    assert got == expected
```

```
def test_show_count_zero():
    got = show_count(0, 'part')
    assert got == 'no parts'
```

下面在 Mypy 的指引下添加类型提示。

8.3.2 让 Mypy 严格要求

指定命令行选项 `--disallow-untyped-defs`, Mypy 报告没有为参数和返回值添加类型提示的函数定义。

使用这个选项检查测试文件, 报告 3 个错误和 1 个提示。

```
.../no_hints/ $ mypy --disallow-untyped-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
messages_test.py:10: error: Function is missing a type annotation
messages_test.py:15: error: Function is missing a return type
annotation
messages_test.py:15: note: Use "-> None" if function does not
return a value
Found 3 errors in 2 files (checked 1 source file)
```

一开始使用渐进式类型时, 我喜欢指定另一个选项: `--disallow-incomplete-defs`。这时, Mypy 报告没有问题。

```
.../no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
Success: no issues found in 1 source file
```

现在, 只在 `messages.py` 中添加返回类型。

```
def show_count(count, word) -> str:
```

这就为 Mypy 提供了一定的信息。再次使用前面的命令行检查 `messages_test.py`, Mypy 将同时检查 `messages.py`。

```
.../no_hints/ $ mypy --disallow-incomplete-defs messages_test.py
messages.py:14: error: Function is missing a type annotation
for one or more arguments
Found 1 error in 1 file (checked 1 source file)
```

现在，可以依次为函数添加类型提示，不让 Mypy 再报告关于函数没有注解的错误。下面是带完整注解的签名，能让 Mypy 满意。

```
def show_count(count: int, word: str) -> str:
```



如果不想每次都输入 `--disallow-incomplete-defs` 等命令行选项，可以把需要的选项保存到 Mypy 配置文件（详见文档）中。设置既可以针对全局，也可以针对单个模块。下面是一个简单的 `mypy.ini` 配置，仅供参考。

```
[mypy]
python_version = 3.9
warn_unused_configs = True
disallow_incomplete_defs = True
```

8.3.3 参数的默认值

示例 8-1 中的 `show_count` 函数只对规则的名词有效。如果复数形式不是直接在后面添加 `'s'`，则应该让用户提供复数形式，如下所示。

```
>>> show_count(3, 'mouse', 'mice')
'3 mice'
```

这里，我们采用“测试驱动开发”。首先，添加针对第三个参数的测试。别忘了为测试函数添加返回值类型提示，否则 Mypy 不做检查。

```
def test_irregular() -> None:
    got = show_count(2, 'child', 'children')
    assert got == '2 children'
```

Mypy 检测到以下错误。

```
.../hints_2/ $ mypy messages_test.py
messages_test.py:22: error: Too many arguments for "show_count"
Found 1 error in 1 file (checked 1 source file)
```

然后，编辑 `show_count` 函数，添加可选的参数 `plural`，如示例 8-3 所示。

示例 8-3 `hints_2/messages.py`: 有一个可选参数的 `show_count` 函数

```
def show_count(count: int, singular: str, plural: str = '') -> str:
    if count == 1:
        return f'1 {singular}'
    count_str = str(count) if count else 'no'
    if not plural:
        plural = singular + 's'
    return f'{count_str} {plural}'
```

现在，Mypy 报告“Success”（成功）。



以下代码有一个 Mypy 没有捕获的类型错误。你能发现吗？

```
def hex2rgb(color=str) -> tuple[int, int, int]:
```

Mypy 报告的错误不是特别有用。

```
colors.py:24: error: Function is missing a type
      annotation for one or more arguments
```

`color` 参数的类型提示应该是 `color: str`。我写的 `color=str` 不是注解，而是把 `color` 的默认值设为 `str`。

根据我的经验，这是一种常见的错误，容易被忽视，尤其是在复杂的类型提示中。

编写类型提示时建议遵守以下代码风格。

- 参数名称和 `:` 之间不留空格，`:` 后加一个空格。
- 参数默认值前面的 `=` 两侧加空格。

根据 PEP 8，除参数默认值前面的 `=` 之外，其他 `=` 两侧不加空格。

使用 flake8 和 blue 检查代码风格

不要天真，你记不住这些规则的。请使用 flake8 和 blue 等工具。flake8 会报告代码风格等问题，blue 则会根据代码格式化工具 black 内置的（大多数）规则重写源码。

在统一代码风格方面，blue 比 black 好，因为 blue 遵守 Python 自身的风格，默认使用单引号，将双引号作为备选。

```
>>> "I prefer single quotes"
'I prefer single quotes'
```

从 CPython 源码中的 `repr()` 等处可以看出 Python 对单引号的偏爱。依赖 `repr()` 的 `doctest` 模块默认使用单引号。

blue 的作者之一 Barry Warsaw 也是 PEP 8 的共同起草人，自 1994 年起一直是 Python 核心开发者，从 2019 年至今（2021 年 7 月）还是 Python 指导委员会（Steering Council）的一员。默认使用单引号是有坚强后盾的。

如果必须使用 black，那么请使用 `black -S` 选项，保持引号原封不动。

8.3.4 使用 None 表示默认值

在示例 8-3 中，为参数 `plural` 注解的类型是 `str`，默认值是 `''`，没有类型冲突。

这是最好的情况，不过有时使用 `None` 表示默认值则更好。如果可选的参数是可变类型，那么 `None` 是唯一合理的默认值（详见 6.5.1 节）。

如果想把 `plural` 参数的默认值设为 `None`，则函数签名要改成下面这样。

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] =
None) -> str:
```

下面来分析一下。

- `Optional[str]` 表示 `plural` 的值可以是一个 `str` 或 `None`。
- 必须显式地提供默认值，即 `= None`。

如果不为 `plural` 分配默认值，则 Python 运行时将把它视作必需的参数。记住，类型提示在运行时会被忽略。

注意，需要从 `typing` 模块中导入 `Optional`。导入类型时，建议使用 `from typing import X` 句法，缩短函数签名的长度。



`Optional` 并不是一个好名称，因为注解不能让参数变成可选的。分配默认值的参数才是可选的。`Optional[str]` 的意思很简单，表明参数的类型可以是 `str` 或 `NoneType`。在 Haskell 语言和 Elm 语言中，相似的类型名为 `Maybe`。

至此，我们对渐进式类型有了一定的认识，接下来讲一下“类型”概念的具体含义。

8.4 类型由受支持的操作定义

各种文献对类型概念的定义不一。这里，假定类型是一系列值和一系列可操作这些值的函数。

——“PEP 483—The Theory of Type Hints”

实践中，最好把受支持的操作当作类型的关键特征。⁴

⁴除了 Enum 类型，Python 未提供控制类型可取值的句法。例如，使用类型提示不能把 Quantity 定义为 1 和 1000 之间的整数，也不能把 AirportCode 定义为 3 个字母的组合。NumPy 提供了 uint8、int16 和其他面向机器的数值类型，但是在 Python 标准库中，我们只有取值范围非常小的类型（NoneType 和 bool）和特别大的类型（float、int、str、各种元组等）。

例如，从操作的可行性来看，下述函数中的 `x` 应当是什么类型？

```
def double(x):  
    return x * 2
```

`x` 参数可以是数值（int、complex、Fraction、numpy.uint32 等），可以是序列（str、tuple、list 和 array），可以是 N 维 numpy.array，也可以是实现或继承参数为整数的 `__mul__` 方法的其他类型。

再来看一下下面这个带注解的 double 函数。这里没有返回类型，请暂时忽略这一点，把注意力放在参数类型上。

```
from collections import abc  
  
def double(x: abc.Sequence):  
    return x * 2
```

类型检查工具将拒绝接受这段代码。如果告诉 Mypy，`x` 是 `abc.Sequence` 类型，那么 Mypy 在遇到 `x * 2` 时将报错，因为抽象基类 `Sequence` 没有实现或继承 `__mul__` 方法。在运行时，这段代码既能成功处理 str、tuple、list、array 等具体的序列，也能处理数值，因为类型提示在运行时会被忽略。但是，类型检查工具只关注显式声明的类型，而 `abc.Sequence` 没有 `__mul__` 方法。

这就是本节标题想要表达的内容。前文给出的两版 double 函数，Python 运行时都接受，`x` 参数的值可以是任何对象。`x * 2` 操作有可能成功，如果 `x` 不支持乘法，则会抛出 `TypeError`。然而，在分

析带注解的 `double` 源码时，Mypy 将报告 `x * 2` 是错的，因为声明的类型 `x: abc.Sequence` 不支持此项操作。

在渐进式类型系统中，以下两种对类型的解读相互影响着彼此。

鸭子类型

该类型是 Smalltalk（面向对象语言的先驱）以及 Python、JavaScript 和 Ruby 采用的解读视角。对象有类型，但是变量（包括参数）没有类型。在实践中，为对象声明的类型无关紧要，重要的是对象具体支持什么操作。如果能调用 `birdie.quack()`，那么在当前上下文中 `birdie` 就是鸭子。根据定义，只有在运行时尝试操作对象时，才会施行鸭子类型相关的检查。这比**名义类型**（nominal typing）更灵活，但代价是运行时潜在的错误更多。⁵

⁵鸭子类型是**结构类型**（structural typing）的一种内隐形式。引入 `typing.Protocol` 之后，Python 3.8 及以上版本也支持结构类型。8.5.10 节将对结构类型进行简单介绍，第 13 章将深入讲解。

名义类型

该类型是 C++、Java 和 C# 采用的解读视角，带注解的 Python 支持这种类型。对象和变量都有类型。但是，对象只存在于运行时，类型检查工具只关心使用类型提示注解变量（包括参数）的源码。如果 `Duck` 是 `Bird` 的子类，那么就可以把 `Duck` 实例赋值给注解为 `birdie: Bird` 的参数。可是在函数主体中，类型检查工具认为 `birdie.quack()` 调用是非法的，因为 `birdie` 名义上是 `Bird` 对象，而该类没有提供 `.quack()` 方法。在运行时，实参是不是 `Duck` 实例并不重要，因为名义类型会在静态检查阶段检查。类型检查工具不运行程序的任何部分，只读取源码。名义类型比鸭子类型更严格，优点是能在构建流水线中，甚至是在 IDE 中输入代码的过程中更早地捕获一些 bug。

示例 8-4 是一个没有实用价值的示例，仅用于比较鸭子类型和名义类型，以及静态类型检查和运行时行为。⁶

⁶继承往往会被过度使用，很难通过务实的简单示例说清个中缘由，所以请把这个动物示例看作对子类型的一种简单说明。

示例 8-4 birds.py

```
class Bird:
    pass

class Duck(Bird): ❶
    def quack(self):
        print('Quack!')

def alert(birdie): ❷
    birdie.quack()

def alert_duck(birdie: Duck) -> None: ❸
    birdie.quack()

def alert_bird(birdie: Bird) -> None: ❹
    birdie.quack()
```

- ❶ Duck 是 Bird 的子类。
- ❷ alert 没有类型提示，因此类型检查工具会忽略它。
- ❸ alert_duck 接受一个类型为 Duck 的参数。
- ❹ alert_bird 接受一个类型为 Bird 的参数。

使用 Mypy 对 birds.py 进行类型检查，报告一个问题。

```
.../birds/ $ mypy birds.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)
```

仅仅分析源码，Mypy 就发现 alert_bird 有问题：类型提示声明的 birdie 参数是 Bird 类型，但是函数主体中调用了

`birdie.quack()`，而 `Bird` 类没有该方法。

在 `daffy.py` 中使用 `birds` 模块试试，如示例 8-5 所示。

示例 8-5 `daffy.py`

```
from birds import *

daffy = Duck()
alert(daffy)           ❶
alert_duck(daffy)      ❷
alert_bird(daffy)      ❸
```

❶ 有效调用，因为 `alert` 没有类型提示。

❷ 有效调用，因为 `alert_duck` 接受的参数为 `Duck` 类型，而 `daffy` 是 `Duck` 对象。

❸ 有效调用，因为 `alert_bird` 接受的参数为 `Bird` 类型，而 `daffy` 也是 `Bird` (`Duck` 的超类) 对象。

运行 `Mypy` 检查 `daffy.py`，报告的错误与在 `birds.py` 中定义的 `alert_bird` 函数内调用 `quack` 一样。

```
.../birds/ $ mypy daffy.py
birds.py:16: error: "Bird" has no attribute "quack"
Found 1 error in 1 file (checked 1 source file)
```

但是，`Mypy` 没有报告 `daffy.py` 本身存在问题，3 个函数调用都有效。

运行 `daffy.py`，结果如下所示。

```
.../birds/ $ python3 daffy.py
Quack!
Quack!
Quack!
```

一切正常！鸭子类型的优点体现得淋漓尽致。

在运行时，Python 不关注声明的类型，仅使用鸭子类型。虽然 Mypy 报告 `alert_bird` 有一个错误，但是在运行时使用 `daffy` 调用完全没问题。一开始，这可能会让很多 Python 程序员感到惊讶：静态类型检查工具有时会在将要执行的程序中发现错误。

然而，几个月之后，如果你接到任务，扩展这个示例，那么你或许会感谢 Mypy。以示例 8-6 中同样使用 `birds` 的 `woody.py` 模块为例。

示例 8-6 `woody.py`

```
from birds import *

woody = Bird()
alert(woody)
alert_duck(woody)
alert_bird(woody)
```

使用 Mypy 检查 `woody.py`，发现两个错误。

```
.../birds/ $ mypy woody.py
birds.py:16: error: "Bird" has no attribute "quack"
woody.py:5: error: Argument 1 to "alert_duck" has incompatible
type "Bird";
expected "Duck"
Found 2 errors in 2 files (checked 1 source file)
```

第一个错误在 `birds.py` 中，前面已经见过，即在 `alert_bird` 中调用 `birdie.quack()`。第二个错误在 `woody.py` 中，`woody` 是 `Bird` 实例，调用 `alert_duck(woody)` 是无效的，因为 `alert_duck` 函数的参数类型应为 `Duck`。所有 `Duck` 都是 `Bird`，但不是所有 `Bird` 都是 `Duck`。

在运行时，`woody.py` 中的调用都不成功，如示例 8-7 中的控制台会话所示。相关说明见各个标号。

示例 8-7 运行时错误和 Mypy 可能提供的帮助

```
>>> from birds import *
>>> woody = Bird()
>>> alert(woody) ❶
Traceback (most recent call last):
...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_duck(woody) ❷
Traceback (most recent call last):
...
AttributeError: 'Bird' object has no attribute 'quack'
>>>
>>> alert_bird(woody) ❸
Traceback (most recent call last):
...
AttributeError: 'Bird' object has no attribute 'quack'
```

❶ Mypy 不能检测到这个错误，因为 `alert` 没有类型提示。

❷ Mypy 能报告这个问题：Argument 1 to "alert_duck" has incompatible type "Bird"; expected "Duck"。

❸ 从示例 8-4 开始，Mypy 一直报告 `alert_bird` 函数的主体有问题："Bird" has no attribute "quack"。

这个小实验表明，鸭子类型更容易上手，也更灵活，但是无法阻止不受支持的操作在运行时导致错误。名义类型在运行代码之前检测错误，但有时会拒绝实际能运行的代码，比如示例 8-5 中的 `alert_bird(daffy)` 调用。即使 `alert_bird` 函数有时可以正常执行，它的名称也不恰当：函数主体中的对象需要支持 `.quack()` 方法，但是 `Bird` 没有这个方法。

这个示例没什么实际意义，函数主体只有一行。现实中的函数更长，可能会把 `birdie` 参数传给更多的函数，而且 `birdie` 参数的原始位置可能相距较远，导致难以查明运行时错误的根源。类型检查工具可以防止许多此类错误在运行时发生。



类型提示的价值很难通过这种小示例体现出来。基准代码体量越大，好处体现得就越明显。鉴于此，拥有数百万行 Python 代码的公司（例如 Dropbox、谷歌和 Facebook）才愿意投资团队和工具，让整个公司都接纳类型提示，并在 CI 流水线中检查日益增长的 Python 基准代码。

本节从简单的 `double()` 函数开始，探索了鸭子类型和名义类型中类型与操作之间的关系。我们还没有为 `double()` 函数添加完整的类型提示。8.5 节将介绍可用于注解函数的重要类型。8.5.10 节将说明为 `double()` 函数添加类型提示的一种好方式。不过，在此之前，还有一些基础类型需要学习。

8.5 注解中可用的类型

大部分 Python 类型可以在类型提示中使用，不过有一些限制和建议。另外，`typing` 模块引入的特殊结构，在语义上或许会让你惊讶。

本节涵盖了可用于注解的所有主要类型：

- `typing.Any`;
- 简单的类型和类;
- `typing.Optional` 和 `typing.Union`;
- 泛化容器，包括元组和映射;
- 抽象基类;
- 泛化可迭代对象;
- 参数化泛型和 `TypeVar`;
- `typing.Protocol`s——**静态鸭子类型**的关键;
- `typing.Callable`;
- `typing.NoReturn`——就此打住比较好。

下面我们来依次介绍。先从一个看起来奇怪，好像没什么用，但是非常重要的类型开始。

8.5.1 Any 类型

Any 类型是渐进式类型系统的基础，是人们熟知的**动态类型**。下面是一个没有类型信息的函数：

```
def double(x):  
    return x * 2
```

在类型检查工具看来，假定其具有以下类型信息：

```
def double(x: Any) -> Any:  
    return x * 2
```

也就是说，`x` 参数和返回值可以是任何类型，二者甚至可以不同。Any 类型支持所有可能的操作。

以下述签名为例，对比一下 Any 和 object。

```
def double(x: object) -> object:
```

这个函数也接受每一种类型的参数，因为任何类型都是 `object` 的子类型。

然而，类型检查工具拒绝以下函数。

```
def double(x: object) -> object:  
    return x * 2
```

这是因为 `object` 不支持 `__mul__` 操作。Mypy 报告的错误如下所示。

```
.../birds/ $ mypy double_object.py  
double_object.py:2: error: Unsupported operand types for *
```

```
("object" and "int")
Found 1 error in 1 file (checked 1 source file)
```

越一般的类型，接口越狭窄，即支持的操作越少。object 类实现的操作比 abc.Sequence 少，abc.Sequence 实现的操作比 abc.MutableSequence 少，abc.MutableSequence 实现的操作比 list 少。

但是，Any 是一种魔法类型，位于类型层次结构的顶部和底部。Any 既是最一般的类型（使用 n: Any 注解的参数可接受任何类型的值），也是最特定的类型（支持所有可能的操作）。至少，在类型检查工具看来是这样。

当然，没有任何一种类型可以支持所有可能的操作，因此使用 Any 不利于类型检查工具完成核心任务，即检测潜在的非法操作，防止运行时异常导致程序崩溃。

子类型与相容

传统的面向对象名义类型系统依靠的是**子类型**关系。对 T1 类及其子类 T2 来说，T2 是 T1 的子类型。

以下述代码为例。

```
class T1:
    ...

class T2(T1):
    ...

def f1(p: T1) -> None:
    ...

o2 = T2()

f1(o2)  # 有效
```

`f1(o2)` 调用运用了里氏替换原则 (Liskov Substitution Principle, LSP)。其实, Barbara Liskov⁷ 是从受支持的操作角度定义子类型的: 用 `T2` 类型的对象替换 `T1` 类型的对象, 如果程序的行为仍然正确, 那么 `T2` 就是 `T1` 的子类型。

⁷MIT 教授、编程语言设计者以及图灵奖得主。

接着上一段代码, 像下面这样做则违背了 LSP。

```
def f2(p: T2) -> None:
    ...

o1 = T1()

f2(o1)  # 类型错误
```

从受支持的操作角度来看, 这完全合理: 作为子类, `T2` 继承了 `T1` 支持的所有操作。因此, 在任何预期 `T1` 实例的地方都可以使用 `T2` 实例。然而, 反过来就不一定成立了, `T2` 可能实现了其他方法, 因此在预期 `T2` 实例的地方不一定都能使用 `T1` 实例。**行为子类型**

(behavioral subtyping) 这种说法更能体现关注的要点是受支持的操作。行为子类型也用于指代 LSP。

在渐进式类型系统中还有一种关系: **相容** (consistent-with)。满足子类型关系必定是相容的, 不过对 `Any` 还有特殊的规定。

相容规则如下。

01. 对 `T1` 及其子类型 `T2`, `T2` 与 `T1` 相容 (里氏替换)。
02. 任何类型都与 `Any` 相容: 声明为 `Any` 类型的参数接受任何类型的对象。
03. `Any` 与任何类型都相容: 始终可以把 `Any` 类型的对象传给预期其他类型的参数。

下面使用前面定义的对象 `o1` 和 `o2` 来说明规则 2 和规则 3。这段代码中的所有调用都是有效的。

```
def f3(p: Any) -> None:
    ...

o0 = object()
o1 = T1()
o2 = T2()

f3(o0)    #
f3(o1)    # 都有效: 规则2
f3(o2)    #

def f4():  # 返回值类型隐含为Any
    ...

o4 = f4()  # 推导出类型为Any

f1(o4)    #
f2(o4)    # 都有效: 规则3
f3(o4)    #
```

任何渐进式类型系统都需要像 `Any` 这样的通配类型。



说白了，类型分析中所说的“推导”就是“推测”。Python 和其他语言的现代化类型检查工具不强求类型注解完整无缺，很多表达式的类型是可以推导出来的。例如，对 `x = len(s) * 10` 来说，类型检查工具不需要显式类型注解就知道 `x` 是 `int` 类型，因为内置函数 `len` 的类型提示是已知的。

接下来探讨可在注解中使用的其他类型。

8.5.2 简单的类型和类

像 `int`、`float`、`str` 和 `bytes` 这样的简单的类型可以直接在类型提示中使用。标准库、外部包中的具体类，以及用户定义的具体类

(例如 `FrenchDeck`、`Vector2d` 和 `Duck`)，也可以在类型提示中使用。

抽象基类在类型提示中也能用到。8.5.7 节讲容器类型时会讨论抽象基类。

对类来说，相容的定义与子类型相似：子类与所有超类相容。

然而，“实用胜过纯粹”，凡事总有例外，详见下面的提示栏。



int 与 complex 相容

内置类型 `int`、`float` 和 `complex` 之间没有名义上的子类型关系，它们都是 `object` 的直接子类。但 PEP 484 声称，`int` 与 `float` 相容，`float` 与 `complex` 相容。从实用角度来看，这是合理的：`int` 实现了 `float` 的所有操作，而且 `int` 还额外实现了 `&`、`|`、`<<` 等按位运算操作。因此，`int` 也与 `complex` 相容。对于 `i = 3`，`i.real` 是 3，`i.imag` 是 0。

8.5.3 Optional 类型和 Union 类型

8.3.4 节提到过特殊类型 `Optional`，其中有一个示例使用它解决了默认值为 `None` 的问题。

```
from typing import Optional

def show_count(count: int, singular: str, plural: Optional[str] =
None) -> str:
```

`Optional[str]` 结构其实是 `Union[str, None]` 的简写形式，表示 `plural` 的类型可以是 `str` 或 `None`。



Python 3.10 为 Optional 和 Union 提供的句法更好

从 Python 3.10 开始, `Union[str, bytes]` 可以写成 `str | bytes`。这种写法输入的内容更少, 也不用从 `typing` 中导入 `Optional` 或 `Union`。下面是以新旧两种句法编写的 `show_count` 函数的 `plural` 参数的类型提示, 可以比较一下。

```
plural: Optional[str] = None    # 旧句法
plural: str | None = None       # 新句法
```

`|` 运算符还可用于构建 `isinstance` 和 `issubclass` 的第二个参数, 例如 `isinstance(x, int | str)`。详见“PEP 604—Complementary syntax for `Union[]`”。

内置函数 `ord` 的签名就用到了 `Union`, 其接受 `str` 或 `bytes` 类型, 并返回一个 `int`。⁸

⁸严格来说, `ord` 接受的 `str` 或 `bytes` 类型必须满足 `len(s) == 1`。但是, 类型系统目前还不能表达这种约束。

```
def ord(c: Union[str, bytes]) -> int: ...
```

下面示例中的函数接受一个 `str`, 但是可以返回一个 `str` 或 `float`。

```
from typing import Union

def parse_token(token: str) -> Union[str, float]:
    try:
        return float(token)
    except ValueError:
        return token
```

尽量避免创建返回 `Union` 类型值的函数, 因为这会给用户带来额外的负担, 迫使他们必须在运行时检查返回值的类型, 判断该如何处理。但是, 在简单的表达式计算器中可以像上一段代码中的 `parse_token` 那样做。



4.10 节介绍过接受 `str` 或 `bytes` 参数的函数，当参数为 `str` 时，返回 `str`，当参数为 `bytes` 时，返回 `bytes`。在那种情况下，返回值类型由输入值类型决定，因此不适合使用 `Union`。为了正确注解这样的函数，需要使用类型变量（参见 8.5.9 节）或重载（参见 15.2 节）。

`Union[]` 至少需要两种类型。嵌套的 `Union` 类型与扁平的 `Union` 类型效果相同。因此，下面的类型提示：

```
Union[A, B, Union[C, D, E]]
```

与下面的类型提示作用一样：

```
Union[A, B, C, D, E]
```

`Union` 所含的类型之间不应相容。例如，`Union[int, float]` 就“画蛇添足”了，因为 `int` 与 `float` 相容。仅使用 `float` 注解的参数也接受 `int` 值。

8.5.4 泛化容器

大多数 Python 容器是异构的。例如，在一个 `list` 中可以混合存放不同的类型。然而，实际使用中这么做没有什么意义。存入容器的对象往往需要进一步处理，因此至少要有一个通用的方法。⁹

⁹对 Python 的原初设计有重大影响的 ABC 语言，限制列表中只能有一种类型的值，与第一项保持一致。

泛型可以用类型参数来声明，以指定可以处理的项的类型。

例如，可以像示例 8-8 那样参数化一个 `list`，约束元素的类型。

示例 8-8 带类型提示的 `tokenize` 函数（Python3.9 及以上版本）

```
def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

在 Python 3.9 及以上版本中，类型提示的意思是 `tokenize` 函数返回一个 `list`，而且各项均为 `str` 类型。

`stuff: list` 和 `stuff: list[Any]` 这两个注解的意思相同，都表示 `stuff` 是一个列表，而且列表中的项可以是任何类型的对象。



对于 Python 3.8 或之前的版本，道理是一样的，只是需要编写的代码更多，详见“早期支持和弃用的容器类型”附注栏。

“PEP 585—Type Hinting Generics In Standard Collections”列出了标准库中接受泛化类型提示的容器。下面只列出了可使用最简单的泛化类型提示形式的容器，例如 `container[item]`。

<code>list</code>	<code>collections.deque</code>	<code>abc.Sequence</code>	
<code>abc.MutableSequence</code>			
<code>set</code>	<code>abc.Container</code>	<code>abc.Set</code>	<code>abc.MutableSet</code>
<code>frozenset</code>	<code>abc.Collection</code>		

`tuple` 和映射类型支持更复杂的类型提示，后文有两节会分别说明。

截至 Python 3.10，没有什么好方法来注解带 `typecode` 构造函数参数（决定数组中存放整数还是浮点数）的 `array.array`。更棘手的问题是，如何检查整数区间，防止在运行时向数组中添加元素而导致 `OverflowError`。例如，使用 `typecode='B'` 创建的数组，只能存放 0 和 255 之间的 `int` 值。目前，Python 的静态类型系统还未解决这个问题。

早期支持和弃用的容器类型

（如果只使用 Python 3.9 或以上版本，则可以跳过这个附注栏。）

对于 Python 3.7 和 Python 3.8, 需要从 `__future__` 中导入相关内容, 才能在内置容器 (例如 `list`) 后面使用 `[]` 表示法, 如示例 8-9 所示。

示例 8-9 带类型提示的 `tokenize` 函数 (Python 3.7 及以上版本)

```
from __future__ import annotations

def tokenize(text: str) -> list[str]:
    return text.upper().split()
```

这种方法不适用于 Python 3.6 或之前的版本。在 Python 3.5 及以上版本中, 要像示例 8-10 那样注解 `tokenize` 函数。

示例 8-10 带类型提示的 `tokenize` 函数 (Python 3.5 及以上版本)

```
from typing import List

def tokenize(text: str) -> List[str]:
    return text.upper().split()
```

最初, 为了支持泛化类型提示, PEP 484 的作者在 `typing` 模块中创建了几十种泛型。表 8-1 列出了其中一部分。完整的列表参阅 `typing` 模块文档。

表 8-1: 部分容器类型及对应的类型提示

容器	对应的类型提示
<code>list</code>	<code>typing.List</code>
<code>set</code>	<code>typing.Set</code>

容器	对应的类型提示
<code>frozenset</code>	<code>typing.FrozenSet</code>
<code>collections.deque</code>	<code>typing.Deque</code>
<code>collections.abc.MutableSequence</code>	<code>typing.MutableSequence</code>
<code>collections.abc.Sequence</code>	<code>typing.Sequence</code>
<code>collections.abc.Set</code>	<code>typing.AbstractSet</code>
<code>collections.abc.MutableSet</code>	<code>typing.MutableSet</code>

“PEP 585—Type Hinting Generics In Standard Collections”发起过一项历时多年的行动，以改进泛化类型提示的可用性。整个过程分为 4 步。

01. 在 Python 3.7 中引入 `from __future__ import annotations`，以 `list[str]` 表示法支持在泛型中使用标准库中的类。
02. Python 3.9 把这种表示法定为标准行为，无须从 `future` 导入即可使用 `list[str]`。
03. 弃用 `typing` 模块中所有冗余的泛型。¹⁰ Python 解释器不对弃用的类型发出警告，因为类型检查工具在检查使用 Python 3.9 或以上版本编写的程序时会发出警告。
04. Python 3.9 发布 5 年后发布的第一个 Python 版本会移除那些冗余的泛型。按照目前的节奏来看，应该是 Python 3.14，即 Python Pi。

¹⁰看到“Module Contents”下方的说明后，我为 `typing` 模块文档做了一点儿贡献，为几个小节添加了弃用警告。当然，这一切都在 Guido van Rossum 的监督之下。

接下来看看如何注解泛化元组。

8.5.5 元组类型

元组类型的注解分 3 种形式说明：

- 用作记录的元组；
- 带有具名字段，用作记录的元组；
- 用作不可变序列的元组。

01. 用作记录的元组

元组用作记录时，使用内置类型 `tuple` 注解，字段的类型在 `[]` 内声明。

举个例子：一个内容为城市名、人口数和所属国家的元组，例如 `('Shanghai', 24.28, 'China')`，类型提示为 `tuple[str, float, str]`。

假如有一个函数，其接受的参数是一对地理坐标，返回值是一个 Geohash。该函数的用法如下所示。

```
>>> shanghai = 31.2304, 121.4737
>>> geohash(shanghai)
'wtw3sjq6q'
```

`geohash` 函数使用 PyPI 中的 `geolib` 包定义，如示例 8-11 所示。

示例 8-11 `coordinates.py`: `geohash` 函数


```
from geolib import geohash as gh # type: ignore ❶

PRECISION = 9

def geohash(lat_lon: tuple[float, float]) -> str: ❷
    return gh.encode(*lat_lon, PRECISION)
```

❶ 这个注释会禁止 Mypy 报告 `geolib` 包没有类型提示。

❷ 注解 `lat_lon` 参数，值为一个 `tuple`，包含两个 `float` 字段。



对于 Python 3.9 以下的版本，要在类型提示中使用 `typing.Tuple`。这个类型已经弃用，不过 2024 年之前仍会被保留在标准库中。

02. 带有具名字段，用作记录的元组

如果想注解带有多个字段的元组，或者代码中多次用到的特定类型的元组，强烈建议使用 `typing.NamedTuple`（参见第 5 章）。示例 8-12 使用 `NamedTuple` 注解了示例 8-11 中的 `geohash` 函数。

示例 8-12 `coordinates_named.py`: 具名元组
`Coordinates` 和 `geohash` 函数

```
from typing import NamedTuple

from geolib import geohash as gh # type: ignore

PRECISION = 9

class Coordinate(NamedTuple):
    lat: float
    lon: float
```

```
def geohash(lat_lon: Coordinate) -> str:
    return gh.encode(*lat_lon, PRECISION)
```

5.2 节讲过, `typing.NamedTuple` 是 `tuple` 子类的制造工厂, 因此 `Coordinate` 与 `tuple[float, float]` 相容, 但是反过来不成立, 毕竟 `NamedTuple` 为 `Coordinate` 额外添加了方法 (例如 `._asdict()`), 另外用户也可以定义方法。

实践中, 可以放心地把 `Coordinate` 实例传给下面定义的 `display` 函数。

```
def display(lat_lon: tuple[float, float]) -> str:
    lat, lon = lat_lon
    ns = 'N' if lat >= 0 else 'S'
    ew = 'E' if lon >= 0 else 'W'
    return f'{abs(lat):0.1f}°{ns}, {abs(lon):0.1f}°{ew}'
```

03. 用作不可变序列的元组

如果想注解长度不定、用作不可变列表的元组, 则只能指定一个类型, 后跟逗号和 `...` (Python 中的省略号, 3 个点, 不是 Unicode 字符 U+2026, 即 HORIZONTAL ELLIPSIS)。

例如, `tuple[int, ...]` 表示项为 `int` 类型的元组。

省略号表示元素的数量 ≥ 1 。可变长度的元组不能为字段指定不同的类型。

`stuff: tuple[Any, ...]` 和 `stuff: tuple` 这两个注解的意思相同, 都表示 `stuff` 是一个元组, 长度不定, 可包含任意类型的对象。

下面的代码使用 `columnize` 函数把一个序列转换成了元组列表 (类似于表格中的行和单元格), 列表中的元组长度不定。最

后，按列显示各项。

```
>>> animals = 'drake fawn heron ibex koala lynx tahr xerus yak
zapus'.split()
>>> table = columnize(animals)
>>> table
[('drake', 'koala', 'yak'), ('fawn', 'lynx', 'zapus'),
 ('heron', 'tahr'),
 ('ibex', 'xerus')]
>>> for row in table:
...     print(''.join(f'{word:10}' for word in row))
...
drake      koala      yak
fawn       lynx       zapus
heron      tahr
ibex       xerus
```

`columnize` 函数的实现如示例 8-13 所示。注意返回值类型。

```
list[tuple[str, ...]]
```

示例 8-13 `columnize.py`: 返回一个列表，列表中的项是字符串元组

```
from collections.abc import Sequence

def columnize(
    sequence: Sequence[str], num_columns: int = 0
) -> list[tuple[str, ...]]:
    if num_columns == 0:
        num_columns = round(len(sequence) ** 0.5)
    num_rows, remainder = divmod(len(sequence), num_columns)
    num_rows += bool(remainder)
    return [tuple(sequence[i:num_rows]) for i in
            range(num_rows)]
```

8.5.6 泛化映射

泛化映射类型使用 `MappingType[KeyType, ValueType]` 形式注解。在 Python 3.9 及以上版本中，内置类型 `dict` 及 `collections` 和 `collections.abc` 中的映射类型都可以这样注解。更早的版本

必须使用 `typing.Dict` 和 `typing` 模块中的其他映射类型，详见 8.5.4 节中的“早期支持和弃用的容器类型”附注栏。

示例 8-14 定义了一个实用的函数，返回反向索引，按名称搜索 Unicode 字符。这是对示例 4-21 的改造，更适合在服务器端使用（详见第 21 章）。

`name_index` 函数的参数是起点和终点两个 Unicode 字符编码，会返回一个名为 `dict[str, set[str]]` 的反向索引，把各个单词映射到名称中含有该词的字符集合上。例如，对于 32 和 64 之间的 ASCII 字符，索引之后，`'SIGN'` 和 `'DIGIT'` 两个词映射的字符集合如下所示。这里还展示了如何搜索名为 `'DIGIT EIGHT'` 的字符。

```
>>> index = name_index(32, 65)
>>> index['SIGN']
{'$', '>', '=', '+', '<', '%', '#'}
>>> index['DIGIT']
{'8', '5', '6', '2', '3', '0', '1', '4', '7', '9'}
>>> index['DIGIT'] & index['EIGHT']
{'8'}
```

示例 8-14 是 `name_index` 函数所在的 `charindex.py` 模块的源码。除了类型提示 `dict[]`，这个示例还有 3 个功能在本书中是首次出现。

示例 8-14 charindex.py

```
import sys
import re
import unicodedata
from collections.abc import Iterator

RE_WORD = re.compile(r'\w+')
STOP_CODE = sys.maxunicode + 1

def tokenize(text: str) -> Iterator[str]: ❶
    """返回全大写的单词构成的可迭代对象"""
    for match in RE_WORD.finditer(text):
        yield match.group().upper()

def name_index(start: int = 32, end: int = STOP_CODE) -> dict[str,
set[str]]:
```

```

index: dict[str, set[str]] = {} ❷
for char in (chr(i) for i in range(start, end)):
    if name := unicodedata.name(char, ''): ❸
        for word in tokenize(name):
            index.setdefault(word, set()).add(char)
return index

```

❶ `tokenize` 是一个生成器函数（详见第 17 章）。

❷ 注解局部变量 `index`。如果不加类型提示，则 `Mypy` 将发出提示：
`Need type annotation for 'index' (hint: "index: dict[<type>, <type>] = ...")`。

❸ `if` 条件中使用了海象运算符 `:=`。这样做是为了把 `unicodedata.name()` 调用的结果赋值给 `name`，并把该结果作为整个表达式的求解结果。如果结果是表示假值的 `''`，则不更新 `index`。¹¹

¹¹有几个示例适合使用 `:=`，但是本书并未介绍这个运算符。详细说明见“PEP 572—Assignment Expressions”。



当把 `dict` 用作记录时，一般来说，所有键都使用 `str` 类型，对应的值是什么类型则取决于键的含义。详见 15.3 节。

8.5.7 抽象基类

发送时要保守，接收时要大方。

——伯斯塔尔定律，又称稳健性法则

表 8-1 列出了 `collections.abc` 中的几个抽象类。理想情况下，函数的参数应接受那些抽象类型（或 Python 3.9 之前的版本中 `typing` 模块中对应的类型），而不是具体类型。这样对调用方来说更加灵活。

以下述函数签名为例。

```
from collections.abc import Mapping

def name2hex(name: str, color_map: Mapping[str, int]) -> str:
```

由于注解的类型是 `abc.Mapping`，因此调用方可以提供 `dict`、`defaultdict` 和 `ChainMap` 的实例，`UserDict` 子类的实例，或者 `Mapping` 的任何子类型。

相比之下，再看下面的签名。

```
def name2hex(name: str, color_map: dict[str, int]) -> str:
```

这里，`color_map` 必须是 `dict` 或其子类型，例如 `defaultDict` 或 `OrderedDict`。特别注意，使用 `collections.UserDict` 的子类无法通过类型检查，尽管 3.6.5 节讲过，建议扩展 `collections.UserDict` 自定义映射。Mypy 会拒绝 `UserDict` 或其衍生类的实例，因为 `UserDict` 不是 `dict` 的子类，二者是同级关系，都是 `abc.MutableMapping` 的子类。¹²

¹²其实，`dict` 是 `abc.MutableMapping` 的虚拟子类（第 13 章将介绍虚拟子类的概念）。现在，只需知道，`issubclass(dict, abc.MutableMapping)` 的结果为 `True`，尽管 `dict` 是用 C 语言实现的，而且未从 `abc.MutableMapping` 继承任何东西。`dict` 只继承 `object`。

因此，一般来说在参数的类型提示中最好使用 `abc.Mapping` 或 `abc.MutableMapping`，不要使用 `dict`（也不要再在遗留代码中使用 `typing.Dict`）。如果 `name2hex` 函数无须改动传入的 `color_map`，则最准确的类型提示是 `abc.Mapping`。如此一来，调用方就不用提供实现了 `setdefault`、`pop` 和 `update` 等方法的对象了，因为这些方法属于 `MutableMapping` 接口，而不是 `Mapping` 接口。这样做体现了伯斯塔尔定律的后半部分：接收时要大方。

伯斯塔尔定律还指出，发送时要保守。因此，函数的返回值始终应该是一个具体对象，即返回值的类型提示应当是具体类型。8.5.4 节中的示例就是如此，返回值类型为 `list[str]`。

```
def tokenize(text: str) -> list[str]:  
    return text.upper().split()
```

在 `typing.List` 的文档中有这样一段话。

泛化版 `list`。可用于注解返回值类型。如果想注解参数，推荐使用抽象容器类型，例如 `Sequence` 或 `Iterable`。

`typing.Dict` 和 `typing.Set` 的文档也有类似的说明。

记住，从 Python 3.9 开始，`collections.abc` 中的大多数抽象基类和 `collections` 中的具体类，以及内置的容器，全都支持泛化类型提示，例如 `collections.deque[str]`。使用 Python 3.8 或之前的版本编写代码时才需要使用 `typing` 模块中对应的容器类型。已经泛化的类很多，完整列表见“PEP 585—Type Hinting Generics In Standard Collections”中的“Implementation”一节。

在结束讨论类型提示中的抽象基类之前，还要讲一下 `numbers` 包中的抽象基类。

论数字塔的倒下

`numbers` 包定义了“PEP 3141—A Type Hierarchy for Numbers”提出的所谓的数字塔（numeric tower）。这座塔是一种抽象基类构成的线性层次结构，`Number` 位于最顶层。

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

这些抽象基类完全适应运行时类型检查，不过静态类型检查不支持它们。PEP 484 中的“Numeric Tower”一节拒绝使用 `numbers` 包中的抽象基类，规定应把内置类型 `complex`、`float` 和 `int` 当作特例（详见 8.5.2 节的提示栏“int 与 complex 相容”）。

13.6.8 节在比较协议和抽象基类时将再次探讨这个问题。

实践中，针对静态类型检查，注解数字参数有以下几种选择。

01. 按照 PEP 488 的建议，使用 `int`、`float` 或 `complex` 中的某个具体类型。
02. 声明一种联合类型，例如 `Union[float, Decimal, Fraction]`。
03. 如果不想硬编码具体类型，可以使用 `SupportsFloat` 等数字协议（详见 13.6.2 节）。

8.5.10 节将介绍一些理解数字协议的预备知识。

接下来介绍类型提示中经常用到的一种抽象基类，即 `Iterable`。

8.5.8 Iterable

前文引用的 `typing.List` 文档推荐使用 `Sequence` 和 `Iterable` 注解函数的参数。

标准库中的 `math.fsum` 函数，其参数的类型提示用的就是 `Iterable`。

```
def fsum(__seq: Iterable[float]) -> float:
```



存根文件和 Typeshed 项目

截至 Python 3.10，标准库不含注解，但是 `Mypy`、`PyCharm` 等可在 `Typeshed` 项目中找到所需的类型提示。这些类型提示位于

一种**存根文件** (stub file) 中, 这是一种特殊的源文件, 扩展名为 .pyi, 文件中保存带注解的函数和方法签名, 没有实现, 有点儿类似于 C 语言的头文件。

math.fsum 函数的签名位于 /stdlib/2and3/math.pyi 文件中。
__seq 开头的下划线是 PEP 484 为仅限位置参数所做的约定, 详见 8.6 节。

示例 8-15 是另一个使用 Iterable 参数的例子, 产生的项是 tuple[str, str] 类型。函数的用法如下所示。

```
>>> l33t = [('a', '4'), ('e', '3'), ('i', '1'), ('o', '0')]
>>> text = 'mad skilled noob powned leet'
>>> from replacer import zip_replace
>>> zip_replace(text, l33t)
'm4d sk1ll13d n00b p0wn3d l33t'
```

示例 8-15 是具体实现。

示例 8-15 replacer.py

```
from collections.abc import Iterable

FromTo = tuple[str, str] ❶
def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ❷
    for from_, to in changes:
        text = text.replace(from_, to)
    return text
```

❶ FromTo 是**类型别名** (type alias)。这里把 tuple[str, str] 赋值给了 FromTo, 这样 zip_replace 函数签名的可读性会好一些。

❷ changes 的类型为 Iterable[FromTo]。这与 Iterable[tuple[str, str]] 的效果一样, 不过签名更短, 可读性更高。



在 Python 3.10 中显式使用 `TypeAlias`

“PEP 613—Explicit Type Aliases”引入了一种特殊类型，即 `TypeAlias`，以让创建类型别名的赋值操作更显眼，也让类型检查更容易。从 Python 3.10 开始，创建类型别名的首选方式如下所示。

```
from typing import TypeAlias

FromTo: TypeAlias = tuple[str, str]
```

`abc.Iterable` 与 `abc.Sequence`

`math.fsum` 和 `replacer.zip_replace` 必须迭代整个 `Iterable` 参数才能返回结果。如果输入值是像 `itertools.cycle` 生成器这样的无穷可迭代对象，则这两个函数将耗尽内存，导致 Python 进程崩溃。尽管存在一定危险，但是对于现代的 Python，经常需要提供接受 `Iterable` 输入值的函数，而且必须完整处理才能返回结果。调用方可以根据需要，选择通过生成器提供输入数据，而不是预先构建好序列，这样在项数较多时可以节省不少内存。

另外，示例 8-13 中的 `columnize` 函数需要一个 `Sequence` 参数，而不是 `Iterable`，因为该函数必须事先获得输入的长度 (`len()`)，算出行数。

与 `Sequence` 一样，`Iterable` 最适合注解参数的类型。用来注解返回值类型的话则太过含糊。函数的返回值类型应该具体、明确。

示例 8-14 中注解返回值类型的 `Iterator` 与 `Iterable` 紧密相关。第 17 章在讲解生成器和经典迭代器时将再探讨 `Iterator` 类型。

8.5.9 参数化泛型和 `TypeVar`

参数化泛型是一种泛型，写作 `list[T]`，其中 `T` 是类型变量，每次使用时会绑定具体的类型。这样可在结果的类型中使用参数的类型。

示例 8-16 定义的 `sample` 函数接受两个参数，一个是元素类型为 `T` 的 `Sequence`，另一个是 `int`。该函数会返回一个 `list`，元素的类型也是 `T`，具体类型由第一个参数决定。

示例 8-16 是具体实现。

示例 8-16 sample.py

```
from collections.abc import Sequence
from random import shuffle
from typing import TypeVar

T = TypeVar('T')

def sample(population: Sequence[T], size: int) -> list[T]:
    if size < 1:
        raise ValueError('size must be >= 1')
    result = list(population)
    shuffle(result)
    return result[:size]
```

在 `sample` 函数中使用类型变量达到的效果通过下面两种情况可以体现。

- 调用时如果传入 `tuple[int, ...]` 类型（与 `Sequence[int]` 相容）的元组，类型参数为 `int`，那么返回值类型为 `list[int]`。
- 调用时如果传入一个 `str`（与 `Sequence[str]` 相容），类型参数为 `str`，那么返回值类型为 `list[str]`。



为什么需要 `TypeVar`?

PEP 484 的作者希望借助 `typing` 模块引入类型提示，不改动语言的其他部分。通过精巧的元编程技术，让类支持 `[]` 运算符

(例如 `Sequence[T]`) 不成问题。但是, 方括号内的 `T` 变量必须在某处定义, 否则要大范围改动 Python 解释器才能让泛型支持特殊的 `[]` 表示法。鉴于此, 我们增加了 `typing.TypeVar` 构造函数, 把变量名称引入当前命名空间。由于 Java、C# 和 TypeScript 等语言不要求事先声明类型变量的名称, 因此没有与 Python 的 `TypeVar` 类对应的结构。

标准库中的 `statistics.mode` 函数也是一例。该函数会返回一系列值中出现次数最多的数据点。

下面是 `statistics.mode` 函数文档给出的一个使用示例。

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

不使用 `TypeVar`, `mode` 函数的签名可能会像示例 8-17 那样。

示例 8-17 `mode_float.py`: 可操作 `float` 及其子类型的 `mode` 函数¹³

¹³这里的实现比 Python 标准库中 `statistics` 模块的实现简单。

```
from collections import Counter
from collections.abc import Iterable

def mode(data: Iterable[float]) -> float:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

`mode` 经常用于处理 `int` 或 `float` 值, 但是 Python 还有其他数值类型, 因此返回值类型最好与 `Iterable` 中的元素类型保持一致。使用 `TypeVar` 可以改进签名。先来看一个看似简单, 但是不正确的参数化签名。

```
from collections.abc import Iterable
from typing import TypeVar

T = TypeVar('T')

def mode(data: Iterable[T]) -> T:
```

第一次出现在签名中时，类型参数 `T` 可以是任何类型。第二次出现时，与第一次的类型相同。

因此，任何可迭代对象都与 `Iterable[T]` 相容，包括 `collections.Counter` 无法处理的不可哈希的可迭代类型。需要限制可以赋予 `T` 的类型。下面两节介绍了两种方式。

01. 受限的 `TypeVar`

`TypeVar` 还接受一些位置参数，以对类型参数施加限制。可以改进 `mode` 函数的签名，以接受指定的几种数值类型，如下所示。

```
from collections.abc import Iterable
from decimal import Decimal
from fractions import Fraction
from typing import TypeVar

NumberT = TypeVar('NumberT', float, Decimal, Fraction)

def mode(data: Iterable[NumberT]) -> NumberT:
```

这样好多了。这也是 2020 年 5 月 25 日 `typeshed` 项目中存根文件 `statistics.pyi` 为 `mode` 函数提供的签名。

然而，`statistics.mode` 文档中还有以下示例。

```
>>> mode(["red", "blue", "blue", "red", "green", "red",
"red"])
'red'
```

你的第一反应可能是在 `NumberT` 定义中添加 `str`。

```
NumberT = TypeVar('NumberT', float, Decimal, Fraction, str)
```

这样做当然可以，但是 `NumberT` 名称就文不对题了。而且，不能发现一种 `mode` 可以处理的类型就添加一个。更好的方法是使用接下来介绍的 `TypeVar` 的另一项功能。

02. 有界的 `TypeVar`

如前所述，在示例 8-17 中，`mode` 函数主体内的 `Counter` 类用于排名。`Counter` 基于 `dict`，因此可迭代对象 `data` 中元素的类型必须是可哈希的。

乍一看，下面的签名行之有效。

```
from collections.abc import Iterable, Hashable

def mode(data: Iterable[Hashable]) -> Hashable:
```

现在的问题是，返回的项是 `Hashable` 类型。`Hashable` 是一个抽象基类，只实现了 `__hash__` 方法。因此，除了调用 `hash()`，类型检查工具不会允许对返回值做其他任何操作。所以，这么做没什么实际意义。

解决方法是使用 `TypeVar` 的另一个可选参数，即关键字参数 `bound`。这个参数会为可接受的类型设定一个上边界。示例 8-18 使用 `bound=Hashable` 指明，类型参数可以是 `Hashable` 或它的任何子类型。¹⁴

示例 8-18 `mode_hashable.py`: 与示例 8-17 类似，不过签名更灵活

```
from collections import Counter
from collections.abc import Iterable, Hashable
from typing import TypeVar
```

```
HashableT = TypeVar('HashableT', bound=Hashable)

def mode(data: Iterable[HashableT]) -> HashableT:
    pairs = Counter(data).most_common(1)
    if len(pairs) == 0:
        raise ValueError('no mode for empty data')
    return pairs[0][0]
```

总结一下：

- 受限的类型变量会把类型设为 `TypeVar` 声明中列出的某个类型；
- 有界的类型变量会把类型设为根据表达式推导出的类型，但前提是推导的类型与 `TypeVar` 的 `bound=` 关键字参数声明的边界相容。



为 `TypeVar` 声明边界的关键字参数名为 `bound=`，因为“绑定”一般指为变量设定值，在 Python 的引用语义下表示为值绑定名称。把那个关键字参数命名为 `boundary=` 可能要好一些，因为不容易让人误解。

`typing.TypeVar` 构造函数还有两个可选参数，即 `covariant` 和 `contravariant`，15.7 节将介绍。

下面让我们通过 `AnyStr` 来结束对 `TypeVar` 的介绍。

03. 预定义的类型变量 `AnyStr`

`typing` 模块提供了一个预定义的类型变量，名为 `AnyStr`。这个类型变量的定义如下所示。

```
AnyStr = TypeVar('AnyStr', bytes, str)
```

很多接受 `bytes` 或 `str` 的函数会使用 `AnyStr`，返回值也是二者之一。

接下来换个话题，讲一下 `typing.Protocol`。这是 Python 3.8 新增的功能，旨在以更符合 Python 风格的方式编写类型提示。

¹⁴我为 `typedsh` 项目贡献了这种方法。从 2020 年 5 月 26 日起，`statistics.pyi` 就是这样注解 `mode` 函数的。

8.5.10 静态协议



在面向对象编程中，把“协议”当作非正式的接口历史久远，可以追溯到 `Smalltalk`。自 Python 诞生伊始，这个概念就存在。然而，对类型提示来说，协议指的是 `typing.Protocol` 的子类，定义接口供类型检查工具核查。这两种协议将在第 13 章详述。本节只是借着函数注解做个简单介绍。

“PEP 544—Protocols: Structural subtyping (static duck typing)”提出的 `Protocol` 类型类似于 Go 语言中的接口：定义协议类型时指定一个或多个方法，在需要使用协议类型的地方，类型检查工具会核查有没有实现指定的方法。

在 Python 中，协议通过 `typing.Protocol` 的子类定义。然而，**实现**协议的类不会与**定义**协议的类建立任何关系，不继承，也不用注册。类型检查工具负责查找可用的协议类型，施行用法检查。

以下问题可以通过 `Protocol` 和 `TypeVar` 解决。假如你想创建 `top(it, n)` 函数，返回可迭代对象 `it` 中排位靠前的 `n` 个元素。

```
>>> top([4, 1, 5, 2, 6, 7, 3], 3)
[7, 6, 5]
>>> l = 'mango pear apple kiwi banana'.split()
>>> top(l, 3)
['pear', 'mango', 'kiwi']
>>>
>>> l2 = [(len(s), s) for s in l]
```



```
>>> l2
[(5, 'mango'), (4, 'pear'), (5, 'apple'), (4, 'kiwi'), (6,
'banana')]
>>> top(l2, 3)
[(6, 'banana'), (5, 'mango'), (5, 'apple')]
```

示例 8-19 使用参数化泛型定义 top 函数。

示例 8-19 top 函数，有一个未定义的类型参数 T

```
def top(series: Iterable[T], length: int) -> list[T]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]
```

那么如何约束 T 呢？T 不能是 Any 或 object，因为 series 必须支持使用 sorted 函数排序。其实，内置函数 sorted 接受任何 Iterable[Any]，但前提是可选参数 key 传入的函数能根据各个元素计算任意的排序键。如果把一个普通对象列表传给 sorted 函数，而不提供 key 参数，那么情况如何呢？下面来试试。

```
>>> l = [object() for _ in range(4)]
>>> l
[<object object at 0x10fc2fca0>, <object object at 0x10fc2fbb0>,
<object object at 0x10fc2fbc0>, <object object at 0x10fc2fbd0>]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'object' and
'object'
```

错误消息表明，sorted 函数会在可迭代对象的元素上使用 < 运算符。支持 < 运算符就可以了吗？再试一下。¹⁵

¹⁵ 打开一个交互式控制台就能通过鸭子类型探索语言功能，这是一件多么美好的事情。在不支持鸭子类型的语言中，我十分怀念这种探索方式。

```
>>> class Spam:
...     def __init__(self, n): self.n = n
...     def __lt__(self, other): return self.n < other.n
...     def __repr__(self): return f'Spam({self.n})'
... 
```

```
>>> l = [Spam(n) for n in range(5, 0, -1)]
>>> l
[Spam(5), Spam(4), Spam(3), Spam(2), Spam(1)]
>>> sorted(l)
[Spam(1), Spam(2), Spam(3), Spam(4), Spam(5)]
```

确实如此，`Spam` 对象构成的列表可以排序，因为 `Spam` 实现了支持 `<` 运算符的特殊方法 `__lt__`。

所以，示例 8-19 中的类型参数 `T` 应该被限定为实现了 `__lt__` 的类型。示例 8-18 需要实现了 `__hash__` 的类型参数，因此可以把类型参数的上边界设为 `typing.Hashable`。但是，对目前遇到的问题，`typing` 或 `abc` 中没有合适的类型，需要自己创建。

示例 8-20 通过 `Protocol` 定义了一个新类型 `SupportsLessThan`。

示例 8-20 comparable.py: 定义协议类型 `SupportsLessThan`

```
from typing import Protocol, Any

class SupportsLessThan(Protocol): ❶
    def __lt__(self, other: Any) -> bool: ... ❷
```

❶ 协议是 `typing.Protocol` 的子类。

❷ 在协议主体中定义一个或多个方法，方法的主体为 `...`。

如果类型 `T` 实现了协议 `P` 定义的所有方法且类型签名匹配，那么 `T` 就与 `P` 相容。

现在，可以使用 `SupportsLessThan` 定义 `top` 函数，如示例 8-21 所示。这一版正常可用了。

示例 8-21 top.py: 使用设定了 `bound=SupportsLessThan` 的 `TypeVar` 定义 `top` 函数

```

from collections.abc import Iterable
from typing import TypeVar

from comparable import SupportsLessThan

LT = TypeVar('LT', bound=SupportsLessThan)

def top(series: Iterable[LT], length: int) -> list[LT]:
    ordered = sorted(series, reverse=True)
    return ordered[:length]

```

下面使用 `pytest` 测试一下 `top` 函数。示例 8-22 是测试套件的一部分，先使用 `top` 处理通过生成器表达式产出的 `tuple[int, str]` 值，再处理一个 `object` 列表。对于 `object` 列表，我们预期抛出 `TypeError` 异常。

示例 8-22 `top_test.py`: `top` 函数测试套件的一部分

```

from collections.abc import Iterator
from typing import TYPE_CHECKING ❶

import pytest
from top import top

# 省略几行

def test_top_tuples() -> None:
    fruit = 'mango pear apple kiwi banana'.split()
    series: Iterator[tuple[int, str]] = ( ❷
        (len(s), s) for s in fruit)
    length = 3
    expected = [(6, 'banana'), (5, 'mango'), (5, 'apple')]
    result = top(series, length)
    if TYPE_CHECKING: ❸
        reveal_type(series) ❹
        reveal_type(expected)
        reveal_type(result)
    assert result == expected

# 有意测试类型错误
def test_top_objects_error() -> None:
    series = [object() for _ in range(4)]
    if TYPE_CHECKING:
        reveal_type(series)
    with pytest.raises(TypeError) as excinfo:

```

```
top(series, 3) ❹
assert "'<' not supported" in str(excinfo.value)
```

❶ `typing.TYPE_CHECKING` 常量在运行时始终为 `False`，不过类型检查工具在做类型检查时会假装值为 `True`。

❷ 为 `series` 变量显式声明类型，这样 Mypy 的输出更易读懂。¹⁶

¹⁶如果不显式声明，那么经 Mypy 推导，`series` 的类型为 `Generator[Tuple[builtins.int, builtins.str*], None, None]`。这个类型虽然冗长，但是与 `Iterator[tuple[int, str]]` 相容，详见 17.12 节。

❸ 这个 `if` 语句禁止在运行测试时执行后面 3 行。

❹ `reveal_type()` 不能在运行时调用，因为它不是常规函数，而是 Mypy 提供的调试设施，因此无须使用 `import` 导入。Mypy 每遇到一个伪函数调用 `reveal_type()` 就输出一个调试消息，显示参数的推导类型。

❺ Mypy 对这一行报错。

上述测试能通过。不过，即使没有 `top.py` 中的类型提示，测试也能通过。其实，我们的目的是使用 Mypy 检查测试文件，确认 `TypeVar` 的声明是正确的。`mypy` 命令的输出如示例 8-23 所示。

示例 8-23 `mypy top_test.py` 的输出（为方便阅读添加了换行）

```
.../comparable/ $ mypy top_test.py
top_test.py:32: note:
    Revealed type is "typing.Iterator[Tuple[builtins.int,
builtins.str]]" ❶
top_test.py:33: note:
    Revealed type is "builtins.list[Tuple[builtins.int,
builtins.str]]"
top_test.py:34: note:
    Revealed type is "builtins.list[Tuple[builtins.int,
builtins.str]]" ❷
```

```
top_test.py:41: note:
    Revealed type is "builtins.list[builtins.object*]" ❸
top_test.py:43: error:
    Value of type variable "LT" of "top" cannot be "object" ❹
Found 1 error in 1 file (checked 1 source file)
```

❶ `reveal_type(series)` 显示 `test_top_tuples` 中的 `series` 是 `Iterator[tuple[int, str]]` 类型，即我显式声明的类型。

❷ `reveal_type(result)` 确认 `top` 调用返回的类型正是我想要的：根据为 `series` 声明的类型，`result` 的类型是 `list[tuple[int, str]]`。

❸ `reveal_type(series)` 显示 `test_top_objects_error` 中的 `series` 是 `list[object*]` 类型。Mypy 在推导的类型后面加了一个 `*`。在这个测试中，我没有为 `series` 注解类型，因此是推导出来的。

❹ 我是有意测试类型错误，Mypy 也确实发现了错误：可迭代对象 `series` 中元素的类型不能是 `object`（必须是 `SupportsLessThan` 类型）。



截至 Mypy 0.910（2021 年 7 月），`reveal_type` 的输出有时不显示我声明的类型，而是显示相容的类型。鉴于此，我没有使用 `typing.Iterator`，而是用了 `abc.Iterator`。请忽略这个细节。Mypy 的输出还是有用的。讨论输出时，我会假装 Mypy 已经修正了这个问题。

与抽象基类相比，协议类型的关键优势在于，类型无须做任何特殊的声明就可以与协议类型相容。如此一来，协议可以利用现有的类型或者不受我们控制的代码实现的类型创建。在需要 `SupportsLessThan` 参数的地方可以直接使用 `SupportsLessThan` 协议，无须衍生或注册 `str`、`tuple`、`float`、`set` 等类型，只要实现了 `__lt__` 方法即可。而且，类型检

查工具仍能发挥作用，因为 `SupportsLessThan` 是显式定义的协议，这与鸭子类型正好相反，鸭子类型隐含的协议对类型检查工具是不可见的。

特殊的类 `Protocol` 由“PEP 544—Protocols: Structural subtyping (static duck typing)”引入。示例 8-21 揭示了这个功能为什么属于**静态鸭子类型**范畴：`top` 函数的 `series` 参数采用的注解方式想表达的意思是：“`series` 的名义类型无关紧要，只要实现 `__lt__` 方法即可。”Python 的鸭子类型并不阻止我们隐含这层意思，只是静态类型检查工具无从知晓。类型检查工具不能读取使用 C 语言编写的 CPython 源码，也无法在控制台中试验，以确定 `sorted` 函数只能处理支持 `<` 运算符的元素。

现在，可以让静态类型检查工具明确知晓鸭子类型的深意。所以我们说，`typing.Protocol` 实现的是**静态鸭子类型**。¹⁷

¹⁷不知道静态鸭子类型这个术语是谁发明的，不过随着 Go 语言的出现，这种说法越来越常见。Go 语言中的接口语义更像 Python 的协议，与 Java 的名义接口不太一样。

关于 `typing.Protocol`，要讲的内容还有很多。第 13 章将比较结构类型、鸭子类型和抽象基类（形式化协议的另一种方式），届时再做讨论。另外，15.2 节将解释如何使用 `@typing.overload` 声明重载的函数签名，其中包括一个大量使用 `typing.Protocol` 和有界的 `TypeVar` 的示例。



使用 `typing.Protocol` 可以注解 8.4 节定义的 `double` 函数，功能没有一点儿损失。关键是要定义一个含有 `__mul__` 方法的协议类。建议你尝试一下。实现方式见 13.6.1 节。

8.5.11 Callable

`collections.abc` 模块提供的 `Callable` 类型（尚未使用 Python 3.9 的用户在 `typing` 模块中寻找）用于注解回调参数或高阶函数返

回的可调用对象。Callable 类型可像下面这样参数化。

```
Callable[[ParamType1, ParamType2], ReturnType]
```

参数列表，即这里的 [ParamType1, ParamType2]，可以包含零或多个类型。

下面以 18.3 节实现的简单交互式解释器中的 repl 函数为例。¹⁸

¹⁸REPL 是 Read-Eval-Print-Loop 的简称，这是交互式解释器的基本行为。

```
def repl(input_fn: Callable[[Any], str] = input) -> None:
```

常规使用过程中，repl 函数使用 Python 内置函数 input 读取用户输入的表达式。然而，如果是做自动化测试或与其他输入源集成，则 repl 函数会接受一个可选的参数 input_fn。这是一个 Callable，参数类型和返回值类型都与 input 相同。

在 typedshed 项目中，内置函数 input 的签名如下所示。

```
def input(__prompt: Any = ...) -> str: ...
```

input 函数的签名与下面的 Callable 类型提示相容。

```
Callable[[Any], str]
```

可选或关键字参数类型没有专门的注解句法。正如 typing.Callable 文档所述：“这种函数类型很少用作回调类型。”如果能让类型提示匹配的签名灵活一些，可以把整个参数列表替换成 ...，如下所示。

```
Callable[..., ReturnType]
```

泛化类型参数与类型层次结构的交互引入了一个新类型概念：型变 (variance)。

Callable 类型的型变

假设一个温度控制系统中有示例 8-24 所示的 `update` 函数。这个函数很简单，调用 `probe` 函数获取当前温度，再调用 `display` 函数向用户显示当前温度。出于教学目的，`probe` 和 `display` 都会作为参数传给 `update`。这个示例的目的是比较两种 `Callable` 注解：一种有返回值类型，另一种有参数类型。

示例 8-24 说明型变

```
from collections.abc import Callable

def update( ❶
    probe: Callable[[], float], ❷
    display: Callable[[float], None] ❸
) -> None:
    temperature = probe()
    # 假设这里有大量控制代码
    display(temperature)

def probe_ok() -> int: ❹
    return 42

def display_wrong(temperature: int) -> None: ❺
    print(hex(temperature))

update(probe_ok, display_wrong) # 类型错误 ❻

def display_ok(temperature: complex) -> None: ❼
    print(temperature)

update(probe_ok, display_ok) # OK ❸
```

❶ `update` 的参数是两个可调用对象。

❷ `probe` 必须是不接受参数并会返回一个 `float` 值的可调用对象。

❸ `display` 接受一个 `float` 参数并会返回 `None`。

④ `probe_ok` 与 `Callable[[], float]` 相容，因为返回 `int` 值对预期 `float` 值的代码没有影响。

⑤ `display_wrong` 不与 `Callable[[float], None]` 相容，因为预期 `int` 参数的函数不一定能处理 `float` 值。例如，Python 函数 `hex` 接受 `int` 值，但是拒绝 `float` 值。

⑥ Mypy 对这一行报错，因为 `display_wrong` 与 `update` 函数的 `display` 参数的类型提示不相容。

⑦ `display_ok` 与 `Callable[[float], None]` 相容，因为接受 `complex` 值的函数也能处理 `float` 参数。

⑧ 通过了 Mypy 的检查。

综上所述，如果预期接受返回 `float` 值的回调，则提供返回 `int` 值的回调是可以的，因为在预期 `float` 值的地方都能使用 `int` 值。

正式地说，`Callable[[], int]` 是 `Callable[[], float]` 的子类型，因为 `int` 是 `float` 的子类型。这意味着，那个 `Callable` 的返回值类型经历了**协变**（**covariant**），因为 `int` 和 `float` 之间具有子类型关系，而且变化方向与 `Callable` 类型中返回值的类型变化方向相同。

反过来，如果回调预期处理 `float` 值，却提供接受 `int` 参数的回调，则会导致类型错误。

正式地说，`Callable[[int], None]` 不是 `Callable[[float], None]` 的子类型。虽然 `int` 是 `float` 的子类型，但是在参数化 `Callable` 类型中，关系是相反的，即 `Callable[[float], None]` 是 `Callable[[int], None]` 的子类型。因此我们说，那个 `Callable` 声明的参数类型经历了**逆变**（**contravariant**）。

15.7 节将进一步说明型变，给出一些不变 (invariant) 类型、协变类型和逆变类型的示例。



目前，可以认为大多数参数化泛型是**不变的**，这样更容易理解。如果声明 `scores: list[float]`，就只能把 `list[float]` 值赋给 `scores`，不能使用声明为 `list[int]` 或 `list[complex]` 的对象。

- 不接受 `list[int]` 对象的原因是，`scores` 中不能存放 `float` 值，而代码可能需要这么做。
- 不接受 `list[complex]` 对象的原因是，代码可能需要排序 `scores`，找出中位数，而 `complex` 未提供 `__lt__`，所以 `list[complex]` 不可排序。

下面是本章要讲的最后一个特殊类型。

8.5.12 NoReturn

这个特殊类型仅用于注解绝不返回的函数的返回值类型。这类函数通常会抛出异常。标准库中有很多这样的函数。

例如，`sys.exit()` 会抛出 `SystemExit`，终止 Python 进程。

在 `typedshed` 项目中，该函数的签名如下所示。

```
def exit(__status: object = ...) -> NoReturn: ...
```

`__status` 是仅限位置参数，而且有默认值。存根文件未明确给出默认值，而是使用 `...` 代替。`__status` 的类型是 `object`，因此也可以是 `None`，所以没必要注解为 `Optional[object]`。

示例 24-6 将使用 `NoReturn` 注解 `__flag_unknown_attrs` 方法。该方法会构建一个对用户友好而且全面的错误消息，然后抛出

AttributeError。

本章内容很多，就快结束了。8.6 节介绍位置参数和变长参数的注解。

8.6 注解仅限位置参数和变长参数

回到示例 7-9 中的 `tag` 函数。上一次见到该函数的签名是在“仅限位置参数”一节。

```
def tag(name, /, *content, class_=None, **attrs):
```

下面是带完整注解的 `tag` 函数签名。由于签名较长，因此按照格式化工具 `blue` 的方式分成了几行。

```
from typing import Optional

def tag(
    name: str,
    /,
    *content: str,
    class_: Optional[str] = None,
    **attrs: str,
) -> str:
```

注意任意个位置参数的类型提示 `*content: str`，这表明这些参数必须是 `str` 类型。在函数主体中，局部变量 `content` 的类型为 `tuple[str, ...]`。

在这个示例中，任意个关键字参数的类型提示是 `**attrs: str`，因此在函数主体中，`attrs` 的类型为 `dict[str, str]`。如果类型提示是 `**attrs: float`，那么在函数主体中，`attrs` 的类型为 `dict[str, float]`。

如果 `attrs` 参数接受不同类型的值，则需要使用 `Union[]` 或 `Any`，注解为 `**attrs: Any`。

针对仅限位置参数的 / 表示法只可在 Python 3.8 及以上版本中使用。在 Python 3.7 或以下版本中，这会导致句法错误。PEP 484 约定，在仅限位置参数的名称前加两个下划线。下面使用 PEP 484 约定的方式注解 tag 函数的签名，这一次分成两行。

```
from typing import Optional

def tag(__name: str, *content: str, class_: Optional[str] = None,
        **attrs: str) -> str:
```

以上两种声明仅限位置参数的方式 Mypy 均可识别并检验。

最后，简单讲一下类型提示及其支持的静态类型系统的局限性。

8.7 类型不完美，测试须全面

大型企业基准代码的维护人员反映，静态类型检查工具能发现很多 bug，而且这个阶段发现的 bug 比上线运行之后发现的 bug 修复成本更低。然而，有必要指出的是，早在引入静态类型之前，自动化测试就已经是行业标准做法，我熟知的公司均已广泛采用。

虽然静态类型优势诸多，但是也不能保证绝对正确。静态类型很难发现以下问题。

误报

代码中正确的类型被检查工具报告有错误。

漏报

代码中不正确的类型没有被检查工具报告有错误。

此外，如果对所有代码都做类型检查，那么我们将失去 Python 的一些表现力。

- 一些便利的功能无法做静态检查，比如像 `config(**settings)` 这种参数拆包。
- 一般来说，类型检查工具对特性（property）、描述符、元类和元编程等高级功能的支持很差，或者根本无法理解。
- 类型检查工具跟不上 Python 版本的变化（有时落后不止一年），可能拒绝使用语言新功能的代码，甚至崩溃。

常见的数据约束在类型系统中无法表达，即使是简单的约束。例如，类型提示不能确保“数量必须是大于 0 的整数”或“标签必须是 6~12 个 ASCII 字母的字符串”。通常，类型提示对捕获业务逻辑中的错误没有帮助。

考虑到这些缺点，类型提示不能作为软件质量的保障支柱，而且盲目使用只会放大缺点。

建议把静态类型检查工具纳入现代 CI 流水线，与测试运行程序、lint 程序等结合在一起使用。CI 流水线的目的是减少软件故障，自动化测试可以捕获许多超出类型提示能力范围的 bug。Python 写出的代码都能使用 Python 测试，有没有类型提示无关紧要。



本节的标题和结论受 Bruce Eckel 的一篇文章启发。¹⁹ 那篇文章题为“Strong Typing vs. Strong Testing”，收录在由 Joel Spolsky 所著的 *The Best Software Writing I* 一书中。Bruce 很喜欢 Python，写了几本讲 C、Java、Scala 和 Kotlin 的书。在那篇文章中，他说他一直推崇静态类型，直到接触 Python 才恍然大悟：“如果一个 Python 程序有足够的单元测试，那么它就可以像 C、Java 或 C# 程序一样稳健（不过 Python 测试编写起来更快）。”

¹⁹本节原文标题为“Imperfect Typing and Strong Testing”。——译者注

至此，我们对 Python 类型提示的介绍将暂告一段落。第 15 章会继续这个话题，涉及泛化类、型变、签名重载、类型校正等。另外，本书

的多个示例中会出现类型提示的身影。

8.8 本章小结

本章首先简要介绍了渐进式类型概念，然后开始实践。不借助读取类型提示的工具很难理解渐进式类型，因此我们在 Mypy 错误报告的指引之下，为一个函数添加了注解。

接着，我们又回到渐进式类型概念上，指出这其实是一种混合概念，综合了 Python 传统的鸭子类型和 Java、C++ 等静态类型语言的名义类型。

本章大部分篇幅分门别类介绍了注解可用的主要类型。本章讲到的很多类型与我们熟悉的 Python 对象类型（例如容器、元组和可调用对象）有关，不过也延伸到了泛型表示法（例如 `Sequence[float]`）。这些类型中有很多是在 `typing` 模块中临时实现的，因为直到 Python 3.9 改造标准类型之后才支持泛化。

有些类型是特殊的实体。`Any`、`Optional`、`Union` 和 `NoReturn` 不关联内存中的实际对象，只存在于类型系统的抽象层面上。

我们研究了参数化泛型和类型变量，为类型提示提供了更大的灵活性，而且不失类型安全性。

引入 `Protocol` 后，参数化泛型更具表现力。`Protocol` 在 Python 3.8 中才出现，还未大范围使用，但是重要性不容忽视。`Protocol` 使得静态鸭子类型成为可能。静态鸭子类型是 Python 内在的鸭子类型和名义类型之间的重要桥梁，令静态类型检查工具能捕获更多的 bug。

介绍一些类型时，我们使用 Mypy 做试验，利用 Mypy 提供的魔法函数 `reveal_type()` 观察类型检查错误和推导的类型。

最后又介绍了如何注解仅限位置参数和变长参数。

类型提示是一个复杂的话题，还在不断发展中。幸运的是，这是可选功能，因此 Python 广泛的用户群体不受影响。请不要听信类型布道者的话，认为所有 Python 代码都需要类型提示。

Python 类型提示由荣誉的仁慈“独裁者”²⁰ 全力推动，为表感激，本章开头和结尾都引用了他的话。

²⁰Benevolent Dictator For Life (BDFL)。参见 Guido van Rossum 对这个称呼的考证文章“Origin of BDFL”。

我不希望在道德上有义务为一个 Python 版本一直添加类型提示。我坚信，类型提示有存在的必要，然而很多时候得不偿失。用与不用由你自己选择，这多好。²¹

——Guido van Rossum

²¹出自 YouTube 视频“Type Hints by Guido van Rossum (March 2015)”。引用的内容从 13 分 40 秒开始。为了表达清楚，我稍微做了修改。

8.9 延伸阅读

Bernát Gábor 在他的一篇质量优秀的博文“The state of type hints in Python”中写道：

需要编写单元测试的地方都应添加类型提示。

我十分推崇测试，不过我经常做验证性编程。当验证想法时，测试和类型提示没有多大作用，只会拖慢进程。

在我能找到的资料中，Gábor 的文章对 Python 类型提示的介绍最好，Geir Arne Hjelle 的“Python Type Checking (Guide)”一文也不错。Claudio Jolowicz 的“Hypermodern Python Chapter 4: Typing”一文则更简短，也讲到了运行时类型检查验证。

如果想深入了解，那么 Mypy 文档是最好的资料。不管使用哪个类型检查工具，Mypy 文档都有参考价值，因为 Mypy 文档不光解读 Mypy 工具自身，也有教程，以及涉及 Python 类型一般性话题的参考页面。Mypy 文档还提供了一份便利的速查表，并对常见问题给出了解决方案。

`typing` 模块文档适合快速浏览，不触及深层细节。“PEP 483—The Theory of Type Hints”对型变做了深入解读，还使用 `Callable` 说明了逆变。最根本的参考资料必然是与类型有关的 20 多个 PEP 文档。PEP 的目标受众是 Python 核心开发人员和 Python 指导委员会，如果没有大量预备知识，则读起来肯定费力。

前面说过，第 15 章将继续探讨类型话题，15.10 节还会提供更多参考资料。表 15-1 列出了截至 2021 年年底已经通过或正在讨论的与类型有关的 PEP。

“Awesome Python Typing”仓库收集了相关工具和资料的链接，具有一定的参考价值。

杂谈

行动起来

忘掉可望不可及的超轻单车，忘掉华丽的运动衫，忘掉夹在小踏板上厚实的鞋子，忘掉无尽里程的磨炼。像小时候那样骑车吧，只有跨上单车你才能感受骑行的真正乐趣。

——Grant Petersen

《单车手册：放在口袋里的单车实用指南》

如果编程不是你的本职工作，而是协助你工作的工具，或者用来学习、捣鼓小项目，又或者只是个人兴趣，那么可能不需要类型提示，就像大多数骑自行车的人不需要硬鞋底和金属防滑钉一样。

动手编程吧。

类型的认知效应

我担心类型提示对 Python 编程风格会产生影响。

我同意大多数 API 用户能从类型提示中受益。但是，Python 吸引我的原因之一是，Python 提供的函数足够强大，完全可以取代整个 API，而且我们自己也可以编写同样强大的函数。以内置函数 `max()` 为例。这个函数功能强大，但不难理解。然而，读到 15.2.1 节你会发现，为了正确注解该函数，要编写 14 行类型提示，这还不包括为类型提示提供支持的一个 `typing.Protocol` 定义和几个 `TypeVar` 定义。

我担心的是，倘若代码库对类型提示提出严格要求，则程序员会打消编写这种函数的念头。

根据维基百科，“语言相对论”（也叫萨丕尔-沃夫假说）是一个“主张语言结构对使用者的世界观或认知有影响的理论”。维基百科进一步说道：

- **强版本**认为，语言**决定**思维，语系限制并决定认知范畴；
- **弱版本**认为，语系和用法只**影响**思维和决策。

语言学家普遍认为强版本是错误的，但有经验证据支持弱版本。

我不知道编程语言方面有没有相关研究，但是根据我的经验，编程语言对我处理问题的方式确实产生了很大的影响。我从事编程工作使用的第一门编程语言是 8 位计算机时代的 Applesoft BASIC。BASIC 不直接支持递归，必须自己动手实现调用栈。所以，我从不考虑使用递归算法或数据结构。我知道这些东西在某种概念层面上是存在的，但是我绝不使用它们解决问题。

几十年后，我接触到了 Elixir。起初，我喜欢用递归解决问题，不加甄别。后来发现，我的很多方案，使用 Elixir `Enum` 模块和

`Stream` 模块中现有的函数更简单。我了解到，地道的 Elixir 应用程序级代码很少显式递归调用，而是使用底层已经实现递归的枚举和流。

语言相对论可以解释一个普遍的观点（也是未经证实的），即学习不同的编程语言，尤其是支持不同编程范式的语言，能让你成为更好的程序员。有了使用 Elixir 的经验，编写 Python 或 Go 代码时，我更有可能使用函数式编程模式。

现在来看具体的例子。

如果 Kenneth Reitz 当初决定（或者受命于领导）注解所有函数，那么 `requests` 包的 API 势必截然不同。他的初衷是写出易于使用、灵活且强大的 API。他成功了，`requests` 包非常受欢迎。2020 年 5 月，根据 PyPI Stats，`requests` 包排名第 4，一天的下载量达到 260 万次。第 1 名是 `requests` 的一个依赖，即 `urllib3`。

2017 年，`requests` 包的维护人员决定，不投入时间编写类型提示。维护人员之一 Cory Benfield 在一封电子邮件中说道：

我觉得符合 Python 风格的 API 最不需要使用类型系统，因为其得到的回报最少。

在那封邮件中，Benfield 给出了一个极端示例：倘若为 `requests.request()` 函数的 `files` 关键字参数添加类型定义，将是下面这样。

```
Optional[
    Union[
        Mapping[
            basestring,
            Union[
                Tuple[basestring, Optional[Union[basestring,
file]]],
                Tuple[basestring, Optional[Union[basestring,
file]]],
```

```

        Optional[basestring]],
        Tuple[basestring, Optional[Union[basestring,
file]]],
        Optional[basestring], Optional[Headers]]
    ]
],
Iterable[
    Tuple[
        basestring,
        Union[
            Tuple[basestring, Optional[Union[basestring,
file]]],
            Tuple[basestring, Optional[Union[basestring,
file]]],
            Optional[basestring]],
        Tuple[basestring, Optional[Union[basestring,
file]]],
        Optional[basestring], Optional[Headers]]
    ]
]
]
]

```

而且，还要先定义以下类型。

```

Headers = Union[
    Mapping[basestring, basestring],
    Iterable[Tuple[basestring, basestring]],
]

```

假如维护人员当初下决心实现 100% 的类型提示覆盖率，那么 `requests` 包还能这么受欢迎吗？另一个重量级包 `SQLAlchemy` 也没有对类型提示太过上心。

这些包之所以优秀，就是因为坚守了 Python 的动态本性。

虽然类型提示有好处，但也要付出代价。

首先，要投入大量时间理解类型系统的机制。这只是一次性成本。

除此之外，还有经常性成本，永无终期。

如果无论什么都要做类型检查，则必将损失部分 Python 表现力。参数拆包（例如 `config(**settings)`）这种好用的功能则超出了类型检查工具的能力范围。

如果想对 `config(**settings)` 这样的调用做类型检查，则必须把每个参数拆解开来。这让我回想起了 35 年前编写的 Turbo Pascal 代码。

对使用元编程的库来说，想添加注解很难，有时根本不可能。是的，元编程有点儿泛滥，不过很多 Python 包这么好用还是得益于元编程。

在大公司中，如果自上而下严格规定，必须添加类型提示，没有例外，那么我敢打赌，我们很快就会看到有人使用代码生成工具，减少 Python 源码中的样板代码——这是动态程度较低的语言的常见做法。

对于某些项目和情形，类型提示是没有意义的。即便能起到一定作用，作用也不大。任何合理的类型提示策略都要包含例外规定。

开创了面向对象编程的图灵奖获得者 Alan Kay 曾经说过：

有些人对类型系统非常虔诚，作为一名数学家，我喜欢类型系统的想法，但从来没有人造出一个范围合理的类型系统。

22

感谢 Guido 把类型定为可选功能。请根据实际需求使用类型，不要一处不落，全部注解。严格遵守 Java 1.5 那种编程风格是不可取的。

鸭子类型优势显著

鸭子类型符合我的思维方式。静态鸭子类型是很好的折中方案，使得静态类型检查成为可能，而且不失一些灵活性，还不囿于某

些名义类型系统的复杂性。

在 PEP 544 之前，类型提示给我的感觉是不太符合 Python 风格。Python 引入 `typing.Protocol` 之后，让我欣喜若狂。“黑暗势力”终于被压制住了。

泛化还是特化？

从 Python 的角度来看，在类型上下文中使用术语“泛化”（generic）落后了。“generic”一般有两个意思：“通用的”和“无商标的”。

以 `list` 和 `list[str]` 为例。前者属于泛化，可接受任何对象；后者属于特化，只接受 `str`。

然而，“泛化”在 Java 中说得通。在 Java 1.5 之前，所有容器（除了神奇的 `array`）都是“特化”的，只能存放 `Object` 引用，从容器中取出的项都要校正类型才能使用。从 Java 1.5 开始，容器接受类型参数，已经“泛化”了。

²²来源是“A Conversation with Alan Kay”。

第 9 章 装饰器和闭包

有很多人抱怨，把这个功能命名为“装饰器”不好。主要原因是，这个名称与《设计模式：可复用面向对象软件的基础》（后面简称《设计模式》）中使用的不一致。**装饰器**这个名称可能更适合在编译器领域使用，因为它会遍历并注解句法树。

——“PEP 318—Decorators for Functions and Methods”

函数装饰器允许在源码中“标记”函数，以某种方式增强函数的行为。这是一个强大的功能，但是如果掌握，则必须理解闭包，即捕获函数主体外部定义的变量。

Python 3.0 引入的保留关键字 `nonlocal` 鲜为人知。作为 Python 程序员，如果严格遵守基于类的面向对象编程方式，那么即便不知道它的存在也不受影响。然而，如果想自己实现函数装饰器，则必须了解闭包的方方面面，因此也就需要掌握 `nonlocal`。

除了在装饰器中有用，闭包还是回调式编程和函数式编程风格的重要基础。

本章的最终目标是解释清楚函数装饰器的工作原理，从最简单的注册装饰器开始，一直讲到较为复杂的参数化装饰器。然而，在实现这一目标之前，需要讨论以下话题：

- Python 如何求解装饰器句法；
- Python 如何判断变量是不是局部的；
- 闭包存在的原因和工作原理；
- `nonlocal` 能解决什么问题。

掌握这些基础知识后，可以再进行进一步探讨装饰器：

- 实现行为良好的装饰器；

- 标准库中强大的装饰器：`@cache`、`@lru_cache` 和 `@singledispatch`；
- 实现一个参数化装饰器。

9.1 本章新增内容

Python 3.9 新引入的缓存装饰器 `functools.cache` 比传统的 `functools.lru_cache` 更简单，因此本章会先对其进行讲述。`functools.lru_cache` 和 Python 3.8 新增的简化版将在 9.9.2 节讨论。

9.9.3 节内容有扩充，增加了类型提示，自 Python 3.7 开始，这是 `functools.singledispatch` 的推荐用法。

9.10 节增加了基于类的示例，即示例 9-27。

我调整了全书结构，把第 1 版的第 6 章（现第 10 章）移到了第二部分末尾。第 1 版中的 7.3 节也移到了第 10 章，与使用可调用对象实现的其他策略设计模式放在一起。

下面先简要介绍装饰器的基础知识，然后再讨论本章开篇列出的各个话题。

9.2 装饰器基础知识

装饰器是一种可调用对象，其参数是另一个函数（被装饰的函数）。

装饰器可能会对装饰的函数做些处理，然后返回函数，或者把函数替换成另一个函数或可调用对象。¹

¹把这句话中的“函数”换成“类”，差不多就是类装饰器的作用。第 24 章将讨论类装饰器。

也就是说，假如有一个名为 `decorate` 的装饰器：

```
@decorate
def target():
    print('running target()')
```

那么上述代码的效果与下述写法一样。

```
def target():
    print('running target()')

target = decorate(target)
```

两种写法的最终结果一样：上述两个代码片段执行完毕后，`target` 名称都会绑定 `decorate(target)` 返回的函数——可能是原来那个名为 `target` 的函数，也可能是另一个函数。

为了确认被装饰的函数被替换了，请看示例 9-1 中的控制台会话。

示例 9-1 装饰器通常会把一个函数替换成另一个函数

```
>>> def deco(func):
...     def inner():
...         print('running inner()')
...         return inner ❶
...
>>> @deco
... def target(): ❷
...     print('running target()')
...
>>> target() ❸
running inner()
>>> target ❹
<function deco.<locals>.inner at 0x10063b598>
```

- ❶ `deco` 返回内部的函数对象 `inner`。
- ❷ 使用 `deco` 装饰 `target`。
- ❸ 调用被装饰的 `target`，运行的其实是 `inner`。
- ❹ 查看对象，发现 `target` 现在是 `inner` 的引用。

严格来说，装饰器只是语法糖。如前所述，装饰器可以像常规的可调用对象那样调用，传入另一个函数。有时，这样做其实更方便，尤其是做**元编程**（在运行时改变程序的行为）时。

综上所述，装饰器有以下 3 个基本性质。

- 装饰器是一个函数或其他可调用对象。
- 装饰器可以把被装饰的函数替换成别的函数。
- 装饰器在加载模块时立即执行。

下面重点讲解第 3 点。

9.3 Python 何时执行装饰器

装饰器的一个关键性质是，它们在被装饰的函数定义之后立即运行。这通常是在**导入时**（例如，当 Python 加载模块时）。以示例 9-2 中的 registration.py 模块为例。

示例 9-2 registration.py 模块

```
registry = [] ❶

def register(func): ❷
    print(f'running register({func})') ❸
    registry.append(func) ❹
    return func ❺

@register ❻
def f1():
    print('running f1()')

@register
def f2():
    print('running f2()')

def f3(): ❼
    print('running f3()')

def main(): ❽
```

```
print('running main()')
print('registry ->', registry)
f1()
f2()
f3()

if __name__ == '__main__':
    main() ⑨
```

- ❶ registry 保存被 @register 装饰的函数引用。
- ❷ register 的参数是一个函数。
- ❸ 为了演示，显示被装饰的函数。
- ❹ 把 func 存入 registry。
- ❺ 返回 func：必须返回函数，这里返回的函数与通过参数传入的函数一样。
- ❻ 使用 @register 装饰 f1 和 f2。
- ❼ 没有装饰 f3。
- ❽ main 首先显示 registry，然后调用 f1()、f2() 和 f3()。
- ❾ 只有把 registration.py 当作脚本运行时才调用 main()。

把 registration.py 当作脚本运行，得到的输出如下所示。

```
$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at
0x100631c80>]
running f1()
running f2()
running f3()
```

注意，`register` 在模块中其他函数之前运行（两次）。调用 `register` 时，传给它的参数是被装饰的函数，例如 `<function f1 at 0x100631bf8>`。

加载模块后，`registry` 中有两个被装饰函数（`f1` 和 `f2`）的引用。这两个函数，以及 `f3`，只在 `main` 显式调用它们时才执行。

如果是导入 `registration.py` 模块（不作为脚本运行），则输出如下所示。

```
>>> import registration
running register(<function f1 at 0x10063b1e0>)
running register(<function f2 at 0x10063b268>)
```

这种情况下，如果查看 `registry` 的值，则得到的输出如下所示。

```
>>> registration.registry
[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]
```

示例 9-2 主要想强调，函数装饰器在导入模块时立即执行，而被装饰的函数只在显式调用时运行。由此可以看出 Python 程序员所说的**导入时**和**运行时**之间有什么区别。

9.4 注册装饰器

考虑到装饰器在真实代码中的常用方式，示例 9-2 有两处不寻常的地方。

- 装饰器函数与被装饰的函数在同一个模块中定义。实际情况是，装饰器通常在一个模块中定义，然后再应用到其他模块中的函数上。
- `register` 装饰器返回的函数与通过参数传入的函数相同。实际上，大多数装饰器会在内部定义一个函数，然后将其返回。

虽然示例 9-2 中的 `register` 装饰器原封不动地返回了被装饰的函数，但是这种技术并非毫无用处。很多 Python 框架会使用这样的装饰器把函数添加到某种中央注册处，例如把 URL 模式映射到生成 HTTP 响应的函数的注册处。这种注册装饰器可能会也可能不会更改被装饰的函数。

不过，大多数装饰器会更改被装饰的函数。通常的做法是，返回在装饰器内部定义的函数，取代被装饰的函数。涉及内部函数的代码基本上离不开闭包。为了理解闭包，需要后退一步，先研究 Python 中的变量作用域规则。

9.5 变量作用域规则

示例 9-3 定义并测试了一个函数，该函数会读取两个变量的值：一个是通过函数的参数传入的局部变量 `a`，另一个是函数没有定义的变量 `b`。

示例 9-3 一个函数，该函数会读取一个局部变量和一个全局变量

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

出现错误并不奇怪。接着示例 9-3，如果先给全局变量 `b` 赋值，然后再调用 `f1`，则不会出错。

```
>>> b = 6
>>> f1(3)
```

```
3  
6
```

下面来看一个可能会让你吃惊的示例。

看一下示例 9-4 中的 `f2` 函数。前两行代码与示例 9-3 中的 `f1` 一样，然后为 `b` 赋值。可是，赋值前面那个 `print` 失败了。

示例 9-4 `b` 是局部变量，因为在函数主体中给它赋值了

```
>>> b = 6  
>>> def f2(a):  
...     print(a)  
...     print(b)  
...     b = 9  
...  
>>> f2(3)  
3  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    File "<stdin>", line 3, in f2  
UnboundLocalError: local variable 'b' referenced before assignment
```

注意，首先输出的是 3，这表明执行了 `print(a)` 语句。但是，第二个语句，即 `print(b)`，绝对不会执行。一开始我很吃惊，觉得会打印 6，因为有一个全局变量 `b`，而且局部变量 `b` 是在 `print(b)` 之后才赋值的。

可事实是，Python 编译函数主体时，判断 `b` 是局部变量，因为在函数内给它赋值了。生成的字节码证实了这种判断。所以，Python 会尝试从局部作用域中获取 `b`。后面调用 `f2(3)` 时，`f2` 的主体顺利获取并打印局部变量 `a` 的值，但是在尝试获取局部变量 `b` 的值时，发现 `b` 没有绑定值。

这不是 bug，而是一种设计选择：Python 不要求声明变量，但是会假定在函数主体中赋值的变量是局部变量。这比 JavaScript 的行为好多了，JavaScript 也不要求声明变量，但是如果忘记把变量声明为局部变量（使用 `var`），则可能会在不知情的情况下破坏全局变量。

在函数中赋值时，如果想让解释器把 `b` 当成全局变量，为它分配一个新值，就要使用 `global` 声明。

```
>>> b = 6
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...
>>> f3(3)
3
6
>>> b
9
```

通过以上示例可以发现两种作用域。

模块全局作用域

在类或函数块外部分配值的名称。

`f3` 函数局部作用域

通过参数或者在函数主体中直接分配值的名称。

变量还有可能出现在第 3 个作用域中，我们称之为“非局部”作用域。这个作用域是闭包的基础，稍后详述。

了解 Python 的变量作用域之后，9.6 节可以讨论闭包了。如果好奇示例 9-3 和示例 9-4 中两个函数生成的字节码有什么区别，请阅读以下附注栏中的内容。

比较字节码

`dis` 模块为反汇编 Python 函数字节码提供了简单的方式。示例 9-5 和示例 9-6 分别是示例 9-3 中的 `f1` 和示例 9-4 中的 `f2` 的字节码。

示例 9-5 反汇编示例 9-3 中的 f1 函数

```
>>> from dis import dis
>>> dis(f1)
 2          0 LOAD_GLOBAL          0 (print) ❶
          3 LOAD_FAST              0 (a) ❷
          6 CALL_FUNCTION          1 (1 positional, 0
keyword pair)
          9 POP_TOP

 3          10 LOAD_GLOBAL          0 (print)
          13 LOAD_GLOBAL          1 (b) ❸
          16 CALL_FUNCTION          1 (1 positional, 0
keyword pair)
          19 POP_TOP
          20 LOAD_CONST            0 (None)
          23 RETURN_VALUE
```

❶ 加载全局名称 print。

❷ 加载局部名称 a。

❸ 加载全局名称 b。

请比较示例 9-5 中 f1 的字节码和示例 9-6 中 f2 的字节码。

示例 9-6 反汇编示例 9-4 中的 f2 函数

```
>>> dis(f2)
 2          0 LOAD_GLOBAL          0 (print)
          3 LOAD_FAST              0 (a)
          6 CALL_FUNCTION          1 (1 positional, 0
keyword pair)
          9 POP_TOP

 3          10 LOAD_GLOBAL          0 (print)
          13 LOAD_FAST              1 (b) ❶
          16 CALL_FUNCTION          1 (1 positional, 0
keyword pair)
          19 POP_TOP

 4          20 LOAD_CONST            1 (9)
          23 STORE_FAST            1 (b)
```

```
26 LOAD_CONST
29 RETURN_VALUE
```

```
0 (None)
```

❶ 加载**局部**名称 `b`。这表明，虽然在后面才为 `b` 赋值，但是编译器会把 `b` 视作局部变量，因为变量的种类（是不是局部变量）在函数主体中不能改变。

运行字节码的 CPython 虚拟机（virtual machine, VM）是栈机器，因此 `LOAD` 操作和 `POP` 操作引用的是栈。深入说明 Python 操作码不在本书范畴之内，不过 `dis` 模块文档有说明。

9.6 闭包

在博客圈，人们有时会把闭包和匿名函数弄混。这是有历史原因的：在函数内部定义函数以前并不常见，也不容易，直到出现匿名函数。而且，只有涉及嵌套函数时才有闭包问题。因此，很多人是同时知道这两个概念的。

其实，闭包就是延伸了作用域的函数，包括函数（姑且叫 `f` 吧）主体中引用的非全局变量和局部变量。这些变量必须来自包含 `f` 的外部函数的局部作用域。

函数是不是匿名的没有关系，关键是它能访问主体之外定义的非全局变量。

这个概念难以掌握，最好通过示例理解。

假如有个名为 `avg` 的函数，它的作用是计算不断增加的系列值的平均值，例如，计算整个历史中某个商品的平均收盘价。新价格每天都在增加，因此计算平均值时要考虑到目前为止的所有价格。

先来看 `avg` 函数的用法。

```
>>> avg(10)
10.0
```



```
>>> avg(11)
10.5
>>> avg(12)
11.0
```

avg 从何而来，它又在哪里保存历史值呢？

初学者可能会像示例 9-7 那样使用基于类的实现。

示例 9-7 average_oo.py：一个计算累计平均值的类

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total / len(self.series)
```

Averager 类的实例是可调对象。

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

示例 9-8 是函数式实现，使用了高阶函数 `make_averager`。

示例 9-8 average.py：一个计算累计平均值的高阶函数

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total / len(series)
```

```
return averager
```

调用 `make_averager`，返回一个 `averager` 函数对象。每次调用，`averager` 都会把参数添加到系列值中，然后计算当前平均值，如示例 9-9 所示。

示例 9-9 测试示例 9-8

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(15)
12.0
```

注意，这两个示例有相似之处：调用 `Averager()` 或 `make_averager()` 得到一个可调用对象 `avg`，它会更新历史值，然后计算当前平均值。在示例 9-7 中，`avg` 是 `Averager` 类的实例；在示例 9-8 中，`avg` 是内部函数 `averager`。不管怎样，只需调用 `avg(n)`，把 `n` 放入系列值中，然后重新计算平均值即可。

作为 `Averager` 类的实例，`avg` 在哪里存储历史值很明显：实例属性 `self.series`。但是，第二个示例中的 `avg` 函数在哪里寻找 `series` 呢？

注意，`series` 是 `make_averager` 函数的局部变量，因为赋值语句 `series = []` 在 `make_averager` 函数的主体中。但是，调用 `avg(10)` 时，`make_averager` 函数已经返回，局部作用域早就“烟消云散”了。

如图 9-1 所示，在 `averager` 函数中，`series` 是**自由变量**（free variable）。自由变量是一个术语，指未在局部作用域中绑定的变量。

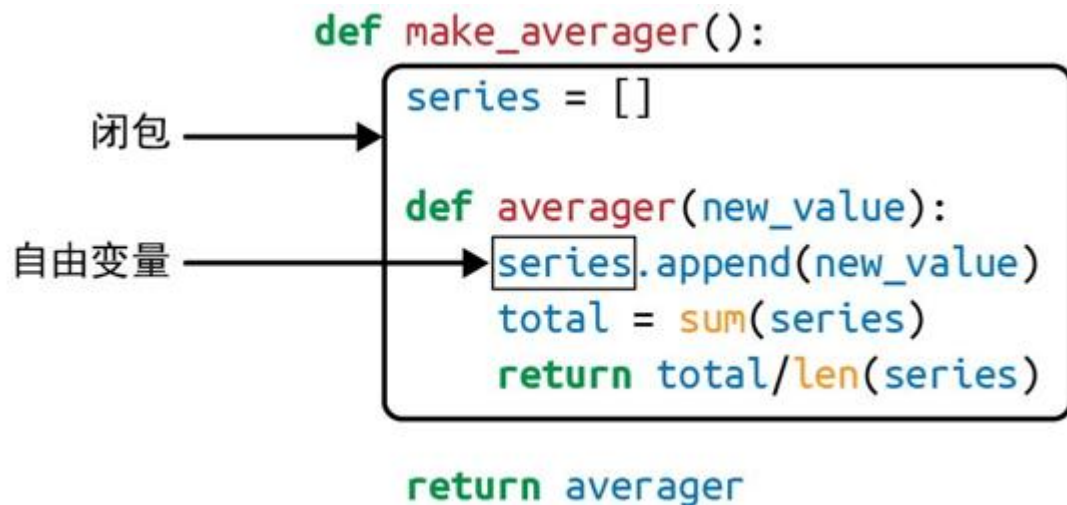


图 9-1: averager 函数的闭包延伸到自身的作用域之外，包含自由变量 series 的绑定

查看返回的 `averager` 对象，我们发现 Python 在 `__code__` 属性（表示编译后的函数主体）中保存局部变量和自由变量的名称，如示例 9-10 所示。

示例 9-10 查看 `make_averager`（参见示例 9-8）创建的函数

```
>>> avg.__code__.co_varnames  
('new_value', 'total')  
>>> avg.__code__.co_freevars  
('series',)
```

`series` 的值在返回的 `avg` 函数的 `__closure__` 属性中。
`avg.__closure__` 中的各项对应 `avg.__code__.co_freevars` 中的一个名称。这些项是 `cell` 对象，有一个名为 `cell_contents` 的属性，保存着真正的值。这些属性的值如示例 9-11 所示。

示例 9-11 接续示例 9-9

```
>>> avg.__code__.co_freevars  
('series',)  
>>> avg.__closure__  
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
```

```
>>> avg.__closure__[0].cell_contents  
[10, 11, 12]
```

综上所述，闭包是一个函数，它保留了定义函数时存在的自由变量的绑定。如此一来，调用函数时，虽然定义作用域不可用了，但是仍可使用那些绑定。

注意，只有嵌套在其他函数中的函数才可能需要处理不在全局作用域中的外部变量。这些外部变量位于外层函数的局部作用域内。

9.7 nonlocal 声明

前面实现 `make_averager` 函数的方法效率不高。在示例 9-8 中，我们把所有值存储在历史数列中，然后在每次调用 `averager` 时使用 `sum` 求和。更好的实现方式是，只存储目前的总值和项数，根据这两个数计算平均值。

示例 9-12 中的实现有缺陷，只是为了阐明观点。你能看出缺陷在哪里吗？

示例 9-12 一个计算累计平均值的高阶函数，不保存所有历史值，但有缺陷

```
def make_averager():  
    count = 0  
    total = 0  
  
    def averager(new_value):  
        count += 1  
        total += new_value  
        return total / count  
  
    return averager
```

使用示例 9-12 中定义的函数，结果如下所示。

```
>>> avg = make_averager()  
>>> avg(10)
```

```
Traceback (most recent call last):
...
UnboundLocalError: local variable 'count' referenced before
assignment
>>>
```

问题是，对于数值或任何不可变类型，`count += 1` 语句的作用其实与 `count = count + 1` 一样。因此，实际上我们在 `averager` 的主体中为 `count` 赋值了，这会把 `count` 变成局部变量。`total` 变量也受这个问题影响。

示例 9-8 则没有这个问题，因为没有给 `series` 赋值，只是调用 `series.append`，并把它传给 `sum` 和 `len`。也就是说，我们利用了“列表是可变对象”这一事实。

但是，数值、字符串、元组等不可变类型只能读取，不能更新。如果像 `count = count + 1` 这样尝试重新绑定，则会隐式创建局部变量 `count`。如此一来，`count` 就不是自由变量了，因此不会保存到闭包中。

为了解决这个问题，Python 3 引入了 `nonlocal` 关键字。它的作用是把变量标记为自由变量，即便在函数中为变量赋予了新值。如果为 `nonlocal` 声明的变量赋予新值，那么闭包中保存的绑定也会随之更新。最新版 `make_averager` 的正确实现如示例 9-13 所示。

示例 9-13 计算累计平均值，不保存所有历史（使用 `nonlocal` 修正）

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

学会使用 `nonlocal` 之后，接下来让我们总结一下 Python 查找变量的方式。

变量查找逻辑

Python 字节码编译器根据以下规则获取函数主体中出现的变量 `x`。²

²感谢技术审校 Leonardo Rochaël 建议做个总结。

- 如果是 `global x` 声明，则 `x` 来自模块全局作用域，并赋予那个作用域中 `x` 的值。³
- 如果是 `nonlocal x` 声明，则 `x` 来自最近一个定义它的外层函数，并赋予那个函数中局部变量 `x` 的值。
- 如果 `x` 是参数，或者在函数主体中赋了值，那么 `x` 就是局部变量。
- 如果引用了 `x`，但是没有赋值也不是参数，则遵循以下规则。
 - 在外层函数主体的局部作用域（非局部作用域）内查找 `x`。
 - 如果在外层作用域内未找到，则从模块全局作用域内读取。
 - 如果在模块全局作用域内未找到，则从 `__builtins__.__dict__` 中读取。

³Python 没有程序全局作用域，只有模块全局作用域。

在对 Python 闭包有了一定的认识后，下面可以使用嵌套函数着手实现装饰器了。

9.8 实现一个简单的装饰器

示例 9-14 定义了一个装饰器，该装饰器会在每次调用被装饰的函数时计时，把运行时间、传入的参数和调用的结果打印出来。

示例 9-14 `clockdeco0.py`：一个会显示函数运行时间的简单的装饰器

```

import time

def clock(func):
    def clocked(*args): ❶
        t0 = time.perf_counter()
        result = func(*args) ❷
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) -> {result!r}')
        return result
    return clocked ❸

```

- ❶ 定义内部函数 `clocked`，它接受任意个位置参数。
- ❷ 这行代码行之有效，因为 `clocked` 的闭包中包含自由变量 `func`。
- ❸ 返回内部函数，取代被装饰的函数。

示例 9-15 演示了 `clock` 装饰器的用法。

示例 9-15 使用 `clock` 装饰器

```

import time
from clockdeco0 import clock

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

if __name__ == '__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(6)')
    print('6! =', factorial(6))

```

运行示例 9-15，输出如下所示。

```
$ python3 clockdeco_demo.py
***** Calling snooze(.123)
[0.12363791s] snooze(0.123) -> None
***** Calling factorial(6)
[0.00000095s] factorial(1) -> 1
[0.00002408s] factorial(2) -> 2
[0.00003934s] factorial(3) -> 6
[0.00005221s] factorial(4) -> 24
[0.00006390s] factorial(5) -> 120
[0.00008297s] factorial(6) -> 720
6! = 720
```

工作原理

如前所述，以下内容：

```
@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

其实等价于以下内容。

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

factorial = clock(factorial)
```

也就是说，在这两种情况下，`factorial` 函数都作为 `func` 参数传给 `clock` 函数（参见示例 9-14），`clock` 函数返回 `clocked` 函数，然后 Python 解释器把 `clocked` 赋值给 `factorial`（前一种情况是在背后赋值）。导入 `clockdeco_demo` 模块，查看 `factorial` 的 `__name__` 属性，会看到如下结果。

```
>>> import clockdeco_demo
>>> clockdeco_demo.factorial.__name__
'clocked'
>>>
```

可见，现在 `factorial` 保存的其实是 `clocked` 函数的引用。自此之后，每次调用 `factorial(n)` 执行的都是 `clocked(n)`。

`clocked` 大致做了下面几件事。

01. 记录初始时间 `t0`。
02. 调用原来的 `factorial` 函数，保存结果。
03. 计算运行时间。
04. 格式化收集的数据，然后打印出来。
05. 返回第 2 步保存的结果。

这是装饰器的典型行为：把被装饰的函数替换成新函数，新函数接受的参数与被装饰的函数一样，而且（通常）会返回被装饰的函数本该返回的值，同时还会做一些额外操作。



Gamma 等人所著的《设计模式》一书是这样概述装饰器模式的：“动态地给一个对象添加一些额外的职责。”函数装饰器符合这种说法。但是，从实现层面上看，Python 装饰器与该书所说的装饰器没有多少相似之处。本章最后的“杂谈”部分会进一步探讨这个话题。

示例 9-14 实现的 `clock` 装饰器有几个缺点：不支持关键字参数，而且遮盖了被装饰函数的 `__name__` 属性和 `__doc__` 属性。示例 9-16 使用 `functools.wraps` 装饰器把相关的属性从 `func` 身上复制到了 `clocked` 中。此外，这个新版还能正确处理关键字参数。

示例 9-16 `clockdeco.py`：改进后的 `clock` 装饰器

```
import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = [repr(arg) for arg in args]
```

```
        arg_lst.extend(f'{k}={v!r}' for k, v in kwargs.items())
        arg_str = ', '.join(arg_lst)
        print(f'[{elapsed:0.8f}s] {name}({arg_str}) ->
{result!r}')
    return result
    return clocked
```

`functools.wraps` 只是标准库中开箱即用的装饰器之一。9.9 节将介绍 `functools` 模块中最让人印象深刻的装饰器，即 `cache`。

9.9 标准库中的装饰器

Python 内置了 3 个用于装饰方法的函数：`property`、`classmethod` 和 `staticmethod`。`property` 将在 22.4 节讨论，`classmethod` 和 `staticmethod` 将在 11.5 节讨论。

示例 9-16 用到了另一个重要的装饰器，即 `functools.wraps`。它的作用是协助构建行为良好的装饰器。标准库中最吸引人的几个装饰器，即 `cache`、`lru_cache` 和 `singledispatch`，均来自 `functools` 模块。下面会分别介绍它们。

9.9.1 使用 `functools.cache` 做备忘

`functools.cache` 装饰器实现了**备忘**（memoization）。⁴ 这是一项优化技术，能把耗时的函数得到的结果保存起来，避免传入相同的参数时重复计算。

⁴注意，这个词的英文没有拼错。memoization 是一个计算机科学术语，与“memorization”（记住）有那么一点儿关系，但不是同一个概念。



`functools.cache` 是 Python 3.9 新增的。如果想使用 Python 3.8 运行本节的示例，请把 `@cache` 换成 `@lru_cache`。对于更早的 Python 版本，必须调用装饰器，写成 `@lru_cache()`（详见 9.9.2 节）。

生成第 n 个斐波那契数这种慢速递归函数适合使用 `@cache`，如示例 9-17 所示。

示例 9-17 生成第 n 个斐波那契数，递归方式非常耗时

```
from clockdeco import clock

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

if __name__ == '__main__':
    print(fibonacci(6))
```

运行 `fibonacci_demo.py` 的结果如下所示。除了最后一行，其他输出都是 `clock` 装饰器生成的。

```
$ python3 fibonacci_demo.py
[0.00000042s] fibonacci(0) -> 0
[0.00000049s] fibonacci(1) -> 1
[0.00006115s] fibonacci(2) -> 1
[0.00000031s] fibonacci(1) -> 1
[0.00000035s] fibonacci(0) -> 0
[0.00000030s] fibonacci(1) -> 1
[0.00001084s] fibonacci(2) -> 1
[0.00002074s] fibonacci(3) -> 2
[0.00009189s] fibonacci(4) -> 3
[0.00000029s] fibonacci(1) -> 1
[0.00000027s] fibonacci(0) -> 0
[0.00000029s] fibonacci(1) -> 1
[0.00000959s] fibonacci(2) -> 1
[0.00001905s] fibonacci(3) -> 2
[0.00000026s] fibonacci(0) -> 0
[0.00000029s] fibonacci(1) -> 1
[0.00000997s] fibonacci(2) -> 1
[0.00000028s] fibonacci(1) -> 1
[0.00000030s] fibonacci(0) -> 0
[0.00000031s] fibonacci(1) -> 1
[0.00001019s] fibonacci(2) -> 1
[0.00001967s] fibonacci(3) -> 2
[0.00003876s] fibonacci(4) -> 3
[0.00006670s] fibonacci(5) -> 5
```

```
[0.00016852s] fibonacci(6) -> 8
8
```

浪费时间的地方很明显：`fibonacci(1)` 调用了 8 次，`fibonacci(2)` 调用了 5 次.....但是，如果增加两行代码，使用 `cache`，那么性能将显著改善，如示例 9-18 所示。

示例 9-18 使用缓存实现，速度更快

```
import functools

from clockdeco import clock

@functools.cache ❶
@clock ❷
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)

if __name__ == '__main__':
    print(fibonacci(6))
```

❶ 这一行可在 Python 3.9 或以上版本中使用。支持更早的 Python 版本的做法见 9.9.2 节。

❷ 这里叠放了装饰器：`@cache` 应用到 `@clock` 返回的函数上。



叠放装饰器

如果想理解叠放装饰器，那么需要记住一点：`@` 是一种语法糖，其作用是把装饰器函数应用到下方的函数上。多个装饰器的行为就像调用嵌套函数一样。以下内容：

```
@alpha
@beta
def my_fn():
    ...
```

等同于以下内容。

```
my_fn = alpha(beta(my_fn))
```

也就是说，首先应用 `beta` 装饰器，然后再把返回的函数传给 `alpha`。

这样一来，对于每个 `n` 值，`fibonacci` 函数只被调用一次。

```
$ python3 fibo_demo_lru.py
[0.00000043s] fibonacci(0) -> 0
[0.00000054s] fibonacci(1) -> 1
[0.00006179s] fibonacci(2) -> 1
[0.00000070s] fibonacci(3) -> 2
[0.00007366s] fibonacci(4) -> 3
[0.00000057s] fibonacci(5) -> 5
[0.00008479s] fibonacci(6) -> 8
8
```

如果要计算 `fibonacci(30)`，使用示例 9-18 中的版本，总计会调用 31 次，耗时 0.000 17 秒，而示例 9-17 中未做缓存的版本在配有 Intel Core i7 处理器的笔记本计算机中则耗时 12.09 秒，因为仅 `fibonacci(1)` 就要调用 832 040 次，总计调用 2 692 537 次。

被装饰的函数所接受的参数必须可哈希，因为底层 `lru_cache` 使用 `dict` 存储结果，字典的键取自传入的位置参数和关键字参数。

除了优化递归算法，`@cache` 在从远程 API 中获取信息的应用程序中也能发挥巨大作用。



如果缓存较大，则 `functools.cache` 有可能耗尽所有可用内存。在我看来，`@cache` 更适合短期运行的命令行脚本使用。对于长期运行的进程，推荐使用 `functools.lru_cache`，并合理设置 `maxsize` 参数（详见 9.9.2 节）。

9.9.2 使用 lru_cache

`functools.cache` 装饰器只是对较旧的 `functools.lru_cache` 函数的简单包装。其实，`functools.lru_cache` 更灵活，而且兼容 Python 3.8 及之前的版本。

`@lru_cache` 的主要优势是可以通过 `maxsize` 参数限制内存用量上限。`maxsize` 参数的默认值相当保守，只有 128，即缓存最多只能有 128 条。

LRU 是“Least Recently Used”的首字母缩写，表示一段时间不用的缓存条目会被丢弃，为新条目腾出空间。

从 Python 3.8 开始，`lru_cache` 有两种使用方式。下面是最简单的方式。

```
@lru_cache
def costly_function(a, b):
    ...
```

另一种方式是从 Python 3.2 开始支持的加上 `()` 作为函数调用。

```
@lru_cache()
def costly_function(a, b):
    ...
```

两种用法都采用以下默认参数。

```
maxsize=128
```

设定最多可以存储多少条目。缓存满了之后，最不常用的条目会被丢弃，为新条目腾出空间。为了得到最佳性能，应将 `maxsize` 设为 2 的次方。如果传入 `maxsize=None`，则 LRU 逻辑将被彻底禁用，因此缓存速度更快，但是条目永远不会被丢弃，这可能会消耗过多内存。`@functools.cache` 就是如此。

```
typed=False
```

决定是否把不同参数类型得到的结果分开保存。例如，在默认设置下，被认为是值相等的浮点数参数和整数参数只存储一次，即 `f(1)` 调用和 `f(1.0)` 调用只对应一个缓存条目。如果设为 `typed=True`，则在不同的条目中存储可能不一样的结果。

以下示例不使用参数的默认值调用 `@lru_cache`。

```
@lru_cache(maxsize=2**20, typed=True)
def costly_function(a, b):
    ...
```

接下来介绍另一个强大的装饰器：`functools.singledispatch`。

9.9.3 单分派泛化函数

假设我们在开发一个调试 Web 应用程序的工具，想生成 HTML，以显示不同类型的 Python 对象。

为此，可能会编写如下函数。

```
import html

def htmlize(obj):
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'
```

这个函数适用于任何 Python 类型，但是现在我们想扩展一下，以特别的方式显示如下类型。

```
str
```

把内部的换行符替换为 `'<br/>\n'`，不使用 `<pre>` 标签，而使用 `<p>`。

```
int
```

以十进制和十六进制显示数（bool 除外）。

list

输出一个 HTML 列表，根据各项的类型进行格式化。

float 和 Decimal

正常输出值，外加分数形式（为什么不呢？）。

我们想要的行为如示例 9-19 所示。

示例 9-19 生成 HTML 的 `htmlize()` 函数，调整了几种对象的输出

```
>>> htmlize({1, 2, 3}) ❶
'<pre>{1, 2, 3}</pre>'
>>> htmlize(abs)
'<pre>&lt;built-in function abs&gt;</pre>'
>>> htmlize('Heimlich & Co.\n- a game') ❷
'<p>Heimlich & Co.<br/>\n- a game</p>'
>>> htmlize(42) ❸
'<pre>42 (0x2a)</pre>'
>>> print(htmlize(['alpha', 66, {3, 2, 1}])) ❹
<ul>
<li><p>alpha</p></li>
<li><pre>66 (0x42)</pre></li>
<li><pre>{1, 2, 3}</pre></li>
</ul>
>>> htmlize(True) ❺
'<pre>True</pre>'
>>> htmlize(fractions.Fraction(2, 3)) ❻
'<pre>2/3</pre>'
>>> htmlize(2/3) ❼
'<pre>0.6666666666666666 (2/3)</pre>'
>>> htmlize(decimal.Decimal('0.02380952'))
'<pre>0.02380952 (1/42)</pre>'
```

❶ `htmlize()` 函数本就针对 `object`，相当于不匹配其他特殊参数类型时的一种兜底实现。

- ❷ `str` 对象也做 HTML 转义，不过是放在 `<p></p>` 内，而且在每个 `'\n'` 之前插入换行标签 `<br/>`。
- ❸ 以十进制和十六进制显示 `int` 值，放在 `<pre></pre>` 内。
- ❹ 列表中的各项根据类型被格式化，整个序列会被渲染成一个 HTML 列表。
- ❺ `bool` 是 `int` 的子类型，不过得到了特殊对待。
- ❻ 以分数形式显示 `Fraction` 对象。
- ❼ 以近似的分数显示 `float` 值和 `Decimal` 值。

单分派函数

因为 Python 不支持 Java 那种方法重载，所以不能使用不同的签名定义 `htmlize` 的变体，以不同的方式处理不同的数据类型。在 Python 中，常见的做法是把 `htmlize` 变成一个分派函数，使用一串 `if/elif/...` 或 `match/case/...` 调用专门的函数，例如 `htmlize_str`、`htmlize_int` 等。这样不仅不便于模块的用户扩展，还显得笨拙：时间一长，分派函数 `htmlize` 的内容会变得很多，而且它与各个专门函数之间的耦合也太紧密。

`functools.singledispatch` 装饰器可以把整体方案拆分成多个模块，甚至可以为第三方包中无法编辑的类型提供专门的函数。使用 `@singledispatch` 装饰的普通函数变成了**泛化函数**（generic function，指根据第一个参数的类型，以不同方式执行相同操作的一组函数）的入口。这才称得上是**单分派**。如果根据多个参数选择专门的函数，那就是**多分派**。具体做法如示例 9-20 所示。

示例 9-20 使用 `@singledispatch` 创建

`@htmlize.register` 装饰器，把多个函数绑在一起组成一个泛化函数

```

from functools import singledispatch
from collections import abc
import fractions
import decimal
import html
import numbers

@singledispatch ❶
def htmlize(obj: object) -> str:
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'

@htmlize.register ❷
def _(text: str) -> str: ❸
    content = html.escape(text).replace('\n', '<br/>\n')
    return f'<p>{content}</p>'

@htmlize.register ❹
def _(seq: abc.Sequence) -> str:
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'

@htmlize.register ❺
def _(n: numbers.Integral) -> str:
    return f'<pre>{n} (0x{n:x})</pre>'

@htmlize.register ❻
def _(n: bool) -> str:
    return f'<pre>{n}</pre>'

@htmlize.register(fractions.Fraction) ❼
def _(x) -> str:
    frac = fractions.Fraction(x)
    return f'<pre>{frac.numerator}/{frac.denominator}</pre>'

@htmlize.register(decimal.Decimal) ❽
@htmlize.register(float)
def _(x) -> str:
    frac = fractions.Fraction(x).limit_denominator()
    return f'<pre>{x} ({frac.numerator}/{frac.denominator})</pre>'

```

❶ @singledispatch 标记的是处理 object 类型的基函数。

❷ 各个专门函数使用 @«base».register 装饰。

③ 运行时传入的第一个参数的类型决定何时使用这个函数。专门函数的名称无关紧要，`_` 是一个不错的选择，简单明了。⁵

⁵可惜，Mypy 0.770 发现多个同名函数会报错。

④ 为每个需要特殊处理的类型注册一个函数，把第一个参数的类型提示设为相应的类型。

⑤ `singledispatch` 支持使用 `numbers` 包中的抽象基类。⁶

⁶尽管 8.5.7 节中的“论数字塔的倒下”否定了 `number` 包中的抽象基类，但是这些抽象基类并没有被弃用，仍在 Python 3 代码中使用。

⑥ `bool` 是 `numbers.Integral` 的子类型，但是 `singledispatch` 逻辑会寻找与指定类型最匹配的实现，与实现在代码中出现的顺序无关。

⑦ 如果不想或者不能为被装饰的类型添加类型提示，则可以把类型传给 `@«base».register` 装饰器。Python 3.4 或以上版本支持这种句法。

⑧ `@«base».register` 装饰器会返回装饰之前的函数，因此可以叠放多个 `register` 装饰器，让同一个实现支持两个或更多类型。⁷

⁷说不定以后只需要一个未参数化的 `@htmlize.register`，而在类型提示中使用 `Union` 类型。我试过，Python 会抛出 `TypeError`，提醒 `Union` 不是类。可见，尽管 `@singledispatch` 支持 PEP 484 中的句法，但是语义还未实现。



`functools.singledispatch` 自 Python 3.4 起就存在了，不过从 Python 3.7 开始才支持类型提示。示例 9-20 中最后两个函数使用的句法支持自 Python 3.4 起的所有版本。

应尽量注册处理抽象基类（例如 `numbers.Integral` 和 `abc.MutableSequence`）的专门函数，而不直接处理具体实现（例

如 `int` 和 `list`)。这样的话，代码支持的兼容类型更广泛。例如，Python 扩展可以子类化 `numbers.Integral`，使用固定的位数实现 `int` 类型。⁸

⁸例如，NumPy 实现了几个针对机器的整数类型和浮点数类型。



在单分派中使用抽象基类或 `typing.Protocol` 可以让代码支持抽象基类或实现协议的类当前和未来的具体子类或虚拟子类。抽象基类的作用和虚拟子类的概念将在第 13 章讨论。

`singledispatch` 机制的一个显著特征是，你可以在系统的任何地方和任何模块中注册专门函数。如果后来在新模块中定义了新类型，则可以轻易添加一个新的自定义函数来处理新类型。此外，还可以为不是自己编写的或者不能修改的类编写自定义函数。

`singledispatch` 是经过深思熟虑之后才添加到标准库中的，功能很多，这里无法一一说明。“PEP 443—Single-dispatch generic functions”是不错的参考资料，不过没有讲到类型提示，毕竟类型提示出现得较晚。`functools` 模块文档有所改善，`singledispatch` 条目下增加了几个使用类型提示的示例。



`@singledispatch` 不是为了把 Java 那种方法重载带入 Python。在一个类中为同一个方法定义多个重载变体比在一个函数中使用一长串 `if/elif/elif/elif` 块要好。但是，这两种方案都有缺陷，因为它们让一个代码单元（类或函数）承担的职责太多。`@singledispatch` 的优点是支持模块化扩展：模块可以为它支持的各个类型注册一个专门的函数。实践中，不可能像示例 9-20 那样把泛化函数的实现都放在同一个模块中。

目前，我们见到了几个接受参数的装饰器，例如 `@lru_cache()` 和示例 9-20 中使用 `@singledispatch` 创建的

`htmlize.register(float)`。9.10 节将说明如何构建接受参数的装饰器。

9.10 参数化装饰器

解析源码中的装饰器时，Python 会把被装饰的函数作为第一个参数传给装饰器函数。那么，如何让装饰器接受其他参数呢？答案是创建一个装饰器工厂函数来接收那些参数，然后再返回一个装饰器，应用到被装饰的函数上。是不是有点儿迷惑？肯定的。下面以我们目前见到的最简单的装饰器 `register` 为例说明，如示例 9-21 所示。

示例 9-21 示例 9-2 中 `registration.py` 模块的删减版，再次给出，方便查看

```
registry = []

def register(func):
    print(f'running register({func})')
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')

print('running main()')
print('registry ->', registry)
f1()
```

9.10.1 一个参数化注册装饰器

为了便于启用或禁用 `register` 执行的函数注册功能，为它提供一个可选的 `active` 参数，当设为 `False` 时，不注册被装饰的函数。实现方式如示例 9-22 所示。从概念上看，这个新的 `register` 函数不是装饰器，而是装饰器工厂函数。调用 `register` 函数才能返回应用到目标函数上的装饰器。

示例 9-22 为了接受参数，新的 register 装饰器必须作为函数调用

```
registry = set() ❶

def register(active=True): ❷
    def decorate(func): ❸
        print('running register'
              f' (active={active})->decorate({func})')
        if active: ❹
            registry.add(func)
        else:
            registry.discard(func) ❺

        return func ❻
    return decorate ❼

@register(active=False) ❸
def f1():
    print('running f1()')

@register() ❹
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

❶ registry 现在是一个 set 对象，这样添加和删除函数的速度更快。

❷ register 接受一个可选的关键字参数。

❸ 内部函数 decorate 是真正的装饰器。注意，它的参数是一个函数。

❹ 只有 active 参数的值（从闭包中获取）是 True 时才注册 func。

❺ 如果 active 不为 True，而且 func 在 registry 中，那就把它删除。

- ⑥ 因为 `decorate` 是装饰器，所以必须返回一个函数。
- ⑦ `register` 是装饰器工厂函数，因此返回 `decorate`。
- ⑧ `@register` 工厂函数必须作为函数调用，并且传入所需的参数。
- ⑨ 即使不传入参数，`register` 也必须作为函数调用 (`@register()`)，返回真正的装饰器 `decorate`。

关键是，`register()` 要返回 `decorate`。应用到被装饰的函数上的是 `decorate`。

示例 9-22 中的代码在 `registration_param.py` 模块中。导入该模块，得到的结果如下所示。

```
>>> import registration_param
running register(active=False)->decorate(<function f1 at
0x10063c1e0>)
running register(active=True)->decorate(<function f2 at
0x10063c268>)
>>> registration_param.registry
[<function f2 at 0x10063c268>]
```

注意，只有 `f2` 出现在了 `registry` 中，`f1` 不在其中，因为传给 `register` 装饰器工厂函数的参数是 `active=False`，所以应用到 `f1` 上的 `decorate` 没有把它添加到 `registry` 中。

如果不使用 `@` 句法，那么就要像常规函数那样调用 `register`。如果想把 `f` 添加到 `registry` 中，那么装饰 `f` 函数的句法是 `register()(f)`；如果不想添加 `f`（或把它删除），则句法是 `register(active=False)(f)`。示例 9-23 演示了如何把函数添加到 `registry` 中，以及如何从中删除函数。

示例 9-23 使用示例 9-22 中的 `registration_param` 模块

```
>>> from registration_param import *
running register(active=False)->decorate(<function f1 at
```

```
0x10073c1e0>
running register(active=True)->decorate(<function f2 at
0x10073c268>)
>>> registry ❶
{<function f2 at 0x10073c268>}
>>> register()(f3) ❷
running register(active=True)->decorate(<function f3 at
0x10073c158>)
<function f3 at 0x10073c158>
>>> registry ❸
{<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
>>> register(active=False)(f2) ❹
running register(active=False)->decorate(<function f2 at
0x10073c268>)
<function f2 at 0x10073c268>
>>> registry ❺
{<function f3 at 0x10073c158>}
```

- ❶ 导入这个模块时，f2 在 registry 中。
- ❷ register() 表达式返回 decorate 并应用到 f3 上。
- ❸ 前一行把 f3 添加到 registry 中。
- ❹ 这个调用从 registry 中删除 f2。
- ❺ 确认 registry 中只有 f3。

参数化装饰器的原理相当复杂，刚刚讨论的那个例子比大多数例子简单。参数化装饰器通常会把被装饰的函数替换掉，而且结构上需要多一层嵌套。接下来会探讨这种函数金字塔。

9.10.2 参数化 clock 装饰器

本节再次探讨 clock 装饰器，为它添加一个功能：让用户传入一个格式字符串，控制被装饰函数的输出，如示例 9-24 所示。

示例 9-24 clockdeco_param.py 模块：参数化 clock 装饰器


```

import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT): ❶
    def decorate(func): ❷
        def clocked(*_args): ❸
            t0 = time.perf_counter()
            _result = func(*_args) ❹
            elapsed = time.perf_counter() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args) ❺
            result = repr(_result) ❻
            print(fmt.format(**locals())) ❼
            return _result ❸
        return clocked ❹
    return decorate ❺

if __name__ == '__main__':

    @clock() ❻
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(.123)

```

❶ `clock` 是参数化装饰器工厂函数。

❷ `decorate` 是真正的装饰器。

❸ `clocked` 包装被装饰的函数。

❹ `_result` 是被装饰的函数返回的真正结果。

❺ `_args` 用于存放 `clocked` 的真正参数，`args` 是用于显示的字符串。

❻ `result` 是 `_result` 的字符串表示形式，用于显示。

❼ 这里使用 `**locals()` 是为了在 `fmt` 中引用 `clocked` 的局部变量。⁹

⁹技术审校 Miroslav Šedivý指出：“但是，代码 lint 程序将报错，提醒有未使用的变量，因为 lint 程序往往忽略 `locals()`。”是的，这再一次表明静态检查工具不鼓励使用 Python 的动态功能，而我和无数的程序员最初就是被这些动态功能吸引才爱上 Python 的。为了让 lint 程序满意，每个局部变量都要写两次：`fmt.format(elapsed=elapsed, name=name, args=args, result=result)`。我可不想这么麻烦。使用静态检查工具时，一定要知道何时应忽略工具报告的错误。

⑧ `clocked` 将取代被装饰的函数，因此它应该返回被装饰的函数返回的值。

⑨ `decorate` 返回 `clocked`。

⑩ `clock` 返回 `decorate`。

⑪ 在当前模块中测试，调用 `clock()` 时不传入参数，因此所应用的装饰器将使用默认的格式字符串。



为简单起见，示例 9-24 基于示例 9-14 中最初实现的 `clock`，而不是示例 9-16 中使用 `@functools.wraps` 改进后的版本，因为那一版增加了一层函数。

在 shell 中运行示例 9-24，得到的结果如下所示。

```
$ python3 clockdeco_param.py
[0.12412500s] snooze(0.123) -> None
[0.12411904s] snooze(0.123) -> None
[0.12410498s] snooze(0.123) -> None
```

示例 9-25 和示例 9-26 中的两个模块也使用 `clockdeco_param`，两个示例的下方给出了输出结果。

示例 9-25 `clockdeco_param_demo1.py`

```
import time
from clockdeco_param import clock

@clock('{name}: {elapsed}s')
```

```
def snooze(seconds):  
    time.sleep(seconds)  
  
for i in range(3):  
    snooze(.123)
```

以上示例的输出如下所示。

```
$ python3 clockdeco_param_demo1.py  
snooze: 0.12414693832397461s  
snooze: 0.1241159439086914s  
snooze: 0.12412118911743164s
```

示例 9-26 clockdeco_param_demo2.py

```
import time  
from clockdeco_param import clock  
  
@clock('{name}({args}) dt={elapsed:0.3f}s')  
def snooze(seconds):  
    time.sleep(seconds)  
  
for i in range(3):  
    snooze(.123)
```

以上示例的输出如下所示。

```
$ python3 clockdeco_param_demo2.py  
snooze(0.123) dt=0.124s  
snooze(0.123) dt=0.124s  
snooze(0.123) dt=0.124s
```



本书第 1 版技术审校之一 Lennart Regebro 认为，装饰器最好通过定义了 `__call__` 方法的类实现，不应像本章这样通过函数实现。我同意类更适合创建重要的装饰器，但是为了讲解这个语言功能的基本思想，函数更易于理解。9.12 节提到了一些工业级装饰器构建技术，尤其不要错过 Graham Dumpleton 的博客文章和 `wrapt` 模块。

9.10.3 节会举一个例子，按照 Regebro 和 Dumpleton 建议的方式创建装饰器。

9.10.3 基于类的 clock 装饰器

再举一个例子。示例 9-27 通过定义 `__call__` 方法的类实现了参数化装饰器 `clock`。请对比一下示例 9-24 和示例 9-27，你更喜欢哪一个？

示例 9-27 `clockdeco_cls.py` 模块：通过类实现参数化装饰器 `clock`

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

class clock: ❶

    def __init__(self, fmt=DEFAULT_FMT): ❷
        self.fmt = fmt

    def __call__(self, func): ❸
        def clocked(*_args):
            t0 = time.perf_counter()
            _result = func(*_args) ❹
            elapsed = time.perf_counter() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)
            result = repr(_result)
            print(self.fmt.format(**locals()))
            return _result
        return clocked
```

❶ 不用定义外层函数 `clock` 了，现在 `clock` 类是参数化装饰器工厂。类名使用的是小写字母 `c`，以此表明这里的实现可以直接替代示例 9-24。

❷ `clock(my_format)` 传入的参数赋值给这里的 `fmt` 参数。类构造函数返回一个 `clock` 实例，`my_format` 被存储为 `self.fmt`。

③ 有了 `__call__` 方法, `clock` 实例就成为可调用对象了。调用实例的结果是把被装饰的函数替换成 `clocked`。

④ `clocked` 包装被装饰的函数。

对函数装饰器的讨论到此结束。第 24 章会介绍类装饰器。

9.11 本章小结

本章涉及一些难以理解的内容。学习之路崎岖不平, 我已经尽可能让路途平坦顺畅。毕竟, 我们已经进入元编程领域了。

本章以编写一个没有内部函数的 `@register` 装饰器作为起始, 最后实现了有两层嵌套函数的参数化装饰器 `@clock()`。

尽管注册装饰器十分简单, 但其是在 Python 框架中有用武之地。第 10 章将使用这种注册方式实现一个“策略”设计模式。

如果想真正理解装饰器, 则不仅需要区分**导入时**和**运行时**, 还要理解变量作用域、闭包和新增的 `nonlocal` 声明。掌握闭包和 `nonlocal` 不仅对构建装饰器有帮助, 在面向事件的 GUI 程序编程和基于回调处理异步 I/O 中也用得到, 遇到适合使用函数式编程的情况更能得心应手。

参数化装饰器基本上涉及至少两层嵌套函数, 如果想使用 `@functools.wraps` 生成装饰器, 为高级技术提供更好的支持, 则嵌套层级可能会更深, 例如示例 9-18 中叠放的装饰器。对更复杂的装饰器来说, 基于类实现或许更易于理解和维护。

本章还以 `functools` 模块中强大的 `@cache` 和 `@singledispatch` 为例, 介绍了标准库中的参数化装饰器。

9.12 延伸阅读

《Effective Python：编写高质量 Python 代码的 90 个有效方法（原书第 2 版）》一书中的第 26 条实践原则给出了函数装饰器的最佳实践，建议始终使用 `functools.wraps`（像示例 9-16 那样）。¹⁰

¹⁰为了尽量保证代码简单易懂，本书中并不是所有示例都遵从 Brett Slatkin（《Effective Python：编写高质量 Python 代码的 90 个有效方法（原书第 2 版）》的作者）的优秀建议。

Graham Dumpleton 写了一系列博客文章，深入剖析了如何实现行为良好的装饰器，第一篇是“[How you implemented your Python decorator is wrong](#)”。他在这方面的渊博知识充分体现在他编写的 `wrapt` 模块中。这个模块旨在简化装饰器和动态函数包装器的实现，即使多层装饰也支持内省，而且行为正确，既可以应用到方法上，也可以作为属性描述符使用。第 23 章会讨论描述符。

《Python Cookbook（第 3 版）中文版》的第 9 章中的几个经典实例构建了基本的装饰器和特别复杂的装饰器，其中，9.6 节提到的装饰器既可以作为常规的装饰器调用（例如 `@clock`），也可以作为装饰器工厂函数调用（例如 `@clock()`）。

Michele Simionato 开发了一个包，根据文档，该包旨在“简化普通程序员使用装饰器的方式，并且通过各种复杂的示例推广装饰器”。这个包是 `decorator`，可以通过 PyPI 安装。

Python Decorator Library 维基页面在 Python 刚添加装饰器功能时就创建了，里面有十几个示例。那个页面是几年前开始编写的，虽然有些技术已经过时了，但仍是很棒的灵感来源。

Fredrik Lundh 写的一篇简短的博客文章“[Closures in Python](#)”解读了闭包这个术语。

“PEP 3104—Access to Names in Outer Scopes”说明了引入 `nonlocal` 声明的原因：重新绑定既不在局部作用域中也不在全局作用域中的名称。这份 PEP 还概述了其他动态语言（Perl、Ruby、

JavaScript 等) 解决这个问题的方式, 以及在 Python 中几种可用的设计方案的优缺点。

“PEP 227—Statically Nested Scopes”更偏重理论, 说明了 Python 2.1 引入的词法作用域。词法作用域在 Python 2.1 中是一种备选方案, 到 Python 2.2 变成了标准方案。此外, 这份 PEP 还说明了 Python 中闭包的基本原理和实现方式的选择。

PEP 443 对单分派泛化函数的基本原理和细节做了说明。Guido van Rossum 很久以前 (2005 年 3 月) 写的一篇博客文章“Five-Minute Multimethods in Python”详细说明了如何使用装饰器实现泛化函数 (也叫多方法, multimethod)。他给出的代码支持多分派 (根据多个位置参数进行分派)。虽然 Guido 写的多方法代码很棒, 但那只是教学示例。如果想使用现代的技术实现多分派泛化函数并在生产环境中使用, 可以用 Martijn Faassen 开发的 Reg。Martijn 还是模型驱动型 REST 式 Web 框架 Morepath 的开发者。

杂谈

动态作用域与词法作用域

任何把函数当作一等对象的语言其设计者都面对一种情况: 作为一等对象, 在某个作用域中定义的函数, 可能会在其他作用域中调用。问题是, 如何求解自由变量? 首先出现的最简单的处理方式是使用“动态作用域”。也就是说, 根据调用函数的环境求解自由变量。

如果 Python 使用动态作用域, 不支持闭包, 那么就可以像下面这样使用 avg 函数 (与示例 9-8 类似)。

```
>>> ### 这不是真实的Python控制台会话!  ###
>>> avg = make_averager()
>>> series = [] ❶
>>> avg(10)
10.0
>>> avg(11) ❷
```

```
10.5
>>> avg(12)
11.0
>>> series = [1] ❸
>>> avg(5)
3.0
```

❶ 使用 `avg` 之前要自己定义 `series = []`，因此必须知道 `averager`（在 `make_averager` 内部）引用的是名为 `series` 的列表。

❷ 在背后使用 `series` 累计要计入平均值的值。

❸ 执行 `series = [1]` 后，之前的列表消失了。同时计算两个独立的累计平均值时，可能发生这种意外。

函数应该是黑盒，把实现隐藏起来，不让用户知道。但是对动态作用域来说，如果函数使用自由变量，那么程序员就必须知道函数的内部细节，这样才能建立正确运行所需的环境。多年前，我与文档准备语言 LaTeX 结下不解之缘，直到读了 *Practical LaTeX*（George Grätzer 著）一书才意识到，LaTeX 变量使用的是动态作用域。难怪我以前觉得 LaTeX 变量那么令人费解！

Emacs Lisp 也使用动态作用域，至少默认如此。Emacs Lisp 手册中的“Dynamic Binding”一节对此有简短说明。

动态作用域易于实现，这大概就是 John McCarthy 创建 Lisp（第一门把函数视作一等对象的语言）时采用这种方式的原因。Paul Graham 写的“The Roots of Lisp”一文对 John McCarthy 关于 Lisp 语言的题为“Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”的论文做了通俗易懂的解读，这篇论文是和贝多芬第九交响曲一样伟大的杰作。Paul Graham 使用通俗易懂的语言翻译了这篇论文，把数学原理转换成了英语和可运行的代码。

Paul Graham 的解读还指出，动态作用域难以实现。下面这段文字引自“The Roots of Lisp”一文。

就连第一个 Lisp 高阶函数示例都因为动态作用域而无法运行，这充分证明了动态作用域的危险性。McCarthy 在 1960 年可能没有全面认识到动态作用域的影响。动态作用域在各种 Lisp 实现中存在的时间特别长，直到 Sussman 和 Steele 在 1975 年开发出 Scheme 为止。词法作用域不会导致 `eval` 的定义变得多么复杂，只是编译器可能更难编写。

如今，词法作用域已成常态：根据定义函数的环境求解自由变量。词法作用域让人更难实现支持一等函数的语言，因为需要支持闭包。不过，词法作用域让代码更易于阅读。Algol 之后出现的语言大多使用词法作用域。值得注意的是，JavaScript 是个例外。在 JavaScript 中，特殊变量 `this` 最让人摸不着头脑，因为根据代码的编写方式，它既可以使用词法作用域，也可以使用动态作用域。

多年来，由于 Python 的 `lambda` 表达式不支持闭包，因此在博客圈的函数式编程极客群体中，这个功能的名声并不好。Python 2.2（2001 年 12 月发布）修正了这个问题，但是博客圈的固有印象不会轻易转变。自此之后，仅仅由于句法上的局限，`lambda` 一直处于尴尬的境地。

Python 装饰器和装饰器设计模式

Python 函数装饰器符合《设计模式》一书中对装饰器模式的一般描述：“动态地给一个对象添加一些额外的职责。就扩展功能而言，装饰器模式比子类化更灵活。”

在实现层面，Python 装饰器与装饰器设计模式不同，但是有些相似之处。

在设计模式中，Decorator 和 Component 是抽象类。为了给具体组件添加行为，具体装饰器的实例要包装具体组件的实例。《设计模式》中是这样说的：

装饰器与其所装饰的组件接口一致，因此它对使用该组件的客户透明。它会将客户请求转发给该组件，并且可能在转发前后执行一些额外的操作（例如绘制一个边框）。透明性使得你可以递归嵌套多个装饰器，从而可以添加任意多的功能。

在 Python 中，装饰器函数相当于 Decorator 的具体子类，而装饰器返回的内部函数相当于装饰器实例。返回的函数包装了被装饰的函数，这相当于设计模式中的组件。返回的函数是透明的，因为它接受相同的参数，符合组件的接口。返回的函数会把调用转发给组件，并可以在转发前后执行额外的操作。因此，前面引用那段话的最后一句可以改成：“透明性使得你可以叠放多个装饰器，从而可以添加任意多的行为。”

注意，我并不是建议在 Python 程序中使用函数装饰器实现装饰器模式。在特定情况下确实可以这么做，但是一般来说，实现装饰器模式时最好使用类表示装饰器和要包装的组件。

第 10 章 使用一等函数实现设计模式

符合模式并不表示做得对。

——Ralph Johnson
经典著作《设计模式》的作者之一¹

¹出自 2014 年 11 月 15 日 Ralph Johnson 在圣保罗大学 IME/CCSL 所做的题为“Root Cause Analysis of Some Faults in Design Patterns”的演讲。

在软件工程中，**设计模式**指解决常见设计问题的一般性方案。阅读本章无须事先知道任何设计模式。我会解释示例中用到的设计模式。

编程设计模式的使用通过 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides（“四人组”）合著的《设计模式》一书普及开来。该书涵盖 23 个模式，通过 C++ 代码定义的类编排。不过，这些模式也可用于其他面向对象语言。

虽然设计模式与语言无关，但这并不意味着每一个模式都能在每一门语言中使用。例如，读到第 17 章你会发现，在 Python 中效仿迭代器模式毫无意义，因为该模式深植语言之中，通过生成器即可使用。在 Python 中，迭代器模式无须“劳烦”类，实现代码也比典型方案少。

《设计模式》的作者在引言中承认，所用的语言决定了哪些模式可用。

程序设计语言的选择非常重要，它将影响人们理解问题的出发点。我们的设计模式采用了 Smalltalk 和 C++ 层的语言特性，这个选择实际上决定了哪些机制可以方便地实现，哪些则不能。如果采用过程式语言，那么可能就要包括诸如“继承”“封装”和“多态”的设计模式。相应地，一些特殊的面向对象语言可以直接支持

我们的某些模式，例如，CLOS 支持多方法概念，这就减少了访问者等模式的必要性。

1996 年，Peter Norvig 在题为“Design Patterns in Dynamic Languages”的演讲中指出，对于《设计模式》一书提出的 23 个模式，有 16 个在动态语言中“要么不见了，要么简化了”（第 9 张幻灯片）。他讨论的是 Lisp 语言和 Dylan 语言，不过很多相关的动态特性在 Python 中也能找到。具体而言，Norvig 建议在有一等函数的语言中重新审视“策略”“命令”“模板方法”和“访问者”等经典模式。

本章的目标是展示在某些情况下，函数可以起到类的作用，而且写出的代码可读性更高且更简洁。我们会把函数作为对象使用，重构策略模式，从而减少大量样板代码。还会讨论一种类似的方案，以简化命令模式。

10.1 本章新增内容

我把本章移到了第二部分末尾，这样就可以在 10.3 节应用一个注册装饰器，以及在示例中使用类型提示了。本章出现的大多数类型提示并不复杂，却能提升代码可读性。

10.2 案例分析：重构策略模式

如果合理利用作为一等对象的函数，则某些设计模式可以简化，而策略模式就是一个很好的例子。本节接下来的内容将说明策略模式的作用，并使用《设计模式》一书中所述的“经典”结构来实现它。如果你熟悉这个经典模式，则可以跳到 10.2.2 节，了解如何使用函数重构代码，极大地减少代码行数。

10.2.1 经典的策略模式

图 10-1 中的 UML 类图指出了策略模式对类的编排。

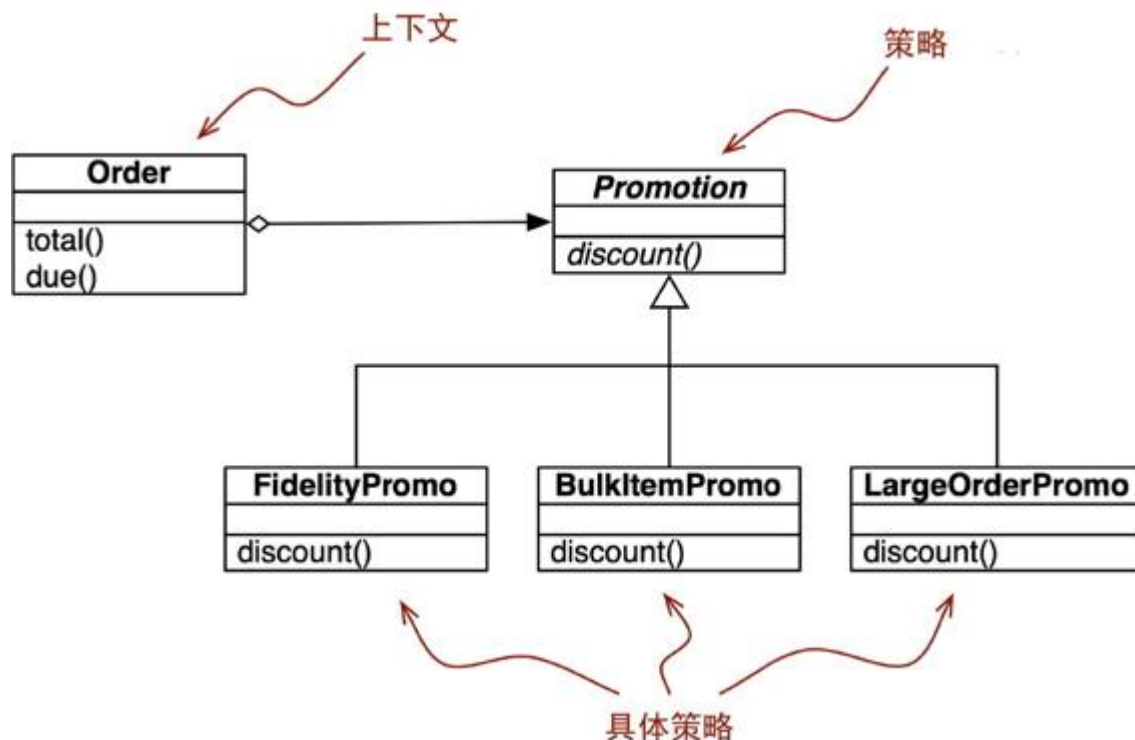


图 10-1：使用策略设计模式处理订单折扣的 UML 类图

《设计模式》一书对策略模式的概述如下。

定义一系列算法，把它们一一封装起来，并且使它们可以相互替换。本模式使得算法可以独立于使用它的客户而变化。

在电商领域应用策略模式的一个典型例子是根据客户的属性或订单中的商品计算折扣。

假如某个网店制定了以下折扣规则。

- 有 1000 或以上积分的顾客，每个订单享 5% 折扣。
- 同一订单中，单个商品的数量达到 20 个或以上，享 10% 折扣。
- 订单中不同商品的数量达到 10 个或以上，享 7% 折扣。

为简单起见，假定一个订单一次只能享用一个折扣。

策略模式的 UML 类图见图 10-1，其中涉及下列内容。

上下文

提供一个服务，把一些计算委托给实现不同算法的可互换组件。在这个电商示例中，上下文是 `Order`，它根据不同的算法计算促销折扣。

策略

实现不同算法的组件共同的接口。在这个示例中，名为 `Promotion` 的抽象类扮演这个角色。

具体策略

策略的具体子类。`FidelityPromo`、`BulkPromo` 和 `LargeOrderPromo` 是这里实现的 3 个具体策略。

示例 10-1 实现了图 10-1 中的方案。按照《设计模式》一书的说明，具体策略由上下文类的客户选择。在这个示例中，实例化订单之前，系统会以某种方式选择一种促销折扣策略，把它传给 `Order` 构造函数。具体如何选择策略，不在这个模式的职责范围内。

示例 10-1 实现 `Order` 类，支持插入式折扣策略

```
from abc import ABC, abstractmethod
from collections.abc import Sequence
from decimal import Decimal
from typing import NamedTuple, Optional

class Customer(NamedTuple):
    name: str
    fidelity: int

class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self) -> Decimal:
```

```

        return self.price * self.quantity

class Order(NamedTuple): # 上下文
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional['Promotion'] = None

    def total(self) -> Decimal:
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'

class Promotion(ABC): # 策略: 抽象基类
    @abstractmethod
    def discount(self, order: Order) -> Decimal:
        """返回折扣金额 (正值) """

class FidelityPromo(Promotion): # 第一个具体策略
    """为积分为1000或以上的顾客提供5%折扣"""

    def discount(self, order: Order) -> Decimal:
        rate = Decimal('0.05')
        if order.customer.fidelity >= 1000:
            return order.total() * rate
        return Decimal(0)

class BulkItemPromo(Promotion): # 第二个具体策略
    """单个商品的数量为20个或以上时提供10%折扣"""

    def discount(self, order: Order) -> Decimal:
        discount = Decimal(0)
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * Decimal('0.1')
        return discount

```

```

class LargeOrderPromo(Promotion): # 第三个具体策略
    """订单中不同商品的数量达到10个或以上时提供7%折扣"""

    def discount(self, order: Order) -> Decimal:
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * Decimal('0.07')
        return Decimal(0)

```

注意，示例 10-1 把 `Promotion` 定义为了抽象基类，这么做是为了使用 `@abstractmethod` 装饰器，明确表明所用的模式。

示例 10-2 是一些 `doctest`，用于在某个实现了上述规则的模块中演示和验证相关操作。

示例 10-2 使用不同促销折扣的 `Order` 类示例

```

>>> joe = Customer('John Doe', 0) ❶
>>> ann = Customer('Ann Smith', 1100)
>>> cart = (LineItem('banana', 4, Decimal('.5')), ❷
...         LineItem('apple', 10, Decimal('1.5')),
...         LineItem('watermelon', 5, Decimal(5)))
>>> Order(joe, cart, FidelityPromo()) ❸
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, FidelityPromo()) ❹
<Order total: 42.00 due: 39.90>
>>> banana_cart = (LineItem('banana', 30, Decimal('.5')), ❺
...                LineItem('apple', 10, Decimal('1.5')))
>>> Order(joe, banana_cart, BulkItemPromo()) ❻
<Order total: 30.00 due: 28.50>
>>> long_cart = tuple(LineItem(str(sku), 1, Decimal(1)) ❼
...                   for sku in range(10))
>>> Order(joe, long_cart, LargeOrderPromo()) ❽
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, LargeOrderPromo())
<Order total: 42.00 due: 42.00>

```

❶ 两位顾客：joe 的积分是 0，ann 的积分是 1100。

❷ 购物车中有 3 个商品。

- ③ FidelityPromo 未给 joe 提供折扣。
- ④ ann 得到了 5% 折扣，因为她的积分超过 1000。
- ⑤ banana_cart 中有 30 把香蕉和 10 个苹果。
- ⑥ BulkItemPromo 为 joe 购买的香蕉优惠了 1.5 美元。
- ⑦ long_order 中有 10 个不同的商品，每个商品的价格为 1 美元。
- ⑧ LargerOrderPromo 为 joe 的整个订单提供 7% 折扣。

示例 10-1 完全可用，但是在 Python 中，如果把函数当作对象使用，则实现同样的功能所需的代码更少。详见 10.2.2 节。

10.2.2 使用函数实现策略模式

在示例 10-1 中，每个具体策略都是一个类，而且都只定义了一个方法，即 `discount`。此外，策略实例没有状态（没有实例属性）。你可能会说，它们看起来像是普通函数——的确如此。示例 10-3 是对示例 10-1 的重构，把具体策略换成了简单的函数，而且去掉了抽象类 `Promo`。`Order` 类也要修改，但改动不大。²

²由于 Mypy 存在 bug，因此不得不使用 `@dataclass` 重新实现 `Order`。你可以忽略这个细节，因为也可以像示例 10-1 那样继承 `NamedTuple`。如果 `Order` 继承 `NamedTuple`，则 Mypy 0.910 在检查 `promotion` 的类型提示时将崩溃。我试过在那一行添加 `# type ignore`，不过 Mypy 依旧崩溃。如果使用 `@dataclass` 构建 `Order`，那么 Mypy 就能正确处理相同的类型提示了。截至 2021 年 7 月 19 日，9397 号工单还没有解决。但愿等你读到这里时已经修正了。

示例 10-3 Order 类和使用函数实现的折扣策略

```
from collections.abc import Sequence
from dataclasses import dataclass
from decimal import Decimal
from typing import Optional, Callable, NamedTuple
```

```

class Customer(NamedTuple):
    name: str
    fidelity: int

class LineItem(NamedTuple):
    product: str
    quantity: int
    price: Decimal

    def total(self):
        return self.price * self.quantity

@dataclass(frozen=True)
class Order: # 上下文
    customer: Customer
    cart: Sequence[LineItem]
    promotion: Optional[Callable[['Order'], Decimal]] = None ❶

    def total(self) -> Decimal:
        totals = (item.total() for item in self.cart)
        return sum(totals, start=Decimal(0))

    def due(self) -> Decimal:
        if self.promotion is None:
            discount = Decimal(0)
        else:
            discount = self.promotion(self) ❷
        return self.total() - discount

    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'

❸

def fidelity_promo(order: Order) -> Decimal: ❹
    """为积分为1000或以上的顾客提供5%折扣"""
    if order.customer.fidelity >= 1000:
        return order.total() * Decimal('0.05')
    return Decimal(0)

def bulk_item_promo(order: Order) -> Decimal:
    """单个商品的数量为20个或以上时提供10%折扣"""
    discount = Decimal(0)

```

```

    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * Decimal('0.1')
    return discount

def large_order_promo(order: Order) -> Decimal:
    """订单中不同商品的数量达到10个或以上时提供7%折扣"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * Decimal('0.07')
    return Decimal(0)

```

- ❶ 这个类型提示的意思是，`promotion` 既可以是 `None`，也可以是接收一个 `Order` 参数并返回一个 `Decimal` 值的可调用对象。
- ❷ 调用可调用对象 `self.promotion`，传入 `self`，计算折扣。
- ❸ 没有抽象类。
- ❹ 各个策略都是函数。



为什么写成 `self.promotion(self)`?

在 `Order` 类中，`promotion` 不是方法，而是一个实例属性，只不过它的值是可调用对象。因此，作为表达式的第一部分，`self.promotion` 的作用是获取可调用对象。为了调用得到的可调用对象，必须提供一个 `Order` 实例，即表达式中的 `self`。因此，表达式中出现了两个 `self`。

23.4 节将解释自动为实例绑定方法的机制。这个机制不适用于 `promotion`，因为 `promotion` 不是方法。

示例 10-3 中的代码比示例 10-1 简短。不仅如此，新的 `Order` 类使用起来更简单，如示例 10-4 中的 `doctest` 所示。

示例 10-4 以函数实现促销折扣的 `Order` 类使用示例

```

>>> joe = Customer('John Doe', 0) ❶
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, Decimal('.5')),
...         LineItem('apple', 10, Decimal('1.5')),
...         LineItem('watermelon', 5, Decimal(5))]
>>> Order(joe, cart, fidelity_promo) ❷
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, fidelity_promo)
<Order total: 42.00 due: 39.90>
>>> banana_cart = [LineItem('banana', 30, Decimal('.5')),
...                LineItem('apple', 10, Decimal('1.5'))]
>>> Order(joe, banana_cart, bulk_item_promo) ❸
<Order total: 30.00 due: 28.50>
>>> long_cart = [LineItem(str(item_code), 1, Decimal(1))
...              for item_code in range(10)]
>>> Order(joe, long_cart, large_order_promo)
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo)
<Order total: 42.00 due: 42.00>

```

❶ 与示例 10-1 一样的测试固件。

❷ 为了把折扣策略应用到 `Order` 实例上，只需把促销函数作为参数传入即可。

❸ 这个测试和下一个测试使用了不同的促销函数。

注意示例 10-4 中的标号：没必要在新建订单时实例化新的促销对象，函数拿来即用。

值得注意的是，《设计模式》一书指出：“策略对象通常是很好的享元（flyweight）。”该书的另一部分对“享元”下了定义：“享元是可共享的对象，可以同时多个上下文中使用。”共享是推荐的做法，这样在每个新的上下文（这里是 `Order` 实例）中使用相同的策略时则不必不断新建具体策略对象，从而减少消耗。因此，为了避免策略模式的一个缺点（运行时消耗），《设计模式》一书建议再使用另一个模式。但是这样做，代码行数和维护成本会不断攀升。

在复杂的情况下，当需要用具体策略维护内部状态时，可能要把策略和享元模式结合起来。但是，具体策略一般没有内部状态，只负责处

理上下文中的数据。此时，一定要使用普通函数，而不是编写只有一个方法的类，再去实现另一个类声明的单函数接口。函数比用户定义的类的实例轻量，而且无须使用享元模式，因为各个策略函数在 Python 加载模块时只创建一次。普通函数也是“可共享的对象，可以同时多个上下文中使用”。

至此，我们使用函数实现了策略模式，由此也出现了其他可能性。假设我们想创建一个“元策略”，为 `Order` 选择最佳折扣。接下来的几节会接着重构，利用函数和模块都是对象这一特点，使用不同的方式实现这个需求。

10.2.3 选择最佳策略的简单方式

下面继续使用示例 10-4 中的顾客和购物车，在此基础上添加 3 个测试，如示例 10-5 所示。

示例 10-5 `best_promo` 函数计算所有折扣，返回幅度最大的那一个

```
>>> Order(joe, long_cart, best_promo) ❶  
<Order total: 10.00 due: 9.30>  
>>> Order(joe, banana_cart, best_promo) ❷  
<Order total: 30.00 due: 28.50>  
>>> Order(ann, cart, best_promo) ❸  
<Order total: 42.00 due: 39.90>
```

- ❶ `best_promo` 为顾客 `joe` 选择 `larger_order_promo`。
- ❷ 订购大量香蕉时，`joe` 使用 `bulk_item_promo` 提供的折扣。
- ❸ 在一个简单的购物车中，`best_promo` 为忠实顾客 `ann` 提供 `fidelity_promo` 优惠的折扣。

`best_promo` 函数的实现特别简单，如示例 10-6 所示。

示例 10-6 `best_promo` 迭代一个函数列表，找出折扣幅度最大的那一个

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo] ❶

def best_promo(order: Order) -> Decimal: ❷
    """选择可用的最佳折扣"""
    return max(promo(order) for promo in promos) ❸
```

❶ `promos` 列出以函数实现的各个策略。

❷ 与其他几个 `*_promo` 函数一样，`best_promo` 函数的参数是一个 `Order` 实例。

❸ 使用生成器表达式把 `order` 传给 `promos` 列表中的各个函数，返回折扣幅度最大的那个函数。

示例 10-6 简单明了，`promos` 是函数列表。习惯函数是一等对象后，自然而然就会构建那种数据结构来存储函数。

虽然示例 10-6 可行，而且易于理解，但是存在一些重复，可能导致不易察觉的 bug：如果想添加新的促销策略，那么不仅要定义相应的函数，还要记得把它添加到 `promos` 列表中；否则，只能把新促销函数作为参数显式传给 `Order`，因为 `best_promo` 不知道新函数的存在。

这个问题的解决方案有好几种，请继续往下读。

10.2.4 找出一个模块中的全部策略

在 Python 中，模块也是一等对象，而且标准库提供了几个处理模块的函数。Python 文档对内置函数 `globals` 的描述如下。

```
globals()
```

返回一个字典，表示当前的全局符号表。这个符号表始终针对当前模块（对函数或方法来说，是指定义它们的模块，而不是调用它们的模块）。

示例 10-7 使用 `globals` 函数帮助 `best_promo` 自动找到了其他可用的 `*_promo` 函数，过程有点儿曲折。

示例 10-7 内省模块的全局命名空间，构建 `promos` 列表

```
from decimal import Decimal
from strategy import Order
from strategy import (
    fidelity_promo, bulk_item_promo, large_order_promo ❶
)

promos = [promo for name, promo in globals().items() ❷
          if name.endswith('_promo') and ❸
          name != 'best_promo' ❹]

def best_promo(order: Order) -> Decimal: ❺
    """选择可用的最佳折扣"""
    return max(promo(order) for promo in promos)
```

❶ 导入促销函数，以便其在全局命名空间中可用。³

³flake8 和 VS Code 都会发出警告，指出虽然导入了这些名称，但是并没有使用它们。按照定义，静态分析工具不能理解 Python 的动态本性。如果凡事都听这些工具的，那么结果必然是以 Python 句法写出类似 Java 那种冗长晦涩、质量低下的代码。

❷ 迭代 `globals()` 返回的字典中的各项。

❸ 只选择以 `_promo` 结尾的值。

❹ 过滤掉 `best_promo` 自身，防止调用 `best_promo` 时出现无限递归。

❺ `best_promo` 没有变化。

收集所有可用促销的另一种方法是，在一个单独的模块中保存所有的策略函数（`best_promo` 除外）。

在示例 10-8 中，最大的变化是内省名为 `promotions` 的独立模块，构建策略函数列表。注意，示例 10-8 要导入 `promotions` 模块，以及提供高阶内省函数的 `inspect` 模块。

示例 10-8 内省单独的 `promotions` 模块，构建 `promos` 列表

```
from decimal import Decimal
import inspect

from strategy import Order
import promotions

promos = [func for _, func in inspect.getmembers(promotions,
inspect.isfunction)]

def best_promo(order: Order) -> Decimal:
    """选择可用的最佳折扣"""
    return max(promo(order) for promo in promos)
```

`inspect.getmembers` 函数用于获取对象（这里是 `promotions` 模块）的属性，第二个参数是可选的判断条件（一个布尔值函数）。这里使用的判断条件是 `inspect.isfunction`，只获取模块中的函数。

不管怎么命名策略函数，示例 10-8 都行之有效。唯一的要求是，`promotions` 模块只能包含计算订单折扣的函数。当然，这是对代码的隐性假设。如果有人在 `promotions` 模块中使用不同的签名来定义函数，那么在 `best_promo` 函数尝试将其应用到订单上时会出错。

可以添加更为严格的测试，审查传给实例的参数，进一步筛选函数。示例 10-8 的目的不是提供完善的方案，而是强调模块内省的一种用途。

动态收集促销折扣函数的一种更为显式的方案是使用简单的装饰器。详见 10.3 节。

10.3 使用装饰器改进策略模式

回顾一下，示例 10-6 的主要问题是，定义中出现了函数的名称，`best_promo` 用来判断哪个折扣幅度最大的 `promos` 列表中也有函数的名称。这种重复是个问题，因为新增策略函数后可能会忘记把它添加到 `promos` 列表中，导致 `best_promo` 悄无声息忽略新策略，为系统引入不易察觉的 bug。示例 10-9 使用 9.4 节介绍的技术解决了这个问题。

示例 10-9 `promos` 列表中的值使用 `promotion` 装饰器填充

```
Promotion = Callable[[Order], Decimal]

promos: list[Promotion] = [] ❶
def promotion(promo: Promotion) -> Promotion: ❷
    promos.append(promo)
    return promo

def best_promo(order: Order) -> Decimal:
    """选择可用的最佳折扣"""
    return max(promo(order) for promo in promos) ❸

@promotion ❹
def fidelity(order: Order) -> Decimal:
    """为积分为1000或以上的顾客提供5%折扣"""
    if order.customer.fidelity >= 1000:
        return order.total() * Decimal('0.05')
    return Decimal(0)

@promotion
def bulk_item(order: Order) -> Decimal:
    """单个商品的数量为20个或以上时提供10%折扣"""
    discount = Decimal(0)
    for item in order.cart:
        if item.quantity >= 20:
```

```
        discount += item.total() * Decimal('0.1')
    return discount

@promotion
def large_order(order: Order) -> Decimal:
    """订单中不同商品的数量达到10个或以上时提供7%折扣"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * Decimal('0.07')
    return Decimal(0)
```

- ❶ `promos` 列表位于模块全局命名空间中，起初是空的。
- ❷ `Promotion` 是注册装饰器，在把 `promo` 函数添加到 `promos` 列表中之后，它会原封不动地返回 `promo` 函数。
- ❸ 无须修改 `best_promos`，因为它依赖于 `promos` 列表。
- ❹ 被 `@promotion` 装饰的函数都会添加到 `promos` 列表中。

与前几种方案相比，这种方案有以下几个优点。

- 促销策略函数无须使用特殊的名称（不用以 `_promo` 结尾）。
- `@promotion` 装饰器突出了被装饰的函数的作用，还便于临时禁用某个促销策略：把装饰器注释掉即可。
- 促销折扣策略可以在其他模块中定义，在系统中的任何地方都行，只要使用了 `@promotion` 装饰器。

10.4 节将讨论命令模式。这个设计模式也常使用单方法类实现，同样也可以换成普通函数。

10.4 命令模式

命令设计模式也可以通过把函数作为参数传递而简化。这一模式对类的编排如图 10-2 所示。

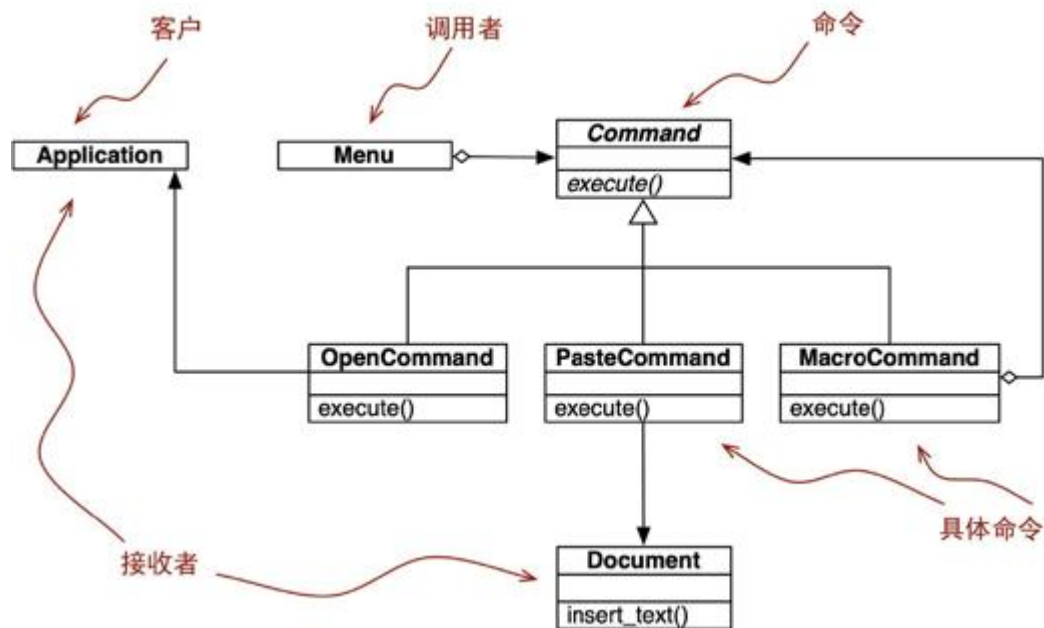


图 10-2：菜单驱动的文本编辑器的 UML 类图，使用命令设计模式实现。各个命令可以有不同的接收者，即实现操作的对象。对 PasteCommand 来说，接收者是 Document；对 OpenCommand 来说，接收者是应用程序

命令模式的目的是解耦调用操作的对象（调用者）和提供实现的对象（接收者）。在《设计模式》一书所举的示例中，调用者是图形应用程序中的菜单项，接收者是被编辑的文档或应用程序自身。

这个模式的做法是，在二者之间放一个 Command 对象，让它实现只有一个方法（execute）的接口，调用接收者中的方法执行所需的操作。这样，调用者无须了解接收者的接口，而且不同的接收者可以适应不同的 Command 子类。调用者有一个具体的命令，通过调用 execute 方法执行。注意，图 10-2 的 MacroCommand 可能会保存一系列命令，它的 execute() 方法会在各个命令上调用相同的方法。

Gamma 等人说过：“命令模式是回调机制的面向对象替代品。”问题是，我们需要回调机制的面向对象替代品吗？有时确实需要，但并非始终需要。

可以不为调用者提供 `Command` 实例，而是给它一个函数。此时，调用者不调用 `command.execute()`，而是直接调用 `command()`。`MacroCommand` 可以通过定义了 `__call__` 方法的类实现。这样，`MacroCommand` 的实例就是可调用对象，各自维护着一个函数列表，供以后调用。`MacroCommand` 的实现如示例 10-10 所示。

示例 10-10 `MacroCommand` 的各个实例都在内部存储着命令列表

```
class MacroCommand:
    """一个执行一组命令的命令"""

    def __init__(self, commands):
        self.commands = list(commands) ❶

    def __call__(self):
        for command in self.commands: ❷
            command()
```

❶ 根据 `commands` 参数构建一个列表，这样能确保参数是可迭代对象，还能在各个 `MacroCommand` 实例中保存各个命令引用的副本。

❷ 调用 `MacroCommand` 实例时，`self.commands` 中的各个命令依序执行。

复杂的命令模式（例如支持撤销操作）可能需要的不仅仅是简单的回调函数。即便如此，也可以考虑使用 Python 提供的几个替代品。

- 像示例 10-10 中 `MacroCommand` 那样的可调用实例，可以保存任何所需的状态，除了 `__call__`，还可以提供其他方法。
- 可以使用闭包在调用之间保存函数的内部状态。

使用一等函数对命令模式的重新审视到此结束。站在一定高度上看，这里采用的方式与策略模式所用的方式类似：把实现单方法接口的类的实例替换成可调用对象。毕竟，每个 Python 可调用对象都实现了单方法接口，这个方法就是 `__call__`。

10.5 本章小结

经典著作《设计模式》出版几年后，Peter Norvig 指出：“在 Lisp 或 Dylan 中，23 个设计模式中有 16 个的实现方式比在 C++ 中更简单，而且能保持同等质量，至少各个模式的某些用途如此。”（Norvig 的“Design Patterns in Dynamic Languages”演讲，第 9 张幻灯片。）Python 有些动态特性与 Lisp 和 Dylan 一样，尤其是本章着重讨论的一等函数。

本章开头引用的那句话是 Ralph Johnson 在纪念《设计模式》英文原书出版 20 周年活动上所说的，他指出了该书的缺点之一：“过多强调设计模式的结果，而没有细说过程。”⁴ 本章从策略模式开始，使用一等函数简化了实现方式。

⁴2014 年 11 月 15 日 Johnson 在 IME-USP 所做的题为“Root Cause Analysis of Some Faults in Design Patterns”的演讲。

很多情况下，在 Python 中使用函数或可调用对象实现回调更自然，这比模仿 Gamma、Helm、Johnson 和 Vlissides 在《设计模式》一书所述的策略或命令模式要好。本章对策略模式的重构和对命令模式的讨论是为了通过示例说明一种更为常见的做法：有时，设计模式或 API 要求组件实现单方法接口，而该方法有一个很宽泛的名称，例如“execute”“run”或“do_it”。在 Python 中，这些模式或 API 通常可以使用作为一等对象的函数实现，从而减少样板代码。

10.6 延伸阅读

《Python Cookbook（第 3 版）中文版》的 8.21 节使用优雅的方式实现了访问者模式，其中的 `NodeVisitor` 类把方法当作一等对象处理。

在设计模式方面，Python 程序员的阅读选择没有其他语言多。

据我所知, *Learning Python Design Patterns* (Gennadiy Zlobin 著) 是目前唯一一本专门讲解 Python 设计模式的书。不过, 该书特别薄 (100 页), 只涵盖了 23 个设计模式中的 8 个。

《Python 高级编程》是目前市面上值得一读的 Python 中级书, 该书最后一章从 Python 程序员的视角介绍了几个经典模式。

Alex Martelli 做过几次关于 Python 设计模式的演讲。他在 EuroPython 2011 上的演讲有视频, 他的个人网站中有一些幻灯片。这些年, 我找到过不同的幻灯片和视频, 长短不一, 因此你要仔细搜索他的名字和“Python Design Patterns”这些词。有家出版商告诉我, Martelli 正在写一本关于这个话题的书。出版后我肯定会拜读。

Java 相关的设计模式书很多, 其中我最喜欢的是《Head First 设计模式 (第二版)》。该书讲解了 23 个经典模式中的 16 个。如果喜欢 Head First 系列丛书的古怪风格, 而且想了解这个主题, 你会喜欢该书的。该书围绕 Java 展开, 不过第二版做了更新, 涵盖了 Java 中的一等函数, 因此部分示例与 Python 版本接近。

如果想换个新鲜的角度, 从支持鸭子类型和一等函数的动态语言入手, 那么《Ruby 设计模式》一书中有很多见解也适用于 Python。虽然 Python 和 Ruby 在句法上有很多区别, 但是二者在语义方面很接近, 比 Java 或 C++ 接近。

在“Design Patterns in Dynamic Languages”演讲中, Peter Norvig 展示了如何使用一等函数和其他动态特性简化几个经典的设计模式, 或者彻底摒除设计模式。

光看《设计模式》一书的“引言”就能让你赚回书钱。在引言中, 几位作者对 23 个设计模式一一编目, 涵盖非常重要和很少使用的模式。人们经常从该书引言中引用两个设计原则, 即“对接口编程, 而不是对实现编程”和“优先使用对象组合, 而不是类继承”。

模式在设计中的应用源自《建筑模式语言》一书。Alexander 的想法是创建一个标准的表现形式，让团队在设计建筑时共享共同的设计决策。M. J. Dominus 在“Design Patterns’Aren’t”演讲中认为，Alexander 对模式最初的愿景更深远、更人性化，在软件工程中的应用也应如此。

杂谈

Python 拥有一等函数和一等类型，Norvig 声称，这些功能对 23 个模式中的 10 个有影响（“Design Patterns in Dynamic Languages”，第 10 张幻灯片）。如第 9 章所述，Python 也有泛型函数（参见 9.9.3 节）。这与 CLOS 中的多方法类似，Gamma 等人建议使用多方法以一种简单的方式实现经典的访问者（Visitor）模式。Norvig 却说，多方法能简化生成器（Builder）模式（第 10 张幻灯片）。可见，设计模式与语言功能无法精确对应。

世界各地的课堂经常使用 Java 示例讲解设计模式。我不止一次听学生说过，他们以为设计模式在任何语言中都有用。《设计模式》一书中提出的 23 个“经典”模式，大多使用 C++ 代码说明，少量使用 Smalltalk。事实证明，这些设计模式能很好地应用到 Java 上。然而，这并不意味着所有模式都能一成不变地在任何语言中运用。该书的作者在开头就明确表明，“一些特殊的面向对象语言可以直接支持我们的某些模式”（完整的引用见本章开头）。

与 Java、C++ 或 Ruby 相比，Python 设计模式方面的图书都很薄。延伸阅读中提到的 *Learning Python Design Patterns*（Gennadiy Zlobin 著）在 2013 年 11 月才出版。而《Ruby 设计模式》在 2007 年就出版了，英文版有 384 页，比 Zlobin 的那本书多出 284 页。

如今，Python 在学术界越来越流行，希望以后会有更多以这门语言讲解设计模式的图书。此外，Java 8 引入了方法引用和匿名函数，这些广受期盼的功能有可能为 Java 催生新的模式实现方式

——要知道，语言会进化，因此运用经典设计模式的方式必定要随之进化。

夺命连环 call

为本书做最后的润色时，技术审校 Leonardo Rochael 提出了一个有趣的问题：

如果函数有 `__call__` 方法，方法也是可调用对象，那么 `__call__` 方法自身有 `__call__` 方法吗？

我不知道他的发现有没有用，但确实有趣。

```
>>> def turtle():
...     return 'eggs'
...
>>> turtle()
'eggs'
>>> turtle.__call__()
'eggs'
>>> turtle.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__.__call__()
'eggs'
>>> turtle.__call__.__call__.__call__.__call__.__call__()
'eggs'
>>>
turtle.__call__.__call__.__call__.__call__.__call__.__call__()
'eggs'
>>>
turtle.__call__.__call__.__call__.__call__.__call__.__call__.__call___.
__call__()
'eggs'
```

子子孙孙无穷匮也。

第三部分 类和协议

- 第 11 章 符合 Python 风格的对象
- 第 12 章 序列的特殊方法
- 第 13 章 接口、协议和抽象基类
- 第 14 章 继承：瑕瑜互见
- 第 15 章 类型提示进阶
- 第 16 章 运算符重载

第 11 章 符合 Python 风格的对象

一个库或框架是否符合 Python 风格，要看它能不能让 Python 程序员以一种简单而自然的方式执行任务。

——Martijn Faassen
Python 和 JavaScript 框架开发者¹

¹摘自 Faassen 的题为“What is Pythonic?”的博客文章。

得益于 Python 数据模型，自定义类型的行为可以像内置类型那样自然。实现如此自然的行为，靠的不是继承，而是**鸭子类型**：只需按照预定行为实现对象所需的方法即可。

前几章分析了很多内置对象的行为。本章会自己定义类，让类的行为跟真正的 Python 对象一样。你在开发应用程序时，不一定要像本章的示例那样实现那么多特殊方法。然而，对库或框架来说，程序员可能希望你定义的类能像 Python 内置的类一样。满足这个预期也算得上是符合“Python 风格”。

本章接续第 1 章，说明如何实现很多 Python 类型中常见的特殊方法。本章包含以下话题：

- 把对象转换成其他类型的内置函数（例如 `repr()`、`bytes()`、`complex()` 等）；
- 通过一个类方法实现备选构造函数；
- 扩展 f 字符串、内置函数 `format()` 和 `str.format()` 方法使用的格式化微语言；
- 实现只读属性；
- 把对象变为可哈希的，以便在集合中及作为 `dict` 的键使用；

- 利用 `__slots__` 节省内存。

本章将开发一个简单的二维欧几里得向量类型 `Vector2d`，在这个过程中涵盖上述全部话题。`Vector2d` 将为第 12 章定义的 N 维向量类奠定基础。

在实现这个类型的中间阶段，本章会讨论如下两个概念：

- 如何以及何时使用 `@classmethod` 装饰器和 `@staticmethod` 装饰器；
- Python 中私有属性和受保护属性的用法、约定和局限。

11.1 本章新增内容

本章引言部分内容换了，第 2 段又增加了几句话，说清“符合 Python 风格”是什么意思。第 1 版直到全书最后才讨论这个问题。

11.6 节有更新，增加了 Python 3.6 引入的 `f` 字符串。但那一节改动不大，因为 `f` 字符串支持的格式化微语言与内置函数 `format()` 和 `str.format()` 方法一样，以前实现的 `__format__` 方法也适用于 `f` 字符串。

本章其他内容基本没变，毕竟自 Python 3.0 以来，特殊方法没什么变化，而且核心理念在 Python 2.2 中就出现了。

从对象表示形式函数开始讲起。

11.2 对象表示形式

每门面向对象语言至少都有一种获取对象字符串表示形式的标准方式。Python 提供了两种方式。

```
repr()
```

以便于开发者理解的方式返回对象的字符串表示形式。Python 控制台或调试器在显示对象时采用这种方式。

```
str()
```

以便于用户理解的方式返回对象的字符串表示形式。使用 `print()` 打印对象时采用这种方式。

第 1 章讲过，在背后支持 `repr()` 和 `str()` 的是特殊方法 `__repr__` 和 `__str__`。

除此之外，还有两个特殊方法（`__bytes__` 和 `__format__`）可为对象提供其他表示形式。`__bytes__` 方法与 `__str__` 方法类似，`bytes()` 函数调用它获取对象的字节序列表示形式。而 `__format__` 方法供 f 字符串、内置函数 `format()` 和 `str.format()` 方法使用，通过调用 `obj.__format__(format_spec)` 以特殊的格式化代码显示对象的字符串表示形式。本章将先讨论 `__bytes__` 方法，随后再讨论 `__format__` 方法。



如果你是 Python 2 用户，那么请记住，在 Python 3 中，`__repr__`、`__str__` 和 `__format__` 都必须返回 Unicode 字符串（`str` 类型）。只有 `__bytes__` 方法应该返回字节序列（`bytes` 类型）。

11.3 再谈向量类

本章将以一个与第 1 章类似的 `Vector2d` 类为例，说明用于生成对象表示形式的众多方法。本节和接下来的几节会逐渐实现这个类。我们期望 `Vector2d` 实例具有的基本行为如示例 11-1 所示。

示例 11-1 `Vector2d` 实例有多种表示形式

```

>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y) ❶
3.0 4.0
>>> x, y = v1 ❷
>>> x, y
(3.0, 4.0)
>>> v1 ❸
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1)) ❹
>>> v1 == v1_clone ❺
True
>>> print(v1) ❻
(3.0, 4.0)
>>> octets = bytes(v1) ❼
>>> octets

b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\
x00\\x10@'
>>> abs(v1) ❽
5.0
>>> bool(v1), bool(Vector2d(0, 0)) ❾
(True, False)

```

❶ Vector2d 实例的分量可以直接通过属性访问（无须调用读值方法）。

❷ Vector2d 实例可以拆包成变量元组。

❸ Vector2d 实例的表示形式模拟源码构建实例的形式。

❹ 这里使用 eval 函数表明 Vector2d 实例的表示形式是对构造函数的准确表述。²

²使用 eval 函数克隆对象是为了说明 repr 方法。使用 copy.copy 函数克隆实例更安全且更快速。

❺ Vector2d 实例支持使用 == 比较，这样便于测试。

❻ print 函数调用 str 函数，对 Vector2d 来说，输出的是一个有序对。

⑦ bytes 函数调用 `__bytes__` 方法，输出实例的二进制表示形式。

⑧ abs 函数调用 `__abs__` 方法，返回 `Vector2d` 实例的模。

⑨ bool 函数调用 `__bool__` 方法，如果 `Vector2d` 实例的模为零，就返回 `False`，否则返回 `True`。

示例 11-1 中的 `Vector2d` 类在 `vector2d_v0.py` 文件中实现（参见示例 11-2）。这段代码基于示例 1-2，支持 `+` 运算和 `*` 运算的方法将在第 16 章实现。另外，为了方便测试，示例 11-2 增加了支持 `==` 运算符的方法。现在，`Vector2d` 用到了几个特殊方法，这些方法提供的操作是 Python 程序员预期设计良好的对象应当提供的。

示例 11-2 `vector2d_v0.py`：目前定义的都是特殊方法

```
from array import array
import math

class Vector2d:
    typecode = 'd' ❶

    def __init__(self, x, y):
        self.x = float(x) ❷
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y)) ❸

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self) ❹

    def __str__(self):
        return str(tuple(self)) ❺

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) + ❻
                bytes(array(self.typecode, self))) ❼

    def __eq__(self, other):
        return tuple(self) == tuple(other) ❽
```

```
def __abs__(self):  
    return math.hypot(self.x, self.y) ⑨  
  
def __bool__(self):  
    return bool(abs(self)) ⑩
```

❶ `typecode` 是类属性，在 `Vector2d` 实例和字节序列之间转换时使用。

❷ 在 `__init__` 方法中把 `x` 和 `y` 转换成浮点数，尽早捕获错误，以防调用 `Vector2d` 构造函数时传入不当参数。

❸ 定义 `__iter__` 方法，把 `Vector2d` 实例变成可迭代对象，这样才能拆包（例如，`x, y = my_vector`）。这个方法的实现方式很简单，直接调用生成器表达式依次产出分量。³

³这一行也可以写成 `yield self.x; yield self.y`。第 17 章将进一步讨论 `__iter__` 特殊方法、生成器表达式和 `yield` 关键字。

❹ `__repr__` 方法使用 `{!r}` 获取各个分量的表示形式，然后插值，构成一个字符串。因为 `Vector2d` 实例是可迭代对象，所以 `*self` 会把 `x` 分量和 `y` 分量提供给 `format` 方法。

❺ 从可迭代的 `Vector2d` 实例中可以轻易得到一个元组，显示为有序对。

❻ 为了生成字节序列，把 `typecode` 转换成字节序列，然后……

❼ ……迭代 `Vector2d` 实例，得到一个数组，再把数组转换成字节序列。

❽ 为了快速比较所有分量，把运算对象转换成元组。对 `Vector2d` 实例来说，虽然可以这样做，但仍有问题。参见下面的警告栏。

❾ 模是 `x` 分量和 `y` 分量构成的直角三角形的斜边长。

⑩ `__bool__` 方法使用 `abs(self)` 计算模，然后把结果转换成布尔值，因此，`0.0` 是 `False`，非零值是 `True`。



示例 11-2 中的 `__eq__` 方法，在两个运算对象都是 `Vector2d` 实例时没有问题，不过拿 `Vector2d` 实例与其他具有相同数值的可迭代对象相比，结果也是 `True`（例如，`Vector(3, 4) == [3, 4]`）。这个行为既可以被视为特性，也可以被视为 bug。第 16 章在讲到运算符重载时将进一步讨论。

我们已经定义了很多基本方法，但是显然少了一个操作：使用 `bytes()` 函数生成的二进制表示形式重建 `Vector2d` 实例。

11.4 备选构造函数

现在可以把 `Vector2d` 实例转换成字节序列了。同理，我们也希望能从字节序列构建 `Vector2d` 实例。在标准库中探索一番之后，我们发现 `array.array` 有个类方法 `.frombytes`（2.10.1 节介绍过）正好符合需求。下面在 `vector2d_v1.py` 文件中为 `Vector2d` 定义一个同名类方法，如示例 11-3 所示。

示例 11-3 `vector2d_v1.py` 的一部分：这段代码只列出了 `frombytes` 类方法，要添加到 `vector2d_v0.py`（参见示例 11-2）定义的 `Vector2d` 类中

```
@classmethod ❶
def frombytes(cls, octets): ❷
    typecode = chr(octets[0]) ❸
    memv = memoryview(octets[1:]).cast(typecode) ❹
    return cls(*memv) ❺
```

❶ `classmethod` 装饰的方法可直接在类上调用。

- ❷ 第一个参数不是 `self`，而是类自身（习惯命名为 `cls`）。
- ❸ 从第一字节中读取 `typecode`。
- ❹ 使用传入的 `octets` 字节序列创建一个 `memoryview`，然后使用 `typecode` 进行转换。⁴

⁴2.10.2 节简单介绍过 `memoryview`，说明了它的 `.cast` 方法。

- ❺ 拆包转换后的 `memoryview`，得到构造函数所需的一对参数。

我们用的 `classmethod` 装饰器是 Python 特有的，下面来讲解一下。

11.5 `classmethod` 与 `staticmethod`

Python 官方教程没有提到 `classmethod` 装饰器，也没有提到 `staticmethod`。学过 Java 面向对象编程的人可能觉得奇怪，为什么 Python 提供两个这样的装饰器，而不是只提供一个？

先来看 `classmethod`。示例 11-3 展示了它的用法：定义操作类而不是操作实例的方法。由于 `classmethod` 改变了调用方法的方式，因此接收的第一个参数是类本身，而不是实例。`classmethod` 最常见的用途是定义备选构造函数，例如示例 11-3 中的 `frombytes`。注意，`frombytes` 的最后一行使用 `cls` 参数构建了一个新实例，即 `cls(*memv)`。

相比之下，`staticmethod` 装饰器也会改变方法的调用方式，使其接收的第一个参数没什么特殊的。其实，静态方法就是普通的函数，只是碰巧位于类的定义体中，而不是在模块层定义。示例 11-4 对 `classmethod` 和 `staticmethod` 的行为做了对比。

示例 11-4 比较 `classmethod` 和 `staticmethod` 的行为

```

>>> class Demo:
...     @classmethod
...     def klassmeth(*args):
...         return args ❶
...     @staticmethod
...     def statmeth(*args):
...         return args ❷
...
>>> Demo.klassmeth() ❸
(<class '__main__.Demo'>,)
>>> Demo.klassmeth('spam')
(<class '__main__.Demo'>, 'spam')
>>> Demo.statmeth() ❹
()
>>> Demo.statmeth('spam')
('spam',)

```

❶ `klassmeth` 返回全部位置参数。

❷ `statmeth` 也返回全部位置参数。

❸ 不管怎样调用 `Demo.klassmeth`，它的第一个参数始终是 `Demo` 类。

❹ `Demo.statmeth` 的行为与普通的函数一样。



`classmethod` 装饰器非常有用，但是我从未见过不得不使用 `staticmethod` 的情况。有些函数即使不直接处理类，也与类联系紧密，因此你会想把函数与类放在一起定义。对于这种情况，在类的前面或后面定义函数，保持二者在同一个模块中基本上就可以了。⁵

⁵ 本书的技术审校之一 Leonardo Rochaël 不同意我对 `staticmethod` 的贬低，作为反驳，他推荐阅读 Julien Danjou 写的一篇题为“The Definitive Guide on How to Use Static, Class or Abstract Methods in Python”的博客文章。Danjou 的这篇文章写得很好，我推荐阅读。但是，我对 `staticmethod` 的观点依然不变。请读者自辨。

现在，我们对 `classmethod` 的作用已经有所了解（而且知道 `staticmethod` 不是特别有用），下面继续讨论对象的表示形式，说明如何支持格式化输出。

11.6 格式化显示

`f` 字符串、内置函数 `format()` 和 `str.format()` 方法会把各种类型的格式化方式委托给相应的 `.__format__(format_spec)` 方法。`format_spec` 是格式说明符，它是：

- `format(my_obj, format_spec)` 的第二个参数；
- `{}` 内代换字段中冒号后面的部分，或者 `fmt.str.format()` 中的 `fmt`。

请看以下示例。

```
>>> brl = 1 / 4.82 # 巴西雷亚尔兑换美元的汇率
>>> brl
0.20746887966804978
>>> format(brl, '0.4f') ❶
'0.2075'
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl) ❷
'1 BRL = 0.21 USD'
>>> f'1 USD = {1 / brl:0.2f} BRL' ❸
'1 USD = 4.82 BRL'
```

❶ 格式说明符是 `'0.4f'`。

❷ 格式说明符是 `'0.2f'`。代换字段中的 `rate` 部分不属于格式说明符，只用于决定把 `.format()` 的哪个关键字参数传给代换字段。

❸ 同样，格式说明符是 `'0.2f'`。`1 / brl` 表达式不属于格式说明符。

第 2 和第 3 个标号指出了一个重要知识点：像 `{0.mass:5.3e}` 这样的格式字符串其实包含两部分，冒号左边的 `0.mass` 在代换字段句法中是字段名，在 `f` 字符串中可以是任意表达式；冒号右边的 `5.3e` 是格式说明符。格式说明符使用的表示法叫格式规范微语言 (Format Specification Mini-Language) 。



如果你对 `f` 字符串、`format()` 和 `str.format()` 感到陌生，根据我的教学经验，最好先学内置函数 `format()`，因为它只使用格式规范微语言。学会这些表示法之后，再阅读“Formatted String Literals”和“Format String Syntax”，学习 `f` 字符串和 `str.format()` 方法使用的代换字段表示法 `{:}`（包括转换标志 `!s`、`!r` 和 `!a`）。`f` 字符串出现之后，`str.format()` 并没有被淘汰：`f` 字符串适用于多数情况，不过有时候格式化字符串不在渲染的地方而是在别处指定。

格式规范微语言为一些内置类型提供了专用的表示代码。例如，`b` 和 `x` 分别表示二进制和十六进制的 `int` 类型，`f` 表示小数形式的 `float` 类型，而 `%` 表示百分数形式。

```
>>> format(42, 'b')
'101010'
>>> format(2 / 3, '.1%')
'66.7%'
```

格式规范微语言是可扩展的，各个类可以自行决定如何解释 `format_spec` 参数。例如，`datetime` 模块中的类的 `__format__` 方法所使用的格式代码与 `strftime()` 函数一样。下面是内置函数 `format()` 和 `str.format()` 方法的几个示例。

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now, '%H:%M:%S')
'18:49:05'
>>> "It's now {:%I:%M %p}".format(now)
"It's now 06:49 PM"
```

如果一个类没有定义 `__format__`，那么该方法就会从 `object` 继承，并返回 `str(my_object)`。由于 `Vector2d` 类有 `__str__` 方法，因此可以这样做。

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
```

然而，如果传入格式说明符，则 `object.__format__` 会抛出 `TypeError`。

```
>>> format(v1, '.3f')
Traceback (most recent call last):
...
TypeError: non-empty format string passed to object.__format__
```

我们将实现自己的格式微语言来解决这个问题。首先，假设用户提供的格式说明符是用于格式化向量中各个 `float` 分量的。我们想达到的效果如下所示。

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

实现这种输出的 `__format__` 方法如示例 11-5 所示。

示例 11-5 `Vector2d.__format__` 方法，第 1 版

```
# 在Vector2d类中定义

def __format__(self, fmt_spec=''):
    components = (format(c, fmt_spec) for c in self) ❶
    return '({}, {})'.format(*components) ❷
```

❶ 使用内置函数 `format` 把 `fmt_spec` 应用到向量的各个分量上，构建一个可迭代的格式化字符串。

❷ 把格式化字符串代入公式 (x, y) 中。

然后，在微语言中添加一个自定义的格式代码：如果格式说明符以 'p' 结尾，就在极坐标中显示向量，即 $\langle r, \theta \rangle$ ，其中 r 是模， θ （西塔）是弧度。格式说明符中的其他部分（'p' 前面）像往常那样解释。



为自定义的格式代码选择字母时，我会避免使用其他类型用过的字母。根据格式规范微语言文档，整数使用的代码是 'bcdoxXn'，浮点数使用的代码是 'eEfFgGn%'，字符串使用的代码是 's'。因此，我为极坐标选的代码是 'p'。因为各个类会使用自己的方式解释格式代码，所以在自定义的格式代码中重复使用代码字母不会出错，但是可能会让用户感到困惑。

为了生成极坐标，我们已经定义了计算模的 `__abs__` 方法，但还要定义一个简单的 `angle` 方法，使用 `math.atan2()` 函数计算角度。`angle` 方法的代码如下所示。

```
# 在Vector2d类中定义

def angle(self):
    return math.atan2(self.y, self.x)
```

现在可以增强 `__format__` 方法，计算极坐标了，如示例 11-6 所示。

示例 11-6 `Vector2d.__format__` 方法，第 2 版，现在能计算极坐标了

```
def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'): ❶
        fmt_spec = fmt_spec[:-1] ❷
```

```
        coords = (abs(self), self.angle()) ❸
        outer_fmt = '<{}, {}>' ❹
    else:
        coords = self ❺
        outer_fmt = '({}, {})' ❻
    components = (format(c, fmt_spec) for c in coords) ❼
    return outer_fmt.format(*components) ❽
```

- ❶ 如果格式代码以 'p' 结尾，就使用极坐标。
- ❷ 从 `fmt_spec` 中删除 'p' 后缀。
- ❸ 构建一个元组来表示极坐标：(`magnitude`, `angle`)。
- ❹ 把外层格式设为一对尖括号。
- ❺ 如果不以 'p' 结尾，则使用 `self` 的 `x` 分量和 `y` 分量构建直角坐标。
- ❻ 把外层格式设为一对圆括号。
- ❼ 使用各个分量生成可迭代的对象，构成格式化字符串。
- ❽ 把格式化字符串代入外层格式。

运行示例 11-6，结果如下所示。

```
>>> format(Vector2d(1, 1), 'p')
'<1.4142135623730951, 0.7853981633974483>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

如本节所示，为用户定义的类型扩展格式规范微语言并不难。

下面换一个不仅仅事关对象外在表现的话题。我们将把 `Vector2d` 变成可哈希的，以便构建向量集合，或者把向量当作 `dict` 的键使用。

11.7 可哈希的 Vector2d

按照定义，目前 Vector2d 实例不可哈希，因此不能放入集合中。

```
>>> v1 = Vector2d(3, 4)
>>> hash(v1)
Traceback (most recent call last):
...
TypeError: unhashable type: 'Vector2d'
>>> set([v1])
Traceback (most recent call last):
...
TypeError: unhashable type: 'Vector2d'
```

为了把 Vector2d 实例变成可哈希的，必须实现 `__hash__` 方法（还需要 `__eq__` 方法，前面已经实现了）。此外，还要让向量实例不可变（详见 3.4.1 节）。

目前，可以为分量赋新值（例如 `v1.x = 7`），Vector2d 类的代码并不阻止这么做。而我们想要的行为如下所示。

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 7
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

为此，要把 `x` 分量和 `y` 分量设为只读特性，如示例 11-7 所示。

示例 11-7 `vector2d_v3.py`：这里只给出了让 Vector2d 不可变的代码，完整的代码清单见示例 11-11

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x) ❶
        self.__y = float(y)
```



```

@property ❷
def x(self): ❸
    return self.__x ❹

@property ❺
def y(self):
    return self.__y

def __iter__(self):
    return (i for i in (self.x, self.y)) ❻

# 其他方法可以参见前面的代码清单

```

❶ 使用两个前导下划线（尾部没有下划线或有一个下划线），把属性标记为私有的。⁶

⁶私有属性的优缺点见 11.10 节。

❷ @property 装饰器把读值方法标记为特性（property）。

❸ 读值方法与公开属性同名，都是 `x`。

❹ 直接返回 `self.x`。

❺ 以同样的方式处理 `y` 特性。

❻ 需要读取 `x` 分量和 `y` 分量的方法可以保持不变，仍然通过 `self.x` 和 `self.y` 读取公开特性，而不必读取私有属性，因此该代码清单省略了这个类余下的代码。



`Vector.x` 和 `Vector.y` 是只读特性。第 22 章将讨论读写特性，届时会深入说明 @property 装饰器。

现在，向量不会被意外修改，有了一定的安全性，接下来可以实现 `__hash__` 方法了。这个方法应该返回一个 `int` 值，理想情况下还要考虑对象属性的哈希值（`__eq__` 方法也是如此），因为相等的对象

应该具有相同的哈希值。特殊方法 `__hash__` 的文档建议根据元组的分量计算哈希值，如示例 11-8 所示。

示例 11-8 `vector2d_v3.py`: 实现 `__hash__` 方法

```
# 在Vector2d类中定义

def __hash__(self):
    return hash((self.x, self.y))
```

实现 `__hash__` 方法之后，向量就变成可哈希的了。

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
(1079245023883434373, 1994163070182233067)
>>> {v1, v2}
{Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```



为了创建可哈希的类型，不一定要实现特性，也不一定要保护实例属性，正确实现 `__hash__` 方法和 `__eq__` 方法即可。但是，可哈希对象的值绝不应该变化，因此我们借机提到了只读特性。

如果你定义的类型有标量数值，那么可能还要实现 `__int__` 方法和 `__float__` 方法（分别被 `int()` 构造函数和 `float()` 构造函数调用），以便在某些情况下强制转换类型。此外，还有用于支持内置构造函数 `complex()` 的 `__complex__` 方法。`Vector2d` 或许应该提供 `__complex__` 方法，这就给读者留作练习吧。

11.8 支持位置模式匹配

目前，`Vector2d` 实例兼容关键字类模式（参见 5.8.2 节）。

在示例 11-9 中，所有关键字模式都能按预期匹配。

示例 11-9 匹配 Vector2d 对象的关键字模式（需要在 Python 3.10 中操作）

```
def keyword_pattern_demo(v: Vector2d) -> None:
    match v:
        case Vector2d(x=0, y=0):
            print(f'{v!r} is null')
        case Vector2d(x=0):
            print(f'{v!r} is vertical')
        case Vector2d(y=0):
            print(f'{v!r} is horizontal')
        case Vector2d(x=x, y=y) if x==y:
            print(f'{v!r} is diagonal')
        case _:
            print(f'{v!r} is awesome')
```

然而，如果使用如下位置模式：

```
case Vector2d(_, 0):
    print(f'{v!r} is horizontal')
```

则会得到如下结果。

```
TypeError: Vector2d() accepts 0 positional sub-patterns (1 given)
```

为了让 Vector2d 支持位置模式，需要添加一个名为 `__match_args__` 的类属性，按照在位置模式匹配中的使用顺序列出实例属性。

```
class Vector2d:
    __match_args__ = ('x', 'y')

    # ....., 等等
```

现在，编写匹配 Vector2d 对象的模式时可以少敲几次键盘了，如示例 11-10 所示。

示例 11-10 匹配 Vector2d 对象的位置模式（需要在 Python 3.10 中操作）

```
def positional_pattern_demo(v: Vector2d) -> None:
    match v:
        case Vector2d(0, 0):
            print(f'{v!r} is null')
        case Vector2d(0):
            print(f'{v!r} is vertical')
        case Vector2d(_, 0):
            print(f'{v!r} is horizontal')
        case Vector2d(x, y) if x==y:
            print(f'{v!r} is diagonal')
        case _:
            print(f'{v!r} is awesome')
```

`__match_args__` 类属性不一定要把所有公开的实例属性都列出来。如果一个类的 `__init__` 方法可能有全都赋值给实例属性的必需参数和可选的参数，那么 `__match_args__` 应当列出必需的参数，而不必列出可选的参数。

现在，暂停一下，看看 `Vector2d` 类目前的代码。

11.9 第 3 版 `Vector2d` 的完整代码

前面一直在定义 `Vector2d` 类，不过每次只给出了部分片段。示例 11-11 是整理后的完整代码清单，保存在 `vector2d_v3.py` 文件中，包含我在开发时编写的 `doctest`。

示例 11-11 `vector2d_v3.py`：完整版

```
"""
一个二维向量类

>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)
3.0 4.0
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
```

```

True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets

b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector2d(0, 0))
(True, False)

```

测试类方法``.frombytes()``::

```

>>> v1_clone = Vector2d.frombytes(bytes(v1))
>>> v1_clone
Vector2d(3.0, 4.0)
>>> v1 == v1_clone
True

```

使用笛卡儿坐标测试``.format()``::

```

>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'

```

测试``.angle``方法::

```

>>> Vector2d(0, 0).angle()
0.0
>>> Vector2d(1, 0).angle()
0.0
>>> epsilon = 10**-8
>>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
True
>>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
True

```

使用极坐标测试``.format()``::

```

>>> format(Vector2d(1, 1), 'p') # doctest:+ELLIPSIS

```

```
'<1.414213..., 0.785398...>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

测试只读特性`x`和`y`::

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 123
Traceback (most recent call last):
...
AttributeError: can't set attribute 'x'
```

测试哈希::

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> len({v1, v2})
2
```

"""

```
from array import array
import math
```

```
class Vector2d:
    __match_args__ = ('x', 'y')

    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    def __repr__(self):
```

```

class_name = type(self).__name__
return '{}({!r}, {!r})'.format(class_name, *self)

def __str__(self):
    return str(tuple(self))

def __bytes__(self):
    return (bytes([ord(self.typecode)]) +
            bytes(array(self.typecode, self)))

def __eq__(self, other):
    return tuple(self) == tuple(other)

def __hash__(self):
    return hash((self.x, self.y))

def __abs__(self):
    return math.hypot(self.x, self.y)

def __bool__(self):
    return bool(abs(self))

def angle(self):
    return math.atan2(self.y, self.x)

def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'):
        fmt_spec = fmt_spec[:-1]
        coords = (abs(self), self.angle())
        outer_fmt = '<{}, {}>'
    else:
        coords = self
        outer_fmt = '({}, {})'
    components = (format(c, fmt_spec) for c in coords)
    return outer_fmt.format(*components)

@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(*memv)

```

小结一下，前两节说明了一些特殊方法，要想得到功能完善的对象，这些方法可能是必备的。



当应用程序真正需要这些特殊方法时才应实现它们。终端用户并不关心应用程序中的对象是否符合“Python 风格”。

另外，如果你的类是供其他 Python 程序员使用的库的一部分，那么你肯定猜不到程序员会对你的对象做什么，他们或许更希望你的代码符合“Python 风格”。

示例 11-11 中的 `Vector2d` 类只是为了教学，我们为它定义了许多与对象表示形式有关的特殊方法。不是每个用户定义的类都要这样做。

11.10 节暂时不继续定义 `Vector2d` 类了，我们将讨论 Python 对私有属性（带两个下划线前缀的属性，例如 `self.__x`）的设计方式及其缺点。

11.10 Python 私有属性和“受保护”的属性

Python 不能像 Java 那样使用 `private` 修饰符创建私有属性，但是它有一个简单的机制，能避免子类意外覆盖“私有”属性。

举个例子。有人编写了一个名为 `Dog` 的类，内部用到了 `mood` 实例属性，但是没有将其开放。现在，你创建了 `Dog` 类的子类 `Beagle`。如果你在毫不知情的情况下又创建了名为 `mood` 的实例属性，那么在继承的方法中就会把 `Dog` 类的 `mood` 属性覆盖。这是难以调试的问题。

为了避免这种情况，如果以 `__mood` 的形式（两个前导下划线，尾部没有或最多有一个下划线）命名实例属性，那么 Python 就会把属性名存入实例属性 `__dict__` 中，而且会在前面加上一个下划线和类名。因此，对 `Dog` 类来说，`__mood` 会变成 `_Dog__mood`；对 `Beagle` 类来说，`__mood` 会变成 `_Beagle__mood`。这个语言功能叫**名称改写**（name mangling）。

示例 11-12 以示例 11-7 中定义的 `Vector2d` 类为例对名称改写进行了说明。

示例 11-12 私有属性的名称会被“改写”，在前面加上 _ 和类名

```
>>> v1 = Vector2d(3, 4)
>>> v1.__dict__
{'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
>>> v1._Vector2d__x
3.0
```

名称改写是一种安全措施，不能保证万无一失：它的目的是避免意外访问，不能防止故意做错事。图 11-1 也是一种保护装置。

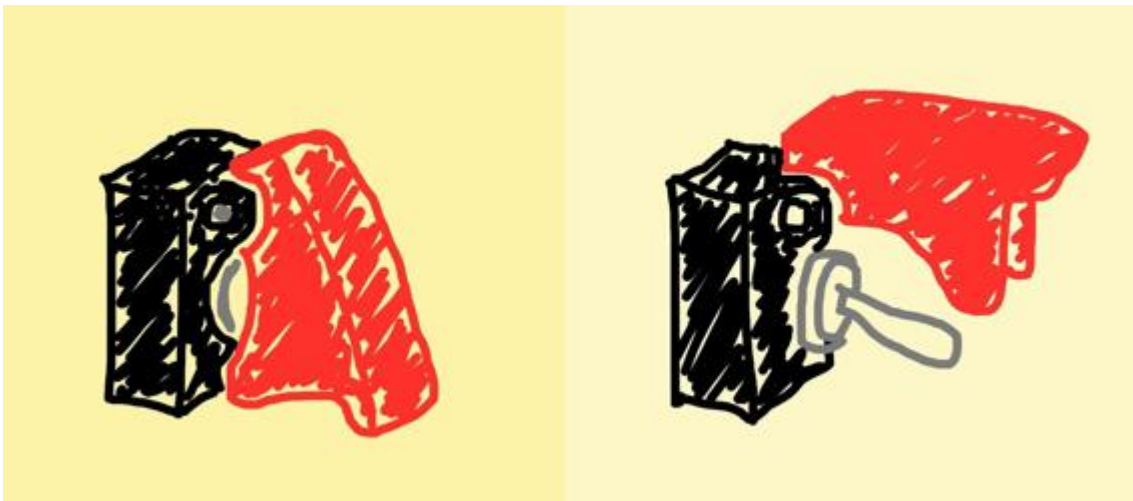


图 11-1：把手上的盖子是一种保护装置，而不是安全装置：它能避免意外触动把手，但是不能防止有意转动

如示例 11-12 的最后一行所示，只要知道改写私有属性名称的机制，任何人都能直接读取私有属性——这实际上对调试和序列化很有用。此外，编写 `v1._Vector2d__x = 7` 这样的代码也能直接为 `Vector2d` 实例的私有分量赋值。但是，如果真在生产环境中这么做了，那么出问题时可别抱怨。

不是所有 Python 程序员都喜欢名称改写功能，也不是所有人都喜欢 `self.__x` 这种头重脚轻的名称。有些人不喜欢这种句法，他们约定使用一个下划线前缀编写“受保护”的属性（例如 `self._x`）。批评使用两个下划线这种改写机制的人认为，应该使用命名约定来避免意外覆盖属性。pip、virtualenv 等项目的开发者 Ian Bicking 指出：

千万千万不要使用两个前导下划线，这是很烦人的自私行为。如果担心名称冲突，则应该明确使用一种名称改写方式（例如 `_MyThing_blahblah`）。这其实与使用双下划线一样，不过自己定的规则比双下划线易于理解。⁷

⁷摘自“Paste Style Guide”。

Python 解释器不会对使用单下划线的属性名做特殊处理，不过这是很多 Python 程序员严格遵守的约定，他们不会在类的外部访问这种属性。⁸ 遵守使用一个下划线标记对象的私有属性很容易，就像遵守使用全大写字母编写常量一样。

⁸不过，在模块中，如果顶层名称使用一个前导下划线，那么的确会有影响：对 `from mymod import *` 来说，`mymod` 中前缀为一个下划线的名称不会被导入。然而，依旧可以使用 `from mymod import _privatefunc` 将其导入。详见 Python 官方教程 6.1 节“More on Modules”。

Python 文档的某些角落把使用一个下划线前缀标记的属性称为“受保护”的属性。⁹ 使用 `self._x` 这种形式的“保护”属性的做法很常见，但很少有人把这种属性叫作“受保护”的属性。有些人甚至将其称为“私有”属性。

⁹`gettext` 模块文档中就有一个例子。

总之，`Vector2d` 的分量都是“私有”的，而且 `Vector2d` 实例都是“不可变”的。我用了两对引号，因为并不能真正实现私有和不可变。¹⁰

¹⁰如果这种说法令你感到沮丧，让你觉得在这方面 Python 应该向 Java 看齐，那么就不要阅读本章的“杂谈”了。我在“杂谈”中探讨了 Java `private` 修饰符的相对优势。

下面继续定义 `Vector2d` 类。11.11 节将讨论一个特殊的属性（不是方法），它会影响对象的内部存储，对内存用量可能也有重大影响，但是对对象的公开接口没什么影响。这个属性是 `__slots__`。

11.11 使用 `__slots__` 节省空间

默认情况下，Python 把各个实例的属性存储在一个名为 `__dict__` 的字典中。3.9 节讲过，字典消耗的内存很多——即使有一些优化措施。但是，如果定义一个名为 `__slots__` 的类属性，以序列的形式存储属性名称，那么 Python 将使用其他模型存储实例属性：

`__slots__` 中的属性名称存储在一个隐藏的引用数组中，消耗的内存比字典少。下面通过几个简单的示例说明一下，先看示例 11-13。

示例 11-13 使用 `__slots__` 的 `Pixel` 类

```
>>> class Pixel:
...     __slots__ = ('x', 'y') ❶
...
>>> p = Pixel() ❷
>>> p.__dict__ ❸
Traceback (most recent call last):
...
AttributeError: 'Pixel' object has no attribute '__dict__'
>>> p.x = 10 ❹
>>> p.y = 20
>>> p.color = 'red' ❺
Traceback (most recent call last):
...
AttributeError: 'Pixel' object has no attribute 'color'
```

❶ `__slots__` 必须在定义类时声明，之后再添加或修改均无效。属性名称可以存储在一个元组或列表中，不过我喜欢使用元组，因为这样可以明确表明 `__slots__` 无法修改。

❷ 创建一个 `Pixel` 实例，因为 `__slots__` 的效果要通过实例体现。

❸ 第一个效果：`Pixel` 实例没有 `__dict__` 属性。

❹ 正常设定 `p.x` 属性和 `p.y` 属性。

❸ 第二个效果：设定不在 `__slots__` 中的属性抛出 `AttributeError`。

目前还没有什么难以理解的。现在，在示例 11-14 中定义一个 `Pixel` 的子类，看看 `__slots__` 违反直觉的一面。

示例 11-14 `OpenPixel` 是 `Pixel` 的子类

```
>>> class OpenPixel(Pixel): ❶
...     pass
...
>>> op = OpenPixel()
>>> op.__dict__ ❷
{}
>>> op.x = 8 ❸
>>> op.__dict__ ❹
{}
>>> op.x ❺
8
>>> op.color = 'green' ❻
>>> op.__dict__ ❼
{'color': 'green'}
```

- ❶ `OpenPixel` 自身没有声明任何属性。
- ❷ 奇怪的事情发生了，`OpenPixel` 实例有 `__dict__` 属性。
- ❸ 即使设定属性 `x`（在基类 `Pixel` 的 `__slots__` 属性中）.....
- ❹ 也不存入实例的 `__dict__` 属性中.....
- ❺ 而是存入实例的一个隐藏的引用数组中。
- ❻ 设定不在 `__slots__` 中的属性.....
- ❼ 存入实例的 `__dict__` 属性中。

示例 11-14 表明，子类只继承 `__slots__` 的部分效果。为了确保子类的实例也没有 `__dict__` 属性，必须在子类中再次声明 `__slots__` 属性。

如果在子类中声明 `__slots__ = ()`（一个空元组），则子类的实例将没有 `__dict__` 属性，而且只接受基类的 `__slots__` 属性列出的属性名称。

如果子类需要额外属性，则在子类的 `__slots__` 属性中列出来，如示例 11-15 所示。

示例 11-15 ColorPixel 也是 Pixel 的子类

```
>>> class ColorPixel(Pixel):
...     __slots__ = ('color',) ❶
>>> cp = ColorPixel()
>>> cp.__dict__ ❷
Traceback (most recent call last):
...
AttributeError: 'ColorPixel' object has no attribute '__dict__'
>>> cp.x = 2
>>> cp.color = 'blue' ❸
>>> cp.flavor = 'banana'
Traceback (most recent call last):
...
AttributeError: 'ColorPixel' object has no attribute 'flavor'
```

❶ 其实，超类的 `__slots__` 属性会被添加到当前类的 `__slots__` 属性中。别忘了，只有一项的元组，因此在那一项后面要加上一个逗号。

❷ ColorPixel 实例没有 `__dict__` 属性。

❸ 可以设定在当前类和超类的 `__slots__` 中声明的属性，其他属性则不能设定。

然而，“节省的内存也可能被再次吃掉”：如果把 `'__dict__'` 这个名称添加到 `__slots__` 列表中，则实例会在各个实例独有的引用数组

中存储 `__slots__` 中的名称，不过也支持动态创建属性，存储在常规的 `__dict__` 中。如果想使用 `@cached_property` 装饰器，就要这么做（详见 22.3.5 节）。

当然，把 `'__dict__'` 添加到 `__slots__` 中可能完全违背了初衷，这取决于各个实例的静态属性和动态属性的数量及其用法。粗心的优化甚至比提早优化还糟糕，往往得不偿失。

此外，还有一个实例属性可能需要注意，即 `__weakref__`。为了让对象支持弱引用（6.6 节简单提过），必须有这个属性。用户定义类默认就有 `__weakref__` 属性。然而，如果类中定义了 `__slots__`，而且想把该类的实例作为弱引用的目标，则必须把 `'__weakref__'` 添加到 `__slots__` 中。

下面来看为 `Vector2d` 添加 `__slots__` 的效果。

11.11.1 简单衡量 `__slots__` 节省的内存

示例 11-16 在 `Vector2d` 中实现了 `__slots__`。

示例 11-16 `vector2d_v3_slots.py`: 只为 `Vector2d` 类增加 `__slots__` 属性

```
class Vector2d:
    __match_args__ = ('x', 'y') ❶
    __slots__ = ('__x', '__y') ❷

    typecode = 'd'
    # 方法与前面的版本一样
```

❶ `__match_args__` 列出位置模式匹配可用的公开属性名称。

❷ 而 `__slots__` 列出的是实例属性名称。这里列出的是私有属性。

为了衡量节省了多少内存，我编写了 `mem_test.py` 脚本。这个脚本在命令行中运行，参数是两版 `Vector2d` 类所在的模块名。我们使用列表推导式构建一个列表，存储 10 000 000 个 `Vector2d` 实例。第一次运行时，使用 `vector2d_v3.Vector2d`（来自示例 11-7），第二次运行时，使用示例 11-16 中带 `__slots__` 的版本，如示例 11-17 所示。

示例 11-17 `mem_test.py` 使用指定模块中定义的 `Vector2d` 类创建 10 000 000 个实例

```
$ time python3 mem_test.py vector2d_v3
Selected Vector2d type: vector2d_v3.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      6,983,680
  Final RAM usage:    1,666,535,424

real 0m11.990s
user 0m10.861s
sys 0m0.978s
$ time python3 mem_test.py vector2d_v3_slots
Selected Vector2d type: vector2d_v3_slots.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      6,995,968
  Final RAM usage:      577,839,104

real 0m8.381s
user 0m8.006s
sys 0m0.352s
```

如示例 11-17 所示，当 10 000 000 个 `Vector2d` 实例使用 `__dict__` 属性时，RAM 用量高达 1.55 GiB，而当 `Vector2d` 有 `__slots__` 属性之后，RAM 用量降到了 551 MiB。而且，带 `__slots__` 属性的版本运行速度也更快。这个测试中使用的 `mem_test.py` 脚本其实只是加载一个模块、检查内存用量和格式化结果，源码放在 `fluentpython/example-code-2e` 中。



处理数百万个具有数值数据的对象其实应该使用 NumPy 数组（参见 2.10.3 节）。NumPy 数组不仅节约内存，还高度优化

了数值处理函数（很多函数一次性处理整个数组）。我设计 `Vector2d` 类的目的只是为了讨论特殊方法，因为不想随意举个含糊不清的示例。

11.11.2 总结 `__slots__` 的问题

如果使用得当，则类属性 `__slots__` 能显著节省内存，不过有几个问题需要注意。

- 每个子类都要重新声明 `__slots__` 属性，以防止子类的实例有 `__dict__` 属性。
- 实例只能拥有 `__slots__` 列出的属性，除非把 `'__dict__'` 加入 `__slots__` 中（但是这样做就失去了节省内存的功效）。
- 有 `__slots__` 的类不能使用 `@cached_property` 装饰器，除非把 `'__dict__'` 加入 `__slots__` 中。
- 如果不把 `'__weakref__'` 加入 `__slots__` 中，那么实例就不能作为弱引用的目标。

本章最后一个话题讨论如何在实例和子类中覆盖类属性。

11.12 覆盖类属性

Python 有一个很独特的功能：类属性可为实例属性提供默认值。`Vector2d` 中有一个名为 `typecode` 的类属性。`__bytes__` 方法两次用到了这个属性，而且都故意使用 `self.typecode` 读取它的值。因为 `Vector2d` 实例本身没有 `typecode` 属性，所以 `self.typecode` 默认获取的是 `Vector2d.typecode` 类属性的值。

但是，如果为不存在的实例属性赋值，那么将创建一个新实例属性。假如为 `typecode` 实例属性赋值，那么同名类属性将不受影响。然而，一旦这样做，实例读取的 `self.typecode` 是实例属性

typecode, 也就是把同名类属性遮盖了。借助这个功能, 可以为各个实例的 typecode 属性定制不同的值。

Vector2d.typecode 属性的默认值是 'd', 即转换成字节序列时使用 8 字节双精度浮点数表示向量的各个分量。如果在转换之前把 Vector2d 实例的 typecode 属性设为 'f', 那么将使用 4 字节单精度浮点数表示各个分量, 如示例 11-18 所示。

示例 11-18 设定原本从类中继承的 typecode 属性, 自定义一个实例属性

```
>>> from vector2d_v3 import Vector2d
>>> v1 = Vector2d(1.1, 2.2)
>>> dumpd = bytes(v1)
>>> dumpd
b'd\x9a\x99\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x99\x01@'
>>> len(dumpd) ❶
17
>>> v1.typecode = 'f' ❷
>>> dumpf = bytes(v1)
>>> dumpf
b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
>>> len(dumpf) ❸
9
>>> Vector2d.typecode ❹
'd'
```

- ❶ 默认的字节序列长度为 17 字节。
- ❷ 把 v1 实例的 typecode 属性设为 'f'。
- ❸ 现在得到的字节序列是 9 字节长。
- ❹ Vector2d.typecode 属性的值不变, 只有 v1 实例的 typecode 属性使用 'f'。



这里在讨论如何添加自定义的实例属性, 因此示例 11-18 使用的是示例 11-11 中不带 `__slots__` 属性的 Vector2d 类。

现在你应该知道为什么要在得到的字节序列前面加上 `typecode` 的值了：为了支持不同的格式。

如果想修改类属性的值，那么必须直接在类上修改，不能通过实例修改。如果想修改所有实例（自身没有 `typecode` 属性）的 `typecode` 属性的默认值，则可以像下面这样做。

```
>>> Vector2d.typecode = 'f'
```

然而，有一种修改方法更符合 Python 风格，而且效果持久，也更有针对性。类属性是公开的，会被子类继承，于是我们经常会创建一个子类，只用于定制类的数据属性。Django 基于类的视图就大量使用了这种技术。具体做法如示例 11-19 所示。

示例 11-19 `ShortVector2d` 是 `Vector2d` 的子类，只覆盖 `typecode` 的默认值

```
>>> from vector2d_v3 import Vector2d
>>> class ShortVector2d(Vector2d): ❶
...     typecode = 'f'
...
>>> sv = ShortVector2d(1/11, 1/27) ❷
>>> sv
ShortVector2d(0.09090909090909091, 0.037037037037037035) ❸
>>> len(bytes(sv)) ❹
9
```

❶ 把 `ShortVector2d` 定义为 `Vector2d` 的子类，只覆盖 `typecode` 类属性。

❷ 为了演示，创建一个 `ShortVector2d` 实例，即 `sv`。

❸ 查看 `sv` 的表示形式。

❹ 确认得到的字节序列长度为 9 字节，而不是之前的 17 字节。

这也说明了在 `Vector2d.__repr__` 方法中为什么没有硬编码 `class_name` 的值，而是使用 `type(self).__name__` 获取，如下所示。

```
# 在Vector2d类中定义:

def __repr__(self):
    class_name = type(self).__name__
    return '{}({!r}, {!r})'.format(class_name, *self)
```

如果硬编码 `class_name` 的值，那么仅为了修改 `class_name` 的值，`Vector2d` 的子类（例如 `ShortVector2d`）就要覆盖 `__repr__` 方法。从实例的类型中读取类名，`__repr__` 方法可以放心继承。

至此，本章通过一个简单的类说明了如何利用数据模型处理 Python 的其他功能，包括提供不同的对象表示形式、实现自定义的格式代码、公开只读属性，以及通过 `hash()` 函数支持集合和映射。

11.13 本章小结

本章的目的是说明如何使用特殊方法和约定的结构来定义行为良好且符合 Python 风格的类。

`vector2d_v3.py`（参见示例 11-11）比 `vector2d_v0.py`（参见示例 11-2）更符合 Python 风格吗？`vector2d_v3.py` 中的 `Vector2d` 类用到的 Python 功能肯定更多，但是 `Vector2d` 类的第 1 版和最后一版相比哪个更符合 Python 风格要看其使用上下文。Tim Peter 写的《Python 之禅》说道：

简洁胜于复杂。

符合 Python 风格的对象应该正好符合所需，而不是堆砌语言功能。开发应用程序时，应该集中精力满足终端用户的需求，仅此而已。编

写供其他程序员使用的库时，应该实现一些特殊方法，提供 Python 程序员预期的行为。例如，`__eq__` 方法对业务需求可能没有必要，但是方便程序员测试。

本章不断改写 `Vector2d` 类是为了提供上下文，以便讨论 Python 的特殊方法和编程约定。回看表 1-1，你会发现本章的几个代码清单说明了以下特殊方法。

- 字符串和字节序列表示形式的方法：`__repr__`、`__str__`、`__format__` 和 `__bytes__`。
- 把对象转换成数值的方法：`__abs__`、`__bool__` 和 `__hash__`。
- 支持测试和哈希的 `__eq__` 方法（外加 `__hash__`）。

为了转换成字节序列，本章还实现了一个名为 `Vector2d.frombytes()` 的备选构造函数，顺便又讨论了 `@classmethod`（十分有用）和 `@staticmethod`（不太有用，使用模块层函数更简单）两个装饰器。`frombytes` 方法的实现方式借鉴了 `array.array` 类中的同名方法。

我们了解到，格式规范微语言可通过 `__format__` 方法扩展。在 `__format__` 方法中，我们要做的是解析 `format_spec`。这就是传给内置函数 `format(obj, format_spec)` 的参数、f 字符串的代换字段 `'{:«format_spec»}'`，以及 `str.format()` 方法处理的字符串。

为了把 `Vector2d` 实例变成可哈希的，先让实例不可变，至少要把 `x` 和 `y` 设为私有属性，再以只读特性公开，以防意外修改。随后，实现 `__hash__` 方法，使用推荐的异或运算符计算实例属性的哈希值。

接着，本章讨论了如何使用 `__slots__` 属性节省内存，以及这么做的要注意的问题。`__slots__` 属性有点儿棘手，因此仅当处理特别多

的实例（数百万个，而不是几千个）时才建议使用。如果真有这么多的数量，那么使用 pandas 或许是最好的选择。

最后，本章说明了如何通过访问实例属性（例如 `self.typecode`）覆盖类属性。我们先创建一个实例属性，然后创建子类，在类中覆盖类属性。

前面多次提到，本章的示例受 Python API 的影响很大，这是我长期研究 Python 标准对象的结果。如果用一句话总结本章的内容，那就是：

要构建符合 Python 风格的对象，就要观察真正的 Python 对象的行为。

——源自古老的中国谚语

11.14 延伸阅读

本章介绍了数据模型的几个特殊方法，因此主要参考资料与第 1 章一样，阅读那些资料能对这个话题有整体了解。为方便起见，这里再次给出之前推荐的 4 份资料，同时再多加几份。

《Python 语言参考手册》中的第 3 章“数据模型”

本章用到的方法大部分见于 3.3.1 节“基本定制”。

Python in a Nutshell, 3rd ed. (Alex Martelli、Anna Ravenscroft 和 Steve Holden 著)

深入讲解特殊方法。

《Python Cookbook（第 3 版）中文版》

通过经典实例演示现代 Python 编程实践。尤其是第 8 章“类与对象”，其中有好几种方案与本章讨论的话题有关。

《Python 参考手册（第 4 版）》

详细说明了数据模型，即使（第 4 版）只涵盖了 Python 2.6 和 Python 3。基础概念都是一样的，自 Python 2.2 统一内置类型和用户定义的类以来，数据模型 API 没有任何变化。

2015 年，也就是写完本书第 1 版那一年，Hynek Schlawack 启动了 `attrs` 包的开发。`attrs` 文档中有这么一句话：

`attrs` 包把你从实现对象协议（那些双下划线方法）的苦差事中解放出来，让你重拾编写类的乐趣。

5.10 节提到过 `attrs` 包，它功能强大，是 `@dataclass` 之外的另一种选择。第 5 章中所述的数据类构建器和 `attrs` 包能自动为类配备几个特殊方法。但是，知道如何自己编写特殊方法才能理解那些包的作用，才能判断是否真正需要使用包，才能在必要时覆盖包生成的方法。

本章涵盖了与对象表示形式有关的全部特殊方法，唯有 `__index__` 和 `__fspath__` 没有讲到。`__index__` 将在 12.5.2 节讨论。本书不讲 `__fspath__`，如果你想学习，请阅读“PEP 519—Adding a file system path protocol”。

意识到应该区分字符串表示形式的早期语言是 Smalltalk。1996 年，Bobby Woolf 写了一篇题为“How to Display an Object as a String: `printString` and `displayString`”的文章，讨论了 Smalltalk 对 `printString` 方法和 `displayString` 方法的实现。11.2 节在说明 `repr()` 和 `str()` 的作用时，从这篇文章中借用了言简意赅的表述，即“便于开发者理解的方式”和“便于用户理解的方式”。

杂谈

特性有助于减少前期投入

在 `Vector2d` 类的第 1 版中，`x` 属性和 `y` 属性是公开的。默认情况下，Python 的所有实例属性和类属性都是公开的。这对向量来说是合理的，因为需要访问分量。虽然这些向量是可迭代的对象，而且可以拆包成一对变量，但我们还是希望能够通过 `my_vector.x` 和 `my_vector.y` 获取各个分量。

如果觉得应该避免意外更新 `x` 属性和 `y` 属性，则可以实现特性，但是代码的其他部分没有变化，`Vector2d` 的公开接口也不受影响，这一点从 `doctest` 中可以得知。我们依然能够访问 `my_vector.x` 和 `my_vector.y`。

这表明可以先以最简单的方式定义类，也就是使用公开属性，因为如果以后需要对读值方法和设值方法增加控制，则可以通过特性实现，这样做对一开始通过公开属性的名称（例如 `x` 和 `y`）与对象交互的代码没有影响。

Java 语言采用的方式则截然相反：Java 程序员不能先定义简单的公开属性，等需要时再实现特性，因为 Java 语言没有特性。因此，在 Java 中编写读值方法和设值方法是常态，就算这些方法没什么实际作用。这是因为 API 不能从简单的公开属性变成读值方法和设值方法，同时又不影响使用那些属性的代码。

此外，Martelli、Ravenscroft 和 Holden 在 *Python in a Nutshell, 3rd ed* 一书中指出，到处使用读值方法和设值方法是愚蠢的行为。如果想编写如下代码：

```
>>> my_object.set_foo(my_object.get_foo() + 1)
```

则这样做就行了。

```
>>> my_object.foo += 1
```

维基的发明人和极限编程先驱 Ward Cunningham 建议问以下问题：“做这件事最简单的方法是什么？”意即，我们应该把焦点放在目标上。¹¹ 提前实现设值方法和读值方法偏离了目标。在 Python 中，可以先使用公开属性，然后等需要时再变成特性。

私有属性的安全性和保障性

Perl 不会强制你保护隐私。你应该待在客厅外，因为你没有收到邀请，而不是因为里面有把枪。

——Larry Wall
Perl 之父

Python 和 Perl 在很多方面的做法截然相反，但是 Guido 和 Larry 似乎都同意要保护对象的隐私。

这些年我教过许多 Java 程序员学习 Python，我发现很多人对 Java 提供的隐私保障推崇备至。可事实是，Java 的 `private` 修饰符和 `protected` 修饰符往往只是为了防止意外发生（一种安全措施）。只有使用 `SecurityManager` 部署 Java 应用程序时才能保障绝对安全，防止恶意访问。但是，实际上很少有人这么做，即便在企业中也少见。

下面通过一个 Java 类来证明这一点，如示例 11-20 所示。

示例 11-20 Confidential.java：定义了一个名为 `secret` 的私有字段的 Java 类

```
public class Confidential {  
  
    private String secret = "";  
  
    public Confidential(String text) {  
        this.secret = text.toUpperCase();  
    }  
}
```


示例 11-20 把 `text` 转换成大写后存入了 `secret` 字段。转换成大写只是为了表明 `secret` 字段中的值全部是大写形式。

要使用 Jython 运行 `expose.py` 脚本才能真正说明问题。该脚本使用内省（Java 称之为“反射”）获取私有字段的值。示例 11-21 中是 `expose.py` 脚本的代码。

示例 11-21 `expose.py`：一段 Jython 代码，该代码会从另一个类中读取一个私有字段

```
#!/usr/bin/env jython
# 注意：截至2020年年底，Jython仍只支持Python 2.7

import Confidential

message = Confidential('top secret text')
secret_field = Confidential.getDeclaredField('secret')
secret_field.setAccessible(True) # 攻破防线
print 'message.secret =', secret_field.get(message)
```

运行示例 11-21，得到的结果如下所示。

```
$ jython expose.py
message.secret = TOP SECRET TEXT
```

我们从 `Confidential` 类的私有字段 `secret` 中读取到了字符串 `'TOP SECRET TEXT'`。

这里没有什么黑魔法：`expose.py` 脚本使用 Java 反射 API 获取私有字段 `'secret'` 的引用，然后调用

`'secret_field.setAccessible(True)'` 把它设为可读的。显然，使用 Java 代码也能做到这一点（不过所需的代码行数是这里的 3 倍多，参见本书代码中的 `Expose.java` 文件）。

如果这个 Jython 脚本或 Java 主程序（例如 `Expose.class`）在 `SecurityManager` 的监管下运行，那么关键调用 `.setAccessible(True)` 就会失败。但是在现实中，很少有人

使用 SecurityManager 部署 Java 应用程序，浏览器以前支持的 Java applet 除外。

我的观点是，Java 中的访问控制修饰符基本上也是安全措施，不能保证万无一失，至少在实践中是这样。因此，安心享受 Python 提供的强大功能，放心去用吧。

¹¹参见“Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V”。

第 12 章 序列的特殊方法

不要通过**叫声、走路姿势**等像不像鸭子来检查它**是不是**鸭子，具体检查什么取决于你想使用语言的哪些行为。

(comp.lang.python, 2000 年 7 月 26 日。)

——Alex Martelli

本章将以第 11 章定义的二维向量类 `Vector2d` 为基础，向前迈出一大步，定义表示多维向量的 `Vector` 类。这个类的行为与 Python 中标准的不可变扁平序列一样。`Vector` 实例中的元素是浮点数，本章结束后 `Vector` 类将支持以下功能。

- 基本的序列协议：`__len__` 和 `__getitem__`。
- 正确表述拥有很多元素的实例。
- 适当的切片支持，生成新的 `Vector` 实例。
- 综合各个元素的值计算哈希值。
- 自定义的格式语言扩展。

此外，本章还将通过 `__getattr__` 方法实现属性的动态存取，以取代 `Vector2d` 使用的只读特性——不过，序列类型通常不会这么做。

在大量代码之间，本章将穿插讨论一个概念：把协议当作正式接口。我们将说明协议和**鸭子类型**之间的关系，以及对自定义类型的实际影响。

12.1 本章新增内容

本章没有太大的变化，12.4 节在结尾处新增了一个“提示栏”，简单讨论 `typing.Protocol`。

12.5.2 节实现的 `__getitem__`（参见示例 12-6）比第 1 版更简洁且更强健，这要得益于鸭子类型和 `operator.index`。本章后面实现的 `Vector` 和第 16 章也做了同样的改动。

下面开始吧！

12.2 `Vector` 类：用户定义的序列类型

本章将使用组合模式实现 `Vector` 类，而不使用继承。向量的分量存储在浮点数数组中，而且还将实现不可变扁平序列所需的方法。

不过，在实现序列方法之前，要先实现一个基准，确保 `Vector` 类与前面定义的 `Vector2d` 类兼容（没必要兼容的地方除外）。

三维以上向量的应用程序

谁需要 1000 维向量呢？信息检索领域经常使用 N 维向量（ N 是很大的数），因为查询的文档和文本使用向量表示，一个单词一个维度。这叫向量空间模型。在这个模型中，一个关键的相关指标是余弦相关性（表示查询的向量与表示文档的向量之间夹角的余弦）。夹角越小，余弦值越趋近于 1，文档与查询的相关性越大。

不过，本章定义的 `Vector` 类是为了教学而举的例子，不会涉及很多数学原理。我们的目的是以序列类型为背景说明 Python 的几个特殊方法。

如果在实际使用中需要做向量运算，那么应该使用 NumPy 和 SciPy。Radim Řehůřek 开发的 PyPI 包 `gensim` 使用 NumPy 和 SciPy 实现了用于处理自然语言和检索信息的向量空间模型。

12.3 Vector 类第 1 版：与 Vector2d 类兼容

Vector 类的第 1 版要尽量与前面定义的 Vector2d 类兼容。

然而，我们会故意不让 Vector 的构造函数与 Vector2d 的构造函数兼容。为了编写 `Vector(3, 4)` 和 `Vector(3, 4, 5)` 这样的代码，是可以让 `__init__` 方法接受任意个参数的（借助 `*args`）。但是，序列类型的构造函数最好接受可迭代对象为参数，因为所有内置的序列类型都是这样做的。示例 12-1 展示了 Vector 类的几种实例化方式。

示例 12-1 测试 `Vector.__init__` 方法和 `Vector.__repr__` 方法

```
>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

除了构造函数的新签名，我还确保了在传入两个分量（例如 `Vector([3, 4])`）时，Vector2d 类（例如 `Vector2d(3, 4)`）的每个测试都能通过，而且得到相同的结果。



如果 Vector 实例的分量超过 6 个，那么 `repr()` 生成的字符串就会使用 `...` 省略一部分，如示例 12-1 中的最后一行所示。包含大量元素的容器类型一定要这么做，因为 `repr` 是用于调试的，你肯定不想让大型对象在控制台或日志中输出几千行内容。使用 `reprlib` 模块可以生成长度有限的表示形式，如示例 12-2 所示。在 Python 2.7 中，`reprlib` 模块的名称是 `repr`。

示例 12-2 是第 1 版 `Vector` 类的实现代码（以示例 11-2 和示例 11-3 中的代码为基础）。

示例 12-2 `vector_v1.py`: 从 `vector2d_v1.py` 衍生而来

```
from array import array
import reprlib
import math

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components) ❶

    def __iter__(self):
        return iter(self._components) ❷

    def __repr__(self):
        components = reprlib.repr(self._components) ❸
        components = components[components.find('['):-1] ❹
        return f'Vector({components})'

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components)) ❺

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.hypot(*self) ❻

    def __bool__(self):
        return bool(abs(self))

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv) ❼
```

❶ `self._components` 是“受保护”的实例属性，把 `Vector` 的分量保存在一个数组中。

❷ 为了迭代，使用 `self._components` 构建一个迭代器，作为返回值。¹

¹`iter()` 函数和 `__iter__` 方法将在第 17 章讨论。

❸ 使用 `reprlib.repr()` 函数生成 `self._components` 的有限长度表示形式（例如 `array('d', [0.0, 1.0, 2.0, 3.0, 4.0, ...])`）。

❹ 把字符串插入 `Vector` 的构造函数调用之前，去掉前面的 `array('d',` 和后面的 `)`。

❺ 直接使用 `self._components` 构建 `bytes` 对象。

❻ 从 Python 3.8 开始，`math.hypot` 接受 N 维坐标点。以前使用的表达式是 `math.sqrt(sum(x * x for x in self))`。

❼ 只需在 `frombytes` 方法的基础上改动最后一行：直接把 `memoryview` 传给构造函数，不用像前面那样使用 `*` 拆包。

这里使用 `reprlib.repr` 的方式需要做些说明。这个函数用于生成大型结构或递归结构的安全表示形式，它会限制输出字符串的长度，用 `'...'` 表示截断的部分。我希望 `Vector` 实例的表示形式是 `Vector([3.0, 4.0, 5.0])`，而不是 `Vector(array('d', [3.0, 4.0, 5.0]))`，因为 `Vector` 实例中的数组是实现细节。由于构造函数的这两种调用方式所构建的 `Vector` 对象是一样的，因此可以选择使用更简单的句法，即传入列表参数。

编写 `__repr__` 方法时，本可以使用表达式 `reprlib.repr(list(self._components))` 生成简化的 `components` 显示形式。然而，这么做有点儿浪费资源，因为要把

`self._components` 中的每个元素复制到一个列表中，然后使用列表的表示形式。我没有这么做，而是直接把 `self._components` 传给 `reprlib.repr` 函数，然后去掉 `[]` 外面的字符，如示例 12-2 中 `__repr__` 方法的第二行所示。



调用 `repr()` 函数的目的是调试，因此绝对不能抛出异常。如果 `__repr__` 方法的实现有问题，则必须处理，尽量输出有用的内容，让用户能够识别接收者 (`self`)。

注意，`__str__` 方法、`__eq__` 方法和 `__bool__` 方法与 `Vector2d` 类中一样，而 `frombytes` 方法也只变了一个字符（最后一行把 `*` 去掉了）。这是 `Vector2d` 可迭代的好处之一。

顺便说一下，本可以让 `Vector` 继承 `Vector2d`，但是我没有那么做，原因有二。第一，两个构造函数不兼容，不适合使用继承。这一点通过适当处理 `__init__` 方法的参数可以解决，不过第二个原因更重要：我想把 `Vector` 类当作单独的示例，以此实现序列协议。接下来我们会先讨论协议这个术语，然后再实现序列协议。

12.4 协议和鸭子类型

如第 1 章所述，在 Python 中创建功能完善的序列类型无须使用继承，实现符合序列协议的方法即可。不过，这里说的协议是什么呢？

在面向对象编程中，协议是非正式的接口，只在文档中定义，不在代码中定义。例如，Python 的序列协议只需要 `__len__` 和 `__getitem__` 这两个方法。任何类（例如 `Spam`），只要使用标准的签名和语义实现了这两个方法，就能用在任何预期序列的地方。`Spam` 是不是哪个类的子类无关紧要，只要提供了所需的方法即可。示例 1-1 就是一例，这里再次给出代码，如示例 12-3 所示。

示例 12-3 示例 1-1 的代码，为了方便参考，再次给出


```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

示例 12-3 中的 `FrenchDeck` 类能充分利用 Python 的很多功能，因为它实现了序列协议，即使代码中并没有声明这一点。任何有经验的 Python 程序员只要看一眼就知道它是序列，即便它是 `object` 的子类也无妨。我们说它是序列，因为它的**行为**像序列，这才是重点。

根据本章开头引用的 Alex Martelli 的帖文，人们称其为**鸭子类型**（duck typing）。

协议是非正式的，没有强制力，因此如果知道类的具体使用场景，那么通常只需要实现协议的一部分。例如，为了支持迭代，只需实现 `__getitem__` 方法，没必要提供 `__len__` 方法。



实现“PEP 544—Protocols: Structural subtyping (static duck typing)”之后，Python 3.8 开始支持**协议类**（protocol class），即 8.5.10 节讲过的 `typing` 结构。这里的“协议”与传统意义有关系，但不完全相同。如果需要区分，那么可以使用**静态协议**指代协议类规定的协议，使用**动态协议**指代传统意义上的协议。二者之间主要的区别是，静态协议的实现必须提供静态类中定义的所有方法。13.3 节会进一步探讨这个话题。

下面，我们将在 `Vector` 类中实现序列协议，该类暂不完全支持切片，稍后再完善。

12.5 `Vector` 类第 2 版：可切片的序列

如 `FrenchDeck` 类所示，如果能委托给对象中的序列属性（例如 `self._components` 数组），则支持序列协议特别简单。下面只有一行代码的 `__len__` 方法和 `__getitem__` 方法是很好的开始。

```
class Vector:
    # 省略了很多行
    # ...

    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        return self._components[index]
```

添加这两个方法之后，以下操作都能执行了。

```
>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[-1]
(3.0, 5.0)
>>> v7 = Vector(range(7))
>>> v7[1:4]
array('d', [1.0, 2.0, 3.0])
```

可以看到，连切片都支持了，不过尚不完美。如果 `Vector` 实例的切片也是 `Vector` 实例，而不是数组，那就更好了。前面那个 `FrenchDeck` 类也有类似的问题：切片得到的是列表。对 `Vector` 来说，如果切片生成普通的数组，那么将会失去大量功能。

想想内置序列类型：切片得到的都是各自类型的新实例，而不是其他类型。

为了把 `Vector` 实例的切片也变成 `Vector` 实例，不能简单地把切片操作委托给数组。要分析传给 `__getitem__` 方法的参数，做适当的处理。

下面来看 Python 如何把 `my_seq[1:3]` 句法变成传给 `my_seq.__getitem__(...)` 的参数。

12.5.1 切片原理

一例胜千言，下面来看示例 12-4。

示例 12-4 观察 `__getitem__` 和切片的行为

```
>>> class MySeq:
...     def __getitem__(self, index):
...         return index ❶
...
>>> s = MySeq()
>>> s[1] ❷
1
>>> s[1:4] ❸
slice(1, 4, None)
>>> s[1:4:2] ❹
slice(1, 4, 2)
>>> s[1:4:2, 9] ❺
(slice(1, 4, 2), 9)
>>> s[1:4:2, 7:9] ❻
(slice(1, 4, 2), slice(7, 9, None))
```

❶ 在这个示例中，`__getitem__` 直接返回传给它的值。

❷ 单个索引，没什么新奇的。

❸ `1:4` 表示法变成了 `slice(1, 4, None)`。

❹ `slice(1, 4, 2)` 的意思是从 1 开始，到 4 结束，步幅为 2。

❶ 神奇的事发生了：如果 [] 中有逗号，那么 `__getitem__` 收到的就是元组。

❷ 元组中甚至可以有多个 `slice` 对象。

现在，来仔细看看 `slice` 本身，如示例 12-5 所示。

示例 12-5 查看 `slice` 类的属性

```
>>> slice ❶
<class 'slice'>
>>> dir(slice) ❷
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'indices', 'start', 'step', 'stop']
```

❶ `slice` 是内置的类型（首次出现于 2.7.2 节）。

❷ 查看 `slice`，我们发现它有 `start`、`stop` 和 `step` 这 3 种数据属性，还有 `indices` 方法。

在示例 12-5 中，调用 `dir(slice)` 得到的结果中有个 `indices` 属性，这是一个方法，作用很大，但是鲜为人知。

`help(slice.indices)` 给出的信息如下。

```
S.indices(len) -> (start, stop, stride)
```

给定长度为 `len` 的序列，计算 `s` 表示的扩展切片的起始 (`start`) 索引和结尾 (`stop`) 索引，以及步幅 (`stride`)。超出边界的索引会被截掉，就像常规切片一样。

换句话说，`indices` 方法开放了内置序列实现的棘手逻辑，可以优雅地处理缺失索引和负数索引，以及长度超过目标序列的切片。这个方

法会“整顿”元组，把 `start`、`stop` 和 `stride` 都变成非负数，而且都落在指定长度序列的边界内。

下面举几个例子。假设有一个长度为 5 的序列，例如 `'ABCDE'`。

```
>>> slice(None, 10, 2).indices(5) ❶
(0, 5, 2)
>>> slice(-3, None, None).indices(5) ❷
(2, 5, 1)
```

❶ `'ABCDE'[:10:2]` 等同于 `'ABCDE'[0:5:2]`。

❷ `'ABCDE'[-3:]` 等同于 `'ABCDE'[2:5:1]`。

在 `Vector` 类中无须使用 `slice.indices()` 方法，因为收到切片参数时，我们委托 `_components` 数组处理。因此，如果没有底层序列类型作为依靠，那么使用这个方法能节省大量时间。

现在知道如何处理切片了，下面来看 `Vector.__getitem__` 方法改进后的实现。

12.5.2 能处理切片的 `__getitem__` 方法

示例 12-6 列出了让 `Vector` 表现为序列所需的两个方法：`__len__` 和 `__getitem__`（后者现在能正确处理切片了）。

示例 12-6 `vector_v2.py` 的部分代码：为 `vector_v1.py` 中的 `Vector` 类（参见示例 12-2）添加 `__len__` 方法和 `__getitem__` 方法

```
def __len__(self):
    return len(self._components)

def __getitem__(self, key):
    if isinstance(key, slice): ❶
        cls = type(self) ❷
        return cls(self._components[key]) ❸
```

```
index = operator.index(key) ❹  
return self._components[index] ❺
```

- ❶ 如果 `key` 参数的值是一个 `slice` 对象.....
- ❷ 就获取实例的类 (`Vector`) , 然后.....
- ❸ 调用类的构造函数, 使用 `_components` 数组的切片构建一个新 `Vector` 实例。
- ❹ 如果从 `key` 中得到的是单个索引.....
- ❺ 就返回 `_components` 中相应的元素。

`operator.index()` 函数背后调用特殊方法 `__index__`。这个函数和特殊方法在 Travis Oliphant 提议的“PEP 357—Allowing Any Object to be Used for Slicing”中定义, 目的是支持 NumPy 中众多的整数类型作为索引和切片参数。`operator.index()` 和 `int()` 之间的主要区别是, 前者只有这一个用途。例如, `int(3.14)` 返回 3, 而 `operator.index(3.14)` 抛出 `TypeError`, 因为 `float` 值不可能是索引。



大量使用 `isinstance` 可能表明面向对象设计得不好, 不过在 `__getitem__` 方法中使用它处理切片是合理的。本书第 1 版还使用 `isinstance` 测试了 `key`, 判断它是不是整数。有了 `operator.index` 就不用再测试了。如果从 `key` 中得不到索引, 那么 `operator.index` 就会抛出 `TypeError`, 而且报错消息非常详细, 可以参见示例 12-7 末尾的错误消息。

把示例 12-6 中的代码添加到 `Vector` 类中之后, 切片行为就正确了, 如示例 12-7 所示。

示例 12-7 测试示例 12-6 中改进的 `Vector.__getitem__` 方法

```

>>> v7 = Vector(range(7))
>>> v7[-1] ❶
6.0
>>> v7[1:4] ❷
Vector([1.0, 2.0, 3.0])
>>> v7[-1:] ❸
Vector([6.0])
>>> v7[1,2] ❹
Traceback (most recent call last):
...
TypeError: 'tuple' object cannot be interpreted as an integer

```

- ❶ 单个整数索引只获取一个分量，值为浮点数。
- ❷ 切片索引创建一个新 `Vector` 实例。
- ❸ 长度为 1 的切片也创建一个 `Vector` 实例。
- ❹ `Vector` 不支持多维索引，因此索引元组或多个切片会抛出错误。

12.6 Vector 类第 3 版：动态存取属性

`Vector2d` 变成 `Vector` 之后，就无法通过名称访问向量的分量（例如 `v.x` 和 `v.y`）了。现在，我们处理的向量可能有大量分量。不过，如果能通过单个字母访问前几个分量的话会比较方便。例如，用 `x`、`y` 和 `z` 代替 `v[0]`、`v[1]` 和 `v[2]`。

我们想额外提供以下句法，用于读取向量的前 4 个分量。

```

>>> v = Vector(range(10))
>>> v.x
0.0
>>> v.y, v.z, v.t
(1.0, 2.0, 3.0)

```

在 `Vector2d` 中，使用 `@property` 装饰器把 `x` 和 `y` 标记为只读特性（参见示例 11-7）。可以在 `Vector` 中编写 4 个特性，但这样太麻

烦。特殊方法 `__getattr__` 提供了更好的方式。

属性查找失败后，解释器会调用 `__getattr__` 方法。简单来说，对于 `my_obj.x` 表达式，Python 会检查 `my_obj` 实例有没有名为 `x` 的属性。如果没有，就到类 (`my_obj.__class__`) 中查找；如果还没有，就沿着继承图继续向上查找；² 如果依旧找不到，则调用 `my_obj` 所属的类中定义的 `__getattr__` 方法，传入 `self` 和属性名称的字符串形式（例如 `'x'`）。

²属性查找机制比这复杂得多，具体细节在第五部分讲解。目前知道这种简单的机制即可。

示例 12-8 中列出的是为 `Vector` 类定义的 `__getattr__` 方法。这个方法的作用很简单，它会检查所查找的属性是不是 `xyzt` 中的某个字母，如果是，就返回对应的向量的分量。

示例 12-8 `vector_v3.py` 的部分代码：向 `Vector` 类中添加 `__getattr__` 方法

```
__match_args__ = ('x', 'y', 'z', 't') ❶

def __getattr__(self, name):
    cls = type(self) ❷
    try:
        pos = cls.__match_args__.index(name) ❸
    except ValueError: ❹
        pos = -1
    if 0 <= pos < len(self._components): ❺
        return self._components[pos]
    msg = f'{cls.__name__!r} object has no attribute {name!r}'
    ❻
    raise AttributeError(msg)
```

❶ 设定 `__match_args__`，让 `__getattr__` 实现的动态属性支持位置模式匹配。³

³尽管 Python 3.10 才出现为模式匹配提供支持的 `__match_args__`，但是在旧版中设定这个属性也没有什么危害。本书第 1 版把这个属性命名为 `shortcut_names`。第 2 版中使用

的新名称有两个职责：一是在 `case` 子句中使用时支持位置模式，二是存储 `__getattr__` 和 `__setattr__` 的特殊逻辑实现的动态属性名称。

- ❷ 获取 `Vector` 类，供后面使用。
- ❸ 尝试获取 `name` 在 `__match_args__` 中的位置。
- ❹ 如果未找到 `name`，那么 `.index(name)` 就会抛出 `ValueError`。此时，把 `pos` 设为 `-1`。（我也想在这里使用 `str.find` 之类的方法，可惜 `tuple` 没有实现这样的方法。）
- ❺ 如果 `pos` 落在分量长度范围内，就返回对应的分量。
- ❻ 如果执行到这里，就抛出 `AttributeError`，输出一个标准消息。

`__getattr__` 方法的实现不难，但是这样实现还不够。看看示例 12-9 中古怪的交互行为。

示例 12-9 不恰当的行为：为 `v.x` 赋值没有抛出错误，但是前后矛盾

```
>>> v = Vector(range(5))
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])
>>> v.x ❶
0.0
>>> v.x = 10 ❷
>>> v.x ❸
10
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0]) ❹
```

- ❶ 使用 `v.x` 获取第一个元素 (`v[0]`)。
- ❷ 为 `v.x` 赋新值。这个操作应该抛出异常。

③ 读取 `v.x`，得到的是新值 10。

④ 可是，向量的分量没变。

你能解释为什么会这样吗？也就是说，如果向量的分量数组中没有新值，那么为什么 `v.x` 会返回 10？如果不能立即给出解释，那么再看看示例 12-8 前面对 `__getattr__` 方法的说明。虽然原因不是很明显，但它是理解本书后面内容的重要基础。

请你自己思考一番，然后再继续往下读，了解具体原因。

示例 12-9 前后矛盾的行为是由 `__getattr__` 的运作方式导致的：仅当对象没有指定名称的属性时，Python 才会调用 `__getattr__` 方法，这是一种后备机制。可是，像 `v.x = 10` 这样赋值之后，`v` 对象就有 `x` 属性了，因此使用 `v.x` 获取 `x` 属性的值时不会再调用 `__getattr__` 方法，解释器会直接返回 `v.x` 绑定的值，即 10。另外，`__getattr__` 方法目前的实现没有考虑到 `self._components` 之外的实例属性，而是从这个属性中获取 `__match_args__` 列出的“虚拟属性”。

为了避免这种前后矛盾的现象，需要改写 `Vector` 类中设置属性的逻辑。

回想第 11 章最后一个 `Vector2d` 示例，如果为 `.x` 或 `.y` 实例属性赋值，就会抛出 `AttributeError`。为了避免歧义，在 `Vector` 类中，当为名称是单个小写字母的属性赋值时，我们也不想抛出那个异常。为此，要实现 `__setattr__` 方法，如示例 12-10 所示。

示例 12-10 `vector_v3.py` 的部分代码：在 `Vector` 类中实现 `__setattr__` 方法

```
def __setattr__(self, name, value):
    cls = type(self)
    if len(name) == 1: ❶
        if name in cls.__match_args__: ❷
```

```

        error = 'readonly attribute {attr_name!r}'
    elif name.islower(): ❸
        error = "can't set attributes 'a' to 'z' in
{cls_name!r}"
    else:
        error = '' ❹
    if error: ❺
        msg = error.format(cls_name=cls.__name__,
attr_name=name)
        raise AttributeError(msg)
    super().__setattr__(name, value) ❻

```

- ❶ 特别处理名称是单个字符的属性。
- ❷ 如果 name 在 `__match_args__` 中，就设置特殊的错误消息。
- ❸ 如果 name 是小写字母，就设置一个针对所有小写字母的错误消息。
- ❹ 否则，把错误消息设为空字符串。
- ❺ 如果错误消息不为空，就抛出 `AttributeError`。
- ❻ 默认情况：在超类上调用 `__setattr__` 方法，提供标准行为。



`super()` 函数用于动态访问超类的方法，对 Python 这种支持多重继承的动态语言来说，必须这么做。程序员经常使用这个函数把子类方法的某些任务委托给超类中适当的方法，如示例 12-10 所示。14.4 节会进一步探讨 `super` 函数。

为了给 `AttributeError` 选择错误消息，我首先查看了内置类型 `complex` 的行为，因为 `complex` 对象是不可变的，而且有一对数据属性：`real` 和 `imag`。如果试图修改任何一个属性，那么 `complex` 实例就会抛出 `AttributeError`，错误消息是 `"can't set attribute"`。而如果尝试为受特性保护的只读属性赋值（像 11.7 节那样做），则得到的错误消息是 `"read-only attribute"`。在

`__setitem__` 方法中为 `error` 字符串选词时，我参考了这两个错误消息，而且更为明确地指出了禁止赋值的属性。

注意，我们没有禁止为所有属性赋值，只是禁止为单个小写字母属性赋值，以防与只读属性 `x`、`y`、`z` 和 `t` 混淆。



我们知道，在类中声明 `__slots__` 属性可以防止设置新实例属性。因此，你可能想使用这个功能，而不像这里所做的那样实现 `__setattr__` 方法。可是，正如 11.11.2 节指出的，不建议只为了避免创建实例属性而使用 `__slots__`。`__slots__` 只应该用于节省内存，而且仅当内存严重不足时才应该这么做。

虽然这个示例不支持为 `Vector` 分量赋值，但是有一个问题要特别注意：大多数时候，如果实现了 `__getattr__` 方法，那么也要定义 `__setattr__` 方法，以防对象的行为不一致。

如果想允许修改分量，则可以实现 `__setitem__` 方法以支持 `v[0] = 1.1` 这样的赋值，以及（或者）实现 `__setattr__` 方法以支持 `v.x = 1.1` 这样的赋值。不过，我们要保持 `Vector` 是不可变的，因为 12.7 节将把它变成可哈希的。

12.7 `Vector` 类第 4 版：哈希和快速等值测试

我们要再次实现 `__hash__` 方法。加上现有的 `__eq__` 方法，这会把 `Vector` 实例变成可哈希的对象。

`Vector2d` 中的 `__hash__` 方法（参见示例 11-8）会计算包含 `self.x` 和 `self.y` 这两个分量的元组的哈希值。现在要处理的分量或许有上千个，因此构建元组消耗的资源可能太多。鉴于此，我们将使用 `^`（异或）运算符依次计算各个分量的哈希值，就像这样：`v[0]`

$\wedge v[1] \wedge v[2]$ 。这正是 `functools.reduce` 函数的作用。虽然之前我说过 `reduce` 没有以往那么常用了，⁴ 但是计算向量所有分量的哈希值非常适合使用这个函数。`reduce` 函数的整体思路如图 12-1 所示。

⁴ `sum`、`any` 和 `all` 涵盖了 `reduce` 的大部分用途。详见“`map`、`filter` 和 `reduce` 的现代替代品”一节的讨论。

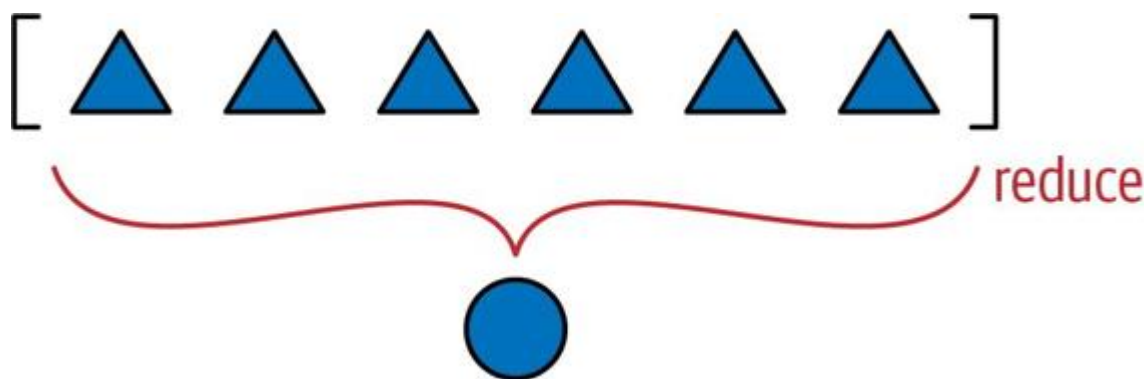


图 12-1：归约函数（`reduce`、`sum`、`any` 和 `all`）把序列或有限的可迭代对象聚合成一个结果

我们已经知道，`sum()` 可以代替 `functools.reduce()`，下面说说它的原理。`reduce()` 的关键思想是，把一系列值归约成单个值。`reduce()` 函数的第一个参数是一个接受两个参数的函数，第二个参数是一个可迭代对象。假如有一个接受两个参数的函数 `fn` 和一个列表 `lst`。调用 `reduce(fn, lst)` 时，`fn` 首先会被应用到第一对元素上，即 `fn(lst[0], lst[1])`，生成第一个结果 `r1`。然后，`fn` 会被应用到 `r1` 和下一个元素上，即 `fn(r1, lst[2])`，生成第二个结果 `r2`。接着，调用 `fn(r2, lst[3])`，生成 `r3`……直到最后一个元素，返回最后得到的结果 `rN`。

使用 `reduce` 函数可以计算 $5!$ （5 的阶乘）。

```
>>> 2 * 3 * 4 * 5 # 想要的结果是5! == 120
120
>>> import functools
```

```
>>> functools.reduce(lambda a,b: a*b, range(1, 6))
120
```

回到哈希问题上。示例 12-11 展示了计算聚合异或的 3 种方式：一种使用 for 循环，其余两种使用 reduce 函数。

示例 12-11 计算整数 0~5 的累计异或的 3 种方式

```
>>> n = 0
>>> for i in range(1, 6): ❶
...     n ^= i
...
>>> n
1
>>> import functools
>>> functools.reduce(lambda a, b: a^b, range(6)) ❷
1
>>> import operator
>>> functools.reduce(operator.xor, range(6)) ❸
1
```

❶ 使用 for 循环和累加器变量计算聚合异或。

❷ 使用 functools.reduce 函数，传入匿名函数。

❸ 使用 functools.reduce 函数，把 lambda 表达式换成 operator.xor。

示例 12-11 的 3 种方式中，我最喜欢最后一种，其次是 for 循环。你呢？

7.8.1 节讲过，operator 模块以函数的形式提供了所有的 Python 中缀运算符。借此可以减少使用 lambda 表达式的必要。

为了使用我喜欢的方式编写 Vector.__hash__ 方法，需要导入 functools 模块和 operator 模块。Vector 类的相关改动如示例 12-12 所示。

示例 12-12 `vector_v4.py` 的部分代码：在 `vector_v3.py` 中的 `Vector` 类的基础上导入两个模块，并添加 `__hash__` 方法

```
from array import array
import reprlib
import math
import functools ❶
import operator ❷

class Vector:
    typecode = 'd'

    # 排版需要，省略了很多行...
    def __eq__(self, other): ❸
        return tuple(self) == tuple(other)

    def __hash__(self):
        hashes = (hash(x) for x in self._components) ❹
        return functools.reduce(operator.xor, hashes, 0) ❺

    # 又省略了很多行...
```

- ❶ 为了使用 `reduce` 函数，导入 `functools` 模块。
- ❷ 为了使用 `xor` 函数，导入 `operator` 模块。
- ❸ `__eq__` 方法没有变化。这里把它列出来是为了将其和 `__hash__` 方法放在一起，因为它们要结合在一起使用。
- ❹ 创建一个生成器表达式，惰性计算各个分量的哈希值。
- ❺ 把 `hashes` 提供给 `reduce` 函数，使用 `xor` 函数计算聚合的哈希值。第三个参数（0）是初始值（参见下面的“警告栏”）。



使用 `reduce` 函数时最好提供第三个参数

`reduce(function, iterable, initializer)`，避免如下异常：`TypeError: reduce() of empty sequence with no initial value`（这个错误消息很棒，不仅说明了问题，

还提供了解决方法)。如果序列为空, 那么返回值就是 `initializer`; 否则, 在归约循环中, 以 `initializer` 作为第一个参数。因此, `initializer` 应该是所执行操作的么元值 (identity value)。例如, 对 `+`、`|` 和 `^` 来说, `initializer` 应该是 0; 而对 `*` 和 `&` 来说, `initializer` 应该是 1。

示例 12-12 实现的 `__hash__` 方法是一种完美的映射归约 (map-reduce) 计算, 如图 12-2 所示。

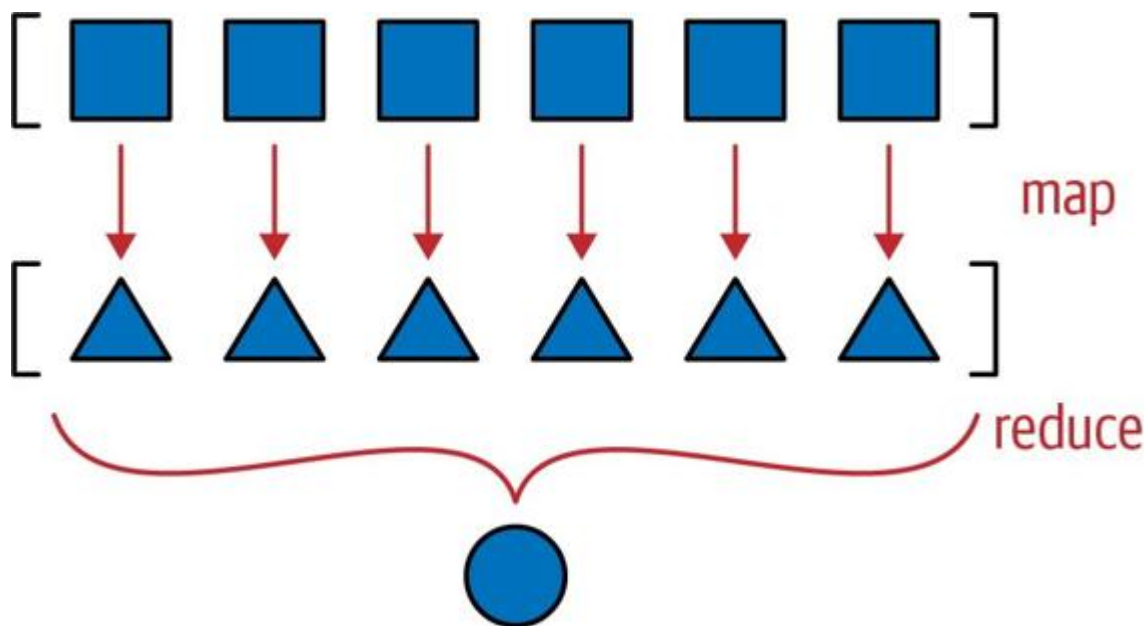


图 12-2: 映射归约: 把函数应用到各个元素上, 生成一个新序列 (映射), 然后计算聚合值 (归约)

映射过程会计算各个分量的哈希值, 归约过程则使用 `xor` 运算符聚合所有的哈希值。把**生成器表达式**替换成 `map` 函数, 映射过程更明显。

```
def __hash__(self):
    hashes = map(hash, self._components)
    return functools.reduce(operator.xor, hashes)
```



在 Python 2 中使用 `map` 函数效率低一些, 因为 `map` 函数要使用结果构建一个新列表。但是在 Python 3 中, `map` 函数是

惰性的，它会创建一个生成器，按需产出结果，因此能节省内存——这与示例 12-8 中使用生成器表达式定义 `__hash__` 方法的原理一样。

既然讲到了归约函数，那就把前面草草实现的 `__eq__` 方法修改一下，减少处理时间和内存用量——至少对大型向量来说如此。示例 11-2 实现的 `__eq__` 方法非常简洁。

```
def __eq__(self, other):
    return tuple(self) == tuple(other)
```

`Vector2d` 和 `Vector` 都可以这样做，甚至还会认为 `Vector([1, 2])` 和 `(1, 2)` 相等。这或许是个问题，不过可以暂且忽略。⁵ 可是，当 `Vector` 实例有上千个分量时，效率十分低下。上述实现方式要完整复制两个运算对象，构建两个元组，而这么做只是为了使用 `tuple` 类型的 `__eq__` 方法。对于 `Vector2d`（只有两个分量），这是个捷径，但是对维数很多的向量来说情况就不同了。示例 12-13 中比较两个 `Vector` 实例或者与一个可迭代对象比较的方式更好。

⁵16.2 节将认真对待 `Vector([1, 2]) == (1, 2)` 问题。

示例 12-13 为了提高比较效率，`Vector.__eq__` 方法在 `for` 循环中使用 `zip` 函数

```
def __eq__(self, other):
    if len(self) != len(other): ❶
        return False
    for a, b in zip(self, other): ❷
        if a != b: ❸
            return False
    return True ❹
```

❶ 如果两个对象的长度不一样，那么它们就不相等。

❷ `zip` 函数生成一个由元组构成的生成器，元组中的元素来自参数传入的各个可迭代对象。如果不熟悉 `zip` 函数，请阅读后面的“出色的

zip 函数”附注栏。前面比较长度的测试是有必要的，因为一旦有一个输入对象耗尽，zip 函数就会立即停止生成值，而且不发出警告。

③ 只要有两个分量不同，就返回 False，退出。

④ 否则，两个对象是相等的。



zip 函数的名称取自拉链，因为此物品把两边的链牙咬合在一起，这形象地说明了 zip(left, right) 的作用。zip 函数与文件压缩没有关系。

示例 12-13 的效率很好，不过用于计算聚合值的整个 for 循环可以替换成一行 all 函数调用。如果所有对应分量的比较结果都是 True，那么结果就是 True。只要有一次比较的结果是 False，all 函数就会返回 False。使用 all 函数实现 __eq__ 方法的方式如示例 12-14 所示。

示例 12-14 使用 zip 函数和 all 函数实现 Vector.__eq__ 方法，逻辑与示例 12-13 一样

```
def __eq__(self, other):
    return len(self) == len(other) and all(a == b for a, b in
zip(self, other))
```

注意，需要先检查两个运算对象的长度是否相同，因为 zip 函数会在最短的那个运算对象耗尽时停止。

我们选择在 vector_v4.py 中采用示例 12-14 中实现的 __eq__ 方法。

出色的 zip 函数

使用 for 循环迭代元素无须处理索引变量，还能避免很多 bug，但是需要一些特殊的实用函数协助，其中一个是内置函数 zip。使用 zip 函数能轻松地并行迭代两个或更多个可迭代对象，返回

的元组可以拆包成变量，分别对应各个输入对象中的一个元素。如示例 12-15 所示。

示例 12-15 内置函数 `zip` 的使用示例

```
>>> zip(range(3), 'ABC') ❶  
<zip object at 0x10063ae48>  
>>> list(zip(range(3), 'ABC')) ❷  
[(0, 'A'), (1, 'B'), (2, 'C')]  
>>> list(zip(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3])) ❸  
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2)]  
>>> from itertools import zip_longest ❹  
>>> list(zip_longest(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3],  
fillvalue=-1))  
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2), (-1, -1, 3.3)]
```

- ❶ `zip` 函数返回一个生成器，按需生成元组。
- ❷ 为了输出，构建一个列表。通常，我们会迭代生成器。
- ❸ 当一个可迭代对象耗尽后，`zip` 不发出警告就停止。
- ❹ `itertools.zip_longest` 函数的行为有所不同，它使用可选的 `fillvalue`（默认值为 `None`）来填充缺失的值，因此可以继续生成元组，直到最后一个可迭代对象耗尽。



Python 3.10 为 `zip()` 增加的选项

本书第 1 版说过，`zip` 毫无征兆地在最短的可迭代对象耗尽后停止，这很是奇怪，优秀的 API 不应这么做。默不作声忽略输入的一部分可能导致难以察觉的 bug。如果各个可迭代对象的长度不同，那么 `zip` 就应该抛出 `ValueError`，就像把可迭代对象拆包到长度不同的变量元组时那样，与 Python 的**快速失败**策略保持一致。“PEP 618—Add Optional Length-Checking To `zip`”提议为 `zip` 函数增加一个可选的参

数 `strict`，以表现这种行为。PEP 618 在 Python 3.10 中已经实现。

`zip` 函数还可以转置以嵌套的可迭代对象表示的矩阵。

```
>>> a = [(1, 2, 3),
...      (4, 5, 6)]
>>> list(zip(*a))
[(1, 4), (2, 5), (3, 6)]
>>> b = [(1, 2),
...      (3, 4),
...      (5, 6)]
>>> list(zip(*b))
[(1, 3, 5), (2, 4, 6)]
```

如果想掌握 `zip` 函数，那么请花点儿时间研究一下这几个示例。

为了避免在 `for` 循环中直接处理索引变量，还经常使用内置生成器函数 `enumerate`。如果不熟悉这个函数，那么一定要阅读“Built-in functions”文档。内置函数 `zip` 和 `enumerate`，以及标准库中其他几个生成器函数将在 17.9 节讨论。

本章最后要像 `Vector2d` 类那样，为 `Vector` 类实现 `__format__` 方法。

12.8 Vector 类第 5 版：格式化

`Vector` 类的 `__format__` 方法类似于 `Vector2d` 类的方法，但是不使用极坐标，而使用球面坐标（也叫“超球面”坐标），因为 `Vector` 类支持 n 个维度，而超过四维后，球体变成了“超球体”。⁶ 因此，我们将把自定义的格式后缀由 `'p'` 改成 `'h'`。

⁶Wolfram Mathworld 网站中有一篇介绍超球体的文章。



11.6 节说过，扩展格式规范微语言时，最好避免重用支持内置类型的格式代码。这里对微语言的扩展还会用到浮点数的格式代码 `'eEfFgGn%'`，而且会保持原意，因此绝对要避免重用代码。整数使用的格式代码是 `'bcdoxXn'`，字符串使用的是 `'s'`。在 `Vector2d` 类中，我选择使用 `'p'` 表示极坐标。使用 `'h'` 表示超球面坐标是不错的选择。

例如，对四维空间 (`len(v) == 4`) 中的 `Vector` 对象来说，`'h'` 代码得到的结果如下：`<r,  $\phi_1$ ,  $\phi_2$ ,  $\phi_3$ >`，其中 `r` 是模 (`abs(v)`)，余下 3 个数是角坐标 ϕ_1 、 ϕ_2 和 ϕ_3 。

下面几个示例摘自 `vector_v5.py` 中的 `doctest`（参见示例 12-16），演示了四维球面坐标格式。

```
>>> format(Vector([-1, -1, -1, -1]), 'h')
'<2.0, 2.0943951023931957, 2.186276035465284, 3.9269908169872414>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
```

在小幅改动 `__format__` 方法之前，要定义两个辅助方法：一个是 `angle(n)`，用于计算某个角坐标（例如 ϕ_1 ）；另一个是 `angles()`，用于返回由所有角坐标构成的可迭代对象。本书不会讲解其中涉及的数学原理，如果你对此感到好奇，可以查看维基百科中的“*n* 维球体”词条，我就是使用那里的几个公式把 `Vector` 分量数组内的笛卡儿坐标转换成球面坐标的。

示例 12-16 是 `vector_v5.py` 脚本的完整代码，包含自 12.3 节以来实现的所有代码和本节实现的自定义格式。

示例 12-16 `vector_v5.py`： `Vector` 类最终版的 `doctest` 和全部代码，带标号那几行是为了支持 `__format__` 方法而添加的代码

```
"""
```

一个多维``Vector``类, 第5版

``Vector``实例使用数值可迭代对象构建::

```
>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

测试二维向量 (结果与``vector2d\_v1.py``一样)::

```
>>> v1 = Vector([3, 4])
>>> x, y = v1
>>> x, y
(3.0, 4.0)
>>> v1
Vector([3.0, 4.0])
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0)
>>> octets = bytes(v1)
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)
5.0
>>> bool(v1), bool(Vector([0, 0]))
(True, False)
```

测试类方法``.frombytes()``::

```
>>> v1_clone = Vector.frombytes(bytes(v1))
>>> v1_clone
Vector([3.0, 4.0])
>>> v1 == v1_clone
True
```

测试三维向量::

```
>>> v1 = Vector([3, 4, 5])
```

```

>>> x, y, z = v1
>>> x, y, z
(3.0, 4.0, 5.0)
>>> v1
Vector([3.0, 4.0, 5.0])
>>> v1_clone = eval(repr(v1))
>>> v1 == v1_clone
True
>>> print(v1)
(3.0, 4.0, 5.0)
>>> abs(v1) # doctest:+ELLIPSIS
7.071067811...
>>> bool(v1), bool(Vector([0, 0, 0]))
(True, False)

```

测试多维向量::

```

>>> v7 = Vector(range(7))
>>> v7
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
>>> abs(v7) # doctest:+ELLIPSIS
9.53939201...

```

测试``.\_\_bytes\_\_``和``.frombytes()``方法::

```

>>> v1 = Vector([3, 4, 5])
>>> v1_clone = Vector.frombytes(bytes(v1))
>>> v1_clone
Vector([3.0, 4.0, 5.0])
>>> v1 == v1_clone
True

```

测试序列行为::

```

>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[len(v1)-1], v1[-1]
(3.0, 5.0, 5.0)

```

测试切片::

```

>>> v7 = Vector(range(7))
>>> v7[-1]
6.0

```

```
>>> v7[1:4]
Vector([1.0, 2.0, 3.0])
>>> v7[-1:]
Vector([6.0])
>>> v7[1,2]
Traceback (most recent call last):
...
TypeError: 'tuple' object cannot be interpreted as an integer
```

测试动态属性访问::

```
>>> v7 = Vector(range(10))
>>> v7.x
0.0
>>> v7.y, v7.z, v7.t
(1.0, 2.0, 3.0)
```

动态属性查找失败情况::

```
>>> v7.k
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 'k'
>>> v3 = Vector(range(3))
>>> v3.t
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 't'
>>> v3.spam
Traceback (most recent call last):
...
AttributeError: 'Vector' object has no attribute 'spam'
```

测试哈希::

```
>>> v1 = Vector([3, 4])
>>> v2 = Vector([3.1, 4.2])
>>> v3 = Vector([3, 4, 5])
>>> v6 = Vector(range(6))
>>> hash(v1), hash(v3), hash(v6)
(7, 2, 1)
```

大多数非整数的哈希码在32位和64位CPython中不一样::

```
>>> import sys
>>> hash(v2) == (384307168202284039 if sys.maxsize > 2**32
```



```
else 357915986)
    True
```

测试使用``format()``格式化二维笛卡尔坐标::

```
>>> v1 = Vector([3, 4])
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

测试使用``format()``格式化三维和七维笛卡尔坐标::

```
>>> v3 = Vector([3, 4, 5])
>>> format(v3)
'(3.0, 4.0, 5.0)'
>>> format(Vector(range(7)))
'(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)'
```

测试使用``format()``格式化二维、三维和四维球面坐标::

```
>>> format(Vector([1, 1]), 'h') # doctest:+ELLIPSIS
'<1.414213..., 0.785398...>'
>>> format(Vector([1, 1]), '.3eh')
'<1.414e+00, 7.854e-01>'
>>> format(Vector([1, 1]), '0.5fh')
'<1.41421, 0.78540>'
>>> format(Vector([1, 1, 1]), 'h') # doctest:+ELLIPSIS
'<1.73205..., 0.95531..., 0.78539...>'
>>> format(Vector([2, 2, 2]), '.3eh')
'<3.464e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 0, 0]), '0.5fh')
'<0.00000, 0.00000, 0.00000>'
>>> format(Vector([-1, -1, -1, -1]), 'h') # doctest:+ELLIPSIS
'<2.0, 2.09439..., 2.18627..., 3.92699...>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
"""
```

```
from array import array
import reprlib
import math
import functools
import operator
```

```

import itertools ❶

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        return iter(self._components)

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('['):-1]
        return f'Vector({components})'

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))

    def __hash__(self):
        hashes = (hash(x) for x in self)
        return functools.reduce(operator.xor, hashes, 0)

    def __abs__(self):
        return math.hypot(*self)

    def __bool__(self):
        return bool(abs(self))

    def __len__(self):
        return len(self._components)

    def __getitem__(self, key):
        if isinstance(key, slice):
            cls = type(self)
            return cls(self._components[key])
        index = operator.index(key)
        return self._components[index]

    __match_args__ = ('x', 'y', 'z', 't')

```

```

def __getattr__(self, name):
    cls = type(self)
    try:
        pos = cls.__match_args__.index(name)
    except ValueError:
        pos = -1
    if 0 <= pos < len(self._components):
        return self._components[pos]
    msg = f'{cls.__name__!r} object has no attribute {name!r}'
    raise AttributeError(msg)

def angle(self, n): ❷
    r = math.hypot(*self[n:])
    a = math.atan2(r, self[n-1])
    if (n == len(self) - 1) and (self[-1] < 0):
        return math.pi * 2 - a
    else:
        return a

def angles(self): ❸
    return (self.angle(n) for n in range(1, len(self)))

def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('h'): # 超球面坐标
        fmt_spec = fmt_spec[:-1]
        coords = itertools.chain([abs(self)],
                                self.angles()) ❹

        outer_fmt = '<{}>' ❺
    else:
        coords = self
        outer_fmt = '({})' ❻
    components = (format(c, fmt_spec) for c in coords) ❼
    return outer_fmt.format(', '.join(components)) ❽

@classmethod
def frombytes(cls, octets):
    typecode = chr(octets[0])
    memv = memoryview(octets[1:]).cast(typecode)
    return cls(memv)

```

❶ 为了在 `__format__` 方法中使用 `chain` 函数，导入 `itertools` 模块。

❷ 使用“ n 维球体”词条中的公式计算某个角坐标。

❸ 创建生成器表达式，按需计算所有角坐标。

- ④ 使用 `itertools.chain` 函数生成**生成器表达式**，无缝迭代向量的模和各个角坐标。
- ⑤ 配置使用尖括号显示球面坐标。
- ⑥ 配置使用圆括号显示笛卡儿坐标。
- ⑦ 创建生成器表达式，按需格式化各个坐标元素。
- ⑧ 把以逗号分隔的格式化分量放入尖括号或圆括号内。



本书在 `__format__`、`angle` 和 `angles` 中大量使用了生成器表达式，不过这样做的目的是让 `Vector` 类的 `__format__` 方法与 `Vector2d` 类处在同一水平上。第 17 章在讨论生成器时会使用 `Vector` 类中的部分代码举例，详细说明生成器的技巧。

本章的任务到此结束。第 16 章将改进 `Vector` 类，让它支持中缀运算符。本章的目的是探讨如何编写容器类广泛使用的几个特殊方法。

12.9 本章小结

本章所举的 `Vector` 示例故意与 `Vector2d` 兼容，不过二者的构造函数签名不同。`Vector` 类的构造函数接受一个可迭代对象，这与内置序列类型一样。`Vector` 的行为之所以像序列，是因为它实现了 `__getitem__` 方法和 `__len__` 方法。借此，我们讨论了协议，这是鸭子类型语言使用的非正式接口。

然后，本章说明了 `my_seq[a:b:c]` 句法背后的原理：创建 `slice(a, b, c)` 对象，交给 `__getitem__` 方法处理。了解这一点之后，我们让 `Vector` 正确处理切片，像符合 Python 风格的序列那样返回新的 `Vector` 实例。

接下来，本章为 `Vector` 实例的头几个分量提供了只读访问功能，使用的是 `my_vec.x` 这样的表示法。这个功能通过 `__getattr__` 方法实现。实现这一功能之后，用户会想通过 `my_vec.x = 7` 这样的写法为头几个分量赋值——这是一个潜在的 bug。为了解决这个问题，本章又实现了 `__setattr__` 方法，通过它禁止为单字母属性赋值。大多数时候，如果定义了 `__getattr__` 方法，那么也要定义 `__setattr__` 方法，这样才能避免行为不一致。

实现 `__hash__` 方法特别适合使用 `functools.reduce` 函数，因为要把异或运算符 $\wedge$ 依次应用到各个分量的哈希值上，生成整个向量的聚合哈希值。在 `__hash__` 方法中使用 `reduce` 函数之后，我们又使用内置归约函数 `all` 实现了效率更高的 `__eq__` 方法。

`Vector` 类的最后一项改进是在 `Vector2d` 的基础上重新实现 `__format__` 方法，这一次，除了支持笛卡儿坐标，还支持了球面坐标。为了定义 `__format__` 方法及其辅助方法，本章用到了很多数学知识和几个生成器，但这些是实现细节。第 17 章会再次讨论生成器。12.7 节的目的是支持自定义格式，从而兑现承诺，让 `Vector` 与 `Vector2d` 兼容，此外还能做更多的事情。

与第 11 章一样，本章经常分析 Python 标准对象的行为，然后模仿，让 `Vector` 的行为符合 Python 风格。

第 16 章将为 `Vector` 实现几个中缀运算符，这一章使用的数学知识比 `angle()` 方法用到的简单多了，但是通过了解 Python 中缀运算符的工作方式，我们对面向对象设计的认识将更进一步。讨论运算符重载之前，让我们暂且告别单个类，来说明一下如何使用接口和继承组织多个类，详见第 13 章和第 14 章。

12.10 延伸阅读

`Vector` 类中的大多数特殊方法在第 11 章定义的 `Vector2d` 类中也有，因此 11.14 节给出的延伸阅读材料同样适合本章。

强大的高阶函数 `reduce` 的作用也可称为合拢、累计、聚合、压缩和注入。更多信息可以参见维基百科中的“Fold (higher-order function)”，这篇文章展示了高阶函数的用途，着重说明了具有递归数据结构的函数式语言。另外，这篇文章中还有一张表格，列出了在很多编程语言中起合拢作用的函数。

“What's New in Python 2.5”简略概括了 `__index__` 的作用：支持 `__getitem__` 方法（参见 12.5.2 节）。“PEP 357—Allowing Any Object to be Used for Slicing”站在一个 C 语言扩展实现者（NumPy 的主要创建人 Travis Oliphant）的角度详细说明了需要 `__index__` 的原因。Oliphant 对 Python 贡献颇多，使 Python 成为领先的科学计算语言，进而使 Python 在机器学习应用程序中处于领先地位。

杂谈

把协议当作非正式接口

协议不是 Python 发明的。Smalltalk 团队，也就是“面向对象”的发明者，使用“协议”这个词表示现在我们称之为接口的功能。某些 Smalltalk 编程环境允许程序员把一组方法标记为协议，但这只不过是一种文档，用于辅助导航，语言不对其施加特定措施。因此，向熟悉正式接口（编译器会施加措施）的人解释“协议”时，我会简单地说它是“非正式接口”。

动态类型语言中的既定协议会自然进化。所谓动态类型是指在运行时检查类型，因为方法签名和变量没有静态类型信息。Ruby 是一门重要的面向对象动态类型语言，它也使用协议。

在 Python 文档中，如果看到“文件类对象”这样的表述，通常说的就是协议。这是一种简短的说法，意思是：“行为基本与文件一致，实现了部分文件接口，满足上下文相关需求的东西。”

你可能觉得只实现协议的一部分不够严谨，但是这样做的优点是简单。《Python 语言参考手册》中的 3.3 节给出了如下建议。

模仿内置类型实现类时，记住一点：模仿的程度对建模的对象来说合理即可。例如，有些序列可能只需要获取单个元素，而不必提取切片。

不要为了满足过度设计的接口契约以及让编译器开心而去实现不必要的方法，要遵守 KISS 原则。

然而，如果想让类型检查工具验证协议的实现，就要严格定义协议。此时，可以使用 `typing.Protocol`。

第 13 章还会讨论协议和接口，这正是那一章的主要话题。

鸭子类型的起源

我相信，Ruby 社区在“鸭子类型”这个术语的推广过程中起到了主要作用，他们向大量 Java 使用者宣扬了这个说法。但是，在 Ruby 或 Python“流行”起来之前，Python 就使用这种表述了。根据维基百科，在面向对象编程中较早使用鸭子做比喻的人是 Alex Martelli，这个比喻出现在他于 2000 年 7 月 26 日发到 Python-list 中的一条消息里，即“polymorphism (was Re: Type checking in python?)”。本章开头引用的那句话就出自那条消息。如果想知道“鸭子类型”这个术语的真正起源，以及很多编程语言对这个面向对象概念的应用，请阅读维基百科中的“Duck typing”词条。

安全的 `__format__` 方法，增强可用性

实现 `__format__` 方法时，我没有采取措施防范 `Vector` 实例拥有大量分量，不过在 `__repr__` 方法中我使用 `reprlib` 做了预防。这是因为，`repr()` 函数用于调试和记录日志，所以必须生成可用的输出。而 `__format__` 方法用于向终端用户显示输

出，这些用户应该想看到整个 `Vector`。如果你觉得这样做危险，那么可以再为格式规范微语言实现一个扩展。

如果是我，我会这么做：默认情况下，格式化的 `Vector` 实例显示有限个分量，比如说 30 个。如果元素数量超过上限，默认的行为是像 `reprlib` 那样，截断超出的部分，就使用 `...` 表示。然而，如果格式说明符末尾是特殊的 `*` 代码（意思是“全部”），则不限制显示的元素数量。因此，用户在不知情的情况下不会被特别长的输出吓倒。如果默认的上限碍事，那么 `...` 的存在对用户是个提醒，用户研究文档后会发现 `*` 格式代码。

寻找符合 Python 风格的求和方式

就像“什么是美”没有确切答案一样，“什么是 Python 风格”也没有标准答案。如果回答“地道的 Python”（我通常会这样说），则不能让人 100% 满意，因为对你来说是“地道的”，在我看来却可能不是。但我可以肯定的是，“地道”并不是指使用最鲜为人知的语言功能。

Python-list 中有一篇发表于 2003 年 4 月的文章，题为“Pythonic Way to Sum n-th List Element?”。这篇文章与本章讨论的 `reduce` 函数有关。

该话题的发起人 Guy Middleton 说他不喜欢使用 `lambda` 表达式，问下面的方案有没有办法改进。⁷

```
>>> my_list = [[1, 2, 3], [40, 50, 60], [9, 8, 7]]
>>> import functools
>>> functools.reduce(lambda a, b: a+b, [sub[1] for sub in
my_list])
60
```

这段代码有很多地道的用法：`lambda`、`reduce` 和列表推导式。最终，这可能会变成人气竞赛，因为它冒犯了讨厌 `lambda` 的人和看不上列表推导式的人——这两种人可都不少。

如果打算使用 `lambda`，那么或许就不应该使用列表推导式——筛选除外，但这里不是筛选。

下面是我给出的方案，这能讨得 `lambda` 拥护者的欢心。

```
>>> functools.reduce(lambda a, b: a + b[1], my_list, 0)
60
```

我没有参与那个话题，而且不会在真实的代码中使用上述方案，因为我非常不喜欢 `lambda` 表达式。这里只是为了举例说明不使用列表推导式可以怎么做。

第一个答复是 Fernando Perez 给出的，他是 IPython 的创建者，他的答案强调了 NumPy 支持 n 维数组和 n 维切片。

```
>>> import numpy as np
>>> my_array = np.array(my_list)
>>> np.sum(my_array[:, 1])
60
```

我觉得 Perez 的方案很棒，不过 Guy Middleton 推崇 Paul Rubin 和 Skip Montanaro 给出的下述方案。

```
>>> import operator
>>> functools.reduce(operator.add, [sub[1] for sub in
my_list], 0)
60
```

随后，Evan Simpson 问道：“这样做有什么错？”

```
>>> total = 0
>>> for sub in my_list:
...     total += sub[1]
...
>>> total
60
```

许多人觉得这也很符合 Python 风格。Alex Martelli 甚至说，Guido 或许就会这么做。

我喜欢 Evan Simpson 的代码，不过也喜欢 David Eppstein 对此给出的评论。

如果想计算列表中各个元素的和，那么写出的代码就应该看起来像是在“计算元素之和”，而不是“迭代元素，维护一个变量 t ，再执行一系列求和操作”。如果不能站在一定高度上表明意图，却让语言去关注底层操作，那还要高级语言干吗？

而后，Alex Martelli 又给出了如下建议。

“求和”是常见操作，我不介意 Python 提供这样的一个内置函数。但是，在我看来，“`reduce(operator.add,...)`”不是好方法。（作为一个读过 *A Programming Language* 的老程序员和函数式语言爱好者，我**应该**喜欢这个方法，但是我并不喜欢。）

随后，Alex 建议提供并实现了 `sum()` 函数。这次讨论之后 3 个月，Python 2.3 就内置了这个函数。因此，Alex 喜欢的句法变成了标准。

```
>>> sum([sub[1] for sub in my_list])
60
```

下一年年末（2004 年 11 月），Python 2.4 发布了，这一版引入了生成器表达式。在我看来，Guy Middleton 那个问题目前最符合 Python 风格的答案如下。

```
>>> sum(sub[1] for sub in my_list)
60
```

这样写不仅比使用 `reduce` 函数更容易理解，而且还能避免空序列导致的陷阱：`sum([])` 的结果是 0，就这么简单。

在这次讨论中，Alex Martelli 指出，Python 2 内置的 `reduce` 函数“成事不足败事有余”，因为它推荐的地道编程方式难以理解。

他的观点最有说服力，因此 Python 3 把 `reduce` 函数移到了 `functools` 模块中。

当然，`functools.reduce` 函数仍有用武之地。实现 `Vector.__hash__` 方法时我就用了它，我觉得我的实现方式算得上符合 Python 风格。

⁷为了在此展示，我稍微修改了这段代码，因为在 2003 年，`reduce` 是内置函数，而在 Python 3 中要导入。此外，我把 `x` 和 `y` 两个名称换成了 `my_list` 和 `sub`（表示子串）。

第 13 章 接口、协议和抽象基类

对接口编程，而不是对实现编程。

——Gamma、Helm、Johnson 和 Vlissides
面向对象设计第一原则¹

¹出自《设计模式》“引言”部分。

面向对象编程全靠接口。如 8.4 节所述，在 Python 中，支撑一个类型的是它提供的方法，也就是接口。

在不同的编程语言中，接口的定义和使用方式不尽相同。从 Python 3.8 开始，有 4 种方式，如图 13-1 中的**类型图**所示。这 4 种方式概述如下。

鸭子类型

自 Python 诞生以来默认使用的类型实现方式。从第 1 章开始，本书一直在研究鸭子类型。

大鹅类型

自 Python 2.6 开始，由抽象基类支持的方式，该方式会在运行时检查对象是否符合抽象基类的要求。大鹅类型是本章的主要话题。

静态类型

C 和 Java 等传统静态类型语言采用的方式。自 Python 3.5 开始，由 `typing` 模块支持，由符合“PEP 484—Type Hints”要求的外部类型检查工具实施检查。本章不涉及该方式。第 8 章的大多数内容和第 15 章讨论了静态类型。

静态鸭子类型

因 Go 语言而流行的方式。由 `typing.Protocol` (Python 3.8 新增) 的子类支持, 由外部类型检查工具实施检查。静态鸭子类型首次出现在 8.5.10 节。

13.1 类型图

图 13-1 描述的 4 种类型实现方式各有优缺点, 相辅相成, 缺一不可。

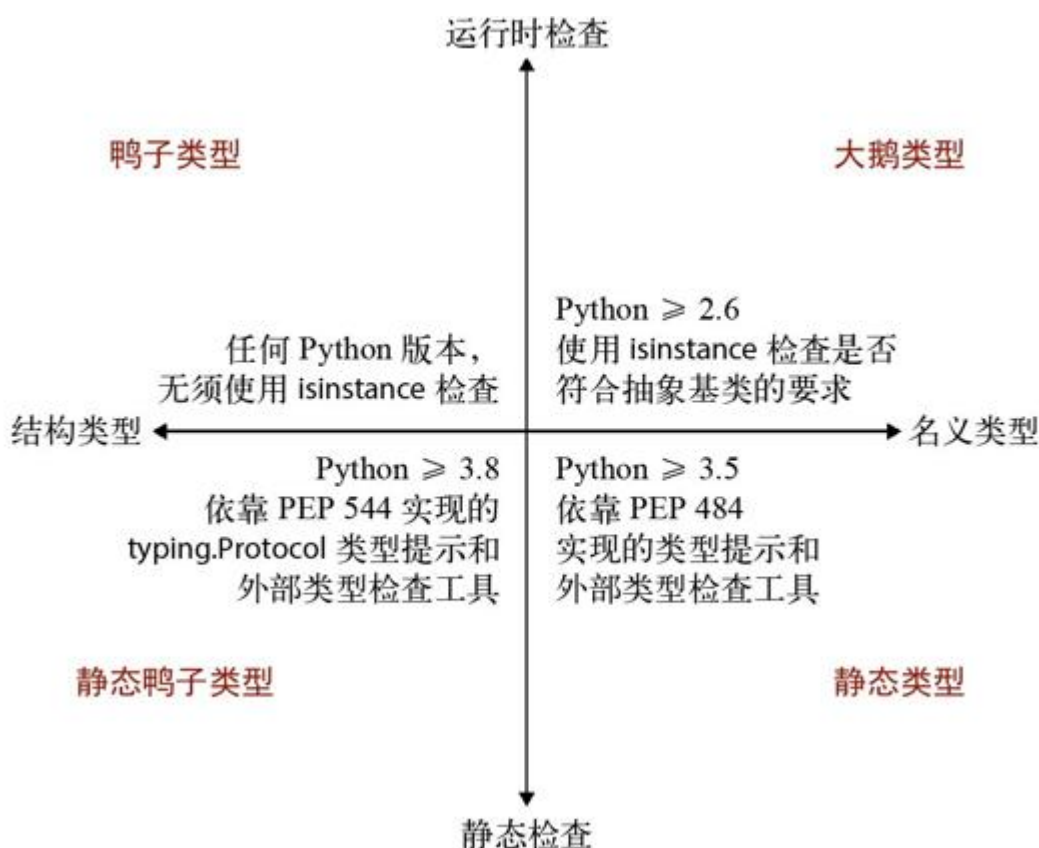


图 13-1: 上半部分是只使用 Python 解释器在运行时检查类型的方式; 下半部分则要借助外部静态类型检查工具, 例如 MyPy 或 PyCharm 等 IDE。左边两象限中的类型基于对象的结构 (对象提供的方法), 与对象所属的类或超类无关; 右边两象限中的类型要求对象有明确的类型名称: 对象所属类的名称, 或者超类的名称

这 4 种方式全都依靠接口，不过静态类型可以只使用具体类型实现（效果差），而不使用协议和抽象基类等接口抽象。本章涵盖围绕接口实现的 3 种类型：鸭子类型、大鹅类型和静态鸭子类型。

本章主要分为 4 部分，探讨类型图（参见图 13-1）中 4 个象限里的 3 个。

- 13.3 节比较两种依赖协议的结构类型，即类型图的左半部分。
- 13.4 节深入探讨我们熟悉的鸭子类型，以及如何在保证灵活性这个主要优势的前提下提升鸭子类型的安全性。
- 13.5 节说明如何使用抽象基类实现更严格的运行时类型检查。这一节内容最多，原因不是大鹅类型更重要，而是本书其他章已经涵盖鸭子类型、静态鸭子类型和静态类型。
- 13.6 节涵盖 `typing.Protocol` 子类（针对静态和运行时类型检查）的用法、实现和设计。

13.2 本章新增内容

本章内容改动幅度较大，与第 1 版中的第 11 章相比，内容增加了约 24%。虽然一些小节和很多段落没有变，但是新增了大量内容。主要变化如下。

- 本章的导言和类型图（参见图 13-1）是新增的。这是本章很多新增内容，以及与 Python 3.8 及以上版本中类型相关的其他章的关键。
- 13.3 节说明动态协议和静态协议之间的异同。
- 13.4.3 节的内容基本上来自本书第 1 版，不过做了更新，为了突出重要性，标题也改了。
- 13.6 节是全新内容，是 8.5.10 节的延续。
- 更新了图 13-2、图 13-3 和图 13-4 中的 `collections.abc` 类图，增加 Python 3.6 引入的抽象基类 `Collection`。

本书第 1 版有一节建议使用 `numbers` 模块中的抽象基类实现大鹅类型。实现大鹅类型时，除了运行时检查，如果还打算使用静态类型检查工具，则应该换用 `typing` 模块中的数值静态协议。13.6.8 节会解释背后的原因。

13.3 两种协议

在计算机科学中，根据上下文，“协议”一词有不同的含义。HTTP 这种网络协议指明了客户端可向服务器发送的命令，例如 `GET`、`PUT` 和 `HEAD`。12.4 节讲过，对象协议指明为了履行某个角色，对象必须实现哪些方法。第 1 章中的 `FrenchDeck` 示例演示了一个对象协议，即序列协议：一个 Python 对象想表现得像一个序列需要提供的方法。

完全实现一个协议可能需要多个方法，不过，通常可以只实现部分协议。下面以示例 13-1 中的 `Vowels` 类为例。

示例 13-1 使用 `__getitem__` 方法实现部分序列协议

```
>>> class Vowels:
...     def __getitem__(self, i):
...         return 'AEIOU'[i]
...
>>> v = Vowels()
>>> v[0]
'A'
>>> v[-1]
'U'
>>> for c in v: print(c)
...
A
E
I
O
U
>>> 'E' in v
True
>>> 'Z' in v
False
```

只要实现 `__getitem__` 方法，就可以按索引获取项，以及支持迭代和 `in` 运算符。其实，特殊方法 `__getitem__` 是序列协议的核心。以下内容摘自《Python/C API 参考手册》中的“序列协议”一节。

```
int PySequence_Check(PyObject *o)
```

如果对象提供序列协议，就返回 1，否则返回 0。注意，除了 `dict` 子类，如果一个 Python 类有 `__getitem__()` 方法，则也返回 1.....

我们预期序列支持 `len()` 函数，也就是要实现 `__len__` 方法。`Vowels` 没有 `__len__` 方法，不过在某些上下文中依然算得上是序列。而有些时候，这就足够了。所以，我经常说协议是“非正式接口”。第一个使用“协议”这个术语的面向对象编程环境 `Smalltalk` 也是这么理解协议的。

在 Python 文档中，除了有关网络编程的内容，“协议”一词基本上是指非正式接口。

Python 3.8 通过“PEP 544—Protocols: Structural subtyping (static duck typing)”之后，“协议”一词在 Python 中多了一种含义——联系紧密，仍有区别。8.5.10 节讲过，PEP 544 提议通过 `typing.Protocol` 的子类定义一个类必须实现（或继承）的一个或多个方法，让静态类型检查工具满意。

需要区分时，我会使用以下两个术语。

动态协议

Python 一直有的非正式协议。动态协议是隐含的，按约定定义，在文档中描述。Python 大多数重要的动态协议由解释器支持，在《Python 语言参考手册》的第 3 章“数据模型”中说明。

静态协议

“PEP 544—Protocols: Structural subtyping (static duck typing)”定义的协议，自 Python 3.8 开始支持。静态协议要使用 `typing.Protocol` 子类显式定义。

二者之间的主要区别如下。

- 对象可以只实现动态协议的一部分，但是如果想满足静态协议，则对象必须提供协议类中声明的每一个方法，即使程序用不到。
- 静态协议可以使用静态类型检查工具确认，动态协议则不能。

两种协议共有一个基本特征：类无须通过名称（例如通过继承）声明支持什么协议。

除了静态协议，Python 还提供了另一种定义显式接口的方式，即抽象基类。

本章余下的内容涵盖动态协议和静态协议，以及抽象基类。

13.4 利用鸭子类型编程

我们以 Python 中两个最重要的协议（序列协议和可迭代协议）为例展开对动态协议的讨论。即使对象只实现了这些协议的最少一部分，也会引起解释器的注意。详见 13.4.1 节。

13.4.1 Python 喜欢序列

Python 数据模型的哲学是尽量支持基本的动态协议。对序列来说，即便是最简单的实现，Python 也会力求做到最好。

图 13-2 展示的是通过一个抽象基类确立的 `Sequence` 接口。Python 解释器和 `list`、`str` 等内置序列根本不依赖那个抽象基类。我只是利用它说明一个功能完善的序列应该支持什么操作。



图 13-2: Sequence 抽象基类和 collections.abc 中相关抽象类的 UML 类图。箭头由子类指向超类。以斜体显示的是抽象方法。Python 3.6 之前的版本中没有 Collection 抽象基类, Sequence 是 Container、Iterable 和 Sized 的直接子类



collections.abc 模块中的大多数抽象基类存在的目的是确立由内置对象实现并且由解释器隐式支持的接口——二者均早于抽象基类。这些抽象基类可作为新类的基础, 并为运行时显式类型检查 (**大鹅类型**) 和静态类型检查工具用到的类型提示提供支持。

从图 13-2 可以看出, 为了确保行为正确, Sequence 的子类必须实现 `__getitem__` 和 `__len__` (来自 Sized)。Sequence 中的其他方法都是具体的, 因此子类可以继承或者提供更好的实现。

再回顾一下示例 13-1 中的 Vowels 类。那个类没有继承 `abc.Sequence`, 而且只实现了 `__getitem__`。

虽然没有 `__iter__` 方法, 但是 Vowels 实例仍然可以迭代。这是因为如果发现有 `__getitem__` 方法, 那么 Python 就会调用它, 传入从 0 开始的整数索引, 尝试迭代对象 (这是一种后备机制)。尽管缺少 `__contains__` 方法, 但是 Python 足够智能, 能正确迭代 Vowels 实例, 因此也能使用 `in` 运算符: Python 做全面检查, 判断指定的项是否存在。

综上所述，鉴于序列类数据结构的重要性，如果没有 `__iter__` 方法和 `__contains__` 方法，则 Python 会调用 `__getitem__` 方法，设法让迭代和 `in` 运算符可用。

第 1 章定义的 `FrenchDeck` 类也没有继承 `abc.Sequence`，但是实现了序列协议的两个方法：`__getitem__` 和 `__len__`。如示例 13-2 所示。

示例 13-2 一摞有序的纸牌（与示例 1-1 相同）

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

第 1 章中的那些示例之所以能用，是因为 Python 会特殊对待看起来像序列的对象。Python 的迭代协议是鸭子类型的一种极端形式：为了迭代对象，解释器会尝试调用两个不同的方法。

需要明确指出的是，本节描述的行为在解释器自身中实现，大多数是用 C 语言实现的，不依赖 `Sequence` 抽象基类的方法。例如，`Sequence` 类中的具体方法 `__iter__` 和 `__contains__` 是对 Python 解释器内置行为的模仿。如果觉得好奇，可以到 `Lib/_collections_abc.py` 文件中阅读这些方法的源码。

下面再分析一个示例，强调协议的动态本性，并解释静态类型检查工具为什么没机会处理动态协议。

13.4.2 使用猴子补丁在运行时实现协议

猴子补丁在运行时动态修改模块、类或函数，以增加功能或修正 bug。例如，网络库 `gevent` 对部分 Python 标准库打了猴子补丁，不借助线程或 `async/await` 实现一种轻量级并发。

示例 13-2 中的 `FrenchDeck` 类缺少一个重要功能：无法洗牌。几年前，我在第一次编写 `FrenchDeck` 示例时实现了 `shuffle` 方法。后来，在对 Python 风格有了深刻理解后我发现，既然 `FrenchDeck` 的行为像序列，那么它就不需要 `shuffle` 方法，因为有现成的 `random.shuffle` 函数可用。根据文档，该函数的作用是“就地打乱序列 `x`”。

标准库中的 `random.shuffle` 函数用法如下所示。

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 2, 9, 7, 8, 3, 1, 4, 0, 6]
```



遵守既定协议很有可能增加利用现有标准库和第三方代码的可能性，这得益于鸭子类型。

然而，如果尝试打乱 `FrenchDeck` 实例，则会出现异常，如示例 13-3 所示。

示例 13-3 `random.shuffle` 函数不能打乱 `FrenchDeck` 实例

```
>>> from random import shuffle
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> shuffle(deck)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

错误消息相当明确：'FrenchDeck' object does not support item assignment ('FrenchDeck' 对象不支持为项赋值)。这个问题的原因在于，shuffle 函数会就地操作，调换容器内项的位置，而 FrenchDeck 只实现了不可变序列协议。可变序列还必须提供 `__setitem__` 方法。

因为 Python 是动态语言，所以可以在运行时修正这个问题，甚至在交互式控制台中就能做到。修正方法如示例 13-4 所示。

示例 13-4 为 FrenchDeck 打猴子补丁，把它变成可变序列，以使 random.shuffle 函数能够对其进行处理（接续示例 13-3）

```
>>> def set_card(deck, position, card): ❶
...     deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card ❷
>>> shuffle(deck) ❸
>>> deck[:5]
[Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'),
Card(rank='4',
suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9',
suit='spades')]
```

- ❶ 定义一个函数，参数为 deck、position 和 card。
- ❷ 把上述函数赋值给 FrenchDeck 类的 `__setitem__` 属性。
- ❸ 现在可以打乱 deck 了，因为我添加了可变序列协议所需的方法。

《Python 语言参考手册》的 3.3.7 节定义了特殊方法 `__setitem__` 的签名。该手册中使用的参数是 `self`, `key`, `value`，而这里使用

的是 `deck`, `position`, `card`。这么做是为了告诉你，Python 方法说到底就是普通函数，把第一个参数命名为 `self` 只是一种约定。在控制台会话中使用那几个参数没问题，不过在 Python 源码文件中最好按照文档那样使用 `self`、`key` 和 `value`。

这里的关键是，`set_card` 函数要知道 `deck` 对象有一个名为 `_cards` 的属性，而且 `_cards` 的值必须是可变序列。然后，我们把 `set_card` 函数依附到 `FrenchDeck` 类上，作为特殊方法 `__setitem__`。这就是**猴子补丁**：在运行时修改类或模块，而不改动源码。虽然猴子补丁很强大，但是打补丁的代码与被打补丁的程序耦合十分紧密，而且往往要处理文档没有明确说明的私有属性。

除了举例说明猴子补丁，示例 13-4 还强调了动态鸭子类型中的协议是动态的：`random.shuffle` 函数不关心参数所属的类，只要那个对象实现了可变序列协议的方法即可。即便对象一开始没有所需的方法也没关系，可以之后再提供。

鸭子类型的安全性看似不可控，而且增加了调试难度，其实不然。13.4.3 节将介绍一些检测动态协议的编程模式，免得我们自己动手检查。

13.4.3 防御性编程和“快速失败”

防御性编程就像防御性驾驶：有一套提高安全的实践，即使是粗心的程序员（或司机）也不会造成灾难。

许多 bug 只有在运行时才能捕获，即使主流的静态类型语言也是如此。² 对于动态类型语言，“快速失败”可以提升程序的安全性，让程序更易于维护。快速失败的意思是尽早抛出运行时错误，例如，在函数主体开头就拒绝无效的参数。

²因此需要自动化测试。

如果一个函数接受一系列项，在内部按照列表处理，那么就不要再通过类型检查强制要求传入一个列表。正确的做法是立即利用参数构建一个列表。示例 13-10 中的 `__init__` 方法就采用了这种编程模式。

```
def __init__(self, iterable):  
    self._balls = list(iterable)
```

这样写出的代码更灵活，因为 `list()` 构造函数能处理任何在内存中放得下的可迭代对象。如果传入的参数不是可迭代对象，那么初始化对象时 `list()` 调用就会快速失败，抛出意义十分明确的 `TypeError` 异常。如果想更明确一些，可以把 `list()` 调用放在 `try/except` 结构中，自定义错误消息。我只会在外部 API 中这么做，因为这样方便基准代码维护人员发现问题。无论如何，出错的调用将出现在调用跟踪的末尾，直指根源。如果没有在类的构造方法中捕获无效参数，而等到类中的其他方法需要操作 `self._balls` 时才发现它不是列表，那就为时已晚了，程序崩溃的根源将很难确定。

当然，如果数据太多，或者按照设计，需要就地修改数据（例如 `random.shuffle`）以满足调用方的利益，那么调用 `list()` 复制数据就不是一个好主意。遇到这种情况，应该使用 `isinstance(x, abc.MutableSequence)` 做运行时检查。

如果害怕传入的是无穷生成器（不常见），则可以先使用 `len()` 获取参数的长度。这样可以拒绝迭代器，安全处理元组、数组，以及其他现有或以后可能出现的实现 `Sequence` 接口的类。调用 `len()` 的开销通常不大，但是作用明显，遇到无效参数会立即抛出错误。

另外，如果接受任何可迭代对象，那么要尽早调用 `iter(x)`，获得一个迭代器（详见 17.3 节）。同样，如果 `x` 不是可迭代对象，则这也会快速失败，抛出一个易于调试的异常。

在这两种情况下，类型提示可以捕获部分问题，但不是所有问题都能提前获知。还记得吗？`Any` 类型与任何类型都**相容**。经推导得出的类

型就可能是 Any。这时，类型检查工具发挥不了什么作用。而且，类型提示不在运行时强制检查。快速失败是最后一道防线。

利用鸭子类型做防御性编程，无须使用 `isinstance()` 或 `hasattr()` 测试就能处理不同的类型。

再举一个例子，模仿 `collections.namedtuple` 处理 `field_names` 参数的方式。`field_names` 参数既可以是各个标识符以空格或逗号分隔的字符串，也可以是标识符序列。利用鸭子类型，可以像示例 13-5 那样处理。

示例 13-5 利用鸭子类型处理一个字符串或由字符串构成的可迭代对象

```
try: ❶
    field_names = field_names.replace(',', ' ').split() ❷
except AttributeError: ❸
    pass ❹
field_names = tuple(field_names) ❺
if not all(s.isidentifier() for s in field_names): ❻
    raise ValueError('field_names must all be valid
identifiers')
```

❶ 假设是一个字符串（EAFP 原则：取得原谅比获得许可容易）。

❷ 把逗号替换成空格，再拆分成名称列表。

❸ 抱歉，`field_names` 的行为不像是字符串：没有 `.replace` 方法，或者返回的结果无法拆分。

❹ 如果抛出 `AttributeError`，说明 `field_names` 不是字符串，那就假设 `field_names` 是由名称构成的可迭代对象。

❺ 为了确保是可迭代对象，也为了留存一份副本，根据现有数据创建一个元组。元组比列表紧凑，还能防止代码意外改动名称。

⑥ 使用 `str.isidentifier` 确保每个名称都是有效的标识符。

示例 13-5 展示的情况说明，有时鸭子类型比静态类型提示更具表现力。类型提示无法表达“`field_names` 必须是一个字符串，各个标识符以空格或逗号分隔。”在 `typeshed` 项目中，`namedtuple` 签名的相关部分如下所示（完整源码见 `stdlib/3/collections/__init__.pyi`）。

```
def namedtuple(
    typename: str,
    field_names: Union[str, Iterable[str]],
    *,
    # 余下的签名省略了
```

可以看到，`field_names` 的类型注解是 `Union[str, Iterable[str]]`。这个注解基本够用，但是不能捕获所有问题。

研究过动态协议之后，接下来换个话题，讨论运行时类型检查更为外显的一种形式，即大鹅类型。

13.5 大鹅类型

抽象类表示接口。

——Bjarne Stroustrup
C++ 之父³

³出自《C++ 语言的设计和演化》。

Python 没有 `interface` 关键字。我们使用抽象基类定义接口，在运行时显式检查类型（静态类型检查工具也支持）。

在 Python 术语表中，“抽象基类”词条很好地解释了抽象基类为鸭子类型语言带来的好处。

抽象基类是对鸭子类型的补充，提供了一种定义接口的方式。相比之下，其他技术（例如 `hasattr()`）则显得笨拙或者不太正确（例如使用魔法方法）。抽象基类引入了虚拟子类，这种类不继承其他类，却能被 `isinstance()` 和 `issubclass()` 识别。详见 `abc` 模块文档。⁴

⁴2020 年 10 月 18 日摘录。

大鹅类型是一种利用抽象基类实现的运行时检查方式。接下来将由 Alex Martelli 详细说明，参见“水禽和抽象基类”附注栏。



非常感谢我的朋友 Alex Martelli 和 Anna Ravenscroft。我在 OSCON 2013 上把本书的原始提纲给二位看时，他们鼓励我提交给 O'Reilly 出版。后来，他们还担任了本书的技术审校，对本书做了全面审查。本书引用最多的就是 Alex 的话。他甚至提出撰写下面这篇短文。接下来交给你了，Alex！

水禽和抽象基类

作者：Alex Martelli

维基百科上说是我协助传播了**鸭子类型**（忽略对象的真正类型，转而关注对象有没有实现所需的方法、签名和语义）这种言简意赅的说法。

对 Python 来说，这基本上是指避免使用 `isinstance` 检查对象的类型（更别提 `type(foo) is bar` 这种更糟的检查方式了，这样做没有任何好处，甚至禁止最简单的继承方式）。

总的来说，**鸭子类型**在很多情况下十分有用。但是在其他情况下，随着发展，通常有更好的方式。事情是这样的……

近代，属和种（包括但不限于水禽所属的鸭科）基本上是根据**表征学**（phenetics）分类的。表征学关注的是形态和举止的相似

性.....主要是**容易观察**的特征。因此使用“鸭子类型”做比喻是贴切的。

然而，平行进化往往会导致不相关的种产生相似的特征，形态和举止方面都是如此，但是生态龛位的相似性是偶然的，不同的种仍属不同的生态龛位。编程语言中也有这种“偶然的相似性”，比如下面这个经典的面向对象编程示例。

```
class Artist:
    def draw(self): ...

class Gunslinger:
    def draw(self): ...

class Lottery:
    def draw(self): ...
```

显然，只因为 `x` 和 `y` 这两个对象刚好都有一个名为 `draw` 的方法，而且调用时不用传入参数（例如 `x.draw()` 和 `y.draw()`），远远不能确保二者可以相互调用，或者具有相同的抽象。也就是说，从这样的调用中不能推导出语义相似性。相反，我们需要一位渊博的程序员**主动**把这种等价维持在一定层次上。

生物和其他学科遇到的这个问题，迫切需要（从很多方面来说，是催生）表征学之外的分类方式解决，这就引出了**支序学**（cladistics）。这种分类学主要根据从共同祖先那里继承的特征分类，而不是单独进化的特征。（近些年，DNA 测序变得既便宜又快速，这使支序学的实用地位变得更高。）

例如，草雁（以前认为与其他鹅类比较相似）和麻鸭（以前认为与其他鸭类比较相似）现在被分到麻鸭亚科（表明二者的相似性比鸭科中其他动物高，因为它们的共同祖先比较接近）。此外，DNA 分析表明，白翅木鸭与美洲家鸭（属于麻鸭）不是很像，至少没有形态和举止看起来那么像，因此把木鸭单独分成了一属，完全不在麻鸭亚科中。

知道这些有什么用呢？视情况而定！比如，逮到一只水禽后，决定如何烹制才最美味时，显著的特征（不是全部，例如一身羽毛并不重要）主要是口感和风味（过时的表征学），这比支序学重要得多。但在其他方面，如对不同病原体的抗性（圈养水禽还是放养），DNA 接近性的作用就大多了……

因此，参照水禽的分类学演化，我建议在**鸭子类型**的基础上补充（不是完全取代，因为在某些时候，鸭子类型还有它的作用）**大鹅类型**。

大鹅类型指的是，只要 `cls` 是抽象基类（`cls` 的元类是 `abc.ABCMeta`），就可以使用 `isinstance(obj, cls)`。

`collections.abc` 中有很多有用的抽象类（Python 标准库文档的 `numbers` 模块中还有一些）。⁵

与具体类相比，抽象基类有很多理论上的优点（参阅《More Effective C++：35 个改善编程与设计的有效方法（中文版）》一书中的“条款 33：将非尾端类设计为抽象类”）。Python 的抽象基类还有一个重要的实用优势：终端用户可以使用 `register` 类方法在代码中把某个类“声明”为一个抽象基类的“虚拟”子类。（为此，被注册的类必须满足抽象基类对方法名称和签名的要求，最重要的是要满足底层语义契约。但是，开发那个类时不用了解抽象基类，更不用继承抽象基类。）这大大打破了严格的强耦合，与面向对象编程人员掌握的知识有很大出入，因此使用继承时要小心。

有时，为了让抽象基类识别子类，甚至不用注册。

其实，抽象基类的本质就是几个特殊方法。

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
```

```
>>> isinstance(Struggle(), abc.Sized)
True
```

可以看出，无须注册，`abc.Sized` 也能把 `Struggle` 识别为自己的子类，只要实现了特殊方法 `__len__` 即可。（要使用正确的句法和语义实现，前者要求没有参数，后者要求返回一个非负整数，指明对象的“长度”。如果不使用规定的句法和语义实现诸如 `__len__` 之类的特殊方法，那么将导致非常严重的问题。）

最后我想说的是：如果实现的类体现了 `numbers`、`collections.abc` 或其他框架中抽象基类的概念，则要么继承相应的抽象基类（必要时），要么把类注册到相应的抽象基类中。开始开发程序时，不要使用提供注册功能的库或框架，要自己动手注册。作为最常见的情况，如果必须检查参数的类型（例如检查是不是“序列”），则可以像下面这样做。

```
isinstance(the_arg, collections.abc.Sequence)
```

此外，**不要**在生产代码中定义抽象基类（或元类）……如果很想这样做，我打赌可能是因为你“找碴儿”。刚拿到新工具的人都有大干一场的冲动。如果能避开这些深奥的概念，那么你（以及未来的代码维护人员）的生活将更愉快，因为代码会变得简洁明了。**再会！**

⁵当然，还可以自己定义抽象基类，但是不建议高级 Python 程序员之外的人这么做。同样，也不建议自己定义元类……我说的“高级 Python 程序员”是指对 Python 语言的一招一式都了如指掌的人。即便对这类人来说，抽象基类和元类也不是常用工具。如此“深层次的元编程”（如果可以这么讲的话），适合框架的作者使用，这样便于众多不同的开发团队独立扩展框架……真正需要这么做的“高级 Python 程序员”不超过 1%。——Alex Martelli

综上所述，**大鹅类型**要求：

- 定义抽象基类的子类，明确表明你在实现既有的接口；
- 运行时检查类型时，`isinstance` 和 `issubclass` 的第二个参数要使用抽象基类，而不是具体类。

Alex 指出，继承抽象基类其实就是实现必要的方法——这也明确表明了开发人员的意图。这个意图还可以通过注册虚拟子类明确表述。



`register` 的具体用法将在 13.5.6 节说明。这里先举一个简单的例子：对于 `FrenchDeck` 类，如果想通过 `issubclass(FrenchDeck, Sequence)` 检查，那么可以使用以下几行代码把 `FrenchDeck` 注册为抽象基类 `Sequence` 的虚拟子类。

```
from collections.abc import Sequence
Sequence.register(FrenchDeck)
```

使用 `isinstance` 和 `issubclass` 测试抽象基类（而不是具体类）更为人接受。如果用于测试具体类，则类型检查将限制多态——面向对象编程的一个重要功能。用于测试抽象基类更加灵活。毕竟，如果某个组件没有通过子类实现抽象基类（但是实现了必要的方法），那么事后还可以注册，让显式类型检查通过。

然而，即使是抽象基类，也不能滥用 `isinstance` 检查，因为用得多了可能导致**代码异味**，即表明面向对象设计不佳。

在一连串 `if/elif/elif` 中使用 `isinstance` 做检查，然后根据对象的类型执行不同的操作，往往是**不好的**做法。此时应该使用多态，即采用一定的方式定义类，让解释器把调用分派给正确的方法，而不使用 `if/elif/elif` 块硬编码分派逻辑。

另外，如果必须强制执行 API 契约，那么通常可以使用 `isinstance` 检查抽象基类。正如本书技术审校 Lennart Regebro 所说：“老兄，如果你想调用我，则必须实现这个。”这对采用插入式架构的系统来说特别有用。在框架之外，鸭子类型通常比类型检查更简单且更灵活。

在那篇短文的最后，Alex 多次强调，要抑制住创建抽象基类的冲动。滥用抽象基类会造成灾难性后果，表明语言太注重表面形式，这对以

实用和务实著称的 Python 可不是好事。在审阅本书的过程中，Alex 在一封电子邮件中写道：

抽象基类是用于封装框架所引入的一般性概念和抽象的，例如“一个序列”和“一个确切的数”。（读者）基本上不需要自己编写新的抽象基类，只要正确使用现有的抽象基类，就能获得 99.9% 的好处，而不用冒着设计不当导致的巨大风险。

下面通过实例讲解大鹅类型。

13.5.1 子类化一个抽象基类

我们将遵循 Martelli 的建议，先利用现有的抽象基类 `collections.MutableSequence`，然后再“斗胆”自己定义。示例 13-6 明确地把 `FrenchDeck2` 声明为了 `collections.MutableSequence` 的子类。

示例 13-6 `frenchdeck2.py`：

`collections.MutableSequence` 的子类 `FrenchDeck2`

```
from collections import namedtuple, abc

Card = namedtuple('Card', ['rank', 'suit'])

class FrenchDeck2(abc.MutableSequence):
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]

    def __setitem__(self, position, value): ❶
        self._cards[position] = value
```

```
def __delitem__(self, position): ❷
    del self._cards[position]

def insert(self, position, value): ❸
    self._cards.insert(position, value)
```

❶ 为了支持洗牌，只需实现 `__setitem__` 方法即可。

❷ 但是，继承 `MutableSequence` 的类必须实现 `__delitem__` 方法，这是 `MutableSequence` 类的一个抽象方法。

❸ 此外，还要实现 `insert` 方法，这是 `MutableSequence` 类的第三个抽象方法。

Python 在导入时（加载并编译 `frenchdeck2.py` 模块时）不检查抽象方法的实现，在运行时实例化 `FrenchDeck2` 类时才真正检查。因此，如果没有正确实现某个抽象方法，那么 Python 就会抛出 `TypeError` 异常，错误消息为 `"Can't instantiate abstract class FrenchDeck2 with abstract methods __delitem__, insert"`。正是这个原因，即便 `FrenchDeck2` 类不需要 `__delitem__` 和 `insert` 提供的行为，也要实现，这是 `MutableSequence` 抽象基类的要求。

如图 13-3 所示，抽象基类 `Sequence` 和 `MutableSequence` 的方法不全是抽象的。

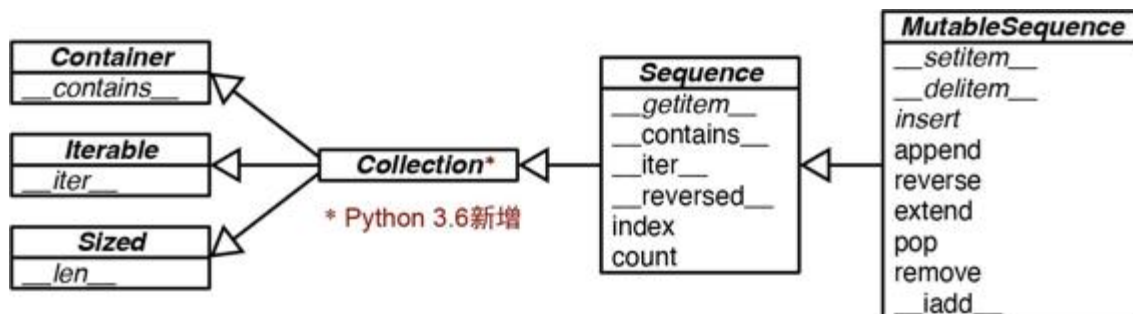


图 13-3: `MutableSequence` 抽象基类和 `collections.abc` 中它的超类的 UML 类图（箭头由子类指向祖先，以斜体显示的名称是抽象方法）

象类和抽象方法)

为了把 `FrenchDeck2` 声明为 `MutableSequence` 的子类，我不得不实现例子中用不到的 `__delitem__` 方法和 `insert` 方法。作为回报，`FrenchDeck2` 从 `Sequence` 继承了 5 个具体方法：

`__contains__`、`__iter__`、`__reversed__`、`index` 和 `count`。另外，`FrenchDeck2` 还从 `MutableSequence` 继承了 6 个方法：`append`、`reverse`、`extend`、`pop`、`remove` 和 `__iadd__`（为就地拼接的 `+=` 运算符提供支持）。

在 `collections.abc` 中，每个抽象基类的具体方法都是作为类的公开接口实现的，因此无须知道实例的内部结构。



作为实现具体子类的人，你可以覆盖从抽象基类继承的方法，以更高效的方式重新实现。例如，`__contains__` 方法会全面扫描序列，但是，如果你定义的序列按顺序保存元素，则可以重新定义 `__contains__` 方法，使用标准库中的 `bisect` 函数做二分查找，从而提升搜索速度。详见本书配套网站中的“Managing Ordered Sequences with Bisect”一文。

为了充分利用抽象基类，要知道有哪些抽象基类可用。接下来介绍 `collections` 包中的抽象基类。

13.5.2 标准库中的抽象基类

从 Python 2.6 开始，标准库提供了多个抽象基类，大都在 `collections.abc` 模块中定义，不过其他地方也有，例如，`io` 包和 `numbers` 包中就有一些抽象基类。但是，`collections.abc` 中的抽象基类最常用。



标准库中有两个名为 `abc` 的模块，这里说的是 `collections.abc`。为了减少加载时间，Python 3.4 在

`collections` 包之外实现这个模块（在 `Lib/_collections_abc.py` 中），因此要与 `collections` 分开导入。另一个 `abc` 模块就是 `abc` (`Lib/abc.py`)，这里定义的是 `abc.ABC` 类。每个抽象基类都依赖 `abc` 模块，但是不用导入它，除非自己动手定义新抽象基类。

图 13-4 是 `collections.abc` 模块中 17 个抽象基类的 UML 类图（简图，没有属性名称）。`collections.abc` 的文档中有一张不错的表格，对这些抽象基类做了总结，说明了它们相互之间的关系，以及各个基类提供的抽象方法和具体方法（叫作“混入方法”）。图 13-4 中有很多多重继承。第 14 章将着重说明多重继承，在讨论抽象基类时通常无须考虑多重继承。⁶

⁶Java 认为多重继承有危害，因此没有提供支持，但是提供了接口：Java 接口可以扩展多个接口，而且 Java 类可以实现多个接口。

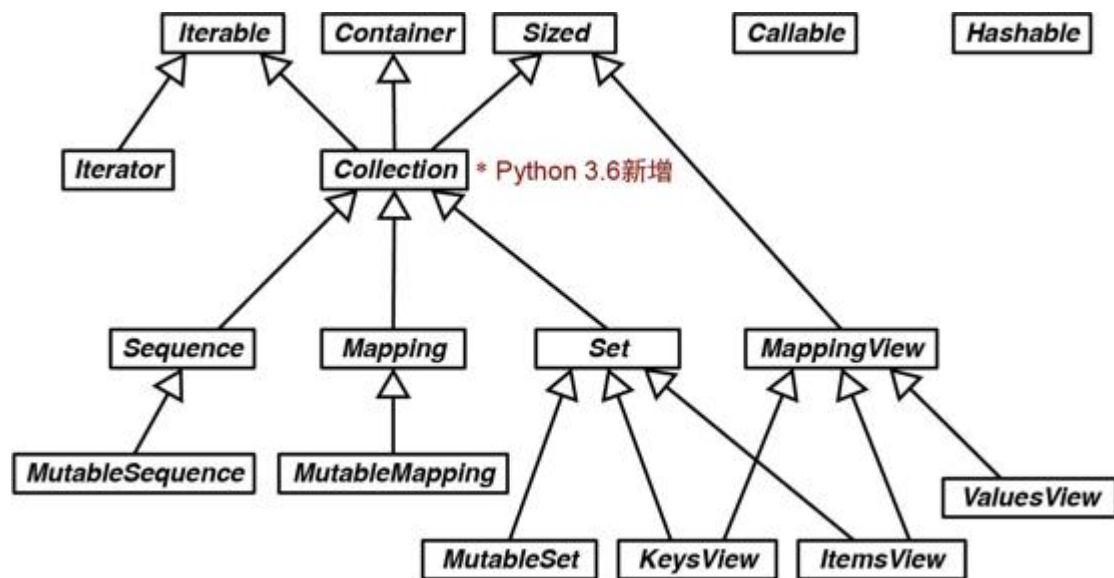


图 13-4: `collections.abc` 模块中抽象基类的 UML 类图

下面详述一下图 13-4 中那一群基类。

`Iterable`、`Container` 和 `Sized`

每个容器都应该继承这 3 个抽象基类，或者实现兼容的协议。Iterable 通过 `__iter__` 方法支持迭代，Container 通过 `__contains__` 方法支持 `in` 运算符，Sized 通过 `__len__` 方法支持 `len()` 函数。

Collection

这个抽象基类是 Python 3.6 新增的，自身没有方法，目的是方便子类化 Iterable、Container 和 Sized。

Sequence、Mapping 和 Set

这 3 个抽象基类是主要的不可变容器类型，而且各自都有可变的子类。MutableSequence 的详细类图见图 13-3，MutableMapping 和 MutableSet 的类图见图 3-1 和图 3-2。

MappingView

在 Python 3 中，映射方法 `.items()`、`.keys()` 和 `.values()` 返回的对象分别实现了 ItemsView、KeysView 和 ValuesView 定义的接口。前两个还实现了丰富的 Set 接口，拥有“集合运算”一节讲到的所有运算符。

Iterator

注意它是 Iterable 的子类。第 17 章将详细讨论。

Callable 和 Hashable

这两个不是容器，只不过因为 `collections.abc` 是标准库中定义抽象基类的第一个模块，而它们又太重要了，因此才被放在这里。它们可以在类型检查中用于指定可调用和可哈希的对象。

检查对象能不能调用，内置函数 `callable(obj)` 比 `instance(obj, Callable)` 使用起来更方便。

如果 `isinstance(obj, Hashable)` 返回 `False`，那么可以确定 `obj` 不可哈希。然而，返回 `True` 则可能是误判。详见下面的附注栏，结果可能不准确。

使用 `isinstance` 检查 `Hashable` 和 `Iterable`，结果可能不准确

使用 `isinstance` 和 `issubclass` 测试抽象基类 `Hashable` 和 `Iterable`，结果很有可能让人误解。

如果 `isinstance(obj, Hashable)` 返回 `True`，那么仅仅表示 `obj` 所属的类实现或继承了 `__hash__` 方法。假如 `obj` 是包含不可哈希项的元组，那么即便 `isinstance` 的检查结果为真，`obj` 仍是不可哈希对象。技术审校 Jürgen Gmach 指出，利用鸭子类型判断一个实例是否可哈希是最准确的，即调用 `hash(obj)`。如果 `obj` 不可哈希，那么该调用就会抛出 `TypeError`。

另外，即使 `isinstance(obj, Iterable)` 返回 `False`，Python 依然可以通过 `__getitem__`（基于 0 的索引）迭代 `obj`（参见第 1 章和 13.4.1 节）。`collections.abc.Iterable` 的文档指出：

判断一个对象是否可以迭代，唯一可靠的方式是调用 `iter(obj)`。

了解了一些现有的抽象基类之后，下面从零开始实现一个抽象基类，然后实际使用，以此实践大鹅类型。这么做的目的不是鼓励所有人自己动手定义抽象基类，而是借此教你如何阅读标准库和其他包中的抽象基类源码。

13.5.3 定义并使用一个抽象基类

本书第 1 版在讲“接口”那一章中给出了如下警告。

抽象基类与描述符和元类一样，是用于构建框架的工具。因此，只有少数 Python 开发者编写的抽象基类不会对用户施加不必要的限制，让他们做无用功。

如今，抽象基类的作用更广，可用在类型提示中，支持静态类型。8.5.7 节讲过，把函数参数类型提示中的具体类型换成抽象基类能为调用方提供更大的灵活性。

为了证明有必要定义抽象基类，需要在框架中找到使用它的场景。想象一下这个场景：你想在网站或移动应用程序中显示随机广告，但是在整个广告清单轮转一遍之前，不重复显示广告。假设我们在构建一个名为 `ADAM` 的广告管理框架。它的职责之一是，支持用户提供随机挑选的无重复类。⁷ 为了让 `ADAM` 的用户明确理解“随机挑选的无重复”组件是什么意思，我们将定义一个抽象基类。

⁷客户可能要审查随机发生器，或者代理想作弊……谁知道呢！

受到“栈”和“队列”（以物体的排放方式说明抽象接口）的启发，我将使用现实世界中的物品命名这个抽象基类：宾果机和彩票机是随机从有限的集合中挑选物品的机器，选出的物品没有重复，直到选完为止。

把这个抽象基类命名为 `Tombola`，这是宾果机和打乱数字的滚动容器的意大利语名。

抽象基类 `Tombola` 有 4 个方法，其中两个是抽象方法。

```
.load(...)
```

把元素放入容器。

```
.pick()
```

从容器中随机拿出一个元素，再返回这个元素。

另外两个是具体方法。

```
.loaded()
```

如果容器中至少有一个元素，就返回 `True`。

```
.inspect()
```

返回由容器中现有的元素构成的元组，不改变容器的内容（内部的顺序不保留）。

图 13-5 展示了抽象基类 `Tombola` 和 3 个具体实现。

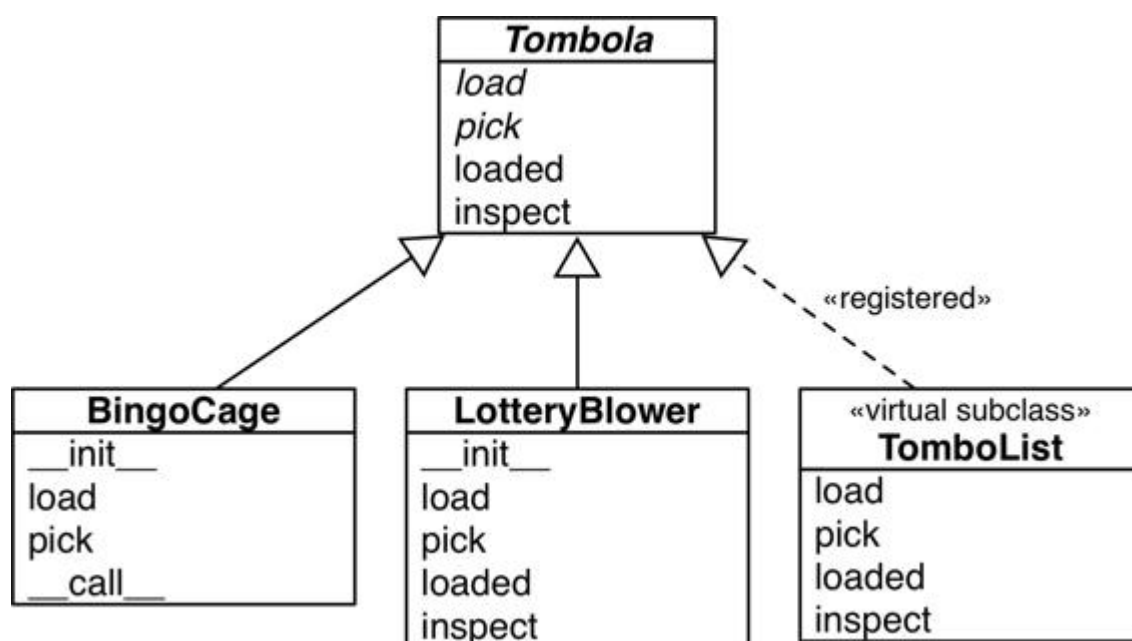


图 13-5：一个抽象基类和 3 个子类的 UML 类图。根据 UML 的约定，抽象基类 `Tombola` 和它的抽象方法使用斜体。虚线箭头表示接口实现，这里表示 `TomboList` 不仅实现了 `Tombola` 接口，还被注册为 `Tombola` 的虚拟子类（详见本章后文）⁸

⁸«registered» 和 «virtual subclass» 不是标准的 UML 术语。这里使用二者表示 Python 类之间的关系。

抽象基类 `Tombola` 的定义如示例 13-7 所示。

示例 13-7 tombola.py: Tombola 是有两个抽象方法和两个具体方法的抽象基类

```
import abc

class Tombola(abc.ABC): ❶

    @abc.abstractmethod
    def load(self, iterable): ❷
        """从可迭代对象中添加元素"""

    @abc.abstractmethod
    def pick(self): ❸
        """随机删除元素，再返回被删除的元素。

        如果实例为空，那么这个方法应该抛出LookupError
        """

    def loaded(self): ❹
        """如果至少有一个元素，就返回True，否则返回False"""
        return bool(self.inspect()) ❺

    def inspect(self):
        """返回由容器中的当前元素构成的有序元组"""
        items = []
        while True: ❻
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items) ❼
        return tuple(items)
```

❶ 继承 `abc.ABC`，定义一个抽象基类。

❷ 抽象方法使用 `@abstractmethod` 装饰器标记，主体通常只有文档字符串。⁹

⁹在抽象基类出现之前，抽象方法会抛出 `NotImplementedError`，表明由子类负责实现。在 `Smalltalk-80` 中，抽象方法的主体会调用从 `object` 继承的 `subclassResponsibility` 方法，抛出错误，错误消息为“My subclass should have overridden one of my messages”。

- ③ 根据文档字符串，如果没有元素可选，那么应该抛出 `LookupError`。
- ④ 抽象基类可以包含具体方法。
- ⑤ 抽象基类中的具体方法只能依赖抽象基类定义的接口（只能使用抽象基类中的其他具体方法、抽象方法或特性）。
- ⑥ 我们不知道具体子类如何存储元素，但可以不断调用 `.pick()` 方法，把 `Tombola` 清空.....
- ⑦ 然后再使用 `.load(...)` 把所有元素放回去。



其实，抽象方法可以有实现代码。即便实现了，子类也必须覆盖抽象方法，但是在子类中可以使用 `super()` 函数调用抽象方法，在此基础上添加功能，而不是从头开始实现。

@`abstractmethod` 装饰器的用法请参见 `abc` 模块的文档。

虽然示例 13-7 中的 `.inspect()` 方法的实现方式有些笨拙，但是表明，有了 `.pick()` 方法和 `.load(...)` 方法，如果想查看 `Tombola` 中的内容，可以先把所有元素挑出，然后再放回去——毕竟我们不知道元素具体是如何存储的。这个示例的目的是强调抽象基类可以提供具体方法，只要仅依赖接口中的其他方法就行。`Tombola` 的具体子类知晓内部数据结构，可以使用更聪明的实现覆盖 `.inspect()` 方法，但这不是强制要求。

示例 13-7 中的 `.loaded()` 方法只有一行代码，但是耗时：调用 `.inspect()` 方法构建有序元组的目的仅仅是在其上调用 `bool()` 函数。虽然这样做没问题，但是具体子类可以做得更好，后文见分晓。

注意，实现 `.inspect()` 方法采用的迂回方式要求捕获 `self.pick()` 抛出的 `LookupError`。`self.pick()` 会抛出 `LookupError` 这一事实也是接口的一部分，但是在 Python 中没办

法明确表明，只能在文档中说明（参见示例 13-7 中抽象方法 `pick` 的文档字符串）。

选择使用 `LookupError` 异常的原因是，在 Python 的异常层次关系中，它与 `IndexError` 和 `KeyError` 有关，而这两个是具体实现 `Tombola` 所用的数据结构最有可能抛出的异常。因此，实现代码可能会抛出 `LookupError`、`IndexError`、`KeyError`，或者符合要求的 `LookupError` 自定义子类。异常的部分层次结构如图 13-6 所示。

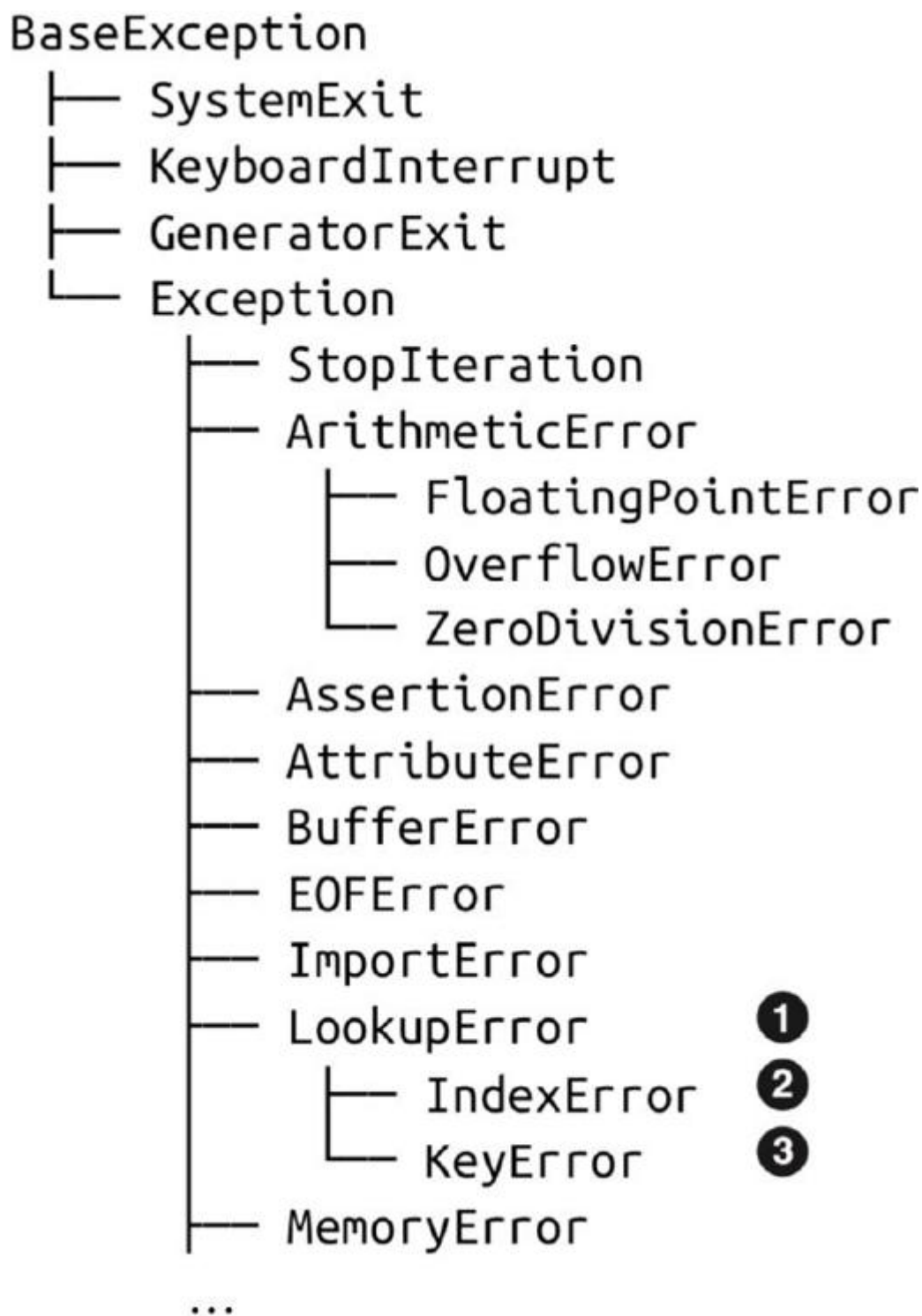


图 13-6: Exception 类的部分层次结构¹⁰

¹⁰完整的层次结构参见 Python 标准库文档中的“5.4. Exception hierarchy”一节。

- ❶ 在 `Tombola.inspect` 方法中处理的是 `LookupError` 异常。
- ❷ `IndexError` 是 `LookupError` 的子类，会在尝试从序列中获取索引超过最后位置的元素时抛出。
- ❸ 当使用不存在的键从映射中获取元素时，抛出 `KeyError` 异常。

我们自己定义的抽象基类 `Tombola` 完成了。为了一睹抽象基类接口所做的检查，下面尝试使用一个有缺陷的实现来“糊弄”`Tombola`，如示例 13-8 所示。

示例 13-8 不符合 `Tombola` 要求的子类无法蒙混过关

```
>>> from tombola import Tombola
>>> class Fake(Tombola): ❶
...     def pick(self):
...         return 13
...
>>> Fake ❷
<class '__main__.Fake'>
>>> f = Fake() ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract
method load
```

- ❶ 把 `Fake` 声明为 `Tombola` 的子类。
- ❷ 创建了 `Fake` 类，目前没有错误。
- ❸ 尝试实例化 `Fake` 时抛出了 `TypeError`。错误消息十分明确，Python 认为 `Fake` 是抽象类，因为它没有实现抽象基类 `Tombola` 声明的抽象方法之一 `load`。

我们的第一个抽象基类定义好了，而且还用它实际验证了一个类。稍后将定义抽象基类 `Tombola` 的子类，在此之前必须说明抽象基类的一些编程规则。

13.5.4 抽象基类句法详解

声明抽象基类的标准方式是继承 `abc.ABC` 或其他抽象基类。

除了 `ABC` 基类和 `@abstractmethod` 装饰器，`abc` 模块还定义了 `@classmethod` 装饰器、`@abstractclassmethod` 装饰器和 `@abstractproperty` 装饰器。然而，后 3 个装饰器在 Python 3.3 中弃用了，因为现在可以在 `@abstractmethod` 之上叠放装饰器，那 3 个就显得多余了。例如，声明抽象类方法的推荐做法如下所示。

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```



在函数上叠放装饰器的顺序通常很重要，`@abstractmethod` 的文档就特别指出：

与其他方法描述符一起使用时，`abstractmethod()` 应该放在最里层.....¹¹

也就是说，在 `@abstractmethod` 和 `def` 语句之间不能有其他装饰器。

¹¹摘自 `abc` 模块文档中的 `@abc.abstractmethod` 词条。

说明抽象基类的句法之后，接下来要实现几个具体子代，实际使用 `Tombola`。

13.5.5 子类化抽象基类 `Tombola`

定义好抽象基类 `Tombola` 之后，要开发两个具体子类，满足 `Tombola` 规定的接口。这两个子类的类图如图 13-5 所示，图中还有 13.5.6 节将要讨论的一个虚拟子类。

示例 13-9 中的 `BingoCage` 类是在示例 7-8 的基础上修改的，使用了更好的随机发生器。`BingoCage` 实现了所需的抽象方法 `load` 和 `pick`。

示例 13-9 bingo.py: `BingoCage` 是 `Tombola` 的具体子类

```
import random

from tombola import Tombola

class BingoCage(Tombola): ❶

    def __init__(self, items):
        self._randomizer = random.SystemRandom() ❷
        self._items = []
        self.load(items) ❸

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items) ❹

    def pick(self): ❺
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self): ❻
        self.pick()
```

❶ `BingoCage` 类会显式扩展 `Tombola` 类。

❷ 假设将在线上游戏中使用这个随机发生器。

`random.SystemRandom` 使用 `os.urandom(...)` 函数实现 `random` API。根据 `os` 模块的文档，`os.urandom(...)` 函数生成“适合用于加密”的随机字节序列。

- ③ 委托 `.load(...)` 方法实现初始加载。
- ④ 没有使用 `random.shuffle()` 函数，而是使用了 `SystemRandom` 实例的 `.shuffle()` 方法。
- ⑤ `pick` 方法的实现方式与示例 7-8 一样。
- ⑥ `__call__` 方法也跟示例 7-8 中的一样。为了满足 `Tombola` 接口，无须实现这个方法，不过额外增加方法也没有危害。

`BingoCage` 从 `Tombola` 中继承了耗时的 `loaded` 方法和笨拙的 `inspect` 方法。这两个方法都可以覆盖，变成示例 13-10 中速度更快的一行代码。这里想表达的观点是，我们可以偷懒，直接从抽象基类中继承不是那么理想的具体方法。从 `Tombola` 中继承的方法没有 `BingoCage` 自己定义的那么快，不过只要 `Tombola` 的子类正确实现 `pick` 方法和 `load` 方法，就能提供正确的结果。

示例 13-10 是 `Tombola` 接口的另一种实现，虽然与之前不同，但完全有效。`LottoBlower` 打乱“数字球”后没有取出最后一个，而是取出了一个随机位置上的球。

示例 13-10 `lotto.py`: `LottoBlower` 是 `Tombola` 的具体子类，覆盖了继承的 `inspect` 方法和 `loaded` 方法

```
import random

from tombola import Tombola

class LottoBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable) ❶

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
```

```
        position = random.randrange(len(self._balls)) ❷
    except ValueError:
        raise LookupError('pick from empty LottoBlower')
    return self._balls.pop(position) ❸

    def loaded(self): ❹
        return bool(self._balls)

    def inspect(self): ❺
        return tuple(self._balls)
```

❶ 初始化方法接受任何可迭代对象，使用传入的参数构建一个列表。

❷ 如果范围为空，那么 `random.randrange(...)` 函数就会抛出 `ValueError`。为了兼容 `Tombola`，可以捕获该异常，重新抛出 `LookupError`。

❸ 否则，从 `self._balls` 中取出随机选中的元素。

❹ 覆盖 `loaded` 方法，避免调用 `inspect` 方法（示例 13-7 中的 `Tombola.loaded` 方法就是这么做的）。可以直接处理 `self._balls`，而不必构建整个元组，从而提升速度。

❺ 覆盖 `inspect` 方法，仅用一行代码。

示例 13-10 中有个习惯做法值得指出：在 `__init__` 方法中，`self._balls` 存储的是 `list(iterable)`，而不是 `iterable` 的引用（没有直接把 `iterable` 赋值给 `self._balls`，为参数创建别名）。13.4.3 节说过，这样做使得 `LottoBlower` 更灵活，因为 `iterable` 参数可以是任何可迭代类型。把元素存入列表中还可以确保能取出元素。就算 `iterable` 参数始终传入列表，`list(iterable)` 也会创建参数的副本，这依然是好的做法，因为要从中删除元素，而客户可能不希望自己提供的列表被修改。¹²

¹²6.5.2 节专门讨论过这种防止混淆别名的问题。

接下来介绍大鹅类型的重要动态特性：使用 `register` 方法声明虚拟子类。

13.5.6 抽象基类的虚拟子类

大鹅类型的一个基本特征（也是值得用水禽来命名的原因之一）是，即便不继承，也有办法把一个类注册为抽象基类的**虚拟子类**。这样做时，我们承诺注册的类忠实地实现了抽象基类定义的接口，而 Python 会相信我们，不再检查。如果我们说谎了，那么常规的运行时异常会把我们捕获。

注册虚拟子类的方式是在抽象基类上调用 `register` 类方法。这么做之后，注册的类就变成了抽象基类的虚拟子类，而且 `issubclass` 函数能够识别这种关系，但是注册的类不会从抽象基类中继承任何方法或属性。



虚拟子类不继承注册的抽象基类，而且任何时候都不检查它是否符合抽象基类的接口，即便在实例化时也不会检查。另外，静态类型检查工具目前也无法处理虚拟子类。详见 Mypy 的 2922 号工单，即“`ABCMeta.register` support”。

`register` 方法通常作为普通函数调用（参见 13.5.7 节），不过也可以作为装饰器使用。在示例 13-11 中，我们使用装饰器句法实现了 `Tombola` 的虚拟子类 `TomboList`，如图 13-7 所示。

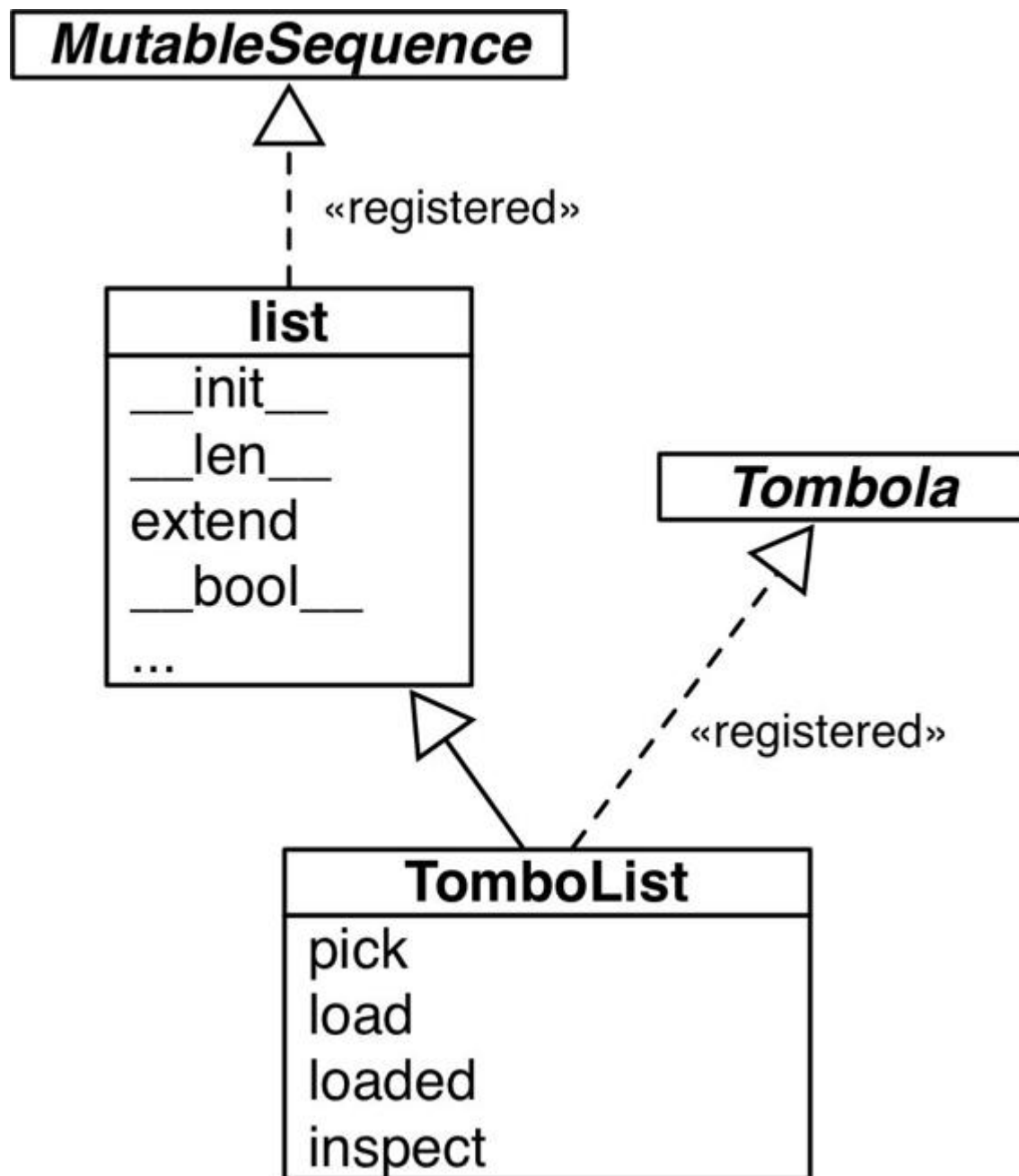


图 13-7: TomboList 的 UML 类图, 它既是 list 的真实子类, 也是 Tombola 的虚拟子类

示例 13-11 tombolist.py: TomboList 是 Tombola 的虚拟子类

```

from random import randrange

from tombola import Tombola

@Tombola.register ❶
class TomboList(list): ❷

    def pick(self):
        if self: ❸
            position = randrange(len(self))
            return self.pop(position) ❹
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend ❺

    def loaded(self):
        return bool(self) ❻

    def inspect(self):
        return tuple(self)

# Tombola.register(TomboList) ❼

```

❶ 把 TomboList 注册为 Tombola 的虚拟子类。

❷ TomboList 扩展 list。

❸ TomboList 从 list 继承布尔值行为，在列表不为空时返回 True。

❹ pick 调用从 list 继承的 self.pop 方法，传入一个随机的元素索引。

❺ TomboList.load 等同于 list.extend。

❻ loaded 委托 bool。¹³

¹³ loaded() 不能采用 load() 那种方式，因为 list 类型没有实现 loaded 所需的 __bool__ 方法。而内置函数 bool 不需要 __bool__ 方法，因为它还可以使用 __len__ 方法。详见 Python 文档中“Built-in Types”一章的“4.1. Truth Value Testing”。

⑦ 始终可以这样调用 `register`。如果需要注册不是自己维护的类，却能满足指定的接口，就可以这么做。

注册之后，可以使用 `issubclass` 函数和 `isinstance` 函数判断 `TomboList` 是不是 `Tombola` 的子类。

```
>>> from tombola import Tombola
>>> from tombolist import TomboList
>>> issubclass(TomboList, Tombola)
True
>>> t = TomboList(range(100))
>>> isinstance(t, Tombola)
True
```

然而，类的继承关系是在一个名为 `__mro__`（Method Resolution Order，方法解析顺序）的特殊类属性中指定的。这个属性的作用很简单，它会按顺序列出类及其超类，而 Python 会按照这个顺序搜索方法。¹⁴ 查看 `TomboList` 类的 `__mro__` 属性，你会发现它只列出了“真实”的超类，即 `list` 和 `object`。

¹⁴14.4 节将专门讲解 `__mro__` 类属性。现在知道这个简单的解释就行了。

```
>>> TomboList.__mro__
(<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)
```

`Tombolist.__mro__` 中没有 `Tombola`，因此 `Tombolist` 没有从 `Tombola` 中继承任何方法。

对抽象基类 `Tombola` 的研究到此结束。13.5.7 节将介绍 `register` 函数在标准库中的使用。

13.5.7 `register` 的实际使用

在示例 13-11 中，我们把 `Tombola.register` 当作一个类装饰器使用。在 Python 3.3 之前，`register` 不能这样使用，必须像示例 13-11 末尾的注释那样，作为一个普通函数在类主体之后调用。然而，即

便是现在，仍然经常把 `register` 当作普通函数调用，注册其他地方定义的类。例如，在 `collections.abc` 模块的源码中，内置类型 `tuple`、`str`、`range` 和 `memoryview` 会像下面这样被注册为 `Sequence` 的虚拟子类。

```
Sequence.register(tuple)
Sequence.register(str)
Sequence.register(range)
Sequence.register(memoryview)
```

另外，还有几个内置类型也会在 `_collections_abc.py` 中注册为抽象基类的虚拟子类。注册过程仅在导入模块时发生是没有问题的，因为如果想使用抽象基类，则必须导入模块。例如，从 `collections.abc` 中导入 `MutableMapping` 之后才能执行 `isinstance(my_dict, MutableMapping)` 检查。

子类化抽象基类或者注册到抽象基类上都能让类通过 `issubclass` 检查和 `isinstance` 检查（后者依赖前者）。但是，有些抽象基类还支持结构类型，详见 13.5.8 节。

13.5.8 使用抽象基类实现结构类型

抽象基类最常用于实现名义类型。假如一个类 `Sub` 会显式继承抽象基类 `AnABC`，或者注册到 `AnABC` 上，那么 `AnABC` 这个名称就和 `Sub` 连在了一起，因此在运行时，`issubclass(AnABC, Sub)` 会返回 `True`。

相比之下，结构类型通过对象公开接口的结构判断对象的类型，如果一个对象实现了某个类型定义的方法，那么该对象就与该类型**相容**。

¹⁵ 动态鸭子类型和静态鸭子类型是实现结构类型的两种方式。

¹⁵类型相容问题详见 8.5.1 节的“子类型与相容”。

其实，某些抽象基类也支持结构类型。Alex 在“水禽和抽象基类”附注栏中说过，未注册的类也可能被识别为抽象基类的子类。下面再次给出他举的例子（增加了 `issubclass` 测试）。

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abc.Sized)
True
```

经 `issubclass` 函数判断，`Struggle` 类是 `abc.Sized` 的子类（进而 `isinstance` 也得出同样的结论），因为 `abc.Sized` 实现了一个名为 `__subclasshook__` 的特殊的类方法。

`Sized` 类的 `__subclasshook__` 方法会检查通过参数传入的类有没有名为 `__len__` 的属性。如果有，就认为是 `Sized` 的虚拟子类。详见示例 13-12。

示例 13-12 源文件 `Lib/_collections_abc.py` 中 `Sized` 的定义

```
class Sized(metaclass=ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__): ❶
                return True ❷
            return NotImplemented ❸
```

❶ 如果 `C.__mro__` 列出的某个类（`C` 及其超类）的 `__dict__` 中有名为 `__len__` 的属性.....

- ② 就返回 `True`，表明 `C` 是 `Sized` 的虚拟子类。
- ③ 否则，返回 `NotImplemented`，让子类检查继续下去。



如果对子类检查的细节感兴趣，可以阅读 Python 3.6 中 `ABCMeta.__subclasscheck__` 方法的源码，该源码位于 `Lib/abc.py` 文件中。注意，源码中有大量条件判断和两个递归调用。在 Python 3.7 中，Ivan Levkivskyi 和 Inada Naoki 使用 C 语言重写了 `abc` 模块的大多数逻辑，性能更好，详见 Python 31333 号工单。`ABCMeta.__subclasshook__` 目前的实现只是调用 `_abc_subclasscheck`。相关的 C 语言源码在 `cpython/Modules/_abc.c` 文件第 605 行。

抽象基类对结构类型的支持就是通过 `__subclasshook__` 实现的。可以使用抽象基类确立接口，使用 `isinstance` 检查该抽象基类，一个完全无关的类仍然能通过 `issubclass` 检查，因为该类实现了特定的方法（或者该类竭力说服了 `__subclasshook__` 为它“担保”）。

那么，自己定义的抽象基类应该实现 `__subclasshook__` 方法吗？或许不应该。在 Python 源码中，我只见过 `Sized` 这种仅有一个特殊方法的抽象基类实现了 `__subclasshook__` 方法，而且只检查那个特殊方法的名称。由于 `__len__` 是“特殊”方法，因此我们可以十分肯定它的作用符合预期。然而，即便是特殊方法和基本的抽象基类，这种假设也是有风险的。例如，虽然映射实现了 `__len__`、`__getitem__` 和 `__iter__`，但是肯定不能把它看作 `Sequence` 的子类型，因为无法通过整数偏移或切片从映射中获取元素。鉴于此，`abc.Sequence` 类没有实现 `__subclasshook__` 方法。

对于我们自己编写的抽象基类，`__subclasshook__` 的可信度并不高。假如有一个名为 `Spam` 的类，它实现或继承了 `load`、`pick`、`inspect` 和 `loaded` 等方法，但是我并不能百分之百确定它的行为

与 Tombola 类似。让程序员把 Spam 定义为 Tombola 的子类，或者使用 `Tombola.register(Spam)` 注册，这样才能板上钉钉。当然，实现 `__subclasshook__` 方法时还可以检查方法签名和其他功能，但是我认为没这个必要。

13.6 静态协议



8.5.10 节介绍过静态协议。我本来打算到本章时再全面讲解协议，但是函数类型提示又不得不提到协议，因为鸭子类型是 Python 的重要基石，而且不涉及协议的静态类型检查无法很好地处理 Python 风格的 API。

本节将通过两个简单的示例来讲解静态协议，顺带讨论一下数值抽象基类和协议。首先说明如何利用静态协议来注解 8.4 节见过的 `double()` 函数，并对它做类型检查。

13.6.1 为 `double` 函数添加类型提示

在向更习惯静态类型语言的程序员介绍 Python 时，我喜欢用简单的 `double` 函数举例。

```
>>> def double(x):
...     return x * 2
...
>>> double(1.5)
3.0
>>> double('A')
'AA'
>>> double([10, 20, 30])
[10, 20, 30, 10, 20, 30]
>>> from fractions import Fraction
>>> double(Fraction(2, 5))
Fraction(4, 5)
```

引入静态协议之前，几乎不可能为 `double` 函数添加完美的类型提示，用途总会受到限制。¹⁶

¹⁶好吧，除了用作示例，`double()` 函数没什么太大的用处。不过，在 Python 3.8 增加静态协议之前，标准库中也有很多函数无法准确注解。我使用协议添加类型提示，修正了 `typeshed` 项目中的很多 bug。比如说，修正“Should Mypy warn about potential invalid arguments to `max`?”工单的拉取请求利用 `_SupportsLessThan` 协议改进了 `max`、`min`、`sorted` 和 `list.sort` 的注解。

得益于鸭子类型，`double` 函数甚至支持未来可能出现的类型，例如 16.5 节的增强的 `Vector` 类。

```
>>> from vector_v7 import Vector
>>> double(Vector([11.0, 12.0, 13.0]))
Vector([22.0, 24.0, 26.0])
```

Python 最初实现类型提示利用的是名义类型系统，注解中的类型名称要与实参的类型名称（或者某个超类的名称）匹配。我们知道，支持必要操作的类型就算实现了协议，而这样的类型可能很多，无法一一列出，因此在 Python 3.8 之前，类型提示无法描述鸭子类型。

有了 `typing.Protocol` 之后，现在可以告诉 Mypy，`double` 函数接受支持 `x * 2` 运算的参数 `x`，如示例 13-13 所示。

示例 13-13 `double_protocol.py`：使用 `Protocol` 定义 `double` 函数

```
from typing import TypeVar, Protocol

T = TypeVar('T') ❶

class Repeatable(Protocol):
    def __mul__(self: T, repeat_count: int) -> T: ... ❷

RT = TypeVar('RT', bound=Repeatable) ❸

def double(x: RT) -> RT: ❹
    return x * 2
```

❶ `T` 在 `__mul__` 签名中使用。

❷ `__mul__` 是 `Repeatable` 协议的核心。`self` 参数通常不注解，因为默认假定为所在的类。这里使用 `T` 是为了确保返回值的类型与 `self` 相同。另外注意，这个协议把 `repeat_count` 限制为 `int` 类型。

❸ 类型变量 `RT` 的上界由 `Repeatable` 协议限定，类型检查工具将要求具体使用的类型实现 `Repeatable` 协议。

❹ 现在，类型检查工具可以确认 `x` 参数是一个可以乘以整数的对象，而且返回值的类型与 `x` 相同。

通过这个示例可以看出，为什么 PEP 544 的标题为“Protocols: Structural subtyping (static duck typing)”。提供给 `double` 函数的实参 `x` 是什么名义类型无关紧要，只要实现了 `__mul__` 方法就行——这就是鸭子类型的好处。

13.6.2 运行时可检查的静态协议

在类型图（参见图 13-1）中，`typing.Protocol` 位于静态检查区域，即图的下半部分。然而，定义 `typing.Protocol` 的子类时，可以借由 `@runtime_checkable` 装饰器让协议支持在运行时使用 `isinstance/issubclass` 检查。这背后的原因是，`typing.Protocol` 是一个抽象基类，因此它支持 13.5.8 节讲过的 `__subclasshook__`。

从 Python 3.9 开始，`typing` 模块提供了 7 个可在运行时检查的协议。下面是其中两个，直接摘自 `typing` 模块的文档。

```
class typing.SupportsComplex
```

抽象基类，有一个抽象方法 `__complex__`。

```
class typing.SupportsFloat
```

抽象基类，有一个抽象方法 `__float__`。

这些协议旨在检查数值类型可否转换类型。如果对象 `o` 实现了 `__complex__`，那么调用 `complex(o)` 应该得到一个 `complex` 值，因为在背后支持内置函数 `complex()` 的就是特殊方法 `__complex__`。

示例 13-14 是 `typing.SupportsComplex` 协议的源码。

示例 13-14 `typing.SupportsComplex` 协议的源码

```
@runtime_checkable
class SupportsComplex(Protocol):
    """具有一个抽象方法__complex__的抽象基类"""
    __slots__ = ()

    @abstractmethod
    def __complex__(self) -> complex:
        pass
```

这个协议的核心是抽象方法 `__complex__`。¹⁷ 在静态类型检查中，如果一个对象实现了 `__complex__` 方法，而且只接受参数 `self`，并且返回一个 `complex` 值，那么就认为该对象与 `SupportsComplex` 协议**相容**。

¹⁷ `__slots__` 属性与目前讨论的话题无关，它是 11.11 节讲过的一种优化措施。

由于 `SupportsComplex` 应用了 `@runtime_checkable` 类装饰器，因此该协议也可以使用 `isinstance` 检查，如示例 13-15 所示。

示例 13-15 在运行时使用 `SupportsComplex`

```
>>> from typing import SupportsComplex
>>> import numpy as np
>>> c64 = np.complex64(3+4j) ❶
>>> isinstance(c64, complex) ❷
False
```

```
>>> isinstance(c64, SupportsComplex) ❸
True
>>> c = complex(c64) ❹
>>> c
(3+4j)
>>> isinstance(c, SupportsComplex) ❺
False
>>> complex(c)
(3+4j)
```

- ❶ `complex64` 是 NumPy 提供的 5 种复数类型之一。
- ❷ NumPy 中的复数类型均不是内置类型 `complex` 的子类。
- ❸ 但是，NumPy 中的复数类型实现了 `__complex__` 方法，因此符合 `SupportsComplex` 协议。
- ❹ 因此，可以使用 NumPy 中的复数类型创建内置的 `complex` 对象。
- ❺ 可惜，内置类型 `complex` 没有实现 `__complex__` 方法。不过，当 `c` 是一个 `complex` 值时，`complex(c)` 能得到正确的结果。

根据最后一点，如果想测试对象 `c` 是不是 `complex` 或 `SupportsComplex`，那么可以为 `isinstance` 的第二个参数提供一个类型元组，如下所示。

```
isinstance(c, (complex, SupportsComplex))
```

另外，还可以使用 `numbers` 模块中定义的抽象基类 `Complex`。内置类型 `complex`，以及 NumPy 中的 `complex64` 类型和 `complex128` 类型都被注册为 `numbers.Complex` 的虚拟子类了，因此可以像下面这样检查。

```
>>> import numbers
>>> isinstance(c, numbers.Complex)
True
```

```
>>> isinstance(c64, numbers.Complex)
True
```

本书第 1 版建议使用 `numbers` 模块中的抽象基类，现在这个建议已经过时，因为静态类型检查工具无法识别那些抽象基类（详见 13.6.8 节）。

本节的目的本是说明运行时可检查的协议可使用 `isinstance` 测试，但是后来才发现这个示例不是特别适合使用 `isinstance`，具体原因见后面的“充分利用鸭子类型”附注栏。



对于外部类型检查工具，使用 `isinstance` 明确检查类型有一个好处：在条件为 `isinstance(o, MyType)` 的 `if` 语句块内，Mypy 可以推导出 `o` 对象的类型与 `MyType` **相容**。

充分利用鸭子类型

在运行时，鸭子类型往往是类型检查的最佳方式。不要调用 `isinstance` 或 `hasattr`，直接在对象上尝试执行所需的操作，如果抛出异常，就处理异常。下面举个例子。

接着前面讨论的内容，假如我们想把对象 `o` 当作复数使用，那么可以这么做。

```
if isinstance(o, (complex, SupportsComplex)):
    # 当`o`可以转换成复数时执行一些操作
else:
    raise TypeError('o must be convertible to complex')
```

对于大鹅类型，则要使用抽象基类 `numbers.Complex`。

```
if isinstance(o, numbers.Complex):
    # 当`o`是`Complex`实例时执行一些操作
else:
    raise TypeError('o must be an instance of Complex')
```

然而，我更喜欢利用鸭子类型，因为取得原谅比获得许可容易（EAFP 原则）。

```
try:
    c = complex(o)
except TypeError as exc:
    raise TypeError('o must be convertible to complex') from exc
```

但是，如果只想抛出 `TypeError`，就省略 `try/except/raise` 语句，直接写成如下形式。

```
c = complex(o)
```

这时，如果 `o` 不是可接受的类型，那么 Python 将抛出异常，输出非常明确的消息。例如，当 `o` 是一个元组时，输出的消息如下所示。

```
TypeError: complex() first argument must be a string or a
number, not 'tuple'
```

我觉得在这种情况下使用鸭子类型效果好得多。

现在，我们知道，在运行时可以利用静态协议检查诸如 `complex` 和 `numpy.complex64` 之类的现有类型。接下来讨论运行时可检查协议的局限性。

13.6.3 运行时协议检查的局限性

如前所述，类型提示在运行时一般会被忽略。使用 `isinstance` 或 `issubclass` 检查静态协议有类似的影响。

例如，实现 `__float__` 方法的类在运行时都被认定是 `SupportsFloat` 的虚拟子类，不管 `__float__` 方法是否返回一个 `float` 值。

请看下面的控制台会话。

```
>>> import sys
>>> sys.version
'3.9.5 (v3.9.5:0a7dcdb13, May 3 2021, 13:17:02) \n[Clang 6.0
(clang-600.0.57)]'
>>> c = 3+4j
>>> c.__float__
<method-wrapper '__float__' of complex object at 0x10a16c590>
>>> c.__float__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
```

在 Python 3.9 中, `complex` 类型确实有 `__float__` 方法, 不过该方法仅仅抛出 `TypeError`, 并输出一个明确的错误消息。如果那个 `__float__` 方法有注解, 则返回值类型应该是 `NoReturn` (参见 8.5.12 节)。

但是, `typedshed` 项目中 `complex.__float__` 的类型提示解决不了这个问题, 因为 Python 的运行时会一般会忽略类型提示, 而且根本无法访问 `typedshed` 项目中的存根文件。

继续前面的 Python 3.9 控制台会话。

```
>>> from typing import SupportsFloat
>>> c = 3+4j
>>> isinstance(c, SupportsFloat)
True
>>> issubclass(complex, SupportsFloat)
True
```

检查的结果容易让人误解。运行时对 `SupportsFloat` 的检查表明, 可以把一个 `complex` 值转换成 `float` 值, 而实际情况却是抛出类型错误。



Python 3.10.0b4 删除了 `complex.__float__` 方法, 解决了 `complex` 类型的这个问题。

但是，类似的问题普遍存在。`isinstance` 或 `issubclass` 只检查有没有特定的方法，不检查方法的签名，更不会检查方法的类型注解。这种行为不会改变，因为在运行时大规模检查类型损耗的性能是不可接受的。¹⁸

¹⁸感谢 PEP 544（关于协议）的作者之一 Ivan Levkivskyi 指出，类型检查不是检查 `x` 的类型是不是 `T`，而是判断 `x` 的类型是否与 `T` 相容。检查相容的开销更大。难怪即便是很短的 Python 脚本，Mypy 也要用几秒才能完成类型检查。

下面来看一下如何在用户定义的类中实现静态协议。

13.6.4 支持静态协议

请回忆一下第 11 章中构建的 `Vector2d` 类。既然一个复数和一个 `Vector2d` 实例都由一对浮点数构成，那么顺理成章，应该支持把 `Vector2d` 转换成 `complex`。

示例 13-16 给出了 `__complex__` 方法的实现，在示例 11-11 中最后一版的基础上增强 `Vector2d` 类。为了支持双向转换，还定义了类方法 `fromcomplex`，执行反向操作，根据 `complex` 值构建 `Vector2d` 实例。

示例 13-16 `vector2d_v4.py`：与 `complex` 相互转换的方法

```
def __complex__(self):
    return complex(self.x, self.y)

@classmethod
def fromcomplex(cls, datum):
    return cls(datum.real, datum.imag) ❶
```

❶ 假设 `datum` 有 `.real` 属性和 `.imag` 属性。更好的实现参见示例 13-17。

根据上述代码，以及 `Vector2d` 现有的 `__abs__` 方法（参见示例 11-11），可以执行以下操作。

```

>>> from typing import SupportsComplex, SupportsAbs
>>> from vector2d_v4 import Vector2d
>>> v = Vector2d(3, 4)
>>> isinstance(v, SupportsComplex)
True
>>> isinstance(v, SupportsAbs)
True
>>> complex(v)
(3+4j)
>>> abs(v)
5.0
>>> Vector2d.fromcomplex(3+4j)
Vector2d(3.0, 4.0)

```

对于运行时类型检查，示例 13-16 可以胜任，但是为了让 Mypy 更好地做静态检查和错误报告，`__abs__` 方法、`__complex__` 方法和 `fromcomplex` 方法应该有类型提示，如示例 13-17 所示。

示例 13-17 vector2d_v5.py: 为当前研究的方法添加注解

```

def __abs__(self) -> float: ❶
    return math.hypot(self.x, self.y)

def __complex__(self) -> complex: ❷
    return complex(self.x, self.y)

@classmethod
def fromcomplex(cls, datum: SupportsComplex) -> Vector2d: ❸
    c = complex(datum) ❹
    return cls(c.real, c.imag)

```

❶ 需要把返回值类型注解为 `float`，否则 Mypy 推导出的类型是 `Any`，而且不检查方法主体。

❷ 即使不注解，Mypy 也能推导出该方法返回一个 `complex` 值。在 Mypy 的某些配置下，这个注解可以避免一个警告。

❸ `SupportsComplex` 确保 `datum` 可以转换成要求的类型。

❹ 有必要显式转换，因为 `SupportsComplex` 类型没有声明下一行用到的 `.real` 属性和 `.imag` 属性。例如，虽然 `Vector2d` 没有这两

个属性，但是实现了 `__complex__` 方法。

如果该模块的顶部有 `from __future__ import annotations`，那么 `fromcomplex` 的返回值类型可以是 `Vector2d`。有了那个导入语句，类型提示将存储为字符串，在导入时（求解函数定义时）不做求解。不从 `__future__` 中导入 `annotations`，`Vector2d` 在那一刻（类尚未完整定义）就是无效引用，应该写为字符串 `'Vector2d'`，假装是向前引用。这个 `__future__` 导入由“PEP 563—Postponed Evaluation of Annotations”引入，在 Python 3.7 中实现。原本计划在 Python 3.10 中把这个行为定为默认行为，但是后来推迟到下一个版本了。¹⁹ 到那时，这个导入语句就是多余的了，但是也没有危害。

¹⁹详见 Python 指导委员会在 `python-dev` 中发布的决定。

接下来介绍如何创建一个静态协议（稍后再扩展）。

13.6.5 设计一个静态协议

研究大鹅类型时，我们定义了抽象基类 `Tombola`（参见 13.5.3 节），现在将使用静态协议定义一个类似的接口。

抽象基类 `Tombola` 有两个抽象方法：`pick` 和 `load`。定义具有这两个方法的静态协议也不难，不过，我从 Go 语言社区学到一项知识：单方法协议实现的静态鸭子类型更有用且更灵活。Go 语言标准库中有多个这样的接口，例如 `Reader`，这是一个 I/O 接口，只要求一个 `read` 方法。以后，如果觉得需要一个更完整的协议，可以把多个协议合而为一。

可以随机从中选择元素的容器，不一定需要重新加载容器，但是肯定需要选择元素的方法。因此，我决定为精简的 `RandomPicker` 协议实现这样一个方法。该协议的代码如示例 13-18 所示，演示用法的测试如示例 13-19 所示。

示例 13-18 randompick.py: 定义 RandomPicker

```
from typing import Protocol, runtime_checkable, Any

@runtime_checkable
class RandomPicker(Protocol):
    def pick(self) -> Any: ...
```



`pick` 方法的返回值类型是 `Any`。15.8 节将说明如何让 `RandomPicker` 支持泛型参数，允许协议的用户指定 `pick` 方法的返回值类型。

示例 13-19 randompick_test.py: 使用 RandomPicker

```
import random
from typing import Any, Iterable, TYPE_CHECKING

from randompick import RandomPicker ❶

class SimplePicker: ❷
    def __init__(self, items: Iterable) -> None:
        self._items = list(items)
        random.shuffle(self._items)

    def pick(self) -> Any: ❸
        return self._items.pop()

def test_isinstance() -> None: ❹
    popper: RandomPicker = SimplePicker([1]) ❺
    assert isinstance(popper, RandomPicker) ❻

def test_item_type() -> None: ❼
    items = [1, 2]
    popper = SimplePicker(items)
    item = popper.pick()
    assert item in items
    if TYPE_CHECKING:
        reveal_type(item) ❸
    assert isinstance(item, int)
```

❶ 定义实现协议的类无须先导入静态协议。这里导入 `RandomPicker` 是因为后面的 `test_isinstance` 会用到。

- ❷ SimplePicker 实现 RandomPicker 协议，但不是后者的子类。这就是静态鸭子类型。
- ❸ 默认的返回值类型就是 Any，因此严格来说，不需要这个注解。但是，加上注解可以明确表明我们实现的是示例 13-18 中的 RandomPicker 协议。
- ❹ 如果想让 Mypy 检查，那么别忘了加上类型提示 -> None。
- ❺ 我为 popper 变量添加了类型提示，指出 Mypy 知道 SimplePicker 是相容的。
- ❻ 这个测试证明，SimplePicker 的实例也是 RandomPicker 的实例。背后的原因是，RandomPicker 应用了 @runtime_checkable 装饰器，而且 SimplePicker 有所需的 pick 方法。
- ❼ 这个测试在 SimplePicker 实例上调用 pick 方法，确认返回一个提供给 SimplePicker 的元素，然后对返回的元素做静态检查和运行时检查。
- ❽ 这一行在 Mypy 的输出中生成一个说明。

我们在示例 8-22 中见过的 `reveal_type` 是能被 Mypy 识别的“魔法”函数，无须导入，而且只能在受 `typing.TYPE_CHECKING` 条件保护的 `if` 块中调用。`typing.TYPE_CHECKING` 条件只在静态类型检查工具眼中为 `True`，在运行时为 `False`。

示例 13-19 中的两个测试均能通过，Mypy 也没有发现任何错误。对于 `pick` 方法返回的 `item`，`reveal_type` 输出的结果如下所示。

```
$ mypy randompick_test.py
randompick_test.py:24: note: Revealed type is 'Any'
```

这是我们创建的第一个协议。下面介绍一些设计协议的建议。

13.6.6 协议设计最佳实践

Go 语言 10 年的静态鸭子类型经验表明，窄协议（narrow protocol）更有用。通常，窄协议只有一个方法，很少超过两个。Martin Fowler 写了一篇定义**角色接口**（role interface）的文章，设计协议时可做考虑。

另外，有时你会发现，协议在使用它的函数附近定义，即在“客户代码”中而不是在库中定义。这样方便调用相关函数创建新类型，也有利于扩展和使用驭件（mock）测试。

窄协议和客户代码协议都能有效避免紧密耦合，正符合接口隔离原则（Interface Segregation Principle）。这个原则可用一句话概括：“不应强迫客户依赖用不到的接口。”

“Contributing to typeshed”页面建议静态协议采用以下命名约定（以下 3 点直接引用原文，未做改动）。

- 使用朴素的名称命名协议，清楚表明概念（例如 `Iterator` 和 `Container`）。
- 使用 `SupportsX` 形式命名提供可调用方法的协议（例如 `SupportsInt`、`SupportsRead` 和 `SupportsReadSeek`）。
- 使用 `HasX` 形式命名有可读属性和可写属性，或者有读值方法和设值方法的协议（例如 `HasItems` 和 `HasFileNo`）。

²⁰所有方法均可调用，因此这一条建议没有说到点子上。改成“提供一个或两个方法”会不会好一点儿？反正这是建议，不是严格规定。

我喜欢 Go 语言标准库采用的一种命名约定：对于只有一个方法的协议，如果方法名称是动词，就在末尾加上“-er”或“-or”，变成名词。例如，不要命名为 `SupportsRead`，而要命名为 `Reader`。此外还有一

些例子：Formatter、Animator 和 Scanner。如果想寻找灵感，可以阅读 Asuka Kenji 写的“Go (Golang) Standard Library Interfaces (Selected)”一文。

保持协议精简的好处是以后方便扩展。通过 13.6.7 节你会发现，衍生现有协议，额外添加方法并不难。

13.6.7 扩展一个协议

13.6.6 节开头提到，Go 语言开发人员定义接口（他们对静态协议的称呼）时倾向于极简主义。很多广泛使用的 Go 语言接口只有一个方法。

如果实际使用中发现协议需要多个方法，那么不要直接为协议添加方法，最好衍生原协议，创建一个新协议。在 Python 中，扩展静态协议有几个问题需要注意，如示例 13-20 所示。

示例 13-20 randompickload.py：扩展 RandomPicker 协议

```
from typing import Protocol, runtime_checkable
from randompick import RandomPicker

@runtime_checkable ❶
class LoadableRandomPicker(RandomPicker, Protocol): ❷
    def load(self, Iterable) -> None: ... ❸
```

❶ 如果希望衍生的协议可在运行时检查，则必须再次应用这个装饰器，因为该装饰器的行为不被继承。²¹

²¹详细原因见“PEP 544—Protocols: Structural subtyping (static duck typing)”中关于 `@runtime_checkable` 那一节。

❷ 每个协议都必须明确把 `typing.Protocol` 列出来，作为基类。另外，再列出要扩展的协议。这与 Python 中的继承不是一回事。²²

²²同样，详细原因见 PEP 544 中的“Merging and extending protocols”一节。

③ 现在是符合“常规”的面向对象编程方式了：只需要声明衍生协议新增的方法。`pick` 方法的声明继承自 `RandomPicker`。

本章对静态协议的定义和使用就讨论到这里了。

最后，再讲一下数值抽象基类，以及取而代之的数值协议。

13.6.8 `numbers` 模块中的抽象基类和 `Numeric` 协议

“论数字塔的倒下”一节讲过，标准库中 `numbers` 包内的抽象基类可用于做运行时类型检查。

如果想检查是不是整数，可以使用 `isinstance(x, numbers.Integral)`。`int`、`bool` (`int` 的子类)，以及外部库中注册为 `numbers` 包中某个抽象基类的虚拟子类的整数类型，都能通过这个测试。例如，`NumPy` 提供了 21 个整数类型，另外还有注册为 `numbers.Real` 的虚拟子类的多个浮点数类型，以及注册为 `numbers.Complex` 的虚拟子类的不同位宽度 (bit width) 的复数。



有点儿奇怪的是，`decimal.Decimal` 没有注册为 `numbers.Real` 的虚拟子类。原因也很好理解，如果在程序中需要使用 `Decimal` 提供的精度，那么你肯定不想与精度低的浮点数混淆。

可惜，`numbers` 包定义的数字塔不是为静态类型检查设计的。根抽象基类 `numbers.Number` 没有方法，因此，对于 `x: Number` 声明，`Mypy` 不会允许你对 `x` 做任何算术运算或者调用任何方法。

既然如此，应该怎么做呢？

`typedshed` 项目是一个很好的选择。这个项目为 Python 标准库提供类型提示，例如，`statistics` 模块的类型提示在存根文件

statistics.pyi 中。在这个文件中你能找到以下定义，很多函数的注解用到了这两个类型。

```
_Number = Union[float, Decimal, Fraction]
_NumberT = TypeVar('_NumberT', float, Decimal, Fraction)
```

这种方式是不错，但是不全面，不支持标准库以外的数值类型。numbers 包中的抽象基类支持在运行时检查外部数值类型，即那些注册为虚拟子类的数值类型。

目前的趋势是使用 typing 模块提供的数值协议（参见 13.6.2 节）。

然而，数值协议在运行时可能会让你失望。13.6.3 节讲过，在 Python 3.9 中，complex 类型虽然实现了 `__float__` 方法，但是该方法仅仅抛出 `TypeError`，警告“无法把复数转换为浮点数”。complex 类型实现的 `__int__` 方法也是如此。在 Python 3.9 中，这些方法的存在导致 `isinstance` 返回的结果让人误解。Python 3.10 把 complex 类型无条件抛出 `TypeError` 的那些方法删除了。²³

²³详见 41974 号工单，即“Remove complex.__float__, complex.__floordiv__, etc”。

另外，NumPy 中复数类型实现的 `__float__` 方法和 `__int__` 方法就好一些，只在第一次使用时发出警告。

```
>>> import numpy as np
>>> cd = np.cdouble(3+4j)
>>> cd
(3+4j)
>>> float(cd)
<stdin>:1: ComplexWarning: Casting complex values to real
discards the imaginary part
3.0
```

反向转换也有问题。内置类型 complex、float 和 int，以及 numpy.float16 和 numpy.uint8，没有实现 `__complex__` 方法，因此 `isinstance(x, SupportsComplex)` 返回 False。

²⁴NumPy 中的复数类型，比如 `np.complex64`，则实现了 `__complex__` 方法，可以转换成内置类型 `complex`。

²⁴我没有测试 NumPy 提供的其他浮点数类型和整数类型。

然而，实际使用中，内置构造函数 `complex()` 能正确处理所有这些类型的实例，不报错也不发出警告。

```
>>> import numpy as np
>>> from typing import SupportsComplex
>>> sample = [1+0j, np.complex64(1+0j), 1.0, np.float16(1.0), 1,
np.uint8(1)]
>>> [isinstance(x, SupportsComplex) for x in sample]
[False, True, False, False, False, False]
>>> [complex(x) for x in sample]
[(1+0j), (1+0j), (1+0j), (1+0j), (1+0j), (1+0j)]
```

由上述代码可知，`isinstance` 对 `SupportsComplex` 的检查，有些是失败的，但是全部都可以成功转换成 `complex` 类型。Guido van Rossum 在 `typing-sig` 邮件列表中指出，内置构造函数 `complex` 只接受一个参数，所以全都可以转换。

另外，对于下面的 `to_complex()` 函数，使用 Mypy 检查时，参数可以接受全部 6 种类型。

```
def to_complex(n: SupportsComplex) -> complex:
    return complex(n)
```

写作本书时，NumPy 没有类型提示，因此 NumPy 中的所有数值类型都是 `Any`。²⁵ 然而，不知为何，虽然在 `typeshed` 项目中仅内置类 `complex` 有 `__complex__` 方法，但 Mypy“知道”内置类型 `int` 和 `float` 可以转换成 `complex`。²⁶

²⁵NumPy 中的数值类型全都注册为 `numbers` 包中相应抽象基类的虚拟子类，但是全被 Mypy 忽略了。

²⁶这是 `typedshed` 项目的善意谎言，截至 Python 3.9，内置类型 `complex` 并没有 `__complex__` 方法。

综上所述，虽然数值类型不应该这么难做类型检查，但是现在的情况是，“PEP 484–Type Hints”有意避开数字塔，含蓄地建议类型检查工具硬编码内置类型 `complex`、`float` 和 `int` 之间的子类型关系。Mypy 就是这样做的，而且从实用角度出发，还认定 `int` 和 `float` 与 `SupportsComplex` 相容，尽管二者没有实现 `__complex__` 方法。



我只在测试与 `complex` 相互转换的操作时发现，使用 `isinstance` 检查与数值相关的 `Supports*` 协议得到的结果出乎意料。如果不使用复数，则可以依赖那些协议，而不使用 `numbers` 包中的抽象基类。

本节的主要结论如下。

- `numbers` 包中的抽象基类对运行时类型检查来说没有问题，但是不适合做静态类型检查。
- `SupportsComplex`、`SupportsFloat` 等数值相关的静态协议完美支持静态类型，但是当涉及复数时，运行时类型检查的结果不可靠。

下面简单总结一下本章内容。

13.7 本章小结

类型图（参见图 13-1）是理解本章内容的关键。简要介绍 4 种类型方式之后，我们对比了分别支持鸭子类型和静态鸭子类型的动态协议和静态协议。这两种协议有一个共同的基本特征，即类不需要显式声明对任何特定协议的支持。只要实现了协议要求的方法，类就支持那个协议。

接下来，重要的一节是 13.4 节。这一节深入探讨了 Python 解释器实现序列和可迭代动态协议的方式，包括对二者的部分实现。我们说明了如何利用猴子补丁为类添加额外的方法，在运行时实现协议。之后又提到了防御性编程，包括不显式使用 `isinstance` 或在 `try/except` 结构中使用 `hasattr` 检查结构类型，以及快速失败原则。

Alex Martelli 在 13.5 节的“水禽和抽象基类”附注栏中介绍大鹅类型之后，我们说明了如何子类化现有的抽象基类、考察了标准库中重要的抽象基类，还从头开始自己创建了一个抽象基类，后来又利用传统的子类化和注册机制提供了具体实现。最后我们了解到，即使是不相关的类，只要提供了抽象基类定义的接口要求的方法，也能被特殊方法 `__subclasshook__` 识别，从而让抽象基类支持结构类型。

13.6 节也很重要，这一节接着 8.5.10 节继续探讨了静态鸭子类型。我们了解到，`@runtime_checkable` 装饰器也利用 `__subclasshook__` 方法在运行时支持结构类型。不过，静态协议最好结合静态类型检查工具使用，连同类型提示，实现更可靠的结构类型。我们还讨论了静态协议的设计和实现，以及如何扩展。最后的 13.6.8 节讲述了被抛弃的数字塔的悲伤故事，还指出了现有替代方案的一些缺点，包括 `SupportsFloat` 等数值静态协议，以及 Python 3.8 在 `typing` 模块中增加的协议。

本章的主旨是告诉你，在现代的 Python 中，我们有 4 种互补的接口编程方法，它们各有优缺点。对于现代的 Python 基准代码，只要体量够大，4 种类型模式都有用武之地。抛下哪一种类型，作为 Python 程序员，你的日子都不会好过。

话又说回来，当初 Python 只支持鸭子类型时可是大受欢迎。JavaScript、PHP 和 Ruby 这些受欢迎的语言，以及不太流行、但是影响深远的 Lisp、Smalltalk、Erlang 和 Clojure，都从鸭子类型的强大和简单中受益匪浅。

13.8 延伸阅读

如果想了解类型的优缺点，以及 `typing.Protocol` 对经过静态检查的基准代码健康状况的重要性，强烈推荐阅读 Glyph Lefkowitz 写的文章“I Want A New Duck: `typing.Protocol` and the future of duck typing”。我还从他的另一篇文章中学到了很多，题为“Interfaces and Protocols——Comparing `zope.interface` and `typing.Protocol`”。`zope.interface` 是为松耦合插件系统定义接口的一种早期机制。Plone CMS、Web 框架 Pyramid 和异步编程框架 Twisted（Glyph 发起的项目）当时采用的都是 `zope.interface`。²⁷

²⁷感谢技术审校 Jürgen Gmach 推荐“Interfaces and Protocols”一文。

优秀的 Python 图书几乎不可避免要讲到鸭子类型。我最喜欢的两本书在本书第 1 版发布后都有更新：《Python 快速入门（第 3 版）》和 *Python in a Nutshell, 3rd ed*（Alex Martelli、Anna Ravenscroft 和 Steve Holden 著）。

Bill Venners 对 Guido van Rossum 的访谈讨论了动态类型的优缺点，访谈内容记录在“Contracts in Python: A Conversation with Guido van Rossum, Part IV”一文中。Martin Fowler 在“Dynamic Typing”一文中对这场辩论做了全面而深入的分析。Martin 还写了“Role Interface”一文，13.6.6 节提到过。那篇文章讲的虽然不是鸭子类型，但是与 Python 协议设计密切相关，比较了窄角色接口和类的广义公开接口。

与 Python 中静态类型（包括静态鸭子类型）相关的信息，最好的资源通常是 Mypy 文档。详见“Protocols and structural subtyping”一章。

下面的资料全与大鹅类型有关。《Python Cookbook（第 3 版）中文版》的 8.12 节讲了抽象基类的定义。该书写在 Python 3.4 之前，因

此没有使用推荐的句法声明抽象基类（子类化 `abc.ABC`），而是使用了 `metaclass` 关键字（本书第 24 章才用到）。除了这个小问题，8.12 节很好地讲解了抽象基类的主要功能。

《Python 标准库》中有一章讲了 `abc` 模块。那一章在 Python Module of the Week 网站中可以在线阅读。该书作者 Hellmann 使用的也是声明抽象基类的旧方式，即

`PluginBase(metaclass=abc.ABCMeta)`。从 Python 3.4 开始，可简化成 `PluginBase(abc.ABC)`。

对于抽象基类，多重继承不可避免，经常用到。基本的容器抽象基类 `Sequence`、`Mapping` 和 `Set` 扩展自 `Collection`，而 `Collection` 又扩展自多个抽象基类（参见图 13-4）。第 14 章会深入探讨这个重要话题。

“PEP 3119—Introducing Abstract Base Classes”讲解了抽象基类的基本原理。“PEP 3141—A Type Hierarchy for Numbers”引入了 `numbers` 模块中的一众抽象基类。Mypy 的 3186 号工单“`int` is not a `Number`?”展开了一场论战，讨论数字为什么不适合在静态类型检查中使用。Alex Waygood 在 StackOverflow 中写了一篇全面的解答，详述注解数值类型的各种方式。我会继续关注 3186 号工单的进展，但愿最后有一个圆满结局，让静态类型与大鹅类型兼容——本该如此。

杂谈

Python 静态类型的 MVP 之旅

我在 Thoughtworks 工作，这家公司是敏捷软件开发领域的全球领导者。在 Thoughtworks，我们经常建议客户创建和部署 MVP，即最简可用产品（minimal viable product）。按照我的同事 Paulo Caroli 在“Lean Inception”一文（发表在 Martin Fowler 的集体博客中）中给出的定义，最简可用产品是“为用户提供产品的简单版本，用于验证关键业务设想”。

自 2006 年以来，Guido van Rossum 和其他核心开发人员在设计和实现静态类型时一直遵循 MVP 策略。首先，Python 3.0 实现的“PEP 3107—Function Annotations”提供了非常有限的语义，只有为函数的参数和返回值附加注解的句法。这样做显然是为了实验并收集反馈——MVP 的关键优势。

8 年后，“PEP 484—Type Hints”提出并获得批准，在 Python 3.5 中实现，语言和标准库都没有变化，只是增加了标准库中其他部分均未依赖的 `typing` 模块。PEP 484 仅支持具有泛型的名义类型（类似于 Java），把具体的静态检查工作交给外部工具。那时，关键功能有所缺失，比如变量注解、内置泛型和协议。尽管有这些限制，但是这个最简可用的类型系统已经体现出了价值，足以吸引拥有特大型 Python 基准代码的公司（例如 Dropbox、谷歌和 Facebook）投入使用，吸引专业的 IDE（例如 PyCharm、Wing 和 VS Code）提供支持。

“PEP 526—Syntax for Variable Annotations”是演进路上的第一步，Python 3.6 对解释器做出了改动。为了支持“PEP 563—Postponed Evaluation of Annotations”和“PEP 560—Core support for typing module and generic types”，Python 3.7 对解释器做了更多改动。加上“PEP 585—Type Hinting Generics In Standard Collections”，Python 3.9 中内置的和标准库中的容器开始接受泛化类型提示。

那几年，一些 Python 用户，包括我，并没有对类型提起兴趣。学习 Go 语言之后，我更加认为 Python 缺少静态鸭子类型是不可理解的，毕竟这门语言的核心优势就是鸭子类型。

可这就是 MVP 策略的本质啊，最简可用产品或许不能让所有潜在用户满意，但是实现起来不费劲，而且可以借助实际使用中得到的反馈指导下一步开发。

如果说我们从 Python 3 中学到了什么，那肯定是渐进式开发比一股脑发布新功能安全。很高兴我们不用等到 Python 4（如果真能

等到的话) 才能引起大公司的注意, 因为大公司已经发现, 与静态类型带来的好处相比, 增加的那点儿复杂性不值一提。

流行语言实现类型的方式

图 13-8 稍微修改了类型图 (参见图 13-1), 加上了支持各种类型实现方式的流行语言。

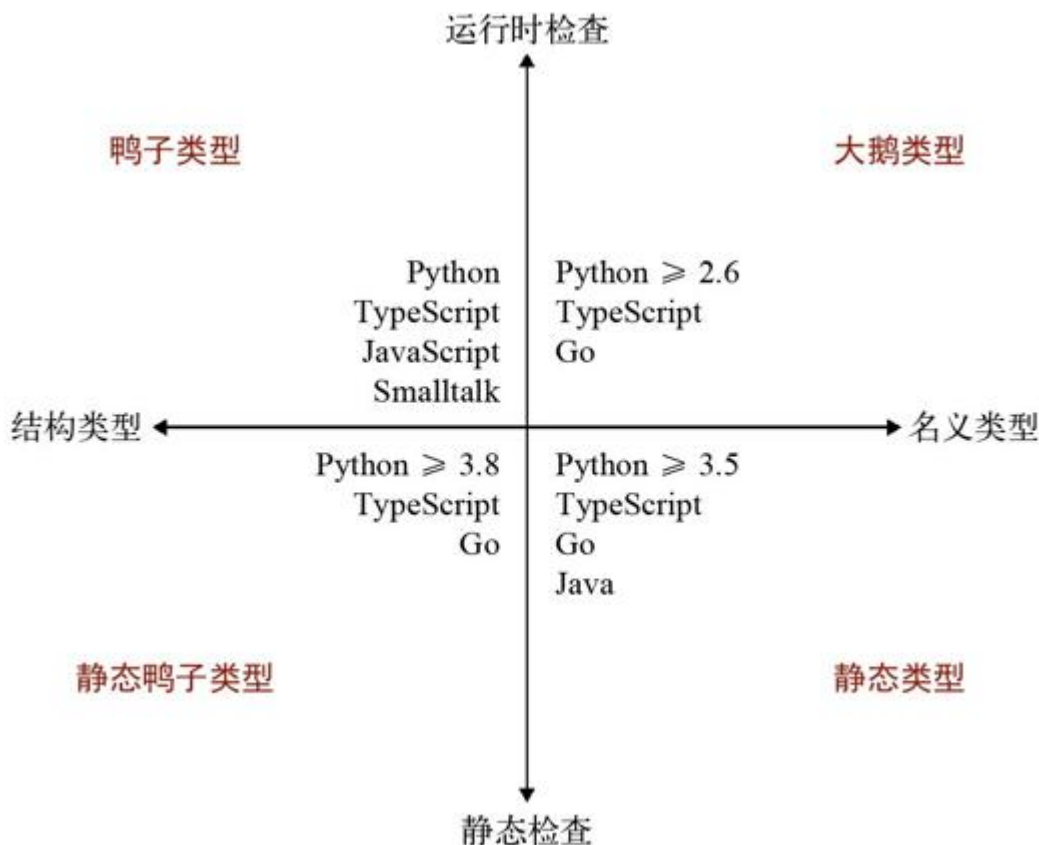


图 13-8: 类型检查的 4 种方式, 以及支持各种方式的部分语言

在我随机考察的有限样本中, 只有 TypeScript 和 Python 3.8 及以上版本支持全部 4 种方式。

Go 语言显然是一门像 Pascal 那样传统的静态类型语言, 但是它开创了静态鸭子类型的先河——至少在当今广泛使用的语言中是这样。我还把 Go 语言放在了大鹅类型象限中, 因为它的类型断言允许在运行时检查和适应不同的类型。

如果我在 2000 年画一个类似的图，那么只有鸭子类型和静态类型象限中有语言。据我所知，20 年前没有支持静态鸭子类型或大鹅类型的语言。可以看到，4 个象限中都至少有 3 门流行语言，这表明很多人发现了 4 种类型实现方式各自的价值。

猴子补丁

猴子补丁的名声不太好。如果滥用，则会导致系统难以理解和维护。补丁通常与目标紧密耦合，因此很脆弱。还有一个问题是，打了猴子补丁的两个库可能相互牵绊，因为第二个库可能撤销了第一个库的补丁。

不过猴子补丁也有它的作用，例如可以在运行时让类实现协议。适配器设计模式通过实现全新的类解决了这种问题。

为 Python 打猴子补丁不难，但是有些局限。与 Ruby 和 JavaScript 不同，Python 不允许为内置类型打猴子补丁。其实，我觉得这是优点，因为这样可以确保 `str` 对象的方法始终是那些。这一局限能减少外部库打的补丁出现冲突的概率。

接口中的隐喻和习惯用法

隐喻能打破壁垒，让人更易于理解。使用“栈”和“队列”描述基本的数据类型就有这样的功效：这两个词清楚地道出了添加或删除元素的方式。另外，Alan Cooper 等人在《About Face 4: 交互设计精髓》一书中写道：

严格奉行隐喻设计毫无必要，却把界面死死地与物理世界的运行机制捆绑在一起。

他说的是用户界面，但对 API 同样适用。不过 Cooper 同意，当“真正合适的”隐喻“正中下怀”时，可以使用隐喻（他用的词是“正中下怀”，因为合适的隐喻可遇不可求）。我觉得本章用宾果机做比喻是合适的，我相信自己。

我读过不少 UI 设计方面的书，《About Face：交互设计精髓》是最好的。我从 Cooper 的书中学到的最宝贵的知识是，不把隐喻当作设计范式，而代之以“习惯用法的界面”。

前面说过，Cooper 说的不是 API，但是，越深入思考他的观点，我越觉得可以将之运用到 Python 中。Python 语言的基本协议就是 Cooper 所说的“习惯用法”。知道“序列”是什么之后，可以把这些知识应用到不同的场合。这正是本书的主要目的：着重讲解这门语言的基本惯用法，让你的代码简洁、高效且可读，把你打造成能流畅写出 Python 代码的程序员。

第 14 章 继承：瑕瑜互见

.....我们需要推翻过去，提出一套全新的继承理论（始终如此）。例如，继承和实例化（一种继承）混淆了语用（例如分解代码以节省空间）和语义（用到它的任务太多了，例如特化、泛化、分化等）。

——Alan Kay
“The Early History of Smalltalk”¹

¹Alan Kay, “The Early History of Smalltalk”, 也可以在线阅读。感谢我的朋友 Christiano Anderson 在我撰写本章时告诉我这篇参考文献。

本章探讨继承和子类化。阅读本章之前需要对这两个概念有基本的了解，你可以阅读 Python 官方教程，也可以利用使用其他主流面向对象语言（例如 Java、C# 或 C++）积累的经验。本章重点讲解以下 4 个 Python 特色功能：

- `super()` 函数；
- 子类化内置类型的缺点；
- 多重继承和方法解析顺序；
- 混入类。

多重继承指一个类可以有多个基类。C++ 支持多重继承，Java 和 C# 则不支持。许多人认为多重继承得不偿失。Java 认为早期的 C++ 基准代码往往滥用多重继承，因此故意没有采纳。

本章会向从未使用过多重继承的人介绍多重继承，并提供一些指导，说明在必须使用继承时如何应对单一继承或多重继承。

2021 年，人们开始强烈反对过度使用一般意义上的继承（不仅仅是多重继承），因为超类和子类紧密耦合。紧密耦合意味着对程序某一部

分的更改可能会对其他部分产生意想不到的深远影响，从而使系统变得脆弱且难以理解。

然而，仍然需要维护使用复杂的类层次结构设计的系统，何况有些框架强制必须使用继承（有时甚至是多重继承）。

本章将通过标准库、Django Web 框架和 Tkinter GUI 工具包说明多重继承的实际运用。

14.1 本章新增内容

Python 没有为本章讨论的话题新增功能，不过根据第 2 版技术审校（尤其是 Leonardo Rochaël 和 Caleb Hattingh）的反馈，我对本章做了大幅改动。

我重写了 14.2 节，重点介绍内置函数 `super()`。另外，还修改了 14.4 节中的示例，深入探讨 `super()` 函数是如何支持**协作多重继承**的。

14.5 节也是新增的。14.6 节重新做了编排，在 Django 和 Tkinter 复杂的继承层次结构之前增加了标准库中较为简单的混入示例。

通过本章标题可以看出，继承的缺点也是本章的主题之一。由于越来越多的开发人员认为继承是一个严重的问题，因此本章在 14.8 节和 14.9 节末尾分别增加了几个段落，指出如何避免继承。

下面先来揭开 `super()` 函数的神秘面纱。

14.2 `super()` 函数

坚持使用内置函数 `super()` 是确保面向对象的 Python 程序可维护性的基本要求。

子类中覆盖超类的方法通常要调用超类中相应的方法。在 `collections` 模块的文档中，“OrderedDict Examples and Recipes”一节有一个示例，演示了推荐的方式。²

²我只修改了示例中的文档字符串，因为原来的文档字符串容易让人误解。原来的文档字符串是“Store items in the order the keys were last added”（按照增加键的顺序存储项），这显然与类名 `LastUpdatedOrderedDict` 表达的意思不符。

```
class LastUpdatedOrderedDict(OrderedDict):
    """按照更新顺序存储项"""

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

为达目的，`LastUpdatedOrderedDict` 覆盖 `__setitem__` 方法，做了以下两件事。

- 通过 `super().__setitem__` 调用超类中对应的方法，插入或更新键-值对。
- 调用 `self.move_to_end`，确保最后更新的 `key` 出现在最后。

调用被覆盖的 `__init__` 方法尤其重要，可以让超类完成它负责的初始化任务。



如果学过 Java 面向对象编程，那么想必你应该知道，Java 构造方法自动调用超类不接受参数的构造方法。Python 则不会这么做。因此，务必习惯像下面这样做。

```
def __init__(self, a, b):
    super().__init__(a, b)
    ... # 其他初始化代码
```

你或许见过不使用 `super()` 函数而是直接在超类上调用方法的代码，如下所示。

```
class NotRecommended(OrderedDict):
    """这是一个反例! """

    def __setitem__(self, key, value):
        OrderedDict.__setitem__(self, key, value)
        self.move_to_end(key)
```

这么做不是不可以，但是不推荐，原因有二。其一，硬编码了基类。`OrderedDict` 名称不仅出现在 `class` 语句中，还出现在 `__setitem__` 方法内。如果后来有人修改了 `class` 语句，更换了基类或者又加了一个，那么说不定会忘记更新 `__setitem__` 方法的主体，埋下 bug。

其二，`super` 实现的逻辑能处理多重继承涉及的类层次结构（详见 14.4 节）。最后，看一下在 Python 2 中如何调用 `super`，旧句法接受两个参数，这可以给我们一定启发。

```
class LastUpdatedOrderedDict(OrderedDict):
    """在Python 2和Python 3中都能正常运行"""

    def __setitem__(self, key, value):
        super(LastUpdatedOrderedDict, self).__setitem__(key,
value)
        self.move_to_end(key)
```

现在，`super` 的两个参数都是可选的。Python 3 字节码编译器通过 `super()` 调用周围的上下文自动提供那两个参数。两个参数的作用如下。

`type`

从哪里开始搜索实现所需方法的超类。默认为 `super()` 调用所在的方法所属的类。

`object_or_type`

接收方法调用的对象（调用实例方法时）或类（调用类方法时）。在实例方法中调用 `super()` 时，默认为 `self`。

无论是我们自己还是编译器提供这两个参数，`super()` 调用都返回一个动态代理对象，在 `type` 参数指定的超类中寻找一个方法（例如这里的 `__setitem__`），把它绑定到 `object_or_type` 上，因此调用那个方法时不用显式传入接收者（`self`）。

在 Python 3 中，依然可以显式为 `super()` 提供第一个参数和第二个参数。³ 但是，只有在特殊情况下才必须这么做，例如测试或调试时跳过部分 MRO，或者绕开不希望从超类得到的行为。

³也可以只提供第一个参数，但是这么做没什么意义，可能不久就会弃用，当然这要征得最初提出 `super()` 的 Guido van Rossum 的同意。详细讨论参见“Is it time to deprecate unbound super methods?”。

下面讨论子类化内置类型的注意事项。

14.3 子类化内置类型很麻烦

在 Python 的早期版本中，内置类型（例如 `list` 或 `dict`）不能子类化。从 Python 2.2 开始，内置类型可以子类化了，但是有一个重要的注意事项：内置类型（使用 C 语言编写）通常不调用用户定义的类覆盖的方法。PyPy 的文档使用简明扼要的语言描述了这个问题，可以参见“Differences between PyPy and CPython”中的“Subclasses of built-in types”一节。

关于内置类型的子类覆盖的方法会不会隐式调用，CPython 没有制定官方规则。基本上，内置类型的方法不会调用子类覆盖的方法。例如，`dict` 的子类覆盖的 `__getitem__()` 方法不会被内置类型的 `get()` 方法调用。

示例 14-1 说明了这个问题。

示例 14-1 内置类型 `dict` 的 `__init__` 方法和 `__update__` 方法会忽略我们覆盖的 `__setitem__` 方法

```

>>> class DoppelDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2) ❶
...
>>> dd = DoppelDict(one=1) ❷
>>> dd
{'one': 1}
>>> dd['two'] = 2 ❸
>>> dd
{'one': 1, 'two': [2, 2]}
>>> dd.update(three=3) ❹
>>> dd
{'three': 3, 'one': 1, 'two': [2, 2]}

```

❶ `DoppelDict.__setitem__` 方法把存入的值重复两次（只是为了提供易于观察的效果）。它把职责委托给了超类。

❷ 继承自 `dict` 的 `__init__` 方法显然忽略了我们覆盖的 `__setitem__` 方法：'one' 的值没有重复。

❸ `[]` 运算符调用我们覆盖的 `__setitem__` 方法，按预期那样工作：'two' 对应的是两个重复的值，即 `[2, 2]`。

❹ 继承自 `dict` 的 `update` 方法也没有使用我们覆盖的 `__setitem__` 方法：'three' 的值没有重复。

内置类型的这种行为违背了面向对象编程的一个基本原则：应始终从实例（`self`）所属的类开始搜索方法，即使在超类实现的类中调用也是如此。这种行为叫作“晚期绑定”（late binding）。在 Alan Kay（因 Smalltalk 出名）看来，这是面向对象编程的关键功能：对于 `x.method()` 形式的调用，具体调用的方法必须在运行时根据接收者 `x` 所属的类确定。⁴ 在这种糟糕的局面中，`__missing__` 方法却能按预期工作（参见 3.5.3 节）。

⁴有趣的是，C++ 有虚方法和非虚方法的概念。虚方法是晚期绑定，而非虚方法在编译时绑定。虽然使用 Python 写出的每个方法都像虚方法那样在晚期绑定，但是以 C 语言实现的内置对象拥有的方法似乎默认是非虚方法（至少对 CPython 来说是这样）。

不只实例内部的调用有这个问题（`self.get()` 不调用 `self.__getitem__()`），内置类型的方法调用的其他类的方法如果被覆盖了，则也不会被调用。示例 14-2 是改编自 PyPy 文档的一个示例。

示例 14-2 `dict.update` 方法会忽略 `AnswerDict.__getitem__` 方法

```
>>> class AnswerDict(dict):
...     def __getitem__(self, key): ❶
...         return 42
...
>>> ad = AnswerDict(a='foo') ❷
>>> ad['a'] ❸
42
>>> d = {}
>>> d.update(ad) ❹
>>> d['a'] ❺
'foo'
>>> d
{'a': 'foo'}
```

- ❶ 不管传入什么键，`AnswerDict.__getitem__` 方法始终返回 42。
- ❷ `ad` 是 `AnswerDict` 实例，以 `('a', 'foo')` 键-值对初始化。
- ❸ `ad['a']` 返回 42，符合预期。
- ❹ `d` 是 `dict` 实例，使用 `ad` 中的值更新 `d`。
- ❺ `dict.update` 方法忽略了 `AnswerDict.__getitem__` 方法。



直接子类化内置类型（例如 `dict`、`list` 或 `str`）容易出错，因为内置类型的方法通常会忽略用户覆盖的方法。不要子类化内置类型，用户自己定义的类应该继承 `collections` 模块中

的类，例如 `UserDict`、`UserList` 和 `UserString`。这些类做了特殊设计，因此易于扩展。

如果不子类化 `dict`，而是子类化 `collections.UserDict`，那么示例 14-1 和示例 14-2 中暴露的问题就能迎刃而解了，如示例 14-3 所示。

示例 14-3 `DoppelDict2` 和 `AnswerDict2` 能像预期那样使用，因为它们扩展的是 `UserDict`，而不是 `dict`

```
>>> import collections
>>>
>>> class DoppelDict2(collections.UserDict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)
...
>>> dd = DoppelDict2(one=1)
>>> dd
{'one': [1, 1]}
>>> dd['two'] = 2
>>> dd
{'two': [2, 2], 'one': [1, 1]}
>>> dd.update(three=3)
>>> dd
{'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
>>>
>>> class AnswerDict2(collections.UserDict):
...     def __getitem__(self, key):
...         return 42
...
>>> ad = AnswerDict2(a='foo')
>>> ad['a']
42
>>> d = {}
>>> d.update(ad)
>>> d['a']
42
>>> d
{'a': 42}
```

为了衡量子类化内置类型所需的额外工作量，我做了一个实验，把示例 3-9 中原本继承 `UserDict` 的 `StrKeyDict` 类改为继承 `dict`。为了让新实现通过整个测试套件，要实现 `__init__` 方法、`get` 方法

和 `update` 方法，因为继承 `dict` 的版本拒绝与覆盖的 `__missing__` 方法、`__contains__` 方法和 `__setitem__` 方法合作。示例 3-9 中那个 `UserDict` 子类有 16 行代码，而这个实验中的 `dict` 子类有 33 行代码。⁵

⁵如果感到好奇，可以查看本书代码中的 `14-inheritance/strkeydict_dictsub.py` 文件。

综上所述，本节所述的问题只发生在 C 语言实现的内置类型内部的方法委托上，而且只影响直接继承内置类型的类。如果子类化使用 Python 编写的类（例如 `UserDict` 或 `MutableMapping`），则不会受此影响。⁶

⁶顺便说一下，在这方面，PyPy 的行为比 CPython“正确”，不过会导致微小的不兼容性。详情参见“Differences between PyPy and CPython”。

与多重继承有关的另一个问题是，如果一个类有两个超类，而且超类中有同名属性，那么 Python 如何确定 `super().attr` 该使用哪个属性呢？14.4 节将对此进行解答。

14.4 多重继承和方法解析顺序

任何实现多重继承的语言都要处理潜在的命名冲突，这种冲突由超类实现同名方法时引起。我们称之为“菱形问题”（diamond problem），如图 14-1 和示例 14-4 所示。

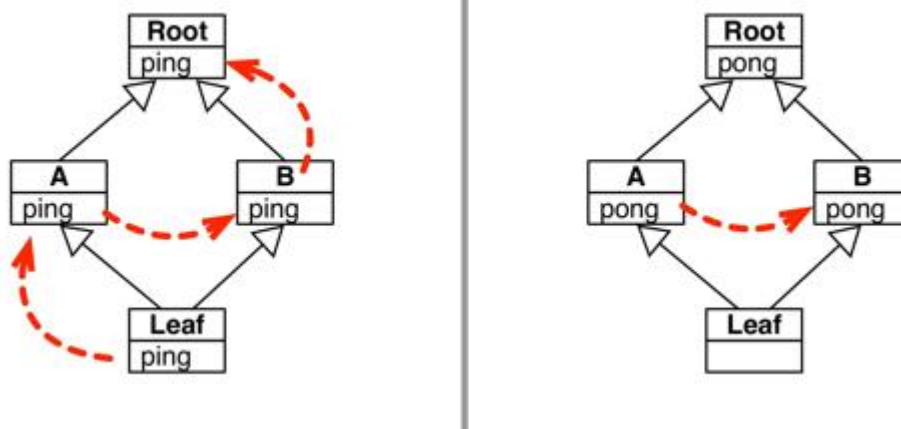


图 14-1: (左) leaf1.ping() 调用的唤醒过程; (右) leaf1.pong() 调用的唤醒过程

示例 14-4 diamond.py: 图 14-1 中的 Leaf 类、A 类、B 类和 Root 类

```
class Root: ❶
    def ping(self):
        print(f'{self}.ping() in Root')

    def pong(self):
        print(f'{self}.pong() in Root')

    def __repr__(self):
        cls_name = type(self).__name__
        return f'<instance of {cls_name}>'

class A(Root): ❷
    def ping(self):
        print(f'{self}.ping() in A')
        super().ping()

    def pong(self):
        print(f'{self}.pong() in A')
        super().pong()

class B(Root): ❸
    def ping(self):
        print(f'{self}.ping() in B')
```

```
        super().ping()

    def pong(self):
        print(f'{self}.pong() in B')

class Leaf(A, B): ❹
    def ping(self):
        print(f'{self}.ping() in Leaf')
        super().ping()
```

❶ Root 不仅提供了 ping 方法和 pong 方法，为了输出更易读的表现形式，还实现了 `__repr__` 方法。

❷ A 类中的 ping 方法和 pong 方法都调用了 `super()`。

❸ B 类中只有 ping 方法调用了 `super()`。

❹ Leaf 类只实现了 ping 方法，而且该方法调用了 `super()`。

下面来看在 Leaf 实例上调用 ping 方法和 pong 方法的效果，如示例 14-5 所示。

示例 14-5 在 Leaf 对象上调用 ping 和 pong

```
>>> leaf1 = Leaf() ❶
>>> leaf1.ping()    ❷
<instance of Leaf>.ping() in Leaf
<instance of Leaf>.ping() in A
<instance of Leaf>.ping() in B
<instance of Leaf>.ping() in Root
>>> leaf1.pong()    ❸
<instance of Leaf>.pong() in A
<instance of Leaf>.pong() in B
```

❶ leaf1 是 Leaf 实例。

❷ 调用 `leaf1.ping()`，唤醒 Leaf、A、B 和 Root 中的 ping 方法，因为前 3 个类中的 ping 方法都调用了 `super().ping()`。

③ 调用 `leaf1.pong()`，唤醒继承树上 A 中的 `pong`，而它又调用 `super.pong()`，唤醒了 `B.pong`。

示例 14-5 和图 14-1 所示的唤醒过程由以下两个因素决定。

- `Leaf` 类的方法解析顺序。
- 各方法中使用的 `super()`。

每个类都有名为 `__mro__` 的属性，它的值是一个元组，按照方法解析顺序列出各个超类，从当前类一直到 `object` 类。⁷ `Leaf` 类的 `__mro__` 属性如下所示。

⁷每个类也都有 `.mro()` 方法，不过这是元类编程（参见 24.2 节）的高级功能。普通用途的类，起作用的是 `__mro__` 属性的内容。

```
>>> Leaf.__mro__ # doctest:+NORMALIZE_WHITESPACE
(<class 'diamond1.Leaf'>, <class 'diamond1.A'>, <class
'diamond1.B'>,
 <class 'diamond1.Root'>, <class 'object'>)
```



图 14-1 可能会让你误以为方法解析顺序使用的是广度优先搜索（breadth-first search），其实这只是图中的类层次结构给人的错觉。方法解析顺序使用公开发布的 C3 算法计算。Michele Simionato 写的“The Python 2.3 Method Resolution Order”一文对该算法在 Python 中的运用做了详细说明。这篇文章一般人读不懂，Simionato 写道：“除非大量使用多重继承，或者继承关系不同寻常，否则无须了解 C3 算法，因此也不用阅读这篇文章。”

方法解析顺序只决定唤醒顺序，至于各个类中相应的方法是否被唤醒，则取决于实现方法时有没有调用 `super()`。

以实验中的 `pong` 方法为例。由于 `Leaf` 类没有覆盖该方法，因此调用 `leaf1.pong()` 唤醒的是 `Leaf.__mro__` 中下一个类（A 类）实现的 `pong` 方法。A.pong 方法调用了 `super().pong()`。方法解析

顺序的下一个类是 B，因此 B.pong 被唤醒。但是，因为 B.pong 方法没有调用 `super().pong()`，所以唤醒过程到此结束。

方法解析顺序不仅考虑继承图，还考虑子类声明罗列超类的顺序。也就是说，在 diamond.py 文件（参见示例 14-4）中，如果把 Leaf 类声明为 `Leaf(B, A)`，那么在 `Leaf.__mro__` 中，B 类将出现在 A 类前面。这会影响 ping 方法的唤醒顺序，而且 `leaf1.pong()` 将通过继承树唤醒 B.pong，但是不唤醒 A.pong 和 Root.pong，因为 B.pong 没有调用 `super()`。

调用 `super()` 的方法叫**协作方法**（cooperative method）。利用协作方法可以实现**协作多重继承**。这两个术语就是字面意思，Python 中的多重继承涉及多个方法的协作。在 B 类中，ping 是协作方法，而 pong 则不是。



非协作方法可能导致不易发现的 bug。很多程序员在读过示例 14-4 之后，可能想当然地认为，既然 A.pong 调用了 `super.pong()`，那么最终也会唤醒 Root.pong。但是，如果先唤醒的是 B.pong，那么到此就结束了。鉴于此，才建议非根类中的每一个方法 m 都调用 `super().m()`。

协作的方法必须具有兼容的签名，因为你永远不知道 A.ping 是在 B.ping 之前还是之后调用。同时继承 A 和 B 的类，其唤醒过程取决于子类声明罗列 A 和 B 的顺序。

Python 是动态语言，因此 `super()` 与方法解析顺序的交互也是动态的。这种动态行为可能导致出人意料的结果，如示例 14-6 所示。

示例 14-6 diamond2.py：演示 `super()` 动态行为的类

```
from diamond import A ❶

class U(): ❷
    def ping(self):
```

```
        print(f'{self}.ping() in U')
        super().ping() ❸

class LeafUA(U, A): ❹
    def ping(self):
        print(f'{self}.ping() in LeafUA')
        super().ping()
```

❶ A 类来自 diamond.py（参见示例 14-4）。

❷ U 类与 diamond 模块中的 A 或 Root 没有关系。

❸ `super().ping()` 会做些什么呢？答案是，视情况而定。请继续往下读。

❹ LeafUA 按顺序继承 U 和 A。

创建一个 U 实例，再调用 ping 将报错。

```
>>> u = U()
>>> u.ping()
Traceback (most recent call last):
...
AttributeError: 'super' object has no attribute 'ping'
```

`super()` 返回的 'super' object 没有 'ping' 属性，因为 U 的方法解析顺序有两个类，即 U 和 object，而后者没有名为 'ping' 的属性。

然而，U.ping 方法并不是毫无用处。请看下面的测试。

```
>>> leaf2 = LeafUA()
>>> leaf2.ping()
<instance of LeafUA>.ping() in LeafUA
<instance of LeafUA>.ping() in U
<instance of LeafUA>.ping() in A
<instance of LeafUA>.ping() in Root
>>> LeafUA.__mro__ # doctest:+NORMALIZE_WHITESPACE
(<class 'diamond2.LeadUA'>, <class 'diamond2.U'>,
 <class 'diamond.A'>, <class 'diamond.Root'>, <class
'object'>)
```

LeafUA 中的 `super().ping()` 调用唤醒了 `U.ping`，而 `U.ping` 也调用 `super().ping()` 唤醒了 `A.ping`，最终唤醒了 `Root.ping`。

注意，LeafUA 的基类是 (U, A) (按此顺序)。如果基类是 (A, U)，那么 `leaf2.ping()` 肯定不会唤醒 `U.ping`，因为 `A.ping` 中的 `super().ping()` 将唤醒 `Root.ping`，而 `Root.ping` 没有调用 `super()`。

在真实的程序中，U 这样的类可以作为**混入类**使用。混入类意在与多重继承中的其他类结合在一起使用，提供额外的功能。14.5 节会研究混入类。

结束对方法解析顺序的讨论之前，来看看图 14-2，该图展示了 Python 标准库中 GUI 工具包 Tkinter 复杂的多重继承图。

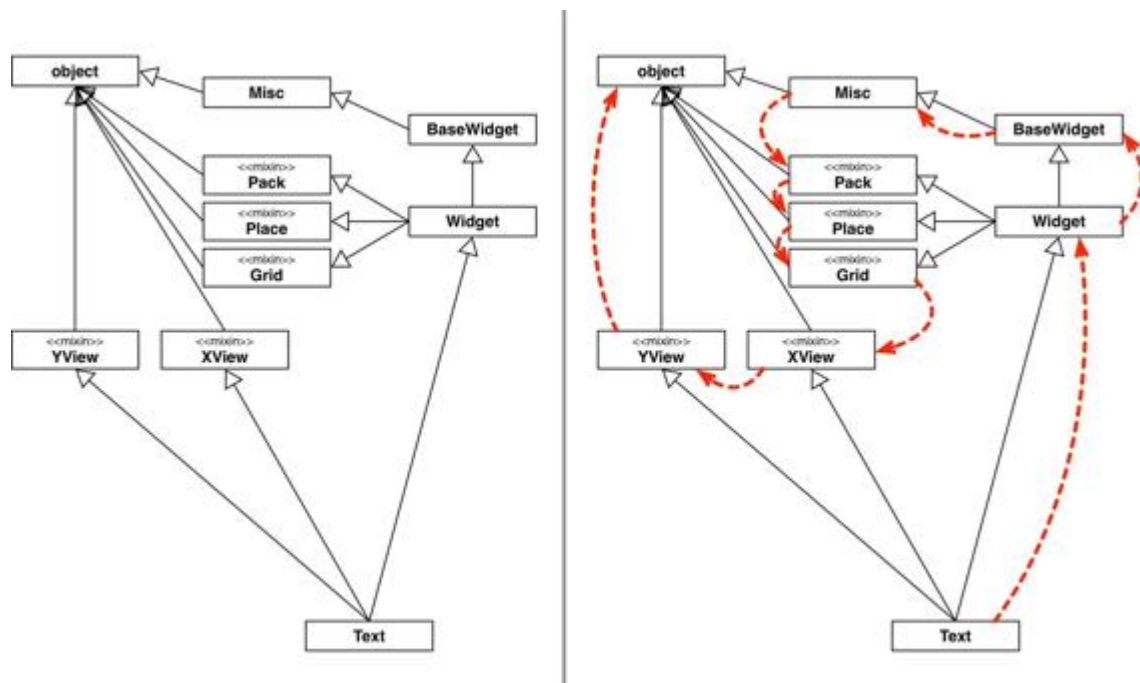


图 14-2：（左）Tkinter 中 `Text` 小组件类及其超类的 UML 类图；（右）使用虚线箭头表示 `Text.__mro__`

研究这幅图时，从底部的 `Text` 类开始。这个类实现了功能完善的多行可编辑文本小组件，其自身就有丰富的功能，不过也从其他类继承了很多方法。左边是常规的 UML 类图，右边加入了一些箭头，表示方法解析顺序。示例 14-7 展示了其定义的便利函数 `print_mro` 的输出。

示例 14-7 `tkinter.Text` 的方法解析顺序

```
>>> def print_mro(cls):
...     print(', '.join(c.__name__ for c in cls.__mro__))
>>> import tkinter
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView,
object
```

14.5 节将讨论混入类。

14.5 混入类

混入类在多重继承中会连同其他类一起被子类化。混入类不能作为具体类的唯一基类，因为混入类不为具体对象提供全部功能，而是增加或定制子类或同级类的行为。



在 Python 和 C++ 中，混入类只是一种约定，语言层面没有显式支持。Ruby 支持显式定义用于混入的模块，即可以引入的为类增加功能的方法集合。C#、PHP 和 Rust 实现的性状 (trait) 也是一种显式混入。

下面举一个简单而实用的混入类示例。

不区分大小写的映射

示例 14-8 中定义的 `UpperCaseMixin` 类在增加或查询键时会把字符串键转换成大写形式，实现一种键不区分大小写的映射。

示例 14-8 uppermixin.py: UpperCaseMixin 类支持不区分大小写的映射

```
import collections

def _upper(key): ❶
    try:
        return key.upper()
    except AttributeError:
        return key

class UpperCaseMixin: ❷
    def __setitem__(self, key, item):
        super().__setitem__(_upper(key), item)

    def __getitem__(self, key):
        return super().__getitem__(_upper(key))

    def get(self, key, default=None):
        return super().get(_upper(key), default)

    def __contains__(self, key):
        return super().__contains__(_upper(key))
```

❶ 这个辅助函数接受的 `key` 参数可以是任何类型，并会尝试返回 `key.upper()` 得到的结果；如果失败，则返回未经修改的 `key`。

❷ 这个混入类实现了映射的 4 个基本方法，总是调用 `super()`，传入尽量转换成大写形式的 `key`。

由于 `UpperCaseMixin` 中的每个方法都调用了 `super()`，因此这个混入类会依赖一个同级类，该类实现或继承了签名相同的方法。为了让混入类发挥作用，在子类的方法解析顺序中，它要出现在其他类前面。也就是说，在类声明语句中，混入类必须出现在基类元组的第一位。示例 14-9 给出了两个例子。

示例 14-9 uppermixin.py: 使用 `UpperCaseMixin` 的两个类

```
class UpperDict(UpperCaseMixin, collections.UserDict): ❶
    pass
```

```
class UpperCounter(UpperCaseMixin, collections.Counter): ❷
    """一个特殊的计数器，字符串键是大写形式""" ❸
```

❶ UpperDict 自身无须实现，不过 UpperCaseMixin 必须是第一个基类，否则将调用 UserDict 中的方法。

❷ UpperCaseMixin 也可供 Counter 使用。

❸ 与其只放 pass，不如提供一个文档字符串，满足 class 语句句法对类主体的要求。

下面的 doctest 摘自 uppermixin.py，针对 UpperDict。

```
>>> d = UpperDict([('a', 'letter A'), (2, 'digit two')])
>>> list(d.keys())
['A', 2]
>>> d['b'] = 'letter B'
>>> 'b' in d
True
>>> d['a'], d.get('B')
('letter A', 'letter B')
>>> list(d.keys())
['A', 2, 'B']
```

下面是 UpperCounter 的简单演示。

```
>>> c = UpperCounter('BaNaNA')
>>> c.most_common()
[('A', 3), ('N', 2), ('B', 1)]
```

UpperDict 和 UpperCounter 看起来像施了魔法一样，但是为了混入 UpperCaseMixin，我对 UserDict 和 Counter 的代码做了深入研究。

比如说，第一次编写的 UpperCaseMixin 没有提供 get 方法，只支持 UserDict，不支持 Counter。UserDict 类从 collections.abc.Mapping 继承了 get，而 get 会调用我实现的 __getitem__。但是，当 UpperCounter 被初始化之后，键没有变

成大写形式。这是因为，`Counter.init` 使用的是 `Counter.update`，而 `Counter.update` 依赖从 `dict` 继承的 `get` 方法。然而，`dict` 类的 `get` 方法不调用 `__getitem__`。这就是 3.5.3 节所讨论问题的关键。这再一次体现了利用继承的程序（即使规模不大）是多么脆弱且令人费解。

14.6 节将通过几个例子来说明多重继承（往往还涉及混入类）。

14.6 多重继承的实际运用

《设计模式》一书中的大部分代码是用 C++ 编写的，但是只有适配器一个模式用到了多重继承。在 Python 中，多重继承也不常用，不过在实际中也会用到，本节将具体说明。

14.6.1 抽象基类也是混入类

在 Python 标准库中，多重继承最明显的用途是用于 `collections.abc` 包。这没什么好争议的，即便 Java 中的接口也支持多重继承，而抽象基类声明的就是接口，只是偶有提供方法的具体实现。⁸

⁸如前所述，Java 8 也允许接口提供方法实现。在 Java 官方教程中，这个功能叫作“默认方法”。

Python 官方文档 `collections.abc` 使用“**混入方法**”来称呼很多容器抽象基类中实现的具体方法。提供混入方法的抽象基类扮演两个角色，既是接口定义，也是混入类。例如，`collections.UserDict` 的实现就依赖 `collections.abc.MutableMapping` 提供的多个混入方法。

14.6.2 `ThreadingMixIn` 和 `ForkingMixIn`

http.server 包提供了 HTTPServer 类和 ThreadingHTTPServer 类。后者是 Python 3.7 中新增的。文档描述如下。

```
class
http.server.ThreadingHTTPServer(server_address,
RequestHandlerClass)
```

这个类与 HTTPServer 作用相同，不过通过 ThreadingMixIn 使用线程处理请求。适合处理 Web 浏览器预先打开的套接字，如果使用 HTTPServer 处理，则等待时间不确定。

在 Python 3.10 中，ThreadingHTTPServer 类的完整源码如下所示。

```
class ThreadingHTTPServer(socketserver.ThreadingMixIn,
HTTPServer):
    daemon_threads = True
```

socketserver.ThreadingMixIn 的源码有 38 行（包括注释和文档字符串），示例 14-10 只给出了梗概。

示例 14-10 Python 3.10 中 Lib/socketserver.py 文件的部分内容

```
class ThreadingMixIn:
    """混入类处理新线程中的每个请求"""

    # 省略8行

    def process_request_thread(self, request, client_address): ❶
        ... # 省略6行

    def process_request(self, request, client_address): ❷
        ... # 省略8行

    def server_close(self): ❸
        super().server_close()
        self._threads.join()
```

❶ `process_request_thread` 没有调用 `super()`，因为这是一个新方法，不是对现有方法的覆盖。该方法的实现调用了 `HTTPServer` 提供或继承的 3 个实例方法。

❷ 覆盖 `HTTPServer` 从 `socketserver.BaseServer` 继承的 `process_request` 方法，启动一个线程，并把具体工作委托给运行在该线程中的 `process_request_thread`。没有调用 `super()`。

❸ `server_close` 调用 `super().server_close()` 以停止接收请求，然后等待 `process_request` 启动的线程完成工作。

`socketserver` 模块的文档会把 `ThreadingMixIn` 和 `ForkingMixIn` 放在一起。后者基于 `os.fork()` 为服务器提供并发支持。`os.fork()` 是用于启动子进程的 API，在兼容 POSIX 的 Unix 类系统中可用。

14.6.3 Django 泛化视图混入类



阅读本节无须了解 Django。我将以这个框架的一小部分为例说明多重继承，所有必要的背景知识都会讲清楚，不过你要有一定的服务器端 Web 开发经验，任何语言或框架都可以。

在 Django 中，一个视图是一个可调用对象，接受表示 HTTP 请求的 `request` 参数，返回表示 HTTP 响应的对象。本节关注的是返回的不同响应。响应既可以像重定向一样简单，没有内容主体，也可以像在线商店的目录页一样复杂，使用一个 HTML 模板渲染，列出多个商品，带有购买按钮和指向详情页面的链接。

最初，Django 提供了一系列函数（叫作泛化视图），以实现一些常见需求。例如，很多网站需要展示搜索结果，搜索到的信息可能占据多页，每一个条目都带有链接，指向详情页面。为此，Django 提供了一个列表视图和一个详情视图，前者用于渲染搜索结果，后者会为各个条目生成详情页面。

然而，这些泛化视图是函数，不可扩展。如果你想要的是类似的效果，与泛化列表视图不完全一样，就要自己从头开始实现。

Django 1.3 不仅引入了基于类的视图，还提供了一系列泛化视图类，包括基类、混入类和拿来即用的具体类。在 Django 3.2 中，基类和混入类位于 `django.views.generic` 包的 `base` 模块中，如图 14-3 所示。图中上半部分是职责差异巨大的两个类：`View` 和 `TemplateResponseMixin`。



如果想研究这些类，可以打开 [Classy Class-Based Views](#) 网站。在这个网站中，可以浏览各个类、了解各个类提供的全部方法（继承的方法、覆盖的方法和增加的方法）、查看图表、浏览文档，以及跳转到 GitHub 中的源码。

`View` 是所有视图的基类（可能是抽象基类），提供了核心功能，例如 `dispatch` 方法。该方法会把职责委托给具体子类的 `get`、`head`、`post` 等方法，处理不同的 HTTP 动词。⁹ `RedirectView` 类只继承 `View`，可以看到，它实现了 `get`、`head`、`post` 等方法。

⁹Django 程序员知道，类方法 `as_view` 是 `View` 接口最明显的部分，不过它与这里的讨论无关。

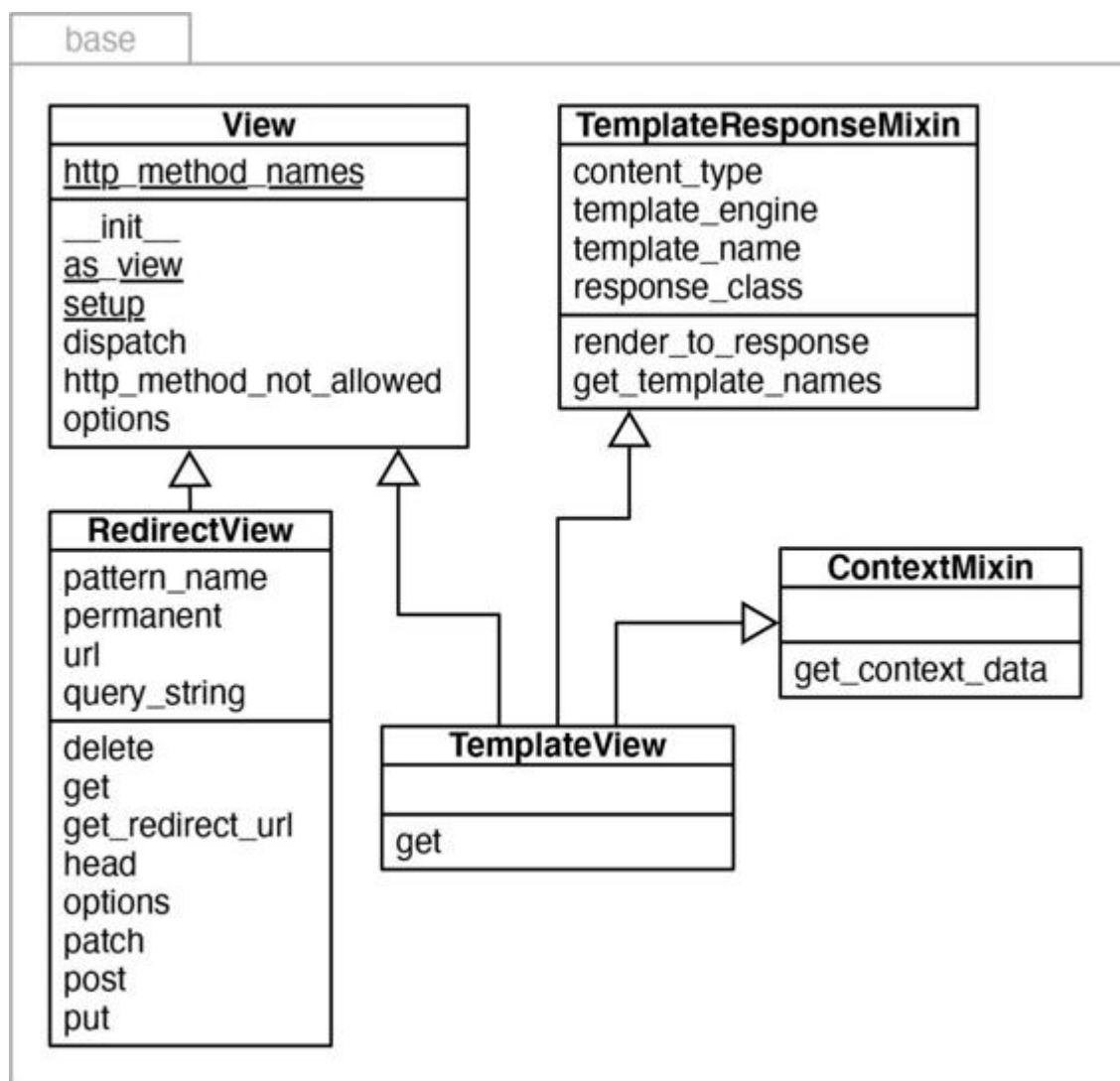


图 14-3: `django.views.generic.base` 模块的 UML 类图

`View` 的具体子类应当实现处理各种请求的方法，可是 `View` 接口为什么要实现那些方法呢？原因在于，子类可以根据实际情况自由选择，只实现需要支持的方法。由于 `TemplateView` 仅用于显示内容，因此只实现了 `get` 方法。如果向 `TemplateView` 发送 HTTP POST 请求，则继承的 `View.dispatch` 方法将检查该类是否没有实现 `post` 方法，并生成 HTTP 405 Method Not Allowed 响应。¹⁰

¹⁰熟悉设计模式的人会发现，Django 的分派机制是模板方法模式的动态变体。之所以说它是动态的，是因为 `View` 类不强制子类实现所有请求处理方法，而是由 `dispatch` 在运行时检查有没有处理具体请求的方法。

TemplateResponseMixin 提供的功能只适用于需要使用模板的视图。例如，RedirectView 没有主体内容，不需要模板，从而没有继承 TemplateResponseMixin。TemplateResponseMixin 为 TemplateView 和其他渲染模板的视图（例如 django.views.generic 子包中的 ListView、DetailView 等）提供行为。django.views.generic.list 模块和部分 base 模块的图解如图 14-4 所示。

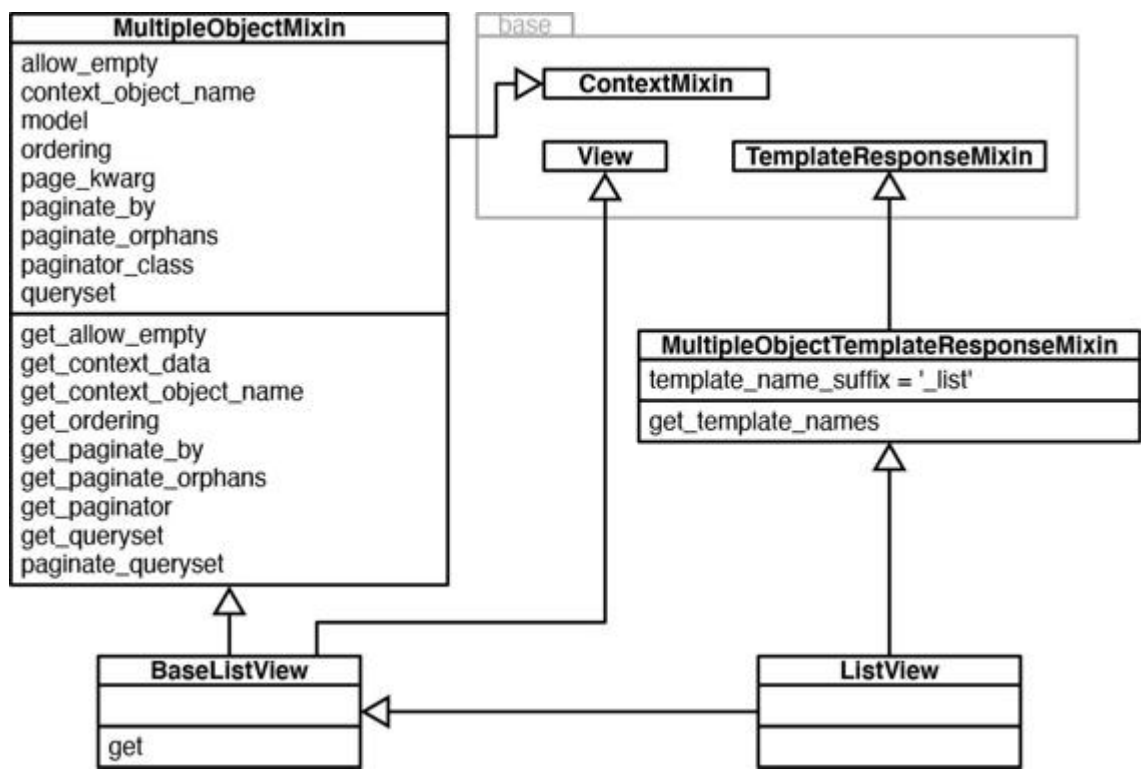


图 14-4: django.views.generic.list 模块的 UML 类图。
base 模块中的 3 个类做了简化（参见图 14-3）。ListView 是聚合类，既没有方法也没有属性

对 Django 用户来说，图 14-4 中最重要的类是 ListView。这是一个聚合类（aggregate class），没有一点代码（主体只有文档字符串）。实例化后，ListView 对象有一个 object_list 实例属性，模板通过迭代这个属性显示页面内容（通常是数据库查询返回的包含

多个对象的结果)。至于如何生成模板迭代的对象，全部交给 `MultipleObjectMixin` 来处理。这个混入类还提供了复杂的分页逻辑，即在一个页面中显示部分结果，并通过链接指向更多页面。

假如你想创建一个渲染模板的视图，但是以 JSON 格式生成一系列对象，这时就要用到 `BaseListView` 了。这个类将 `View` 和 `MultipleObjectMixin` 的功能结合在一起，剔除了烦琐的模板机制，非常方便用户扩展。

以 Django 基于类的视图 API 为例说明多重继承比 Tkinter 合适。尤其是，这样很容易看出混入类的作用：每个混入类都有明确的作用，而且名称都以 `...Mixin` 结尾。

基于类的视图还没有被 Django 用户普遍接受。许多人确实使用它们，但是作为不透明的盒子，其用途有限。当需要实现新效果时，许多 Django 程序员并未尝试重用基于类的视图和混入，而是继续编写包揽全部职责的视图函数。

学习如何利用以及如何扩展基于类的视图，满足特定的应用程序需求，确实需要一些时间，但是我认为这是值得的。基于类的视图消除了大量样板代码，使重用解决方案变得更加容易，甚至改善了团队沟通，例如，为模板以及传递给模板上下文的变量定义标准名称。基于类的视图把 Django 视图带上了“正确的轨道”。

14.6.4 Tkinter 中的多重继承

在 Python 标准库中，GUI 工具包 Tkinter 把多重继承用到了极致。图 14-2 中展示的方法解析顺序是 Tkinter 小组件层次结构的一部分，图 14-5 则列出了 `tkinter` 基包中的全部小组件类（`tkinter.ttk` 子包中还有一些）。

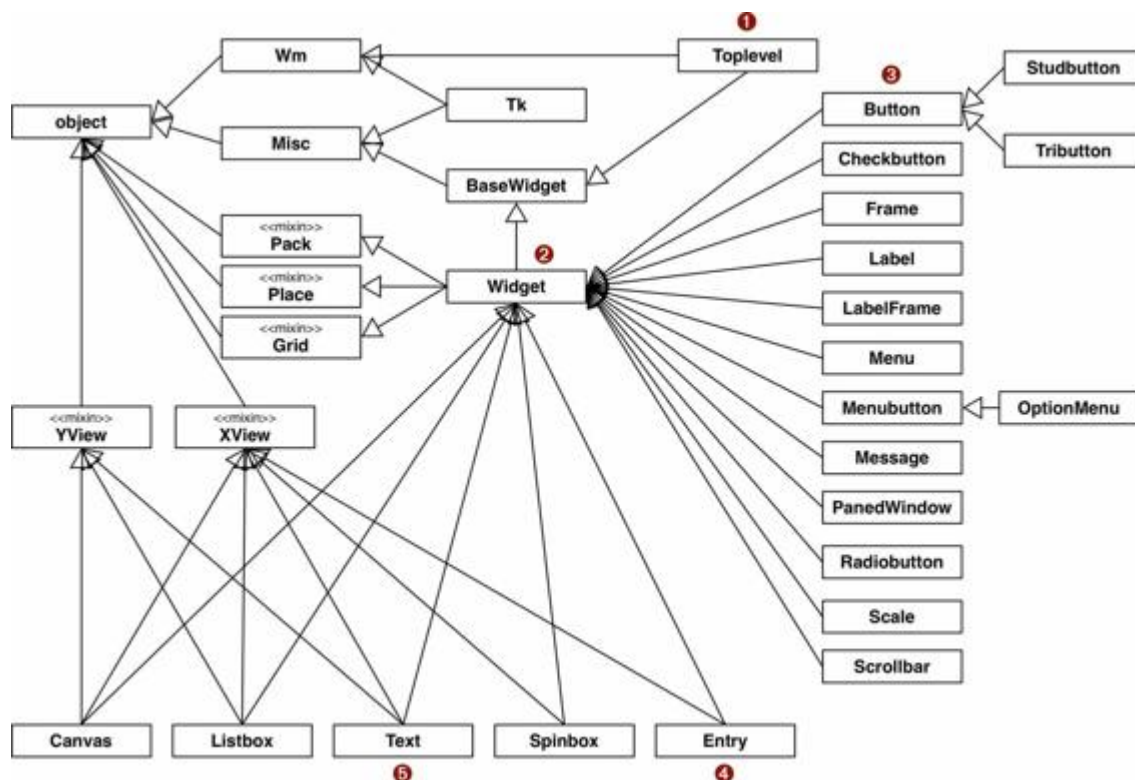


图 14-5: Tkinter GUI 类层次结构的 UML 简图；使用 «mixin» 标记的类通过多重继承为其他类提供具体方法

写作本书时，Tkinter 已经 25 岁了，因此其不能代表当下的最佳实践。但是，它能表明没有意识到多重继承的缺点时程序员是如何使用多重继承的。14.7 节在讨论一些好的做法时，会把 Tkinter 作为反面教材。

来看图 14-5 中的几个类。

- ❶ Toplevel: 表示 Tkinter 应用程序中顶层窗口的类。
- ❷ Widget: 窗口中所有可见对象的超类。
- ❸ Button: 普通的按钮小组件。
- ❹ Entry: 单行可编辑文本字段。

⑤ Text：多行可编辑文本字段。

这几个类的方法解析顺序如下所示。这些输出是使用示例 14-7 中定义的 `print_mro` 函数得到的。

```
>>> import tkinter
>>> print_mro(tkinter.Toplevel)
Toplevel, BaseWidget, Misc, Wm, object
>>> print_mro(tkinter.Widget)
Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Button)
Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Entry)
Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView,
object
```



按照现在的标准，Tkinter 的类层次结构太深了。Python 标准库中很少有超过 3 或 4 层的具体类，Java 类库也是如此。不过，应当指出，Java 类库中层次结构最深的差不多也是与 GUI 编程相关的包，例如 `java.awt` 和 `javax.swing`。Smalltalk 的现代化免费版本 Squeak 提供了富有创造性且强大的 GUI 工具包 Morphic，其类层次结构也很深。根据我的经验，GUI 工具包最能发挥继承的作用。

在类之间的关系方面要注意以下几点。

- `Toplevel` 是所有图形类中唯一没有继承 `Widget` 的，因为它是顶层窗口，行为不像小组件，例如，它不能依附到窗口或窗体上。`Toplevel` 继承自 `Wm`，后者提供直接访问宿主窗口管理器的函数，用于设置窗口标题和配置窗口边框等。
- `Widget` 直接继承自 `BaseWidget`，并继承了 `Pack`、`Place` 和 `Grid`。后 3 个类是几何管理器，负责在窗口或窗体中排布小组件。各个类封装了不同的布局策略和小组件位置 API。
- `Button` 与大多数小组件一样，只是 `Widget` 的子代，也间接继承 `Misc`，后者为各个小组件提供了大量方法。

- Entry 是 Widget 和 XView 的子类，后者支持横向滚动。
- Text 是 Widget、XView 和 YView 的子类，后者提供纵向滚动功能。

下面将讨论多重继承一些好的做法，看看 Tkinter 有没有践行。

14.7 应对多重继承

本章开篇引用 Alan Kay 的那段话现在依然正确：程序员在实践中没有通用的继承理论。我们只能从经验法则、设计模式、“最佳实践”、巧妙的缩写词、禁忌等之中寻找一些方向，却没有一应俱全的准则。

使用继承（即使避开多重继承）容易得出令人费解和脆弱的设计。我们还没有完整的理论，下面是避免把类图搅乱的一些建议。

14.7.1 优先使用对象组合，而不是类继承

本节的标题是《设计模式》一书给出的第二个面向对象设计原则，这也是我能提供的最好的建议。一旦适应了继承，很容易过度使用。按整齐的层次结构摆放物品能唤起我们的秩序感，而程序员这样做只是为了好玩。

利于组合的设计更灵活。例如，`tkinter.Widget` 类可以不从所有几何管理器中继承方法，而是让小组件实例存有一个几何管理器的引用，通过它调用方法。毕竟，不应将小组件“看作”几何管理器，但是可以通过委托使用几何管理器提供的服务。如果以后需要增加新的几何管理器，那么现有的小组件类层次结构可以保持不变，而且无须担心名称冲突。即便是单一继承，这个原则也能提高灵活性，因为子类化是一种紧密耦合，而且“参天大继承树”更容易被风吹倒。

组合和委托可以取代混入，把行为带给不同的类，但是不能取代接口继承来定义类型的层次结构。

14.7.2 理解不同情况下使用继承的原因

使用多重继承时，一定要明确一开始为什么创建子类。主要原因如下。

- 继承接口，创建子类型，实现“是什么”关系。这种情况最好使用抽象基类。
- 继承实现，通过重用避免代码重复。这种情况可以利用混入。

其实这两条经常同时出现，不过只要可能，一定要明确意图。通过继承重用代码是实现细节，通常可以换用组合和委托。而接口继承则是框架的支柱。如果可能，接口继承只应使用抽象基类作为基类。

14.7.3 使用抽象基类显式表示接口

在现代的 Python 中，如果类的作用是定义接口，则应该显式地把它定义为抽象基类或者 `typing.Protocol` 的子类。抽象基类只能子类化 `abc.ABC` 或其他抽象基类。继承多个抽象基类不是问题。

14.7.4 通过混入明确重用代码

如果一个类的作用是提供方法实现以供多个不相关的子类重用，但不体现“是什么”关系，那么就应该把那个类明确地定义为**混入类**。从概念上讲，混入不定义新类型，只是打包方法，便于重用。混入类绝对不能实例化，而且具体类不能只继承混入类。混入类应该提供某方面的特定行为，只实现少量关系非常紧密的方法。混入类不应保持任何内部状态，即不应有实例属性。

Python 中没有表明一个类是混入类的正式方式，因此强烈建议在名称后面加上 `Mixin`。

14.7.5 为用户提供聚合类

如果一个类的结构主要继承自混入类，自身没有添加结构或行为，那么这样的类称为**聚合类**。

——Booch 等人¹¹

¹¹出自 Grady Booch 等人所著的 *Object Oriented Analysis and Design, 3rd ed.*

如果抽象基类或混入类的某种组合对客户代码非常有用，那么就提供一个类，以易于理解的方式把它们结合起来。

例如，下面是图 14-4 右下方 Django `ListView` 类的完整源码。

```
class ListView(MultipleObjectTemplateResponseMixin, BaseListView):
    """
    Render some list of objects, set by `self.model` or
    `self.queryset`.
    `self.queryset` can actually be any iterable of items, not
    just a queryset.
    """
```

`ListView` 的主体是空的，但是这个类提供了一项有用的服务：把混入类和基类结合在一起，作为整体使用。

`tkinter.Widget` 也是如此，它有 4 个基类，自身却一个方法或属性也没有，只有一个文档字符串。得益于聚合类 `Widget`，使用所需的混入类即可创建新的小组件，无须记住各自在声明中的顺序。

注意，聚合类的主体不一定为空，只是经常为空而已。

14.7.6 仅子类化为子类化设计的类

技术审校 Leonardo Rochael 在审阅本章时，建议加上以下警告栏。



子类化复杂的类并覆盖类的方法容易出错，因为超类中的方法可能在不知不觉中忽略子类覆盖的行为。应尽量避免覆盖方法，至少要抑制冲动，再容易扩展的类也不轻易子类化。如果必须子类化，则还要看原类是不是为了扩展而设计的。

这是很好的建议，但是怎么知道一个类是不是为了扩展而设计的呢？

应该先查阅文档（有时是文档字符串，甚至是代码中的注释）。例如，Python 的 `socketserver` 包是“一个针对网络服务器的框架”。这个包中的 `BaseServer` 类是为了子类化而设计的，从名称中就能看出来。而且，文档和源码中的文档字符串也都明确指出了，类中的方法应由子类覆盖。

在 Python 3.8 及以上版本中，有一种新方式可以明确表明这种设计约束，即“PEP 591— Adding a final qualifier to typing”引入的 `@final` 装饰器。这个装饰器可以应用到类或方法上，IDE 或类型检查工具见到它就知道不应该子类化类或覆盖方法。¹²

¹²PEP 591 还引入了 `Final` 注解，以把变量或属性标记为不可赋予新值或覆盖。

14.7.7 避免子类化具体类

子类化具体类比子类化抽象基类和混入类还危险，因为具体类的实例通常有内部状态，覆盖依赖内部状态的方法时很容易破坏状态。即使是调用 `super()` 的协作方法，而且在使用 `__x` 句法的私有属性中保存内部状态，覆盖方法时也有无数种方式引入 bug。

在 13.5 节的“水禽和抽象基类”附注栏中，Alex Martelli 引用了《More Effective C++：35 个改善编程与设计的有效方法（中文版）》一书中的一句话：“非尾端类都应设计为抽象类。”也就是说，Meyer 建议只有抽象类可以子类化。

如果子类化是为了重用代码，那么想要重用的代码应该放入抽象基类的混入方法中，或者放入名称可明确表明意图的混入类中。

接下来，从这些建议入手分析 Tkinter。

14.7.8 Tkinter 的好、不好以及令人厌恶的方面

前几节给出的建议，Tkinter 大都没有采用，14.7.5 节例外。但是，Tkinter 做得并不好，因为使用组合模式把几何管理器集成到 `Widget` 中或许更好，详见 14.7.1 节。

记住一点，自 1994 年发布的 Python 1.1 起，Tkinter 就在标准库中了。Tkinter 的底层是 Tcl 语言优秀的 GUI 工具包 Tk。Tcl/Tk 组合原本不是面向对象的，因此 Tk API 基本上就是一堆函数。尽管没有使用面向对象方式实现，但是这个工具包的理念极具面向对象思想。

`tkinter.Widget` 类的文档字符串开头说它是“内部类”。这或许表明，`Widget` 应该被定义为抽象基类。虽然 `Widget` 自身没有方法，但是它定义了接口。它传达的意思是：“每个 Tkinter 小组件都会提供基本的方法（`__init__`、`destroy`，以及众多 Tk API 函数），此外还会提供 3 个几何管理器中的全部方法。”你可以不同意这是定义接口的好方式（太宽泛了），但是这样确实能定义接口，`Widget` 就把接口“定义”为超类接口的联合。

封装 GUI 应用程序逻辑的 `Tk` 类继承自 `Wm` 和 `Misc`，这两个类既不是抽象类，也不是混入类（`Wm` 不算混入类，因为 `TopLevel` 的超类只有它一个）。`Misc` 类的名称本身明显是**代码异味**。`Misc` 有 100 多个方法，而且所有小组件类都会继承它。为什么每个小组件都要处理剪切板、文本选择、计时器等？我们并不能把文本粘贴到按钮上，也不能选择滚动条上的文字。应该将 `Misc` 拆分成几个专门的混入类，而且不是所有小组件都应该继承这些混入类。

说实在的，作为 Tkinter 的用户，根本无须知道或使用多重继承。那些都是隐藏起来的实现细节，你在自己的代码中只需实例化或子类化

小组件类。不过，如果想查找自己需要的方法，那么在控制台中输入 `dir(tkinter.Button)`，你会发现列出了足足 214 个属性，此时你就是多重继承的受害者。



尽管存在这些问题，但 Tkinter 稳定、灵活，而且 `tkinter.ttk` 包提供的小组件很精美，外观还可以定制。此外，有些原始的小组件（例如 `Canvas` 和 `Text`），功能异常强大。用不了几个小时，你就能把一个 `Canvas` 对象打造成简单的拖拽绘图应用程序。如果对 GUI 编程感兴趣，那么 Tkinter 和 Tcl/Tk 绝对值得一用。

对继承的剖析到此结束。

14.9 本章小结

本章首先在单一继承语境下研究了 `super()` 函数。然后讨论了子类化内置类型引起的问题：内置类型的原生方法使用 C 语言实现，不调用子类中覆盖的方法，唯有极少数例外。因此，当需要定制 `list`、`dict` 或 `str` 类型时，子类化 `UserList`、`UserDict` 或 `UserString` 更简单。这些类都在 `collections` 模块中定义，它们其实是对内置类型的包装，并把操作委托给内置类型——这是标准库中优先选择组合而不使用继承的 3 个例子。如果所需的行为与内置类型区别很大，那么更容易的做法或许是子类化 `collections.abc` 模块中相应的抽象基类，然后自己实现。

本章余下的内容着重探讨了多重继承这把双刃剑。我们说明了 `__mro__` 类属性中蕴藏的方法解析顺序，有了这一机制，继承方法的名称不会再发生冲突。我们还了解了内置函数 `super()` 在多重继承下的行为（有时令人意外）。`super()` 的行为是为了支持混入类。我们以不区分大小写的映射 `UpperCaseMixin` 为例，展开了对混入类的研究。

我们了解了 Python 的抽象基类，以及 `socketserver` 包中的线程和派生混入类是如何使用多重继承和混入方法的。Django 基于类的视图和 GUI 工具包 Tkinter 都充分利用了多重继承。虽然 Tkinter 不能代表当前的最佳实践，但是可以借此说明旧系统可能会把类层次结构搞得过度复杂。

本章最后给出了应对继承的 7 条建议，并以 Tkinter 的类层次结构为例做了补充说明。

现在的趋势是摒除继承（甚至是单一继承）。21 世纪出现的语言中，Go 是最成功的一个。Go 语言没有“类”结构，不过可以通过封装字段的结构体构建类型，而且可以为结构体依附方法。Go 语言中的接口由编译器通过结构类型（**静态鸭子类型**）检查，这与 Python 3.8 引入的协议类型非常相似。Go 语言为构建类型和组合接口提供了特殊句法，但是不支持继承（甚至接口之间也不能继承）。

因此，关于继承，最好的建议或许是：尽可能避免使用。而现实中，我们往往没有选择，因为我们使用的框架有自己的设计选择。

14.10 延伸阅读

至于可读性，适当的组合比继承要好。由于读代码的频率比写代码要高得多，因此一般情况下要避免使用子类，特别是不要混合多种类型的继承，也不要使用继承来共享代码。

——Hynek Schlawack
Subclassing in Python Redux

在本书的最终评审中，技术审校 Jürgen Gmach 向我推荐了 Hynek Schlawack 写的“Subclassing in Python Redux”一文（上述引文的出处）。Schlawack 是流行的 `attrs` 包的作者，也是异步编程框架 Twisted（由 Glyph Lefkowitz 在 2002 年启动）的核心贡献者。据 Schlawack 说，随着时间的推移，核心团队意识到他们在设计中过度

使用了子类化。他写的那篇文章很长，引用了一些重要的文章和演讲。强烈推荐阅读。

在那篇文章的结论中，Hynek Schlawack 还写道：“不要忘了，更多时候，你需要的只是一个函数而已。”我同意这个观点，正是这样，本书才在类和继承之前先深入探讨了函数。我的目标是告诉你，利用标准库中现有的类，通过函数可以完成很多工作，不要动不动就自己创建类。

Guido van Rossum 写的“Unifying types and classes in Python 2.2”一文介绍了子类化内置类型、`super` 函数，以及描述符和元类等高级功能。自此之后，这些功能基本没有大的变化。Python 2.2 是语言演化史上的一个奇迹，新增了一整套强大的功能，而且没有破坏向后兼容性。用不用这些新功能，全由你自己决定。如果想使用，那么只需显式地将 `object` 子类化（直接或间接），创建所谓的“新型类”。在 Python 3 中，所有类都是对 `object` 的子类化。

《Python Cookbook（第 3 版）中文版》中有多个经典实例展示了 `super()` 和混入类的用途。可以从 8.7 节“调用父类中的方法”开始阅读，受到启发之后再循着内部引用阅读其他内容。

Raymond Hettinger 写的“Python's `super()` considered super!”一文，从积极的角度解读了 `super` 和多重继承的运作原理。这篇文章是对 James Knight 写的“Python's Super is nifty, but you can't use it”（以前题为“Python's Super Considered Harmful”）一文的回应。Martijn Pieters 在回答“How to use `super()` with one argument?”问题时对 `super` 做了简明而深入的说明，还讲到了它与描述符（详见第 23 章）的关系。`super` 就是这样，在基础场景中用法简单，但是当涉及 Python 那些最高级的动态功能时，又变成了强大而复杂的工具——这是其他语言少有的。

尽管以上几篇文章的题目都提到了内置函数 `super`，但它不是真正的问题，况且在 Python 3 中已经没有了在 Python 2 中那么令人讨厌了。真正的问题是多重继承，它天生复杂，难以处理。Michele Simionato

不再止步于批评，他在“Setting Multiple Inheritance Straight”一文中给出了解决方案：他实现了性状，这是一种受限的混入，源自 Self 语言。Simionato 写了一系列博客文章，对 Python 的多重继承进行了探讨，包括“The wonders of cooperative inheritance, or using super in Python 3”“Mixins considered harmful”（第一部分和第二部分）和“Things to Know About Python Super”（第一部分、第二部分和第三部分）。最早的文章使用的是 Python 2 的 `super` 句法，不过依然值得一读。

我读过《面向对象分析与设计（第 3 版）》，强烈推荐给你。这是面向对象思维的通用入门书，与具体的编程语言无关。能这样不带偏见地讨论多重继承的书很少。

现在，人们更倾向于摒除继承，具体方法可以参考两份资料。一份是 Brandon Rhodes 写的 *Python Design Patterns* 中的“The Composition Over Inheritance Principle”一文。另一份是 Augie Fackler 和 Nathaniel Manista 在 PyCon 2013 上所做的演讲，题为“The End Of Object Inheritance & The Beginning Of A New Modularity”。Fackler 和 Manista 的演讲讨论了如何围绕接口和处理实现接口的函数来组织系统，避免类和继承的紧密耦合和失效模式。这让我想起了 Go 语言的很多处理方式，他们提倡在 Python 中也采用类似的做法。

杂谈

想想哪些类是真正需要的

起初，我们推动继承思想是为了让新手顺利使用只能由专家设计的框架开发作品。

——Alan Kay

“The Early History of Smalltalk”

大多数程序员编写应用程序，而不开发框架。即便是开发框架的那些人，大多数时候也是在编写应用程序。编写应用程序时，通

常无须设计类的层次结构。如果需要自己编写类，那么基本上也是子类化抽象基类或框架提供的类。作为应用程序开发人员，我们极少需要编写作为其他类的超类的类。我们自己编写的类大部分是末端类（例如，继承树的末端）。

作为应用程序开发人员，如果你发现自己在构建多层类层次结构，那么可能是发生了下述事件中的一个或多个。

- 你在重新发明轮子。去找框架或库，它们提供的组件可以在应用程序中重用。
- 你使用的框架设计不良。去寻找替代品。
- 你在过度设计。记住要遵守 **KISS 原则**。
- 你厌烦了编写应用程序，决定新造一个框架。恭喜，祝你好运！

这些事情你可能都会遇到：你厌倦了，决定重新发明轮子，自己构建设计过度和设计不良的框架，因此不得不编写一个又一个类去解决“鸡毛蒜皮”的小事。希望你能乐在其中，至少得到应有的回报。

内置类型的不当行为是 bug 还是有意为之？

内置类型 `dict`、`list` 和 `str` 是 Python 的底层基础，因此速度必须快，与这些内置类型有关的任何性能问题都会对其他所有代码产生重大影响。于是，CPython 走了捷径，故意让内置类型的方法行为不当，即不调用被子类覆盖的方法。解决这一困境的可能方式之一是，为这些类型分别提供两种实现：一种供内部使用，针对解释器优化；另一种供外部使用，便于扩展。

但是等等，我们已经拥有这些了：`UserDict`、`UserList` 和 `UserString` 虽然没有内置类型的速度快，但是易于扩展。CPython 采用的这种务实方式意味着，我们也要在自己的应用程序中使用做了优化但是难以子类化的实现。这是合理的，因为我

们每天都使用 `dict`、`list` 和 `str`，但是很少需要定制映射、列表或字符串。我们只需知道其中涉及的取舍。

其他语言对继承的支持

术语“面向对象”是 Alan Kay 发明的，而 Smalltalk 只支持单一继承，不过有些派生版以不同的方式支持多重继承，例如现代的 Squeak 和 Smalltalk 方言 Pharo 就支持性状——这是实现混入类的语言结构，而且能避免多重继承的一些问题。

C++ 是第一门实现多重继承的流行语言，但是这一功能被滥用了，因此意欲取代 C++ 的 Java 不支持多重继承（没有混入类）。不过，Java 8 引入了默认方法，这使得 Java 接口与 C++ 和 Python 用于定义接口的抽象类十分相似。Java 之后，使用最广泛的 JVM 语言要数 Scala 了，而它实现了性状。

支持性状的其他语言还有最新稳定版 PHP 和 Groovy，以及 Rust 和 Raku（以前的 Perl 6）。¹³ 因此可以说，性状是 2021 年的趋势。

Ruby 对多重继承的态度很明确：对其不支持，但是引入了混入。Ruby 类的主体中可以包含模块，这样模块中定义的方法就变成了类实现的一部分。这是“纯粹”的混入，不涉及继承，因此 Ruby 混入显然不会影响所在类的类型。这种方式凸显了混入的优点，避免了许多常见问题。

最近广受瞩目的两门语言——Go 和 Julia——对继承的支持极其有限。这两门语言也有“对象”概念，而且支持多态，但是回避“类”（`class`）这个术语。

Go 语言完全不支持继承。Julia 有类型层次结构，但是子类型不继承结构，只继承行为，而且只能为抽象类型创建子类型。此外，Julia 的方法使用多重分派，这是 9.9.3 节所述机制的高级形式。

¹³我的朋友 Leonardo Rochael（也是本书技术审校）比我解释得更好：“Perl 6 一直在刷存在感，但是迟迟不发布，正在逐渐耗尽 Perl 的发展动力。现在，Perl 作为一门独立的语言持续开发（目前的版本号是 5.34），没有受到大张旗鼓的 Perl 6 什么影响。”

第 15 章 类型提示进阶

惨痛的教训告诉我，对于小型程序，动态类型就够了，而大型程序则需要更规范的方式。如果语言能做出规范，那么当然比“放任自流”要好。

——Guido van Rossum
Monty Python 剧团的“粉丝”¹

¹出自 YouTube 中的一个视频，题为“A Language Creators' Conversation: Guido van Rossum, James Gosling, Larry Wall, and Anders Hejlsberg”。这是 2019 年 4 月 2 日一场现场直播的录像。引用的内容从 1:32:05 开始，为简洁起见，做了修改。

本章接续第 8 章，讲解 Python 的渐进式类型系统。主要涵盖以下话题。

- 重载的函数签名。
- 使用 `typing.TypedDict` 为用作记录的字典添加类型提示。
- 类型校正。
- 运行时访问类型提示。
- 泛型。
 - 声明泛化类。
 - 型变：类型不变、协变和逆变。
 - 泛化静态协议。

15.1 本章新增内容

这是本书第 2 版新增的一章。下面从重载开始讲解。

15.2 重载的签名

Python 函数可以接受不同的参数组合。这些不同的参数组合使用 `@typing.overload` 装饰器注解。如果函数的返回值类型取决于两个或以上参数的类型，那么这个功能就十分必要。

以内置函数 `sum` 为例。`help(sum)` 输出的内容如下所示。

```
>>> help(sum)
sum(iterable, /, start=0)
    Return the sum of a 'start' value (default: 0) plus an
    iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric
    values and may
    reject non-numeric types.
```

内置函数 `sum` 是用 C 语言编写的，`typeshed` 项目在 `builtins.pyi` 文件中为其提供了重载的类型提示。

```
@overload
def sum(__iterable: Iterable[_T]) -> Union[_T, int]: ...
@overload
def sum(__iterable: Iterable[_T], start: _S) -> Union[_T, _S]: ...
```

先来看一下重载的整体句法。在存根文件（.pyi）中，关于 `sum` 的代码只有这些。具体实现可能在另一个文件中。省略号（...）没有特殊作用，只是为了满足句法的要求，为函数提供主体，类似于 `pass`。因此，.pyi 文件是有效的 Python 文件。

8.6 节讲过，`__iterable` 中的两个前导下划线是 PEP 484 制定的约定，表示仅限位置参数，供 Mypy 检查。也就是说，可以调用 `sum(my_list)`，但是不能调用 `sum(__iterable = my_list)`。

类型检查工具根据参数按顺序匹配各个重载的签名。

`sum(range(100), 1000)` 调用不匹配第一个重载的签名，因为该签名只有一个参数，但是它匹配第二个签名。

在常规的 Python 模块中也可以使用 `@overload`，重载的签名放在函数具体的签名和实现前面。在一个 Python 模块中，`sum` 函数的注解和实现方式如示例 15-1 所示。

示例 15-1 mysum.py：定义带有重载的签名的 `sum` 函数

```
import functools
import operator
from collections.abc import Iterable
from typing import overload, Union, TypeVar

T = TypeVar('T')
S = TypeVar('S') ❶

@overload
def sum(it: Iterable[T]) -> Union[T, int]: ... ❷
@overload
def sum(it: Iterable[T], /, start: S) -> Union[T, S]: ... ❸
def sum(it, /, start=0): ❹
    return functools.reduce(operator.add, it, start)
```

❶ 第二个重载的签名需要这个类型变量。

❷ 这个签名针对简单的情况，即 `sum(my_iterable)`。结果的类型既可能是 `T`，与 `my_iterable` 产出的元素类型相同，也可能是 `int`，此时可迭代对象为空，因为 `start` 参数的默认值为 0。

❸ 提供的 `start` 参数可以是任何类型 `S`，因此结果的类型是 `Union[T, S]`。这就是需要 `S` 的原因。如果继续使用 `T`，那么 `start` 的类型就要与 `Iterable[T]` 的元素类型相同。

❹ 函数具体实现中的签名没有类型提示。

只有一行代码的函数却用了那么多行注解，我知道这样做有点儿“劳民伤财”。但是，至少这不是随便举的一个例子。

如果想通过阅读代码来学习 `@overload`，那么可以参考 `typedsh` 项目，该项目中有数百个示例。写作本书时，`typedsh` 项目的存根文件

为 Python 内置函数提供了 186 个重载的签名，比标准库还多。



充分利用渐进式类型

追求注解全覆盖可能导致代码充斥太多噪声，有价值的信息不多。为了简化类型提示而重构也会导致 API 烦琐难用。有时，我们应该务实一点儿，部分代码没有类型提示也没关系。

符合 Python 风格的 API 往往难以注解。15.2.1 节将给出一个示例，你会发现，为了注解灵活的内置函数 `max`，需要 6 个重载的签名。

15.2.1 重载 `max` 函数

利用 Python 强大动态功能的函数往往难以添加类型提示。

在研究 `typedsh` 项目的过程中，我发现了 4051 号 bug 报告。这个报告说，内置函数 `max()` 的参数不能是 `None`，但是当传入 `None` 或者可能产出 `None` 的可迭代对象时，`Mypy` 并未发出警告。在这两种情况下，都会得到像下面这样的运行时异常。

```
TypeError: '>' not supported between instances of 'int' and 'NoneType'
```

`max` 函数的文档以下面这句话开头：

返回一个可迭代对象中最大的项，或者两个或以上参数中最大的那一个。

在我看来，这句话再明白不过了。

但是，如果按照这样的说法注解函数，那我不禁有如下疑问：注解什么？是一个可迭代对象还是两个或以上参数？

现实情况更加复杂，因为 `max` 函数还接受两个可选的关键字参数，即 `key` 和 `default`。

为了让你看清 `max` 函数的工作机制与重载的注解之间的关系，我用 Python 重新实现了 `max` 函数（内置函数 `max` 是用 C 语言实现的），如示例 15-2 所示。

示例 15-2 mymax.py: 使用 Python 重写 `max` 函数

```
# 省略导入语句和变量定义，详见示例15-3

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

# 省略重载的类型提示，详见示例15-3

def max(first, *args, key=None, default=MISSING):
    if args:
        series = args
        candidate = first
    else:
        series = iter(first)
        try:
            candidate = next(series)
        except StopIteration:
            if default is not MISSING:
                return default
            raise ValueError(EMPTY_MSG) from None
    if key is None:
        for current in series:
            if candidate < current:
                candidate = current
    else:
        candidate_key = key(candidate)
        for current in series:
            current_key = key(current)
            if candidate_key < current_key:
                candidate = current
                candidate_key = current_key
    return candidate
```

这个示例的关注点不是 `max` 函数的逻辑，因此我不会过多解释具体实现。这里的重点是 `MISSING` 常量。该常量的值是一个独特的

object 实例，用作哨符。MISSING 是 default= 关键字参数的默认值，让 max 函数接受 default=None，而且区分以下两种情况。

01. 用户没有为 default= 提供值，也就是缺失该参数。此时，如果 first 是一个空可迭代对象，那么 max 函数就会抛出 ValueError。
02. 用户为 default= 提供了值，包括 None。此时，如果 first 是一个空可迭代对象，那么 max 函数就会返回提供的值。

为了修正 4051 号工单，我编写了示例 15-3 中的代码。²

²感谢 typeshed 项目的维护人员之一 Jelle Zijlstra，他教会我很多，还把我最初写的 9 个重载的签名精简到了 6 个。

示例 15-3 mymax.py：模块的上半部分，包括导入语句、变量定义和重载的签名

```
from collections.abc import Callable, Iterable
from typing import Protocol, Any, TypeVar, overload, Union

class SupportsLessThan(Protocol):
    def __lt__(self, other: Any) -> bool: ...

T = TypeVar('T')
LT = TypeVar('LT', bound=SupportsLessThan)
DT = TypeVar('DT')

MISSING = object()
EMPTY_MSG = 'max() arg is an empty sequence'

@overload
def max(__arg1: LT, __arg2: LT, *args: LT, key: None = ...) -> LT:
    ...

@overload
def max(__arg1: T, __arg2: T, *args: T, key: Callable[[T], LT]) -> T:
    ...

@overload
def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...

@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) -> T:
```

```

...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...,
        default: DT) -> Union[LT, DT]:
    ...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...

```

我使用 Python 实现的 `max` 函数与类型导入和声明的长度差不多。得益于鸭子类型，我的代码没有使用 `isinstance` 做检查，但是达到的类型检查效果与类型提示相当——当然只是运行时检查。

`@overload` 的关键优势是，可以根据参数的类型尽量准确声明返回值的类型。下面将以一个或两个为一组，深入研究 `max` 函数重载的签名。

01. 参数实现了 `SupportsLessThan`，但是没有提供 `key` 和 `default`

```

@overload
def max(__arg1: LT, __arg2: LT, *_args: LT, key: None = ...) -> LT:
    ...
# ...省略几行...
@overload
def max(__iterable: Iterable[LT], *, key: None = ...) -> LT:
    ...

```

在这两种情况下，输入的是一个单独的参数，类型为实现了 `SupportsLessThan` 协议的 `LT`，或者是一个可迭代对象，项的类型也是 `LT`。`max` 函数的返回值类型与实参或项的类型相同（详见“有界的 `TypeVar`”一节）。

下面是可匹配这两个重载签名的示例。

```

max(1, 2, -3) # 返回2
max(['Go', 'Python', 'Rust']) # 返回'Rust'

```

02. 提供了 key, 但未提供 default

```
@overload
def max(__arg1: T, __arg2: T, *_args: T, key: Callable[[T],
LT]) -> T:
    ...
# ...省略几行...
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT]) ->
T:
    ...
```

输入可以是一个个单独的值, 类型为 `T`, 或者是可迭代对象, 类型为 `Iterable[T]`, 而且 `key=` 必须是可调用对象, 接受同为 `T` 类型的参数, 返回实现了 `SupportsLessThan` 协议的值。
`max` 函数的返回值类型与实参相同。

下面是可匹配这两个重载签名的示例。

```
max(1, 2, -3, key=abs)    # 返回-3
max(['Go', 'Python', 'Rust'], key=len)    # 返回'Python'
```

03. 提供了 default, 但未提供 key

```
@overload
def max(__iterable: Iterable[LT], *, key: None = ...,
        default: DT) -> Union[LT, DT]:
    ...
```

输入值是一个可迭代对象, 项的类型为实现了 `SupportsLessThan` 协议的 `LT`。当可迭代对象为空时, `default=` 是返回值。因此, `max` 函数的返回值类型必须是 `LT` 类型和 `default` 参数类型的联合。

下面是可匹配这个重载签名的示例。

```
max([1, 2, -3], default=0) # 返回2
max([], default=None) # 返回None
```

04. 提供了 key 和 default

```
@overload
def max(__iterable: Iterable[T], *, key: Callable[[T], LT],
        default: DT) -> Union[T, DT]:
    ...
```

输入包括以下内容。

- 一个可迭代对象，项的类型为任意类型 T 。
- 一个可调用对象，该对象接受类型为 T 的参数，返回实现了 `SupportsLessThan` 协议的 LT 类型值。
- 一个默认值，类型为任意类型 DT 。

`max` 函数的返回值类型必须是 T 类型和 `default` 参数类型的联合。

```
max([1, 2, -3], key=abs, default=None) # 返回-3
max([], key=abs, default=None) # 返回None
```

15.2.2 重载 `max` 函数的启示

有了类型提示，对于 `max([None, None])` 之类的调用，Mypy 将输出以下错误消息。

```
mymax_demo.py:109: error: Value of type variable "_LT" of "max"
cannot be "None"
```


另外，为了给类型检查工具提供支持，要编写这么多行注解可能会让人打消念头，不愿编写 `max` 这样简便灵活的函数。如果还要重新实现 `min` 函数，我肯定会重用 `max` 函数的大多数实现，在此基础上适当改动。`min` 函数重载的签名基本没有变化，唯有函数名称变了，我要复制并粘贴所有重载的签名。

我的朋友 João S. O. Bueno（我认识的最聪明的 Python 开发人员之一）在一篇推文中说道：

`max` 函数的签名虽然难以表达，但是也没有超出人类的理解能力，在我看来，注解标记的表现能力十分有限，跟 Python 没法比。

下面开始研究类型结构 `TypedDict`。这个结构没有我最初认为的那么有用，但也不是一无是处。通过 `TypedDict` 可以看出静态类型在处理动态结构（例如 JSON 数据）上的局限性。

15.3 TypedDict



处理动态数据结构（例如 JSON API 的响应）时容易误用 `TypedDict` 来避免错误。通过本节的示例，你会发现，必须在运行时才能正确处理 JSON，不能依靠静态类型检查。在运行时使用类型提示检查 JSON 等结构时，可以借助 PyPI 中的 `pydantic` 包。

Python 字典有时被当作记录使用，以键表示字段名称，字段的值可以是不同的类型。

例如，下面是使用 JSON 或 Python 描述一本书的记录。

```
{ "isbn": "0134757599",  
  "title": "Refactoring, 2e",  
  "authors": ["Martin Fowler", "Kent Beck"],  
  "pagecount": 478 }
```

在 Python 3.8 之前，没有什么好方法可以注解这样的记录，因为映射类型中的所有值必须是同种类型（详见 8.5.6 节）。

对于上述 JSON 对象，下面两个注解都不完美。

```
Dict[str, Any]
```

值可以是任何类型。

```
Dict[str, Union[str, int, List[str]]]
```

难以理解，而且没有体现字段名称与对应的字段类型之间的关系：`title` 的值应为一个 `str`，不能是 `int` 或 `List[str]`。

“PEP 589—TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys”解决了这个问题。示例 15-4 是一个简单的 `TypedDict`。

示例 15-4 books.py: 定义 `BookDict`

```
from typing import TypedDict

class BookDict(TypedDict):
    isbn: str
    title: str
    authors: list[str]
    pagecount: int
```

乍一看，`typing.TypedDict` 好像是一个数据类构建器，类似于第 5 章讲到的 `typing.NamedTuple`。

这是句法类似引起的误会。`TypedDict` 与数据类构建器千差万别。`TypedDict` 仅为类型检查工具而生，在运行时没有作用。

`TypedDict` 有以下两个作用。

- 使用与类相似的句法注解字典，为各个“字段”的值提供类型提示。

- 通过一个构造函数告诉类型检查工具，字典应具有指定的键和指定类型的值。

在运行时，TypedDict 构造函数（例如 BookDict）相当于一种安慰剂，其实作用与使用同样的参数调用 dict 构造函数相同。

BookDict 创建的是普通字典，这也就意味着：

- 伪类声明中的“字段”不创建实例属性；
- 不能通过初始化方法为“字段”指定默认值；
- 不允许定义方法。

下面来看 BookDict 在运行时的行为，如示例 15-5 所示。

示例 15-5 使用 BookDict，但是用法不太恰当

```
>>> from books import BookDict
>>> pp = BookDict(title='Programming Pearls', ❶
...               authors='Jon Bentley', ❷
...               isbn='0201657880',
...               pagecount=256)
>>> pp ❸
{'title': 'Programming Pearls', 'authors': 'Jon Bentley', 'isbn':
'0201657880',
 'pagecount': 256}
>>> type(pp)
<class 'dict'>
>>> pp.title ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'title'
>>> pp['title']
'Programming Pearls'
>>> BookDict.__annotations__ ❺
{'isbn': <class 'str'>, 'title': <class 'str'>, 'authors':
typing.List[str],
 'pagecount': <class 'int'>}
```

❶ 可以像 dict 构造函数那样调用 BookDict，传入关键字参数；也可以传入一个包含字典字面量的字典参数。

❷ 糟糕，我忘了 `authors` 的值应该是一个列表。不过，渐进式类型意味着在运行时不检查类型。

❸ 调用 `BookDict` 得到的结果是一个普通的字典.....

❹ 因此，不能使用 `object.field` 表示法读取数据。

❺ 类型提示在 `BookDict.__annotations__` 中，不在 `pp` 中。

没有类型检查工具，`TypedDict` 充其量算是注释，可以为阅读代码的人提供些许帮助，仅此而已。相比之下，第 5 章探讨的类构建器即使不使用类型检查工具，也是十分有用的，因为类构建器可在运行时生成或增强自定义的类，而且可以实例化。另外，类构建器还提供了多个有用的方法或函数，详见表 5-1。

示例 15-6 构建了一个有效的 `BookDict`，然后尝试对它执行了一些操作。通过这个示例可以看出，`TypedDict` 能协助 `Mypy` 捕获错误（参见示例 15-7）。

示例 15-6 `demo_books.py`：对 `BookDict` 执行有效操作和无效操作

```
from books import BookDict
from typing import TYPE_CHECKING

def demo() -> None: ❶
    book = BookDict( ❷
        isbn='0134757599',
        title='Refactoring, 2e',
        authors=['Martin Fowler', 'Kent Beck'],
        pagecount=478
    )
    authors = book['authors'] ❸
    if TYPE_CHECKING: ❹
        reveal_type(authors) ❺
    authors = 'Bob' ❻
    book['weight'] = 4.2
    del book['title']
```

```
if __name__ == '__main__':  
    demo()
```

- ❶ 别忘了添加返回值类型，以防 Mypy 忽略这个函数。
- ❷ 这是一个有效的 BookDict：所有键都有，而且值是正确的类型。
- ❸ Mypy 根据 BookDict 中 'authors' 键的注解推导 authors 的类型。
- ❹ 只有对这个程序做类型检查时，`typing.TYPE_CHECKING` 的值才为 `True`。在运行时，始终为假值。
- ❺ 前面的 `if` 语句能防止在运行时调用 `reveal_type(authors)`。`reveal_type` 不是 Python 运行时函数，而是 Mypy 提供的调试设施，因此无须使用 `import` 导入。输出参见示例 15-7。
- ❻ `demo` 函数的最后 3 行是无效操作，导致的错误消息参见示例 15-7。

对示例 15-6 中的 `demo_books.py` 做类型检查，结果如示例 15-7 所示。

示例 15-7 对 `demo_books.py` 做类型检查

```
.../typeddict/ $ mypy demo_books.py  
demo_books.py:13: note: Revealed type is 'built-ins.list[built-ins.str]' ❶  
demo_books.py:14: error: Incompatible types in assignment  
      (expression has type "str", variable has type  
      "List[str]") ❷  
demo_books.py:15: error: TypedDict "BookDict" has no key 'weight'  
❸  
demo_books.py:16: error: Key 'title' of TypedDict "BookDict"  
cannot be deleted ❹  
Found 3 errors in 1 file (checked 1 source file)
```

❶ 这是 `reveal_type(authors)` 输出的说明。

❷ `authors` 变量的类型通过初始化该变量的 `book['authors']` 表达式的类型推导而来。不能为 `List[str]` 类型的变量赋予一个 `str` 值。类型检查工具通常不允许变量的类型发生变化。³

³截至 2020 年 5 月，`pytype` 允许这样做。不过，`pytype` 的 FAQ 页面提到，未来将禁止这种行为。详见该页面的“Why didn't `pytype` catch that I changed the type of an annotated variable?”问题。

❸ 不能为 `BookDict` 定义中没有的键赋值。

❹ 不能删除 `BookDict` 定义中出现的键。

下面在函数签名中使用 `BookDict`，对函数调用做类型检查。

假设需要根据图书记录生成如下 XML。

```
<BOOK>
  <ISBN>0134757599</ISBN>
  <TITLE>Refactoring, 2e</TITLE>
  <AUTHOR>Martin Fowler</AUTHOR>
  <AUTHOR>Kent Beck</AUTHOR>
  <PAGECOUNT>478</PAGECOUNT>
</BOOK>
```

如果你是使用 `MicroPython` 编写代码，嵌入一个小型微控制器，则可能会编写类似示例 15-8 中的函数。⁴

⁴我喜欢使用 `lxml` 包生成并解析 XML。这个包上手简单，功能全面，而且速度快。可惜，`lxml` 包和 Python 标准库中的 `ElementTree` 超出了这个假想的微控制器的 RAM 限制。

示例 15-8 books.py: `to_xml` 函数

```
AUTHOR_ELEMENT = '<AUTHOR>{}</AUTHOR>'
```

```
def to_xml(book: BookDict) -> str: ❶
    elements: list[str] = [] ❷
    for key, value in book.items():
```

```

        if isinstance(value, list): ❸
            elements.extend(
                AUTHOR_ELEMENT.format(n) for n in value) ❹
        else:
            tag = key.upper()
            elements.append(f'<{tag}>{value}</{tag}>')
    xml = '\n\t'.join(elements)
    return f'<BOOK>\n\t{xml}\n</BOOK>'

```

❶ 这个示例的目的是在函数签名中使用 `BookDict`。

❷ 一开始为空的容器通常需要注解，否则 `Mypy` 无法推导出元素的类型。⁵

⁵`Mypy` 文档对此做了讨论，在“Common issues and solutions”页面中的“Types of empty collections”一节。

❸ `Mypy` 能理解 `isinstance` 所做的检查，并会在这个代码块中把 `value` 看作一个 `list`。

❹ 最初，守护这个代码块的 `if` 条件是 `key == 'authors'`，但是 `Mypy` 报错：“object” has no attribute “__iter__”。这是因为，经 `Mypy` 推导，`book.items()` 返回的 `value` 的类型为 `object`，而 `object` 不支持这个生成器表达式所需的 `__iter__` 方法。后来换成 `isinstance` 就好了，因为 `Mypy` 知道，在这个代码块中，`value` 的值是一个 `list`。

示例 15-9 定义了一个解析 JSON 字符串并返回 `BookDict` 的函数。

示例 15-9 books_any.py: `from_json` 函数

```

def from_json(data: str) -> BookDict:
    whatever = json.loads(data) ❶
    return whatever ❷

```

❶ `json.loads()` 的返回值类型为 `Any`。⁶

⁶Brett Cannon 和 Guido van Rossum 等人自 2016 年就开始讨论如何为 `json.loads()` 添加类型提示。详见 Mypy 182 号工单，即“Define a JSON type”。

❷ 可以返回 `whatever` (`Any` 类型)，因为 `Any` 与任何类型都兼容，包括声明的返回值类型 `BookDict`。

要特别注意示例 15-9 的第 2 个标号处。Mypy 不会标记那行代码有问题，但是在运行时，`whatever` 的值或许不满足 `BookDict` 结构——甚至可能根本就不是字典。

运行 Mypy 时，如果指定了 `--disallow-any-expr`，则 `from_json` 函数主体中的两行都会报错。

```
.../typeddict/ $ mypy books_any.py --disallow-any-expr
books_any.py:30: error: Expression has type "Any"
books_any.py:31: error: Expression has type "Any"
Found 2 errors in 1 file (checked 1 source file)
```

上述代码片段中的第 30 行和第 31 行就是 `from_json` 函数的主体。为了静默这两个类型错误，可以在 `whatever` 变量的初始化语句中添加一个类型提示，如示例 15-10 所示。

示例 15-10 books.py: 带有变量注解的 `from_json` 函数

```
def from_json(data: str) -> BookDict:
    whatever: BookDict = json.loads(data) ❶
    return whatever ❷
```

❶ 把类型为 `Any` 的表达式赋值给带有类型提示的变量，即使加上 `--disallow-any-expr`，也不报错。

❷ 现在，`whatever` 的类型是 `BookDict`，即声明的返回值类型。



不要被示例 15-10 中错误的类型安全意识迷惑了！仔细分析代码可知，类型检查工具无法预测 `json.loads()` 一定会返

回类似 `BookDict` 结构的数据。只有运行时验证能确保这一点。

静态类型检查无法避免本身就具有不确定性的代码出现错误。

`json.loads()` 就是这样的代码，它在运行时可能会构建不同类型的 Python 对象，如示例 15-11、示例 15-12 和示例 15-13 所示。

示例 15-11 `demo_not_book.py`: `from_json` 返回一个无效的 `BookDict`，但是对 `to_xml` 来说是有效的

```
from books import to_xml, from_json
from typing import TYPE_CHECKING

def demo() -> None:
    NOT_BOOK_JSON = """
        {"title": "Andromeda Strain",
         "flavor": "pistachio",
         "authors": true}
        """
    not_book = from_json(NOT_BOOK_JSON) ❶
    if TYPE_CHECKING: ❷
        reveal_type(not_book)
        reveal_type(not_book['authors'])

    print(not_book) ❸
    print(not_book['flavor']) ❹

    xml = to_xml(not_book) ❺
    print(xml) ❻

if __name__ == '__main__':
    demo()
```

❶ 这一行得不到有效的 `BookDict`，参见 `NOT_BOOK_JSON` 的内容。

❷ 让 `Mypy` 揭示几个类型。

❸ 没问题，`print` 能处理 `object` 等各种类型。

❹ `BookDict` 没有 `'flavor'` 键，但是 `JSON` 源中有。结果如何？

⑤ 记住，签名是 `def to_xml(book: BookDict) -> str:`。

⑥ 输出的 XML 是什么样的呢？

下面使用 Mypy 检查 `demo_not_book.py`，如示例 15-12 所示。

示例 15-12 使用 Mypy 检查 `demo_not_book.py` 得到的报告 (为了方便阅读，调整了格式)

```
.../typeddict/ $ mypy demo_not_book.py
demo_not_book.py:12: note: Revealed type is
    'TypedDict('books.BookDict', {'isbn': built-ins.str,
                                   'title': built-ins.str,
                                   'authors': built-ins.list[built-
ins.str],
                                   'pagecount': built-ins.int})' ❶
demo_not_book.py:13: note: Revealed type is 'built-ins.list[built-
ins.str]' ❷
demo_not_book.py:16: error: TypedDict "BookDict" has no key
'flavor' ❸
Found 1 error in 1 file (checked 1 source file)
```

❶ 揭示的类型是名义类型，不是 `not_book` 在运行时的内容。

❷ 同样，这是 `not_book['authors']` 的名义类型，同 `BookDict` 中定义的类型。不是运行时类型。

❸ 这是 `print(not_book['flavor'])` 那一行导致的错误，因为名义类型中没有 `'flavor'` 键。

现在，运行 `demo_not_book.py`，得到的输出如示例 15-13 所示。

示例 15-13 运行 `demo_not_book.py` 得到的输出

```
.../typeddict/ $ python3 demo_not_book.py
{'title': 'Andromeda Strain', 'flavor': 'pistachio', 'authors':
True} ❶
pistachio ❷
<BOOK> ❸
    <TITLE>Andromeda Strain</TITLE>
```

```
<FLAVOR>pistachio</FLAVOR>
<AUTHORS>True</AUTHORS>
</BOOK>
```

❶ 这其实不是 `BookDict` 结构。

❷ `not_book['flavor']` 的值。

❸ `to_xml` 函数的参数是一个 `BookDict`，但是没有相关的运行时检查，也就无法做出限制。

通过示例 15-13 可知，`demo_not_book.py` 的输出没有实际意义，但是运行时并未报错。使用 `TypedDict` 处理 JSON 数据没有得到多少类型安全保障。

站在鸭子类型的角度审视示例 15-8 中的 `to_xml` 函数，参数 `book` 必须提供 `.items()` 方法，返回一个产出二元组 `((key, value))` 的可迭代对象，其中：

- `key` 必须有 `.upper()` 方法；
- `value` 可以为任何值。

本节的演示表明，处理具有动态结构的数据（例如 JSON 或 XML）时，`TypedDict` 根本不能取代运行时数据验证。真想做数据验证，可以使用 `pydantic`。

`TypedDict` 还有其他功能，包括支持可选键、有限形式的继承，以及另一种声明句法。如果想进一步了解，请阅读“PEP 589—`TypedDict`: Type Hints for Dictionaries with a Fixed Set of Keys”。

现在调转话题，介绍一个最好避免，但有时又不可避免的函数，即 `typing.cast`。

15.4 类型校正

任何类型系统都不完美，静态类型检查工具、typeshed 项目中的类型提示，以及第三方包中的类型提示也是如此。

`typing.cast()` 是一个特殊函数，可用于处理不受控制的代码中存在的类型检查问题或不正确的类型提示。Mypy 0.930 文档指出：

类型校正用于消除类型检查工具发出的虚假警告，在类型检查工具无法完全理解事态时提供些许帮助。

在运行时，`typing.cast` 什么也不做。该函数的实现如下所示。

```
def cast(typ, val):
    """Cast a value to a type.
    This returns the value unchanged. To the type checker this
    signals that the return value has the designated type, but at
    runtime we intentionally don't check anything (we want this
    to be as fast as possible).
    """
    return val
```

PEP 484 要求类型检查工具务必“听信”`cast` 中所述的类型。PEP 484 中的“Casts”一节给出了一个需要通过 `cast` 为类型检查工具提供指引的示例。

```
from typing import cast

def find_first_str(a: list[object]) -> str:
    index = next(i for i, x in enumerate(a) if isinstance(x, str))
    # 至少有一个字符串才能执行到这里
    return cast(str, a[index])
```

在生成器表达式上调用 `next()` 函数时，要么返回一个字符串项，要么抛出 `StopIteration`。因此，没有异常抛出时，`find_first_str` 始终返回一个字符串，而且 `str` 是声明的返回值类型。

然而，如果最后一行只有 `return a[index]`，那么 Mypy 推导出的返回值类型将是 `object`，因为 `a` 参数声明的类型是

`list[object]`。所以，必须通过 `cast()` 指引 Mypy。⁷

⁷这个示例故意使用 `enumerate` 迷惑类型检查工具。如果直接产出字符串，不遍历 `enumerate` 索引，那么 Mypy 便能正确分析，也就不需要 `cast()` 指引。

下面再举一个例子，使用 `cast` 函数纠正 Python 标准库中一个过时的类型提示。在示例 21-12 中，我使用 `asyncio` 包创建了一个 `Server` 对象，然后想获取服务器监听的地址。起初，我编写了下面这行代码。

```
addr = server.sockets[0].getsockname()
```

但是，Mypy 报告如下错误。

```
Value of type "Optional[List[socket]]" is not indexable
```

2021 年 5 月，`typeshed` 项目中 `Server.sockets` 的类型提示对 Python 3.6 是有效的，`sockets` 属性可以是 `None`。但是在 Python 3.7 中，`sockets` 变成了一个特性，读值方法始终返回一个 `list`（服务器没有套接字时为 `None`）。而且，从 Python 3.8 开始，那个读值方法可以返回一个 `tuple`（作为不可变序列使用）。

由于不能立即修正 `typeshed` 项目，⁸ 因此我添加了像下面这样的 `cast` 调用。

⁸我向 `typeshed` 报告了这个问题，并发布了 5535 号工单，即“Wrong type hint for `asyncio.base_events.Server.sockets` attribute”。该问题很快就被 Sebastian Rittau 修正了。不过，我还是决定保留这个示例，因为通过该示例可以说明 `cast` 函数的一个常见用途，而且我编写的 `cast` 调用在修正后也没有危害。

```
from asyncio.trsock import TransportSocket
from typing import cast

# ...省略很多行...

socket_list = cast(tuple[TransportSocket, ...],
```

```
server.sockets)
    addr = socket_list[0].getsockname()
```

为了编写这个 `cast` 调用，我花了好几个小时，因为不光要弄清问题，还要阅读 `asyncio` 包的源码，找出套接字的正确类型：文档中没有体现的 `asyncio.trsock` 模块中的 `TransportSocket` 类。为了提升代码可读性，我还添加了两个 `import` 语句和一行代码。⁹ 现在，代码更安全了。

⁹老实说，我最初是在 `server.sockets[0]` 那行后面加上了 `# type: ignore` 注释。因为我发现，`asyncio` 包的文档和一个测试用例也是这样做的。所以，我怀疑，问题并不是由我的代码引起的。

细心的读者可能注意到了，如果 `sockets` 为空，那么 `sockets[0]` 将抛出 `IndexError`。然而，以我对 `asyncio` 包的理解，这在示例 21-12 中不可能发生，因为在我读取 `sockets` 属性时，`server` 已经做好接受连接的准备，不可能为空。况且，`IndexError` 是运行时错误，即便是 `print([][0])` 这么明显的问题，`Mypy` 也发现不了。



不要过于依赖使用 `cast` 静默 `Mypy` 报错，`Mypy` 报错肯定是有原因的。经常使用 `cast` 也是一种“代码异味”，可能意味着你的团队使用的类型提示有误，或者你的基准代码使用了低质量的依赖。

尽管有缺点，但是 `cast` 也有合理的用途。Guido van Rossum 就说过：

偶尔调用 `cast()`，偶尔编写 `# type: ignore` 注释，何错之有？¹⁰

¹⁰2020 年 5 月 19 日发布到 `typing-sig` 邮件列表中的消息。

彻底禁止使用 `cast` 显然是不明智的，主要原因是其他变通方法更糟。

- `# type: ignore` 提供的信息量更少。¹¹
- 使用 `Any` 有连锁反应。`Any` 与所有类型**相容**，一旦滥用，类型推导可能导致级联效应，破坏类型检查工具对代码中其他部分的检错能力。

¹¹使用 `# type: ignore[code]` 句法可以指明静默的是 Mypy 的哪个错误代码，但是错误代码并不总是容易理解。详见 Mypy 文档中的“Error codes”页面。

当然，不是所有类型问题都能使用 `cast` 纠正。有时需要使用 `# type: ignore`，偶尔还要使用 `Any`，甚至可以不为函数添加类型提示。

接下来讨论在运行时使用注解。

15.5 在运行时读取类型提示

Python 会在导入时读取函数、类和模块中的类型提示，把类型提示存储在 `__annotations__` 属性中。下面以示例 15-14 中的 `clip` 函数为例进行说明。¹²

¹²这里不会详细讨论 `clip` 函数的实现，如果感到好奇，可以在 `clip_annot.py` 中阅读整个模块的代码。

示例 15-14 clipannot.py: `clip` 函数带注解的签名

```
def clip(text: str, max_len: int = 80) -> str:
```

类型提示以字典形式存储在该函数的 `__annotations__` 属性中。

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

`'return'` 键对应示例 15-14 中 `->` 符号后面的返回值类型提示。

注意，与参数的默认值一样，注解在导入时由解释器求解。因此，注解中用的值是 Python 类 `str` 和 `int`，而不是字符串 `'str'` 和 `'int'`。导入时求解注解是截至 Python 3.10 的标准行为，如果 PEP 563 或 PEP 649 变成标准行为，则情况可能有变。

15.5.1 注解在运行时的问题

类型提示使用量的增加会引起两个问题。

- 如果类型提示很多，那么导入模块使用的 CPU 和内存会更多。
- 引用尚未定义的类型需要使用字符串，而不是真正的类型。

这两个问题都有内在原因。第一个问题的原因前面已经讲过：注解在导入时由解释器求解，存储在 `__annotations__` 属性中。下面详细分析第二个问题。

鉴于“向前引用”问题（类型提示需要引用同一模块后部定义的类），有时必须以字符串形式存储注解。然而，源码中经常见到的一种现象看起来根本不像向前引用：方法返回同一类的新对象。由于类对象直到 Python 完全求解类主体之后才被定义，因此类型提示必须使用类名的字符串形式。请看下面的例子。

```
class Rectangle:
    # ...省略几行...
    def stretch(self, factor: float) -> 'Rectangle':
        return Rectangle(width=self.width * factor)
```

截至 Python 3.10，涉及向前引用的类型提示必须使用字符串，这是标准做法。静态类型检查工具从一开始就考虑到了这个问题。

但是，在运行时，当获取 `stretch` 方法的 `return` 注解时，得到的是字符串 `'Rectangle'`，而不是真正的类型（`Rectangle` 类）。因此，要设法确定得到的字符串是什么意思。

`typing` 模块中有 3 个函数和一个类被归类为内省辅助工具，其中最重要的是 `typing.get_type_hints` 函数。该函数的文档做了如下说明。

```
get_type_hints(obj, globals=None, locals=None,
               include_extras=False)
```

.....得到的结果通常与 `obj.__annotations__` 相同。不过，以字符串字面量表示的向前引用会放在 `globals` 命名空间和 `locals` 命名空间中求解。.....



从 Python 3.10 开始，应使用新增的 `inspect.get_annotations(...)` 函数代替 `typing.get_type_hints`。但是，一些读者可能还没有使用 Python 3.10，因此本节的示例仍然使用 `typing.get_type_hints`。这个函数自 Python 3.5 添加 `typing` 模块之后就可用了。

“PEP 563—Postponed Evaluation of Annotations”已经通过，无须再使用字符串编写注解了，而且类型提示的运行时开销也减少了。这个 PEP 的主要目的在“摘要”中表达得很清楚，即下面这句话。

本 PEP 提议更改函数注解和变量注解，不在定义函数时求解，在**注解**中保留字符串形式。

从 Python 3.7 开始，开头有以下 `import` 语句的模块都按上述方式处理注解。

```
from __future__ import annotations
```

为了演示具体效果，把示例 15-14 中的 `clip` 函数复制到 `clip_annot_post.py` 模块中，并在该模块顶部加上 `__future__` 导入语句。

在控制台中，导入那个模块，读取 `clip` 函数的注解，结果如下所示。

```
>>> from clip_annot_post import clip
>>> clip.__annotations__
{'text': 'str', 'max_len': 'int', 'return': 'str'}
```

可以看到，现在所有类型提示都是普通的字符串，尽管事实上在定义 `clip` 函数时并没有使用括在引号内的字符串（参见示例 15-14）。

`typing.get_type_hints` 函数可以解析很多类型提示，包括 `clip` 函数中的类型提示。

```
>>> from clip_annot_post import clip
>>> from typing import get_type_hints
>>> get_type_hints(clip)
{'text': <class 'str'>, 'max_len': <class 'int'>, 'return': <class 'str'>}
```

调用 `get_type_hints` 得到的是真正的类型，即便有时候原始类型提示是括在引号内的字符串。这是在运行时读取类型提示的推荐方式。

PEP 563 的行为计划在 Python 3.10 中变成默认行为，无须导入 `__future__`。但是，FastAPI 和 pydantic 的维护人员指出，这个变化会导致两个包中在运行时依赖类型提示的代码失效，而 `get_type_hints` 的可靠性不高。

在 python-dev 邮件列表随后的讨论中，PEP 563 的作者Łukasz Langa 总结了 `get_type_hints` 函数的一些局限性。

.....事实证明，`typing.get_type_hints()` 有一定的局限性，在运行时使用通常消耗较大，而且更重要的是，其无法解析所有类型。最常见的例证是在非全局上下文（例如内部类、函数中的类等）中生成的类型。最明显的是涉及向前引用的情况：类中的方法如果接受或返回同一类型的对象，那么在使用类生成器

时，`typing.get_type_hints()` 也不能正确处理。可以使用一些技巧进行处理，但是基本上都不优雅。¹³

¹³2021 年 4 月 16 日发布的“PEP 563 in light of PEP 649”消息。

Python 指导委员会决定推迟到 Python 3.11 或更晚的版本再把 PEP 563 定为默认行为，为开发人员留出更多时间，为 PEP 563 尝试解决的问题找出更好的方案，而不破坏运行时广泛使用的类型提示。“PEP 649—Deferred Evaluation Of Annotations Using Descriptors”是一种正在商讨的可行方案，不过最终或许会给出其他折中方案。

综上所述，截至 Python 3.10，在运行时读取类型提示不完全可靠，而这种情况有可能在 2022 年发生变化。



大规模使用 Python 的公司希望从静态类型中获益，但是在导入时求解类型代价太大，它们负担不起。静态检查在开发人员的工作站和专用的 CI 服务器中实施，而在生产容器中，加载模块的频率和数量都高得多，这种规模的成本是不可忽视的。

这在 Python 社区中造成了两类人之间的紧张关系，一类人希望类型提示仅存储为字符串（降低加载成本），另一类人则想在运行时使用类型提示，例如 `pydantic` 和 `FastAPI` 的创建者和用户，他们宁愿存储类型对象，也不自己动手求解注解，毕竟这不是一项简单的任务。

15.5.2 解决这个问题

鉴于目前不稳定的局势，如果需要在运行时读取注解，可以参考以下两点建议。

- 不要直接读取 `__annotations__` 属性，使用 `inspect.get_annotations`（自 Python 3.10 可用）或 `typing.get_type_hints`（自 Python 3.5 可用）。

- 自己编写一个函数，简单包装 `inspect.get_annotations` 或 `typing.get_type_hints`。在基准代码中调用自己编写的那个函数，这样当以后行为有变时，只需修改一个函数即可。

示例 24-5 中定义的 `Checked` 类（第 24 章会具体分析）就采用了第二点建议，下面是该类的前几行代码。

```
class Checked:
    @classmethod
    def _fields(cls) -> dict[str, type]:
        return get_type_hints(cls)
    # .....省略余下的代码.....
```

类方法 `Checked._fields` 的作用是防止模块的其他部分直接使用 `typing.get_type_hints`。如果未来 `get_type_hints` 的行为有变，需要额外的逻辑，或者你想换用 `inspect.get_annotations`，那么只需修改 `Checked._fields` 方法，程序的其他部分不受影响。



鉴于类型提示的运行时审查行为正在讨论之中，也提出了一些修改建议，请务必阅读官方文档中的“Annotations Best Practices”一文。在通往 Python 3.11 的路上，这篇文章可能会做更新。这篇实践指导由 Larry Hastings 撰写，他是“PEP 649—Deferred Evaluation Of Annotations Using Descriptors”的作者。针对“PEP 563—Postponed Evaluation of Annotations”提出的运行时问题，PEP 649 给出了另一种解决方案。

本章将在余下的内容中讨论泛化。首先讲解如何定义可由用户参数化的泛化类。

15.6 实现一个泛化类

示例 13-7 中定义的抽象基类 `Tombola` 为类似宾果机的类定义了接口，示例 13-10 中的 `LottoBlower` 类是一个具体实现。本节研究 `LottoBlower` 的一个泛化版本，用法如示例 15-15 所示。

示例 15-15 `generic_lotto_demo.py`: 使用一个泛化的彩票摇奖机类

```
from generic_lotto import LottoBlower

machine = LottoBlower[int](range(1, 11)) ❶

first = machine.pick() ❷
remain = machine.inspect() ❸
```

❶ 实例化泛化类需要提供具体的类型参数，例如这里的 `int`。

❷ Mypy 能正确推导出 `first` 是一个 `int` 值.....

❸ 以及 `remain` 是一个整数元组。

另外，如果与参数化类型相悖，则 Mypy 还会报告详细的消息，如示例 15-16 所示。

示例 15-16 `generic_lotto_errors.py`: Mypy 报告的错误

```
from generic_lotto import LottoBlower

machine = LottoBlower[int]([1, .2])
## error: List item 1 has incompatible type "float"; ❶
##         expected "int"

machine = LottoBlower[int](range(1, 11))

machine.load('ABC')
## error: Argument 1 to "load" of "LottoBlower" ❷
##         has incompatible type "str";
##         expected "Iterable[int]"
## note: Following member(s) of "str" have conflicts:
## note:         Expected:
## note:         def __iter__(self) -> Iterator[int]
```

```
## note:          Got:
## note:          def __iter__(self) -> Iterator[str]
```

❶ 实例化 `LottoBlower[int]`，Mypy 报告类型不能为 `float`。

❷ 当调用 `.load('ABC')` 时，Mypy 给出了不能使用 `str` 值的原因：`str.__iter__` 会返回 `Iterator[str]`，但是 `LottoBlower[int]` 需要的是 `Iterator[int]`。

这个泛化类的实现如示例 15-17 所示。

示例 15-17 generic_lotto.py：一个泛化的彩票摇奖机类

```
import random

from collections.abc import Iterable
from typing import TypeVar, Generic
from tombola import Tombola

T = TypeVar('T')

class LottoBlower(Tombola, Generic[T]): ❶

    def __init__(self, items: Iterable[T]) -> None: ❷
        self._balls = list[T](items)

    def load(self, items: Iterable[T]) -> None: ❸
        self._balls.extend(items)

    def pick(self) -> T: ❹
        try:
            position = random.randrange(len(self._balls))
        except ValueError:
            raise LookupError('pick from empty LottoBlower')
        return self._balls.pop(position)

    def loaded(self) -> bool: ❺
        return bool(self._balls)

    def inspect(self) -> tuple[T, ...]: ❻
        return tuple(self._balls)
```

- ❶ 泛化类声明通常使用多重继承，因为需要子类化 `Generic`，以声明形式类型参数（这里的 `T`）。
- ❷ `__init__` 方法的 `items` 参数是 `Iterable[T]` 类型。如果使用 `LottoBlower[int]` 实例化，则类型是 `Iterable[int]`。
- ❸ `load` 方法也有同样的约束。
- ❹ 对于 `LottoBlower[int]`，返回值类型 `T` 会变成 `int`。
- ❺ 这个方法没有使用类型变量。
- ❻ 最后，使用 `T` 设置返回的元组中项的类型。



`typing` 模块文档中的“User-defined generic types”一节内容简短，含有示例，而且讲到了这里没有提及的一些细节。

现在，我们知道如何实现泛化类了。下面定义与泛化有关的术语。

泛型基本术语

下面给出几个我认为对研究泛化有用的定义。¹⁴

¹⁴这些术语出自《Effective Java 中文版（原书第 3 版）》。定义和示例是我编写的。

泛型

具有一个或多个类型变量的类型。

示例：`LottoBlower[T]` 和 `abc.Mapping[KT, VT]`。

形式类型参数

泛型声明中出现的类型变量。

示例：`abc.Mapping[KT, VT]` 中的 `KT` 和 `VT`。

参数化类型

使用具体类型参数声明的类型。

示例：`LottoBlower[int]` 和 `abc.Mapping[str, float]`。

具体类型参数

声明参数化类型时为参数提供的具体类型。

示例：`LottoBlower[int]` 中的 `int`。

下一个话题讨论如何让泛型更灵活，介绍协变、逆变和不变等概念。

15.7 型变



如果没有在其他语言中使用泛化的经验，那么本节对你来说将是本书最难理解的部分。型变概念抽象难懂，而且需要严谨的表述，导致本节就像数学书中的内容一样。

其实，真正需要关注型变的基本上是代码库作者，因为只有他们才需要支持新的泛化容器类型，或者提供基于回调的 API。不过，为了降低复杂度，可以仅支持不变容器——Python 标准库基本上就是这么做的。所以，第一遍阅读本书时，可以跳过本节，或者只读介绍不变类型那几节。

本书首次介绍**型变**的概念是在“Callable 类型的型变”一节，当时是将其应用于参数化泛化的 `Callable` 类型。本节将扩展这个概念，涵盖泛化容器类型，利用现实中的具体事物类比这个抽象的概念。

假设一所学校的食堂规定，只允许安装果汁自动售货机，¹⁵ 不允许安装一般的饮料自动售货机，以防止售卖学校董事会禁止的苏打水。¹⁶

¹⁵我第一次见到以食堂类比型变是在 Erik Meijer 为《Dart 编程语言》一书写的序中。

¹⁶比禁止卖书好多了！

15.7.1 一个不变的自动售货机

下面试着为食堂的规定建模：定义一个泛化的 `BeverageDispenser` 类，参数化饮料的类型。详见示例 15-18。

示例 15-18 `invariant.py`：类型定义和 `install` 函数

```
from typing import TypeVar, Generic

class Beverage: ❶
    """任何饮料"""

class Juice(Beverage):
    """任何果汁"""

class OrangeJuice(Juice):
    """使用巴西橙子制作的美味果汁"""

T = TypeVar('T') ❷

class BeverageDispenser(Generic[T]): ❸
    """一个参数化饮料类型的自动售货机"""
    def __init__(self, beverage: T) -> None:
        self.beverage = beverage

    def dispense(self) -> T:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None: ❹
    """安装一个果汁自动售货机"""
```

❶ `Beverage`、`Juice` 和 `OrangeJuice` 构成了一种类型层次结构。

- ❷ 简单的 `TypeVar` 声明。
- ❸ `BeverageDispenser` 参数化了饮料的类型。
- ❹ `install` 是模块全局函数。该函数的类型提示会执行只能安装果汁自动售货机的规定。

按照示例 15-18 的定义，以下代码是有效的。

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)
```

但是，以下代码无效。

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##           expected "BeverageDispenser[Juice]"
```

不接受可售卖任何饮料 (`Beverage`) 的自动售货机，因为食堂要求自动售货机只能售卖果汁 (`Juice`) 。

让人不解的是，以下代码也无效。

```
orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[OrangeJuice]"
##           expected "BeverageDispenser[Juice]"
```

使用 `OrangeJuice` 特化的自动售货机也不允许安装，只允许安装 `BeverageDispenser[Juice]`。

`BeverageDispenser[OrangeJuice]` 与 `BeverageDispenser[Juice]` 不兼容（尽管 `OrangeJuice` 是 `Juice` 的子类型），按照类型相关的术语，我们说 `BeverageDispenser(Generic[T])` 是不变的。

诸如 `list` 和 `set` 之类 Python 可变的容器类型都是不变的。示例 15-17 中的 `LottoBlower` 类也是不变的。

15.7.2 一个协变的自动售货机

如果想灵活一些，把自动售货机建模为可接受某些饮料类型及其子类型的泛化类，则必须让它支持协变。`BeverageDispenser` 类的声明如示例 15-19 所示。

示例 15-19 covariant.py: 类型定义和 `install` 函数

```
T_co = TypeVar('T_co', covariant=True) ❶

class BeverageDispenser(Generic[T_co]): ❷
    def __init__(self, beverage: T_co) -> None:
        self.beverage = beverage

    def dispense(self) -> T_co:
        return self.beverage

def install(dispenser: BeverageDispenser[Juice]) -> None: ❸
    """安装一个果汁自动售货机"""
```

❶ 声明类型变量时，设置 `covariant=True`。`_co` 后缀是 `typedsh` 项目采用的一种约定，表明这是协变的类型参数。

❷ 使用 `T_co` 参数化特殊的 `Generic` 类。

❸ `install` 函数的类型提示与示例 15-18 相同。

以下代码能正常运行，因为现在对可协变的 `BeverageDispenser` 来说，`Juice` 和 `OrangeJuice` 都是有效的自动售货机。

```
juice_dispenser = BeverageDispenser(Juice())
install(juice_dispenser)

orange_juice_dispenser = BeverageDispenser(OrangeJuice())
install(orange_juice_dispenser)
```

但是，不接受售卖任何饮料的 Beverage 自动售货机。

```
beverage_dispenser = BeverageDispenser(Beverage())
install(beverage_dispenser)
## mypy: Argument 1 to "install" has
## incompatible type "BeverageDispenser[Beverage]"
##           expected "BeverageDispenser[Juice]"
```

这就是协变，参数化自动售货机子类型关系的变化方向与类型参数子类型关系的变化方向相同。

15.7.3 一个逆变的垃圾桶

现在，对食堂配备垃圾桶的规则建模。假设食物和饮料全都使用可生物降解的材料包装，而且剩饭剩菜和一次性餐具也可生物降解。食堂内的垃圾桶必须适合存放可生物降解的废弃物。



对这个教学用的示例，需要做一些约定，以一种精简的层次结构简化垃圾分类。

- Refuse 是最一般的垃圾类型。所有垃圾都是废弃物。
- Biodegradable 是特殊的垃圾类型，随着时间的推移，可被生物降解。某些废弃物（Refuse）不是可生物降解垃圾（Biodegradable）。
- Compostable 是特殊的可生物降解垃圾（Biodegradable），可在堆肥箱或堆肥设施中转化为有机肥料。根据我们的定义，不是所有可生物降解垃圾都是可制作成肥料的垃圾（Compostable）。

为了对食堂可接受的垃圾桶规则进行建模，需要引入“逆变”的概念，如示例 15-20 所示。

示例 15-20 contravariant.py: 类型定义和 install 函数

```

from typing import TypeVar, Generic

class Refuse: ❶
    """任何废弃物"""

class Biodegradable(Refuse):
    """可生物降解的废弃物"""

class Compostable(Biodegradable):
    """可制成肥料的废弃物"""

T_contra = TypeVar('T_contra', contravariant=True) ❷

class TrashCan(Generic[T_contra]): ❸
    def put(self, refuse: T_contra) -> None:
        """在倾倒之前存放垃圾"""

def deploy(trash_can: TrashCan[Biodegradable]):
    """放置一个垃圾桶，存放可生物降解的废弃物"""

```

❶ 一种废弃物类型层次结构：Refuse 是最一般的类型，Compostable 是最具体的类型。

❷ 按约定，T_contra 表示逆变类型变量。

❸ TrashCan 对废弃物的类型实行逆变。

按照上述定义，以下类型的垃圾桶是可接受的。

```

bio_can: TrashCan[Biodegradable] = TrashCan()
deploy(bio_can)

trash_can: TrashCan[Refuse] = TrashCan()
deploy(trash_can)

```

更一般的 TrashCan[Refuse] 是可接受的，因为它可以存放任何废弃物，包括可生物降解的废弃物 (Biodegradable)。然而，TrashCan[Compostable] 不可接受，因为它不能存放可生物降解的废弃物 (Biodegradable)。

```
compost_can: TrashCan[Compostable] = TrashCan()
deploy(compost_can)
## mypy: Argument 1 to "deploy" has
## incompatible type "TrashCan[Compostable]"
##           expected "TrashCan[Biodegradable]"
```

下面总结一下相关概念。

15.7.4 型变总结

型变是一种难以描述的性质。下面我们会总结不变类型、协变类型和逆变类型等概念，并提供一些经验法则，用于推断型变种类。

01. 不变类型

不管实参之间是否存在关系，当两个参数化类型之间不存在超类型或子类型关系时，泛型 L 是不变的。也就是说，如果 L 是不变的，那么 $L[A]$ 就不是 $L[B]$ 的超类型或子类型。两个方向都是不相容的。

前文说过，Python 中的可变容器默认是不可变的。`list` 类型就是一例：`list[int]` 与 `list[float]` **不相容**，反之亦然。

一般来说，如果一个形式类型参数既出现在方法参数的类型提示中，又出现在方法的返回值类型中，那么该参数必须是不可变的，因为要确保更新容器和从容器中读取时的类型安全性。

举个例子，下面是 `typeshed` 项目中内置类型 `list` 的部分类型提示。

```
class list(MutableSequence[_T], Generic[_T]):
    @overload
    def __init__(self) -> None: ...
    @overload
    def __init__(self, iterable: Iterable[_T]) -> None: ...
    # ...省略部分行...
    def append(self, __object: _T) -> None: ...
    def extend(self, __iterable: Iterable[_T]) -> None: ...
```

```
def pop(self, __index: int = ...) -> _T: ...  
# ...
```

注意，`_T` 既出现在了 `__init__`、`append` 和 `extend` 等方法的参数中，也是 `pop` 方法的返回值类型。如果 `_T` 可以协变或逆变，则无法保障这种类型的类型安全性。

02. 协变类型

给定两个类型 `A` 和 `B`，`B` 与 `A` **相容**，而且均不是 `Any`。有些作者使用符号 `<:` 和 `:>` 表示类型之间的关系，如下所示。

`A :> B`

`A` 是 `B` 的**超类型**，或者 `A` 与 `B` 类型相同。

`B <: A`

`B` 是 `A` 的**子类型**，或者 `B` 与 `A` 类型相同。

对于 `A :> B`，当满足 `C[A] :> C[B]` 时，泛型 `C` 是可协变的。

注意，在前后两种情况中，`:>` 符号的方向是相同的，`A` 在 `B` 的左边。协变的泛型遵循具体类型参数的子类型关系。

不可变容器可以是协变的。例如，文档中使用约定的命名方式 `T_co` 指明 `typing.FrozenSet` 有一个可协变的类型变量。

```
class FrozenSet(frozenset, AbstractSet[T_co]):
```

使用 `:>` 表示参数化类型，如下所示。

```
float :> int  
frozenset[float] :> frozenset[int]
```

迭代器也可以是协变的。迭代器不是 `frozenset` 这种只读的容器，而是只产生输出。只要预期产出浮点数的 `abc.Iterator[float]`，就可以放心使用产出整数的 `abc.Iterator[int]`。基于同样的原因，`Callable` 类型的返回值类型也可以是协变的。

03. 逆变类型

对于 $A :> B$ ，当满足 $K[A] <: K[B]$ 时，泛型 K 是可逆变的。

可逆变的泛型可以逆转具体类型参数的子类型关系。

`TrashCan` 类就是一例。

```
Refuse :> Biodegradable
TrashCan[Refuse] <: TrashCan[Biodegradable]
```

可逆变的容器通常是只写的数据结构（也叫“接收器”，sink）。标准库中没有这样的容器，不过一些类型有可逆变的类型参数。

`Callable[[ParamType, ...], ReturnType]` 中的参数类型是可逆变的，不过 `ReturnType` 是可协变的（详见“Callable 类型的型变”一节）。另外，`Generator`、`Coroutine` 和 `AsyncGenerator` 都有一个可逆变的类型参数。17.13.3 节会讲解 `Generator` 类型，第 21 章会讲解 `Coroutine` 和 `AsyncGenerator`。

以上关于型变的讨论，主要是想告诉你，可逆变的形式参数可以定义用于调用或向对象发送数据的参数的类型，而可协变的形式参数可以定义对象产生的输出的类型——根据对象的不同，可以是产出值的类型或返回值的类型。17.13 节会解释“发送”和“产出”的含义。

根据“输出可协变，输入可逆变”的结论，可以得出一些有用的指导方针。

04. 型变经验法则

最后，根据以下几条经验法则，可以推知具体的型变种类。

- 如果一个形式类型参数定义的是从对象中获取的数据类型，那么该形式类型参数可能是协变的。
- 如果一个形式类型参数定义的是对象初始化之后向对象中输入的数据类型，那么该形式类型参数可能是逆变的。
- 如果一个形式类型参数定义的是从对象中获取的数据类型，同时也是向对象中输入的数据类型，那么该形式类型参数必定是不变的。
- 为保险起见，形式类型参数最好是不变的。

`Callable[[ParamType, ...], ReturnType]` 体现了第 1 条和第 2 条：`ReturnType` 是协变的，各个 `ParamType` 是逆变的。

默认情况下，`TypeVar` 创建的形式参数是不变的，标准库中的可变容器都是这样注解的。

17.13.3 节将继续讨论型变。

接下来讨论如何定义泛化静态协议，通过几个新示例讲解协变。

15.8 实现泛化静态协议

Python 3.10 标准库提供了几个泛化静态协议。`typing` 模块中的 `SupportsAbs` 就是一个，实现方式如下所示。

```

@runtime_checkable
class SupportsAbs(Protocol[T_co]):
    """An ABC with one abstract method __abs__ that is covariant
    in its
        return type."""
    __slots__ = ()

    @abstractmethod
    def __abs__(self) -> T_co:
        pass

```

T_co 按命名约定，声明如下。

```

T_co = TypeVar('T_co', covariant=True)

```

多亏了 SupportsAbs，在 Mypy 看来，示例 15-21 中的代码是有效的。

示例 15-21 abs_demo.py: 使用泛化的 SupportsAbs 协议

```

import math
from typing import NamedTuple, SupportsAbs

class Vector2d(NamedTuple):
    x: float
    y: float

    def __abs__(self) -> float: ❶
        return math.hypot(self.x, self.y)

def is_unit(v: SupportsAbs[float]) -> bool: ❷
    """v 的模接近1时为'True'"""
    return math.isclose(abs(v), 1.0) ❸

assert issubclass(Vector2d, SupportsAbs) ❹

v0 = Vector2d(0, 1) ❺
sqrt2 = math.sqrt(2)
v1 = Vector2d(sqrt2 / 2, sqrt2 / 2)
v2 = Vector2d(1, 1)
v3 = complex(.5, math.sqrt(3) / 2)
v4 = 1 ❻
assert is_unit(v0)
assert is_unit(v1)
assert not is_unit(v2)

```

```
assert is_unit(v3)
assert is_unit(v4)

print('OK')
```

- ❶ 定义 `__abs__`, 让 `Vector2d` 与 `SupportsAbs` **相容**。
- ❷ 使用 `float` 参数化 `SupportsAbs`.....
- ❸ 确保 Mypy 允许 `abs(v)` 作为 `math.isclose` 的第一个参数。
- ❹ 由于 `SupportsAbs` 的定义中有 `@runtime_checkable`, 因此这是一个有效的运行时断言。
- ❺ 以下代码和运行时断言均能通过 Mypy 检查。
- ❻ `int` 类型也与 `SupportsAbs` **相容**。根据 `typedsh` 项目, `int.__abs__` 返回一个 `int` 值, 而 `int` 与 `is_unit` 函数中 `v` 参数声明的 `float` 类型参数**相容**。

同样, 可以为示例 13-18 中的 `RandomPicker` 协议 (只有一个返回 `Any` 的 `pick` 方法) 编写一个泛化版本。

示例 15-22 实现了泛化的 `RandomPicker` 协议, `pick` 方法的返回值类型可协变。

示例 15-22 generic_randompick.py: 定义泛化的 `RandomPicker` 协议

```
from typing import Protocol, runtime_checkable, TypeVar

T_co = TypeVar('T_co', covariant=True) ❶

@runtime_checkable
class RandomPicker(Protocol[T_co]): ❷
    def pick(self) -> T_co: ... ❸
```

- ❶ 声明可协变的 `T_co`。
- ❷ 使用可协变的形式类型参数泛化 `RandomPicker`。
- ❸ 使用 `T_co` 作为返回值类型。

泛化的 `RandomPicker` 协议可以协变，因为在返回值类型中使用了唯一的形式参数。

15.9 本章小结

本章从使用 `@overload` 的简单示例开始，然后详细研究了一个更复杂的示例：使用多个重载的签名正确注解内置函数 `max`。

接着讲解了特殊结构 `typing.TypedDict`。之所以选择在本章介绍 `typing.TypedDict`，而不是在第 5 章与 `typing.NamedTuple` 一起介绍，是因为 `TypedDict` 不是类构建器。`TypedDict` 只用于为值是字典的变量或参数添加类型提示，指明各个字符串键对应的值是什么类型（把字典用作记录，通常是在处理 JSON 数据时）。那一节有点儿长，因为 `TypedDict` 可能会给人一种安全错觉，而且我想告诉你，从动态的映射中获取静态的结构化记录时，运行时检查和错误处理是不可避免的。

接下来讨论了为类型检查工具提供指引的 `typing.cast` 函数。使用 `cast` 函数之前务必三思，过度使用会妨碍类型检查工具开展工作。

之后讲解了在运行时访问类型提示。重点是使用 `typing.get_type_hints`，而不要直接读取 `__annotations__` 属性。然而，对于某些注解，该函数或许不可靠。我们了解到，Python 核心开发人员仍在研究适当的方法，让类型提示在运行时可用的同时减少对 CPU 和内存的影响。

最后几节讨论了泛化。我们首先实现了泛化的 `LottoBlower` 类（后来得知，这是一个不变的泛化类）。随后定义了 4 个基本术语：泛型、形式类型参数、参数化类型和具体类型参数。

接下来是型变相关的话题，以“现实中”的食堂饮料自动售货机和垃圾桶为例，讲解了不变类型、协变类型和逆变类型。之后总结了相关概念，给出了正式定义，还分析了 Python 标准库对这些概念的应用。

最后讲解了如何定义泛化静态协议。首先分析了标准库中的 `typing.SupportsAbs` 协议。然后“照葫芦画瓢”，定义了 `RandomPicker` 协议，该协议比第 13 章中实现的版本更加严格。



Python 的类型系统是一个庞大的话题，而且发展迅速。本章并没有涵盖所有知识。我选择关注的话题，要么广泛适用，要么特别难理解，要么是重要的概念，因此可能在很长一段时期内都有参考意义。

15.10 延伸阅读

Python 的静态类型系统在最初设计时就很复杂，而且一年比一年复杂。表 15-1 列出了截至 2021 年 5 月我知道的所有 PEP。如果要涵盖所有内容，则可能需要再写一本书。

表 15-1：涉及类型提示的 PEP。编号后带 * 的 PEP 表示十分重要，在 `typing` 模块的文档开篇有提及。“Python 版本”列中的问号表示对应的 PEP 正在商讨或者尚未实现；“n/a”表示信息性 PEP，不对应具体的 Python 版本

PEP	标题	Python 版本	年份

PEP	标题	Python 版本	年份
3107	Function Annotations	3.0	2006
483*	The Theory of Type Hints	n/a	2014
484*	Type Hints	3.5	2014
482	Literature Overview for Type Hints	n/a	2015
526*	Syntax for Variable Annotations	3.6	2016
544*	Protocols: Structural subtyping (static duck typing)	3.8	2017
557	Data Classes	3.7	2017
560	Core support for typing module and generic types	3.7	2017
561	Distributing and Packaging Type Information	3.7	2017
563	Postponed Evaluation of Annotations	3.7	2017
586*	Literal Types	3.8	2018
585	Type Hinting Generics In Standard Collections	3.9	2019
589*	TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys	3.8	2019

PEP	标题	Python 版本	年份
591*	Adding a final qualifier to typing	3.8	2019
593	Flexible function and variable annotations	?	2019
604	Allow writing union types as X Y	3.10	2019
612	Parameter Specification Variables	3.10	2019
613	Explicit Type Aliases	3.10	2020
645	Allow writing optional types as x?	?	2020
646	Variadic Generics	?	2020
647	User-Defined Type Guards	3.10	2021
649	Deferred Evaluation Of Annotations Using Descriptors	?	2021
655	Marking individual TypedDict items as required or potentially-missing	?	2021

Python 官方文档很难及时跟进所有 PEP，因此 Mypy 文档是必不可少的参考资料。据我所知，*Robust Python* (Patrick Viafore 著) 是第一本全面介绍 Python 静态类型系统的书。你现在读的这本有可能是第二本。

“型变”这个晦涩的话题在 PEP 484 中有专门的一节讲解，另外在 Mypy 文档的“Generics”页面以及含有大量信息的“Common Issues”页

面也有涉及。

如果打算使用与 `typing.get_type_hints` 函数互补的 `inspect` 模块，“PEP 362—Function Signature Object”值得一读。

如果对 Python 的历史感兴趣，可以阅读 Guido van Rossum 于 2004 年 12 月 23 日发表的“Adding Optional Static Typing to Python”一文。

来自伦斯勒理工学院和 IBM 公司 TJ Watson 研究中心的 Ingkarat Rak-amnouykit 等人发布的研究报告“Python 3 Types in the Wild: A Tale of Two Type Systems”，对 GitHub 中的开源项目使用类型提示的情况展开调查，结果表明，大多数项目没有使用类型提示，而且使用类型提示的项目大多没有使用类型检查工具。我觉得这篇报告最有趣的部分是对 Mypy 和谷歌的 pytype 语义差异的讨论，他们得出的结论是：二者本质上是不同的类型系统。

开创渐进式类型的论文有两篇：Gilad Bracha 的“Pluggable Type Systems”以及 Eric Meijer 和 Peter Drayton 的“Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages”。¹⁷

¹⁷连脚注都不错过的读者应该记得，以食堂类比解释型变就是借鉴自 Erik Meijer。

其他语言也实现了一些相同的想法。阅读有关那些语言的书中相关的内容，让我学到很多。

- *Atomic Kotlin* (Bruce Eckel 和 Svetlana Isakova 著)
- 《Effective Java 中文版（原书第 3 版）》
- 《编程与类型系统》
- 《TypeScript 编程》
- 《Dart 编程语言》¹⁸

¹⁸该书针对 Dart 1。Dart 2 变化极大，包括类型系统。不过，该书作者 Gilad Bracha 是编程语言设计领域重要的研究者，该书站在他的角度分析了 Dart 的设计，还是值得一读的。

如果想了解对类型系统的批评性意见，推荐阅读 Victor Youdaiken 写的两篇文章：“Bad ideas in type theory”和“Types considered harmful II”。

最后，我惊讶地发现了 Ken Arnold 写的“Generics Considered Harmful”一文。Arnold 自 Java 诞生伊始就是核心贡献者，还是官方认可的《Java 程序设计语言》一书前 4 版的作者之一（Java 首席设计师 James Gosling 也是作者之一）。

遗憾的是，Arnold 的批评同样适用于 Python 的静态类型系统。在阅读有关类型的 PEP 时，每每看到各种规则和特殊情况，我总是能想起 Arnold 那篇文章中的这段话。

这就引出了我经常提到的一个有关 C++ 的问题，我称之为“N 阶例外中的例外规则”。就好比你说了一句话：“对甲来说，除非甲做了乙，而且满足丙，那么你可以做丁。”

幸好，与 Java 和 C++ 相比，Python 有一个关键优势：类型系统是可选的。如果觉得太麻烦，则可以静默类型检查工具，忽略类型提示。

杂谈

类型无底洞

使用类型检查工具，有时迫不得已要导入不需要知道的类，而除了编写类型提示外，这些类在代码中根本用不到。这些类没有文档记录，可能是因为包的作者认为它们是实现细节。标准库中就有两例。

为了在 15.4 节的 `server.sockets` 示例中使用 `cast()`，我翻阅了 `asyncio` 包的大量文档，还浏览了该包中多个模块的源码，最后在文档中没有记录的 `asyncio.trsock` 模块中找到了同样没有记录的 `TransportSocket` 类。直接使用

`socket.socket` 是不对的，因为根据源码中的文档字符串，`TransportSocket` 显然不是 `socket.socket` 的子类型。

为示例 19-13（简单演示 `multiprocessing`）添加类型提示时，我也陷入了类似的困境。那个示例使用的 `SimpleQueue` 对象，通过调用 `multiprocessing.SimpleQueue()` 得到。但是，我不能在类型提示中使用那个名称，因为 `multiprocessing.SimpleQueue` 不是一个类，而是文档没有记录的 `multiprocessing.BaseContext` 类的绑定方法，构建并返回 `SimpleQueue` 类（在文档没有记录的 `multiprocessing.queues` 模块中定义）的一个实例。

在这两种情况下，仅仅为了编写一个类型提示，就要花几个小时寻找该导入的类，而且文档中没有线索。当然，我是写书，必须研究透彻。如果是编写应用程序代码，我应该不会为了一行类型不全的代码浪费这么多精力，仅仅写上 `# type: ignore` 就可以搞定。有时，这是唯一合算的方案。

其他语言的型变表示法

型变是一个晦涩的话题，而且 Python 的类型提示句法没有发挥多大作用。PEP 484 中的一句话直接证明了这一点：

协变或逆变不是类型变量的性质，而是使用类型变量定义的泛化类的性质。¹⁹

既然如此，协变和逆变为什么通过 `TypeVar` 声明，而不在泛化类上声明？

PEP 484 的作者对自己的工作提出了严格要求，即在不对解释器进行任何更改的情况下支持类型提示。鉴于此，需要引入 `TypeVar` 来定义类型变量，还要滥用 `[]`，为泛化提供 `Klass[T]` 句法——不像 C#、Java、Kotlin 和 TypeScript 等其

他流行的语言，使用 `Klass<T>` 表示法。这些语言都不要求在使用类型变量之前先声明。

此外，Kotlin 和 C# 的句法直接在类或接口声明中明确类型参数是协变的、逆变的还是不变的。

在 Kotlin 中，`BeverageDispenser` 类可以像下面这样声明。

```
class BeverageDispenser<out T> {  
    // ...  
}
```

形式类型参数中的 `out` 修饰符表示 `T` 是一种“输出”类型，因此 `BeverageDispenser` 是可协变的。

你或许能猜到如何声明 `TrashCan`。

```
class TrashCan<in T> {  
    // ...  
}
```

由于 `T` 是“输入”形式类型参数，因此 `TrashCan` 是可逆变的。

如果没有 `in` 或 `out`，那么类的参数是不变的。

根据“型变经验法则”一节给出的经验法则，很容易判断在形式类型参数中应该使用 `out` 还是 `in`。

鉴于此，Python 中的类型变量应该使用以下命名约定表示协变和逆变。

```
T_out = TypeVar('T_out', covariant=True)  
T_in = TypeVar('T_in', contravariant=True)
```

然后，像下面这样定义类。

```
class BeverageDispenser(Generic[T_out]):  
    ...  
  
class TrashCan(Generic[T_in]):  
    ...
```

现在改变 PEP 484 确立的命名约定是不是有点儿晚了？

¹⁹摘自 PEP 484 中“Covariance and Contravariance”一节的最后一段。

第 16 章 运算符重载

有些事情让我不安，比如运算符重载。我决定不支持运算符重载，这完全是个人选择，因为我见过太多 C++ 程序员滥用它。

——James Gosling
Java 之父¹

¹摘自“The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling”一文。

在 Python 中，可以使用如下公式计算复利。

```
interest = principal * ((1 + rate) ** periods - 1)
```

像 `1 + rate` 这样位于操作数之间的运算符叫作**中缀运算符**。在 Python 中，中缀运算符可以处理任何类型。因此，如果真想处理金额，则可以确保 `principal`、`rate` 和 `periods` 都是精确的数值（Python 类 `decimal.Decimal` 的实例），这样，上述公式就能得到精确的结果。

而在 Java 中，如果为了得到精确结果，把 `float` 换成 `BigDecimal`，就不能使用中缀运算符了，因为 Java 中缀运算符只可用于原始类型。为了让计算公式正确处理 `BigDecimal` 数值，在 Java 中要像下面这样写。

```
BigDecimal interest = principal.multiply(BigDecimal.ONE.add(rate)
    .pow(periods).subtract(BigDecimal.ONE));
```

显然，使用中缀运算符写出的公式可读性更高。为此，运算符必须能重载，让用户定义或扩展的类型（例如 NumPy 数组）也支持中缀运算符表示法。一门易于使用的高级语言，再加上支持运算符重载，这

或许是 Python 在数据科学领域（包括金融和科学应用程序）取得巨大成功的关键原因。

在 1.3.1 节中，我们为 `Vector` 类简略实现了几个运算符。示例 1-2 中的 `__add__` 方法和 `__mul__` 方法是为了展示如何使用特殊方法重载运算符，不过有些小问题被我们忽视了。此外，示例 11-2 中定义的 `Vector2d.__eq__` 方法认为 `Vector(3, 4) == [3, 4]` 为真，这可能并不合理。本章将解决这些问题，另外还将讨论以下话题：

- 中缀运算符如何表明自己无法处理某个操作数；
- 使用鸭子类型或大鹅类型处理不同类型的操作数；
- 众多比较运算符（例如 `==`、`>`、`<=` 等）的特殊行为；
- 增量赋值运算符（例如 `+=`）的默认处理方式和重载方式。

16.1 本章新增内容

大鹅类型是 Python 的关键组成部分，但是静态类型不支持 `numbers` 包中的抽象基类，因此我把示例 16-11 中针对 `numbers.Real` 的 `isinstance` 检查换掉了，改用鸭子类型。²

²Python 标准库中的其他抽象基类对大鹅类型和静态类型仍有价值。`numbers` 包中的抽象基类存在的问题详见 13.6.8 节。

本书第 1 版出版时，Python 3.5 还在内测阶段，矩阵乘法运算符 `@` 是在一个附注栏中介绍的。现在，那部分内容融入了本章正文，位于 16.6 节中。另外，我还利用大鹅类型重新实现了 `__matmul__` 方法，其比本书第 1 版中的 `__matmul__` 方法更安全，而且不失灵活性。

16.11 节新增了几份参考资料，包括 Guido van Rossum 写的一篇博客文章。还提到了两个在数学领域之外有效运用运算符重载的库：`pathlib` 和 `Scapy`。

16.2 运算符重载入门

运算符重载的作用是让用户定义的对象能够使用中缀运算符（例如 `+` 和 `|`）或一元运算符（例如 `-` 和 `~`）。说得宽泛一些，在 Python 中，函数调用（`()`）、属性访问（`.`）以及项访问和切片（`[]`）也是运算符，不过本章只讨论一元运算符和中缀运算符。

在某些圈子中，运算符重载的名声并不好。这个语言功能可能（已经）被滥用，让程序员困惑，导致 bug 和意料之外的性能瓶颈。但是，如果使用得当，则会把 API 变得更好用，把代码变得更易于阅读。Python 施加了一些限制，在灵活性、可用性和安全性方面取得了很好的平衡：

- 不能改变内置类型的运算符表达的意思；
- 不能新建运算符，只能重载现有运算符；
- 有些运算符不能重载：`is`、`and`、`or` 和 `not`（不过按位运算符 `&`、`|` 和 `~` 可以重载）。

第 12 章已经为 `Vector` 定义了一个中缀运算符，即由 `__eq__` 方法支持的 `==`。本章将改进 `__eq__` 方法的实现，更好地处理 `Vector` 以外的操作数类型。然而，在运算符重载方面，数量众多的比较运算符（`==`、`!=`、`>`、`<`、`>=` 和 `<=`）是特例，因此本章将首先在 `Vector` 中重载 4 个算术运算符：一元运算符 `-` 和 `+`，以及中缀运算符 `+` 和 `*`。

先来介绍一元运算符，这是最简单的算数运算符。

16.3 一元运算符

《Python 语言参考手册》中的 6.6 节“一元算术和位运算”列出了 3 个一元运算符。下面是这 3 个运算符和其对应的特殊方法。

- （由 `__neg__` 实现）

一元取反算术运算符。如果 x 是 -2 ，那么 $-x == 2$ 。

$+$ (由 `__pos__` 实现)

一元取正算术运算符。通常 $x == +x$ ，但也有一些例外。如果感到好奇，请阅读后面的“ x 和 $+x$ 何时不相等”附注栏。

$\sim$ (由 `__invert__` 实现)

按位否定 (取反) 整数，定义为 $\sim x == -(x+1)$ 。如果 x 是 2 ，那么 $\sim x == -3$ 。

《Python 语言参考手册》中的第 3 章还把内置函数 `abs()` 列为一元运算符。前文说过，它对应的特殊方法是 `__abs__`。

支持一元运算符很简单，只需实现相应的特殊方法。这些特殊方法只有一个参数，即 `self`。特殊方法的实现要符合所在类的逻辑。此外，还应遵守运算符的一个基本规则：始终返回新对象。也就是说，不要修改 `self`，应创建并返回合适类型的新实例。

对 $-$ 和 $+$ 来说，结果可能是与 `self` 同属一类的实例。对于一元运算符 $+$ ，如果接收者是不可变对象，就应该返回 `self`；否则，应返回 `self` 的副本。对于 `abs()`，结果应该是一个标量。

对于 $\sim$ ，很难说什么结果是合理的，因为处理的可能不是整数的位。在数据分析包 `pandas` 中， $\sim$ 运算符取反布尔筛选条件，相关示例见 `pandas` 文档中的“Boolean indexing”一节。

如前所述，我们将为第 12 章定义的 `Vector` 类实现几个新运算符。示例 16-1 列出了示例 12-16 实现的 `__abs__` 方法，以及新增加的一元运算符方法 `__neg__` 和 `__pos__`。

示例 16-1 `vector_v6.py`: 把一元运算符 $-$ 和 $+$ 添加到示例 12-16 中


```
def __abs__(self):
    return math.hypot(*self)

def __neg__(self):
    return Vector(-x for x in self) ❶

def __pos__(self):
    return Vector(self) ❷
```

❶ 为了计算 $-v$ ，构建一个新 `Vector` 实例，把 `self` 的每个分量都取反。

❷ 为了计算 $+v$ ，构建一个新 `Vector` 实例，`self` 的每个分量保持不变。

还记得吗？`Vector` 实例是可迭代的，而且 `Vector.__init__` 方法接受一个可迭代的参数，因此 `__neg__` 和 `__pos__` 的实现短小精悍。

我们不打算实现 `__invert__` 方法，因此如果用户试图对 `Vector` 实例做 $\sim v$ 运算，那么 Python 就会抛出 `TypeError`，而且会输出明确的错误消息“bad operand type for unary $\sim$: 'Vector'”。

以下内容讨论了一个奇怪的问题，能增加你对一元运算符 $+$ 的认识。

x 和 $+x$ 何时不相等

每个人都觉得 $x == +x$ 是对的，而且在 Python 中，大部分情况下是这样。但是，我在标准库中找到了两例 $x != +x$ 的情况。

第一例与 `decimal.Decimal` 类有关。如果 x 是在算术运算上下文中创建的 `Decimal` 实例，然后在不同的上下文中计算 $+x$ ，那么 $x != +x$ 。例如， x 所在的上下文使用某个精度，而计算 $+x$ 时，精度变了，如示例 16-2 所示。

示例 16-2 算术运算上下文的精度变化可能导致 x 不等于 $+x$

```
>>> import decimal
>>> ctx = decimal.getcontext()    ❶
>>> ctx.prec = 40                 ❷
>>> one_third = decimal.Decimal('1') / decimal.Decimal('3')    ❸
>>> one_third                     ❹
Decimal('0.3333333333333333333333333333333333333333333333333')
>>> one_third == +one_third      ❺
True
>>> ctx.prec = 28                ❻
>>> one_third == +one_third      ❼
False
>>> +one_third                   ❽
Decimal('0.333333333333333333333333333333333333333333333333')
```

- 1 获取当前全局算术运算的上下文引用。
- 2 把算术运算上下文的精度设为 40。
- 3 使用当前精度计算 $1/3$ 。
- 4 查看结果，小数点后有 40 位。
- 5 `one_third == +one_third` 返回 True。
- 6 把精度降到 28，这是 Decimal 算术运算的默认精度。
- 7 现在，`one_third == +one_third` 返回 False。
- 8 查看 `+one_third`，小数点后有 28 位。

这是因为每个 `+one_third` 表达式都会使用 `one_third` 的值创建一个新 `Decimal` 实例，但是使用的精度来自当前算术运算上下文。

`x += x` 的第二例在 `collections.Counter` 的文档中。`Counter` 类实现了几个算术运算符，包括中缀运算符 `+`，其作用是把两个 `Counter` 实例的计数器加在一起。然而，从实用角度出发，相加时，负值或零值计数会从结果中剔除。而一元运算符 `+` 等同于加上

一个空 Counter，因此它会产生一个新 Counter，而且仅保留大于零的计数器。如示例 16-3 所示。

示例 16-3 一元运算符 + 得到一个新 Counter 实例，不含零值或负值计数器

```
>>> ct = Counter('abracadabra')
>>> ct
Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
>>> ct['r'] = -3
>>> ct['d'] = 0
>>> ct
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 0, 'r': -3})
>>> +ct
Counter({'a': 5, 'b': 2, 'c': 1})
```

可以看到，+ct 返回的 Counter 实例只有大于零的计数器。

下面回归正题。

16.4 重载向量加法运算符 +

Vector 类是一种序列类型。按照《Python 语言参考手册》中的 3.3.7 节“模拟容器类型”所说，序列应该支持用于拼接的 + 运算符和用于重复复制的 * 运算符。然而，我们将使用向量数学运算实现运算符 + 和 *。虽然这么做更难一些，但是对 Vector 类型来说更有意义。



如果用户想拼接或重复复制 Vector 实例，那么可以把 Vector 实例转换成元组或列表，运用相关的运算符之后再转换回来——这归功于 Vector 是可迭代对象，而且可以由一个可迭代对象构建。

```
>>> v_concatenated = Vector(list(v1) + list(v2))
>>> v_repeated = Vector(tuple(v1) * 5)
```

两个欧几里得向量相加，得到一个新向量，各分量是两个向量中相应分量之和。

```
>>> v1 = Vector([3, 4, 5])
>>> v2 = Vector([6, 7, 8])
>>> v1 + v2
Vector([9.0, 11.0, 13.0])
>>> v1 + v2 == Vector([3 + 6, 4 + 7, 5 + 8])
True
```

如果尝试把两个长度不同的 `Vector` 实例加在一起会怎样呢？此时可以抛出错误，但是根据实际运用情况（例如信息检索），最好使用零来填充较短的那个向量。我们想要的效果如下所示。

```
>>> v1 = Vector([3, 4, 5, 6])
>>> v3 = Vector([1, 2])
>>> v1 + v3
Vector([4.0, 6.0, 5.0, 6.0])
```

确定这些基本要求之后，`__add__` 方法的实现如示例 16-4 所示。

示例 16-4 `Vector.__add__` 方法，第 1 版

```
# 在Vector类中定义

def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    ❶
    return Vector(a + b for a, b in pairs) ❷
```

❶ `pairs` 是一个生成器，会生成 `(a, b)` 形式的元组，其中 `a` 来自 `self`，`b` 来自 `other`。如果 `self` 和 `other` 的长度不同，那么可以使用 `fillvalue` 填充较短的那个可迭代对象缺少的值。

❷ 构建一个新 `Vector` 实例，使用生成器表达式计算 `pairs` 中各个 `(a, b)` 的和。

注意，`__add__` 会返回一个新 `Vector` 实例，没有更改 `self` 或 `other`。



实现一元运算符和中缀运算符的特殊方法一定不能修改操作数。使用这些运算符的表达式预期结果是创建新对象。只有增量赋值运算符可以修改第一个操作数，即 `self`（详见 16.9 节）。

根据示例 16-4，可以把 `Vector` 加到 `Vector2d` 上，还可以把 `Vector` 加到元组或任何生成数值的可迭代对象上，如示例 16-5 所示。

示例 16-5 第 1 版 `Vector.__add__` 方法也支持 `Vector` 之外的对象

```
>>> v1 = Vector([3, 4, 5])
>>> v1 + (10, 20, 30)
Vector([13.0, 24.0, 35.0])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v1 + v2d
Vector([4.0, 6.0, 5.0])
```

示例 16-5 中的两个加法都能如我们所期待的那样计算。这是因为 `__add__` 使用了 `zip_longest(...)`，它能处理任何可迭代对象，而且构建新 `Vector` 实例的生成器表达式仅仅是把 `zip_longest(...)` 生成的一对值相加 (`a + b`)，因此可以使用任何生成数值项的可迭代对象。

然而，如果对调操作数，那么混合类型的加法就会失败，如示例 16-6 所示。

示例 16-6 如果左侧操作数不是 `Vector` 对象，那么第 1 版 `Vector.__add__` 方法将无法处理

```
>>> v1 = Vector([3, 4, 5])
>>> (10, 20, 30) + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "Vector") to tuple
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v2d + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2d' and
'Vector'
```

为了支持涉及不同类型对象的运算，Python 为中缀运算符特殊方法提供了特殊的分派机制。对于表达式 $a + b$ ，解释器将执行以下几步操作（另见图 16-1）。

01. 如果 a 有 `__add__` 方法，而且不返回 `NotImplemented`，就调用 `a.__add__(b)`，返回结果。
02. 如果 a 没有 `__add__` 方法，或者调用 `__add__` 方法返回 `NotImplemented`，就检查 b 有没有 `__radd__` 方法，如果有，而且不返回 `NotImplemented`，就调用 `b.__radd__(a)`，返回结果。
03. 如果 b 没有 `__radd__` 方法，或者调用 `__radd__` 方法返回 `NotImplemented`，就抛出 `TypeError`，并在错误消息中指明**不支持操作数的类型**。



`__radd__` 是 `__add__` 的“反射”（reflected）版本或“反向”（reversed）版本。我喜欢把它叫作“反向”特殊方法。³

³这两个术语在 Python 文档中都有使用。《Python 语言参考手册》第 3 章用的是“reflected”（反射），而 Python 标准库文档中 `numbers` 模块文档的“9.1.2.2. Implementing the arithmetic operations”一节用的是“forward”（正向）方法和“reverse”（反向）方法。我觉得后者更好，因为“正向”和“反向”明确指出了方向，而“反射”则没有这种效果。

最简单可用的 `__radd__` 实现如示例 16-7 所示。

示例 16-7 Vector 类的 `__add__` 方法和 `__radd__` 方法

```
# 在Vector类中定义

def __add__(self, other): ❶
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    return Vector(a + b for a, b in pairs)

def __radd__(self, other): ❷
    return self + other
```

❶ `__add__` 方法没有变化，与示例 16-4 中一样。这里列出 `__add__` 是因为 `__radd__` 要用它。

❷ `__radd__` 直接委托 `__add__`。

`__radd__` 通常就是这么简单：直接调用适当的运算符，这里就是委托 `__add__`。任何满足交换律的运算符都能这么做。`+` 在处理数值和向量时满足交换律，在拼接 Python 序列时则不满足。

如果 `__radd__` 只是调用 `__add__`，那么还有一种同效方法。

```
def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    return Vector(a + b for a, b in pairs)

__radd__ = __add__
```

示例 16-7 中的方法可以处理 `Vector` 对象，或者任何生成数值项的可迭代对象，例如 `Vector2d` 实例、整数元组或浮点数数组。但是，如果提供的对象不可迭代，则 `__add__` 会抛出异常，而且提供的错误消息不是很有用，如示例 16-8 所示。

示例 16-8 `Vector.__add__` 方法的操作数应是可迭代对象


```
>>> v1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 328, in __add__
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
TypeError: zip_longest argument #2 must support iteration
```

如果一个操作数是可迭代对象，但是它生成的项不能与 `Vector` 中的浮点数项相加，那么给出的消息也没什么用，如示例 16-9 所示。

示例 16-9 `Vector.__add__` 方法的操作数应是生成数值项的可迭代对象

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs)
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components)
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

我尝试把 `Vector` 加到一个字符串上，但错误消息中出现的是 `float` 和 `str`。

示例 16-8 和示例 16-9 揭露的问题比晦涩难懂的错误消息更严重：如果由于类型不兼容而导致运算符特殊方法无法返回有效的结果，那么应该返回 `NotImplemented`，而不是抛出 `TypeError`。如果返回 `NotImplemented`，那么另一个操作数所属的类型还有机会执行运算，即 Python 会尝试调用反向方法。

为了遵守鸭子类型精神，不能测试操作数 `other` 或者所含元素的类型。要捕获异常，然后返回 `NotImplemented`。如果解释器还未反转操作数，那么它将尝试去做。如果反向方法调用返回 `NotImplemented`，则 Python 会抛出 `TypeError`，返回一个标准

的错误消息，例如“unsupported operand type(s) for +: Vector and str”。

示例 16-10 是实现 `Vector` 加法的特殊方法的最终版。

示例 16-10 `vector_v6.py`: 添加到 `vector_v5.py` (参见示例 12-16) 中的 `+` 运算符方法

```
def __add__(self, other):
    try:
        pairs = itertools.zip_longest(self, other,
fillvalue=0.0)
        return Vector(a + b for a, b in pairs)
    except TypeError:
        return NotImplemented

def __radd__(self, other):
    return self + other
```

注意，`__add__` 方法现在会捕获 `TypeError`，返回 `NotImplemented`。



如果中缀运算符方法抛出异常，那么它将终止运算符分派机制。对于 `TypeError`，通常最好将其捕获，然后返回 `NotImplemented`。这样解释器就会尝试调用反向运算符方法，如果操作数是不同的类型，那么对调之后，反向运算符方法可能会正确计算。

至此，我们编写的 `__add__` 方法和 `__radd__` 方法安全重载了 `+` 运算符。接下来重载另一个中缀运算符：`*`。

16.5 重载标量乘法运算符 `*`

`Vector([1, 2, 3]) * x` 是什么意思？如果 `x` 是数值，就是计算标量积 (scalar product)，结果是一个新 `Vector` 实例，各个分量都

乘以 x ——这也叫元素级乘法 (elementwise multiplication) 。

```
>>> v1 = Vector([1, 2, 3])
>>> v1 * 10
Vector([10.0, 20.0, 30.0])
>>> 11 * v1
Vector([11.0, 22.0, 33.0])
```



操作数涉及 `Vector` 的积还有一种，叫两个向量的点积 (dot product)。如果把一个向量看作 $1 \times N$ 矩阵，把另一个向量看作 $N \times 1$ 矩阵，那么这就是矩阵乘法。`Vector` 类的这种运算会在 16.6 节实现。

回到标量积话题。依然先实现最简单可用的 `__mul__` 方法和 `__rmul__` 方法。

```
# 在Vector类中定义

def __mul__(self, scalar):
    return Vector(n * scalar for n in self)

def __rmul__(self, scalar):
    return self * scalar
```

这两个方法确实可用，但是提供不兼容的操作数时会出问题。`scalar` 参数的值必须是数值，与浮点数相乘得到的积是另一个浮点数（因为 `Vector` 类在内部使用浮点数数组）。因此，不能使用 `complex` 数，但可以是 `int`、`bool` (`int` 的子类)，甚至 `fractions.Fraction` 实例等标量。示例 16-11 中的 `__mul__` 方法没有明确检查 `scalar` 的类型，而是把它转换成了一个 `float` 值，如果失败就返回 `NotImplemented`——充分利用鸭子类型。

示例 16-11 `vector_v7.py`: 增加 `*` 运算符方法

```
class Vector:
    typecode = 'd'

    def __init__(self, components):
```

```

        self._components = array(self.typecode, components)

# 排版需要，省略了很多方法

def __mul__(self, scalar):
    try:
        factor = float(scalar)
    except TypeError: ❶
        return NotImplemented ❷
    return Vector(n * factor for n in self)

def __rmul__(self, scalar):
    return self * scalar ❸

```

❶ 如果 `scalar` 不能转换为 `float` 值.....

❷ 则说明不知道如何处理，因此返回 `NotImplemented`，让 Python 尝试在操作数 `scalar` 上调用 `__rmul__` 方法。

❸ 这里，`__rmul__` 方法只需执行 `self * scalar`，委托 `__mul__` 方法。

有了示例 16-11 中的代码之后，就可以拿 `Vector` 实例乘以常规和不那么寻常的标量数值类型了。

```

>>> v1 = Vector([1.0, 2.0, 3.0])
>>> 14 * v1
Vector([14.0, 28.0, 42.0])
>>> v1 * True
Vector([1.0, 2.0, 3.0])
>>> from fractions import Fraction
>>> v1 * Fraction(1, 3)
Vector([0.3333333333333333, 0.6666666666666666, 1.0])

```

现在，`Vector` 可以乘以标量值了。下面说明如何实现 `Vector` 与 `Vector` 的乘积。



在本书第 1 版中，示例 16-11 利用的是大鹅类型，使用 `isinstance(scalar, numbers.Real)` 检查 `__mul__` 方法

的 `scalar` 参数。现在，我尽量避免使用 `numbers` 包中的抽象基类，一是因为 PEP 484 不支持，二来觉得不应该在运行时使用无法静态检查的类型。

另外，也可以选择检查 `typing.SupportsFloat` 协议（参见 13.6.2 节）。但是，我最终选择利用鸭子类型，因为我认为流畅的 Python 程序员应该熟练掌握这种编程模式。

当然，我也没有抛弃大鹅类型。本书第 2 版新增的示例 16-12 就是一例。

16.6 把 @ 当作中缀运算符使用

众所周知，@ 符号是函数装饰器的前缀，但是从 2015 年开始，其也可用作中缀运算符。多年来，NumPy 中的点积一直写作 `numpy.dot(a, b)`。使用函数调用表示法难以把较长的数学公式使用的数学表示法转换成 Python 代码，⁴ 经数值计算社区的游说，Python 3.5 实现了“PEP 465—A dedicated infix operator for matrix multiplication”。如今，两个 NumPy 数组的点积可以写作 `a @ b`。

⁴这个问题的详细讨论见后面的“杂谈”部分。

@ 运算符由特殊方法 `__matmul__`、`__rmatmul__` 和 `__imatmul__` 支持，名称取自“matrix multiplication”（矩阵乘法）。目前，标准库还没有使用这些方法，不过自 Python 3.5 起，解释器已能识别。因此，NumPy 团队以及我们自己可以让用户定义的类型支持 @ 运算符。为了正确处理这个新运算符，Python 解析器也做了更改（`a @ b` 在 Python 3.4 中是错误的句法）。

下面这些简单的测试展示了如何使用 @ 运算符处理 `Vector` 实例。

```
>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
>>> va @ vz == 38.0 # 1*5 + 2*6 + 3*7
```

```
True
>>> [10, 20, 30] @ vz
380.0
>>> va @ 3
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
```

相关特殊方法的实现如示例 16-12 所示。

示例 16-12 vector_v7.py: @ 运算符相关的方法

```
class Vector:
    # 排版需要, 省略了很多方法

    def __matmul__(self, other):
        if (isinstance(other, abc.Sized) and ❶
            isinstance(other, abc.Iterable)):
            if len(self) == len(other): ❷
                return sum(a * b for a, b in zip(self, other)) ❸
            else:
                raise ValueError('@ requires vectors of equal
length.')
        else:
            return NotImplemented

    def __rmatmul__(self, other):
        return self @ other
```

❶ 两个操作数都必须实现 `__len__` 和 `__iter__`

❷ 而且长度相同.....

❸ 以便正确运用 `sum`、`zip` 和生成器表达式。



Python 3.10 中 `zip()` 函数的新功能

自 Python 3.10 起, 内置函数 `zip` 接受一个可选的仅限关键字参数 `strict`。当指定 `strict=True` 时, 如果可迭代对象的长度不同, 那么 `zip` 函数就会抛出 `ValueError`。`strict` 参数的默

认值为 `False`。新增的严格行为符合 Python 的**快速失败**原则。在示例 16-12 中，内层 `if` 可以替换成 `try/except ValueError`，并在 `zip` 调用中添加 `strict=True`。

示例 16-12 充分利用了**大鹅类型**。如果测试操作数 `other` 是否为 `Vector` 对象，那么 `@` 运算符的操作数就不能为列表或数组，这样用户就失去了一定的灵活性。只要其中一个操作数是 `Vector`，我们实现的 `@` 运算符就支持另一个操作数是 `abc.Sized` 和 `abc.Iterable` 的实例。这两个抽象基类都实现了 `__subclasshook__`，因此凡是提供 `__len__` 和 `__iter__` 的对象都满足测试条件——不用真正子类化或注册为虚拟子类（详见 13.5.8 节）。就 `Vector` 类而言，它没有子类化 `abc.Sized` 或 `abc.Iterable`，但是能通过针对二者的 `isinstance` 检查，因为 `Vector` 类有必要的方法。

下面总结一下 Python 支持的算术运算符。16.8 节会深入探讨一类特殊的运算符：比较运算符。

16.7 算术运算符总结

通过实现 `+`、`*` 和 `@`，本章讲解了编写中缀运算符最常用的模式。这些技术对表 16-1 中列出的所有运算符都适用（16.9 节会讨论就地运算符）。

表 16-1：中缀运算符方法的名称（就地运算符用于增量赋值；比较运算符在表 16-2 中）

运算符	正向方法	反向方法	就地方法	说明
<code>+</code>	<code>__add__</code>	<code>__radd__</code>	<code>__iadd__</code>	加法或拼接

运算符	正向方法	反向方法	就地方法	说明
-	<code>__sub__</code>	<code>__rsub__</code>	<code>__isub__</code>	减法
*	<code>__mul__</code>	<code>__rmul__</code>	<code>__imul__</code>	乘法或重复复制
/	<code>__truediv__</code>	<code>__rtruediv__</code>	<code>__itruediv__</code>	除法
//	<code>__floordiv__</code>	<code>__rfloordiv__</code>	<code>__ifloordiv__</code>	整除
%	<code>__mod__</code>	<code>__rmod__</code>	<code>__imod__</code>	求模
<code>divmod()</code>	<code>__divmod__</code>	<code>__rdivmod__</code>	<code>__idivmod__</code>	返回由整除的商和模数组成的元组
<code>**</code> 、 <code>__pow()</code>	<code>__pow__</code>	<code>__rpow__</code>	<code>__ipow__</code>	求幂 ^a
@	<code>__matmul__</code>	<code>__rmatmul__</code>	<code>__imatmul__</code>	矩阵乘法
&	<code>__and__</code>	<code>__rand__</code>	<code>__iand__</code>	位与
	<code>__or__</code>	<code>__ror__</code>	<code>__ior__</code>	位或
^	<code>__xor__</code>	<code>__rxor__</code>	<code>__ixor__</code>	位异或
<<	<code>__lshift__</code>	<code>__rlshift__</code>	<code>__ilshift__</code>	按位左移

运算符	正向方法	反向方法	就地方法	说明
>>	<code>__rshift__</code>	<code>__rrshift__</code>	<code>__irshift__</code>	按位右移

`a pow` 的第 3 个参数 `modulo` 是可选的：`pow(a, b, modulo)`。直接调用特殊方法时也支持这个参数，例如 `a.pow (b, modulo)`。

数量众多的比较运算符使用的规则稍有不同。

16.8 众多比较运算符

Python 解释器对众多比较运算符（`==`、`!=`、`>`、`<`、`>=` 和 `<=`）的处理与前文类似，不过在两个方面有重大区别。

- 正向调用和反向调用使用的是同一系列方法。相应规则如表 16-2 所示。例如，对 `==` 来说，正向调用和反向调用都是 `__eq__` 方法，只是把参数对调了。而正向的 `__gt__` 方法调用的是反向的 `__lt__` 方法，并把参数对调。
- 对于 `==` 和 `!=`，如果缺少反向方法或返回 `NotImplemented`，那么 Python 就会比较对象的 ID，而不抛出 `TypeError`。

表 16-2：众多比较运算符：如果正向方法返回 `NotImplemented`，就调用反向方法

分组	中缀运算符	正向方法调用	反向方法调用	后备机制
相等性	<code>a == b</code>	<code>a.__eq__(b)</code>	<code>b.__eq__(a)</code>	返回 <code>id(a) == id(b)</code>
	<code>a != b</code>	<code>a.__ne__(b)</code>	<code>b.__ne__(a)</code>	返回 <code>not (a == b)</code>

分组	中缀运算符	正向方法调用	反向方法调用	后备机制
排序	<code>a > b</code>	<code>a.__gt__(b)</code>	<code>b.__lt__(a)</code>	抛出 <code>TypeError</code>
	<code>a < b</code>	<code>a.__lt__(b)</code>	<code>b.__gt__(a)</code>	抛出 <code>TypeError</code>
	<code>a >= b</code>	<code>a.__ge__(b)</code>	<code>b.__le__(a)</code>	抛出 <code>TypeError</code>
	<code>a <= b</code>	<code>a.__le__(b)</code>	<code>b.__ge__(a)</code>	抛出 <code>TypeError</code>

了解这些规则之后，下面来分析并改进 `Vector.__eq__` 方法的行为。这个方法在 `vector_v5.py`（参见示例 12-16）中的定义方式如下所示。

```
class Vector:
    # 省略了很多行

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
```

这个方法的行为如示例 16-13 所示。

示例 16-13 将 `Vector` 实例与 `Vector` 实例、`Vector2d` 实例和元组进行比较

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d ❷
True
>>> t3 = (1, 2, 3)
```

```
>>> va == t3 ❸
True
```

- ❶ 两个具有相同数值分量的 `Vector` 实例是相等的。
- ❷ 如果 `Vector` 实例的分量与 `Vector2d` 实例的分量相等，那么二者就相等。
- ❸ 如果 `Vector` 实例的分量与元组或其他任何可迭代对象的数值项相等，那么二者也相等。

示例 16-13 中的结果可能不是很理想。我们真的希望一个 `Vector` 与包含相同数值的元组相等吗？我对这一点没有强制规定，要由应用程序上下文决定。不过，《Python 之禅》说道：

如果存在多种可能，则不要猜测。

对操作数过度宽容可能导致令人惊讶的结果，而程序员讨厌惊喜。

从 Python 自身来找线索，我们发现 `[1,2] == (1, 2)` 的结果是 `False`。因此，我们要保守一点儿，做些类型检查。如果第二个操作数是 `Vector` 实例（或者 `Vector` 子类的实例），那么就使用 `__eq__` 方法的当前逻辑。否则，返回 `NotImplemented`，让 Python 处理。如示例 16-14 所示。

示例 16-14 `vector_v8.py`：改进 `Vector` 类的 `__eq__` 方法

```
def __eq__(self, other):
    if isinstance(other, Vector): ❶
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
    else:
        return NotImplemented ❷
```

- ❶ 如果操作数 `other` 是 `Vector` 实例（或者 `Vector` 子类的实例），就像之前那样比较。

❷ 否则，返回 `NotImplemented`。

使用示例 16-14 中的新版 `Vector.__eq__` 方法运行示例 16-13 中的测试，得到的结果如示例 16-15 所示。

示例 16-15 与示例 16-13 一样的测试：最后一个结果变了

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3 ❸
False
```

❶ 结果与之前一样，符合预期。

❷ 结果与之前一样，但是为什么呢？稍后解释。

❸ 结果不同了，这才是我们想要的。但是为什么会这样？请往下读……

在示例 16-15 中的 3 个结果里，第一个没变，后两个变了。这是因为示例 16-14 中的 `__eq__` 方法返回了 `NotImplemented`。比较 `Vector` 实例与 `Vector2d` 实例 (`vc == v2d`) 的具体步骤如下。

01. 为了求解 `vc == v2d`，Python 调用 `Vector.__eq__(vc, v2d)`。
02. 经 `Vector.__eq__(vc, v2d)` 确认，`v2d` 不是 `Vector` 实例，因此返回 `NotImplemented`。
03. Python 得到 `NotImplemented` 结果，尝试调用 `Vector2d.__eq__(v2d, vc)`。

04. `Vector2d.__eq__(v2d, vc)` 把两个操作数都变成元组，然后比较，结果是 `True` (`Vector2d.__eq__` 方法的代码在示例 11-11 中)。

在示例 16-15 中，比较 `Vector` 实例和元组 (`va == t3`) 的具体步骤如下。

01. 为了求解 `va == t3`，Python 调用 `Vector.__eq__(va, t3)`。
02. 经 `Vector.__eq__(va, t3)` 确认，`t3` 不是 `Vector` 实例，因此返回 `NotImplemented`。
03. Python 得到 `NotImplemented` 结果，尝试调用 `tuple.__eq__(t3, va)`。
04. `tuple.__eq__(t3, va)` 不知道 `Vector` 是什么，因此返回 `NotImplemented`。
05. 对于 `==`，如果反向调用返回 `NotImplemented`，则 Python 会比较对象的 ID，做最后一搏。

不用实现支持 `!=` 运算符的 `__ne__` 方法，因为从 `object` 继承的 `__ne__` 方法的后备行为即可满足需求：只要定义了 `__eq__` 方法，而且不返回 `NotImplemented`，`__ne__` 就会对 `__eq__` 返回的结果取反。

也就是说，对示例 16-15 中的对象来说，使用 `!=` 运算符比较的结果正确。

```
>>> va != vb
False
>>> vc != v2d
False
>>> va != (1, 2, 3)
True
```

从 `object` 继承的 `__ne__` 方法的运作方式与下述代码类似，不过原版是用 C 语言实现的。⁵

⁵object.__eq__ 和 object.__ne__ 的逻辑在 object_richcompare 函数中，位于 CPython 源码的 Objects/typeobject.c 文件中。

```
def __ne__(self, other):
    eq_result = self == other
    if eq_result is NotImplemented:
        return NotImplemented
    else:
        return not eq_result
```

讨论完重要的中缀运算符重载之后，下面换一类运算符：增量赋值运算符。

16.9 增量赋值运算符

Vector 类已经支持增量赋值运算符 += 和 *= 了。这是因为当增量赋值运算符的接收者是可变对象时，将新建实例，重新绑定左侧变量。

+= 和 *= 的实际使用如示例 16-16 所示。

示例 16-16 使用 += 和 *= 操作 Vector 实例

```
>>> v1 = Vector([1, 2, 3])
>>> v1_alias = v1 ❶
>>> id(v1) ❷
4302860128
>>> v1 += Vector([4, 5, 6]) ❸
>>> v1 ❹
Vector([5.0, 7.0, 9.0])
>>> id(v1) ❺
4302859904
>>> v1_alias ❻
Vector([1.0, 2.0, 3.0])
>>> v1 *= 11 ❼
>>> v1 ❽
Vector([55.0, 77.0, 99.0])
>>> id(v1)
4302858336
```

- ❶ 创建一个别名，方便后面查看 `Vector([1, 2, 3])` 对象。
- ❷ 记住一开始绑定给 `v1` 的 `Vector` 实例的 ID。
- ❸ 增量加法运算。
- ❹ 结果与预期相符.....
- ❺ 但是创建了新的 `Vector` 实例。
- ❻ 查看 `v1_alias`，确认原来的 `Vector` 实例没被修改。
- ❼ 增量乘法运算。
- ❽ 同样，结果与预期相符，但是创建了新的 `Vector` 实例。

如果一个类没有实现表 16-1 列出的就地运算符，那么增量赋值运算符就只是语法糖：`a += b` 的作用与 `a = a + b` 完全一样。对于不可变类型，这是预期行为，而且，如果定义了 `__add__` 方法，那么不用编写额外的代码，`+=` 就能使用。

然而，如果实现了像 `__iadd__` 这样的就地运算符方法，那么计算 `a += b` 的结果时就会调用就地运算符方法。这种运算符的名称表明，它们会就地修改左侧操作数，而不创建新对象作为结果。



不可变类型（例如我们定义的 `Vector` 类）一定不能实现就地更改特殊方法。这是明显的事实，不过还是有必要指出来。

为了展示如何实现就地运算符，我们将扩展示例 13-9 中的 `BingoCage` 类，实现 `__add__` 方法和 `__iadd__` 方法。

把这个子类命名为 `AddableBingoCage`。示例 16-17 是我们想让 `+` 运算符具有的行为。

示例 16-17 使用 + 运算符新建 AddableBingoCage 实例

```
>>> vowels = 'AEIOU'
>>> globe = AddableBingoCage(vowels) ❶
>>> globe.inspect()
('A', 'E', 'I', 'O', 'U')
>>> globe.pick() in vowels ❷
True
>>> len(globe.inspect()) ❸
4
>>> globe2 = AddableBingoCage('XYZ') ❹
>>> globe3 = globe + globe2
>>> len(globe3.inspect()) ❺
7
>>> void = globe + [10, 20] ❻
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +:
'AddableBingoCage' and 'list'
```

- ❶ 创建一个 `globe` 实例，该实例包含 5 项（`vowels` 中的各个字母）。
- ❷ 从中取出一项，确认它在 `vowels` 中。
- ❸ 确认 `globe` 的项数减少到 4 个了。
- ❹ 创建第二个实例，该实例包含 3 项。
- ❺ 把前两个实例加在一起，创建第三个实例。这个实例有 7 项。
- ❻ `AddableBingoCage` 实例无法与列表相加，抛出 `TypeError`。那个错误消息是 `__add__` 方法返回 `NotImplemented` 时 Python 解释器输出的。

`AddableBingoCage` 是可变的，其实现 `__iadd__` 方法后的行为如示例 16-18 所示。

示例 16-18 可以使用 += 运算符载入现有的 AddableBingoCage 实例（接续示例 16-17）

```
>>> globe_orig = globe ❶
>>> len(globe.inspect()) ❷
4
>>> globe += globe2 ❸
>>> len(globe.inspect())
7
>>> globe += ['M', 'N'] ❹
>>> len(globe.inspect())
9
>>> globe is globe_orig ❺
True
>>> globe += 1 ❻
Traceback (most recent call last):
...
TypeError: right operand in += must be 'Tombola' or an
iterable
```

- ❶ 创建一个别名，方便后面检查对象的标识。
- ❷ 现在，globe 有 4 项。
- ❸ AddableBingoCage 实例可以从同属一类的其他实例那里接收项。
- ❹ += 右侧的操作数可以是任何可迭代对象。
- ❺ 在这个示例中，globe 始终指代 globe_orig 对象。
- ❻ AddableBingoCage 实例不能与非可迭代对象相加，原因详见错误消息。

注意，与 + 相比，+= 运算符对第二个操作数更宽容。+ 运算符的两个操作数必须是相同类型（这里是 AddableBingoCage），如若不然，结果的类型可能让人摸不着头脑。而 += 的情况更明确，因为就地修改左侧操作数，所以结果的类型是确定的。



通过观察内置类型 `list` 的工作方式，我确定了对 `+` 和 `+=` 的行为做什么限制。`my_list + x` 只能用于把两个列表加到一起，而 `my_list += x` 可以使用右侧可迭代对象 `x` 中的项扩展左侧列表。`list.extend()` 的行为也是如此，它的参数可以是任何可迭代对象。

明确 `AddableBingoCage` 的行为之后，下面来看实现方式，如示例 16-19 所示。回想一下示例 13-9，`BingoCage` 是示例 13-7 中抽象基类 `Tombola` 的具体子类。

示例 16-19 bingoaddable.py: `AddableBingoCage` 扩展了 `BingoCage` 以支持 `+` 和 `+=`

```
from tombola import Tombola
from bingo import BingoCage

class AddableBingoCage(BingoCage): ❶

    def __add__(self, other):
        if isinstance(other, Tombola): ❷
            return AddableBingoCage(self.inspect() +
other.inspect())
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, Tombola):
            other_iterable = other.inspect() ❸
        else:
            try:
                other_iterable = iter(other) ❹
            except TypeError: ❺
                msg = ('right operand in += must be '
                        "'Tombola' or an iterable")
                raise TypeError(msg)
            self.load(other_iterable) ❻
        return self ❼
```

❶ `AddableBingoCage` 扩展了 `BingoCage`。

- ❷ `__add__` 方法的第二个操作数只能是 `Tombola` 实例。
- ❸ 在 `__iadd__` 中，如果 `other` 是 `Tombola` 实例，就从中获取项。
- ❹ 否则，尝试使用 `other` 创建迭代器。⁶

⁶第 17 章会讨论内置函数 `iter`。这里使用 `tuple(other)` 也是可以的，但是 `.load(...)` 方法迭代参数时要构建元组，消耗较大。

- ❺ 如果尝试失败，就抛出异常，告知用户该怎么做。如果可能，错误消息应该明确指导用户如何解决问题。
- ❻ 如果能执行到这里，就把 `other_iterable` 载入 `self`。
- ❼ 重要提醒：可变对象的增量赋值特殊方法必须返回 `self`。这是用户预期的行为。

通过示例 16-19 中 `__add__` 和 `__iadd__` 返回结果的方式可以总结出就地运算符的原理。

`__add__`

调用 `AddableBingoCage` 构造函数构建一个新实例，作为结果返回。

`__iadd__`

把修改后的 `self` 作为结果返回。

最后，示例 16-19 中还有一点要注意：从设计上看，`AddableBingoCage` 不用定义 `__radd__` 方法，因为不需要。如果右侧操作数是相同类型，那么正向方法 `__add__` 会处理。因此，当 Python 计算 `a + b` 时，如果 `a` 是 `AddableBingoCage` 实例，而 `b` 不是，则返回 `NotImplemented`，此时或许可以让 `b` 所属的类接手

处理。然而，如果是表达式 `b + a`，而 `b` 不是 `AddableBingoCage` 实例，并返回了 `NotImplemented`，那么 Python 最好放弃，抛出 `TypeError`，因为无法处理 `b`。



一般来说，如果中缀运算符的正向方法（例如 `__mul__`）只处理与 `self` 属于同一类型的操作数，那么就无须实现对应的反向方法（例如 `__rmul__`），因为按照定义，反向方法是为了处理类型不同的操作数。

对 Python 运算符重载的讨论到此结束。

16.10 本章小结

本章首先说明了 Python 对运算符重载施加的一些限制：禁止重载内置类型的运算符，而且限于重载现有的运算符，不过有几个例外（`is`、`and`、`or` 和 `not`）。

随后讲解了如何重载一元运算符，实现了 `__neg__` 方法和 `__pos__` 方法。接着，重载中缀运算符，首先是由 `__add__` 方法提供支持的 `+` 运算符。我们得知，一元运算符和中缀运算符的结果应该是新对象，绝不能修改操作数。为了支持其他类型，我们返回了特殊的 `NotImplemented` 值（不是异常），让解释器尝试对调操作数，然后调用运算符的反向特殊方法（例如 `__radd__`）。图 16-1 中的流程图概述了 Python 处理中缀运算符的算法。

如果操作数的类型不同，那么就要检测出不能处理的操作数。本章使用两种方式处理这个问题：一种是利用鸭子类型，直接尝试执行运算，如果有问题，就捕获 `TypeError` 异常；另一种是显式使用 `isinstance` 测试，`__mul__` 方法和 `__matmul__` 方法就是这么做的。这两种方式各有优劣：鸭子类型更灵活，显式检查则更能预知结果。

一般来说，代码库应该利用鸭子类型，让用户能使用尽可能多的对象，不管类型，只要支持必要的操作。然而，在鸭子类型下，Python 的运算符分派算法可能输出容易让人误解的错误消息或预料之外的结果。鉴于此，在重载运算符的特殊方法时，通常更有效的方法是使用 `isinstance` 测试抽象基类，检查类型。这就是 Alex Martelli 所说的大鹅类型（详见 13.5 节）。大鹅类型能很好地平衡灵活性和安全性，因为现有的或以后可能出现的用户定义的类型可以声明为抽象基类的真正子类或虚拟子类。而且，如果抽象基类实现了 `__subclasshook__`，那么对象只要提供了所需的方法，无须子类化或注册就能通过针对抽象基类的 `isinstance` 检查。

接下来的话题是数量众多的比较运算符。我们通过 `__eq__` 方法实现了 `==`，而且发现 Python 在 `object` 基类中通过 `__ne__` 方法为 `!=` 提供了便利的实现。Python 处理这些运算符的方式与 `>`、`<`、`>=` 和 `<=` 稍有不同，具体而言是选择反向方法的逻辑不同，此外 Python 还会特别处理 `==` 和 `!=` 的后备机制：从不抛出错误，因为 Python 会比较对象的 ID，做最后一搏。

最后，本章专门讨论了增量赋值运算符。我们发现，Python 处理这种运算符的默认方式是把它们当作常规的运算符外加赋值操作，即 `a += b` 其实被当成 `a = a + b` 处理。这个操作始终会创建新对象，因此可变类型和不可变类型都适用。对于可变对象，可以实现就地特殊方法，例如支持 `+=` 的 `__iadd__` 方法，然后修改左侧操作数的值。为了举例说明，我们把不可变的 `Vector` 类放到一边，为 `BingoCage` 的子类实现了 `+=` 运算符，把项添加到随机选号池中，这与内置的 `list` 类型把 `+=` 当成 `list.extend()` 方法的快捷方式类似。在实现的过程中，我们得知，在可接受的类型方面，`+` 应该比 `+=` 严格。对于序列类型，`+` 通常要求两个操作数属于同一类型，而 `+=` 的右侧操作数往往可以是任何可迭代对象。

16.11 延伸阅读

Guido van Rossum 写了一篇文章为运算符重载辩护，题为“Why operators are useful”。Trey Hunner 在“Tuple ordering and deep comparisons in Python”一文中指出，Python 提供的众多比较运算符比很多来自其他编程语言的程序员想象中更灵活且更强大。

在 Python 编程中，运算符重载经常使用 `isinstance` 做测试。这种测试充分利用了大鹅类型（详见 13.5 节）。如果你跳过了那一节，请务必翻回去阅读。

运算符特殊方法的主要参考资料是《Python 语言参考手册》中的第 3 章。Python 标准库文档的 `numbers` 模块文档中的“9.1.2.2. Implementing the arithmetic operations”一节也值得一读。

Python 3.4 增加的 `pathlib` 包就灵活利用了运算符重载。`Path` 类重载了 `/` 运算符，根据字符串构建文件系统路径，如文档中下面这个示例所示。

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
```

在非算术运算符方面，用于“发送、嗅探、剖析和伪造网络分组”的 Scapy 库也利用了运算符重载。在 Scapy 中，`/` 把来自不同网络层的字段堆叠起来构建分组。详见文档中的“Stacking layers”一节。

如果想实现比较运算符，可以研究一下 `functools.total_ordering`。这是一个类装饰器，自动为只定义几个比较运算符的类生成全部比较运算符。详见 `functools` 模块文档。

如果对动态类型语言的运算符方法分派机制感兴趣，推荐阅读两篇具有重大意义的论文：Dan Ingalls（Smalltalk 团队的创始成员）写的“A Simple Technique for Handling Multiple Polymorphism”以及 Kurt J. Hebel 和 Ralph Johnson（Johnson 因参与撰写《设计模式》一书而

出名)写的“Arithmetic and Double Dispatching in Smalltalk-80”。这两篇论文深入分析了动态类型语言(例如 Smalltalk、Python 和 Ruby)的多态。Python 没有使用这两篇论文中所述的双重分派来处理运算符。Python 使用的正向运算符和反向运算符更便于用户定义,类支持双重分派,但是这种方式需要解释器做些特殊处理。相比之下,经典的双重分派是一种通用技术,Python 和任何面向对象语言都能使用,而且不只适用于中缀运算符。其实,Ingalls、Hebel 和 Johnson 描述双重分派使用的示例完全不同。

本章开篇引用的那段话以及“杂谈”中引用的两段话均出自“The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling”一文,刊登于 *Java Report*, 5(7), July 2000 和 *C++ Report*, 12(7), July/August 2000 上。如果对编程语言设计感兴趣,请务必阅读这篇访谈。

杂谈

运算符重载的优缺点

如本章开篇引用的那段话所述,James Gosling 决定不让 Java 支持运算符重载。在那次访谈中(“The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling”),他说道:

大约 20%~30% 的人觉得运算符重载是罪恶之源。有些人对运算符的重载惹怒了很多,因为他们使用 + 做列表插入,导致生活一团糟。这类问题大都源于一个事实:世界上有成千上万个运算符,只有少数几个适合重载。因此,我们要挑选,但是有时所做的决定违背直觉。

Guido van Rossum 对运算符重载采取了一种折中方式:不放任用户随意创建像 <=> 或 :- 这样的新运算符,这样禁止了用户对运算符的异想天开,而且能让 Python 解析器保持简单。此

外，Python 还禁止重载内置类型的运算符，这个限制也能增加可读性和可预知的性能。

Gosling 接着说道：

社区中约有 10% 的人能正确地使用和真正关心运算符重载，对这些人来说，运算符重载是极其重要的。这部分人几乎专门处理数值，在这一领域中，为了符合人类的直觉，表示法特别重要，因为他们进入这一领域时，直觉中已经知道 + 的意思，他们知道“a + b”中的 a 和 b 可以是复数、矩阵或其他合理的东西。

当然，语言不支持运算符重载也有好处。我见过一种观点，认为对于系统编程，C 比 C++ 更好，因为 C++ 中的运算符重载会使耗费资源的操作显得微不足道。Go 和 Rust 这两门现代语言很成功，它们能把代码编译成二进制可执行文件。但这两门语言的选择截然不同：Go 不支持运算符重载，而 Rust 支持。

重载的运算符，如果使用得当，的确能让代码更易于阅读和编写。对现代的高级语言来说，这是个好功能。

惰性求值一瞥

如果仔细看示例 16-9 中的调用跟踪，你就会发现生成器表达式做惰性求值的证据。示例 16-20 再次给出了那些调用跟踪，不过加上了一些标号。

示例 16-20 与示例 16-9 一样

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs) ❶
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components) ❷
  File "vector_v6.py", line 329, in <genexpr>
```



```
    return Vector(a + b for a, b in pairs) ❸
TypeError: unsupported operand type(s) for +: 'float' and
'str'
```

❶ `Vector` 调用的 `components` 参数是一个生成器表达式。这一步没问题。

❷ 把 `components` 生成器表达式传给 `array` 构造函数。在这里，Python 尝试迭代生成器表达式，因此要求解第一个项，即 `a + b`。这里抛出了 `TypeError`。

❸ 异常向上冒泡，到达 `Vector` 构造函数调用，在这里报告出来。

这表明，生成器表达式不是在源码中定义它的位置而是在最后时刻才求解。

相比之下，如果像 `Vector([a + b for a, b in pairs])` 这样调用 `Vector` 构造函数，那么这里会立即抛出异常，因为列表推导式会尝试构建一个列表，作为参数传给 `Vector()` 调用。此时，根本不会触及 `Vector.__init__` 方法的主体。

第 17 章将详细讨论生成器表达式，但是我不想让示例中偶然出现的惰性求值迹象漏过去。

第四部分 控制流

- 第 17 章 迭代器、生成器和经典协程
- 第 18 章 `with`、`match` 和 `else` 块
- 第 19 章 Python 并发模型
- 第 20 章 并发执行器
- 第 21 章 异步编程

第 17 章 迭代器、生成器和经典协程

当我发现自己的程序中用到了模式，我就觉得这表明某个地方出错了。程序的形式应该仅仅反映它所解决的问题。代码中其他任何外加的形式都是一个信号，（至少对我来说）表明我对问题的抽象还不够深入，也经常提醒我，自己正在手动完成的事情，本应该写代码通过宏的扩展自动实现。

——Paul Graham
Lisp 黑客和风险投资人¹

¹摘自博客文章“Revenge of the Nerds”。这篇文章收录在 Paul Graham 个人文集《黑客与画家》中，详见 ituring.cn/book/3019。——编者注

迭代是数据处理的基石：程序将计算应用于数据序列，从像素到核苷酸。如果数据在内存中放不下，则需要**惰性**获取数据项，即按需一次获取一项。这就是迭代器的作用。本章说明 Python 语言是如何内置**迭代器**模式的，这样就避免了自己动手去实现。

在 Python 中，所有容器都是**可迭代对象**。Python 使用可迭代对象提供的**迭代器**支持以下操作：

- `for` 循环；
- 列表、字典和集合推导式；
- 拆包赋值；
- 构造容器实例。

本章涵盖以下话题：

- Python 如何使用内置函数 `iter()` 处理可迭代对象；
- 在 Python 中如何实现经典的迭代器模式；

- 如何使用生成器函数或生成器表达式代替经典的迭代器模式；
- 详细分析生成器函数的工作原理；
- 利用标准库中通用的生成器函数；
- 使用 `yield from` 表达式组合生成器；
- 为什么生成器和经典协程看似相同，实则差别很大，不能混淆。

17.1 本章新增内容

17.11 节的内容在第 1 版中只有 1 页，第 2 版增加到 6 页，通过更简单的实验演示 `yield from` 构造的生成器行为，还一步步开发一个示例，遍历树状数据结构。

第 2 版新增了几节，说明 `Iterable`、`Iterator` 和 `Generator` 类型提示。

17.13 节涵盖的内容在第 1 版中足足有 40 页，第 2 版缩减到 9 页²，仅简单介绍相关话题。

²本节所说的页码均指英文版页码。——编者注

我们先来研究内置函数 `iter()` 如何把序列变得可以迭代。

17.2 单词序列

我们将实现一个 `Sentence` 类，以此开启探索可迭代对象的旅程。向这个类的构造函数传入包含一些文本的字符串，然后便可以逐个单词迭代。`Sentence` 类的第 1 版要实现序列协议，这个类的对象可以迭代，因为所有序列都可迭代——这一点第 1 章已经说过，不过现在要具体说明为什么。

示例 17-1 定义的 `Sentence` 类通过索引从文本中提取单词。

示例 17-1 sentence.py: 把句子拆分成单词序列

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text) ❶

    def __getitem__(self, index):
        return self.words[index] ❷

    def __len__(self): ❸
        return len(self.words)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text) ❹
```

❶ `.findall` 函数返回一个字符串列表，里面的元素是正则表达式的全部非重叠匹配。

❷ `self.words` 存储 `.findall` 返回的结果，因此直接返回指定索引位上的单词。

❸ 为了完善序列协议，我们实现了 `__len__` 方法。不过，为了让对象可以迭代，没必要实现这个方法。

❹ `reprlib.repr` 这个实用函数用于生成大型数据结构的简略字符串表示形式。³

³首次使用 `reprlib` 模块是在 12.3 节。

默认情况下，`reprlib.repr` 函数生成的字符串最多有 30 个字符。`Sentence` 类的用法见示例 17-2 中的控制台会话。

示例 17-2 测试 Sentence 实例能否迭代

```
>>> s = Sentence('"The time has come," the Walrus said,') ❶
>>> s
Sentence('"The time ha... Walrus said,') ❷
>>> for word in s: ❸
...     print(word)
The
time
has
come
the
Walrus
said
>>> list(s) ❹
['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

❶ 传入一个字符串，创建一个 Sentence 实例。

❷ 注意，`__repr__` 方法的输出中包含 `reprlib.repr` 函数生成的
....。

❸ Sentence 实例可以迭代，稍后说明原因。

❹ 因为可以迭代，所以 Sentence 对象可用于构建列表和其他可迭代类型。

在接下来的内容中，我们还将开发其他几版 Sentence 类，而且都能通过示例 17-2 中的测试。不过，示例 17-1 中的实现与其他实现都不同，因为这一版 Sentence 类也是序列，可以按索引获取单词。

```
>>> s[0]
'The'
>>> s[5]
'Walrus'
>>> s[-1]
'said'
```

Python 程序员都知道，序列可以迭代。下面具体说明为什么。

17.3 序列可以迭代的原因：iter 函数

需要迭代对象 `x` 时，Python 自动调用 `iter(x)`。

内置函数 `iter` 执行以下操作。

01. 检查对象是否实现了 `__iter__` 方法，如果实现了就调用它，获取一个迭代器。
02. 如果没有实现 `__iter__` 方法，但是实现了 `__getitem__` 方法，那么 `iter()` 创建一个迭代器，尝试按索引（从 0 开始）获取项。
03. 如果尝试失败，则 Python 抛出 `TypeError` 异常，通常会提示“`'C' object is not iterable`”（`C` 对象不可迭代），其中 `C` 是目标对象所属的类。

所有 Python 序列都可迭代的原因是，按照定义，序列都实现了 `__getitem__` 方法。其实，标准的序列也都实现了 `__iter__` 方法，因此你也应该这么做。之所以能通过 `__getitem__` 方法迭代，是为了向后兼容，而未来可能不会再这么做——可是，Python 3.10 还未废弃这个行为，我怀疑根本就不会移除。

13.4.1 节提到过，这是鸭子类型的极端形式：不仅实现了特殊方法 `__iter__` 的对象被视作可迭代对象，实现了 `__getitem__` 方法的对象也被视作可迭代对象，如下所示。

```
>>> class Spam:
...     def __getitem__(self, i):
...         print('->', i)
...         raise IndexError()
...
>>> spam_can = Spam()
>>> iter(spam_can)
<iterator object at 0x10a878f70>
>>> list(spam_can)
-> 0
[]
>>> from collections import abc
```

```
>>> isinstance(spam_can, abc.Iterable)
False
```

如果一个类提供了 `__getitem__` 方法，那么内置函数 `iter()` 接受的迭代对象就可以是该类的实例，据此构建一个迭代器。Python 的迭代机制以从 0 开始的索引调用 `__getitem__` 方法，没有剩余项时抛出 `IndexError`。

注意，虽然 `spam_can` 可以迭代（它的 `__getitem__` 方法可以提供项），但是无法通过针对 `abc.Iterable` 的 `isinstance` 检查。

在大鹅类型理论中，可迭代对象的定义简单一些，不过没那么灵活：如果实现了 `__iter__` 方法，那就认为对象是可迭代的。此时，不需要子类化，也不用注册，因为 `abc.Iterable` 实现了 `__subclasshook__`（详见 13.5.8 节）。下面举个例子。

```
>>> class GooseSpam:
...     def __iter__(self):
...         pass
...
>>> from collections import abc
>>> issubclass(GooseSpam, abc.Iterable)
True
>>> goose_spam_can = GooseSpam()
>>> isinstance(goose_spam_can, abc.Iterable)
True
```



从 Python 3.10 开始，检查对象 `x` 能否迭代，最准确的方法是调用 `iter(x)` 函数，如果不可迭代，则处理 `TypeError` 异常。这比使用 `isinstance(x, abc.Iterable)` 更准确，因为 `iter(x)` 函数还会考虑过时的 `__getitem__` 方法，而抽象基类 `Iterable` 则不考虑。

迭代对象之前显式检查对象是否可迭代或许没必要，毕竟尝试迭代不可迭代的对象时，Python 抛出的异常信息很明确：`TypeError: 'C' object is not iterable`。如果除了抛出 `TypeError` 异

常之外还要做进一步处理，可以使用 `try/except` 块，不要显式检查。如果要保存对象，等以后再迭代，或许可以显式检查，因为这种情况下尽早捕获错误有利于调试。

内置函数 `iter()` 更常被 Python 自身使用，我们自己很少用到。`iter()` 函数还有另一种用法，不过鲜为人知。

使用 `iter` 处理可调对象

调用 `iter()` 时，传入两个参数可以为函数或任何可迭代对象创建迭代器。对于这种用法，第一个参数必须是一个可迭代对象，重复调用（不传入参数）产生值；第二个参数是**哨符**，即一种标记值，如果可调用对象返回哨符，则迭代器抛出 `StopIteration`，而不产出哨符。

下述示例使用 `iter` 函数掷一个 6 面骰子，直到点数为 1。

```
>>> def d6():
...     return randint(1, 6)
...
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x10a245270>
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

注意，这里的 `iter` 函数返回一个 `callable_iterator`。这个示例中的 `for` 循环可能会运行很长时间，但是绝不显示 1，因为 1 是哨符。与常规的迭代器一样，这个示例中的 `d6_iter` 对象一旦耗尽就没用了。如果想重来一次，则必须再次调用 `iter()` 函数重新构建迭代器。

`iter()` 函数的文档中有以下说明和示例代码。

`iter()` 的第二种形式可用于构建按块读取工具。例如，从一个二进制数据库文件中读取固定宽度的块，直至文件末尾。

```
from functools import partial

with open('mydata.db', 'rb') as f:
    read64 = partial(f.read, 64)
    for block in iter(read64, b''):
        process_block(block)
```

为了表达清楚，我在原示例的基础上增加了 `read64` 赋值。`partial()` 函数不可省略，因为提供给 `iter()` 的可调用对象必须不接受参数。在这个示例中，哨符是一个空 `bytes` 对象，这就是无字节可读时 `f.read` 返回的对象。

17.4 节详述可迭代对象与迭代器之间的关系。

17.4 可迭代对象与迭代器

由 17.3 节的说明可以推知下述定义。

可迭代对象

使用内置函数 `iter` 可以获取迭代器的对象。如果对象实现了能返回**迭代器**的 `__iter__` 方法，那么对象就是可迭代的。序列都可以迭代。实现了 `__getitem__` 方法，而且接受从 0 开始的索引，这种对象也可以迭代。

我们要明确可迭代对象与迭代器之间的关系：Python 从可迭代对象中获取迭代器。

下面是一个简单的 `for` 循环，迭代一个字符串。这里，字符串 `'ABC'` 是可迭代对象。表面上看不出来，但是背后有一个迭代器。

```
>>> s = 'ABC'
>>> for char in s:
```

```
...     print(char)
...
A
B
C
```

假如没有 `for` 语句，我们就不得不使用 `while` 循环模拟，要像下面这样写。

```
>>> s = 'ABC'
>>> it = iter(s) ❶
>>> while True:
...     try:
...         print(next(it)) ❷
...     except StopIteration: ❸
...         del it ❹
...         break ❺
...
A
B
C
```

- ❶ 根据可迭代对象构建迭代器 `it`。
- ❷ 不断在迭代器上调用 `next` 函数，获取下一项。
- ❸ 没有剩余项时，迭代器抛出 `StopIteration`。
- ❹ 释放对 `it` 的引用，即废弃迭代器对象。
- ❺ 退出循环。

`StopIteration` 表明迭代器已耗尽。内置函数 `iter()` 在内部自行处理 `for` 循环和其他迭代上下文（例如列表推导式、可迭代对象拆包等）中的 `StopIteration` 异常。

Python 标准的迭代器接口有以下两个方法。

`__next__`

返回序列中的下一项，如果没有项了，则抛出 `StopIteration`。

`__iter__`

返回 `self`，以便在预期可迭代对象的地方使用迭代器，例如 `for` 循环中。

这个接口由抽象基类 `collections.abc.Iterator` 确立。这个抽象基类定义了抽象方法 `__next__`，还从 `Iterable` 类继承了抽象方法 `__iter__`。详见图 17-1。

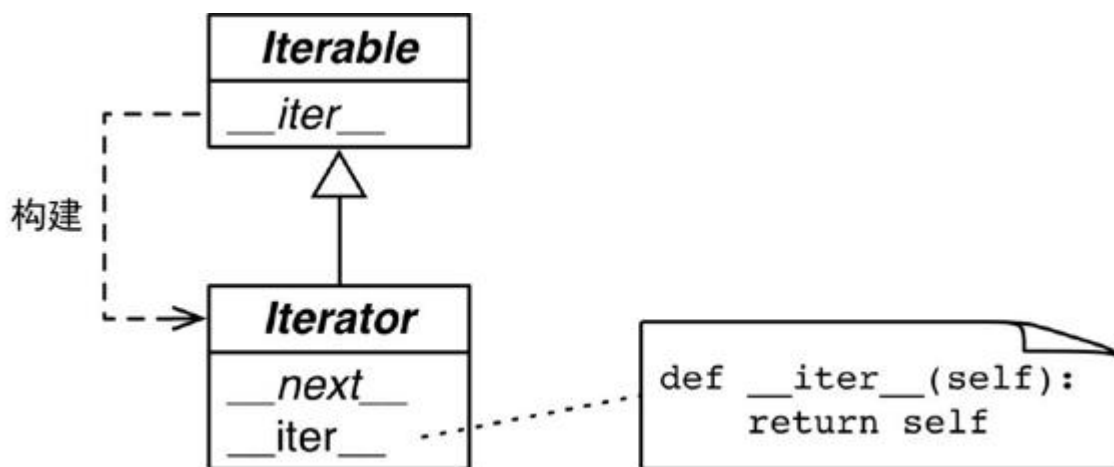


图 17-1：抽象基类 `Iterable` 和 `Iterator`。以斜体显示的是抽象方法。具体的 `Iterable.__iter__` 方法应该返回一个新的 `Iterator` 实例。具体的 `Iterator` 类必须实现 `__next__` 方法。`Iterator.__iter__` 方法直接返回实例本身

示例 17-3 是 `collections.abc.Iterator` 的源码。

示例 17-3 `abc.Iterator` 类，摘自 `Lib/_collections_abc.py`

```
class Iterator(Iterable):
    __slots__ = ()
    @abstractmethod
```

```

    def __next__(self):
        'Return the next item from the iterator. When exhausted,
raise
StopIteration'
        raise StopIteration

    def __iter__(self):
        return self

    @classmethod
    def __subclasshook__(cls, C): ❶
        if cls is Iterator:
            return _check_methods(C, '__iter__', '__next__') ❷
        return NotImplemented

```

❶ `__subclasshook__` 为 `isinstance` 和 `issubclass` 所做的结构类型检查提供支持，详见 13.5.8 节。

❷ `__check_methods` 遍历类的 `__mro__` 属性，检查基类有没有实现指定的方法。该函数也在 `Lib/_collections_abc.py` 模块中定义。如果实现了指定的方法，那么 `C` 类将被识别为 `Iterator` 的虚拟子类。也就是说，`issubclass(C, Iterable)` 返回 `True`。



在 Python 3 中，抽象基类 `Iterator` 定义的抽象方法是 `it.__next__()`，而在 Python 2 中是 `it.next()`。一如既往，我们应该避免直接调用特殊方法，使用 `next(it)` 即可。这个内置函数在 Python 2 和 Python 3 中都能完成既定任务，这对从 Python 2 到 Python 3 迁移基准代码的人有一定用处。

在 Python 3.9 中，`Lib/types.py` 模块的源码里有下面这段注释。

```

# Iterators in Python aren't a matter of type but of protocol. A
large
# and changing number of builtin types implement *some* flavor of
# iterator. Don't check the type! Use hasattr to check for both
# "__iter__" and "__next__" attributes instead.

```

其实，这就是抽象基类 `abc.Iterator` 中 `__subclasshook__` 方法的作用。



根据 `Lib/types.py` 中的建议，以及 `Lib/_collections_abc.py` 中的实现逻辑，检查对象 `x` 是否为迭代器，最好的方式是调用 `isinstance(x, abc.Iterator)`。得益于 `Iterator.__subclasshook__` 方法，即使对象 `x` 所属的类不是 `Iterator` 类的真实子类或虚拟子类，也能这样检查。

回到示例 17-1 中定义的 `Sentence` 类，在 Python 控制台中能清楚地看出 `iter()` 是如何构建迭代器的，以及 `next()` 是如何使用迭代器的。

```
>>> s3 = Sentence('Life of Brian') ❶
>>> it = iter(s3) ❷
>>> it # doctest: +ELLIPSIS
<iterator object at 0x...>
>>> next(it) ❸
'Life'
>>> next(it)
'of'
>>> next(it)
'Brian'
>>> next(it) ❹
Traceback (most recent call last):
...
StopIteration
>>> list(it) ❺
[]
>>> list(iter(s3)) ❻
['Life', 'of', 'Brian']
```

- ❶ 创建一个 `Sentence` 实例 `s3`，包含 3 个单词。
- ❷ 从 `s3` 中获取迭代器。
- ❸ 调用 `next(it)`，获取下一个单词。
- ❹ 没有单词了，因此迭代器抛出 `StopIteration` 异常。
- ❺ 耗尽后，迭代器始终抛出 `StopIteration`，表明内部已空。

⑥ 如果想再次迭代，则必须重新构建迭代器。

因为迭代器只需 `__next__` 和 `__iter__` 两个方法，所以除了调用 `next()` 并捕获 `StopIteration` 异常之外，没有其它办法检查是否还有剩余项。此外，也没有办法“重置”迭代器。如果想再次迭代，那就要调用 `iter()`，传入之前构建迭代器的可迭代对象。传入迭代器本身也没用，因为前面说过，`Iterator.__iter__` 方法的实现方式是返回 `self`，所以无法重置已经耗尽的迭代器。

这种简化的接口是合理的，因为现实中不是所有迭代器都可以重置。例如，从网络中读取分组的迭代器就不能重置。⁴

⁴感谢技术审校 Leonardo Rochael 提供这个精妙的例子。

得益于内置函数 `iter()` 对序列的特殊处理，示例 17-1 中的第 1 版 `Sentence` 类可以迭代。接下来将定义另一版 `Sentence` 类，实现 `__iter__` 方法，返回迭代器。

17.5 为 `Sentence` 类实现 `__iter__` 方法

下面几版 `Sentence` 类实现标准的可迭代对象接口。先通过迭代器模式实现，然后利用生成器函数实现。

17.5.1 `Sentence` 类第 2 版：经典迭代器

本节实现的 `Sentence` 类根据《设计模式》一书给出的模型，实现经典迭代器设计模式。注意，这不符合 Python 的习惯做法，后面重构时会说明原因。不过，通过这一版能明确可迭代的容器与迭代器之间的关系。

示例 17-4 中定义的 `Sentence` 类可以迭代，因为它实现了特殊方法 `__iter__`，构建并返回一个 `SentenceIterator` 实例。可迭代对象和迭代器之间的关系就是这样建立的。

示例 17-4 `sentence_iter.py`: 使用迭代器模式实现 `Sentence` 类

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self): ❶
        return SentenceIterator(self.words) ❷

class SentenceIterator:

    def __init__(self, words):
        self.words = words ❸
        self.index = 0 ❹

    def __next__(self):
        try:
            word = self.words[self.index] ❺
        except IndexError:
            raise StopIteration() ❻
        self.index += 1 ❼
        return word ❽

    def __iter__(self): ❾
        return self
```


- ❶ 与前一版相比，这里只多了一个 `__iter__` 方法。这一版没有 `__getitem__` 方法，为的是明确表明这个类之所以可以迭代，是因为实现了 `__iter__` 方法。
- ❷ 根据可迭代协议，`__iter__` 方法实例化并返回一个迭代器。
- ❸ `SentenceIterator` 存储一个单词列表引用。
- ❹ `self.index` 确定下一个要获取的单词。
- ❺ 获取 `self.index` 索引位上的单词。
- ❻ 如果 `self.index` 索引位上没有单词，则抛出 `StopIteration` 异常。
- ❼ 递增 `self.index` 的值。
- ❽ 返回单词。
- ❾ 实现 `self.__iter__` 方法。

示例 17-4 中的代码能通过示例 17-2 中的测试。

注意，对这个示例来说，其实没必要在 `SentenceIterator` 类中实现 `__iter__` 方法，不过这么做是对的，因为迭代器应该实现 `__next__` 和 `__iter__` 两个方法，而且这么做能让迭代器通过 `issubclass(SentenceIterator, abc.Iterator)` 测试。如果让 `SentenceIterator` 子类化 `abc.Iterator`，那么它会继承具体方法 `abc.Iterator.__iter__`。

这一版的工作量很大（对习惯了便利的 Python 程序员来说确实如此）。注意，`SentenceIterator` 类的大多数代码在处理迭代器的内部状态，稍后会说明如何简化。不过，在此之前我们先稍微离题，讨论一个看似合理实则错误的实现捷径。

17.5.2 不要把可迭代对象变成迭代器

构建可迭代对象和迭代器时经常会出现错误，原因是混淆了二者。要知道，可迭代对象有个 `__iter__` 方法，每次都实例化一个新迭代器；而迭代器要实现 `__next__` 方法，返回单个元素，此外还要实现 `__iter__` 方法，返回迭代器本身。

因此，迭代器也是可迭代对象，但是可迭代对象不是迭代器。

除了 `__iter__` 方法之外，你可能还想在 `Sentence` 类中实现 `__next__` 方法，让 `Sentence` 实例既是可迭代对象，也是自身的迭代器。可是，这种想法非常糟糕。Alex Martelli 在 Google 有大量 Python 代码审查经验，他指出，这也是常见的反模式。

《设计模式》一书讲解迭代器设计模式时，在“适用性”一节中说：

迭代器模式可用来：

- 访问一个聚合对象的内容而无须暴露它的内部表示；
- 支持对聚合对象的多种遍历；
- 为遍历不同的聚合结构提供一个统一的接口（即支持多态迭代）。

为了“支持多种遍历”，必须能从同一个可迭代对象中获取多个独立的迭代器，而且各个迭代器要能维护自身的内部状态。因此这一模式正确的实现方式是，每次调用 `iter(my_iterable)` 都新建一个独立的迭代器。这就是本例需要 `SentenceIterator` 类的原因。

至此，我们演示了如何正确实现经典迭代器模式。现在，把本节内容忘掉。Python 从 Barbara Liskov 发明的 CLU 语言中借鉴了 `yield` 关键字，因此我们无须自己动手编写实现迭代器的代码。

17.5.3 节将展示如何使用更地道的方式实现 `Sentence` 类。

17.5.3 Sentence 类第 3 版：生成器函数

实现相同功能，但符合 Python 习惯的方式是，用生成器代替 `SentenceIterator` 类。先看示例 17-5，然后详细说明生成器。

示例 17-5 sentence_gen.py：使用生成器实现 `Sentence` 类

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for word in self.words: ❶
            yield word ❷

        ❸

# 完成! ❹
```

❶ 迭代 `self.words`。

❷ 产出当前的 `word`。

❸ 无须 `return` 语句；这个函数可以直接“落空”，自动返回。不管有没有 `return` 语句，生成器函数都不抛出 `StopIteration` 异常，而是在全部值生成完毕后直接退出。⁵

⁵Alex Martelli 审查这段代码时建议简化这个方法的主体，直接使用 `return iter(self.words)`。当然，他是对的，毕竟调用 `self.words.__iter__()` 得到的就是迭代器。不过，这里我用的是 `for` 循环，而且用到了 `yield` 关键字，这样做是为了介绍

生成器函数的句法（详见 17.5.4 节）。Leonardo Rochaël 在审查本书第 2 版时，提出了简化 `__iter__` 方法主体的另一种方式：`yield from self.words`。本章后面也会介绍 `yield from`。

④ 不用再单独定义一个迭代器类！

我们又使用一种不同的方式实现了 `Sentence` 类，而且也能通过示例 17-2 中的测试。

在示例 17-4 定义的 `Sentence` 类中，`__iter__` 方法调用 `SentenceIterator` 构造函数创建一个迭代器并将其返回。而在示例 17-5 中，迭代器其实是生成器对象，在调用 `__iter__` 方法时自动创建，因为这里的 `__iter__` 方法是生成器函数。

接下来全面说明生成器的工作原理。

17.5.4 生成器的工作原理

只要 Python 函数的主体中有 `yield` 关键字，该函数就是生成器函数。调用生成器函数，返回一个生成器对象。也就是说，生成器函数是生成器工厂。



普通函数与生成器函数在句法上唯一的区别是，后者的主体中有 `yield` 关键字。有些人认为定义生成器函数应该使用一个新的关键字，例如 `gen`，而不该继续使用 `def`，但是 Guido 不同意。他的理由参见“PEP 255—Simple Generators”。⁶

⁶有时，我会在生成器函数的名称中加上 `gen` 前缀或后缀，不过这不是通用做法。显然，如果实现的是迭代器，那就不能这么做，因为所需的特殊方法必须命名为 `__iter__`。

示例 17-6 以一个简单的函数说明生成器的行为。⁷

⁷感谢 David Kwast 建议使用这个示例。

示例 17-6 一个产出 3 个数值的生成器函数

```
>>> def gen_123():
...     yield 1 ❶
...     yield 2
...     yield 3
...
>>> gen_123  # doctest: +ELLIPSIS
<function gen_123 at 0x...> ❷
>>> gen_123()  # doctest: +ELLIPSIS
<generator object gen_123 at 0x...> ❸
>>> for i in gen_123(): ❹
...     print(i)
1
2
3
>>> g = gen_123() ❺
>>> next(g) ❻
1
>>> next(g)
2
>>> next(g)
3
>>> next(g) ❼
Traceback (most recent call last):
...
StopIteration
```

❶ 生成器函数的主体中通常多次使用 `yield`，不过这不是必要条件。这里我重复 3 次使用 `yield`。

❷ 仔细看，`gen_123` 是函数对象。

❸ 但是调用时，`gen_123()` 返回一个生成器对象。

❹ 生成器对象实现了 `Iterator` 接口，因此生成器对象也可以迭代。

❺ 为了仔细检查，我们把生成器对象赋值给 `g`。

❻ 因为 `g` 是迭代器，所以调用 `next(g)` 会获取 `yield` 产出的下一项。

⑦ 生成器函数返回后，生成器对象抛出 `StopIteration` 异常。

生成器函数创建一个生成器对象，包装生成器函数的主体。把生成器对象传给 `next()` 函数时，生成器函数提前执行函数主体中的下一个 `yield` 语句，返回产出的值，并在函数主体的当前位置暂停。最终，函数的主体返回时，Python 创建的外层生成器对象抛出 `StopIteration` 异常——这一点与 `Iterator` 协议一致。



我觉得，使用准确的词语描述从生成器中获取值的过程，有助于理解生成器。说生成器“返回”值，会让人摸不着头脑。应该这样说：函数返回值；调用生成器函数返回生成器；生成器产出值。生成器不以常规的方式“返回”值：生成器函数主体中的 `return` 语句触发生成器对象抛出 `StopIteration` 异常。如果生成器中有 `return x` 语句，则调用方能从 `StopIteration` 异常中获得 `x` 的值，但是我们往往把这个操作交给 `yield from` 句法（详见 17.13.2 节）。

示例 17-7 使用 `for` 循环更清楚地说明了生成器函数主体的执行过程。

示例 17-7 运行时打印消息的生成器函数

```
>>> def gen_AB():
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> for c in gen_AB():
...     print('-->', c)
...
start
--> A
continue
--> B
```

```
end.      ⑩  
>>>      ⑪
```

- ❶ 在 `for` 循环中，第 1 次隐式调用 `next()` 函数（标号❹）打印 `'start'`，然后停在第 1 个 `yield` 语句处，生成值 `'A'`。
- ❷ 在 `for` 循环中，第 2 次隐式调用 `next()` 函数打印 `'continue'`，然后停在第 2 个 `yield` 语句处，生成值 `'B'`。
- ❸ 第 3 次调用 `next()` 函数打印 `'end.'`，然后到达函数主体的末尾，导致生成器对象抛出 `StopIteration` 异常。
- ❹ 迭代时，`for` 机制的作用与 `g = iter(gen_AB())` 一样，用于获取生成器对象，并在每次迭代时调用 `next(g)`。
- ❺ 循环打印 `-->` 和 `next(g)` 返回的值。但是，生成器函数中的 `print` 函数输出结果之后才会看到这个输出。
- ❻ 文本 `start` 是生成器主体中 `print('start')` 输出的结果。
- ❼ 生成器主体中的 `yield 'A'` 语句产出值 `A`，提供给 `for` 循环使用，而 `A` 会赋值给变量 `c`，最终输出 `--> A`。
- ❽ 第 2 次调用 `next(g)`，继续迭代，生成器主体中的代码由 `yield 'A'` 前进到 `yield 'B'`。文本 `continue` 由生成器主体中的第 2 个 `print` 语句输出。
- ❾ `yield 'B'` 语句产出值 `B`，提供给 `for` 循环使用，而 `B` 会赋值给变量 `c`，所以循环打印出 `--> B`。
- ❿ 第 3 次调用 `next(it)`，继续迭代，前进到生成器函数的末尾。文本 `end.` 由生成器主体中的第 3 个 `print` 语句输出。

❶ 生成器函数运行到末尾，生成器对象抛出 `StopIteration` 异常。`for` 机制捕获异常，循环完美终止。

现在，希望你已经知道示例 17-5 中 `Sentence.__iter__` 方法的作用了：`__iter__` 是一个生成器函数，调用时构建一个实现了 `Iterator` 接口的生成器对象，因此不再需要 `SentenceIterator` 类。

这一版 `Sentence` 类比前一版简短多了，但是还不够惰性。如今，人们认为惰性是好的特质，至少在编程语言和 API 中是如此。惰性实现是指尽可能延后生成值。这样做能节省内存，或许还可以避免浪费 CPU 循环。

17.6 节以这种惰性方式定义 `Sentence` 类。

17.6 惰性实现版本

`Sentence` 类的最后两版使用 `re` 模块中的惰性函数实现。

17.6.1 `Sentence` 类第 4 版：惰性生成器

`Iterator` 接口在设计时考虑到了惰性：`next(my_iterator)` 一次产出一项。与“惰性”相对的是“及早”，其实，惰性求值和及早求值是编程语言理论方面的术语。

目前实现的几版 `Sentence` 类都不具有惰性，因为 `__init__` 方法及早构建好了文本中的单词列表，然后将其绑定到 `self.words` 属性上。这样就得处理整个文本，列表使用的内存量可能与文本本身一样多（或许更多，这取决于文本中有多少非单词字符）。如果用户只需迭代前几个单词，那么大多数工作属于白费力气。你可能在思索一个问题：“这个操作在 Python 中有惰性处理方式吗？”那么我告诉你，答案通常是肯定的。

`re.finditer` 函数是 `re.findall` 函数的惰性版本，返回的不是列表，而是一个生成器，按需产出 `re.MatchObject` 实例。如果匹配项较多，那么 `re.finditer` 函数能节省大量内存。我们将使用这个函数让第 4 版 `Sentence` 类变得懒惰，即只在需要时才从文本中读取下一个单词。代码如下例 17-8 所示。

示例 17-8 `sentence_gen2.py`：使用一个调用生成器函数 `re.finditer` 的生成器函数实现 `Sentence` 类

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text ❶
    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        for match in RE_WORD.finditer(self.text): ❷
            yield match.group() ❸
```

❶ 不再需要 `words` 列表。

❷ `finditer` 函数构建一个迭代器，包含 `self.text` 中匹配 `RE_WORD` 的单词，产出 `MatchObject` 实例。

❸ `match.group()` 方法从 `MatchObject` 实例中提取匹配的文本。

生成器已经极大地简化了代码，不过使用生成器表达式甚至能把代码变得更简短。

17.6.2 `Sentence` 类第 5 版：惰性生成器表达式

简单的生成器函数，如前面的 `Sentence` 类中所使用的（参见示例 17-8），可以替换成生成器表达式。列表推导式构建列表，而生成器表达式构建生成器对象。二者行为之间的比较如示例 17-9 所示。

示例 17-9 先在列表推导式中使用生成器函数 `gen_AB`，然后在生成器表达式中使用

```
>>> def gen_AB(): ❶
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()] ❷
start
continue
end.
>>> for i in res1: ❸
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB()) ❹
>>> res2
<generator object <genexpr> at 0x10063c240>
>>> for i in res2: ❺
...     print('-->', i)
...
start ❻
--> AAA
continue
--> BBB
end.
```

❶ `gen_AB` 函数与示例 17-7 中的一样。

❷ 列表推导式及早迭代 `gen_AB()` 函数返回的生成器对象产生的项，即 'A' 和 'B'。注意，下面的输出是 `start`、`continue` 和 `end.`。

❸ 这个 `for` 循环迭代列表推导式构建的 `res1` 列表。

④ 生成器表达式返回一个生成器对象，`res2`。该生成器在这里没有使用。

⑤ 仅当 `for` 循环迭代 `res2` 时，这个生成器才从 `gen_AB` 中获取项。`for` 循环每迭代一次就隐式调用一次 `next(res2)`，而它又在 `gen_AB()` 返回的生成器对象上调用 `next()`，提前执行到下一个 `yield` 语句。

⑥ 注意，`gen_AB()` 的输出与 `for` 循环中 `print` 函数的输出交替出现。

可以使用生成器表达式进一步精简 `Sentence` 类的代码，如示例 17-10 所示。

示例 17-10 `sentence_genexp.py`: 使用生成器表达式实现 `Sentence` 类

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')

class Sentence:

    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        return (match.group() for match in
                RE_WORD.finditer(self.text))
```

与示例 17-8 唯一的区别是 `__iter__` 方法，这里所用的不是生成器函数了（没有 `yield`），而是使用生成器表达式构建生成器，然后将其返回。不过，最终效果一样：`__iter__` 方法的调用方得到一个生成器对象。

生成器表达式是语法糖，完全可以替换成生成器函数，不过有时使用生成器表达式更便利。17.7 节说明生成器表达式的用途。

17.7 何时使用生成器表达式

在示例 12-16 中，为了实现 `Vector` 类，我用了几个生成器表达式，`__eq__`、`__hash__`、`__abs__`、`angle`、`angles`、`format`、`__add__` 和 `__mul__` 方法中各有一个。在这些方法中，使用列表推导式也行，不过立即返回的列表占用的内存更多。

通过示例 17-10 可知，生成器表达式是创建生成器的简洁句法，无须先定义函数再调用。不过，生成器函数更为灵活，可以使用多个语句实现复杂的逻辑，甚至可以作为协程使用（见 17.13 节）。

简单的情况下可以使用生成器表达式，因为这样扫一眼就知道代码的作用，如 `Vector` 示例所示。

根据我的经验，选择使用哪种句法很容易判断：如果生成器表达式要分成多行编写，那么我倾向于定义生成器函数，以便提高可读性。



句法提示

如果函数或构造函数只有一个参数，传入生成器表达式时不用先写一对调用函数的括号，再写一对括号围住生成器表达式，只写一对括号就行了，如示例 12-16 中 `__mul__` 方法对 `Vector` 构造函数的调用所示。复述如下。

```
def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented
```

然而，如果生成器表达式后面还有其他参数，那就必须使用括号围住，否则会抛出 `SyntaxError` 异常。

目前所见的 `Sentence` 示例利用生成器实现经典迭代器模式，即从容器中获取项。不过，生成器也可用于生成不受数据源限制的值。17.8 节将举例说明。

不过，接下来要先简单讨论一下有所交集的两个概念，即**迭代器**和**生成器**。

对比迭代器和生成器

在 Python 官方文档和基准代码中，迭代器和生成器两个术语的使用前后不一、时有变化。本书采纳以下定义。

迭代器

泛指实现了 `__next__` 方法的对象。迭代器用于生成供客户代码使用的数据，即客户代码通过 `for` 循环或其他迭代方式，或者直接在迭代器上调用 `next(it)` 驱动迭代器。不过，显式调用 `next()` 并不常见。实际上，我们在 Python 中使用的迭代器多数都是生成器。

生成器

由 Python 编译器构建的迭代器。为了创建生成器，我们不实现 `__next__` 方法，而是使用 `yield` 关键字得到生成器函数（创建生成器对象的工厂）。生成器表达式是构建生成器对象的另一种方式。生成器对象提供了 `__next__` 方法，因此生成器对象是迭代器。Python 3.5 之后，还可以使用 `async def` 声明异步生成器（详见第 21 章）。

Python 术语表最近添加了术语“生成器迭代器”（`generator iterator`），指代由生成器函数构建的生成器对象。根据“生成器表达

式”词条，生成器表达式返回“迭代器”。

但是，根据 Python 文档，两种方式返回的都是生成器对象。

```
>>> def g():
...     yield 0
...
>>> g()
<generator object g at 0x10e6fb290>
>>> ge = (c for c in 'XYZ')
>>> ge
<generator object <genexpr> at 0x10e936ce0>
>>> type(g()), type(ge)
(<class 'generator'>, <class 'generator'>)
```

17.8 一个等差数列生成器

经典迭代器模式作用很简单——遍历数据结构。不过，即便不是从容器中获取项，而是获取序列中即时生成的下一个值时，也用得到这种基于方法的标准接口。例如，内置函数 `range` 用于生成有穷整数等差数列（arithmetic progression, AP）。那么，如果想生成整数之外的等差数列，该怎么办呢？

下面我们在控制台中对稍后实现的 `ArithmeticProgression` 类做一些测试，如示例 17-11 所示。这里，构造函数的签名是 `ArithmeticProgression(begin, step[, end])`。内置函数 `range()` 的完整签名是 `range(start, stop[, step])`。我选择实现不同的签名是因为创建等差数列时必须指定公差（`step`），而末项（`end`）是可选的。我还把参数的名称由 `start/stop` 改成了 `begin/end`，以明确表明签名不同。示例 17-11 中的每个测试都调用 `list()` 函数查看生成的值。

示例 17-11 演示 `ArithmeticProgression` 类的用法

```
>>> ap = ArithmeticProgression(0, 1, 3)
>>> list(ap)
[0, 1, 2]
```

```
>>> ap = ArithmeticProgression(1, .5, 3)
>>> list(ap)
[1.0, 1.5, 2.0, 2.5]
>>> ap = ArithmeticProgression(0, 1/3, 1)
>>> list(ap)
[0.0, 0.3333333333333333, 0.6666666666666666]
>>> from fractions import Fraction
>>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
>>> list(ap)
[Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
>>> from decimal import Decimal
>>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
>>> list(ap)
[Decimal('0'), Decimal('0.1'), Decimal('0.2')]
```

注意，在得到的等差数列中，数值的类型与 `begin + step` 的类型一致。如果需要，则会根据 Python 算术运算的规则强制转换类型。在示例 17-11 中，我们得到了由 `int`、`float`、`Fraction` 和 `Decimal` 数值构成的列表。示例 17-12 给出 `ArithmeticProgression` 类的实现。

示例 17-12 `ArithmeticProgression` 类

```
class ArithmeticProgression:

    def __init__(self, begin, step, end=None):           ❶
        self.begin = begin
        self.step = step
        self.end = end # None -> 无穷数列

    def __iter__(self):
        result_type = type(self.begin + self.step)      ❷
        result = result_type(self.begin)                 ❸
        forever = self.end is None                       ❹
        index = 0
        while forever or result < self.end:               ❺
            yield result                                  ❻
            index += 1
            result = self.begin + self.step * index       ❼
```

❶ `__init__` 方法需要两个参数：`begin` 和 `step`。`end` 是可选参数，如果值是 `None`，则生成无穷数列。

❷ 获取 `self.begin` 与 `self.step` 之和的类型。例如，如果一个 是 `int` 类型，另一个是 `float` 类型，那么 `result_type` 是 `float` 类型。

❸ 这一行把 `self.begin` 赋值给 `result`，不过先强制转换成前面的加法算式得到的类型。⁸

⁸Python 2 内置了 `coerce()` 函数，不过 Python 3 没有内置。开发团队觉得没必要内置，因为算术运算符会隐式应用数值强制转换规则。所以，为了让数列的首项与其他项的类型一样，我能想到的最好的方式是，先做加法运算，然后使用计算结果的类型强制转换生成的结果。我在 Python 邮件列表中问了这个问题，Steven D'Aprano 给出了极妙的答复。

❹ 为了提高可读性，我创建了 `forever` 变量。如果 `self.end` 属性的值是 `None`，那么 `forever` 的值是 `True`，因此生成的是无穷数列。

❺ 这个循环要么一直执行下去，要么当 `result` 大于或等于 `self.end` 时结束。如果循环退出了，那么这个函数也随之退出。

❻ 生成当前的 `result` 值。

❼ 计算可能存在的下一个结果。这个值可能永远不产出，因为 `while` 循环可能会终止。

在示例 17-12 中的最后一行，我没有直接使用 `self.step` 不断增加 `result`，而是选择使用 `index` 变量，把 `self.begin` 与 `self.step` 和 `index` 的乘积相加，计算 `result` 的各个值，以此降低处理浮点数时累积效应致错的风险。简单做个实验就能看出二者的区别。

```
>>> 100 * 1.1
110.00000000000001
>>> sum(1.1 for _ in range(100))
109.99999999999982
>>> 1000 * 1.1
1100.0
```



```
>>> sum(1.1 for _ in range(1000))
1100.0000000000086
```

示例 17-12 中定义的 `ArithmeticProgression` 类能按预期那样使用。这个示例也使用生成器函数实现特殊方法 `__iter__`。然而，如果一个类只是为了构建生成器而去实现 `__iter__` 方法，那还不如直接使用生成器函数。毕竟，生成器函数是制造生成器的工厂。

示例 17-13 定义了一个名为 `aritprog_gen` 的生成器函数，作用与 `ArithmeticProgression` 类一样，只不过代码量更少。如果把 `ArithmeticProgression` 类换成 `aritprog_gen` 函数，示例 17-11 中的测试也都能通过。⁹

⁹本书代码中 17-it-generator/ 目录下有 `doctest`，以及一个 `aritprog_runner.py` 脚本，用于测试 `aritprog*.py` 脚本的所有版本。

示例 17-13 生成器函数 `aritprog_gen`

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

示例 17-13 很棒，不过请牢记，标准库中有许多现成的生成器可用。接下来使用 `itertools` 模块实现一个更简短的版本。

使用 `itertools` 模块生成等差数列

Python 3.10 中的 `itertools` 模块提供了 20 个生成器函数，结合起来使用能实现很多有趣的用法。

例如，`itertools.count` 函数返回的生成器能产出数值。如果不传入参数，那么 `itertools.count` 函数产出从 0 开始的整数数列。

不过，我们可以提供可选的 `start` 和 `step` 值，实现与 `aritprog_gen` 函数类似的效果。

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1
>>> next(gen)
1.5
>>> next(gen)
2.0
>>> next(gen)
2.5
```



`itertools.count` 函数永不停止，因此，如果调用 `list(count())`，那么 Python 会创建一个特别大的列表，占满现存的所有内存。在调用失败之前，计算机会疯狂地运转。

`itertools.takewhile` 函数则不同，它返回一个使用另一个生成器的生成器，在指定的条件求值结果为 `False` 时停止。因此，可以把这两个函数结合在一起使用，编写以下代码。

```
>>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

示例 17-14 利用 `takewhile` 和 `count` 函数，写出的代码更简短。

示例 17-14 `aritprog_v3.py`：与前面的 `aritprog_gen` 函数作用相同

```
import itertools

def aritprog_gen(begin, step, end=None):
    first = type(begin + step)(begin)
    ap_gen = itertools.count(first, step)
    if end is None:
```

```
    return ap_gen
return itertools.takewhile(lambda n: n < end, ap_gen)
```

注意，示例 17-14 中的 `aritprog_gen` 不是生成器函数，因为主体中没有 `yield` 关键字。但它与生成器函数一样，也返回一个生成器。

然而，`itertools.count` 不断累加 `step`，因此得到的浮点数数列没有示例 17-13 那么精准。

示例 17-14 想阐释的观点是，实现生成器之前要知道标准库中有什么可用，否则很可能会重新发明轮子。鉴于此，17.9 节会介绍一些现成的生成器函数。

17.9 标准库中的生成器函数

标准库提供了很多生成器，有用于逐行迭代的纯文本文件对象，还有出色的 `os.walk` 函数，在遍历目录树的过程中产出文件名，递归搜索文件系统就像 `for` 循环那样简单。

`os.walk` 生成器函数的作用令人赞叹，不过本节专注于通用的函数：参数为任意可迭代对象，返回值是生成器，产出选中的、计算出的或重新排列的项。下面几张表格概述了其中的 24 个函数，有些是内置的，有些在 `itertools` 和 `functools` 模块中。为了方便，我按照函数的功能统一分组，而不管函数是在哪里定义的。

第一组是用于筛选的生成器函数：从输入的可迭代对象中产出项的子集，而且不修改项本身。与 `takewhile` 函数一样，表 17-1 中的多数函数接受一个 `predicate` 参数。这个参数的值是一个布尔函数，接受一个参数，应用到输入中的每一项上，用于判断项是否包含在输出中。

表 17-1：用于筛选的生成器函数

模块	函数	说明
itertools	<code>compress(it, selector_it)</code>	并行处理两个可迭代对象；如果 <code>selector_it</code> 中的项为真值，那么产出 <code>it</code> 中对应的项
itertools	<code>dropwhile(predicate, it)</code>	处理 <code>it</code> ，跳过 <code>predicate</code> 的计算结果为真值的项，然后产出剩下的项（不再进一步检查）
(内置)	<code>filter(predicate, it)</code>	把 <code>it</code> 中的各个元素传给 <code>predicate</code> ，如果 <code>predicate(item)</code> 返回真值，那么产出对应的元素；如果 <code>predicate</code> 是 <code>None</code> ，那么只产出真值元素
itertools	<code>filterfalse(predicate, it)</code>	与 <code>filter</code> 函数的作用类似，不过 <code>predicate</code> 的逻辑是相反的： <code>predicate</code> 返回假值时产出对应的项
itertools	<code>islice(it, stop)</code> 或 <code>islice(it, start, stop, step=1)</code>	产出 <code>it</code> 的切片，作用类似于 <code>s[:stop]</code> 或 <code>s[start:stop:step]</code> ，不过 <code>it</code> 可以是任何可迭代对象，而且这个函数惰性执行操作
itertools	<code>takewhile(predicate, it)</code>	<code>predicate</code> 返回真值时产出对应的项，然后立即停止，不再继续检查

示例 17-15 在控制台中演示表 17-1 中各个函数的用法。

示例 17-15 演示用于筛选的生成器函数

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark'))
```

```

['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1, 0, 1, 1, 0, 1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']

```

下一组是用于映射的生成器函数：在输入的可迭代对象（`map` 和 `starmap` 函数可处理多个可迭代对象）中的各项上做计算，产出计算结果。¹⁰ 表 17-2 中的生成器针对输入的可迭代对象中的每一项产出一个结果。如果输入来自多个可迭代对象，那么第一个可迭代对象耗尽后就停止输出。

¹⁰这里所说的“映射”与字典没有关系，而与内置函数 `map` 有关。

表 17-2：用于映射的生成器函数

模块	函数	说明
<code>itertools</code>	<code>accumulate(it, [func])</code>	产出累计求和；如果提供了 <code>func</code> ，那么把前两个项传给它，然后把计算结果和下一项传给它，以此类推，产出最后结果
(内置)	<code>enumerate(iterable, start=0)</code>	产出 <code>(index, item)</code> 形式的二元组，其中 <code>index</code> 从 <code>start</code> 开始计数， <code>item</code> 则从 <code>iterable</code> 中获取

模块	函数	说明
(内置)	<code>map(func, it1, [it2, ..., itN])</code>	把 <code>it</code> 中的各项依次传给 <code>func</code> , 产出结果; 如果传入 N 个可迭代对象, 那么 <code>func</code> 必须接受 N 个参数, 而且并行处理各个可迭代对象
<code>itertools</code>	<code>starmap(func, it)</code>	把 <code>it</code> 中的各项依次传给 <code>func</code> , 产出结果; 输入的可迭代对象应该产出可迭代的项 <code>iit</code> , 然后以 <code>func(*iit)</code> 形式调用 <code>func</code>

示例 17-16 演示 `itertools.accumulate` 函数的几种用法。

示例 17-16 演示 `itertools.accumulate` 生成器函数

```
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> import itertools
>>> list(itertools.accumulate(sample)) ❶
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min)) ❷
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max)) ❸
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul)) ❹
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800] ❺
```

❶ 累计和。

❷ 累计最小值。

❸ 累计最大值。

❹ 累计积。

- ❶ 从 1 到 10，计算各个数的阶乘。

表 17-2 中其他函数的演示如示例 17-17 所示。

示例 17-17 演示用于映射的生成器函数

```
>>> list(enumerate('albatroz', 1)) ❶
[(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7,
'o'), (8, 'z')]
>>> import operator
>>> list(map(operator.mul, range(11), range(11))) ❷
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8])) ❸
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8])) ❹
[(0, 2), (1, 4), (2, 8)]
>>> import itertools
>>> list(itertools.starmap(operator.mul, enumerate('albatroz',
1))) ❺
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'oooooooo',
'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b / a,
...     enumerate(itertools.accumulate(sample), 1))) ❻
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
5.0, 4.375, 4.888888888888889, 4.5]
```

- ❶ 从 1 开始，为单词中的字母编号。
- ❷ 从 0 到 10，计算各个整数的平方。
- ❸ 计算两个可迭代对象中对应位置上的两项之积，最短的可迭代对象耗尽后停止。
- ❹ 作用等同于内置函数 `zip`。
- ❺ 根据字母所在的位置（从 1 开始），把字母重复相应的次数。
- ❻ 累计平均值。

接下来这一组生成器函数用于合并，这些函数都从输入的多个可迭代对象中产出项。chain 和 chain.from_iterable 按顺序（一个接一个）处理输入的可迭代对象，而 product、zip 和 zip_longest 并行处理输入的各个可迭代对象，详见表 17-3。

表 17-3：合并多个可迭代对象的生成器函数

模块	函数	说明
itertools	chain(it1, ..., itN)	先产出 it1 中的所有项，然后产出 it2 中的所有项，以此类推，无缝衔接
itertools	chain.from_iterable(it)	产出 it 生成的各个可迭代对象中的项，一个接一个，无缝衔接；it 产出的项也应是可迭代对象，例如元组列表
itertools	product(it1, ..., itN, repeat=1)	计算笛卡儿积：从输入的各个可迭代对象中获取项，合并成 N 元组，与嵌套的 for 循环效果一样；repeat 指明重复处理多少次输入的可迭代对象
(内置)	zip(it1, ..., itN, strict=False)	从输入的各个可迭代对象中并行获取项，产出由此构成的 N 元组，只要有一个可迭代对象耗尽，就静默停止，除非指定了 strict=True ^a
itertools	zip_longest(it1, ..., itN, fillvalue=None)	从输入的各个可迭代对象中并行获取项，产出由此构成的 N 元组，直到最长的可迭代对象耗尽才停止，空缺的值使用 fillvalue 填充

^a 仅限关键字参数 strict 是 Python 3.10 中新增的。指定 strict=True 时，如果可迭代对象的长度不同，则抛出 ValueError。为了向后兼容，默认值为 False。

示例 17-18 展示生成器函数 `itertools.chain` 和 `zip` 及其相关函数的用法。再次提醒，`zip` 函数的名称出自 `zip fastener` 或 `zipper`（拉链，与 ZIP 压缩没有关系）。“出色的 `zip` 函数”附注栏介绍过 `zip` 和 `itertools.zip_longest` 函数。

示例 17-18 演示用于合并的生成器函数

```
>>> list(itertools.chain('ABC', range(2))) ❶
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC'))) ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC'))) ❸
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5), [10, 20, 30, 40])) ❹
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5))) ❺
[('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
>>> list(itertools.zip_longest('ABC', range(5), fillvalue='?')) ❻
[('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]
```

- ❶ 调用 `chain` 函数时通常传入两个或更多个可迭代对象。
- ❷ 只传入一个可迭代对象，`chain` 函数没什么用。
- ❸ 但是，`chain.from_iterable` 函数从可迭代对象中获取每一项，然后按顺序把各项串联起来，前提是各项本身也是可迭代对象。
- ❹ `zip` 可以并行处理任意个可迭代对象，不过只要有一个可迭代对象耗尽，生成器就停止。在 Python 3.10 及以上版本中，指定 `strict=True` 参数，如果有一个可迭代对象比其他可迭代对象先耗尽，则抛出 `ValueError`。
- ❺ `itertools.zip_longest` 函数的作用与 `zip` 类似，不过输入的所有可迭代对象都会被从头到尾处理，如果需要则会填充 `None`。
- ❻ `fillvalue` 关键字参数指定填充的值。

`itertools.product` 生成器是计算笛卡儿积的惰性方式。在 2.3.3 节，我们在多个 `for` 子句中使用列表推导式计算过笛卡儿积。此外，也可以使用包含多个 `for` 子句的生成器表达式，以惰性方式计算笛卡儿积。示例 17-19 演示 `itertools.product` 函数的用法。

示例 17-19 演示 `itertools.product` 生成器函数

```
>>> list(itertools.product('ABC', range(2))) ❶
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'spades hearts diamonds clubs'.split()
>>> list(itertools.product('AK', suits)) ❷
[('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A',
'spades'), ('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K',
'clubs')]
>>> list(itertools.product('ABC')) ❸
[('A',), ('B',), ('C',)]
>>> list(itertools.product('ABC', repeat=2)) ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
>>> list(itertools.product(range(2), repeat=3))
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
(1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> rows = itertools.product('AB', range(2), repeat=2)
>>> for row in rows: print(row)
...
('A', 0, 'A', 0)
('A', 0, 'A', 1)
('A', 0, 'B', 0)
('A', 0, 'B', 1)
('A', 1, 'A', 0)
('A', 1, 'A', 1)
('A', 1, 'B', 0)
('A', 1, 'B', 1)
('B', 0, 'A', 0)
('B', 0, 'A', 1)
('B', 0, 'B', 0)
('B', 0, 'B', 1)
('B', 1, 'A', 0)
('B', 1, 'A', 1)
('B', 1, 'B', 0)
('B', 1, 'B', 1)
```

- ❶ 3 个字符的字符串与 2 个整数的区间得到的笛卡儿积是 6 个元组（因为 $3 * 2$ 等于 6）。
- ❷ 2 张牌（'AK'）与 4 种花色得到的笛卡儿积是 8 个元组。
- ❸ 传入一个可迭代对象，product 函数产出一系列一元组——不是特别有用。
- ❹ repeat=N 关键字参数告诉 product 函数重复 N 次处理输入的各个可迭代对象。

有些生成器函数会从一项中产出多个值，从而扩充输入的可迭代对象，如表 17-4 所示。

表 17-4：把输入的各项扩充成多个输出项的生成器函数

模块	函数	说明
itertools	combinations(it, out_len)	把 it 产出的 out_len 个项组合在一起，然后产出
itertools	combinations_with_replacement(it, out_len)	把 it 产出的 out_len 个项组合在一起，然后产出，包含重复项的组合
itertools	count(start=0, step=1)	从 start 开始不断产出数值，按 step 指定的步幅增加
itertools	cycle(it)	从 it 中产出各项，存储各项的副本，然后按顺序重复不断地产出整个序列

模块	函数	说明
itertools	<code>pairwise(it)</code>	返回输入的可迭代对象中连续的重叠对 ^a
itertools	<code>permutations(it, out_len=None)</code>	把 <code>out_len</code> 个 <code>it</code> 产出的项排列在一起，然后产出这些排列； <code>out_len</code> 的默认值等于 <code>len(list(it))</code>
itertools	<code>repeat(item, [times])</code>	重复不断地产出指定的项，除非 <code>times</code> 指定次数

^a `itertools.pairwise` 是 Python 3.10 新增的函数。

`itertools` 模块中的 `count` 和 `repeat` 函数返回的生成器“无中生有”，二者均不接受可迭代对象作为输入。我们在 17.8 节的“使用 `itertools` 模块生成等差数列”中见过 `itertools.count` 函数。`cycle` 生成器备份输入的可迭代对象，然后重复产出其中的项。示例 17-20 演示 `count`、`cycle`、`pairwise` 和 `repeat` 的用法。

示例 17-20 演示 `count`、`cycle`、`pairwise` 和 `repeat` 的用法

```

>>> ct = itertools.count() ❶
>>> next(ct) ❷
0
>>> next(ct), next(ct), next(ct) ❸
(1, 2, 3)
>>> list(itertools.islice(itertools.count(1, .3), 3)) ❹
[1, 1.3, 1.6]
>>> cy = itertools.cycle('ABC') ❺
>>> next(cy)
'A'
>>> list(itertools.islice(cy, 7)) ❻

```

```
['B', 'C', 'A', 'B', 'C', 'A', 'B']
>>> list(itertools.pairwise(range(7))) ❶
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]
>>> rp = itertools.repeat(7) ❷
>>> next(rp), next(rp)
(7, 7)
>>> list(itertools.repeat(8, 4)) ❸
[8, 8, 8, 8]
>>> list(map(operator.mul, range(11), itertools.repeat(5))) ❹
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

- ❶ 使用 `count` 函数构建生成器 `ct`。
- ❷ 获取 `ct` 中的第一项。
- ❸ 不能使用 `ct` 构建列表，因为 `ct` 是无穷的，所以我获取接下来的 3 项。
- ❹ 如果使用 `islice` 或 `takewhile` 函数做了限制，可以从 `count` 生成器中构建列表。
- ❺ 使用 `'ABC'` 构建一个 `cycle` 生成器，然后获取第一项，即 `'A'`。
- ❻ 只有受到 `islice` 函数的限制，才能构建列表。这里获取接下来的 7 项。
- ❼ `pairwise` 根据输入中的各项产出一个二元组，包含当前项和下一项（前提是有下一项）。
- ❽ 构建一个 `repeat` 生成器，始终产出数值 7。
- ❾ 传入 `times` 参数可以限制 `repeat` 生成器产出的元素数量，这里数值 8 产出 4 次。
- ❿ `repeat` 函数的常见用途：为 `map` 函数提供固定参数。这里提供的是乘数 5。

在 `itertools` 模块的文档中，`combinations`、`combinations_with_replacement` 和 `permutations` 生成器函数，连同 `product` 函数，称为**组合生成器**（combinatoric generator）。`itertools.product` 函数和其余的**组合**函数有紧密的联系，如示例 17-21 所示。

示例 17-21 组合生成器函数从输入的各项中产出多个值

```
>>> list(itertools.combinations('ABC', 2)) ❶
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations_with_replacement('ABC', 2)) ❷
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
>>> list(itertools.permutations('ABC', 2)) ❸
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(itertools.product('ABC', repeat=2)) ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
 ('C', 'A'), ('C', 'B'), ('C', 'C')]
```

❶ 'ABC' 中每两项 (`len()==2`) 的各种组合。在生成的元组中，项的顺序无关紧要（可以视作集合）。

❷ 'ABC' 中每两项 (`len()==2`) 的各种组合，包括重复项的组合。

❸ 'ABC' 中每两项 (`len()==2`) 的各种排列。在生成的元组中，项的顺序有重要意义。

❹ 'ABC' 和 'ABC' (`repeat=2` 的效果) 的笛卡儿积。

本节要讲的最后一组生成器函数用于产出输入的可迭代对象中的全部项，不过会以某种方式重新排列。其中有两个函数返回多个生成器，分别是 `itertools.groupby` 和 `itertools.tee`。这一组里的另一个生成器函数，内置函数 `reversed`，是本节所述的函数中唯一一个不接受可迭代对象，而只接受序列为参数的函数。这在情理之中，因为 `reversed` 函数从后向前产出元素，而这仅当序列的长度已知时

才可行。不过，这个函数按需产出各项，因此无须创建反转的副本。我把 `itertools.product` 函数划分为用于合并的生成器，列在表 17-3 中，因为那一组函数都处理多个可迭代对象，而表 17-5 中的生成器最多只接受一个可迭代对象。

表 17-5：用于重新排列元素的生成器函数

模块	函数	说明
<code>itertools</code>	<code>groupby(it, key=None)</code>	产出 <code>(key, group)</code> 形式的二元组，其中 <code>key</code> 是分组标准， <code>group</code> 是生成器，用于产出分组内的项
(内置)	<code>reversed(seq)</code>	从后向前，倒序产出 <code>seq</code> 中的项， <code>seq</code> 必须是序列，或者是实现了特殊方法 <code>reversed</code> 的对象
<code>itertools</code>	<code>tee(it, n=2)</code>	产出一个由 <code>n</code> 个生成器组成的元组，每个生成器单独产出输入的可迭代对象中的项

示例 17-22 演示 `itertools.groupby` 函数和内置函数 `reversed` 的用法。注意，`itertools.groupby` 假定输入的可迭代对象可以使用分组标准排序，即使不排序，至少也要使用指定的标准分组各项。本书技术审校 Miroslav Šedivý 提供了一个用例：按时间顺序排列 `datetime` 对象，然后使用 `groupby` 为工作日分组，一组是星期一的数据，一组是星期二的数据……然后再从下一周的星期一开始，以此类推。

示例 17-22 演示 `itertools.groupby` 函数的用法

```
>>> list(itertools.groupby('LLLLAAGGG')) ❶
[('L', <itertools._grouper object at 0x102227cc0>),
 ('A', <itertools._grouper object at 0x102227b38>),
 ('G', <itertools._grouper object at 0x102227b70>)]
>>> for char, group in itertools.groupby('LLLLAAAGG'): ❷
```

```

...     print(char, '->', list(group))
...
L -> ['L', 'L', 'L', 'L']
A -> ['A', 'A',]
G -> ['G', 'G', 'G']
>>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
...             'bat', 'dolphin', 'shark', 'lion']
>>> animals.sort(key=len) ❸
>>> animals
['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
'giraffe', 'dolphin']
>>> for length, group in itertools.groupby(animals, len): ❹
...     print(length, '->', list(group))
...
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
>>> for length, group in itertools.groupby(reversed(animals),
len): ❺
...     print(length, '->', list(group))
...
7 -> ['dolphin', 'giraffe']
5 -> ['shark', 'eagle']
4 -> ['lion', 'bear', 'duck']
3 -> ['bat', 'rat']
>>>

```

- ❶ groupby 函数产出 (key, group_generator) 形式的元组。
- ❷ 处理 groupby 函数返回的生成器要嵌套迭代，这里外层使用 for 循环，内层使用 list 构造函数。
- ❸ 按照单词长度排序 animals。
- ❹ 再次遍历 key 和 group 值对，把 key 显示出来，并把 group 展开成列表。
- ❺ 这里使用 reverse 生成器从右向左迭代 animals。

这一组里的最后一个生成器函数是 `iterator.tee`，它的行为十分独特：从输入的一个可迭代对象中产出多个生成器，每个生成器都可以

产出输入中的各项。产出的各个生成器可以单独使用，如示例 17-23 所示。

示例 17-23 `itertools.tee` 函数产出多个生成器，每个生成器都可以产出输入中的各项

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x10222abc8>, <itertools._tee object at 0x10222ac08>]
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
>>> next(g2)
'B'
>>> list(g1)
['B', 'C']
>>> list(g2)
['C']
>>> list(zip(*itertools.tee('ABC')))
[('A', 'A'), ('B', 'B'), ('C', 'C')]
```

注意，这一节的示例多次把不同的生成器函数组合在一起使用。这正是这些函数的优点：它们的参数都是生成器，返回的结果也是生成器，因此能以很多不同的方式结合在一起使用。

下面研究标准库中另一类善于处理可迭代对象的函数。

17.10 可迭代的归约函数

表 17-6 中的函数都接受一个可迭代对象，返回单个结果。这些函数叫“归约”函数、“合拢”函数或“累加”函数。其实，这里列出的每个内置函数都可以使用 `functools.reduce` 函数实现，之所以内置是因为使用它们便于解决常见的问题。`functools.reduce` 函数的详细说明参阅 12.7 节。

对于 `all` 和 `any` 函数，有一项重要的优化措施是 `functools.reduce` 函数做不到的：这两个函数会短路，即一旦确定了结果就立即停止使用迭代器。参见示例 17-24 中 `any` 函数的最后一个测试。

表 17-6：读取可迭代对象，返回单个值的内置函数

模块	函数	说明
(内置)	<code>all(it)</code>	<code>it</code> 中的所有项都为真值时返回 <code>True</code> ，否则返回 <code>False</code> ； <code>all([])</code> 返回 <code>True</code>
(内置)	<code>any(it)</code>	只要 <code>it</code> 中有为真值的项就返回 <code>True</code> ，否则返回 <code>False</code> ； <code>any([])</code> 返回 <code>False</code>
(内置)	<code>max(it, [key=,] [default=])</code>	返回 <code>it</code> 中值最大的项； ^a <code>key</code> 是排序函数，与在 <code>sorted</code> 函数中的作用一样；如果可迭代对象为空，那么返回 <code>default</code>
(内置)	<code>min(it, [key=,] [default=])</code>	返回 <code>it</code> 中值最小的项； ^b <code>key</code> 是排序函数，与在 <code>sorted</code> 函数中的作用一样；如果可迭代对象为空，那么返回 <code>default</code>
<code>functools</code>	<code>reduce(func, it, [initial])</code>	把前两项传给 <code>func</code> ，然后把结果和第三项传给 <code>func</code> ，以此类推，返回最后的结果；如果提供了 <code>initial</code> ，那就把它当作第一组值的第一项
(内置)	<code>sum(it, start=0)</code>	<code>it</code> 中所有项的总和；如果提供可选的 <code>start</code> 值，那就把它加上（计算浮点数的加法时，可以使用 <code>math.fsum</code> 函数提高精度）

^a 也可以像这样调用：`max(arg1, arg2, ..., [key=?])`，此时返回参数中的最大值。

b 也可以像这样调用: `min(arg1, arg2, ..., [key=?])`, 此时返回参数中的最小值。

`all` 和 `any` 函数的操作演示如示例 17-24 所示。

示例 17-24 把几个序列传给 `all` 和 `any` 函数得到的结果

```
>>> all([1, 2, 3])
True
>>> all([1, 0, 3])
False
>>> all([])
True
>>> any([1, 2, 3])
True
>>> any([1, 0, 3])
True
>>> any([0, 0.0])
False
>>> any([])
False
>>> g = (n for n in [0, 0.0, 7, 8])
>>> any(g) ❶
True
>>> next(g) ❷
8
```

❶ `any` 迭代 `g`, 直到 `g` 产出 7。此时, `any` 停止, 返回 `True`。

❷ 因此, 下一个值是 8。

还有一个内置函数也接受一个可迭代对象, 返回相关结果, 即 `sorted` 函数。`reversed` 是生成器函数, 而 `sorted` 不同, 它构建并返回一个新列表。毕竟, 要读取输入的可迭代对象中的每一项才能排序, 而且排序的对象是列表, 因此 `sorted` 操作完成后返回排序后的列表。我在这里提到 `sorted` 函数, 是因为它可以处理任意的可迭代对象。

当然, `sorted` 函数和归约函数只能处理最终会停止的可迭代对象。否则, 这些函数一直收集项, 永远无法返回结果。



如果你坚持读到了这里，那么你已经读完了本章最重要和最有用的内容。余下几节讨论的是生成器的高级功能，例如 `yield from` 结构和经典协程，大部分人平时见不着也用不到。

另外，还有几节讨论可迭代对象、迭代器和经典协程的类型提示。

`yield from` 句法提供了一种组合生成器的新方式，接下来的 17.11 节会说明。

17.11 `yield from`: 从子生成器中产出

Python 3.3 新增的 `yield from` 表达式句法可把一个生成器的工作委托给一个子生成器。

引入 `yield from` 之前，如果一个生成器根据另一个生成器生成的值产出值，则需要使用 `for` 循环。

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...
>>> def gen():
...     yield 1
...     for i in sub_gen():
...         yield i
...     yield 2
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
2
```

使用 `yield from` 可以达到相同的效果，如示例 17-25 所示。

示例 17-25 测试驱动 `yield from`

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...
>>> def gen():
...     yield 1
...     yield from sub_gen()
...     yield 2
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
2
```

在示例 17-25 中，`for` 循环是**客户代码**，`gen` 是**委托生成器**，`sub_gen` 是**子生成器**。注意，`yield from` 暂停 `gen`，`sub_gen` 接手，直到它耗尽。`sub_gen` 产出的值绕过 `gen`，直接传给客户代码中的 `for` 循环。在此期间，`gen` 处在暂停状态，看不到绕过它的那些值。当 `sub_gen` 耗尽后，`gen` 恢复执行。

子生成器中有 `return` 语句时，返回一个值，在委托生成器中，通过含有 `yield from` 的表达式可以捕获那个值，如示例 17-26 所示。

示例 17-26 `yield from` 获取子生成器的返回值

```
>>> def sub_gen():
...     yield 1.1
...     yield 1.2
...     return 'Done!'
...
>>> def gen():
...     yield 1
...     result = yield from sub_gen()
...     print('<--', result)
...     yield 2
```

```
...
>>> for x in gen():
...     print(x)
...
1
1.1
1.2
<-- Done!
2
```

了解 `yield from` 的基本作用之后，下面通过几个简单的示例说明它的实际用途。

17.11.1 重新实现 `chain`

通过表 17-3 可知，`itertools` 模块中有一个 `chain` 生成器，它从多个可迭代对象中产出项，先迭代第一个可迭代对象，然后迭代第二个，一直到最后一个可迭代对象。在 Python 中，可以使用嵌套的 `for` 循环来自己实现 `chain` 函数。¹¹

¹¹`chain` 和 `itertools` 模块中的多数函数是使用 C 语言编写的。

```
>>> def chain(*iterables):
...     for it in iterables:
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> r = range(3)
>>> list(chain(s, r))
['A', 'B', 'C', 0, 1, 2]
```

上述代码片段中的 `chain` 生成器依次委托各个可迭代对象 `it`，在内部 `for` 循环中驱使 `it` 产出各个值。那个内部循环可以替换成 `yield from` 表达式，如以下控制台清单所示。

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i
...
```

```
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

在这个示例中完全可以使用 `yield from`，而且代码更易阅读，不过收效甚微，更像是一种语法糖。下面再举一个复杂的例子。

17.11.2 遍历树状结构

本节编写一个脚本，使用 `yield from` 遍历一个树状结构。我们会像婴儿学步一般，逐步编写这个脚本。

这个示例处理的树状结构是 Python 的异常层次结构。不过，适当改造之后也可以遍历目录树等其他树状结构。

截至 Python 3.10，异常层次结构分为 5 层，最底层（第 0 层）是 `BaseException`。我们的第一步就是显示第 0 层。

给定一个根类，示例 17-27 中的 `tree` 生成器产出根类的名称后就停止。

示例 17-27 `tree/step0/tree.py`：产出根类的名称后停止

```
def tree(cls):
    yield cls.__name__

def display(cls):
    for cls_name in tree(cls):
        print(cls_name)

if __name__ == '__main__':
    display(BaseException)
```

示例 17-27 的输出只有一行。

```
BaseException
```

下一步到第 1 层。tree 生成器产出根类的名称以及各个直接子类的名称。为了体现层次结构，子类的名称有缩进。我们希望看到的输出如下所示。

```
$ python3 tree.py
BaseException
    Exception
    GeneratorExit
    SystemExit
    KeyboardInterrupt
```

示例 17-28 可以生成这样的输出。

示例 17-28 tree/step1/tree.py：产出根类和直接子类的名称

```
def tree(cls):
    yield cls.__name__, 0
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

- ❶ 为了缩进输出，产出类的名称及其在层次结构中的层级。
- ❷ 使用特殊方法 `__subclasses__` 获取子类列表。
- ❸ 产出子类的名称和层级 1。
- ❹ 使用 4 个空格乘以倍数 `level` 构建缩进字符串。在第 0 层，得到的是一个空字符串。

示例 17-29 对 `tree` 做了重构，把特殊的根类与子类分开，子类在 `sub_tree` 生成器中处理。在 `yield from` 处，`tree` 生成器暂停，`sub_tree` 接手，产出值。

示例 17-29 `tree/step2/tree.py`: `tree` 产出根类的名称，然后委托 `sub_tree`

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls) ❶

def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1 ❷

def display(cls):
    for cls_name, level in tree(cls): ❸
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

❶ 委托 `sub_tree` 产出子类的名称。

❷ 产出各个子类的名称和层级 1。由于 `tree` 中有 `yield from sub_tree(cls)`，因此这些值完全绕过生成器函数 `tree.....`

❸ 直接在这里被接收。

秉持婴儿学步原则，我将以我能想到的最简单方式编写到达第 2 层的代码。采用深度优先算法遍历树状结构，产出第 1 层的各个节点之后，在回到第 1 层之前，要产出各个节点在第 2 层上的子节点。这一步可以使用嵌套的 `for` 循环完成，如示例 17-30 所示。

示例 17-30 tree/step3/tree.py: sub_tree 采用深度优先算法遍历第 1 层和第 2 层

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls)

def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1
        for sub_sub_cls in sub_cls.__subclasses__():
            yield sub_sub_cls.__name__, 2

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

运行示例 17-30 中的 step3/tree.py 脚本，得到的结果如下所示。

```
$ python3 tree.py
BaseException
    Exception
        TypeError
        StopAsyncIteration
        StopIteration
        ImportError
        OSError
        EOFError
        RuntimeError
        NameError
        AttributeError
        SyntaxError
        LookupError
        ValueError
        AssertionError
        ArithmeticError
        SystemError
        ReferenceError
        MemoryError
        BufferError
```

```
Warning
GeneratorExit
SystemExit
KeyboardInterrupt
```

你或许已经明白了，不过我还是想继续坚持婴儿学步原则，再嵌套一层 `for` 循环，到达第 3 层。程序的其余部分没有变化，示例 17-31 只给出了 `sub_tree` 生成器。

示例 17-31 `tree/step4/tree.py` 中的 `sub_tree` 生成器

```
def sub_tree(cls):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, 1
        for sub_sub_cls in sub_cls.__subclasses__():
            yield sub_sub_cls.__name__, 2
            for sub_sub_sub_cls in sub_sub_cls.__subclasses__():
                yield sub_sub_sub_cls.__name__, 3
```

通过示例 17-31 可以清晰地看到处理模式：一层 `for` 循环对应一层子类。如果一层 `for` 循环产出第 N 层子类，那么下一层循环就进入第 $N+1$ 层。

17.11.1 节讲过，我们可以把驱动生成器的嵌套 `for` 循环替换成 `yield from`。现在，我们也可以这么做：让 `sub_tree` 接受一个 `level` 参数，递归地从 `sub_tree` 中产出，以传入的当前子类为新的根类，并指定层级。详见示例 17-32。

示例 17-32 `tree/step5/tree.py`：递归地从 `sub_tree` 中产出，只要内存够用

```
def tree(cls):
    yield cls.__name__, 0
    yield from sub_tree(cls, 1)

def sub_tree(cls, level):
    for sub_cls in cls.__subclasses__():
        yield sub_cls.__name__, level
        yield from sub_tree(sub_cls, level+1)
```

```
def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
        print(f'{indent}{cls_name}')

if __name__ == '__main__':
    display(BaseException)
```

示例 17-32 可以遍历任意深度的树状结构，只受 Python 的递归次数限制。Python 默认允许有 1000 个待执行的函数。

任何靠谱的递归教程都会强调，必须提供一个基准情形，以防止无限递归。基准情形是一种条件分支，通常使用 `if` 语句实现，一旦满足就返回，不再递归调用。在示例 17-32 中，`sub_tree` 没有这样一个 `if` 语句，但是 `for` 循环隐含一个条件：如果 `cls.__subclasses__()` 返回一个空列表，则循环主体不执行，因此也就不递归调用。也就是说，这里的基准情形是 `cls` 类没有子类。此时，`sub_tree` 什么也不产出，直接返回。

示例 17-32 满足要求，不过还可以使用到达第 3 层时采用的模式（示例 17-31）进一步精简：一层 `for` 循环产出第 N 层子类，那么下一层循环就进入第 $N+1$ 层。在示例 17-32 中，我们把下一层循环替换成了 `yield from`。现在，可以把 `tree` 和 `sub_tree` 合并成一个生成器。示例 17-33 是这个示例的最后一步。

示例 17-33 tree/step6/tree.py：递归调用 `tree`，传入递增的 `level` 参数

```
def tree(cls, level=0):
    yield cls.__name__, level
    for sub_cls in cls.__subclasses__():
        yield from tree(sub_cls, level+1)

def display(cls):
    for cls_name, level in tree(cls):
        indent = ' ' * 4 * level
```

```
print(f'{indent}{cls_name}')
```

```
if __name__ == '__main__':  
    display(BaseException)
```

17.11 节开头讲过，`yield from` 建立子生成器与客户代码之间的直接联系，绕过委托生成器。把生成器用作协程时，这种联系就变得十分重要，它不仅产出值，而且还利用客户代码提供的值（详见 17.13 节）。

了解 `yield from` 之后，我们换个话题，讲解可迭代对象和迭代器的类型提示。

17.12 泛化可迭代类型

Python 标准库中有很多函数接受的参数是可迭代对象。在我们编写的代码中，这些函数可以像 `zip_replace` 函数（参见示例 8-15）那样，使用 `collections.abc.Iterable` 注解（如果必须支持 Python 3.8 或之前的版本，那就使用 `typing.Iterable` 注解，详见“早期支持和弃用的容器类型”附注栏）。具体做法见示例 17-34。

示例 17-34 `replacer.py`：返回一个产生字符串元组的迭代器

```
from collections.abc import Iterable  
  
FromTo = tuple[str, str] ❶  
  
def zip_replace(text: str, changes: Iterable[FromTo]) -> str: ❷  
    for from_, to in changes:  
        text = text.replace(from_, to)  
    return text
```

❶ 定义类型别名。非必需，不过可以让下面的类型提示更易于理解。从 Python 3.10 开始，`FromTo` 应该有类型提示 `typing.TypeAlias`，明确表明这一行的意图：`FromTo: TypeAlias = tuple[str, str]`。

❷ 注解 `changes` 参数，接受一个产生 `FromTo` 形式元组的 `Iterable`。

`Iterator` 类型没有 `Iterable` 类型那么常用，但是编写方式也不难。示例 17-35 是我们熟悉的斐波纳契数列生成器，附带注解。

示例 17-35 `fibo_gen.py`: `fibonacci` 返回一个产出整数的生成器

```
from collections.abc import Iterator

def fibonacci() -> Iterator[int]:
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

注意，`Iterator` 类型用于注解含有 `yield` 关键字的生成器函数，以及我们自己动手编写的带有 `__next__` 方法的迭代器类。此外，还有一个 `collections.abc.Generator` 类型（以及已经弃用的 `typing.Generator`），用于注解生成器对象，但是把生成器当作迭代器使用时，无须多此一举。

使用 `Mypy` 检查示例 17-36，你会发现，`Iterator` 类型其实是 `Generator` 类型简化的特殊情况。

示例 17-36 `itergentype.py`: 注解迭代器的两种方式

```
from collections.abc import Iterator
from keyword import kwlist
from typing import TYPE_CHECKING

short_kw = (k for k in kwlist if len(k) < 5) ❶

if TYPE_CHECKING:
    reveal_type(short_kw) ❷

long_kw: Iterator[str] = (k for k in kwlist if len(k) >= 4) ❸
```

```
if TYPE_CHECKING: ❷
    reveal_type(long_kw)
```

- ❶ 这个生成器表达式获取字符数少于 5 的 Python 关键字。
- ❷ Mypy 推导出的类型为 `typing.Generator[builtins.str*, None, None]`。¹²

¹²截至 0.910 版，Mypy 依然使用弃用的 `typing` 类型。

- ❸ 这个生成器表达式也产出字符串，不过我明确给出了类型提示。
- ❹ 揭示类型：`typing.Iterator[builtins.str]`。

`abc.Iterator[str]` 与 `abc.Generator[str, None, None]` 相容，因此 Mypy 检查示例 17-36 没有报错。

`Iterator[T]` 是 `Generator[T, None, None]` 的简写形式，二者的意思都是“产出项的类型为 `T`，但是不利用或返回值的生成器”。能利用和返回值的生成器是协程，将在 17.13 节讨论。

17.13 经典协程



“PEP 342—Coroutines via Enhanced Generators”引入 `.send()` 方法和其他功能后，生成器可以用作协程。在 PEP 342 中，“协程”一词的意思与我在这里使用的意思一样。

然而，Python 官方文档和标准库在指代用作协程的生成器时用语却不统一。因此，我不得不加上限定语，使用“经典协程”，与新出现的“原生协程”对象区分开。

Python 3.5 发布之后，“协程”通常就是指“原生协程”。但是，PEP 342 还未废弃，经典协程最初的作用没有改变，尽管已经不受

asyncio 支持。

Python 中的经典协程不太容易理解，因为经典协程其实就是生成器，只不过以另一种方式使用。因此，我们要暂停目前的讨论，转而研究 Python 中另一个有两种用法的功能。

2.4 节讲过，tuple 实例可以用作记录，也可以用作不可变序列。用作记录时，元组的项数是固定的，每一项可以是不同类型的值。用作不可变列表时，元组的长度随意，所有项都应具有相同的类型。因此，使用类型提示注解元组有两种方式。

```
# 一个城市记录，包含名称、国家（地区）和人口：
city: tuple[str, str, int]

# 一个不可变序列，包含一系列域名：
domains: tuple[str, ...]
```

生成器也有类似的情况。生成器通常用作迭代器，但是也可以用作协程。**协程**其实就是生成器函数，通过主体中含有 yield 关键字的函数创建。自然，**协程对象**就是生成器对象。尽管 Python 生成器和协程的底层 C 语言实现是一样的，但是二者的适用情形差别很大，因此类型提示的写法也不相同。

```
# readings 遍历可以绑定产出 float 值的迭代器或生成器
readings: Iterator[float]

# sim_taxi 变量可以绑定一个表示出租车的协程，模拟离散事件
# 该变量产出事件，接收浮点数时间戳，返回仿真过程中的行程次数
sim_taxi: Generator[Event, float, int]
```

更让人不解的是，typing 模块的作者决定把那个类型命名为 Generator，事实上，它描述的 API 是用作协程的生成器对象，而生成器更常用作简单的迭代器。

typing 模块的文档像下面这样描述 Generator 的形式类型参数。

```
Generator[YieldType, SendType, ReturnType]
```

仅当把生成器用作协程时，`SendType` 才有意义。那个类型参数是 `gen.send(x)` 调用中 `x` 的类型。倘若创建生成器的目的是用作迭代器，而不是协程，则调用 `.send()` 将报错。同样，`ReturnType` 也只在注解协程时有意义，因为迭代器不像常规函数那样可以返回值。对于用作迭代器的生成器，唯一合理的操作是调用 `next(it)`，可以直接调用，也可以通过 `for` 循环和其他迭代形式间接调用。`YieldType` 是 `next(it)` 调用返回值的类型。

`Generator` 类型的类型参数与 `typing.Coroutine` 相同。

<code>Coroutine[YieldType, SendType, ReturnType]</code>

其实，`typing.Coroutine` 的文档也说过：“类型变量的型变和顺序与 `Generator` 一样。”但是，`typing.Coroutine`（已弃用）和 `collections.abc.Coroutine`（自 Python 3.9 起可用的泛型）仅用于注解原生协程，不能注解经典协程。经典协程的类型提示只能使用模糊不清的 `Generator[YieldType, SendType, ReturnType]`。

David Beazley 的几场演讲和讲习班对经典协程的分析最为全面。在他 PyCon 2009 的课程讲义中，有一张幻灯片的标题是“Keeping It Straight”，内容如下。

- 生成器生产供迭代的数据。
- 协程是数据的消费者。
- 为了保持头脑清醒，不要混淆这两个概念。
- 协程与迭代没有关系。
- 注意：虽然在协程中可以使用 `yield` 产出值，但这并不是专门为了迭代。¹³

¹³“A Curious Course on Coroutines and Concurrency”，第 33 张幻灯片，“Keeping It Straight”。

接下来看看如何使用经典协程。

17.13.1 示例：使用协程计算累计平均值

第 9 章讨论闭包时，我们分析了可用于计算累计平均值的对象：示例 9-7 定义的是一个类；示例 9-13 定义的是一个高阶函数，生成一个闭包，在多次调用之间记录 `total` 和 `count` 变量的值。示例 17-37 展示了如何使用协程实现相同的功能。¹⁴

¹⁴这个示例的灵感来自 Jacob Holm 在 Python-ideas 邮件列表中发布的一个代码片段，他发布的消息题为“Yield-From: Finalization guarantees”。在那个消息的后续回复中，又出现几个不同版本。Holm 在 003912 号消息中进一步说明了自己的想法。

示例 17-37 coroaverager.py：定义一个计算累计平均值的协程

```
from collections.abc import Generator

def averager() -> Generator[float, float, None]: ❶
    total = 0.0
    count = 0
    average = 0.0
    while True: ❷
        term = yield average ❸
        total += term
        count += 1
        average = total/count
```

❶ 这个函数返回一个生成器，该生成器产出 `float` 值，通过 `.send()` 接受 `float` 值，而且不返回有用的值。¹⁵

¹⁵其实，如果没有异常导致循环终止，那么这个函数永不返回。对于 Mypy 0.910，`None` 和 `typing.NoReturn` 都可以作为生成器的返回值类型参数。不过，那个位置也可以使用 `str`，可见 Mypy 目前还不能全面分析协程代码。

❷ 这个无限循环表明，只要客户代码不断发送值，它就会一直产出平均值。

❸ 这里的 `yield` 语句暂停执行协程，把结果发给客户，而且稍后还用于接收调用方后面发给协程的值，再次开始无限循环迭代。

使用协程的好处是，`total` 和 `count` 声明为局部变量即可，在协程暂停并等待下一次调用 `.send()` 期间，无须使用实例属性或闭包保持上下文。正是这一点吸引人们在异步编程中把回调换成协程，因为在多次激活之间，协程能保持局部状态。

示例 17-38 是使用 `averager` 协程的 `doctest`。

示例 17-38 `coroaverager.py`: 示例 17-37 中定义的累计平均值协程的 `doctest`

```
>>> coro_avg = averager() ❶
>>> next(coro_avg) ❷
0.0
>>> coro_avg.send(10) ❸
10.0
>>> coro_avg.send(30)
20.0
>>> coro_avg.send(5)
15.0
```

❶ 创建协程对象。

❷ 开始执行协程。这里产出 `average` 变量的初始值，即 0.0。

❸ 真正计算累计平均值：多次调用 `.send()` 方法，产出当前平均值。

在示例 17-38 中，调用 `next(coro_avg)` 后，协程向前执行到 `yield`，产出 `average` 变量的初始值。另外，也可以调用 `coro_avg.send(None)` 开始执行协程——这其实就是内置函数 `next()` 的作用。但是，不能发送 `None` 之外的值，因为协程只能在 `yield` 处暂停时接受发送的值。调用 `next()` 或 `.send(None)` 向前执行到第一个 `yield` 的过程叫作“预激协程”。

每次激活之后，协程在 `yield` 处暂停，等待发送值。

`coro_avg.send(10)` 那一行发送一个值，激活协程，`yield` 表达式把得到的值（10）赋给 `term` 变量。循环余下的部分更新 `total`、`count` 和 `average` 这 3 个变量的值。`while` 循环的下一次迭代产出 `average` 变量的值，协程在 `yield` 关键字处再一次暂停。

细心的读者可能迫切地想知道如何终止执行 `averager` 实例（例如 `coro_avg`），因为主体中有一个无限循环。其实，我们一般不终止生成器，因为一旦没有对生成器的有效引用，生成器就会被当作垃圾回收。如果你想显式终止协程，那就使用 `.close()` 方法，如示例 17-39 所示。

示例 17-39 `coroaverager.py`：接续示例 17-38

```
>>> coro_avg.send(20) ❶
16.25
>>> coro_avg.close() ❷
>>> coro_avg.close() ❸
>>> coro_avg.send(5) ❹
Traceback (most recent call last):
...
StopIteration
```

❶ `coro_avg` 是示例 17-38 创建的实例。

❷ `.close()` 方法在暂停的 `yield` 表达式处抛出 `GeneratorExit`。如果协程函数没有处理，则该异常终止协程。`GeneratorExit` 被包装协程的生成器对象捕获，因此我们没有看到报错。

❸ 在已经关闭的协程上调用 `.close()` 方法没有效果。

❹ 在已经关闭的协程上调用 `.send()`，抛出 `StopIteration`。

除了 `.send()` 方法，“PEP 342—Coroutines via Enhanced Generators”还为协程引入了返回一个值的方式，详见 17.13.2 节。

17.13.2 让协程返回一个值

本节再来研究一个计算平均值的协程。这一版不产出过程中得到的结果，而是最终返回一个元组，指明项数和平均值。我把代码清单分成了两部分，分别放在示例 17-40 和示例 17-41 中。

示例 17-40 coroaverager2.py：文件上半部分

```
from collections.abc import Generator
from typing import Union, NamedTuple

class Result(NamedTuple): ❶
    count: int # type: ignore ❷
    average: float

class Sentinel: ❸
    def __repr__(self):
        return f'<Sentinel>'

STOP = Sentinel() ❹

SendType = Union[float, Sentinel] ❺
```

❶ 示例 17-41 中的 `averager2` 协程返回一个 `Result` 实例。

❷ `Result` 其实是 `tuple` 的子类。`tuple` 有一个 `.count()` 方法，这里用不到。`# type: ignore` 注释阻止 `Mypy` 发出警告，提醒有一个 `count` 字段。¹⁶

¹⁶我考虑过换个名称，但是对于这个协程，`count` 是最好的名称，而且本书中相关的示例也都使用这个名称，因此，在 `Result` 中，这个字段最好不要改成其他名称。如果一味顺从静态类型检查工具，则会导致代码晦涩难懂、异常复杂，因此我会毫不犹豫地使用 `# type: ignore`，借此规避检查工具的局限性和恼人的提醒。

❸ 创建哨符的类。声明 `__repr__` 方法是为了提高字符串表示形式的可读性。

❹ 我将使用这个哨符让协程停止收集数据，并返回一个结果。

❶ 协程返回类型 `Generator` 的第二个类型参数 (`SendType`) 使用这个类型别名。

这里定义的 `SendType` 在 Python 3.10 中也能使用，但是，如果不需要支持较低版本，最好从 `typing` 中导入 `TypeAlias`，像下面这样编写。

```
SendType: TypeAlias = float | Sentinel
```

`typing.Union` 换成 `|` 之后，代码更简洁、可读性更高，我甚至不愿再创建别名，而是直接在 `averager2` 的签名中编写。

```
def averager2(verbose: bool=False) -> Generator[None, float | Sentinel, Result]:
```

下面来看这个协程的代码（见示例 17-41）。

示例 17-41 `coroaverager2.py`：返回一个结果的协程

```
def averager2(verbose: bool = False) -> Generator[None, SendType, Result]: ❶
    total = 0.0
    count = 0
    average = 0.0
    while True:
        term = yield ❷
        if verbose:
            print('received:', term)
        if isinstance(term, Sentinel): ❸
            break
        total += term ❹
        count += 1
        average = total / count
    return Result(count, average) ❺
```

❶ 这个协程产出值的类型是 `None`，因为它不产出数据。接收的数据类型是 `SendType`，最后返回一个 `Result` 元组。

- ❷ 像这样使用 `yield`，只存在于协程中，目的是消耗数据。这个 `yield` 产出 `None`，从 `.send(term)` 接收 `term`。
- ❸ 如果 `term` 是哨符，则跳出循环。正是因为有这个 `isinstance` 检查.....
- ❹ Mypy 才允许我把 `term` 加到 `total` 上，也不会通过报错来提醒我不能把一个 `float` 值加到一个可能为 `float` 或 `Sentinel` 的对象上。
- ❺ 只有把哨符发送给协程，才能执行到这一行。

下面来看如何使用这个协程。首先举一个简单的例子（见示例 17-42），不生成结果。

示例 17-42 coroaverager2.py：关闭协程

```
>>> coro_avg = averager2()
>>> next(coro_avg)
>>> coro_avg.send(10) ❶
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> coro_avg.close() ❷
```

- ❶ 注意，`averager2` 不产出在过程中得到的结果。它产出 `None`，被 Python 控制台忽略。
- ❷ 调用 `.close()`，协程停止，但不返回结果，因为协程中 `yield` 所在的那行抛出了 `GeneratorExit` 异常，所以执行不到 `return` 语句。

示例 17-43 让这个协程返回结果。

示例 17-43 coroaverager2.py：捕获 `StopIteration` 异常，返回一个 `Result`

```

>>> coro_avg = averager2()
>>> next(coro_avg)
>>> coro_avg.send(10)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> try:
...     coro_avg.send(STOP) ❶
... except StopIteration as exc:
...     result = exc.value ❷
...
>>> result ❸
Result(count=3, average=15.5)

```

❶ 发送 STOP 哨符，协程跳出循环，返回一个 Result。然后，包装协程的生成器对象抛出 StopIteration。

❷ 把 StopIteration 实例的 value 属性绑定为终止协程的 return 语句的值。

❸ 信不信由你！

从 StopIteration 异常中“偷取”协程的返回值，感觉不是标准做法。然而，“PEP 342—Coroutines via Enhanced Generators”就是这样规定的，StopIteration 异常的文档和《Python 语言参考手册》中的 6.2.9 节“yield 表达式”也是这么做的。

委托生成器可以使用 yield from 句法直接获取协程的返回值，如示例 17-44 所示。

示例 17-44 coroaverager2.py：捕获 StopIteration 异常，返回一个 Result

```

>>> def compute():
...     res = yield from averager2(True) ❶
...     print('computed:', res) ❷
...     return res ❸
...
>>> comp = compute() ❹
>>> for v in [None, 10, 20, 30, STOP]: ❺

```



```
...     try:
...         comp.send(v) ❹
...     except StopIteration as exc: ❺
...         result = exc.value
received: 10
received: 20
received: 30
received: <Sentinel>
computed: Result(count=3, average=20.0)
>>> result ❸
Result(count=3, average=20.0)
```

❶ res 获得 averager2 的返回值。yield from 机制在处理表示协程终止的 StopIteration 异常时获取返回值。把 verbose 参数设为 True，让协程打印接收到的值，方便观察操作过程。

❷ 运行这个生成器时，留意这一行的输出。

❸ 返回结果。这个结果也包含在 StopIteration 中。

❹ 创建委托协程的对象。

❺ 这个循环驱动委托的协程。

❻ 发送的第一个值是 None，用于预激协程。最后一个值是让协程停止的哨符。

❼ 捕获 StopIteration，获取 compute 的返回值。

❽ averager2 和 compute 输出完毕后，再查看 Result 实例。

以上示例虽然没有做太多事情，但是代码不易理解。使用 .send() 调用驱动协程和获取结果的过程复杂难懂，唯有 yield from 例外。但是，我们只能在委托生成器或协程中使用那个句法，而且最终，委托生成器或协程要使用复杂的代码驱动，如示例 17-44 所示。

从这些示例可以看出，直接使用协程既麻烦又让人摸不着头脑。再加上异常处理和协程的 `.throw()` 方法，情况会变得更加复杂。本书不会介绍 `.throw()` 方法，因为它与 `.send()` 方法一样，只有在自己动手驱动协程时才有用，而我不建议那样做，除非你从头开发一个基于协程的框架。

在实践中，使用协程的生产性工作需要专门的框架支持。从 Python 3.3 就出现的 `asyncio` 包就是为了支持经典协程而引入的。Python 3.5 引入原生协程之后，Python 核心开发人员正在逐步淘汰 `asyncio` 对经典协程的支持。但是，底层机制是十分相似的。`async def` 句法使得代码中的原生协程更容易被发现，这是一大进步。在原生协程内部，委托其他协程的 `yield from` 换成了 `await`。原生协程会在第 21 章详尽探讨。

本章即将结束，最后讨论一个“烧脑”话题：协程类型提示中的协变和逆变。

17.13.3 经典协程的泛化类型提示

15.7.4 节的“逆变类型”中提到过，`typing.Generator` 是标准库中为数不多的几个有逆变类型参数的类型。学习经典协程之后，现在可以分析这个泛型了。

在 Python 3.6 中，`typing.py` 模块的 `typing.Generator` 类型是像下面这样声明的。¹⁷

¹⁷从 Python 3.7 开始，`typing.Generator` 及与 `collections.abc` 中的抽象基类对应的其他类型围绕相应的抽象基类做了包装重构，因此相关的泛化参数没有出现在 `typing.py` 源码文件中。鉴于此，我在这里引用了 Python 3.6 的源码。

```
T_co = TypeVar('T_co', covariant=True)
V_co = TypeVar('V_co', covariant=True)
T_contra = TypeVar('T_contra', contravariant=True)

# 省略了很多行
```

```
class Generator(Iterator[T_co], Generic[T_co, T_contra, V_co],
               extra=_G_base):
```

根据以上声明，`Generator` 类型提示包含 3 个类型参数（前文已经见过）。

```
my_coro : Generator[YieldType, SendType, ReturnType]
```

从形式参数中的类型变量可以看出，`YieldType` 和 `ReturnType` 可以协变，`SendType` 可以逆变。为了理解其中的原因，可以把 `YieldType` 和 `ReturnType` 看作“输出”类型，描述的都是从协程对象（即用作协程的生成器对象）中获取的数据。

这两个类型参数可以协变的原因不难理解，因为预取产出浮点数的协程自然可以使用产出整数的协程。这是 `YieldType` 参数可以协变的原因，同样也适用于可以协变的 `ReturnType` 参数。

根据 15.7.4 节的“协变类型”提出的表示法，可协变的第 1 个和第 3 个参数可以使用 `:>` 符号表达，指向相同的方向。

```
float :> int
Generator[float, Any, float] :> Generator[int, Any, int]
```

`YieldType` 和 `ReturnType` 体现了 15.7.4 节的“型变经验法则”给出的第一条法则。

- 如果一个形式类型参数定义的是从对象中获取的数据类型，那么该形式类型参数可能是协变的。

另一方面，`SendType` 是“输入”参数，是协程对象 `.send(value)` 方法的 `value` 参数的类型。如果客户代码需要向协程发送浮点数，那就不能使用 `SendType` 为 `int` 的协程，因为 `float` 不是 `int` 的子类型。也就是说，`float` 与 `int` 不相容。但是，可以使用 `SendType` 为 `complex` 的协程，因为 `float` 是 `complex` 的子类型，即 `float` 与 `complex` 相容。

可逆变的第 2 个参数使用 `:>` 表示法描述更形象。

```
float :> int
Generator[Any, float, Any] <: Generator[Any, int, Any]
```

这体现了型变经验法则中的第二条。

- 如果一个形式类型参数定义的是对象初始化之后向对象中输入的数据类型，那么该形式类型参数可能是逆变的。

对型变的愉快讨论到此结束，本书最长的一章也就此落下帷幕。

17.14 本章小结

Python 语言对迭代的支持如此深入，我经常说，Python 已经融合 (grok) 了迭代器。¹⁸Python 从语义上集成迭代器模式是个很好的例证，说明设计模式在各种编程语言中使用的方式并不相同。在 Python 中，自己动手实现的经典迭代器（如示例 17-4 所示）并没有什么实际用途，只能用作教学示例。

¹⁸根据“新黑客字典” (Jargon File)， “grok”的意思是不仅学会了新知识，而且还要充分吸收知识，做到“人剑合一”。

本章编写了一个类的多个版本，用于读取内容可能很多的文件，并迭代里面的单词。我们先介绍了如何使用内置函数 `iter()` 把类似序列的对象创建为迭代器；然后定义了一个含有 `__next__()` 方法的类，实现经典迭代器；最后使用生成器多次重构 `Sentence` 类，使代码更简洁、更易读。

接下来，我们编写了一个用于生成等差数列的生成器，还说明了如何利用 `itertools` 模块简化代码。随后，概述了标准库中多数通用的生成器函数。

然后，通过 `chain` 和 `tree` 示例，以简单的生成器为背景，我们学习了 `yield from` 表达式。

最后一节重点探讨了经典协程。Python 3.5 增加原生协程之后，这个话题的重要性有所减弱。经典协程在实践中难以使用，不过它是原生协程的基础，而且 `await` 就源自 `yield from` 表达式。

我们还顺带着介绍了 `Iterable`、`Iterator` 和 `Generator` 类型的类型提示。其中，`Generator` 带有可逆变的类型参数，这是少见的实例。

17.15 延伸阅读

在《Python 语言参考手册》中，6.2.9 节从技术层面深入解释了生成器。定义生成器函数的 PEP 是“PEP 255—Simple Generators”。

`itertools` 模块的文档写得很棒，包含大量示例。虽然那个模块里的函数是用 C 语言实现的，不过文档展示了如何使用 Python 实现部分函数，这通常要利用模块里的其他函数。用法示例也很好，例如，有一个代码片段说明如何使用 `accumulate` 函数计算带利息的分期付款，得出每次要还多少钱。文档中还有一节是“`Itertools Recipes`”，说明如何使用 `itertools` 模块中的现有函数实现额外的高性能函数。

在 Python 标准库之外，推荐你了解 `More Itertools` 包。这个包提供了很多强大的生成器，而且延续 `itertools` 模块的优良传统，文档中有大量示例和一些实用秘笈。

《Python Cookbook（第 3 版）中文版》的第 4 章有 16 个经典实例涵盖迭代器和生成器话题，虽然角度不同，但都关注实际应用。有几个使用 `yield from` 的经典实例，具有一定启发性。

Sebastian Rittau (typeshed 的主要贡献者) 在 2006 年写的一篇短文中解释了迭代器为什么应该可以迭代。那篇文章题为“Java: Iterators are not Iterable”。

“PEP 380—Syntax for Delegating to a Subgenerator”中的“What's New in Python 3.3”一节通过示例说明了 `yield from` 句法。

David Beazley 是 Python 生成器和协程的终极权威。他与 Brian Jones 合著的《Python Cookbook 中文版 (第 3 版)》一书中有很多使用协程编写的经典实例。Beazley 在 PyCon 期间开设的课程兼有深度和广度, 享有盛名。首先是 PyCon US 2008 期间的“Generator Tricks for Systems Programmers”课程, 在 PyCon US 2009 期间又开设了声名远播的“A Curious Course on Coroutines and Concurrency”课程。他在蒙特利尔 PyCon 2014 期间开设了“Generators: The Final Frontier”课程。在这次课程中, 他举了更多并发的例子, 与第 21 章的话题联系更大。Dave 根本不担心学员的大脑负荷, 甚至在“The Final Frontier”课程的最后一部分用协程代替了经典访问者模式, 实现了一个算术表达式计算器。

使用协程能以多种新方式组织代码, 不过与递归和多态 (动态分派) 一样, 要花点儿时间才能习惯。James Powell 写了一篇文章, 题为“Greedy algorithm with coroutines”。他在这篇文章中使用协程重写了经典的算法。

《Effective Python: 编写高质量 Python 代码的 59 个有效方法》(Brett Slatkin 著) 一书中有一章篇幅不长, 但是内容精彩, 题为“考虑用协程来并发地运行多个函数”。该书第 2 版删掉了那一章, 不过网上还有样章, 可以在线阅读。Slatkin 给出的使用 `yield from` 驱动协程的示例是我见过最棒的。那个示例实现了 John Conway 发明的“生命游戏” (Game of Life), 使用协程管理游戏运行过程中各个细胞的状态。我重构了那个“生命游戏”示例, 把 Slatkin 书中的函数和类与测试代码分开了。我还编写了 `doctest` 形式的测试, 因此不运行脚本就能看到各个协程和类的输出。重构后的示例发布在 GitHub Gist 网站上。

杂谈

Python 中极简的迭代器接口

《设计模式》一书讲解迭代器模式时，在“实现”一节中说道：¹⁹

迭代器的最小接口由 First、Next、IsDone 和 CurrentItem 操作组成。

不过，这句话有个脚注：

甚至可以将 Next、IsDone 和 CurrentItem 并入到一个操作中，该操作前进到下一个对象并返回这个对象，如果遍历结束，那么这个操作返回一个特定的值（例如，0）标志该迭代结束。这样我们就使这个接口变得更小了。

这与 Python 的做法接近：只用一个 `__next__` 方法完成这项工作。不过，为了表明迭代结束，这个方法没有使用哨符，因为哨符可能被意外忽略，而是使用 `StopIteration` 异常。简单且正确，这正是 Python 之道。

可插拔的生成器

只要管理着大型数据集，就有可能在实践中找到机会使用生成器。下面是我第一次围绕生成器构建解决方案的真实故事。

多年前，我在 BIREME 工作，这是 PAHO/WHO（Pan-American Health Organization/ World Health Organization，泛美卫生组织 / 世界卫生组织）在巴西圣保罗运营的一家数字图书馆。BIREME 制作的众多书目数据集中包含 LILACS（Latin American and Caribbean Health Sciences index，拉美和加勒比地区健康科学索引）和 SciELO（Scientific Electronic Library Online，巴西电子科学在线图书馆），这两个数据库完整索引了这一地区发表的健康科学研究文献。

从 20 世纪 80 年代后期开始，管理 LILACS 的数据库系统是 CDS/ISIS。这是 UNESCO 开发的非关系型文档数据库。我的工作之一是探索替代方案，把 LILACS 移植到现代的开源文档数据库（最终还要移植大得多的 SciELO），例如 CouchDB 或 MongoDB。在此期间，我写了一篇论文，题为“From ISIS to CouchDB: Databases and Data Models for Bibliographic Records”，阐释了半结构化数据模型和使用 JSON 一类的记录表示 CDS/ISIS 数据的不同方式。

在探索的过程中，我编写了一个 Python 脚本，把 CDS/ISIS 文件转换成适合导入 CouchDB 或 MongoDB 的 JSON 文件。起初，这个脚本使用 CDS/ISIS 导出的 ISO-2709 格式读取文件。读写过程必须采用渐进方式，因为完整的数据集比主内存大得多。解决方法很简单，主 `for` 循环每次从 `.iso` 文件中读取一个记录，转换后写入 `.json` 文件输出。

然而，在实际操作中有必要让 `isis2json.py` 脚本支持 CDS/ISIS 的另一种数据格式，即 BIREME 在生产环境中使用的二进制 `.mst` 文件，以避免导出 ISO-2709 格式时消耗过多资源。当时，我遇到一个问题：用来读取 ISO-2709 和 `.mst` 文件的库提供的 API 差别很大。而输出 JSON 格式的循环已经很复杂了，因为这个脚本要接受多个命令行选项，用于调整输出的记录结构。在同一个 `for` 循环中使用两个不同的 API，同时还要生成 JSON，这样太难以管理了。

我采用的解决方法是隔离读取逻辑，将其写进一对生成器函数中，一个函数支持一种输入格式。最终，我把 `isis2json.py` 脚本分成了 4 个函数。这个脚本使用 Python 2 编写。²⁰

下面概述这个脚本的结构。

```
main
```


`main` 函数使用 `argparse` 模块读取命令行选项，配置输出记录的结构。根据输入文件的扩展名，`main` 函数选择一个合适的生成器函数，逐个读取并产出记录。

```
iter_iso_records
```

这个生成器函数读取 `.iso` 文件（假设是 ISO-2709 格式），接受两个参数：一个是文件名，另一个是 `isis_json_type`，即一个与记录结构有关的选项。在这个函数的 `for` 循环中，每次迭代读取一个记录，然后创建一个空字典，把数据填充进字段之后产出字典。

```
iter_mst_records
```

这也是一个生成器函数，用于读 `.mst` 文件。²¹ 查看 `isis2json.py` 脚本的源码后你会发现，这个函数不像 `iter_iso_records` 函数那么简单，不过接口和整体结构是相同的：参数是文件名和 `isis_json_type`，`for` 循环每次迭代构建并产出一个字典，表示一个记录。

```
write_json
```

这个函数把记录输出为 JSON 格式，而且一次输出一个记录。它的参数很多，其中第一个参数 (`input_gen`) 是对某个生成器函数的引用：`iter_iso_records` 或 `iter_mst_records`。`write_json` 函数中的主 `for` 循环迭代 `input_gen` 引用的生成器产出的字典，根据命令行选项设定的方式处理，然后把 JSON 格式的记录附加到输出文件中。

我利用生成器函数解耦了读逻辑和写逻辑。当然，解耦二者最简单的方法应该是把所有记录读进内存，然后写入磁盘。可是这样并不可行，因为数据集非常大。使用生成器的话，则可以交叉读写，因此这个脚本可以处理任意大小的文件。另外，从不同的输

入格式中读取记录需要不一样的逻辑，而各个独特的读逻辑也与调整记录结构的写逻辑区分开了。

现在，如果 `isis2json.py` 脚本需要再支持一种输入格式，比如，美国国会图书馆用于表示 ISO-2709 格式数据的 MARCXML 文档格式，那么只需再添加一个生成器函数来实现读逻辑，而复杂的 `write_json` 函数无须任何改动。

这不是什么尖端科技，而是一个真实的案例。通过它，我们看到了生成器的高效和灵活。使用生成器处理数据库时，我们把记录看成数据流，这样消耗的内存量最低，而且不管数据有多少都能处理。

¹⁹ 《设计模式：可复用面向对象软件的基础》第 174 页。

²⁰ 代码是用 Python 2 编写的，因为有一个可选依赖是一个 Java 库，名为 Bruma，使用 Jython 运行时可以导入。而目前 Jython 还不支持 Python 3。

²¹ 用来读取复杂的 .mst 二进制文件的库其实是用 Java 编写的，因此只有使用 Jython 解释器 2.5 或以上版本运行 `isis2json.py` 脚本才能使用这个功能。详情参见 README.rst 文件。因为依赖在需要使用的生成器函数中导入，所以即便只有一个外部依赖可用，这个脚本仍能运行。

第 18 章 with、match 和 else 块

最终，上下文管理器可能几乎与子程序（subroutine）自身一样重要。目前，我们只了解了上下文管理器的皮毛。.....Basic 等很多语言都有 with 语句。但是，在各种语言中 with 语句的作用不同，而且做的都是简单的事，虽然可以避免不断使用点号查找属性，但是不会做事前准备和事后清理。不要觉得名字一样，就意味着作用也一样。with 语句是非常了不起的功能。¹

——Raymond Hettinger
雄辩的 Python 布道者

¹节选自 PyCon US 2013 主题演讲“What Makes Python Awesome”；关于 with 的部分从 23:00 开始，到 26:15 结束。

本章讨论一些在其他语言中不常见的控制流功能，正因为不常见，所以 Python 用户往往忽视或没有充分使用这些功能。我们要讨论的功能如下：

- with 语句和上下文管理器协议；
- 匹配模式的 match/case；
- for、while 和 try 语句的 else 子句。

with 语句设置一个临时上下文，交给上下文管理器对象控制，并且负责清理上下文。这么做能避免错误并减少样板代码，因此 API 更安全、更易于使用。除了自动关闭文件之外，with 块还有很多用途。

前面已经讲过模式匹配，本章探讨如何把语言的语法表达成序列模式，并使用 match/case 创建易于理解和扩展的语言处理器。我们将研制一个完整的解释器，实现 Scheme 语言的一小部分功能。你可以

依葫芦画瓢，使用类似的技术开发模板语言或 DSL (Domain-Specific Language, 领域特定语言)，在大型系统中编码业务规则。

`else` 子句没什么神奇的，但是结合 `for`、`while` 和 `try` 使用，有助于表达意图。

18.1 本章新增内容

18.3 节是新增的。

18.2.1 节有更新，涵盖了 Python 3.6 之后 `contextlib` 模块新增的几个功能，以及 Python 3.10 新引入的带括号的上下文管理器句法。

先从强大的 `with` 语句讲起。

18.2 上下文管理器和 `with` 块

上下文管理器对象存在的目的是管理 `with` 语句，就像迭代器的存在是为了管理 `for` 语句一样。

`with` 语句的目的是简化一些常用的 `try/finally` 结构。这种结构可以保证一段代码运行完毕后执行某项操作，即便那段代码由于 `return` 语句、异常或 `sys.exit()` 调用而终止，也执行指定的操作。`finally` 子句中的代码通常用于释放重要的资源，或者还原临时改变的状态。

Python 社区为上下文管理器找到了新颖的用法，颇具创造性。下面是标准库中的一些示例。

- 在 `sqlite3` 模块中管理事务。见文档中的“Using the connection as a context manager”一节。
- 安全地处理锁、条件和信号量。详见 `threading` 模块文档。

- 为 Decimal 对象的算术运算定制环境。见 decimal.localcontext 文档。
- 为了测试给对象打补丁。见 unittest.mock.patch 函数文档。

上下文管理器接口包含 `__enter__` 和 `__exit__` 两个方法。with 语句开始运行时，Python 在上下文管理器对象上调用 `__enter__` 方法。with 块运行结束，或者由于什么原因终止后，Python 在上下文管理器对象上调用 `__exit__` 方法。

最常见的例子是确保关闭文件对象。示例 18-1 具体演示如何使用 with 语句关闭文件。

示例 18-1 把文件对象当成上下文管理器使用

```
>>> with open('mirror.py') as fp: ❶
...     src = fp.read(60) ❷
...
>>> len(src)
60
>>> fp ❸
<_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
>>> fp.closed, fp.encoding ❹
(True, 'UTF-8')
>>> fp.read(60) ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

- ❶ fp 绑定打开的文件，因为文件的 `__enter__` 方法返回 self。
- ❷ 从 fp 中读取 60 个 Unicode 字符。
- ❸ fp 变量仍然可用。与函数不同，with 块不定义新的作用域。
- ❹ 可以读取 fp 对象的属性。

❸ 但是不能使用 `fp` 继续读取文本，因为在 `with` 块的末尾，Python 调用 `TextIOWrapper.__exit__` 方法把文件关闭了。

示例 18-1 中标号❶那行代码道出了不易察觉但很重要的一点：求解 `with` 后面的表达式得到的结果是上下文管理器对象，不过，绑定到目标变量（在 `as` 子句中）上的值是在上下文管理器对象上调用 `__enter__` 方法返回的结果。

碰巧，`open()` 函数返回一个 `TextIOWrapper` 实例，而该实例的 `__enter__` 方法返回 `self`。不过，其他类的 `__enter__` 方法可能会返回其他对象，而不返回上下文管理器实例。

不管控制流以哪种方式退出 `with` 块，都在上下文管理器对象上调用 `__exit__` 方法，而不是在 `__enter__` 方法返回的对象上调用。

`with` 语句的 `as` 子句是可选的。对 `open` 函数来说，必须加上 `as` 子句，以便获取文件的引用，在引用上调用方法。不过，有些上下文管理器返回 `None`，因为没什么有用的对象能提供给用户。

示例 18-2 使用一个精心设计的上下文管理器执行操作，强调上下文管理器与 `__enter__` 方法返回的对象之间是有区别的。

示例 18-2 测试驱动 LookingGlass 上下文管理器类

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what: ❶
...     print('Alice, Kitty and Snowdrop') ❷
...     print(what)
...
pordwonS dna yttiK ,ecila
YKCOWREBBAJ
>>> what ❸
'JABBERWOCKY'
>>> print('Back to normal.') ❹
Back to normal.
```

❶ 这个上下文管理器是一个 LookingGlass 实例，Python 在上下文管理器上调用 `__enter__` 方法，把返回结果绑定到 `what` 上。

❷ 打印一个字符串，然后打印 `what` 变量的值。打印出的内容是反向的。

❸ 现在，`with` 块执行完毕。可以看到，`__enter__` 方法返回的值，即存储在 `what` 变量中的值，是字符串 `'JABBERWOCKY'`。

❹ 输出不再是反向的了。

示例 18-3 是 LookingGlass 类的实现。

示例 18-3 mirror.py: LookingGlass 上下文管理器类的代码

```
import sys

class LookingGlass:

    def __enter__(self): ❶
        self.original_write = sys.stdout.write ❷
        sys.stdout.write = self.reverse_write ❸
        return 'JABBERWOCKY' ❹

    def reverse_write(self, text): ❺
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback): ❻
        sys.stdout.write = self.original_write ❼
        if exc_type is ZeroDivisionError: ❽
            print('Please DO NOT divide by zero!')
            return True ❾

❿
```

❶ Python 调用 `__enter__` 方法，除 `self` 之外不传入其他参数。

❷ 把原来的 `sys.stdout.write` 方法保存起来，供后面恢复。

- ③ 为 `sys.stdout.write` 打猴子补丁，替换成自己编写的方法。
- ④ 返回 'JABBERWOCKY' 字符串，这样才有内容存入目标变量 `what`。
- ⑤ 这是取代 `sys.stdout.write` 的方法，反转 `text` 参数的内容，然后调用原来的实现。
- ⑥ 如果一切正常，那么 Python 调用 `__exit__` 方法时传入的参数是 `None, None, None`；如果抛出了异常，那么这 3 个参数是异常数据，详见下面的说明。
- ⑦ 恢复成原来的 `sys.stdout.write` 方法。
- ⑧ 如果有异常，而且是 `ZeroDivisionError` 类型，那么打印一个消息.....
- ⑨ 然后返回 `True`，告诉解释器，异常已被处理。
- ⑩ 如果 `__exit__` 方法返回 `None` 或其他假值，那么 `with` 块抛出的任何异常都会向上冒泡。



在实际使用中，如果应用接管了标准输出，则可能会暂时把 `sys.stdout` 换成类似文件的其他对象，然后再切换成原来的版本。`contextlib.redirect_stdout` 上下文管理器就能做到这一点：只需传入类似文件的对象，用于替代 `sys.stdout`。

解释器调用 `__enter__` 方法时，除了隐式的 `self` 之外，不传入任何参数。传给 `__exit__` 方法的 3 个参数列举如下。

`exc_type`

异常类（例如 `ZeroDivisionError`）。

`exc_value`

异常实例。有时会有参数传给异常构造函数，例如错误消息，这些参数可以通过 `exc_value.args` 获取。

`traceback`

traceback 对象。²

²在 `try/finally` 语句的 `finally` 块中调用 `sys.exc_info()`，得到的就是 `__exit__` 接收的这 3 个参数。鉴于 `with` 语句是为了取代多数 `try/finally` 语句，而且通常需要调用 `sys.exc_info()` 来判断需要做什么清理操作，这种行为是合理的。

上下文管理器的具体工作方式参见示例 18-4。在这个示例中，我们在 `with` 块之外使用 `LookingGlass` 类，因此可以手动调用 `__enter__` 和 `__exit__` 方法。

示例 18-4 在 `with` 块之外使用 `LookingGlass` 类

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass() ❶
>>> manager # doctest: +ELLIPSIS
<mirror.LookingGlass object at 0x...>
>>> monster = manager.__enter__() ❷
>>> monster == 'JABBERWOCKY' ❸
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager # doctest: +ELLIPSIS
>... ta tcejbo ssalGnikooL.rorrim<
>>> manager.__exit__(None, None, None) ❹
>>> monster
'JABBERWOCKY'
```

❶ 实例化并查看 `manager` 实例。

❷ 在 `manager` 上调用 `__enter__()` 方法，把结果存储在 `monster` 中。

③ `monster` 的值是字符串 `'JABBERWOCKY'`。打印出的 `True` 标识符是反向的，因为 `stdout` 的所有输出都经过 `__enter__` 方法中打补丁的 `write` 方法处理。

④ 调用 `manager.__exit__`，还原成之前的 `stdout.write`。



Python 3.10 中带括号的上下文管理器

Python 3.10 采用了新型解析器，比旧的 LL(1) 解析器更强大，可用的句法也多了。其中一个新句法是带括号的上下文管理器，如下所示。

```
with (  
    CtxManager1() as example1,  
    CtxManager2() as example2,  
    CtxManager3() as example3,  
):  
    ...
```

在 Python 3.10 之前，嵌套 `with` 块才能达到同样的效果。

标准库中的 `contextlib` 包提供了一些函数、类和装饰器，方便构建、组合和使用上下文管理器。

18.2.1 `contextlib` 包中的实用工具

自己定义上下文管理器类之前，请先看一下 Python 文档中的“`contextlib—Utilities for with-statement contexts`”。也许你想实现的功能已经提供，或者有一些类或可调对象能简化实现过程。

除了示例 18-3 后面提到的 `redirect_stdout` 上下文管理器，Python 3.5 还增加了 `redirect_stderr`，作用与前者一样，只不过它把输出发给 `stderr`。

`contextlib` 包中还有下面一些函数。

`closing`

如果对象提供了 `close()` 方法，但没有实现 `__enter__` / `__exit__` 接口，则可以使用这个函数构建上下文管理器。

`suppress`

构建临时忽略指定异常的上下文管理器。

`nullcontext`

Python 3.7 新增。一个什么也不做的上下文管理器，可以简化未实现合适上下文管理器的对象周围的条件逻辑。当你不确定 `with` 块之前的条件代码有没有为 `with` 语句提供上下文管理器时，就可以使用 `nullcontext` 代替。

`contextlib` 模块提供的几个类和一个装饰器比以上函数的使用范围更广。

`@contextmanager`

这个装饰器把简单的生成器函数变成上下文管理器，免得创建类去实现上下文管理器协议。详见 18.2.2 节。

`AbstractContextManager`

Python 3.6 新增。确立上下文管理器接口的抽象基类。子类化该基类创建上下文管理器类会更容易一些。

`ContextDecorator`

这个基类用于定义基于类的上下文管理器。这种上下文管理器也可用作函数装饰器，在受管理的上下文中运行整个函数。

`ExitStack`

这个上下文管理器能进入多个上下文管理器。`with` 块结束时，`ExitStack` 按照后进先出（LIFO）顺序调用栈中各个上下文管理器的 `__exit__` 方法。如果你事先不知道 `with` 块要进入多少个上下文管理器，则可以使用这个类。例如，同时打开任意一个文件列表中的所有文件。

Python 3.7 还为 `contextlib` 模块增加了 `AbstractAsyncContextManager`、`@asynccontextmanager` 和 `AsyncExitStack`。它们的作用与名称中不带 `async` 的版本类似，只不过供新出现的 `async with` 语句（见第 21 章）使用。

在这些实用工具中，使用最广泛的是 `@contextmanager` 装饰器，因此有必要进一步说明。这个装饰器也有迷惑人的一面，它与迭代无关，却要使用 `yield` 语句。

18.2.2 使用 `@contextmanager`

`@contextmanager` 装饰器是一个巧妙且实用的工具，将 Python 的 3 个不同功能结合在一起：函数装饰器、生成器和 `with` 语句。

使用 `@contextmanager` 能减少创建上下文管理器的样板代码，因为不用编写一个完整的类来定义 `__enter__` 和 `__exit__` 方法，而只需实现有一个含有 `yield` 语句的生成器，生成想让 `__enter__` 方法返回的值。

在使用 `@contextmanager` 装饰的生成器中，`yield` 把函数的主体分成两部分：`yield` 前面的所有代码在 `with` 块开始时（解释器调用 `__enter__` 方法时）执行，`yield` 后面的代码在 `with` 块结束时（调用 `__exit__` 方法时）执行。

示例 18-5 使用一个生成器函数代替示例 18-3 中的 `LookingGlass` 类。

示例 18-5 mirror_gen.py: 使用生成器实现上下文管理器

```
import contextlib
import sys

@contextlib.contextmanager ❶
def looking_glass():
    original_write = sys.stdout.write ❷

    def reverse_write(text): ❸
        original_write(text[::-1])

    sys.stdout.write = reverse_write ❹
    yield 'JABBERWOCKY' ❺
    sys.stdout.write = original_write ❻
```

- ❶ 应用 `contextmanager` 装饰器。
- ❷ 保留原来的 `sys.stdout.write` 方法。
- ❸ 后面的 `reverse_write` 可以调用 `original_write`, 因为 `original_write` 在闭包中。
- ❹ 把 `sys.stdout.write` 替换成 `reverse_write`。
- ❺ 产出一个值, 这个值绑定到 `with` 语句中 `as` 子句的目标变量上。这个函数在该处暂停, `with` 块的主体开始执行。
- ❻ 控制权跳出 `with` 块后, 继续执行 `yield` 之后的代码, 在这里是恢复原来的 `sys.stdout.write`。示例 18-6 演示了 `looking_glass` 函数的实际使用。

示例 18-6 测试驱动 `looking_glass` 上下文管理器函数

```
>>> from mirror_gen import looking_glass
>>> with looking_glass() as what: ❶
...     print('Alice, Kitty and Snowdrop')
...     print(what)
... 
```

```
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
>>> print('back to normal')
back to normal
```

❶ 与示例 18-2 唯一的区别是上下文管理器的名称：LookingGlass 变成了 looking_glass。

其实，`contextlib.contextmanager` 装饰器把函数包装成实现了 `__enter__` 和 `__exit__` 方法的类。³

³类的名称是 `_GeneratorContextManager`。如果想了解具体的工作方式，可以阅读 Python 3.10 中 `Lib/contextlib.py` 文件里的源码。

那个类的 `__enter__` 方法有以下作用。

01. 调用生成器函数，获取生成器对象（姑且称之为 `gen`）。
02. 调用 `next(gen)`，驱动生成器对象执行到 `yield` 关键字所在的位置。
03. 返回 `next(gen)` 产出的值，以便把产出的值绑定到 `with/as` 语句中的目标变量上。

`with` 块终止时，`__exit__` 方法做以下几件事。

01. 检查有没有把异常传给 `exc_type`；如果有，则调用 `gen.throw(exception)`，在生成器函数主体中 `yield` 关键字所在的行抛出异常。
02. 否则，调用 `next(gen)`，恢复执行生成器函数主体中 `yield` 后面的代码。

示例 18-5 有一个严重问题：如果 `with` 块有异常抛出，则 Python 解释器会将其捕获，然后在 `looking_glass` 函数的 `yield` 表达式中再次抛出。但是，那里没有处理错误的代码，因此 `looking_glass`

函数将终止执行，永远无法恢复成原来的 `sys.stdout.write` 方法，导致系统处于无效状态。

示例 18-7 添加了一些代码，专门处理 `ZeroDivisionError` 异常。现在，`looking_glass` 的功能与示例 18-3 中基于类的版本相同。

示例 18-7 `mirror_gen_exc.py`：基于生成器的上下文管理器，而且实现了异常处理——从外部看，行为与示例 18-3 相同

```
import contextlib
import sys

@contextlib.contextmanager
def looking_glass():
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = '' ❶
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError: ❷
        msg = 'Please DO NOT divide by zero!'
    finally:
        sys.stdout.write = original_write ❸
        if msg:
            print(msg) ❹
```

❶ 创建一个变量，保存可能出现的错误消息。与示例 18-5 相比，这是第一处改动。

❷ 处理 `ZeroDivisionError` 异常，设置一个错误消息。

❸ 撤销对 `sys.stdout.write` 方法所做的猴子补丁。

❹ 如果有错误消息，就把它打印出来。

前面说过，为了告诉解释器异常已经处理了，`__exit__` 方法会返回一个真值，此时解释器压制（suppress）异常。然而，如果 `__exit__` 方法没有显式返回一个值，那么解释器得到的是 `None`，此时异常向上冒泡。使用 `@contextmanager` 装饰器时，默认行为是相反的：装饰器提供的 `__exit__` 方法假定发给生成器的所有异常都得到处理了，因此应该压制异常。



使用 `@contextmanager` 装饰器时，要把 `yield` 语句放在 `try/finally` 语句中（或者放在 `with` 块中），这是不可避免的，因为我们永远不知道上下文管理器的用户会在 `with` 块中做什么。⁴

⁴这条提示直接引自 Leonardo Rochael 的评论，他是本书的技术审校之一。说得好，Leo！

`@contextmanager` 还有一个鲜为人知的功能：它装饰的生成器也可用作装饰器。⁵ 这是因为 `@contextmanager` 是由 `contextlib.ContextDecorator` 类实现的。

⁵至少我和其他技术审校不知道，是 Caleb Hattingh 告诉我们的。感谢 Caleb！

示例 18-8 把示例 18-5 中的 `looking_glass` 上下文管理器当作装饰器使用。

示例 18-8 `looking_glass` 上下文管理器也可以用作装饰器

```
>>> @looking_glass()
... def verse():
...     print('The time has come')
...
>>> verse() ❶
emoc sah emit ehT
>>> print('back to normal') ❷
back to normal
```


❶ `looking_glass` 在 `verse` 主体运行之前和之后完成了自己的任务。

❷ 确认原来的 `sys.write` 已经恢复。

请你对比一下示例 18-8 和示例 18-6 (`looking_glass` 用作上下文管理器)。

在标准库之外，Martijn Pieters 实现的就地文件重写上下文管理器也用到了 `@contextmanager`，如示例 18-9 所示。

示例 18-9 一个就地重写文件的上下文管理器

```
import csv

with inplace(csvfilename, 'r', newline='') as (infh, outfh):
    reader = csv.reader(infh)
    writer = csv.writer(outfh)

    for row in reader:
        row += ['new', 'columns']
        writer.writerow(row)
```

`inplace` 函数是一个上下文管理器，为同一个文件提供了两个句柄（这个示例中的 `infh` 和 `outfh`），以便同时读写同一个文件。这比标准库中的 `fileinput.input` 函数（顺便说一下，这个函数也提供一个上下文管理器）好用多了。

如果你想研究 Martijn 实现 `inplace` 的源码（在他写的文章“Easy inplace file rewriting”中给出了），找到 `yield` 关键字，在此之前的所有代码都用于设置上下文：先创建一个备份文件，然后打开并产出 `__enter__` 方法返回的可读文件和可写文件句柄的引用。`yield` 关键字之后的代码相当于 `__exit__` 方法，负责把文件句柄关闭。如果什么地方出错了，那就从备份中恢复文件。

我们对 `with` 语句和上下文管理器的概述到此结束，下面通过一个完整的示例探讨 `match/case`。

18.3 案例分析：lis.py 中的模式匹配

2.6 节的“使用模式匹配序列实现一个解释器”以 Peter Norvig 的 lis.py 解释器（移植到 Python 3.10 的版本）为例，分析了 `evaluate` 函数中用到的序列模式。本节将视野放宽，谈一谈 lis.py 的整体机制，同时探讨 `evaluate` 函数中的所有 `case` 子句，不仅说明各个模式，而且还将分析解释器在各个 `case` 子句中做了什么。

除了进一步讲解模式匹配，本节还有 3 个目的。

01. 观察 Norvig 的 lis.py 通过地道的 Python 代码体现的编程美感。
02. 坚持简单的语言设计哲学如何让 Scheme 成为大师级作品。
03. 学习解释器的工作原理，深入理解 Python 和其他（解释型或编译型）编程语言。

分析 Python 代码之前，我们要先了解一下 Scheme，为分析案例奠定基础——说不定你以前没听说过 Scheme 或 Lisp。

18.3.1 Scheme 句法

与 Python 不同，Scheme 不区分表达式和语句，也没有中缀运算符，所有表达式都使用前置表示法，例如 `(+ x 13)`，而不是 `x + 13`。函数调用也使用这种前置表示法，例如 `(gcd x 13)`。

`(define x 13)` 是一种特殊形式，相当于 Python 中的赋值语句，即 `x = 13`。Scheme 和多数 Lisp 方言使用的表示法叫作 **S 表达式**。⁶

⁶Lisp 中一层层的括号被人诟病，不过合理的缩进和一个好的编辑器基本能解决这个问题。在可读性方面，主要的问题是函数调用也使用同样的表示法，例如 `(f ...)`，以及一些特殊形式，例如 `(define ...)`、`(if ...)` 和 `(quote ...)`，给人的感觉是根本不像函数调用。

示例 18-10 是一段简单的 Scheme 示例。

示例 18-10 使用 Scheme 计算最大公约数

```
(define (mod m n)
  (- m (* n (quotient m n))))

(define (gcd m n)
  (if (= n 0)
      m
      (gcd n (mod m n))))

(display (gcd 18 45))
```

示例 18-10 用到了 3 个 Scheme 表达式，其中两个是函数定义，即 `mod` 和 `gcd`，还有一个调用 `display`，输出 `(gcd 18 45)` 的结果 9。示例 18-11 使用 Python 改写这段代码（比欧几里得算法的英语说明还短）。

示例 18-11 作用同示例 18-10，用 Python 编写

```
def mod(m, n):
    return m - (m // n * n)

def gcd(m, n):
    if n == 0:
        return m
    else:
        return gcd(n, mod(m, n))

print(gcd(18, 45))
```

地道的 Python 应该使用 `%` 运算符，而不是自己定义 `mod` 函数，而且 `while` 循环的效率比递归高。我之所以定义两个函数，是为了让前后两个示例保持对应，方便你理解前面的 Scheme 代码。

Scheme 没有 `while` 或 `for` 之类的迭代控制流命令，迭代通过递归实现。注意，在 Scheme 和 Python 示例中都没有赋值。大量使用递归和尽量少使用赋值是函数式编程的标志性特点。⁷

⁷为了让通过递归实现的迭代保持实用和高效，Scheme 和其他函数式语言实现了真尾调用（proper tail call）。详见“杂谈”。

下面开始分析 lis.py 的 Python 3.10 版本。

18.3.2 导入和类型

示例 18-12 是 lis.py 的前几行。在 Python 3.10 中才能使用 `TypeAlias` 和类型联合运算符 `|`。

示例 18-12 lis.py：文件顶部

```
import math
import operator as op
from collections import ChainMap
from itertools import chain
from typing import Any, TypeAlias, NoReturn

Symbol: TypeAlias = str
Atom: TypeAlias = float | int | Symbol
Expression: TypeAlias = Atom | list
```

这里定义了以下几个类型。

`Symbol`

只是 `str` 的别名。在 lis.py 中，`Symbol` 表示标识符。没有支持切片、拆分等操作的字符串数据类型。⁸

⁸但是，Norvig 的第 2 版解释器 lispy.py 支持字符串数据类型，还支持句法宏、续体和真尾调用等高级功能。不过，lispy.py 的长度差不多是 lis.py 的 3 倍，理解难度也大不少。

`Atom`

一种简单的句法元素，例如一个数值或一个 `Symbol`，与由独立部分构成的复合结构（例如列表）相反。

`Expression`

Scheme 程序的基本单元，即由原子结构和列表（可以嵌套）构成的表达式。

18.3.3 解析器

Norvig 的解析器只有 36 行代码（见示例 18-13），展现了 Python 在处理 S 表达式这种简单的递归句法上的强大功能。不过，lis.py 没有考虑字符串数据、注释、宏等 Scheme 标准功能，以免情况变得复杂。

示例 18-13 lis.py：负责解析的主要函数

```
def parse(program: str) -> Expression:
    "从字符串中读取Scheme表达式。"
    return read_from_tokens(tokenize(program))

def tokenize(s: str) -> list[str]:
    "把字符串转换成词法单元列表。"
    return s.replace('(', ' ( ').replace(')', ' ) ').split()

def read_from_tokens(tokens: list[str]) -> Expression:
    "从一系列词法单元中读取表达式。"
    # 排版需要，省略了很多解析代码
```

这部分的主要函数是 `parse`，它读取一个字符串形式的 S 表达式，返回一个 `Expression` 对象。根据示例 18-12 中的定义，`Expression` 可以是一个 `Atom` 或 `list`，或者二者的嵌套形式。

Norvig 在 `tokenize` 函数中采用了一个聪明的技巧，他在输出中的各个圆括号两侧添加了空格，然后拆分，得到以 `'('` 和 `)'` 分隔的词法单元列表。之所以可以这么做，是因为 lis.py 实现的 Scheme 子集没有字符串类型，因此每个 `'('` 或 `)'` 都是表达式的定界符。递归解析代码在 `read_from_tokens` 函数中，这个函数有 14 行代码。我们的重点是解释器的其他部分，这个函数略过不讲。

下面是一些 doctest，摘自 `lispy/py3.10/examples_test.py`。

```
>>> from lis import parse
>>> parse('1.5')
1.5
>>> parse('ni!')
```

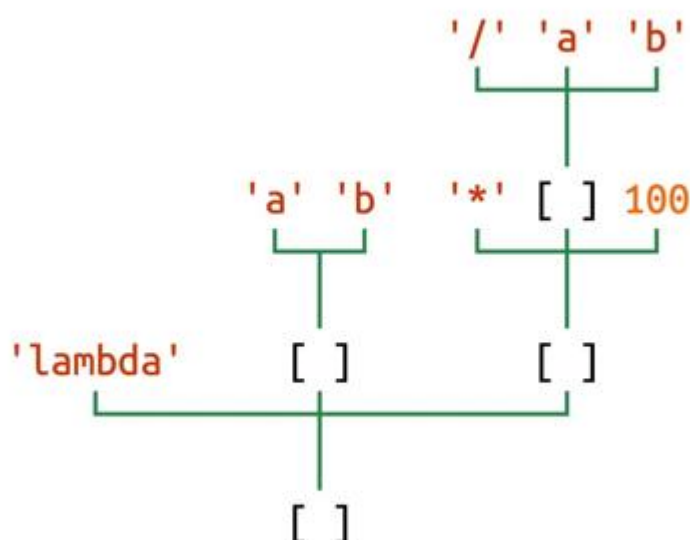
```
'ni!'
>>> parse('(gcd 18 45)')
['gcd', 18, 45]
>>> parse('''
... (define double
...     (lambda (n)
...         (* n 2)))
... ''')
['define', 'double', ['lambda', ['n'], ['*', 'n', 2]]]
```

这个 Scheme 子集的解析规则很简单。

01. 像数值的词法单元解析为 `float` 或 `int`。
02. 除数值、`'('` 和 `)'` 之外，其他的单元都解析为 `Symbol`，即用作标识符的字符串。这其中包括 `+`、`set!` 和 `make-counter` 等原始文本，它们在 Scheme 中是有效的标识符，但在 Python 中不是。
03. `'('` 和 `)'` 之间的表达式递归解析为包含原子结构或嵌套列表的列表，嵌套列表可能还包含原子结构和嵌套列表。

按照 Python 解释器的术语来说，`parse` 输出的是抽象句法树 (Abstract Syntax Tree, AST)：方便起见，把 Scheme 程序表示成嵌套列表，用一种树状结构展示，最外层列表是树干，内层列表是树枝，原子结构则是树叶（见图 18-1）。

```
(lambda (a b) (* (/ a b) 100))
```



```
['lambda', ['a', 'b'], ['*', ['/', 'a', 'b'], 100]]
```

图 18-1: 一个 Scheme lambda 表达式的源码表示（具体句法）、树状表示和一组 Python 对象表示（抽象句法）

18.3.4 环境

Environment 类扩展 collections.ChainMap，增加 change 方法，更新串联的某个字典中的值，如示例 18-14 所示。ChainMap 实例在 self.maps 属性中存储串联的映射。change 方法为 Scheme 中的 (set! ...) 形式（详见后文）提供支持。

示例 18-14 lis.py: Environment 类

```
class Environment(ChainMap[Symbol, Any]):
    "ChainMap的子类，允许就地更改项。"

    def change(self, key: Symbol, value: Any) -> None:
        "找到key在何处定义，更新对应的值。"
        for map in self.maps:
            if key in map:
```

```
        map[key] = value # type: ignore[index]
    return
    raise KeyError(key)
```

注意，`change` 方法只更新现有的键。⁹ 试图更新未找到的键，抛出 `KeyError`。

⁹在我检查这一章时，`typedshed` 项目的 6042 号工单尚未解决，因此我加上了 `# type: ignore[index]` 注释。在 `typedshed` 项目中，`ChainMap` 使用 `MutableMapping` 注解，但是根据 `maps` 属性的类型提示，该属性的值是 `Mapping` 列表，这间接导致 `Mypy` 把整个 `ChainMap` 当作不可变对象。

`Environment` 的作用如以下 `doctest` 所示。

```
>>> from lis import Environment
>>> inner_env = {'a': 2}
>>> outer_env = {'a': 0, 'b': 1}
>>> env = Environment(inner_env, outer_env)
>>> env['a'] ❶
2
>>> env['a'] = 111 ❷
>>> env['c'] = 222
>>> env
Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 1})
>>> env.change('b', 333) ❸
>>> env
Environment({'a': 111, 'c': 222}, {'a': 0, 'b': 333})
```

❶ 读取值时，`Environment` 的效果与 `ChainMap` 相同：从左向右在嵌套的映射中搜索键。因此，`outer_env` 中 `a` 的值被 `inner_env` 中的值遮盖了。

❷ 使用 `[]` 覆盖新值或插入新项，总是先应用于第一个映射（本例中的 `inner_env`）。

❸ `env.change('b', 333)` 寻找 `'b'` 键，在 `outer_env` 中就地赋予新值。

接下来是 `standard_env()` 函数（见示例 18-15），它构建并返回一个 `Environment` 对象，加载预定义的函数——类似于 Python 中始终可用的 `__builtins__` 模块。

示例 18-15 `lis.py`: `standard_env()` 函数构建并返回全局环境

```
def standard_env() -> Environment:
    "含有Scheme标准过程的环境。"
    env = Environment()
    env.update(vars(math))    # sin、cos、sqrt、pi等
    env.update({
        '+': op.add,
        '-': op.sub,
        '*': op.mul,
        '/': op.truediv,
        # 这里省略了很多运算符定义
        'abs': abs,
        'append': lambda *args: list(chain(*args)),
        'apply': lambda proc, args: proc(*args),
        'begin': lambda *x: x[-1],
        'car': lambda x: x[0],
        'cdr': lambda x: x[1:],
        # 这里省略了很多函数定义
        'number?': lambda x: isinstance(x, (int, float)),
        'procedure?': callable,
        'round': round,
        'symbol?': lambda x: isinstance(x, Symbol),
    })
    return env
```

可以看到，`env` 映射加载了以下内容：

- Python `math` 模块中的所有函数；
- Python `op` 模块中的部分运算符；
- 使用 Python `lambda` 表达式构建的一些简单而强大的函数；
- Python 内置函数，有些名称有变，例如 `callable` 变成了 `procedure?`；有些直接映射，例如 `round`。

18.3.5 REPL

Norvig 实现的 REPL (read-eval-print loop, “读取 - 求值 - 输出”循环) 不难理解, 但是对用户不友好 (见示例 18-16)。如果没有向 `lis.py` 提供命令行参数, 模块底部定义的 `main()` 函数调用 `repl()` 函数。在 `lis.py>` 提示符下, 必须输入正确且完整的表达式, 倘若忘记闭合括号, 则 `lis.py` 直接崩溃。¹⁰

¹⁰研究 Norvig 的 `lis.py` 和 `lispy.py` 的过程中, 我自己派生了一个版本, 名为 `mylis`。这个版本添加了一些功能, REPL 接受局部 S 表达式, 提示你继续输入。这就像 Python 的 REPL 一样, 它知道我们还没输入完毕, 在下一行输出 ... 提示符, 等我们输入完整的表达式或语句之后才求解。`mylis` 还能优雅地处理几个错误, 不过还是容易崩溃, 不如 Python 的 REPL 稳固。

示例 18-16 REPL 函数

```
def repl(prompt: str = 'lis.py> ') -> NoReturn:
    """提示-读取-求值-输出"循环。"
    global_env = Environment({}, standard_env())
    while True:
        ast = parse(input(prompt))
        val = evaluate(ast, global_env)
        if val is not None:
            print(lispstr(val))

def lispstr(exp: object) -> str:
    """把Python对象转换成Lisp理解的字符串。"
    if isinstance(exp, list):
        return '(' + ' '.join(map(lispstr, exp)) + ')'
    else:
        return str(exp)
```

这两个函数的作用简述如下。

```
repl(prompt: str = 'lis.py> ') -> NoReturn
```

调用 `standard_env()` 为全局环境提供内置函数, 然后进入无限循环, 读取并解析输入的各行, 在全局环境中求值, 最后显示结果 (除非为 `None`)。 `evaluate` 可能会修改 `global_env`。假如用户定义了新的全局变量或具名函数, 这些内容存储在环境的第一个映射

中，即 `repl` 函数第一行内 `Environment` 构造函数调用中那个空字典。

```
lispstr(exp: object) -> str
```

与 `parse` 函数的作用相反：提供表示一个表达式的 Python 对象，`lispstr` 函数返回相应的 Scheme 源码。例如，提供 `['+', 2, 3]`，得到的结果是 `'(+ 2 3)'`。

18.3.6 求值函数

现在，我们可以欣赏 Norvig 实现的表达式求值函数是多么美妙了——我使用 `match/case` 重新实现的版本更上一层楼。示例 18-17 中的 `evaluate` 函数接受的参数是一个 `Expression`（由 `parse` 构建）和一个 `Environment`。

`evaluate` 函数的主体只有一个 `match` 语句，匹配对象是表达式 `exp`。`case` 子句中的模式表达 Scheme 的句法和语义，一目了然。

示例 18-17 `evaluate` 函数计算表达式的值

```
KEYWORDS = ['quote', 'if', 'lambda', 'define', 'set!']

def evaluate(exp: Expression, env: Environment) -> Any:
    "在环境中求解表达式。"
    match exp:
        case int(x) | float(x):
            return x
        case Symbol(var):
            return env[var]
        case ['quote', x]:
            return x
        case ['if', test, consequence, alternative]:
            if evaluate(test, env):
                return evaluate(consequence, env)
            else:
                return evaluate(alternative, env)
        case ['lambda', [*parms], *body] if body:
            return Procedure(parms, body, env)
        case ['define', Symbol(name), value_exp]:
```

```
env[name] = evaluate(value_exp, env)
case ['define', [Symbol(name), *parms], *body] if body:
    env[name] = Procedure(parms, body, env)
case ['set!', Symbol(name), value_exp]:
    env.change(name, evaluate(value_exp, env))
case [func_exp, *args] if func_exp not in KEYWORDS:
    proc = evaluate(func_exp, env)
    values = [evaluate(arg, env) for arg in args]
    return proc(*values)
case _:
    raise SyntaxError(lispstr(exp))
```

下面依次分析各个 case 子句的作用。有时，我会添加注释，给出解析成 Python 列表时匹配模式的 S 表达式。doctest 摘自 examples_test.py，用于演示各个 case 子句。

01. 求解数值

```
case int(x) | float(x):
    return x
```

匹配对象：

int 或 float 实例。

操作结果：

原封不动返回。

实例演示：

```
>>> from lis import parse, evaluate, standard_env
>>> evaluate(parse('1.5'), {})
1.5
```

02. 求解符号

```
case Symbol(var):  
    return env[var]
```

匹配对象:

Symbol 实例, 即用作标识符的字符串。

操作结果:

在 env 中查找 var, 返回找到的值。

实例演示:

```
>>> evaluate(parse('+'), standard_env())  
<built-in function add>  
>>> evaluate(parse('ni!'), standard_env())  
Traceback (most recent call last):  
  ...  
KeyError: 'ni!'
```

03. (quote ...)

特殊形式 quote 把原子结构和列表视作数据, 而不是需要求解的表达式。

```
# (quote (99 bottles of beer))  
case ['quote', x]:  
    return x
```

匹配对象:

以 'quote' 符号开始、后跟一个表达式 x 的列表。

操作结果:

不求值, 直接返回 x。

实例演示：

```
>>> evaluate(parse('(quote no-such-name)'), standard_env())
'no-such-name'
>>> evaluate(parse('(quote (99 bottles of beer))'),
standard_env())
[99, 'bottles', 'of', 'beer']
>>> evaluate(parse('(quote (/ 10 0))'), standard_env())
['/', 10, 0]
```

若没有 `quote`，则上述测试中的每个表达式都会抛出错误。

- 在环境中查找 `no-such-name`，抛出 `KeyError`。
- `(99 bottles of beer)` 无法求解，因为数值 99 不是用于命名特殊形式、运算符或函数的 `Symbol`。
- `(/ 10 0)` 抛出 `ZeroDivisionError`。

为什么编程语言有保留关键字

`quote` 的作用虽然简单，但是在 Scheme 中不能实现为函数。对于 `(quote (f 10))` 表达式，`quote` 的特别之处是，它阻止解释器求解 `(f 10)`，真正得到的结果就是一个列表，包含一个 `Symbol` 和一个 `int`。相比之下，对于 `(abs (f 10))` 这样的函数调用，解释器先求解 `(f 10)`，然后调用 `abs`。所以，`quote` 是一个保留关键字，因为它必须作为一种特殊形式处理。

一般来说，保留关键字起到如下作用。

- 引入特殊的求值规则，例如 `quote` 和 `lambda`——不求解任何子表达式。
- 变更控制流，例如 `if` 和函数调用——也有特殊的求值规则。
- 管理环境，例如 `define` 和 `set`。

Python 和其他编程语言也有保留关键字，原因同上。请想一想 `def`、`if`、`yield`、`import` 和 `del` 在 Python 中有什么作用。

04. (`if ...`)

```
# (if (< x 0) 0 x)
case ['if', test, consequence, alternative]:
    if evaluate(test, env):
        return evaluate(consequence, env)
    else:
        return evaluate(alternative, env)
```

匹配对象：

一个列表，以 `'if'` 开头，后跟 3 个表达式：`test`、`consequence` 和 `alternative`。

操作结果：

求解 `test`：

- 为真时，求解 `consequence`，返回求值结果；
- 否则，求解 `alternative`，返回求值结果。

实例演示：

```
>>> evaluate(parse('(if (= 3 3) 1 0)'), standard_env())
1
>>> evaluate(parse('(if (= 3 4) 1 0)'), standard_env())
0
```

`consequence` 和 `alternative` 分支必须是单个表达式。如果一个分支需要多个表达式，则可以通过 `(begin exp1 exp2...)` 组合。在 `lis.py` 中，`begin` 是一个函数，详见示例 18-15。

05. (lambda ...)

Scheme 的 `lambda` 形式定义匿名函数。它没有 Python 中的 `lambda` 表达式那么多限制，在 Scheme 中，任何函数都可以使用 `(lambda ...)` 句法编写。

```
# (lambda (a b) (/ (+ a b) 2))
case ['lambda' [*parms], *body] if body:
    return Procedure(parms, body, env)
```

匹配对象：

一个列表，以 `'lambda'` 开头，后跟：

- 一个列表，表示一个或多个参数名称；
- 一个或多个表达式，汇入 `body`（卫语句确保 `body` 不为空）。

操作结果：

使用参数名称、表达式列表（作为函数主体）和当前环境创建一个 `Procedure` 实例，返回该实例。

实例演示：

```
>>> expr = '(lambda (a b) (* (/ a b) 100))'
>>> f = evaluate(parse(expr), standard_env())
>>> f # doctest: +ELLIPSIS
<lis.Procedure object at 0x...>
>>> f(15, 20)
75.0
```

`Procedure` 类实现闭包概念，即一个可调用对象，存有参数名称、函数主体和定义函数时环境的引用。稍后分析 `Procedure` 类的代码。

06. (define ...)

有两种句法形式用到 `define` 关键字。最简单的形式如下。

```
# (define half (/ 1 2))
case ['define', Symbol(name), value_exp]:
    env[name] = evaluate(value_exp, env)
```

匹配对象：

一个列表，以 `'define'` 开头，后跟一个 `Symbol` 和一个表达式。

操作结果：

求解表达式，把求值结果放入 `env` 中，以 `name` 为键。

实例演示：

```
>>> global_env = standard_env()
>>> evaluate(parse('(define answer (* 7 6))'), global_env)
>>> global_env['answer']
42
```

这个 `case` 子句的 `doctest` 创建了一个 `global_env`，目的是确认 `evaluate` 把 `answer` 放入了 `Environment`。

使用这种简单的 `define` 形式可以创建变量，或者以 `(lambda ...)` 为 `value_exp`，为匿名函数绑定名称。

标准的 Scheme 为定义具名函数提供了简洁方式，即 `define` 的第二种形式。

```
# (define (average a b) (/ (+ a b) 2))
case ['define', [Symbol(name), *parms], *body] if body:
    env[name] = Procedure(parms, body, env)
```

匹配对象：

一个列表，以 'define' 开头，后跟：

- 一个列表，以一个 `Symbol(name)` 开头，后跟零项或多项，汇入名为 `parms` 的列表；
- 一个或多个表达式，汇入 `body`（卫语句确保 `body` 不为空）。

操作结果：

- 使用参数名称、表达式列表（作为函数主体）和当前环境创建一个 `Procedure` 实例；
- 把 `Procedure` 实例放入 `env`，以 `name` 为键。

示例 18-18 中的 `doctest` 定义一个名为 `%` 的函数，计算百分比，并把该函数添加到 `global_env` 中。

示例 18-18 定义一个名为 `%` 的函数，计算百分比

```
>>> global_env = standard_env()
>>> percent = '(define (% a b) (* (/ a b) 100))'
>>> evaluate(parse(percent), global_env)
>>> global_env['%'] # doctest: +ELLIPSIS
<lis.Procedure object at 0x...>
>>> global_env['%'](170, 200)
85.0
```

调用 `evaluate` 后，我们检查 `%` 有没有绑定一个接受两个数值参数并返回一个百分比的 `Procedure` 实例。

匹配第二种 `define` 形式的 `case` 子句不要求 `parms` 中的项全是 `Symbol` 实例。构建 `Procedure` 实例之前可以对此检查，不过我没有这么做，为的是保持与 Norvig 的版本一样简单易懂。

07. (set! ...)

set! 形式更改已定义变量的值。¹¹

```
# (set! n (+ n 1))
case ['set!', Symbol(name), value_exp]:
    env.change(name, evaluate(value_exp, env))
```

匹配对象:

一个列表, 以 'set!' 开头, 后跟一个 Symbol 和一个表达式。

操作结果:

使用表达式的求值结果更新 env 中 name 的值。

Environment.change 方法从局部环境到全局环境依序遍历, 把找到的第一个 name 更新为新值。如果不是为了实现 'set!' 关键字, 那么在这个解释器中, 使用 Environment 类型的地方都可以使用 Python 的 ChainMap。

Python 的 nonlocal 和 Scheme 的 set! 解决同样的问题

set! 形式与 Python 中的 nonlocal 关键字作用类似: 声明 nonlocal x 之后, 可以使用 x = 10 更新局部作用域外部定义的 x 变量。在 Python 中, 如未声明 nonlocal x, 则 x = 10 始终创建一个局部变量 (详见 9.7 节)。

类似地, (set! x 10) 更新可能在函数局部环境外部定义的 x。相比之下, (define x 10) 中的 x 始终是局部变量, 在局部环境中创建或更新。

nonlocal 和 (set! ...) 都是为了更新闭包中变量存储的程序状态。示例 9-13 使用 nonlocal 实现一个计算累计

平均值的函数，在闭包中存储 `count` 和 `total`。下面使用 `lis.py` 定义的 Scheme 子集实现同样的功能。

```
(define (make-averager)
  (define count 0)
  (define total 0)
  (lambda (new-value)
    (set! count (+ count 1))
    (set! total (+ total new-value))
    (/ total count)
  )
)
(define avg (make-averager)) ❶
(avg 10) ❷
(avg 11) ❸
(avg 15) ❹
```

❶ 使用 `lambda` 定义的内部函数创建一个闭包，把变量 `count` 和 `total` 的值初始化为 0。得到的闭包绑定给 `avg`。

❷ 返回 10.0。

❸ 返回 10.5。

❹ 返回 12.0。

上述代码是 `lispy/py3.10/examples_test.py` 中的一个测试。

下面分析函数调用。

08. 函数调用

```
# (gcd (* 2 105) 84)
case [func_exp, *args] if func_exp not in KEYWORDS:
    proc = evaluate(func_exp, env)
    values = [evaluate(arg, env) for arg in args]
    return proc(*values)
```

匹配对象：

一个有一项或更多项的列表；

卫语句确保 `func_exp` 不在 `['quote', 'if', 'define', 'lambda', 'set!']` 中——示例 18-17 中位于 `evaluate` 前面；

这个模式匹配任何包含一个或多个表达式的列表，把第一个表达式绑定给 `func_exp`，余下的表达式作为列表（可能为空）绑定给 `args`。

操作结果：

- 求解 `func_exp`，得到函数 `proc`；
- 求解 `args` 中的各项，构建参数值列表；
- 调用 `proc`，传入各个参数值，返回结果。

实例演示：

```
>>> evaluate(parse('( % (* 12 14) (- 500 100) ')), global_env)
42.0
```

这个 `doctest` 接续示例 18-18，假定 `global_env` 中有一个名为 `%` 的函数。传给 `%` 函数的是算术表达式，以此强调参数在调用函数之前求值。

这个 `case` 子句的卫语句不可或缺，因为 `[func_exp, *args]` 能匹配任何含有一项或多项的序列。然而，如果 `func_exp` 是某个关键字，而且没有匹配前面任何一种情况，那就说明句法有误。

09. 捕获句法错误

如果 `exp` 不匹配前面任何一种情况，那么兜底 `case` 子句抛出 `SyntaxError`。

```
case _:  
    raise SyntaxError(lispstr(exp))
```

下面的例子解析一个格式有误的 `(lambda ...)`，抛出 `SyntaxError`。

```
>>> evaluate(parse('(lambda is not like this)'),  
standard_env())  
Traceback (most recent call last):  
...  
SyntaxError: (lambda is not like this)
```

如果匹配函数调用的 `case` 子句没有拒绝关键字的卫语句，那么 `(lambda is not like this)` 表达式会被当作函数调用处理，抛出 `KeyError`，因为 `'lambda'` 不在环境中——就好像 `lambda` 不是 Python 内置函数。

¹¹很多编程教程一开始就会教你赋值，但是在经典的 Scheme 图书《计算机程序的构造和解释（原书第 2 版）》（Harold Abelson 等人著，即人们常说的 SICP 或“巫师书”）中，直到第 151 页才出现 `set!`。函数式编程与命令式和面向对象编程不同，不改变状态也能做很多事。

18.3.7 实现闭包的 Procedure 类

`Procedure` 类命名为 `Closure` 或许好一些，这样才能表明它的作用，即在环境中定义的一个函数。函数定义包括参数的名称和构成函数主体的表达式。环境的作用是在调用函数时为**自由变量**提供值。自由变量指函数主体中出现的除参数、局部变量和全局变量以外的变量。**闭包**和**自由变量**概念在 9.6 节已经讲过。

我们已经学过在 Python 中如何使用闭包，下面深入探究 `lis.py` 是如何实现闭包的。

```

class Procedure:
    "用户定义的Scheme过程。"

    def __init__( ❶
        self, parms: list[Symbol], body: list[Expression], env:
Environment
    ):
        self.parms = parms ❷
        self.body = body
        self.env = env

    def __call__(self, *args: Expression) -> Any: ❸
        local_env = dict(zip(self.parms, args)) ❹
        env = Environment(local_env, self.env) ❺
        for exp in self.body: ❻
            result = evaluate(exp, env)
        return result ❼

```

- ❶ 使用 `lambda` 或 `define` 形式定义函数时调用。
- ❷ 保存函数名称、主体表达式和环境，供后面使用。
- ❸ 由 `case [func_exp, *args]` 子句中最后一行内的 `proc(*values)` 调用。
- ❹ 以 `self.parms` 为局部变量的名称，以提供的 `args` 为值，构建 `local_env` 映射。
- ❺ 构建一个新的组合 `env`，把 `local_env` 放在开头，`self.env`（定义函数时保存的环境）放在后面。
- ❻ 迭代 `self.body` 中的各个表达式，在组合的 `env` 中求解。
- ❼ 返回最后一个表达式的求值结果。

在 `lis.py` 中，`evaluate` 后面有几个简单的函数：`run` 读取并执行完整的 Scheme 程序，`main` 根据命令行参数调用 `run` 或 `repl`——与 Python 类似。那两个函数没有什么新知识，我就不讲了。我的目的是

向你展现 Norvig 这个小型解释器的精妙之处，并进一步分析闭包的原理，以及 `match/case` 对 Python 功能的补充。

本节讲了很多模式匹配，最后我们要正式确立 OR 模式的概念。

18.3.8 使用 OR 模式

用 `|` 分隔的模式称为 OR 模式：只要有一个子模式匹配，整个模式就匹配。18.3.6 节中“求解数值”分析的那个模式就是 OR 模式。

```
case int(x) | float(x):  
    return x
```

在 OR 模式中，所有子模式必须使用相同的变量。做出这个限制的目的在于确保无论哪个子模式匹配，变量在看守表达式和 `case` 主体中均可用。



在 `case` 子句上下文中，`|` 运算符有特殊的意义。它不触发处理 `a | b` 一类表达式的特殊方法 `__or__`，在其他上下文中，可能会重载这个运算符，根据运算符对象的不同，执行并集或者整数按位或（integer bitwise-or）等操作。

OR 模式不要求必须在模式的顶层，在子模式中也可以使用 `|`。假如我们希望 `lis.py` 既接受希腊字母 λ ¹²，也接受 `lambda` 关键字，模式可以像下面这样写。

¹² λ (U+03BB) 在 Unicode 标准中的正式名称是 GREEK SMALL LETTER LAMDA。你没看错，在 Unicode 数据中，这个字符的名称就是“lamda”，没有“b”。

```
# ( $\lambda$  (a b) (/ (+ a b) 2) )  
case ['lambda' | ' $\lambda$ ', [*parms], *body] if body:  
    return Procedure(parms, body, env)
```

接下来，转到本章的第三个，也是最后一个话题：在 Python 中，`else` 子句可以出现在哪些不寻常的位置。

18.4 先做这个，再做那个：if 语句之外的 else 块

这个语言功能不是什么秘密，只不过没有得到重视：else 子句不仅能在 if 语句中使用，还能在 for、while 和 try 语句中使用。

for/else、while/else 和 try/else 的语义联系紧密，不过与 if/else 差别很大。起初，else 这个单词的意思甚至阻碍了我对这些功能的理解，但是最终我习惯了。

else 子句的规则如下。

for

仅当 for 循环运行完毕时（即 for 循环没有被 break 语句中止）才运行 else 块。

while

仅当 while 循环因条件为假值而退出时（即 while 循环没有被 break 语句中止）才运行 else 块。

try

仅当 try 块没有抛出异常时才运行 else 块。官方文档还指出：“else 子句抛出的异常不由前面的 except 子句处理。”

在所有情况下，如果异常或者 return、break 或 continue 语句导致控制权跳到了复合语句的主块之外，则 else 子句也被跳过。



我觉得除了 if 语句之外的其他语句使用 else 关键字是非常不明智的选择。“else”蕴含着“排他性”这层意思，例如“要么运行这个循环，要么做那件事”。可是，在循环中，else 的语义恰好

相反：“运行这个循环，然后做那件事。”因此，使用 `then` 关键字或许更好。`then` 在 `try` 语句的上下文中也说得通：“尝试运行这个，然后做那件事。”然而，添加新关键字属于语言的重大变化，不是那么容易下决心的。

在这些语句中使用 `else` 子句通常能让代码更易于阅读，而且能省去一些麻烦，不用设置控制标志或者添加额外的 `if` 语句。

在循环中使用 `else` 子句的一般方式如下述代码片段所示。

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```

一开始，你可能觉得没必要在 `try/except` 块中使用 `else` 子句。毕竟，在下述代码片段中，只有 `dangerous_call()` 不抛出异常，`after_call()` 才会执行，对吧？

```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

然而，`after_call()` 不应该放在 `try` 块中。为了清晰和准确，只应把可能抛出预期异常的语句放在 `try` 块中。因此，像下面这样写更好。

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

现在很明确，`try` 块防守的是 `dangerous_call()` 可能出现的错误，而不是 `after_call()`。而且很明显，只有 `try` 块不抛出异常，`after_call()` 才会执行。

在 Python 中，`try/except` 不仅用于处理错误，还常用于控制流程。为此，Python 官方术语表还定义了一个缩略词（口号）。

EAFP

取得原谅比获得许可容易（Easier to ask for forgiveness than permission）。这是一种常用的 Python 编程风格，先假定存在有效的键或属性，如果假设不成立，那么捕获异常。这种风格简单明快，特点是代码中有很多 `try` 和 `except` 语句。与其他很多语言一样（例如 C 语言），这种风格的对立面是 LBYL 风格。

接下来，Python 术语表定义了 LBYL。

LBYL

三思而后行（Look before you leap）。这种编程风格在调用函数或查找属性或键之前显式测试前提条件。与 EAFP 风格相比，这种风格的特点是代码中有很多 `if` 语句。在多线程环境中，LBYL 风格可能会在“检查”和“行事”的空隙引入条件竞争。例如，对 `if key in mapping: return mapping[key]` 这段代码来说，如果在测试之后——但在查找之前——另一个线程从映射中删除了那个键，那么这段代码就会失败。这个问题可以使用锁或者 EAFP 风格解决。

如果选择使用 EAFP 风格，那就要更深入地了解 `else` 子句，并在 `try/except` 语句中合理使用。



讨论 `match` 语句时，有些人（包括我）认为 `match` 语句也应该有一个 `else` 子句。最后的决定是不需要这个子句，因为

`case _`: 具有同样的作用。¹³

¹³观察 python-dev 邮件列表中的这场讨论, 我认为 `else` 被拒的一个原因是对 `else` 在 `match` 中如何缩进没有达成一致: `else` 应该是与 `match` 还是与 `case` 同级缩进?

下面对本章做个小结。

18.5 本章小结

本章首先讨论了上下文管理器和 `with` 语句的作用。很快我们就知道, 除了自动关闭打开的文件之外, `with` 语句还有很多用途。我们自己动手实现了一个上下文管理器——定义了有 `__enter__` / `__exit__` 方法的 `LookingGlass` 类, 说明了如何在 `__exit__` 方法中处理异常。Raymond Hettinger 在 PyCon US 2013 上所做的主题演讲传达了一个重要的观点: `with` 不仅能管理资源, 还可用于去掉常规的设置和清理代码, 或者在另一个过程前后执行的操作。¹⁴

¹⁴“What Makes Python Awesome?”, 第 21 张幻灯片。

随后, 我们分析了标准库中 `contextlib` 模块里的函数。其中, `@contextmanager` 装饰器能把包含一个 `yield` 语句的简单生成器变成上下文管理器——这比定义一个至少包含两个方法的类要更简洁。我们使用 `looking_glass` 生成器函数实现了 `LookingGlass` 类的功能, 还讨论了使用 `@contextmanager` 时如何处理异常。

接下来, 我们研究了 Peter Norvig 编写的优雅的 `lis.py`。这是使用地道的 Python 编写的一个 Scheme 解释器, 我又使用 `match/case` 重构了 `evaluate` 函数——对任何解释器来说都是核心。若想理解 `evaluate`, 就需要懂一点 Scheme, 还要有一个 S 表达式解析器、一个简单的 REPL, 并通过 `collection.ChainMap` 的子类 `Environment` 构造嵌套作用域。可见, 透过 `lis.py` 不仅能进一步探

讨模式匹配，还能学习更多其他知识。我们学习了解释器的各个部分是如何协同工作的，借此对 Python 自身的核心功能也有了更深的理解：为什么要有保留关键字，作用域规则运行机制，以及如何构建和使用闭包。

18.6 延伸阅读

《Python 语言参考手册》中的第 8 章“复合语句”全面说明了 `if`、`for`、`while` 和 `try` 语句中 `else` 子句的作用。关于 `try/except` 语句（有或没有 `else` 子句）是否符合 Python 风格，Raymond Hettinger 在 Stack Overflow 中对“Is it a good practice to use try-except-else in Python?”这一问题做出了精彩回答。在《Python 技术手册（第 3 版）》（Alex Martelli 等人著）中，有一章是关于异常的，那一章极好地讨论了 EAFP 风格。Alex 认为“取得原谅比获得许可容易”是由计算领域的先驱 Grace Hopper 首先提出的。

在 Python 标准库文档中，第 4 章“Built-in Types”中有一节专门说明了上下文管理器的类型。Python 语言参考中还有 `__enter__` / `__exit__` 这两个特殊方法的文档，在“With Statement Context Managers”一节中。上下文管理器由“PEP 343—The ‘with’ Statement”引入。

在 PyCon US 2013 的主题演讲中，Raymond Hettinger 强调，`with` 语句是“这门语言的一项迷人功能”。在这次大会上，他在“Transforming Code into Beautiful, Idiomatic Python”演讲中还展示了上下文管理器的几个有趣应用。

Jeff Preshing 写的一篇博客文章很有趣，题为“The Python `_with_Statement` by Example”，他以图形库 `pycairo` 为例说明了上下文管理器。

`contextlib.ExitStack` 类基于 Nikolaus Rath 最初的提议。他写了一篇短文，说明为什么需要这样一个类，题为“On the Beauty of

Python's ExitStack”。在那篇文章中，Rath 提出，ExitStack 与 Go 语言中的 defer 语句（我认为这是 Go 语言最优秀的功能之一）作用相似，但是更灵活。

Beazley 和 Jones 在《Python Cookbook（第 3 版）中文版》一书中发明了上下文管理器的独特用途。“8.3 让对象支持上下文管理协议”一节实现了一个 LazyConnection 类，它的实例是上下文管理器，在 with 块中能自动打开和关闭网络连接。“9.22 以简单的方式定义上下文管理器”一节编写了一个用于统计代码运行时间的上下文管理器，还编写了一个使用事务修改 list 对象的上下文管理器：在 with 块中创建 list 实例的副本，所有改动都针对副本；仅当 with 块没有抛出异常，正常执行完毕之后，才用副本替代原来的列表。这样做简单又巧妙。

Peter Norvig 在“(How to Write a (Lisp) Interpreter (in Python))”和“(An ((Even Better) Lisp) Interpreter (in Python))”两篇文章中分析了他实现的小型 Scheme 解释器。lis.py 和 lispy.py 的代码在 norvig/pytudes 仓库中。我的 fluentpython/lispy 是从 lis.py 派生的 mylis，已更新到 Python 3.10，REPL、命令行集成、示例和测试更完善，还有进一步学习 Scheme 的资料。目前，Racket 是最好的 Scheme 方言，也是学习和试验的最佳环境。

杂谈

取出面包

在 PyCon US 2013 的主题演讲“What Makes Python Awesome”中，Raymond Hettinger 说他第一次看到 with 语句的提案时，觉得“有点晦涩难懂”。这和我一开始的反应类似。PEP 通常难以理解，PEP 343 尤甚。

然后，Hettinger 告诉我们，他认识到在计算机语言的发展历程中，**子程序**（subroutine）是最重要的发明。如果有一系列操作，例如 A;B;C 和 P;B;Q，那么可以把 B 拿出来，变成子程序。

这就好比把三明治的馅儿取出来，这样我们便能使用金枪鱼搭配不同的面包。可是，如果我们想把面包取出来，使用小麦面包夹不同的馅儿呢？这就是 `with` 语句实现的功能。`with` 语句是对子程序的补充。Hettinger 接着说道：

`with` 语句是非常了不起的功能。我建议你在实践中深挖这个功能的用途。使用 `with` 语句或许能做一些意义深远的事情。`with` 语句的最佳用途尚未被发掘出来。我预计，如果有好的用途，其他语言以及未来的语言会借鉴这个功能。或许，你正在参与的事情几乎与子程序的发明一样意义深远。

Hettinger 承认，他夸大了 `with` 语句的作用。尽管如此，`with` 语句仍是一个十分有用的功能。他用三明治类比，道出 `with` 语句是子程序的补充；那一刻，我的脑海中浮现了许多可能性。

如果你想让任何人信服 Python 是出色的语言，那么一定要观看 Hettinger 的主题演讲。关于上下文管理器的部分从 23:00 开始，到 26:15 结束。不过，整个主题演讲都很精彩。

真尾调用实现高效递归

标准的 Scheme 实现需要提供**真尾调用**（proper tail calls, PTC），通过递归迭代替代命令式语言中的 `while` 循环。一些作者把真尾调用称作**尾调用优化**（tail call optimization, TCO）。其实，尾调用优化与真尾调用不是一回事。

尾调用是指一个函数返回一个函数（可以是同一个函数，也可以不是）调用的结果。示例 18-10 和示例 18-11 中的 `gcd` 函数在 `if` 语句的假值分支中做（递归）尾调用。

然而，下面的 `factorial` 函数没有做尾调用。

```
def factorial(n):  
    if n < 2:
```

```
    return 1
return n * factorial(n - 1)
```

最后一行虽然调用了 `factorial`，但不是尾调用，因为返回值不是递归调用的结果，而是乘以 `n` 之后才返回。

下面的版本用到了尾调用，因此是**尾递归**（tail recursive）。

```
def factorial_tc(n, product=1):
    if n < 1:
        return product
    return factorial_tc(n - 1, product * n)
```

Python 不支持真尾调用，因此编写尾递归函数没什么好处。就以上两个版本而言，我觉得第一版更简短，也更易于理解。实际使用中，别忘了 Python 有 `math.factorial` 可用，它使用 C 语言实现，没用到递归。我想说的是，即便是实现了真尾调用的语言，也不是每个递归函数都能从中受益，只有那些精心编写，做了尾调用的函数才能从中受益。

对于支持真尾调用的语言，解释器遇到尾调用时会跳入被调用函数的主体，不创建新的栈帧，可以节省内存。有些编译型语言也实现了真尾调用，不过或许是一种优化措施，可以禁用。

对于尾调用优化的定义，以及真尾调用在非严格意义上属于函数式语言的编程语言（例如 Python 或 JavaScript）中有多大的价值，目前没有普遍共识。人们往往期待函数式语言支持真尾调用功能，而不是锦上添花的优化措施。如果一门语言除了递归之外没有其他迭代机制，那么从实用的角度来看，就需要有真尾调用。Norvig 的 `lis.py` 没有实现真尾调用，但是后来实现的 `lisper.py` 更完善，支持真尾调用。

对真尾调用说不的 Python 和 JavaScript

CPython 没有实现真尾调用，或许永远不会。Guido van Rossum 在“Final Words on Tail Calls”一文中道出了个中原委。

在那篇文章中，下面这段话是关键，概括了主旨：

就我个人而言，我觉得对某些语言来说，（真尾调用）是很好的功能，但是并不适合 Python：仅消除部分调用的栈跟踪肯定会让很多用户困惑，这些用户没有受到尾调用信仰的熏陶，但是只要在调试器中跟踪过几个调用就已经熟悉了调用语义。

2015 年，真尾调用被纳入 JavaScript 的 ECMAScript 6 标准。截至 2021 年 10 月，WebKit 中的 JavaScript 解释器已经实现。WebKit 是 Safari 使用的引擎。其他主流浏览器的 JavaScript 解释器都不支持真尾调用，依赖 Google 为 Chrome 开发的 V8 引擎的 Node.js 也不支持。根据“ECMAScript 6 兼容表”，转译器和 JavaScript 的泥子脚本，例如 TypeScript、ClojureScript 和 Babel，也都不支持真尾调用。

实现方拒绝支持真尾调用的原因众多，不过大都与 Guido van Rossum 所言相似：真尾调用提高了所有人的调试难度，只有少数坚持使用递归进行迭代的人能从中受益。详细说明见 Graham Marlow 写的“What happened to proper tail calls in JavaScript?”一文。

有些时候，递归是最佳方案，即便是对于不支持真尾调用的 Python。在早前一篇关于这个话题的文章中，Guido 写道：

.....典型的 Python 实现允许 1000 次递归，这对于未使用递归的代码和依赖遍历的代码（例如典型的解析树）来说足够了，但是不够递归迭代大型列表的代码使用。

我同意 Guido 和大多数 JavaScript 实现方的观点：真尾调用不适合 Python 或 JavaScript。与受限的 `lambda` 句法相比，缺少真尾调用是阻碍以函数式风格编写 Python 程序的主要因素。

如果你想了解真尾调用在解释器中的工作原理，又觉得 Norvig 的 `lispy.py` 过于复杂（代码多、功能多），可以看一下 `mylis_2`。与真尾调用相关的部分位于 `evaluate` 函数中那个无限循环内匹配函数调用的 `case` 子句中，同时满足两个条件时，开始尾调用，解释器跳入下一个 `Procedure` 的主体，而不递归调用 `evaluate`。那些小型解释器揭示了抽象的强大：尽管 Python 没有实现真尾调用，但是依然可以（也不太难）用 Python 实现支持真尾调用的解释器。我的实现方式借鉴了 Peter Norvig 的代码，感谢教授！

Norvig 对 `evaluate()` 中模式匹配的看法

我把 `lis.py` 的 Python 3.10 版本给 Peter Norvig 看过。他喜欢用模式匹配实现的版本，不过提出了不同的方案。他不喜欢我编写的卫语句，换作是他，他会为每个关键字编写一个 `case` 子句，在 `case` 块中测试，为 `SyntaxError` 提供更具体的消息，例如指明主体为空。这样一来，`case [func_exp, *args] if func_exp not in KEYWORDS:` 中的卫语句就不需要了，因为所有关键字在针对函数调用的 `case` 子句之前都已经处理了。

为 `mylis` 增加更多功能时，我或许会采纳 Norvig 的建议。在示例 18-17 中，`evaluate` 的组织方式更多是为了方便教学，因为要与示例 2-11 中的 `if/elif/...` 块保持对应，借助 `case` 子句讲解更多的模式匹配功能，并且表明使用模式匹配写出的代码更简洁。

第 19 章 Python 并发模型

并发指同时处理多件事。

并行指同时做多件事。

二者不同，但有联系。

一个关于结构，一个关于执行。

并发用于制定方案，用来解决可能（但未必）并行的问题。

——Rob Pike
Go 语言联合创造者¹

¹摘自“Concurrency Is Not Parallelism”演讲的第 8 张幻灯片。

本章讲的是如何让 Python“同时处理多件事”。这可能涉及并发或并行编程——即使热衷于行话的学者不同意这样使用术语。本章开篇就采用 Rob Pike 的非正式定义，但是请注意，我发现有些论文和图书所声称的并行计算，实际上基本上都是并发。²

²我曾与 Imre Simon 教授一起研究和工作，他说科学界有两个重要过错：使用不同的词表示相同的事物，以及使用同一个词表示不同的事物。Imre Simon（1943—2009）是巴西计算机科学先驱，对自动机理论（Automata Theory）有杰出贡献，开创了热带数学（Tropical Mathematics）领域。

在 Pike 看来，并行是并发的一种特殊情况。所有并行系统都是并发的，但不是所有并发系统都是并行的。在 21 世纪初，我们使用单核设备在 GNU Linux 上同时处理 100 个进程。一台拥有 4 个 CPU 核的现代笔记本计算机，在正常情况下，任何时间段内运行的进程数随随便便都会超过 200 个。如果并行执行 200 个任务，则需要 200 个核。因此，多数计算实际是并发的，而不是并行的。操作系统管理着

数百个进程，确保每个进程都有机会取得进展，即使 CPU 本身同时做的事情不能超过 4 件。

本章假定你事先不具备并发或并行编程知识。简要介绍相关概念之后，我们将通过简单的示例学习和比较 Python 为并发编程提供的 3 个核心包：`threading`、`multiprocessing` 和 `asyncio`。

本章后 30% 内容概述可以增强 Python 应用性能和伸缩性的第三方工具、库、应用服务器和分布式任务队列。这些都是重要的话题，虽然这是一本关注 Python 语言核心功能的书，可我觉得本书第 2 版有必要稍微讲一讲，因为 Python 之所以适合并发和并行计算，除了标准库提供了相关支持以外，还有第三方的功劳。这就是为什么 YouTube、Dropbox、Instagram、Reddit 等网站在初期使用 Python 作为主要语言也能实现规模化的原因——尽管有人一直声称“Python 不具伸缩性”。

19.1 本章新增内容

本章是第 2 版新增的。19.4 节中的旋转指针示例原先在讲 `asyncio` 那一章。这一版对那些示例做了改进，而且借此首次展示 Python 的 3 种并发方式：线程、进程和原生协程。

余下的内容都是全新的，只有少数几个段落与第 1 版讲 `concurrent.futures` 和 `asyncio` 的章节相同。

19.7 节与本书其他部分不同，没有代码示例。那一节的目的是提及一些值得学习的重要工具，让你利用 Python 标准库之外的工具实现高性能的并发和并行。

19.2 全景概览

导致并发编程困难的因素很多，但我想谈谈最基本的因素：启动线程或进程十分容易，关键是如何跟踪线程或进程。³

³本节是我的朋友 Bruce Eckel 提议写的，他是多本图书的作者，涉及 Kotlin、Scala、Java 和 C++。

调用一个函数，发出调用的代码开始阻塞，直到函数返回。因此，你知道函数什么时候执行完毕，而且能轻松地得到函数的返回值。如果函数可能抛出异常，则把函数调用放在 `try/except` 块中，捕获错误。

这些熟悉的概念在你启动线程或进程后都不可用了：你无法轻松地得知操作何时结束，若想获取结果或捕获错误，则需要设置某种通信信道，例如消息队列。

此外，启动线程或进程有一定消耗，仅仅为了计算一个结果就退出，肯定得不偿失。通常，更好的选择是让各个线程或进程进入一个“职程”（worker），循环等待要处理的输入，以此分摊启动成本。但是，这又进一步增加了通信难度，还会引起更多问题。如果不需要职程了，那么如何退出呢？怎样退出才能做到不中断作业，避免留下未处理完毕的数据和未释放的资源（例如打开的文件）呢？同样，解决这些问题通常涉及消息和队列。

协程的启动成本很低。使用 `await` 关键字启动的协程，返回值容易获取，可以安全取消，捕获异常的位置也明确。但是，协程通常由异步框架启动，因此监控难度与线程或进程相当。

最后，正如后面将要介绍的，Python 协程和线程不适合 CPU 密集型任务。

鉴于此，并发编程需要学习新的概念和编程模式。首先，我们要对核心概念确立统一认识。

19.3 术语定义

本章和接下来的第 20 章和第 21 章会用到以下术语。

并发

处理多个待定任务，一次处理一个或并行处理多个（如果条件允许），直到所有任务最终都成功或失败。对于单核 CPU，如果操作系统的调度程序支持交叉执行待定任务，也能实现并发。并发也叫多任务处理（multitasking）。

并行

同时执行多个计算任务的能力。需要一个多核 CPU、多个 CPU、一个 GPU 或一个集群中的多台计算机。

执行单元

并发执行代码的对象的统称，每个对象的状态和调用栈是独立的。Python 原生支持 3 种执行单元：**进程**、**线程**和**协程**。

进程

计算机程序运行时的一个实例，消耗内存和部分 CPU 时间。现代桌面操作系统通常同时管理数百个进程，每个进程都隔离在自己的私有内存空间中。进程通过管道、套接字或内存映射文件进行通信——这些方式都只能携带原始字节。Python 对象必须序列化（转换）为原始字节才能从一个进程传递到另一个进程。这个过程耗费资源，而且不是所有 Python 对象都可以序列化。进程可以派生子进程，子进程彼此之间以及与父进程之间是隔离的。进程支持**抢占式多任务处理**机制：操作系统调度程序定期抢占（挂起）运行中的进程，让其他进程运行。这意味着冻结的进程理论上不会冻结整个系统。

线程

单个进程中的执行单元。一个进程启动后，只使用一个线程，即主线程。通过调用操作系统 API，进程可以创建更多线程，执行并发操作。一个进程内的线程共享相同的内存空间（存储活动的 Python 对象）。因此，线程之间可以轻松地共享数据，但是如果多个线程同时更新同一个对象，则可能导致数据损坏。与进程一样，线程在操作系统调度程序的监督下也可以实现抢占式多任务处理。对于同一份作业，线程消耗的资源比进程少。

协程

可以挂起自身并在以后恢复的函数。在 Python 中，**经典协程**由生成器函数构建，**原生协程**使用 `async def` 定义。17.13 节已经介绍过经典协程，原生协程的用法将在第 21 章探讨。Python 协程通常在**事件循环**（也在同一个线程中）的监督下在单个线程中运行。

`asyncio`、`Curio` 或 `Trio` 等异步编程框架为基于协程的非阻塞 I/O 提供了事件循环和支持库。协程支持协作式多任务处理：一个协程必须使用 `yield` 或 `await` 关键字显式放弃控制权，另一个协程才可以并发（而非并行）开展工作。这意味着，协程中只要有导致阻塞的代码，事件循环和其他所有协程的执行就都会受到阻塞——这一点与进程和线程的抢占式多任务处理形成鲜明对比。另外，对于同一份作业，协程消耗的资源比线程或进程少。

队列

一种数据结构，可以放入和取出项，顺序通常是先入先出（FIFO）。独立的执行单元可以通过队列交换应用数据和控制消息，例如错误代码和终止信号。队列的实现因底层并发模型而异：Python 标准库中的 `queue` 包提供的队列类支持线程，`multiprocessing` 和 `asyncio` 包则实现了其他队列类。`queue` 和 `asyncio` 包中还有非先入先出队列：`LifoQueue` 和 `PriorityQueue`。

锁

一种供执行单元用来同步操作和避免数据损坏的对象。更新共享数据结构时，当前代码应持有相关的锁，并告诉程序的其他部分等到锁被释放后再访问这个数据结构。最简单的锁是互斥锁。锁的实现取决于底层并发模型。

争用

对有限资源的争夺。当多个执行单元尝试访问共享资源（例如锁或存储器）时，就会发生资源争用。当计算密集型进程或线程必须等待操作系统调度程序为其分配 CPU 时间时，还会发生 CPU 争用。

接下来使用这些术语分析 Python 对并发的支持。

进程、线程和声名狼藉的 Python GIL

下面分 10 点说明这些概念在 Python 编程中的应用。

01. Python 解释器的每个实例是一个进程。使用 `multiprocessing` 或 `concurrent.futures` 库可以启动额外的 Python 进程。Python 的 `subprocess` 库用于启动运行外部程序（不管使用何种语言编写）的进程。
02. Python 解释器仅使用一个线程运行用户的程序和内存垃圾回收程序。使用 `threading` 或 `concurrent.futures` 库可以启动额外的 Python 线程。
03. 对对象引用计数和解释器其他内部状态的访问受一个锁的控制，这个锁是“全局解释器锁”（Global Interpreter Lock, GIL）。任意时间点上只有一个 Python 线程可以持有 GIL。这意味着，任意时间点上只有一个线程能执行 Python 代码，与 CPU 核数量无关。
04. 为了防止一个 Python 线程无限期持有 GIL，Python 的字节码解释器默认每 5 毫秒暂停当前 Python 线程⁴，释放 GIL。被暂停的线程可以再次尝试获得 GIL，但是如果有其他线程等待，那么操作系统调度程序可能会从中挑选一个线程开展工作。

05. 我们编写的 Python 代码无法控制 GIL。但是，耗时的任务可由内置函数或 C 语言（以及其他能在 Python/C API 层级接合的语言）扩展释放 GIL。
06. Python 标准库中发起系统调用⁵的函数均可释放 GIL。这包括所有执行磁盘 I/O、网络 I/O 的函数，以及 `time.sleep()`。NumPy/SciPy 库中很多 CPU 密集型函数，以及 `zlib` 和 `bz2` 模块中执行压缩和解压操作的函数，也都释放 GIL。⁶
07. 在 Python/C API 层级集成的扩展也可以启动不受 GIL 影响的非 Python 线程。这些不受 GIL 影响的线程无法更改 Python 对象，但是可以读取或写入内存中支持缓冲协议的底层对象，例如 `bytearray`、`array.array` 和 NumPy 数组。
08. GIL 对使用 Python 线程进行网络编程的影响相对较小，因为 I/O 函数释放 GIL，而且与内存读写相比，网络读写的延迟始终很高。各个单独的线程无论如何都要花费大量时间等待，所以线程可以交错执行，对整体吞吐量不会产生重大影响。正如 David Beazley 所言：“Python 线程非常擅长什么都不做。”⁷
09. 对 GIL 的争用会降低计算密集型 Python 线程的速度。对于这类任务，在单线程中依序执行的代码更简单，速度也更快。
10. 若想在多核上运行 CPU 密集型 Python 代码，必须使用多个 Python 进程。

⁴这个时间间隔使用 `sys.getswitchinterval()` 获取，使用 `sys.setswitchinterval(s)` 设置。

⁵系统调用指用户的代码调用操作系统内核的函数。I/O、计时器和锁都是通过系统调用获得的内核服务。

⁶Antoine Pitrou（为 Python 3.2 贡献了时间分片 GIL 逻辑）发到 `python-dev` 邮件列表中的一个消息特别提到了 `zlib` 和 `bz2` 模块。

⁷来源：“Generators: The Final Frontier”教程第 106 张幻灯片。

`threading` 模块的文档对此做了很好的概括。⁸

⁸摘自“Thread objects”一节最后一段。

CPython 实现细节：由于 CPython 有 GIL，因此同一时间只有一个线程能执行 Python 代码（尽管有些旨在提升性能的库可以克服这个限制）。如果你希望应用程序充分地利用多核设备的计算资源，那么建议使用 `multiprocessing` 或 `concurrent.futures.ProcessPoolExecutor`。然而，如果你想同时运行多个 I/O 密集型任务，那么线程仍是最合适的模型。

前一段开头指出那是“CPython 实现细节”，因为 GIL 不是 Python 语言规定的机制。Jython 和 IronPython 就没有 GIL。可惜，二者落后较多，还停留在 Python 2.7。高性能的 PyPy 解释器的 2.7 和 3.7 版本（截至 2021 年 6 月的最新版）也有 GIL。



本节没有提到协程，因为默认情况下，协程共用同一个 Python 线程，而且受异步框架提供的事件循环监管，所以不受 GIL 影响。在异步程序中也可以使用多个线程，但是最佳实践是在同一个线程中运行事件循环和所有协程，其他线程负责执行特定的任务。详见 21.8 节。

概念讲得差不多了，接下来分析代码。

19.4 一个演示并发的“Hello World”示例

在讨论线程以及如何避免 GIL 的过程中，Python 贡献者 Michele Simionato 发布了一个示例，可以看作演示并发的“Hello World”示例，即能展示 Python“一心二用”最简单的程序。

Simionato 的程序使用的是 `multiprocessing`，经我修改，又分别实现了使用 `threading` 和 `asyncio` 的版本。下面先讲 `threading`

版本。如果你学习过如何在 Java 或 C 语言中使用线程，那么这个版本对你而言或许并不陌生。

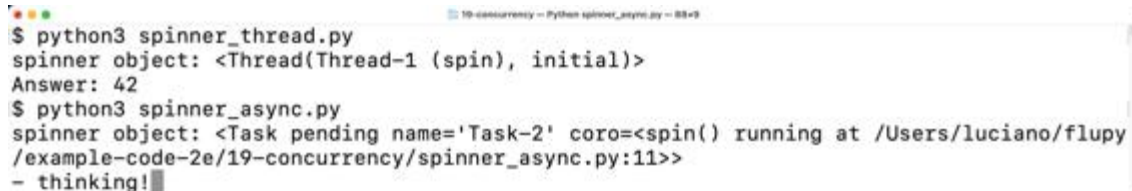
19.4.1 使用线程实现旋转指针

接下来几个示例的想法很简单：启动一个函数，阻塞 3 秒，期间在终端展示字符动画，让用户知道程序正在运转，没有停滞。

这个脚本在界面上的相同位置依次显示字符串 "\|/-" 中的各个字符，实现旋转指针动画。⁹ 当缓慢的计算结束后，旋转指针那一行内容清空，显示结果：Answer: 42。

⁹有很多 Unicode 字符可用于实现简单的动画，例如盲文图案。简单起见，我使用的是 ASCII 字符 "\|/-"。

图 19-1 包括这个旋转指针的两个版本，一个用线程实现，另一个用协程实现。



```
$ python3 spinner_thread.py
spinner object: <Thread(Thread-1 (spin), initial)>
Answer: 42
$ python3 spinner_async.py
spinner object: <Task pending name='Task-2' coro=<spin() running at /Users/luciano/flupy
/example-code-2e/19-concurrency/spinner_async.py:11>>
- thinking!
```

图 19-1： spinner_thread.py 和 spinner_async.py 两个脚本的输出类似：旋转指针对象的字符串表示形式和文本“Answer: 42”。在这张截图中， spinner_async.py 脚本仍在运行中，显示的是动画消息“/ thinking!”； 3 秒后，那一行将被替换成“Answer: 42”

首先分析 spinner_thread.py 脚本。示例 19-1 是该脚本的前两个函数，示例 19-2 是余下代码。

示例 19-1 spinner_thread.py: spin 和 slow 函数

```
import itertools
import time
```

```
from threading import Thread, Event

def spin(msg: str, done: Event) -> None: ❶
    for char in itertools.cycle(r'\|/'): ❷
        status = f'\r{char} {msg}' ❸
        print(status, end='', flush=True)
        if done.wait(.1): ❹
            break ❺
        blanks = ' ' * len(status)
        print(f'\r{blanks}\r', end='') ❻

def slow() -> int:
    time.sleep(3) ❼
    return 42
```

❶ 这个函数将在单独的线程中运行。done 参数的值是一个 `threading.Event` 实例，一个用于同步线程的简单对象。

❷ 这是一个无限循环，因为 `itertools.cycle` 一次产出一个字符，一直反复迭代字符串。

❸ 用文本实现动画的技巧：使用 ASCII 回车符（`'\r'`）把光标移到行头。

❹ 如果其他线程设置了这个事件，则 `Event.wait(timeout=None)` 方法返回 `True`；经过 `timeout` 指定的时间后，返回 `False`。这里把暂停时间设为 0.1 秒，作用是把动画的帧率设为 10fps。如果你希望指针旋转快一些，那就把暂停时间值设置小一些。

❺ 退出无限循环。

❻ 显示空格，并把光标移到开头，清空状态行。

❼ `slow()` 由主线程调用。假设有一个 API 调用通过网络发送，速度很慢。调用 `sleep` 阻塞主线程，但是 GIL 已被释放，因此指针还能继续旋转。



通过这个示例，我们了解到的最重要的一点是，调用 `time.sleep()` 阻塞所在的线程，但是释放 GIL，其他 Python 线程可以继续运行。

`spin` 和 `slow` 两个函数并发执行。主线程（启动程序时唯一的线程）将启动一个新线程运行 `spin`，然后调用 `slow`。Python 故意没有提供终止线程的 API，如果想终止线程，则必须向线程发送相应的消息。

在 Python 中，协调线程的信号机制，使用 `threading.Event` 类最简单。`Event` 实例有一个内部布尔标志，开始时为 `False`。调用 `Event.set()` 可把这个标志设为 `True`。这个标志为 `False` 时，在一个线程中调用 `Event.wait()`，该线程将被阻塞，直到另一个线程调用 `Event.set()`，致使 `Event.wait()` 返回 `True`。使用 `Event.wait(s)` 设置一个暂停时间（单位为秒），经过这段时间后，`Event.wait(s)` 调用返回 `False`，如果另一个线程调用 `Event.set()`，则立即返回 `True`。

示例 19-2 中的 `supervisor` 函数使用一个 `Event` 实例向 `spin` 函数发送退出信号。

示例 19-2 `spinner_thread.py`: `supervisor` 和 `main` 函数

```
def supervisor() -> int: ❶
    done = Event() ❷
    spinner = Thread(target=spin, args=('thinking!', done)) ❸
    print(f'spinner object: {spinner}') ❹
    spinner.start() ❺
    result = slow() ❻
    done.set() ❼
    spinner.join() ❽
    return result

def main() -> None:
    result = supervisor() ❾
    print(f'Answer: {result}')
```

```
if __name__ == '__main__':  
    main()
```

- ❶ supervisor 返回 slow 的结果。
- ❷ threading.Event 实例是协调 main 线程和 spinner 线程活动的关键，详见下面的分析。
- ❸ 创建一个 Thread 实例，target 关键字参数的值是一个函数，args 参数的值是一个元组，即传给 target 函数的位置参数。
- ❹ 显示 spinner 对象。输出是 <Thread(Thread-1, initial)>，其中 initial 是线程的状态，表示尚未启动。
- ❺ 启动 spinner 线程。
- ❻ 调用 slow，阻塞 main 线程。同时，次线程运行旋转指针动画。
- ❼ 把 Event 标志设为 True，终止 spin 函数中的 for 循环。
- ❽ 等待，直到 spinner 线程结束。
- ❾ 运行 supervisor 函数。我之所以分别定义 main 和 supervisor 两个函数，是为了与示例 19-4 中的 asyncio 版本保持对应。

main 线程设置 done 事件后，spinner 线程终将收到信号，干净退出。

下面来看一下使用 multiprocessing 包实现的版本。

19.4.2 使用进程实现旋转指针

multiprocessing 包支持在单独的 Python 进程而非线程中运行并发任务。创建 multiprocessing.Process 实例后，一个全新的 Python 解释器以子进程的形式在后台启动。由于每个 Python 进程都有自己的 GIL，因此程序可以使用所有可用的 CPU 核——但是，最终还是取决于操作系统的调度程序。19.6 节再讲具体影响，对这个简单的程序来说没什么实质差别。

本节的目的是介绍 multiprocessing 包，展示它的 API 与 threading API 的对应关系，方便你把使用线程的简单程序改用进程实现。使用 multiprocessing 包实现的版本如示例 19-3 所示。

示例 19-3 spinner_proc.py：只给出了改动部分，其他代码都与 spinner_thread.py 一样

```
import itertools
import time
from multiprocessing import Process, Event ❶
from multiprocessing import synchronize ❷

def spin(msg: str, done: synchronize.Event) -> None: ❸

# spin函数的余下部分以及slow函数没有改动，与spinner_thread.py一样

def supervisor() -> int:
    done = Event()
    spinner = Process(target=spin, ❹
                      args=('thinking!', done))
    print(f'spinner object: {spinner}') ❺
    spinner.start()
    result = slow()
    done.set()
    spinner.join()
    return result

# main函数也没有改动
```

❶ multiprocessing API 基本模仿 threading API，不过类型提示和 Mypy 还是揭示了一处区别：multiprocessing.Event 是函

数 (`threading.Event` 是类) , 返回 `synchronize.Event` 实例.....

② 因此需要导入 `multiprocessing.synchronize`.....

③ 才能编写这个类型提示。

④ `Process` 类的基本用法与 `Thread` 相似。

⑤ `spinner` 对象显示为 `<Process name='Process-1' parent=14868 initial>`, 其中 14868 是运行 `spinner_proc.py` 的 Python 实例的进程 ID。

`threading` 和 `multiprocessing` 的 API 基本相同, 但是实现方式差别很大, 而且为了处理多进程编程增加的复杂度, `multiprocessing` 的 API 更多。例如, 把线程换成进程后, 一个难点是如何在被操作系统隔离且无法共享 Python 对象的进程之间通信。为此, 跨进程传递的对象需要序列化和反序列化, 这样一来开销就增加了。在示例 19-3 中, 跨进程传递的数据只有 `Event` 状态。在 `multiprocessing` 模块底层的 C 代码中, `Event` 状态通过操作系统底层信号量实现。¹⁰

¹⁰信号量是一种基础构件, 可用于实现其他同步机制。Python 为线程、进程和协程提供了多个信号量类。21.7.1 节将用到 `asyncio.Semaphore`。



从 Python 3.8 开始, 标准库提供了 `multiprocessing.shared_memory` 包, 但是不支持用户定义类的实例。除了原始字节, 这个包还允许进程共享一个 `ShareableList`。这是一个可变序列类型, 存放固定数量的项, 项的类型可以是 `int`、`float`、`bool` 和 `None`, 以及单项不超过 10MB 的 `str` 和 `bytes`。详见 `ShareableList` 的文档。

使用进程和线程实现的版本讲完了, 下面来看使用协程实现的版本。

19.4.3 使用协程实现旋转指针



第 21 章将专门讲解如何使用协程实现异步编程。本节只是简单介绍，为了与线程和进程两个并发模型相比较。因此，我们将忽略很多细节。

线程和进程由操作系统调度程序分配 CPU 时间驱动。相比之下，协程由应用级事件循环驱动：事件循环管理待定协程队列，逐个驱动，监视由协程发起的 I/O 操作触发的事件，在各个事件发生时把控制权传回相应的协程。事件循环、库协程以及用户协程都在同一个线程中执行。因此，在协程中花费的任何时间都会减慢事件循环，以及所有其他协程。

从 `main` 函数入手，再分析 `supervisor` 函数，更易于理解旋转指针程序的协程版本。详见示例 19-4。

示例 19-4 `spinner_async.py`: `main` 函数和 `supervisor` 协程

```
def main() -> None: ❶
    result = asyncio.run(supervisor()) ❷
    print(f'Answer: {result}')

async def supervisor() -> int: ❸
    spinner = asyncio.create_task(spin('thinking!')) ❹
    print(f'spinner object: {spinner}') ❺
    result = await slow() ❻
    spinner.cancel() ❼
    return result

if __name__ == '__main__':
    main()
```

❶ 在这个程序中，`main` 是唯一的常规函数，其他的都是协程。

❷ `asyncio.run` 函数启动事件循环，驱动这个协程，最终也将启动其他协程。`main` 函数保持阻塞，直到 `supervisor` 返回值。

`supervisor` 的返回值将变成 `asyncio.run` 的返回值。

③ 原生协程使用 `async def` 定义。

④ `asyncio.create_task` 调度 `spin` 最终执行，现在立即返回一个 `asyncio.Task` 实例。

⑤ `spinner` 对象的字符串表示形式形如 `<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:11>>`。

⑥ `await` 关键字调用 `slow`，阻塞 `supervisor`，直到 `slow` 返回。`slow` 的返回值将赋予 `result`。

⑦ `Task.cancel` 方法在 `spin` 协程中抛出 `CancelledError` 异常，详见示例 19-5。

示例 19-4 展示了运行协程的 3 种主要方式。

```
asyncio.run(coro())
```

在常规函数中调用，驱动一个协程对象，通常作为程序中所有异步代码的入口，例如本例中的 `supervisor`。这个调用保持阻塞，一直到 `coro` 的主体返回。`run()` 调用的返回值是 `coro` 主体的返回值。

```
asyncio.create_task(coro())
```

在协程中调用，调度另一个协程最终执行。这个调用不中止当前协程。返回一个 `Task` 实例，包装协程对象，提供控制和查询协程状态的方法。

```
await coro()
```

在协程中调用，把控制权转给 `coro()` 返回的协程对象。中止当前协程，直到 `coro` 的主体返回。这个异步等待表达式的值是 `coro` 主体的返回值。



记住，通过 `coro()` 调用协程立即返回一个协程对象，但是不运行 `coro` 函数的主体。协程的主体由事件循环驱动。

下面分析示例 19-5 中的 `spin` 和 `slow` 协程。

示例 19-5 spinner_async.py: `spin` 和 `slow` 协程

```
import asyncio
import itertools

async def spin(msg: str) -> None: ❶
    for char in itertools.cycle(r'\|/'):
        status = f'\r{char} {msg}'
        print(status, flush=True, end='')
        try:
            await asyncio.sleep(.1) ❷
        except asyncio.CancelledError: ❸
            break
    blanks = ' ' * len(status)
    print(f'\r{blanks}\r', end='')

async def slow() -> int:
    await asyncio.sleep(3) ❹
    return 42
```

❶ 与 `spinner_thread.py`（见示例 19-1）不同，这里不需要通过 `Event` 信号指明 `slow` 的工作已经做完。

❷ 使用 `await asyncio.sleep(.1)` 代替 `time.sleep(.1)`，暂停时不阻塞其他协程。详见本例后面的实验。

❸ 在控制这个协程的 `Task` 实例上调用 `cancel` 方法，抛出 `asyncio.CancelledError`。捕获到这个异常就退出循环。

④ `slow` 协程也使用 `await asyncio.sleep` 代替 `time.sleep`。

实验：故意破坏，深入理解

建议你做一下这个实验，以深入理解 `spinner_async.py` 的运行机制。导入 `time` 模块，然后把 `slow` 协程中的 `await asyncio.sleep(3)` 替换成 `time.sleep(3)`，如示例 19-6 所示。

示例 19-6 `spinner_async.py`：把 `await asyncio.sleep(3)` 替换成 `time.sleep(3)`

```
async def slow() -> int:
    time.sleep(3)
    return 42
```

自己观察代码的行为比阅读文字更能加深印象。请你动手试一试，我等你。

实验中你会看到如下输出。

01. 显示旋转指针对象，形如 `<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:12>>`。
02. 旋转指针一直没有出现。程序挂起 3 秒。
03. 显示 `Answer: 42`，程序终止。

理解以上行为的关键是要知道，使用 `asyncio` 的 Python 代码只有一个执行流，除非显式启动额外的线程或进程。这意味着，任何时间点上都只有一个协程在执行。若想实现并发，则要把控制权由一个协程传给另一个协程。对于这个实验，我们把注意力集中在 `supervisor` 和 `slow` 协程上（见示例 19-7）。

示例 19-7 `spinner_async_experiment.py`：`supervisor` 和 `slow` 协程

```
async def slow() -> int:
    time.sleep(3) ❹
    return 42

async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!')) ❶
    print(f'spinner object: {spinner}') ❷
    result = await slow() ❸
    spinner.cancel() ❺
    return result
```

❶ 创建 `spinner` 任务，驱动 `spin` 最终执行。

❷ 这个输出表明 `Task` 处于“待定”状态。

❸ 这个 `await` 表达式把控制权转给 `slow` 协程。

❹ `time.sleep(3)` 阻塞 3 秒，程序什么也做不了，因为主线程被阻塞了，而主线程是唯一的线程。操作系统继续其他活动。3 秒过后，`sleep` 不再阻塞，`slow` 返回。

❺ `slow` 一旦返回，`spinner` 任务就被取消。控制流始终没有触及 `spin` 协程的主体。

`spinner_async_experiment.py` 给我们上了重要的一课，详见下面的警告栏。



在 `asyncio` 创建的协程中千万不要使用 `time.sleep(...)`，除非你想暂停整个程序。如果希望协程空闲一段时间，什么也不做，那么应该使用 `await asyncio.sleep(DELAY)`，把控制权交还 `asyncio` 事件循环，驱动其他待定协程。

greenlet 和 gevent

讲到协程并发，就不得不提 greenlet 包。¹¹ 这个包已经存在很多年，使用广泛，通过轻量级协程（叫作 greenlets）支持协作式多任务处理，而不使用 `yield` 或 `await` 等特殊句法，因此更易于集成到现有的顺序执行基准代码中。SQLAlchemy 1.4 ORM 内部使用 greenlets 实现新的异步 API，与 `asyncio` 兼容。

网络库 `gevent` 对 Python 标准库中的 `socket` 模块打了猴子补丁，把部分代码换成了 greenlets，以防止阻塞。在很大程度上，`gevent` 对周围的代码是透明的，因此顺序执行的应用程序和库（例如数据库驱动）无须大改就能执行并发网络 I/O。使用 `gevent` 的开源项目很多，包括大量部署的 Gunicorn（19.7.4 节会提到）。

¹¹感谢技术审校 Caleb Hattingh 和 Jürgen Gmach 的提醒，不然我就忘了 greenlet 和 gevent。

19.4.4 对比几版 supervisor 函数

`spinner_thread.py` 和 `spinner_async.py` 的行数相当。`supervisor` 函数是这几个示例的核心。本节详细比较几个版本中的 `supervisor` 函数。示例 19-8 只列出了示例 19-2 中的 `supervisor` 函数。

示例 19-8 `spinner_thread.py`：线程版 `supervisor` 函数

```
def supervisor() -> int:
    done = Event()
    spinner = Thread(target=spin,
                     args=('thinking!', done))
    print('spinner object:', spinner)
    spinner.start()
    result = slow()
    done.set()
    spinner.join()
    return result
```

为了方便比较，示例 19-9 列出了示例 19-4 中的 `supervisor` 协程。

示例 19-9 spinner_async.py: 异步版 supervisor 协程

```
async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!'))
    print('spinner object:', spinner)
    result = await slow()
    spinner.cancel()
    return result
```

这两版 supervisor 的异同概括如下。

- `asyncio.Task` 的作用基本等同于 `threading.Thread`。
- `Task` 驱动协程对象，而 `Thread` 调用可调对象。
- 协程通过 `await` 关键字显式让出控制权。
- 我们不自己动手实例化 `Task` 对象，而是把协程传给 `asyncio.create_task(...)` 创建。
- `asyncio.create_task(...)` 返回的 `Task` 对象随机被调度运行，但是 `Thread` 实例必须调用 `start` 方法才运行。
- 在线程版 supervisor 中，`slow` 是常规函数，直接由主线程调用。在异步版 supervisor 中，`slow` 是协程，由 `await` 驱动。
- 没有可以从外部终止线程的 API，必须发送信号，例如设置 `Event` 对象 `done`。而任务有实例方法 `Task.cancel()`，在中止协程主体的 `await` 表达式处抛出 `CancelledError`。
- supervisor 协程必须在 `main` 函数中使用 `asyncio.run` 启动。

经过比较，你应该对 `asyncio` 如何编排并发作业有了更深的了解。相比之下，你或许更熟悉 `Threading` 模块的运作机制。

关于线程和协程还有最后一点需要注意：但凡用线程做过重要的编程任务，你就会知道判断程序的状态有多么难，因为调度程序随时可能中断线程。因此，一定不能忘记持锁，保护程序的关键部分，以防止多步操作执行到一半被中断——这可能导致数据处于无效状态。

而使用协程时，我们编写的代码默认防止中断，因为必须显式使用 `await` 关键字，程序的其余部分才能运行。按照定义，协程本身即可“同步”，不用通过锁同步多个线程的操作，因为任何时刻都只能运行一个协程。想放弃控制权时，我们使用 `await` 把控制权交还调度程序。这也是可以安全取消协程的原因：按照定义，协程只能在中止它的 `await` 表达式处被取消，所以可以通过处理 `CancelledError` 异常执行清理工作。

`time.sleep()` 调用阻塞执行，但是什么也不做。接下来，我们来试验一个 CPU 密集型调用，借此深入理解 GIL，同时也了解 CPU 密集型函数在异步代码中的效果。

19.5 GIL 真正的影响

对于示例 19-1 中的线程版，把 `slow` 函数中的 `time.sleep(3)` 调用换成使用某个库发起的 HTTP 客户端请求之后，旋转指针仍然运转。这是因为一个设计精良的网络库在等待网络响应的过程中会释放 GIL。

同样，也可以把 `slow` 协程中的 `asyncio.sleep(3)` 表达式换掉，让 `await` 等待异步网络库返回响应，这是因为一个设计精良的异步网络库在等待响应期间会把控制权交还事件循环，让指针保持旋转。

对于 CPU 密集型代码，情况就不同了。以示例 19-10 中的 `is_prime` 函数为例，该函数在参数为素数时返回 `True`，否则返回 `False`。

示例 19-10 `primes.py`：一个易懂的素数检测函数，摘自 Python 文档中的 `ProcessPoolExecutor` 示例

```
def is_prime(n: int) -> bool:
    if n < 2:
        return False
    if n == 2:
```



```
        return True
    if n % 2 == 0:
        return False

    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
        if n % i == 0:
            return False
    return True
```

在我现在使用的办公笔记本电脑中，
`is_prime(5_000_111_000_222_021)` 调用耗时近 3.3 秒。¹²

¹²—一台 15 英寸的 MacBook Pro 2018，配置为 Intel Core i7 CPU，6 核，2.2GHz。

快速问答

请你根据目前所讲的内容回答以下 3 个问题。其中有一个问题很难回答（至少让我有点犯难）。

按以下方式修改代码，对旋转指针动画有什么影响？假设 `n = 5_000_111_000_222_021`——在我的设备中耗时 3.3 秒才判断出来它是素数。

01. 在 `spinner_proc.py` 中，把 `time.sleep(3)` 换成 `is_prime(n)` 调用。
02. 在 `spinner_thread.py` 中，把 `time.sleep(3)` 换成 `is_prime(n)` 调用。
03. 在 `spinner_async.py` 中，把 `await asyncio.sleep(3)` 换成 `is_prime(n)` 调用。

请先不要运行代码或者往下读，试着自己回答这些问题。然后，再根据题设要求，修改 `spinner_*.py` 示例。

下面按难易程度给出答案。

01. `multiprocessing` 版答案

旋转指针受一个子进程控制，因此在父进程进行素数检测的过程中，指针持续旋转。¹³

02. threading 版答案

旋转指针由一个次线程控制，因此在主线程做素数检测期间指针持续旋转。

一开始，我没有回答出来这个问题，我以为旋转指针会停滞，因为我过于高估了 GIL 的影响。

对这个示例来说，指针得以保持旋转的原因是 Python 每隔 5 毫秒（默认值）中止运行线程，其他待命线程可以获得 GIL。所以，运行 `is_prime` 的主线程每 5 毫秒中断一次，次线程复苏，迭代一次 `for` 循环，直到在 `done` 事件上调用 `wait` 方法，释放 GIL。随后，主线程再次获得 GIL，`is_prime` 接着计算 5 毫秒。

这对本例没有明显影响，因为 `spin` 函数迭代一次的速度很快，收到 `done` 事件后就释放 GIL，所以几乎没有争用 GIL。多数时候，GIL 由运行 `is_prime` 的主线程持有。

这个实验很简单，只涉及两个线程，因此我们才能使用线程处理计算密集型任务。在用到的两个线程中，一个独占 CPU，另一个 1 秒只复苏 10 次，更新旋转指针。

但是，如果有两个或以上线程都想占用大量 CPU 时间，那么程序运行速度要比顺序执行的代码更慢。

03. asyncio 版答案

对于 `spinner_async.py` 示例，在 `slow` 协程中调用 `is_prime(5_000_111_000_222_021)`，指针根本不旋转。这与示例 19-6 把 `await asyncio.sleep(3)` 替换成 `time.sleep(3)` 的效果一样。控制流由 `supervisor` 传给 `slow`，再传给 `is_prime`。`is_prime` 一旦返回，则 `slow` 随之返回，`supervisor` 恢复执行，立即取消 `spinner` 任务，使其根本来不及执行。程序看上去冻结大约 3 秒，然后显示结果。

使用 `sleep(0)` 小憩一下

为了保持指针旋转，一种办法是把 `is_prime` 定义为协程，在 `await` 表达式中定期调用 `asyncio.sleep(0)`，把控制权交还事件循环，如示例 19-11 所示。

示例 19-11 `spinner_async_nap.py`：现在 `is_prime` 是协程

```
async def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
        if n % i == 0:
            return False
        if i % 100_000 == 1:
            await asyncio.sleep(0) ❶
    return True
```

❶ 每 50 000 次迭代（因为 `range` 的步幅是 2）休眠一次。

`asyncio` 仓库中的 284 号工单详细讨论了 `asyncio.sleep(0)` 的用法。

然而，需要注意的是，这将拖慢 `is_prime`，以及事件循环和整个程序的速度——后二者才是重点。换成每 10 万次迭代执行一次 `await asyncio.sleep(0)` 之后，在我的设备中，旋转指针依然顺滑，只是用时达到了 4.9 秒，比原来的 `primes.is_prime` 函数慢了将近 50%（参数同为 `5_000_111_000_222_021`）。

使用 `await asyncio.sleep(0)` 只是权宜之计，更稳妥的办法是重构异步代码，把 CPU 密集型计算委托给另一个进程。第 21 章将给出一个使用 `asyncio.loop.run_in_executor` 的方案。另外，还可以使用任务队列（详见 19.7.5 节）。

目前，我们所做的实验只调用了一个 CPU 密集型函数。19.6 节说明如何并发执行多个 CPU 密集型调用。

¹³现在来看确实如此，因为你使用的大概率是现代操作系统，支持**抢占式多任务处理**。NT 时代以前的 Windows 和 OSX 时代以前的 macOS 不支持“抢占式”，任何进程都有可能占用 100% 的 CPU，导致整个系统假死。如今，我们还没有彻底摆脱这个问题，但是请相信我这个白胡子老头儿：20 世纪 90 年代，每个用户都经历过这种痛苦，唯一的解决办法就是硬重启。

19.6 自建进程池



我写这一节是为了展示如何使用多个进程处理 CPU 密集型任务，以及使用队列分配任务和收集结果的常见模式。第 20 章将给出一种为进程分配任务更简单的方式：

`concurrent.futures` 包中的 `ProcessPoolExecutor`（内部使用队列）。

本节将编写一个程序，计算 2 和 9 999 999 999 999 999 ($10^{16} - 1$) 之间或大于 2^{53} 的 20 个整数样本是否为素数。样本中的素数有大有小，还有可以分解为大小素数的合数。

我们以 sequential.py 程序的性能作为基准。下面是某次运行的结果。

```
$ python3 sequential.py
      2   P   0.000001s
142702110479723   P   0.568328s
299593572317531   P   0.796773s
3333333333333301   P   2.648625s
3333333333333333       0.000007s
3333335652092209       2.672323s
4444444444444423   P   3.052667s
4444444444444444       0.000001s
4444444448888889       3.061083s
5555553133149889       3.451833s
5555555555555503   P   3.556867s
5555555555555555       0.000007s
6666666666666666       0.000001s
6666666666666719   P   3.781064s
6666667141414921       3.778166s
7777777536340681       4.120069s
7777777777777753   P   4.141530s
7777777777777777       0.000007s
9999999999999917   P   4.678164s
9999999999999999       0.000007s
Total time: 40.31
```

结果分 3 列显示：

- 第 1 列是待检查的数；
- 第 2 列在数为素数时显示 P，否则显示空白；
- 第 3 列是检查相应数是否为素数的用时。

在这个示例中，总用时单独计算，近似于每次检查的时间之和，如示例 19-12 所示。

示例 19-12 sequential.py：对一个小数据集做素数检测（顺序执行版）

```
#!/usr/bin/env python3

"""
sequential.py: CPU密集型工作的顺序执行版、多进程版
和多线程版的比较基准。
"""
```

```

from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class Result(NamedTuple): ❶
    prime: bool
    elapsed: float

def check(n: int) -> Result: ❷
    t0 = perf_counter()
    prime = is_prime(n)
    return Result(prime, perf_counter() - t0)

def main() -> None:
    print(f'Checking {len(NUMBERS)} numbers sequentially:')
    t0 = perf_counter()
    for n in NUMBERS: ❸
        prime, elapsed = check(n)
        label = 'P' if prime else ' '
        print(f'{n:16} {label} {elapsed:9.6f}s')

    elapsed = perf_counter() - t0 ❹
    print(f'Total time: {elapsed:.2f}s')

if __name__ == '__main__':
    main()

```

❶ `check` 函数（下一个标号）返回一个 `Result` 元组，包含 `is_prime` 调用返回的布尔值和用时。

❷ `check(n)` 调用 `is_prime(n)`，并计算用时，返回一个 `Result`。

❸ 调用 `check` 检查样本中的每个数，显示结果。

❹ 计算并显示总用时。

19.6.1 基于进程的方案

下一个示例 (procs.py) 使用多个进程把素数检测分配给多个 CPU 核。我运行 procs.py 的用时如下。

```
$ python3 procs.py
Checking 20 numbers with 12 processes:
          2    P    0.000002s
33333333333333333333    0.000021s
44444444444444444444    0.000002s
55555555555555555555    0.000018s
66666666666666666666    0.000002s
  142702110479723    P    1.350982s
77777777777777777777    0.000009s
  299593572317531    P    1.981411s
99999999999999999999    0.000008s
3333333333333333301    P    6.328173s
33333335652092209    6.419249s
4444444444888888889    7.051267s
4444444444444444423    P    7.122004s
55555553133149889    7.412735s
5555555555555555503    P    7.603327s
6666666666666666719    P    7.934670s
66666667141414921    8.017599s
7777777536340681    8.339623s
777777777777777753    P    8.388859s
999999999999999917    P    8.117313s
20 checks in 9.58s
```

输出的最后一行表明，procs.py 的用时是 sequential.py 的约 23.77%。

19.6.2 理解用时

注意，最后一列是检查相应数的用时。例如，`is_prime(77777777777777753)` 用时近 8.4 秒，返回 `True`。与此同时，其他进程在并行检查其他数。

要检查的数共有 20 个。我编写的 `procs.py` 启动多个工作进程，具体数量等于 `multiprocessing.cpu_count()` 获取的 CPU 核数。

这里，总用时比每个检查的用时之和少很多。启动进程和进程间通信有一定开销，因此多进程版本的用时仅是顺序执行版本的约

23.77%。这已经很不错了，但是考虑到代码动用了我这台笔记本计算机的全部 CPU 核，启动了 12 个进程，结果还是有点让人失望。



在我用来撰写本章的 MacBook Pro 中，`multiprocessing.cpu_count()` 函数返回 12。这台笔记本计算机的 CPU 是 6 核 Core i7，但是操作系统报告有 12 个 CPU，原因在于 Intel 使用了超线程技术，一个核可以执行两个线程。然而，超线程技术只能在同一个核中的两个线程繁忙程度不同时才能发挥最大功效，比如一个线程未命中缓存，停顿下来等待数据，另一个线程正忙于处理数值。当然，天下没有免费的午餐，我这台笔记本计算机在处理不需要使用大量内存的计算密集型工作时（例如这里所做的简单的素数检测），性能也对得起那个 6 核 CPU。

19.6.3 利用多核进行素数检测的程序代码

把计算工作委托给线程或进程时，我们的代码不能直接调用承担工作的函数（简称职程），因此无法轻易获得返回值。职程由线程或进程库驱动，最终会产生一个结果，存储在某个地方。在并发编程中（也包括分布式系统），经常使用队列协调职程和收集结果。

`procs.py` 中新增的代码大都用于设置和使用队列。这个文件顶部的内容在示例 19-13 中。



`multiprocessing` 中的 `SimpleQueue` 是 Python 3.9 新增的。如果你使用的 Python 版本较老，可以把示例 19-13 中的 `SimpleQueue` 换成 `Queue`。

示例 19-13 `procs.py`：素数检测多进程版本的导入、类型和函数

```
import sys
from time import perf_counter
```



```

from typing import NamedTuple
from multiprocessing import Process, SimpleQueue, cpu_count ❶
from multiprocessing import queues ❷

from primes import is_prime, NUMBERS

class PrimeResult(NamedTuple): ❸
    n: int
    prime: bool
    elapsed: float

JobQueue = queues.SimpleQueue[int] ❹
ResultQueue = queues.SimpleQueue[PrimeResult] ❺

def check(n: int) -> PrimeResult: ❻
    t0 = perf_counter()
    res = is_prime(n)
    return PrimeResult(n, res, perf_counter() - t0)

def worker(jobs: JobQueue, results: ResultQueue) -> None: ❼
    while n := jobs.get(): ❽
        results.put(check(n)) ❾
        results.put(PrimeResult(0, False, 0.0)) ❿

def start_jobs(
    procs: int, jobs: JobQueue, results: ResultQueue ⓫
) -> None:
    for n in NUMBERS:
        jobs.put(n) ⓫
    for _ in range(procs):
        proc = Process(target=worker, args=(jobs, results)) ⓫
        proc.start() ⓫
        jobs.put(0) ⓫

```

❶ multiprocessing 尽量模仿 threading, 提供了 multiprocessing.SimpleQueue, 但这是一个方法, 绑定在底层 BaseContext 类的一个预定义的实例上。SimpleQueue 用于构建队列, 不能在类型提示中使用。

❷ 类型提示所需的 SimpleQueue 类在 multiprocessing.queues 中。

③ `PrimeResult` 包含要进行素数检测的数。`n` 与其他结果字段放在一起方便后面显示结果。

④ 为一种 `SimpleQueue` 创建类型别名。`main` 函数（示例 19-14）发给进程处理的数就是这个类型。

⑤ 再为另一种 `SimpleQueue` 创建类型别名。`main` 函数收集的结果是这个类型。队列中的值是一个个元组，由做素数检测的数和一个 `Result` 元组构成。

⑥ 这个函数与 `sequential.py` 中的类似。

⑦ `worker` 的参数是两个队列，一个存放要检查的数，另一个存放结果。

⑧ 这里，我使用 `0` 作为“毒药丸”，即职程结束的信号。如果 `n` 不是 `0`，则继续循环。¹⁴

¹⁴对这个示例来说，使用 `0` 作为哨符很方便。另一个常用的值是 `None`。使用 `0` 可以简化 `PrimeResult` 的类型提示和 `worker` 的代码。

⑨ 调用素数检测函数，把结果放入 `PrimeResult` 队列。

⑩ 发回一个 `PrimeResult(0, False, 0.0)`，让主循环知道这个职程结束了。

⑪ `procs` 是并行检测素数的进程数量。

⑫ 把要检查的数放入 `jobs` 队列。

⑬ 为每个职程派生一个子进程。每个子进程都有各自的 `worker` 函数，单独运行其中的循环，直到从 `jobs` 队列中获取 `0`。

⑭ 启动各个子进程。

⑮在各个进程中把 0 放入队列，终止进程。

循环、哨符和毒药丸

示例 19-13 中的 `worker` 函数遵循了并发编程中的一个常见模式：通过无限循环获取队列中的项，把各个项交给一个函数处理，执行真正的工作。当队列产生一个哨符时，循环结束。在这个模式中，停止进程的哨符通常叫作“毒药丸”。

`None` 经常用作哨符，但是如果数据流中有 `None`，那就不合适了。为了获取一个唯一值用作哨符，经常会调用 `object()`。但是，一旦跨进程就不能这么做，因为在进程间通信的 Python 对象必须序列化，经过 `pickle.dump` 和 `pickle.load` 处理之后，反序列化得到的 `object` 实例就与原实例不同了（二者的比较结果是不等）。`None` 有一个很好的替代品，即内置对象 `Ellipsis`（我们熟悉的...），因为它经过序列化之后也不失同一性。

Python 标准库使用的哨符多种多样。“PEP 661—Sentinel Values”提出了一种标准的哨符类型。截至 2021 年 9 月，这个 PEP 还处在草案阶段。

接下来分析 `procs.py` 中的 `main` 函数，如示例 19-14 所示。

示例 19-14 `procs.py`：素数检测多进程版本的 `main` 函数

```
def main() -> None:
    if len(sys.argv) < 2: ❶
        procs = cpu_count()
    else:
        procs = int(sys.argv[1])
    print(f'Checking {len(NUMBERS)} numbers with {procs}
processes:')
    t0 = perf_counter()
    jobs: JobQueue = SimpleQueue() ❷
    results: ResultQueue = SimpleQueue()
    start_jobs(procs, jobs, results) ❸
```

```

        checked = report(procs, results) ❹
        elapsed = perf_counter() - t0
        print(f'{checked} checks in {elapsed:.2f}s') ❺

def report(procs: int, results: ResultQueue) -> int: ❻
    checked = 0
    procs_done = 0
    while procs_done < procs: ❼
        n, prime, elapsed = results.get() ❽
        if n == 0: ❾
            procs_done += 1
        else:
            checked += 1 ❿
            label = 'P' if prime else ' '
            print(f'{n:16} {label} {elapsed:9.6f}s')
    return checked

if __name__ == '__main__':
    main()

```

- ❶ 如未提供命令行参数，则把进程数设为 CPU 核数；否则，创建第一个参数指定数量的进程。
- ❷ `jobs` 和 `results` 是示例 19-13 定义的队列。
- ❸ 启动 `proc` 进程，处理 `jobs` 中的作业，结果放入 `results`。
- ❹ 获取并显示结果。`report` 在❻处定义。
- ❺ 显示检测了多少个数，以及总用时。
- ❻ 参数是 `procs` 的数量和存放结果的队列。
- ❼ 在所有进程都结束之前一直循环。
- ❽ 获取一个 `PrimeResult`。在队列上调用 `.get()`，在队列中有项之前一直阻塞。也可以让这个操作不阻塞，或者设置暂停时间。详见 `SimpleQueue.get` 文档。

⑨ 如果 `n` 等于零，那就说明一个进程退出了，递增 `procs_done` 计数。

⑩ 否则，递增 `checked` 计数（记录检测了多少个数），并显示结果。

结果的返回顺序与提交作业的顺序不同，所以我在每一个 `PrimeResult` 元组中都放一个 `n`。如若不然，哪个结果对应哪个数便无从得知。

如果主进程在全部子进程结束之前退出了，那么你会在调用回溯中看到令人困惑的 `FileNotFoundError` 异常（由 `multiprocessing` 的一个内部锁导致）。调试并发代码本来就难，调试 `multiprocessing` 更是难上加难，因为所有复杂的细节都隐藏在看似线程的表象之下。幸好，第 20 章将要讲到的 `ProcessPoolExecutor` 用法更简单，也更稳健。



在本书抢读版中，示例 19-14 中有一处条件竞争，感谢读者 Michael Albert 指出。条件竞争是一种 bug，它出现与否取决于并发执行单元执行操作的顺序。如果“A”在“B”之前发生，则一切正常；一旦“B”先发生，就会出错。

如果你对这个 bug 感到好奇，可以查看代码差异，顺便了解我是如何修正的：[example-code-2e/commit/2c123057](https://github.com/example-code-2e/commit/2c123057)。不过请注意，后来我又做了重构，把 `main` 的部分职责委托给了 `start_jobs` 和 `report` 函数。那个目录中有一个 `README.md` 文件，对这个问题和解决方案做了说明。

19.6.4 实验：进程数多一些或少一些

可以运行一下 `procs.py`，通过参数设置不同数量的工作进程，如下所示。

```
$ python3 procs.py 2
```

以上命令启动两个工作进程，得到结果的运行时间大概是 sequential.py 的一半——前提是你的设备至少有双核 CPU，而且没有忙于运行其他程序。

我分别使用 1 到 20 个进程运行 procs.py，每个进程数运行 12 次，共计 240 次。对于每个进程数，计算 12 次运行时间的中位值，基于此绘制了图 19-2。

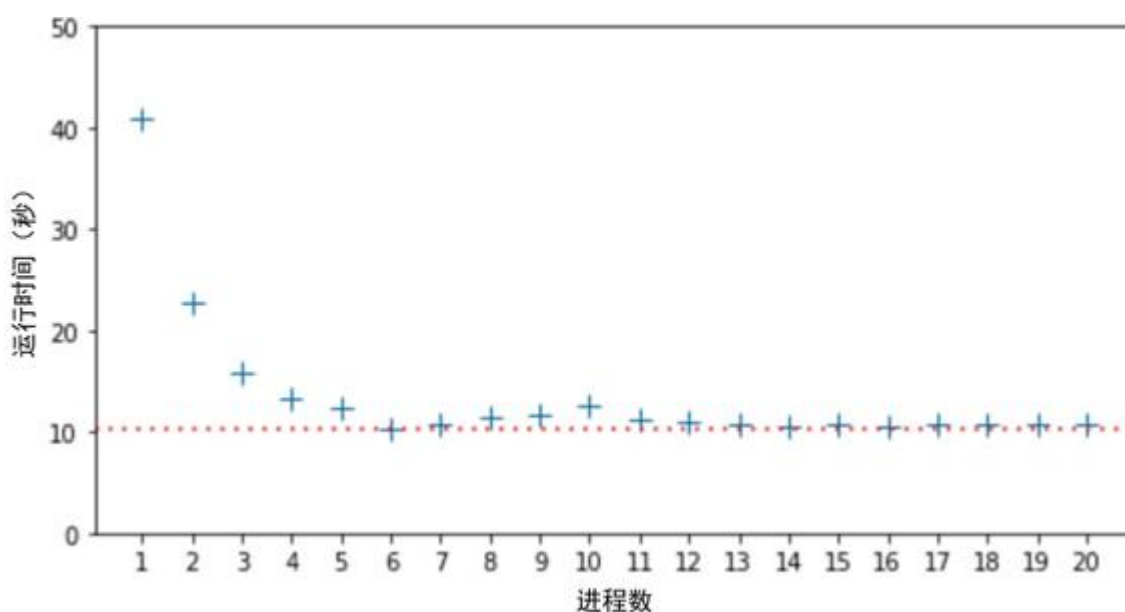


图 19-2：分别使用 1 到 20 个进程运行，得到对应的运行时间中位值。使用 1 个进程运行的中位值最大，为 40.81 秒。使用 6 个进程运行的中位值最小，为 10.39 秒（图中的虚线）

在我这台 6 核笔记本电脑中，使用 6 个进程的运行时间中位值最小，为 10.39 秒，如图 19-2 中的虚线所示。我原以为超过 6 个进程后运行时间会增加，因为涉及 CPU 争用，但是在 10 个进程时达到最大值 12.51 秒之后，从 11 个进程开始性能又提升了，而 13 到 20 个进程基本就保持稳定了，运行时间中位值只比 6 个进程的最小值高一点——这是我没想到的，也说不清具体原因。

19.6.5 基于线程的方案并不可靠

我还编写了 `threads.py`，把 `procs.py` 中的 `multiprocessing` 换成了 `threading`。这两个版本的代码几乎一样——对于简单的示例，在两个 API 之间转换，大都如此。由于存在 GIL，而且 `is_prime` 是计算密集型函数，所以线程版比示例 19-12 中的顺序执行版更慢，而且随着线程数量的增加，速度还会变慢，因为涉及 CPU 争用，以及上下文切换的开销。为了切换到新线程，操作系统需要保存 CPU 寄存器，还要更新程序计数器和栈指针，这个过程中的一些副作用开销较大，例如让 CPU 缓存失效，甚至交换内存页。

接下来的第 20 章和第 21 章会继续探讨 Python 并发编程：第 20 章使用高级的 `concurrent.futures` 库管理线程和进程，第 21 章使用 `asyncio` 库进行异步编程。

本章余下几节旨在回答以下问题：

考虑到目前讨论的这么多局限，Python 为何还能在多核世界蓬勃发展？

19.7 多核世界中的 Python

Herb Sutter 写的“The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”一文经常被人引用。下面这段话就摘自那篇文章。

从 Intel 和 AMD 到 Sparc 和 PowerPC，主要的处理器制造商和架构为了提高 CPU 性能已经用尽了大多数传统方法。它们不再设法提升时钟速度和线性指令吞吐量，而是全部转向超线程和多核架构。（2005 年 3 月，可在线阅读。）

Sutter 所说的“免费午餐”是指不用开发人员插手，软件速度就能提升的趋势，因为 CPU 执行顺序代码的速度一年比一年快。从 2004 年开

始，这个趋势一去不回：时钟速度和执行优化进入平台期，如今任何显著的性能提升都只能凭借多核或超线程技术，而且只有并发执行的代码能从中受益。

Python 的故事始于 20 世纪 90 年代初，那时 CPU 顺序执行代码的速度仍呈指数级增长。在那个年代，除了超级计算机领域之外，几乎没人讨论多核 CPU。当时，引入 GIL 的决定理所当然。有了 GIL，解释器在单核上运行的速度更快，而且实现也更简单。¹⁵ 在 GIL 的帮助下，通过 Python/C API 编写简单的扩展也更容易。

¹⁵ Ruby 语言的创造者 Yukihiro Matsumoto 在他的解释器中使用 GIL，可能也是出于同样的原因。



我特意强调了“简单的扩展”，因为不是所有扩展都需要处理 GIL。使用 C 语言或 Fortran 编写的函数，运行速度可能比用 Python 编写的同等函数快数百倍。¹⁶ 因此，许多情况下，可能不需要大费周章释放 GIL，利用多核 CPU。所以，我们应该感谢 GIL，正因为得益于 GIL，我们才有数量众多的 Python 扩展可用——显然，这是 Python 语言现今如此流行的关键原因之一。

¹⁶ 上大学时，有一项作业是让我们用 C 语言实现 LZW 压缩算法。不过，一开始我是用 Python 实现的，目的是检验我对规范的理解。后来我用 C 语言实现的版本，速度快了约 900 倍。

尽管存在 GIL，但是 Python 在需要并发或并行执行的应用领域依然能够蓬勃发展，这要归功于努力解决 CPython 局限性的库和软件架构。

接下来，我们将探讨在当今这个多核 CPU 与分布式计算的年代，Python 在系统管理、数据科学和服务端应用程序开发等领域如何发挥作用。

19.7.1 系统管理

Python 广泛用于管理大型服务器集群、路由器、负载均衡程序和网络接入存储（network- attached storage, NAS）。Python 也是软件定义网络（software-defined networking, SDN）的首选语言。主流云服务供应商都提供 Python 库和相关教程，一些是供应商自己制作的，还有一些是 Python 社区用户制作的。

在这个领域中，人们使用 Python 脚本发送命令，由远程设备执行，实现配置任务自动化。由于很少涉及 CPU 密集型操作，因此使用线程或协程即可。比如，第 20 章将要讲解的 `concurrent.futures` 包就可以同时多台远程设备中执行相同的操作，复杂性并没有增加多少。

在标准库之外还有一些流行的 Python 项目可用于管理服务器集群，例如 Ansible 和 Salt 等工具，以及 Fabric 等库。

支持协程和 `asyncio` 的系统管理库也在不断增加。2016 年，Facebook 的生产工程团队报告指出：“我们越来越依赖 Python 3.4 引入的 `AsyncIO`，原先的 Python 2 代码基迁移到新版之后，性能得到了巨大提升。”

19.7.2 数据科学

Python 可以很好地服务于数据科学（包括人工智能）和科学计算。这些领域中的应用程序属于计算密集型，但是 Python 用户受益于用 C、C++、Fortran、Cython 等语言编写的众多数值计算库，其中许多能够利用多核设备、GPU 和（或）异构集群中的分布式并行计算。

截至 2021 年，Python 数据科学生态系统有大量优秀的工具，下面简单介绍其中几个。

Jupyter 项目

提供两个基于浏览器的界面（Jupyter Notebook 和 JupyterLab），供用户运行和记录可在远程计算机上通过网络运行的

分析代码。二者都是 Python 和 JavaScript 混合型应用程序，支持用不同语言编写的计算内核，然后通过 ZeroMQ（供分布式应用程序使用的异步消息库）集成。“Jupyter”这个名称源自 Notebook 最先支持的 3 门语言：Julia、Python 和 R。构建在 Jupyter 工具之上的生态系统丰富多样，其中一个 Bokeh。这是一个强大的交互式可视化库，凭借现代化的 JavaScript 引擎和浏览器的高强性能，用户可以浏览大型数据集或不断更新的数据流，并与之交互。

TensorFlow 和 PyTorch

根据 O'Reilly 在 2021 年 1 月发布的学习资源使用报告，在过去的 2020 年，这是最受关注的两个深度学习框架。这两个项目都是用 C++ 编写的，都能利用多个核、GPU 和集群。它们支持的语言很多，但重点支持的是 Python，大多数用户使用的也是 Python。TensorFlow 由 Google 开发，供内部使用；PyTorch 由 Facebook 开发。

Dask

这是一个并行计算库，可以把工作交托给本地进程或设备集群。Dask 主页上说，Dask“在一些全球最大的超级计算机中做过测试”。Dask 提供的 API 非常接近 NumPy、pandas 和 scikit-learn——当今数据科学和机器学习领域最流行的库。Dask 可以在 JupyterLab 或 Jupyter Notebook 中使用，利用 Bokeh 不仅可以实现数据可视化，而且还可以实现交互式仪表板，以接近实时的方式显示数据流和跨进程、跨设备的计算。Dask 的功能令人印象深刻，建议你观看 Matthew Rocklin（该项目的维护者之一）录制的 15 分钟演示视频。Rocklin 在视频中利用分布在 8 台 AWS EC2 设备中的 64 个核使用 Dask 处理数据。

以上只是一些示例，说明了数据科学社区如何充分利用 Python 制定解决方案，以克服 CPython 运行时的局限性。

19.7.3 服务器端 Web 和移动开发

Python 广泛用于 Web 应用程序开发和为移动应用程序提供支持的后端 API 开发。Google、YouTube、Dropbox、Instagram、Quora 和 Reddit 等公司是如何成功构建 Python 服务器端应用程序，全天候为数亿用户提供服务的呢？同样，答案远远超出 Python 提供的“开箱即用”功能。

在讨论支持 Python 弹性伸缩的工具之前，我必须引用 ThoughtWorks 《技术雷达》中的一段忠告。

对高性能的渴望、对 Web-Scale¹⁷ 的渴望

我们看到许多团队因为选择了复杂的工具、框架或架构而遇到麻烦，因为他们觉得“可能需要伸缩”。像 Twitter 和 Netflix 这样的公司需要支持极端负载，因此需要这些架构，但他们也有非常熟练的开发团队，能够应对由此带来的复杂性。多数时候，我们并不需要这类工程技术。团队应该克制对 Web-Scale 的渴望，选择更简单可行的方案。¹⁸

¹⁷根据 Gartner 的定义，Web-Scale 用于描述一些头部互联网公司通过应用新进程、新架构和新实践所实现的灵活性和扩展性。简而言之，Web-Scale 是一种处理大量数据和流量的架构。——编者注

¹⁸ Thoughtworks 技术顾问委员会，《技术雷达》2015 年 11 月刊。

Web-Scale 的关键是一个支持横向伸缩的架构。在这样的架构中，所有系统都是分布式系统，没有任何一门语言能够包揽全部解决方案。

分布式系统是当前学术研究的一个领域，但幸运的是，一些从业者以扎实的研究和实践经验为基础，撰写了多本通俗易懂的图书。其中一位是《数据密集型应用系统设计》的作者 Martin Kleppmann。

Kleppmann 的这本书中有多幅架构图，图 19-3 是其中第一幅。图中有部分组件与 Python 相关，有一些我用过，还有一些我有所了解。

- 应用缓存：¹⁹memcached、Redis、Varnish。

- 关系数据库：PostgreSQL、MySQL。
- 文档数据库：Apache CouchDB、MongoDB。
- 全文索引：Elasticsearch、Apache Solr。
- 消息队列：RabbitMQ、Redis。

¹⁹ HTTP 缓存（由应用代码直接使用）与应用缓存不同，用于伺服图像、CSS 和 JavaScript 文件等静态资源。如果加到图 19-3 中的话，那么 HTTP 缓存应该放在顶部靠近边缘处。内容分发网络（Content Delivery Network，CDN）也是一种 HTTP 缓存，部署在靠近应用用户的数据中心内。

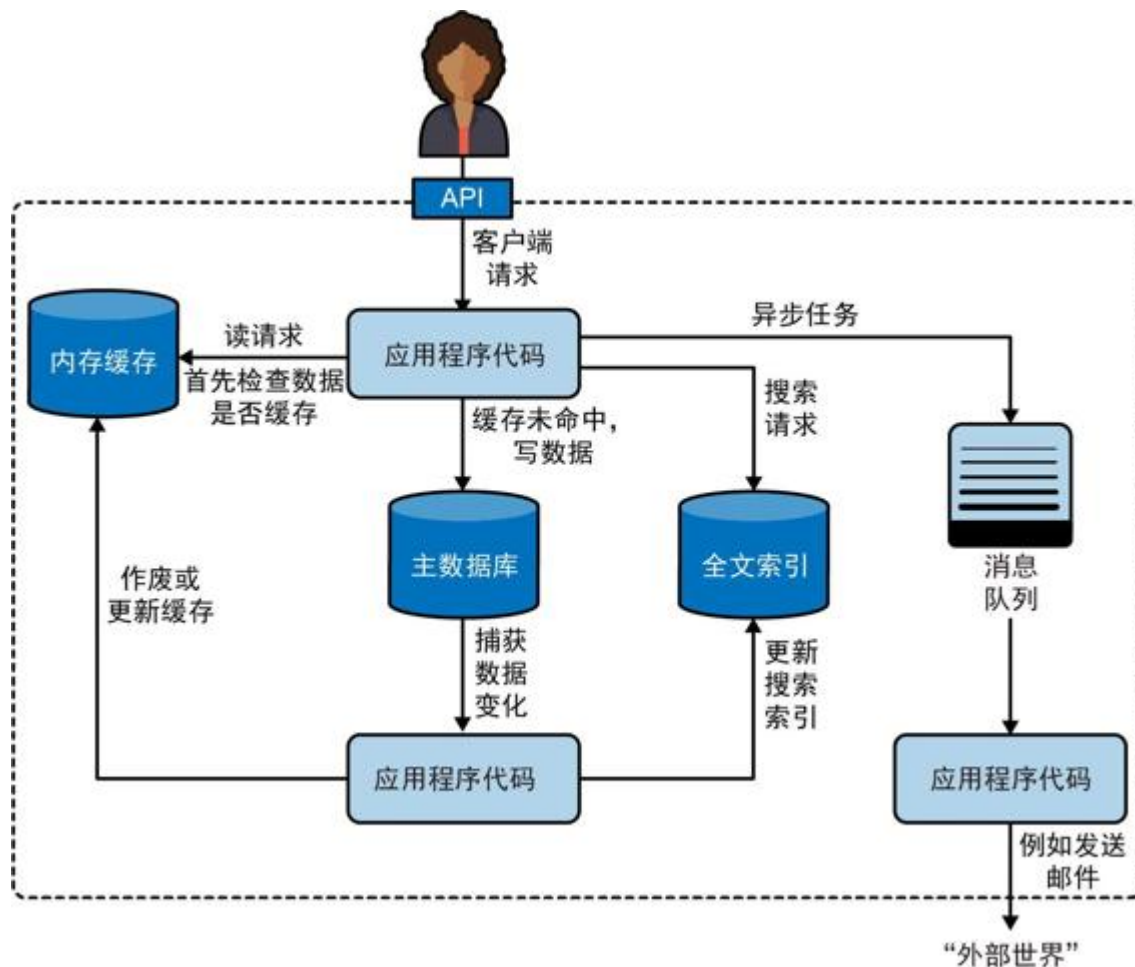


图 19-3：由多个组件构成的系统可能采用的一种架构²⁰

²⁰ 这幅架构图改编自《数据密集型应用系统设计》（Martin Kleppmann 著）一书中的图 1-1。

在这些类别中，还有其他工业级开源产品。主流云供应商也有自己的专属替代方案。

Kleppmann 给出的图是通用的，与具体语言无关——那本书的内容也是如此。Python 服务器端应用程序通常部署如下两个特定的组件。

- 一个应用程序服务器，在 Python 应用程序的多个实例之间分配负载。应用程序服务器应放在图 19-3 靠近顶部的位置，先于应用程序代码处理客户端请求。
- 一个任务队列，包装图 19-3 右侧的消息队列，提供更易于使用的高级 API，为设备中运行的进程分配任务。

这两个组件是 Python 服务器端部署推荐的最佳实践，接下来的 19.7.4 节和 19.7.5 节将深入探讨。

19.7.4 WSGI 应用程序服务器

WSGI (Web Server Gateway Interface, Web 服务器网关接口) 是 Python 框架或应用程序接收 HTTP 服务器请求并向其发送响应的标准 API。²¹WSGI 应用程序服务器管理一个或多个进程，运行你的应用程序，借此最大限度地利用 CPU。

²¹有些人一个字母一个字母地说出 WSGI，另一些人则把它当成一个单词拼读，音似“whisky”。

图 19-4 是典型的 WSGI 部署。



如果想把图 19-3 和图 19-4 这两幅图合二为一，那么图 19-4 虚线框中的内容应放在图 19-3 顶部“应用程序代码”框处。

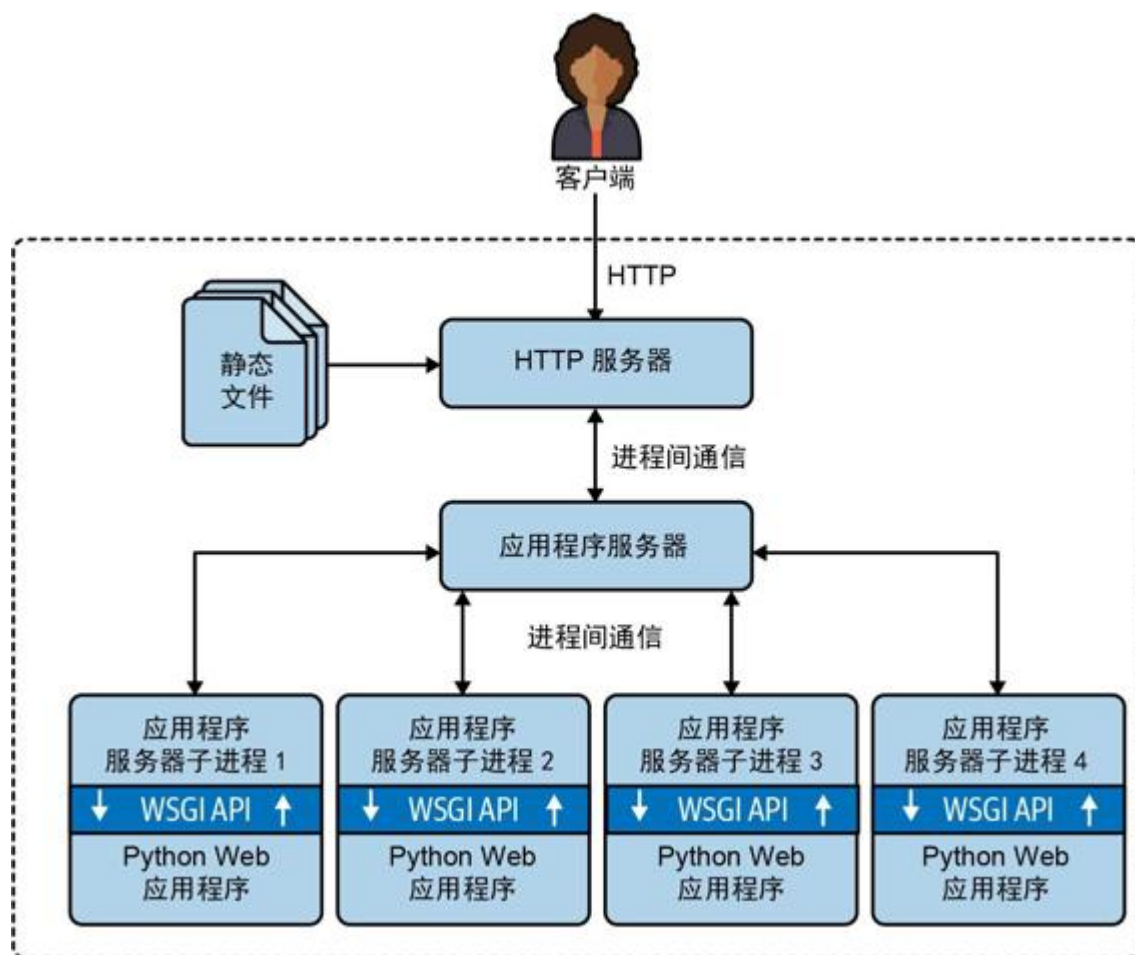


图 19-4：客户端连接 HTTP 服务器，后者负责分发静态文件，并把其他请求转发给应用服务器；应用程序服务器派生子进程，利用多个 CPU 核运行应用程序代码。WSGI API 是应用服务器与 Python 应用程序代码之间的黏合剂

Python Web 项目最常使用以下应用程序服务器。

- `mod_wsgi`
- `uWSGI`²²
- `Gunicorn`
- `NGINX Unit`

²²拼写时，`uWSGI` 开头是小写英文字母“u”，但是按照希腊字母“μ”发音，因此整个名称可以读作“micro-whisky”，“k”发“g”音。

如果你使用 Apache HTTP 服务器，那么 `mod_wsgi` 是最佳选择。`mod_wsgi` 与 WSGI 一样古老，不过一直有人积极维护，近期还提供了命令行启动程序，名为 `mod_wsgi-express`，配置变得更简单，而且更适合在 Docker 容器中使用。

据我所知，新开发的项目大都首选 uWSGI 和 Gunicorn。二者经常搭配 NGINX HTTP 服务器使用。除了本职工作，uWSGI 还提供了大量额外功能，例如应用缓存、任务队列、类似 cron 的定期任务等。另一方面，uWSGI 的配置难度比 Gunicorn 大很多。²³

²³ Bloomberg 的工程师 Peter Sperl 和 Ben Green 写的“Configuring uWSGI for Production Deployment”一文，揭露了 uWSGI 的很多默认设置并不适合一般的部署场景。Sperl 在 EuroPython 2019 所做的演讲总结了推荐设置，强烈建议 uWSGI 用户采纳。

2018 年发布的 NGINX Unit 是知名 HTTP 服务器和反向代理制造商 NGINX 新推出的产品。

`mod_wsgi` 和 Gunicorn 仅支持 Python Web 应用程序，而 uWSGI 和 NGINX Unit 还支持其他语言。详见各个工具的文档。

我主要是想告诉你，这些应用程序服务器全都可以派生多个 Python 进程，运行使用 Django、Flask、Pyramid 等框架编写的传统 Web 应用程序（老旧的顺序执行的代码），充分利用服务器的所有 CPU 核。这也解释了为什么 Python Web 开发人员无须学习 `threading`、`multiprocessing` 或 `asyncio` 模块就能找到工作：应用程序服务器在无形中已经处理好了并发。



异步服务器网关接口 (ASGI)

WSGI API 是同步的，不支持使用 `async/await` 创建协程，而在 Python 中，协程是实现 WebSocket 或 HTTP 长轮询最有效的方式。对此，接任 WSGI 的 ASGI (Asynchronous Server Gateway Interface) 规范应运而生。ASGI 为 aiohttp、Sanic、

FastAPI 等 Python 异步 Web 框架而设计，Django 和 Flask 等传统框架也可以借此逐步增加异步功能。

接下来介绍可以绕过 GIL，让服务器端 Python 应用程序实现高性能的另一种方式。

19.7.5 分布式任务队列

应用程序服务器把请求分发给运行应用程序代码的某个 Python 进程之后，应用需要快速响应，因为你希望进程能尽快完成工作，开始处理下一个请求。然而，有些请求发起的操作要用较长的时间才能处理完毕，例如发送邮件或生成 PDF。这就是分布式任务队列所要解决的问题。

在提供 Python API 的开源任务队列中，Celery 和 RQ 最出名。云供应商也会提供专属的任务队列。

这些产品对消息队列进行包装，提供高级 API，把任务委托给可能运行在不同设备中的职程。



在任务队列的语境下，不使用传统的客户端和服务端术语，而是使用“生产者”和“消费者”这两个词。例如，Django 视图处理程序**生产**作业请求，将其放入队列，供一个或多个 PDF 渲染进程**消费**。

任务队列的用途很多，下面直接引用 Celery 的 FAQ。

- 运行后台作业。例如，尽快完成 Web 请求，然后增量更新用户页面。这会给用户留下性能好和“迅捷”的印象，而实际工作可能需要一段时间才能结束。
- 在 Web 请求完成后执行某些操作。
- 异步执行某些操作，失败后重试，确保操作一定完成。
- 安排定期工作。

除了解决这些显而易见的问题之外，任务队列还支持横向伸缩。生产者和消费者得到解耦：生产者不直接指挥消费者，而是把请求放入队列；消费者不需要知道关于生产者的任何信息（如果需要知道，可以使用请求中关于生产者的信息）。至关重要的一点是，随着需求增长，你可以轻松地添加更多进程来执行任务。这就是 Celery 和 RQ 被称为分布式任务队列的原因。

回想一下，前面那个简单的 `procs.py`（见示例 19-13）使用了两个队列：一个用于作业请求，另一个用于收集结果。Celery 和 RQ 的分布式架构使用了类似的模式。二者都支持使用 NoSQL 数据库 Redis 作为消息队列和结果存储器。Celery 还支持其他消息队列，例如 RabbitMQ 或 Amazon SQS，也支持使用其他数据库存储结果。

我们对 Python 并发模型的介绍到此结束。接下来的第 20 章和第 21 章将继续这个话题，着重讲解标准库中的 `concurrent.futures` 和 `asyncio` 包。

19.8 本章小结

本章首先介绍了一些理论知识，然后使用 Python 的 3 种原生并发编程模型实现了旋转指针脚本：

- 使用 `threading` 包实现线程版；
- 使用 `multiprocessing` 包实现进程版；
- 使用 `asyncio` 包实现异步协程版。

接下来，我们通过一个实验探讨了 GIL 的真正影响：把旋转指针动画示例换成了素数检测的示例，观察由此产生的行为。经过比较后发现，`asyncio` 不善于处理 CPU 密集型函数，因为它们会阻塞事件循环。尽管存在 GIL，但是该实验的线程版能顺利完成工作，这是因为 Python 定期中断线程，而且该示例只用到两个线程，一个执行计算密集型工作，另一个每秒仅更新动画 10 次。`multiprocessing` 版本能绕开 GIL，启动一个新进程处理动画，而主线程负责素数检测。

随后是一个计算素数的示例，凸显了 `multiprocessing` 和 `threading` 之间的区别，也证实了只有进程才能让 Python 从多核 CPU 中受益。对于繁重的计算，受 Python 的 GIL 限制，线程版比顺序执行版的表现还要糟糕。

讨论 Python 并发和并行计算，一定绕不开 GIL，但是不应过于高估它的影响。这正是 19.7 节想表达的观点。例如，在系统管理中使用 Python，很多时候并不受 GIL 的影响。另一方面，数据科学和服务端开发社区已经使用工业级解决方案绕开了 GIL，可以满足特定的需求。19.7.4 节和 19.7.5 节分别提到了支持 Python 服务器端应用程序弹性伸缩的两个常用工具：WSGI 应用服务器和分布式任务队列。

19.9 延伸阅读

本章延伸阅读材料较多，分为几节单列。

19.9.1 使用线程和进程实现并发

第 20 章将要讲解的 `concurrent.futures` 库背后涉及线程、进程、锁和队列，这些对你而言都是隐藏的，由高度抽象的 `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 负责处理。如果你想进一步学习使用这些底层对象进行并发编程，建议先阅读 Jim Anderson 写的“An Intro to Threading in Python”一文。《Python 3 标准库》（Doug Hellmann 著）一书中的“使用进程、线程和协程提供并发性”一章也有相关内容。

《Effective Python：编写高质量 Python 代码的 90 个有效方法》（Brett Slatkin 著）、《Python 参考手册（第 4 版）》（David Beazley 著）和《Python 技术手册（第 3 版）》（Alex Martelli 等人著）虽是一般性 Python 参考资料，但是对 `threading` 和 `multiprocessing` 也有大量篇幅着墨。`multiprocessing` 官方文档内容丰富，在“Programming guidelines”一节给出了实践建议。

Jesse Noller 和 Richard Oudkerk 起草的“PEP 371—Addition of the multiprocessing package to the standard library”把 `multiprocessing` 包加入了标准库。这个包的官方文档在一个大小为 93KB 的 `.rst` 文件中，大约 63 页，是 Python 标准库文档中最长的章节之一。

High Performance Python, 2nd ed. (Micha Gorelick 和 Ian Ozsvald 著) 有一章是关于 `multiprocessing` 的，其中有一个素数检测的示例，使用的策略与本章的 `procs.py` 示例不同：对于每个数，把 2 和 `sqrt(n)` 之间的因数分成几组，在各个线程中分别迭代。这种“各个击破”的方式在科学计算应用程序中经常使用，因为涉及的数据集往往规模巨大，而且工作站（或集群）的 CPU 核比一般用户的设备多。服务器端系统处理来自很多用户的请求，让每个进程从头到尾处理一个计算会更简单、更高效，也可以减少进程间通信和协调的开销。除了 `multiprocessing`，Gorelick 和 Ozsvald 还提出了许多其他用于开发和部署高性能数据科学应用的方式，以便利用多个核、GPU、集群、分析器，以及 Cython 和 Numba 等编译器。最后一章“现场教训”汇集了许多其他 Python 高性能计算从业者提供的简短案例分析，不容错过。

《Python 进阶：实际应用开发实战》(Matthew Wilkes 著) 是难得的一本好书，通过简短的示例解释理论概念，还带领读者构建了一个具体应用程序，可在生产环境使用。这个程序是一个数据聚合器，类似于 DevOps 监控系统或 IoT 分布式传感器的数据收集器。这本书中有两章涵盖 `threading` 和 `asyncio` 并发编程。

Parallel Programming with Python (Jan Palach 著) 一书解读了并发和并行背后的核心概念，涵盖 Python 标准库和 Celery。

Using Asyncio in Python (Caleb Hattingh 著) 一书的第 2 章题为“The Truth About Threads”（线程的真相）。²⁴ 那一章讨论了线程的优缺点（引用了权威资料，令人信服），明确指出线程的基本问题

与 Python 或 GIL 没有任何关系。以下内容摘自 *Using Asyncio in Python* 一书的第 14 页。

²⁴ Caleb 是本书第 2 版的技术审校之一。

我想强调的是：

- 使用线程的代码难以理解。
- 对于大规模并发操作（上千个并发任务），线程是一种低效模型。

如果你想自己探索（工作不能丢啊），了解线程和锁的理解难度，可以尝试 *The Little Book of Semaphores*（Allen Downey 著）一书中的练习。这本书中的练习，有的容易，有的非常难，有的甚至无解；不过，即便是那些简单的题目也能让你大开眼界。

19.9.2 GIL

如果你对 GIL 感兴趣，请记住一点：我们在 Python 代码中无法控制 GIL。关于 GIL 的权威资料是 C-API 文档：*Thread State and the Global Interpreter Lock*。Python Library and Extension FAQ 页面回答了“Can't we get rid of the Global Interpreter Lock?”这一问题。Guido van Rossum 写的“It isn't Easy to Remove the GIL”一文和 Jesse Noller（`multiprocessing` 包的贡献者）写的“Python Threads and the Global Interpreter Lock”一文也不容错过。

CPython Internals（Anthony Shaw 著）一书从 C 语言编程层面解读了 CPython 3 解释器的实现。这本书中“Parallelism and Concurrency”（并行和并发）一章篇幅最长，深入剖析了 Python 对线程和进程的原生支持，还讲解了在扩展中如何使用 C/Python API 管理 GIL。

最后，David Beazley 在“Understanding the Python GIL”演讲中对 GIL 做了详细探讨。²⁵ 在第 54 张幻灯片中，Beazley 指出，在一项

特殊的基准测试中，引入新 GIL 算法之后，Python 3.2 的处理时间反而上升了。在 Beazley 提交的 7946 号工单中，Antoine Pitrou（实现了新 GIL 算法）在一条评论中指出，这个问题在真实的工作负载下并不明显。

²⁵感谢 Lucas Bruniati 给我发送这次演讲的资源。

19.9.3 标准库之外的并发世界

本书专注于核心语言功能和标准库的核心部分。*Full Stack Python* 一书对本书内容是一个很好的补充，涉及整个 Python 生态系统，涵盖开发环境、数据处理、Web 开发和 DevOps 等。

前面提到的 *High Performance Python, 2nd ed.* 和 *Parallel Programming with Python* 这两本书不仅涵盖如何使用 Python 标准库进行并发编程，还用大量篇幅讲解了第三方库和工具。*Distributed Computing with Python*（Francesco Pierfederici 著）一书不仅涵盖标准库，还说明了如何使用云供应商和 HPC（High-Performance Computing，高性能计算）集群。

Matthew Rocklin 于 2019 年 6 月发布的“Python, Performance, and GPUs”一文介绍了“Python 使用 GPU 加速器的新情况”。

“Instagram 部署的 Django Web 框架（全部使用 Python 编写）规模，目前世界第一。”这是 Min Ni（Instagram 软件工程师）所写的“Web Service Efficiency at Instagram with Python”一文的开篇第一句话。这篇文章介绍了 Instagram 用来优化 Python 基准代码效率的指标和工具，以及在“每天部署 30 ~ 50 次”后端的情况下，用来检测和诊断性能衰退的指标和工具。

Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices（Harry Percival 和 Bob Gregory 著）一书讲解了 Python 服务器端应用程序的架构模式。

João S. O. Bueno 开发的 `lelo` 和 Nat Pryce 开发的 `python-parallelize` 这两个库易于使用，通过进程并行处理任务，方式优雅。`lelo` 包定义的 `@parallel` 装饰器可用在任何函数上，并在另一个进程中执行函数，这样函数就不阻塞了，像变魔术一样。`python-parallelize` 包提供的 `parallelize` 生成器把 `for` 循环分配给多个 CPU 执行。这两个包底层使用的都是 `multiprocessing` 库。

Python 核心开发人员 Eric Snow 维护着 Multicore Python 维基页面，记录了他和其他人为改进 Python 对并行执行的支持所做的努力。Snow 是“PEP 554—Multiple Interpreters in the Stdlib”的作者。如果获批并实现，PEP 554 将为未来的功能增强奠定基础，最终不借助有额外开销的 `multiprocessing` 库就可以让 Python 使用多个核。目前最大的障碍之一是，如何解决多个活动的子解释器和假定只有一个解释器的扩展之间复杂的交互。

另一位 Python 维护人员 Mark Shannon 创建了一个有用的表格，对 Python 的并发模型做了比较。在 `python-dev` 邮件列表中，他与 Eric Snow 和其他开发人员讨论子解释器时引用了这个表。表中的“`Ideal CSP`”列指的是 Tony Hoare 于 1978 年提出的“通信顺序进程”（Communicating sequential processes）理论模型。Go 语言也支持共享对象，这违反了 CSP 的一个基本约束：执行单元应该通过通道传递消息进行通信。

Stackless Python（简称 Stackless）是 CPython 的一个分支，实现了微线程（microthread）——这是应用程序级轻量级线程，与操作系统线程不是一回事。大型多人在线游戏《星战前夜》就构建在 Stackless 之上。有一段时间，Stackless 是由开发这款游戏的 CCP 公司的几位工程师维护的。Stackless 的某些功能后来使用 Pypy 解释器和 `greenlet` 包重新实现了。`greenlet` 包是网络库 `gevent` 的核心，`gevent` 库又是应用程序服务器 `Gunicorn` 的基础。

并发编程的**参与者模型**（actor model）是高度可伸缩的 Erlang 和 Elixir 语言的核心，也是 Scala 和 Java 的 Akka 框架的模型。如果你

想在 Python 中尝试参与者模型，可以了解一下 Thespian 库和 Pykka 库。

下面推荐的资料很少或者根本没有提及 Python，但是，如果你对本章的话题感兴趣，那还是有一定关联的。

19.9.4 Python 之外的并发和伸缩世界

《RabbitMQ 实战：高效部署分布式消息队列》（Alvaro Videla 和 Jason J. W. Williams 著）一书对 RabbitMQ 和高级消息队列协议（Advanced Message Queuing Protocol, AMQP）的介绍精彩绝伦，提供的示例涵盖 Python、PHP 和 Ruby。无论你使用什么技术栈，即便在 RabbitMQ 背后使用 Celery，我都建议你阅读这本书。这本书涵盖分布式消息队列相关的概念、动机和模式，以及如何操作和调校 RabbitMQ，实现弹性伸缩。

我从《七周七并发模型》（Paul Butcher 著）一书中学到了很多知识。这本书的副标题一针见血：理不顺的线程（When Threads Unravel）。该书第 1 章介绍了核心概念，以及 Java 使用线程和锁进行编程面对的挑战。²⁶ 余下 6 章分别讲解作者认为更好的并发和并行编程替代方案，涉及不同的语言、工具和库，示例涵盖 Java、Clojure、Elixir 和 C 语言（讨论使用 OpenCL 框架进行并行编程那一章）。CSP 模型以 Clojure 代码为例说明，不过这种方式的推广离不开 Go 语言的功劳。参与者模型使用 Elixir 语言举例说明。该书作者对讨论参与者模型那一章做了增订，换成了 Scala 语言和 Akka 框架。除非你对 Scala 已有了解，否则还是建议你学习更简单的 Elixir，这门语言更适合用于试验参与者模型和 Erlang/OTP 分布式系统平台。

²⁶ Python 中 `threading` 和 `concurrent.futures` 两个模块的 API 深受 Java 标准库的影响。

Thoughtworks 的 Unmesh Joshi 在 Martin Fowler 的博客中发表了“Patterns of Distributed Systems”（分布式系统模式）系列文章，

索引页对这个话题做了很好的归纳，给出了各个模式的链接。Joshi 一直在增加模式，现有的模式也都是在关键任务系统中多年辛苦积累的经验之精华。

《数据密集型应用系统设计》（Martin Kleppmann 著）是宝贵难得的一本书，作者是一位具有深厚行业经验和高深学术背景的从业者。Kleppmann 先后在 LinkedIn 和两家创业公司从事大规模数据基础设施方面的工作，后来到剑桥大学研究分布式系统。Kleppmann 在书中的每一章末尾给出了大量参考文献，其中不乏最新的研究成果。这本书中还有许多深具启发性的图表和精美的概念图。

我有幸参加了 Francesco Cesarini 在 OSCON 2016 上举办的可靠分布式系统架构讲习班，讲座题为“Designing and architecting for scalability with Erlang/OTP”（O'Reilly 学习平台中有视频）。别被标题迷惑了，Cesarini 在视频的 9:35 处解释道：

我要讲的内容很少是专门针对 Erlang 的.....但是你要知道，Erlang 可以消除许多附带的难题，使系统具有弹性、永不崩溃，而且可伸缩。因此，如果你熟悉 Erlang 或者运行在 Erlang 虚拟机上的其他语言，那么你就会更容易理解我要讲的内容。

这次讲习班基于《高伸缩性系统：Erlang/OTP 大型分布式容错设计》（Francesco Cesarini 和 Steve Vinoski 著）一书的最后 4 章。

开发分布式系统虽然具有挑战，但也令人振奋。不过之前讲过，不要一味追求 Web-Scale。“KISS 原则”始终是软件工程不可遗忘的建议。

读一读 Frank McSherry、Michael Isard 和 Derek G. Murray 发表的论文“Scalability! But at what COST?”。几位作者指出，学术研讨会上提出的并行图形处理系统需要数百个核才能胜过“合格的单线程实现”。他们还发现，有些系统“在记录的所有配置中，性能都比不上一个线程”。

这些发现让我想起了黑客经常说的一句嘲讽话：

我的 Perl 脚本比你的 Hadoop 集群还快。

杂谈

画地为牢，规行矩止

我在一台 TI-58 计算器上学会了编程。TI-58 计算器使用的“语言”类似于汇编语言。在那个层面上，所有“变量”都是全局的，而且没有奢侈的结构化控制流语句。你要根据条件跳转，依据 CPU 寄存器或标志的值，使用跳转指令直接跳到某处（当前位置前面或后面）执行。

基本上，你在汇编语言中可以做任何事情，这就是挑战所在：你无拘无束，随时可能犯错；一旦做出修改，维护人员也不易理解代码。

我学习的第二门语言是 8 位计算机附带的非结构化 BASIC——与很久以后出现的 Visual Basic 千差万别。BASIC 有 FOR、GOSUB 和 RETURN 语句，可是依然没有局部变量概念。GOSUB 不支持传递参数，差不多就是一种高级的 GOTO，把一个返回行数放在栈中，为 RETURN 提供跳转目标。子程序可以自行前往，获得全局数据，把结果也放在那里。其他形式的控制流只能利用 IF 和 GOTO 的组合临时实现，同样，你还是可以跳转到程序的任何一行。

使用跳转和全局变量编程几年之后，我开始学习 Pascal。我还记得，当时我努力转换思路，重新建立“结构化编程”概念。那时，我不得不在只有一个入口点的代码块周围使用控制流语句。我不能再随意跳转到任何指令。全局变量在 BASIC 中是不可避免的，但当时是一种禁忌。我需要重新考虑数据流，显式地把参数传递给函数。

我的下一个挑战是学习面向对象编程。说到底，面向对象编程是具有更多约束和多态的结构化编程。信息被隐藏起来了，我不得不重新思考应把数据存放在什么地方。我记得自己不止一次感到沮丧，因为我必须重构代码才能让我正在编写的方法获取封装在一个对象中的信息，我的方法竟然无法直接得到那些信息。

函数式编程语言又增加了其他约束，经历几十年的命令式编程和面向对象编程之后，“不可变”是最难接受的概念。然而，一旦习惯这些约束，就会发现这是一种幸事，因为写出的代码容易理解多了。

缺乏约束是基于线程和锁模型进行并发编程的主要问题。Paul Butcher 在总结《七周七并发模型》一书的第 2 章时写道：

使用这个模型最大的缺点在于无助。语言设计者很容易将其集成到某一门语言中，但对于我们这些可怜的程序员，编程语言层面并没有提供足够的帮助。

以下几点可以体现线程和锁模型不受约束。

- 线程可以共享访问任意可变的数据结构。
- 调度程序几乎可以在任何时刻中断线程，包括执行 `a += 1` 这种简单操作的过程中。在源码表达式层面，原子操作十分少见。
- 锁往往是**建议性的**（advisory）。这是一个术语，意思是在更新共享的数据结构之前，你自己必须显式持有锁。如果你忘了，万一另一个线程尽忠职守，持有锁，更新了同一份数据，那肯定会产生混乱。

相比之下，参与者模式可以实施一些约束。在参与者模式中，执行单元叫作**参与者**。²⁷

- 参与者可以有内部状态，但是不能与其他参与者分享状态。
- 参与者之间通过收发消息进行通信。

- 消息中只能存有数据的副本，不能引用可变的数据。
- 一个参与者一次只处理一个消息。对于单个参与者，没有并发执行概念。

当然，如果遵循这些规则，你可以在任何语言中采用参与者风格进行编程。你还可以在 C 语言中使用面向对象编程习惯，甚至在汇编中使用结构化编程模式。但如果要实现这些，则需要每个接触代码的人都达成一致并严于律己。

按照 Erlang 和 Elixir 的实现，参与者模型无须管理锁，因为所有数据类型都是不可变的。

线程和锁不会消失。我只是觉得，编写应用程序时，宝贵的时间应该用在内核模块或数据库上，而不能浪费在这种底层的实体上。

我永远保留改变主意的权利。但是现在，我确信参与者模型是最合理、最通用的并发编程模型。CSP (Communicating Sequential Processes, 通信顺序进程) 也是合理的，但是在 Go 语言中缺少部分约束。CSP 的思想是协程（或 Go 语言中的 goroutines）使用队列（在 Go 中叫作**通道**）交换数据和同步。但是，Go 语言也支持内存共享和锁。我看过一本关于 Go 语言的书，提倡使用共享内存和锁代替通道，美其名曰“为了性能”。真是旧习难改。

²⁷ Erlang 社区使用术语“进程”表示参与者。在 Erlang 中，每个进程都是其自身循环中的一个函数，因此进程是非常轻量级的。在一台设备中，同时激活的进程可以达到数百万个——这里的进程与本章其他部分讨论的重量级操作系统进程没有联系。这也体现了 Simon 教授所说的两个重要过错：使用不同的词表示相同的事物，以及使用同一个词表示不同的事物。

第 20 章 并发执行器

抨击线程的往往是系统开发者，他们考虑的使用场景对一般的应用程序开发者来说，也许一生都不会遇到。……应用程序开发者遇到的使用场景，99% 的情况下只需知道如何派生一堆独立的线程，然后用队列收集结果。

——Michele Simionato
深度思考 Python 的人¹

¹摘自 Michele Simionato 发表的文章“Threads, processes and concurrency in Python: some thoughts”，副标题为“Removing the hype around the multicore (non) revolution and some (hopefully) sensible comment about threads and other forms of concurrency”。

本章主要讨论实现了 `concurrent.futures.Executor` 接口的类。这些类对 Michele Simionato 所说的“派生一堆独立线程，通过队列收集结果”的模式进行了封装，使用起来特别容易，不仅能用于线程，而且还能用于进程处理计算密集型任务。

本章还会介绍“future”这一概念。这种对象表示异步执行的操作，类似于 JavaScript 中的 `promise`。这个概念的作用很大，是 `concurrent.futures` 模块和 `asyncio` 包（将在第 21 章讨论）的基础。

20.1 本章新增内容

本章原来的标题是“使用 future 处理并发”，现在改成了“并发执行器”，因为本章主要讨论执行器这个重要的高级功能。future 是底层对象，主要在 20.2.3 节探讨，本章余下内容不再涉及。

现在，HTTP 客户端示例全都换用了新的 HTTPX 库。这个库同时提供了同步和异步 API。

由于 Python 3.7 中的 `http.server` 包增加了多线程服务器，因此现在 20.5 节中实验的设置更简单了。之前，标准库中只有单线程版 `BaseHttpServer`，不适合做并发客户端实验，所以本书第 1 版只能借助外部工具。

20.3 节演示如何使用执行器简化 19.6.3 节中的代码。

最后，我把多数理论知识移到了新增的第 19 章。

20.2 并发网络下载

为了高效处理网络 I/O，势必需要并发，我们不应漫无目的地等下去，在远程设备发回响应期间，可以让应用程序做些其他的事情。²

²尤其考虑到一些云供应商按设备使用时间收费，与 CPU 使用量无关。

为了通过代码说明这一点，我编写了 3 个简单的程序，从网上下载 20 个国家的国旗图像。第一个程序 `flags.py` 依序下载，下载完一个图像，并将其保存在本地之后，才请求下一个图像。另外两个脚本并发下载，几乎同时请求所有图像，下载完一个保存一个。`flags_threadpool.py` 脚本使用 `concurrent.futures` 包，`flags_asyncio.py` 脚本使用 `asyncio` 包。

示例 20-1 是运行这 3 个脚本得到的结果，每个脚本都运行 3 次。示例 20-1 中显示的结果是运行几次之后收集的，CDN 已经建立了缓存。

示例 20-1 运行 `flags.py`、`flags_threadpool.py` 和 `flags_asyncio.py` 脚本得到的结果

```
$ python3 flags.py
BD BR CD CH DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN ❶
20 flags downloaded in 7.26s ❷
$ python3 flags.py
BD BR CD CH DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
```

```
20 flags downloaded in 7.20s
$ python3 flags.py
BD BR CD CH DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.09s
$ python3 flags_threadpool.py
DE BD CH JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
20 flags downloaded in 1.37s ❸
$ python3 flags_threadpool.py
EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CH VN ET IR
20 flags downloaded in 1.60s
$ python3 flags_threadpool.py
BD DE EG CH ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
20 flags downloaded in 1.22s
$ python3 flags_asyncio.py ❹
BD BR IN ID TR DE CH US IR PK PH FR RU NG VN ET MX EG JP CD
20 flags downloaded in 1.36s
$ python3 flags_asyncio.py
RU CH BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
20 flags downloaded in 1.27s
$ python3 flags_asyncio.py
RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CH JP PH CD TR ❺
20 flags downloaded in 1.42s
```

❶ 每次运行脚本后，首先显示下载过程中下载完毕的国家代码，最后显示一个消息，说明用时。

❷ flags.py 脚本下载 20 个图像平均用时 7.18 秒。

❸ flags_threadpool.py 脚本平均用时 1.40 秒。

❹ flags_asyncio.py 脚本平均用时 1.35 秒。

❺ 注意国家代码的顺序：对并发下载的脚本来说，每次下载的顺序都不同。

两个并发下载的脚本之间性能差异不大，不过都比依序下载的脚本快 5 倍多。这只是一个任务，下载 20 个几千字节的文件。如果把下载的文件数量增加到几百个，并发下载的脚本能比依序下载的脚本快 20 倍或更多。



在公网中测试 HTTP 并发客户端可能不小心变成拒绝服务 (Denial-of-Service, DoS) 攻击, 或者有这么做的嫌疑。示例 20-1 不受此影响, 毕竟我们只发起 20 个请求。本章后文将使用 `http.server` 包运行测试脚本。

下面我们来分析示例 20-1 中的两个测试脚本——`flags.py` 和 `flags_threadpool.py`, 看看它们的实现方式。第三个脚本 `flags_asyncio.py` 留到第 21 章再分析。现在把 3 个脚本放在一起演示是为了表明以下两点。

01. 对于网络 I/O 操作, 不管使用哪种并发结构——线程或协程——只要代码写得没有问题, 吞吐量都比依序执行的代码高很多。
02. 对于可以控制发起多少请求的 HTTP 客户端, 线程与协程之间的性能差异不明显。³

³对于同时接收很多客户端访问的服务器来说, 区别还是有的: 协程的伸缩能力更好, 因为协程使用的内存比线程少很多, 而且没有切换上下文的开销 (详见 19.6.5 节)。

下面分析代码。

20.2.1 依序下载脚本

示例 20-1 运行的第一个脚本 `flags.py` 的实现如示例 20-2 所示。这个脚本没有什么特别之处, 不过实现并发下载的脚本时会重用其中多数代码和设置, 还是有必要分析一下。



简单起见, 示例 20-2 没有处理异常。异常稍后处理, 这里我们想集中说明代码的基本结构, 以便和并发下载的脚本进行对比。

示例 20-2 `flags.py`: 依序下载的脚本; 另外两个脚本会重用其中几个函数

```

import time
from pathlib import Path
from typing import Callable

import httpx ❶
POP20_CC = ('CH IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split() ❷

BASE_URL = 'http://mp.ituring.com.cn/files' ❸
DEST_DIR = Path('downloaded') ❹

def save_flag(img: bytes, filename: str) -> None: ❺
    (DEST_DIR / filename).write_bytes(img)

def get_flag(cc: str) -> bytes: ❻
    url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
    resp = httpx.get(url, timeout=6.1, ❼
                     follow_redirects=True) ❽
    resp.raise_for_status() ❾
    return resp.content

def download_many(cc_list: list[str]) -> int: ❿
    for cc in sorted(cc_list): ⓫
        image = get_flag(cc)
        save_flag(image, f'{cc}.gif')
        print(cc, end=' ', flush=True) ⓫
    return len(cc_list)

def main(downloader: Callable[[list[str]], int]) -> None: ⓫
    DEST_DIR.mkdir(exist_ok=True) ⓫
    t0 = time.perf_counter() ⓫
    count = downloader(POP20_CC)
    elapsed = time.perf_counter() - t0
    print(f'\n{count} downloads in {elapsed:.2f}s')

if __name__ == '__main__':
    main(download_many) ⓫

```

❶ 导入 `httpx` 库。这个库不在标准库中，按约定，在标准库中的模块之后导入，而且两部分之间有一个空行。

❷ 列出人口最多的 20 个国家的 ISO 3166 国家代码，按照人口数量降序排列。

- ③ 存放国旗图像的目录。
- ④ 保存图像的本地目录。
- ⑤ 把 `img` 字节序列保存到 `DEST_DIR` 目录中，命名为 `filename`。
- ⑥ 指定国家代码，构建 URL，然后下载图像，返回响应中的二进制内容。
- ⑦ 最好为网络操作指定一个合理的超时时间，防止莫名其妙阻塞几分钟。
- ⑧ HTTPX 默认不跟踪重定向。⁴

⁴其实，这个示例无须设置 `follow_redirects=True`，不过我想以此强调 HTTPX 和 `requests` 之间的区别。另外，设置 `follow_redirects=True` 之后，我还可以灵活应对未来的变化，说不定我会把图像文件移到其他地方。我觉得 HTTPX 默认设置 `follow_redirects=False` 是合理的，因为预期之外的重定向隐藏着不必要的请求，不易排查错误。

- ⑨ 这个脚本没有处理错误，但是这个方法在 HTTP 状态码不是 2XX 时抛出异常——为免失败后悄无声息，强烈建议调用该方法。
- ⑩ `download_many` 函数是与并发实现比较的关键。
- ⑪ 按字母表顺序迭代国家代码列表，明确表明输出的顺序与输入一致。返回下载的国旗数量。
- ⑫ 在同一行中一次显示一个国家代码，展示下载进度。`end=' '` 参数把常规的换行符替换为一个空格，在同一行依次显示各个国家代码。`flush=True` 参数不可缺少，因为 Python 的输出默认以行为单位缓冲，即 Python 只在换行符后显示可打印的字符。
- ⑬ 调用 `main` 函数必须传入执行下载任务的函数。如此一来，我们可以把 `main` 作为库函数使用，在 `threadpool` 和 `asyncio` 示例中

传入 `download_many` 的其他实现。

⑭ 如果需要，则创建 `DEST_DIR` 目录。如果该目录已存在，则不抛出错误。

⑮ 记录并报告 `downloader` 函数的运行时间。

⑯ 调用 `main` 函数，传入 `download_many` 函数。



HTTPX 库受符合 Python 习惯用法的 `requests` 包启发，不过底层建构更符合现代化思想。更重要的是，HTTPX 同时提供了同步和异步 API，因此本章和第 21 章的所有 HTTP 客户端示例都可以使用。Python 标准库中的 `urllib.request` 模块只有同步 API，而且对用户不友好。

`flags.py` 脚本中没有什么新知识，只是作为与其他脚本对比的基准，而且我把它当作一个库使用，避免实现其他脚本时重复编写代码。下面分析使用 `concurrent.futures` 模块重新实现的版本。

20.2.2 使用 `concurrent.futures` 模块下载

`concurrent.futures` 模块的功能主要由 `ThreadPoolExecutor` 和 `ProcessPoolExecutor` 类提供。这两个类实现的 API 能分别在不同的线程和进程中执行可调用对象。这两个类在内部维护着一个工作线程或进程池，以及分配任务和收集结果的队列。不过，这个接口抽象的层级很高，像下载国旗这种简单的案例，无须关心任何实现细节。

示例 20-3 展示如何使用 `ThreadPoolExecutor.map` 方法，以最简单的方式实现并发下载。

示例 20-3 `flags_threadpool.py`：使用 `futures.ThreadPoolExecutor` 类实现多线程下载的脚本

```

from concurrent import futures

from flags import save_flag, get_flag, main ❶

def download_one(cc: str): ❷
    image = get_flag(cc)
    save_flag(image, f'{cc}.gif')
    print(cc, end=' ', flush=True)
    return cc

def download_many(cc_list: list[str]) -> int:
    with futures.ThreadPoolExecutor() as executor: ❸
        res = executor.map(download_one, sorted(cc_list)) ❹

    return len(list(res)) ❺

if __name__ == '__main__':
    main(download_many) ❻

```

❶ 重用 `flags` 模块（见示例 20-2）中的几个函数。

❷ 下载单个图像的函数。这是在各个线程中执行的函数。

❸ 实例化 `ThreadPoolExecutor`，作为上下文管理器。

`executor.__exit__` 方法将调用

`executor.shutdown(wait=True)`，在所有线程都执行完毕前阻塞线程。

❹ `map` 方法的作用与内置函数 `map` 类似，不过 `download_one` 函数会在多个线程中并发调用。`map` 方法返回一个生成器，通过迭代可以获取各个函数调用返回的值——这里，每次调用 `download_one` 返回一个国家代码。

❺ 返回获取的结果数量。如果有线程抛出异常，那么异常在 `list` 构造方法尝试从 `executor.map` 返回的迭代器中获取相应的返回值时抛出。

⑥ 调用 `flags` 模块中的 `main` 函数，传入 `download_many` 函数的并发版。

注意，示例 20-3 中的 `download_one` 函数其实是示例 20-2 中 `download_many` 函数的 `for` 循环体。编写并发代码时经常这样重构：把依序执行的 `for` 循环改成函数，并发调用。



示例 20-3 特别短，因为我重用了依序下载的 `flags.py` 脚本中的多个函数。`concurrent.futures` 最大的优势是方便在现有的依序执行代码之上添加并发执行逻辑。

`ThreadPoolExecutor` 构造函数接受多个参数，这里没有用到。其中，第一个参数最为重要，即 `max_workers`。该参数设置最多执行多少个工作线程。`max_workers` 的值为 `None` 时（默认值），从 Python 3.8 开始，`ThreadPoolExecutor` 使用以下表达式决定线程数量。

```
max_workers = min(32, os.cpu_count() + 4)
```

`ThreadPoolExecutor` 文档解释了这么做的依据。

这个默认值为 I/O 密集型任务保留至少 5 个线程。对于那些释放了 GIL 的 CPU 密集型任务，最多使用 32 个 CPU 核。这样能够避免在多核设备中不知不觉使用特大量资源。

现在，`ThreadPoolExecutor` 在启动 `max_workers` 个工作线程之前也会重用空闲的工作线程。

总之，计算得到的 `max_workers` 值是合理的，而且 `ThreadPoolExecutor` 不轻易启动新线程。理解 `max_workers` 背后的逻辑或许有助于你决定何时以及如何设置该参数。

我们用的库是 `concurrency.futures`，可是在示例 20-3 中没有见到 `future` 对象，你不禁会想，`future` 对象在哪里呢？20.2.3 节会解答这个问题。

20.2.3 `future` 对象在哪里

`future` 对象是 `concurrent.futures` 模块和 `asyncio` 包的核心组件，可是，作为这两个库的用户，我们有时却见不到 `future` 对象。示例 20-3 在背后用到了 `future` 对象，但是我编写的代码没有直接使用它。本节概述 `future` 对象，还会举一个例子，展示具体用法。

从 Python 3.4 起，标准库中有两个名为 `Future` 的类：

`concurrent.futures.Future` 和 `asyncio.Future`。二者的作用相同：两个 `Future` 类的实例都表示可能已经完成或者尚未完成的延迟计算。这与 Twisted 中的 `Deferred` 类、Tornado 中的 `Future` 类，以及现代 JavaScript 中的 `Promise` 类似。

`future` 对象封装待完成的操作，可以放入队列，完成的状态可以查询，得到结果（或异常）后可以获取。

但是要记住一点，`future` 对象不应自己动手创建，只能由并发框架（`concurrent.futures` 或 `asyncio`）实例化。原因很简单：

`future` 对象表示终将运行的操作，必须排期运行，而这是框架的工作。具体而言，只有把可调用对象提交给某个

`concurrent.futures.Executor` 子类执行时才创建

`concurrent.futures.Future` 实例。例如，

`Executor.submit()` 方法接受一个可调用对象，排期执行，返回一个 `Future` 实例。

应用程序代码不应改变 `future` 对象的状态，并发框架在 `future` 对象表示的延迟计算结束后改变 `future` 对象的状态，而我们无法掌控计算何时结束。

两种 future 对象都有 `.done()` 方法。该方法不阻塞，返回一个布尔值，指明 future 对象包装的可调用对象是否已经执行。客户代码通常不询问 future 对象是否运行结束，而是等待通知。因此，两个 Future 类都有 `.add_done_callback()` 方法：提供一个可调用对象，在 future 对象执行完毕后调用；这个可调用对象的唯一参数是 future 对象。注意，回调的这个可调用对象与 future 对象包装的函数在同一个工作线程或进程中运行。

此外，还有 `.result()` 方法。该方法在两个 Future 类中的作用相同，当 future 对象运行结束后，返回可调用对象的结果，或者重新抛出执行可调用对象时抛出的异常。可是，如果 future 对象没有运行结束，那么 `result` 方法在两个 Future 类中的行为相差甚远。对于 `concurrency.futures.Future` 实例，调用 `f.result()` 方法将阻塞调用方所在的线程，直到有结果返回。此时，`result` 方法可以接受可选的 `timeout` 参数，如果在指定的时间内 future 对象没有运行完毕，则抛出 `TimeoutError` 异常。`asyncio.Future.result` 方法不支持设定超时时间，对于 `asyncio` 包，获取 future 对象的结果首选 `await`。不过，`await` 对 `concurrency.futures.Future` 实例不起作用。

这两个库中有几个函数返回 future 对象，其他函数则使用 future 对象，以用户易于理解的方式实现自身。示例 20-3 中的 `Executor.map` 方法属于后者，它返回一个迭代器，迭代器的 `__next__` 方法调用各个 future 对象的 `result` 方法，因此我们得到的是各个 future 对象的结果，而非 future 对象本身。

为了从实用的角度理解 future 对象，可以使用 `concurrent.futures.as_completed` 函数重写示例 20-3。这个函数的参数是一个 future 对象构成的可迭代对象，返回值是一个迭代器，在 future 对象运行结束后产出 future 对象。

为了使用 `futures.as_completed` 函数，只需修改 `download_many` 函数，把较高级的 `executor.map` 调用换成两个

for 循环：一个用于创建并排定 future 对象，另一个用于获取 future 对象的结果。同时，我们将添加几个 print 调用，显示运行结束前后的 future 对象。修改后的 download_many 函数如示例 20-4 所示，代码行数由 5 增加到 17，不过现在我们能一窥神秘的 future 对象了。其他函数不变，与示例 20-3 中一样。

示例 20- flags_threadpool_futures.py：把 download_many 函数中的 executor.map 换成 executor.submit 和 futures.as_completed

```
def download_many(cc_list: list[str]) -> int:
    cc_list = cc_list[:5] ❶
    with futures.ThreadPoolExecutor(max_workers=3) as executor: ❷
        to_do: list[futures.Future] = []
        for cc in sorted(cc_list): ❸
            future = executor.submit(download_one, cc) ❹
            to_do.append(future) ❺
            print(f'Scheduled for {cc}: {future}') ❻

        for count, future in
enumerate(futures.as_completed(to_do), 1): ❼
            res: str = future.result() ❽
            print(f'{future} result: {res!r}') ❾

    return count
```

- ❶ 这次演示只使用人口最多的 5 个国家。
- ❷ 把 max_workers 设为 3，以便在输出中观察待完成的 future 对象。
- ❸ 按照字母表顺序迭代国家代码，强调返回的结果是无序的。
- ❹ executor.submit 方法排定可调用对象的执行时间，返回一个 future 对象，表示待执行的操作。
- ❺ 存储各个 future 对象，后面传给 as_completed 函数。

- ⑥ 显示一个消息，包含国家代码和对应的 future 对象。
- ⑦ `as_completed` 函数在 future 对象运行结束后产出 future 对象。
- ⑧ 获取 future 对象的结果。
- ⑨ 显示 future 对象及其结果。

注意，在这个示例中调用 `future.result()` 方法绝不会阻塞，因为 future 由 `as_completed` 函数产出。运行示例 20-4 得到的输出如示例 20-5 所示。

示例 20-5 `flags_threadpool_futures.py` 的输出

```
$ python3 flags_threadpool_futures.py
Scheduled for BR: <Future at 0x100791518 state=running> ❶
Scheduled for CH: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending> ❷
Scheduled for US: <Future at 0x101807128 state=pending>
CH <Future at 0x100791710 state=finished returned str> result:
'CH' ❸
BR ID <Future at 0x100791518 state=finished returned str> result:
'BR' ❹
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result:
'IN'
US <Future at 0x101807128 state=finished returned str> result:
'US'

5 downloads in 0.70s
```

- ❶ 按字母表顺序排定 future 对象。future 对象的 `repr()` 方法显示 future 对象的状态；可以看到，前 3 个 future 对象的状态是 `running`，因为有 3 个工作线程。
- ❷ 后两个 future 对象的状态是 `pending`，等待有线程可用。

③ 这一行里的第一个 `CH` 是运行在一个工作线程中的 `download_one` 函数输出的，随后的内容是 `download_many` 函数输出的。

④ 这里有两个线程输出国家代码，然后主线程中的 `download_many` 函数输出第一个线程的结果。



建议你自己运行 `flags_threadpool_futures.py` 试试。多次运行，你会发现结果的顺序大有不同。如果把 `max_workers` 参数的值增大到 5，那么结果的顺序变化更大。把 `max_workers` 参数的值设为 1，代码将依序运行，结果的顺序始终与调用 `submit` 的顺序一致。

我们分析了两个使用 `concurrent.futures` 实现的下载脚本：使用 `ThreadPoolExecutor.map` 方法的示例 20-3 和使用 `futures.as_completed` 函数的示例 20-4。如果你对 `flags_asyncio.py` 脚本的代码好奇，可以看一眼示例 21-3（第 21 章会分析）。

下面简单说明如何在 CPU 密集型作业中使用 `concurrent.futures` 轻松绕过 GIL。

20.3 使用 `concurrent.futures` 启动进程

`concurrent.futures` 模块的文档副标题是“Launching parallel tasks”（执行并行任务）。这个模块实现的是真正的并行计算，因为它使用 `ProcessPoolExecutor` 类把工作分配给多个 Python 进程处理。

`ProcessPoolExecutor` 和 `ThreadPoolExecutor` 类都实现了 `Executor` 接口，因此使用 `concurrent.futures` 模块能特别轻松

地把基于线程的方案转成基于进程的方案。

下载国旗的示例或其他 I/O 密集型作业，使用 `ProcessPoolExecutor` 类得不到什么好处。这一点易于验证，只需把示例 20-3 中下面这几行：

```
def download_many(cc_list: list[str]) -> int:
    with futures.ThreadPoolExecutor() as executor:
```

改成：

```
def download_many(cc_list: list[str]) -> int:
    with futures.ProcessPoolExecutor() as executor:
```

`ProcessPoolExecutor` 构造函数也有一个 `max_workers` 参数，默认值为 `None`。这里，执行器限制职程的数量不能超过 `os.cpu_count()` 返回的数字。

相较线程，进程使用的内存更多，启动时间更长，因此 `ProcessPoolExecutor` 的价值在 CPU 密集型作业中才能体现出来。下面回到 19.6 节中的素数检测，使用 `concurrent.futures` 重写。

重写多核版素数检测程序

19.6.3 节分析的 `procs.py` 脚本使用 `multiprocessing` 做一些大数的素数检测。示例 20-6 中的 `proc_pool.py` 程序使用 `ProcessPoolExecutor` 解决同样的问题。从第一个导入语句到最后的 `main()` 调用，`procs.py` 有 43 行代码（不含空行），而 `proc_pool.py` 只有 31 行，减少了约 28%。

示例 20-6 `proc_pool.py`：使用 `ProcessPoolExecutor` 重写 `procs.py`

```

import sys
from concurrent import futures ❶
from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class PrimeResult(NamedTuple): ❷
    n: int
    flag: bool
    elapsed: float

def check(n: int) -> PrimeResult:
    t0 = perf_counter()
    res = is_prime(n)
    return PrimeResult(n, res, perf_counter() - t0)

def main() -> None:
    if len(sys.argv) < 2:
        workers = None ❸
    else:
        workers = int(sys.argv[1])

    executor = futures.ProcessPoolExecutor(workers) ❹
    actual_workers = executor._max_workers # type: ignore ❺

    print(f'Checking {len(NUMBERS)} numbers with {actual_workers}
processes:')

    t0 = perf_counter()

    numbers = sorted(NUMBERS, reverse=True) ❻
    with executor: ❼
        for n, prime, elapsed in executor.map(check, numbers): ❽
            label = 'P' if prime else ' '
            print(f'{n:16} {label} {elapsed:9.6f}s')

    time = perf_counter() - t0
    print(f'Total time: {time:.2f}s')

if __name__ == '__main__':
    main()

```

❶ 没必要导入 multiprocessing、SimpleQueue 等。一切都隐藏在 concurrent.futures 背后。

- ❷ `PrimeResult` 元组和 `check` 函数与 `procs.py` 中的一样，但是现在不需要那些队列和 `worker` 函数了。
- ❸ 未提供命令行参数时，我们不自己决定 `workers` 的数量，而是把值设为 `None`，让 `ProcessPoolExecutor` 来决定。
- ❹ 在❷处的 `with` 块之前构建 `ProcessPoolExecutor` 实例，以便在下一行显示线程的具体数量。
- ❺ `_max_workers` 是 `ProcessPoolExecutor` 的实例属性，文档中没有记载。我决定使用它显示 `workers` 变量的值为 `None` 时有多少线程。不出所料，`Mypy` 报错了，因此我加上了 `type: ignore` 注释，用来静默报错。
- ❻ 倒序排列要检查的数。这里将显示 `proc_pool.py` 与 `procs.py` 在行为上的差别。详见本例后面的说明。
- ❼ 使用 `executor` 作为上下文管理器。
- ❽ `executor.map` 调用返回 `check` 返回的 `PrimeResult` 实例，顺序与 `numbers` 参数相同。

运行示例 20-6，会发现结果的出现顺序完全是倒序的，如示例 20-7 所示。相比之下，`procs.py` 的输出（见 19.6.1 节）则取决于各个数的素数检测难度。例如，`procs.py` 在靠近顶部的位置显示 `7 777 777 777 777 777 777` 的结果，因为它有一个较小的因子 7，所以 `is_prime` 很快就能判断它不是素数。

相比之下，`7 777 777 536 340 681` 是 $88\,191\,709^2$ ，`is_prime` 要用很长时间才能判断它是合数。判断 `7 777 777 777 777 753` 是素数的时间更长——因此，在 `procs.py` 的输出中，这两个数出现在靠近末端的位置。

运行 `proc_pool.py`，你不仅会注意到结果倒序显示，还会发现显示 9 999 999 999 999 999 的结果之后，程序卡住了。

示例 20-7 `proc_pool.py` 的输出

```
$ ./proc_pool.py
Checking 20 numbers with 12 processes:
9999999999999999          0.000024s ❶
99999999999999917 P 9.500677s ❷
7777777777777777          0.000022s ❸
77777777777777753 P 8.976933s
7777777536340681          8.896149s
6666667141414921          8.537621s
66666666666666719 P 8.548641s
6666666666666666          0.000002s
5555555555555555          0.000017s
55555555555555503 P 8.214086s
5555553133149889          8.067247s
44444444488888889          7.546234s
4444444444444444          0.000002s
44444444444444423 P 7.622370s
33333335652092209          6.724649s
3333333333333333          0.000018s
33333333333333301 P 6.655039s
299593572317531 P 2.072723s
142702110479723 P 1.461840s
2 P 0.000001s
Total time: 9.65s
```

- ❶ 这一行很快就显示。
- ❷ 这一行用时 9.5 秒才显示。
- ❸ 余下各行几乎立即显示。

`proc_pool.py` 的行为缘何如此？原因如下。

- 前面说过，`executor.map(check, numbers)` 返回结果的顺序与 `numbers` 中数的顺序一致。
- `proc_pool.py` 默认使用的职程数量与 CPU 核数量相等——`max_workers` 为 `None` 时，`ProcessPoolExecutor` 的行为。

在我的笔记本电脑中，是 12 个进程。

- 由于 `numbers` 是倒序提交的，因此首先检测 9 999 999 999 999 999。该数的因子是 9，得到结果的速度很快。
- 第二个数是 9 999 999 999 999 917，这是样本中最大的素数，检测用时比其他数都长。
- 同时，余下的 11 个进程检测其他数，它们可能是素数、因子较大的合数，或者因子非常小的合数。
- 当负责检测 9 999 999 999 999 917 的进程最终判断它是素数之后，其他进程已经完成工作，因此结果立即显示出来。



虽然 `proc_pool.py` 的处理过程没有 `procs.py` 那么明显，但是总体执行时间基本与图 19-2 中的相同（进程数和 CPU 核数一样）。

我们知道，并发程序的行为不是那么简单直白。下面再做一个实验，帮你厘清 `Executor.map` 的操作。

20.4 实验 `Executor.map` 方法

本节研究 `Executor.map`：使用包含 3 个进程的 `ThreadPoolExecutor` 实例，运行 5 个可调用对象，输出带有时间戳的消息。代码在示例 20-8 中，输出在示例 20-9 中。

示例 20-8 `demo_executor_map.py`：简单演示 `ThreadPoolExecutor` 类的 `map` 方法

```
from time import sleep, strftime
from concurrent import futures

def display(*args): ❶
    print(strftime('%H:%M:%S'), end=' ')
    print(*args)

def loiter(n): ❷
    msg = '{}loiter({}): doing nothing for {}s...'
    sleep(n)
```

```

        display(msg.format('\t'*n, n, n))
        sleep(n)
        msg = '{}loiter({}): done.'
        display(msg.format('\t'*n, n))
        return n * 10 ❸

def main():
    display('Script starting.')
    executor = futures.ThreadPoolExecutor(max_workers=3) ❹
    results = executor.map(loiter, range(5)) ❺
    display('results:', results) ❻
    display('Waiting for individual results:')
    for i, result in enumerate(results): ❼
        display(f'result {i}: {result}')

if __name__ == '__main__':
    main()

```

❶ 这个函数的作用很简单，把传入的参数打印出来，并在前面加上 [HH:MM:SS] 格式的时间戳。

❷ loiter 函数的作用更简单，只是在开始时显示一个消息，然后休眠 n 秒，最后在结束时再显示一个消息；消息使用制表符缩进，缩进量由 n 的值确定。

❸ loiter 函数返回 $n * 10$ ，以便让我们了解收集结果的方式。

❹ 创建 ThreadPoolExecutor 实例，有 3 个线程。

❺ 把 5 个任务提交给 executor。因为只有 3 个线程，所以只有 3 个任务立即开始：loiter(0)、loiter(1) 和 loiter(2)，这些是非阻塞调用。

❻ 立即显示调用 executor.map 方法的结果：一个生成器，如示例 20-9 中的输出所示。

❼ for 循环中的 enumerate 函数隐式调用 next(results)，这个函数又在（内部）表示第一个任务（loiter(0)）的 future 对象 _f

上调用 `_f.result()`。`result` 方法会阻塞，直到 `future` 对象运行结束，因此这个循环每次迭代都要等待下一个结果做好准备。

建议你运行示例 20-8，看着结果逐个显示出来。此外，还可以修改传给 `ThreadPoolExecutor` 的 `max_workers` 参数，以及 `executor.map` 方法中 `range` 函数的参数。或者自己挑选几个值，以列表的形式传给 `map` 方法，得到不同的延迟。

示例 20-9 是某次运行示例 20-8 得到的输出。

示例 20-9 某次运行 `demo_executor_map.py`（见示例 20-8）得到的输出

```
$ python3 demo_executor_map.py
[15:56:50] Script starting. ❶
[15:56:50] loiter(0): doing nothing for 0s... ❷
[15:56:50] loiter(0): done.
[15:56:50]          loiter(1): doing nothing for 1s... ❸
[15:56:50]          loiter(2): doing nothing for 2s...
[15:56:50] results: <generator object result_iterator at
0x106517168> ❹
[15:56:50]          loiter(3): doing nothing for 3s...
❺
[15:56:50] Waiting for individual results:
[15:56:50] result 0: 0 ❻
[15:56:51]          loiter(1): done. ❼
[15:56:51]          loiter(4): doing nothing
for 4s...
[15:56:51] result 1: 10 ❽
[15:56:52]          loiter(2): done. ❾
[15:56:52] result 2: 20
[15:56:53]          loiter(3): done.
[15:56:53] result 3: 30
[15:56:55]          loiter(4): done. ❿
[15:56:55] result 4: 40
```

❶ 这次运行从 15:56:50 开始。

❷ 第一个线程执行 `loiter(0)`，因此休眠 0 秒，甚至在第二个线程开始之前就结束，不过具体情况因人而异。⁵

⁵具体情况因人而异：对线程来说，你永远不知道某一时刻事件的具体排序；有可能在另一台设备中会看到 `loiter(1)` 在 `loiter(0)` 结束之前开始，这是因为 `sleep` 函数总会释放 GIL。因此，即使休眠 0 秒，Python 也可能会切换到另一个线程。

③ `loiter(1)` 和 `loiter(2)` 立即开始（因为线程池中有 3 个线程，可以并发运行 3 个函数）。

④ 这一行表明，`executor.map` 方法返回的结果（`results`）是一个生成器。不管有多少任务，也不管 `max_workers` 的值是多少，目前都不会阻塞。

⑤ `loiter(0)` 运行结束了，第一个线程可以启动第四个线程，运行 `loiter(3)`。

⑥ 此时执行过程可能阻塞，具体情况取决于传给 `loiter` 函数的参数：`results` 生成器的 `__next__` 方法必须等到第一个 `future` 对象运行结束。此时不会阻塞，因为 `loiter(0)` 在循环开始前已经结束。注意，这一点之前的所有事件都在同一时刻发生，即 15:56:50。

⑦ 1 秒后，即 15:56:51，`loiter(1)` 运行完毕。这个线程闲置，可以开始运行 `loiter(4)`。

⑧ 显示 `loiter(1)` 的结果：10。现在，`for` 循环会阻塞，等待 `loiter(2)` 的结果。

⑨ 同上：`loiter(2)` 运行结束，显示结果；`loiter(3)` 也一样。

⑩ 2 秒后，`loiter(4)` 运行结束，因为 `loiter(4)` 在 15:56:51 时开始，空等了 4 秒。

`Executor.map` 函数易于使用，不过通常最好等结果准备好之后再获取，不要考虑提交的顺序。为此，要把 `Executor.submit` 方法和 `futures.as_completed` 函数结合起来使用，像示例 20-4 那样。20.5.2 节再回过头来讨论这种方式。



`executor.submit` 和 `futures.as_completed` 这个组合比 `executor.map` 更灵活，因为 `submit` 方法能处理不同的可调用对象和参数，而 `executor.map` 只适用于使用不同参数调用同一个可调用对象。此外，传给 `futures.as_completed` 函数的一系列 `future` 对象可以来自多个执行器，例如一些由 `ThreadPoolExecutor` 实例创建，另一些由 `ProcessPoolExecutor` 实例创建。

下一节根据新的需求继续实现下载国旗的示例，这一次不使用 `executor.map` 方法，而是迭代 `futures.as_completed` 函数返回的结果。

20.5 显示下载进度并处理错误

前面说过，20.2 节中的几个脚本没有处理错误，这样做是为了便于阅读和比较 3 种方案（依序、多线程和异步）的结构。

为了处理可能出现的各种错误，我创建了 `flags2` 系列示例。

`flags2_common.py`

这个模块中包含所有 `flags2` 示例通用的函数和设置，例如 `main` 函数，负责解析命令行参数、计时和报告结果。这个脚本中的代码其实是提供支持的，与本章的话题没有直接关系，书中不列出源码。

`flags2_sequential.py`

能正确处理错误，以及显示进度条的依序下载版 HTTP 客户端。`flags2_threadpool.py` 脚本会用到这个模块里的 `download_one` 函数。

`flags2_threadpool.py`

基于 `futures.ThreadPoolExecutor` 类实现的并发版 HTTP 客户端，演示如何处理错误，以及集成进度条。

`flags2_asyncio.py`

与前一个脚本的作用相同，不过使用 `asyncio` 和 `httpx` 实现。这个脚本将在 21.7 节分析。



测试并发客户端时要小心

在公开的 HTTP 服务器上测试并发 HTTP 客户端时要小心，因为每秒可能会发起很多请求，这相当于是拒绝服务（DoS）攻击。访问公开的服务器时一定要对客户端限流。测试时，应该在本地架设 HTTP 服务器。具体方法见“搭建测试服务器”提示栏。

`flags2` 系列示例最明显的特色是，有使用 `tqdm` 包实现的文本动画进度条。图 20-1 中有两个截图，分别是 `flags2_threadpool.py` 脚本运行中和运行结束后看到的输出。

```
(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
|###-----| 300/676 44% [elapsed: 00:08 left: 00:10, 36.17 iters/sec]

(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
-----
147 flags downloaded.
360 not found.
169 errors.
Elapsed time: 18.10s
(.venv34) lontra:countries luciano$
```

图 20-1：左上：`flags2_threadpool.py` 运行中，显示着 `tqdm` 包生成的进度条；右下：同一个终端窗口，脚本运行完毕后

`tqdm` 项目的 `README.md` 文件中有一个 `.gif` 动画，演示了最简单的用法。安装 `tqdm` 包之后，在 Python 控制台中输入以下代码，在注释那里会看到进度条动画。

```
>>> import time
>>> from tqdm import tqdm
>>> for i in tqdm(range(1000)):
...     time.sleep(.01)
...
>>> # -> 进度条会出现在这里 <-
```

除了这个精妙的效果之外，`tqdm` 函数的实现方式也很有趣：能处理任何可迭代对象，生成一个迭代器；使用这个迭代器时，显示进度条和完成全部迭代预计的剩余时间。为了计算预计剩余时间，`tqdm` 函数要获取一个能使用 `len` 函数确定长度的可迭代对象，或者通过 `total=` 参数指定预期的项数。借助在 `flags2` 系列示例中集成 `tqdm`，我们可以深入了解这几个并发脚本的运作方式，因为我们必须使用 `futures.as_completed` 函数和 `asyncio.as_completed` 函数，这样 `tqdm` 函数才能在每个 `future` 对象运行结束后更新进度。

`flags2` 系列示例的另一个特色是提供了命令行接口。3 个脚本接受的选项相同，运行任意一个脚本时指定 `-h` 选项就能看到所有选项。示例 20-10 显示的是帮助文本。

示例 20-10 `flags2` 系列脚本的帮助界面

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT]
                             [-s LABEL]
                             [-v]
                             [CC [CC ...]]

Download flags for country codes. Default: top 20 countries by
population.

positional arguments:
  CC                      country code or 1st letter (eg. B for
BA...BZ)

optional arguments:
  -h, --help              show this help message and exit
  -a, --all               get all available flags (AD to ZW)
  -e, --every             get flags for every possible code
(AA...ZZ)
  -l N, --limit N        limit to N first codes
```

```
-m CONCURRENT, --max_req CONCURRENT
                        maximum concurrent requests (default=30)
-s LABEL, --server LABEL
                        Server to hit; one of DELAY, ERROR, LOCAL,
REMOTE
                        (default=LOCAL)
-v, --verbose           output detailed progress info
```

所有选项都是可选的。其中，`-s/--server` 对测试最重要：选择测试时使用的 HTTP 服务器和端口。这个选项的值可以设为下述 4 个字符串（不区分大小写），用于确定脚本从哪里下载国旗。

LOCAL

使用 `http://localhost:8000/flags`；这是默认值。你需要配置一个本地 HTTP 服务器，响应 8000 端口的请求。搭建说明见下面的提示栏。

REMOTE

我搭建的公开网站托管在一个共享服务器中。请不要使用太多并发请求访问这个网站。该网站域名由 Cloudflare CDN 处理，因此第一次下载可能有点慢，不过一旦 CDN 有了缓存，速度就会变快。

DELAY

使用 `http://localhost:8001/flags`；这个服务器延迟 HTTP 响应，监听的端口是 8001。为了方便试验，我编写了 `slow_server.py` 脚本，放在本书代码的 `20-futures/getflags/` 目录下。搭建说明见下面的提示栏。

ERROR

使用 `http://localhost:8002/flags`；这个服务器监听 8002 端口，引入了一些 HTTP 错误。搭建说明见下面的提示栏。



搭建测试服务器

如果你没有供测试使用的本地 HTTP 服务器，可以使用我编写的说明搭建。这份说明仅用到了 Python 3.9 及以上版本的标准库（没有外部库），放在本书代码的 20-executors/getflags/README.adoc 文件中。简单来说，README.adoc 说明了如何使用如下 3 个服务器。

```
python3 -m http.server
```

8000 端口上的 LOCAL 服务器。

```
python3 slow_server.py
```

8001 端口上的 DELAY 服务器，响应之前随机延迟 0.5 ~ 5 秒。

```
python3 slow_server.py 8002 --error-rate .25
```

8002 端口上的 ERROR 服务器，除了随机延迟之外，还有 25% 的概率返回“418 I'm a teapot”错误响应。

默认情况下，各个 flags2*.py 脚本使用默认的并发连接数（各脚本有所不同）从 LOCAL 服务器（<http://localhost:8000/flags>）中下载人口最多的 20 个国家的国旗。示例 20-11 是全部使用默认值运行 flags2_sequential.py 脚本得到的输出。运行这个脚本之前，需要按照“测试并发客户端时要小心”注意框中的说明搭建好本地服务器。

示例 20-11 全部使用默认值运行 flags2_sequential.py 脚本：
LOCAL 服务器，人口最多的 20 国国旗，1 个并发连接

```
$ python3 flags2_sequential.py
LOCAL site: http://localhost:8000/flags
Searching for 20 flags: from BD to VN
1 concurrent connection will be used.
-----
20 flags downloaded.
Elapsed time: 0.10s
```

我们可以使用多种不同的方式选择下载哪些国家的国旗。示例 20-12 展示如何下载国家代码以字母 A、B 或 C 开头的所有国旗。

示例 20-12 运行 `flags2_threadpool.py` 脚本，从 `DELAY` 服务器中下载国家代码以 A、B 或 C 开头的所有国旗

```
$ python3 flags2_threadpool.py -s DELAY a b c
DELAY site: http://localhost:8001/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.
-----
43 flags downloaded.
35 not found.
Elapsed time: 1.72s
```

不管使用什么方式选择国家代码，下载的国旗数量都可以使用 `-l/--limit` 选项限制。示例 20-13 演示如何发起 100 个请求，结合 `-a` 和 `-l` 选项下载 100 面国旗。

示例 20-13 运行 `flags2_asyncio.py` 脚本，使用 100 个并发请求 (`-m 100`) 从 `ERROR` 服务器中下载 100 面国旗 (`-al 100`)

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8002/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
-----
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

以上是 `flags2` 系列示例的用户界面。下面分析实现方式。

20.5.1 flags2 系列示例处理错误的方式

这 3 个示例在负责下载一个文件的函数 (`download_one`) 中使用相同的策略处理 HTTP 404 错误 (未找到)。其他异常则向上冒泡，交

给 `download_many` 函数或 `supervisor` 协程 (`asyncio` 版) 处理。

我们还是先分析依序下载版的代码，因为这一版更易于理解，而且使用线程池的脚本重用了这里的大部分代码。示例 20-14 给出的是 `flags2_sequential.py` 和 `flags2_threadpool.py` 脚本真正用于下载的函数。

示例 20-14 `flags2_sequential.py`: 负责下载基本函数;
`flags2_threadpool.py` 脚本重用了这两个函数

```
from collections import Counter
from http import HTTPStatus

import httpx
import tqdm # type: ignore ❶

from flags2_common import main, save_flag, DownloadStatus ❷

DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1

def get_flag(base_url: str, cc: str) -> bytes:
    url = f'{base_url}/{cc}/{cc}.gif'.lower()
    resp = httpx.get(url, timeout=3.1, follow_redirects=True)
    resp.raise_for_status() ❸
    return resp.content

def download_one(cc: str, base_url: str, verbose: bool = False) ->
DownloadStatus:
    try:
        image = get_flag(base_url, cc)
    except httpx.HTTPStatusError as exc: ❹
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND ❺
            msg = f'not found: {res.url}'
        else:
            raise ❻
    else:
        save_flag(image, f'{cc}.gif')
        status = DownloadStatus.OK
        msg = 'OK'
```



```
if verbose: ❷
    print(cc, msg)

return status
```

❶ 导入显示进度条的 `tqdm` 库，让 `Mypy` 跳过检查。⁶

⁶截至 2021 年 9 月，`tqdm` 包没有类型提示。这没什么，太阳照常升起。类型是可选的，感谢 Guido！

❷ 从 `flags2_common` 模块中导入两个函数和一个枚举。

❸ 如果 HTTP 状态码不在 `range(200, 300)` 范围内，抛出 `HTTPStetusError`。

❹ `download_one` 函数捕获 `HTTPStatusError`，特别处理 HTTP 404 错误.....

❺ 方法是，把局部变量 `status` 设为 `DownloadStatus.NOT_FOUND`。`DownloadStatus` 是从 `flags2_common.py` 中导入的一个枚举。

❻ 重新抛出其他 `HTTPStatusError` 异常，向上冒泡，传给调用方。

❼ 如果在命令行中设定了 `-v/--verbose` 选项，显示国家代码和状态消息。这就是详细模式中看到的进度信息。

示例 20-15 给出的是 `download_many` 函数的依序下载版。代码虽然简单，还是有必要分析一下，以便后面与并发版对比。我们要关注的是报告进度、处理错误和统计下载数量的方式。

示例 20-15 `flags2_sequential.py`：实现依序下载的 `download_many` 函数

```

def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  _unused_concur_req: int) -> Counter[DownloadStatus]:
    counter: Counter[DownloadStatus] = Counter() ❶
    cc_iter = sorted(cc_list) ❷
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter) ❸
    for cc in cc_iter:
        try:
            status = download_one(cc, base_url, verbose) ❹
        except httpx.HTTPStatusError as exc: ❺
            error_msg = 'HTTP error {resp.status_code} - {resp.reason_phrase}'
            error_msg = error_msg.format(resp=exc.response)
        except httpx.RequestError as exc: ❻
            error_msg = f'{exc} {type(exc)}'.strip()
        except KeyboardInterrupt: ❼
            break
        else: ❽
            error_msg = ''

        if error_msg:
            status = DownloadStatus.ERROR ❾
        counter[status] += 1 ❿
        if verbose and error_msg: ⓫
            print(f'{cc} error: {error_msg}')

    return counter

```

❶ 这个 Counter 实例用于统计不同的下载状态：

DownloadStatus.OK、DownloadStatus.NOT_FOUND 或 DownloadStatus.ERROR。

❷ cc_iter 存放通过参数传入的国家代码列表，按字母表顺序排列。

❸ 如果不是详细模式，则把 cc_iter 传给 tqdm 函数，返回一个迭代器，产出 cc_iter 中的项，同时显示进度条动画。

❹ 不断调用 download_one 函数。

⑤ `get_flag` 抛出的 HTTP 状态码异常和未被 `download_one` 处理的异常在这里处理。

⑥ 其他与网络有关的异常在这里处理。除此之外的异常会中止脚本，因为调用 `download_many` 函数的 `flags2_common.main` 函数中没有 `try/except` 块。

⑦ 用户按 `Ctrl-C` 组合键时退出循环。

⑧ 如果没有异常从 `download_one` 函数中逃出，则清空错误消息。

⑨ 如果有错误，则把局部变量 `status` 设为相应的状态。

⑩ 递增相应状态的计数器。

⑪ 如果是详细模式，而且有错误，则显示带有当前国家代码的错误消息。

⑫ 返回 `counter`，以便 `main` 函数能在最终的报告中显示数量。

下面分析重构后的线程池版示例——`flags2_threadpool.py`。

20.5.2 使用 `futures.as_completed` 函数

为了集成 `tqdm` 进度条，并处理各个请求中的错误，`flags2_threadpool.py` 脚本用到我们见过的

`futures.ThreadPoolExecutor` 类和 `futures.as_completed` 函数。示例 20-16 是 `flags2_threadpool.py` 脚本的完整代码清单。这个脚本只实现了 `download_many` 函数，其他函数都重用 `flags2_common.py` 和 `flags2_sequential.py` 脚本里的。

示例 20-16 `flags2_threadpool.py`：完整的代码清单

```
from collections import Counter
from concurrent.futures import ThreadPoolExecutor, as_completed
```

```

import httpx
import tqdm # type: ignore

from flags2_common import main, DownloadStatus
from flags2_sequential import download_one ❶

DEFAULT_CONCUR_REQ = 30 ❷
MAX_CONCUR_REQ = 1000 ❸

def download_many(cc_list: list[str],
                  base_url: str,
                  verbose: bool,
                  concur_req: int) -> Counter[DownloadStatus]:
    counter: Counter[DownloadStatus] = Counter()
    with ThreadPoolExecutor(max_workers=concur_req) as executor:
❹
        to_do_map = {} ❺
        for cc in sorted(cc_list): ❻
            future = executor.submit(download_one, cc,
                                     base_url, verbose) ❼

            to_do_map[future] = cc ❽
        done_iter = as_completed(to_do_map) ❾
        if not verbose:
            done_iter = tqdm.tqdm(done_iter, total=len(cc_list))
❿
        for future in done_iter: ⓫
            try:
                status = future.result() ⓬
            except httpx.HTTPStatusError as exc: ⓭
                error_msg = 'HTTP error {resp.status_code} -
{resp.reason_phrase}'
                error_msg = error_msg.format(resp=exc.response)
            except httpx.RequestError as exc:
                error_msg = f'{exc} {type(exc)}'.strip()
            except KeyboardInterrupt:
                break
            else:
                error_msg = ''

            if error_msg:
                status = DownloadStatus.ERROR
                counter[status] += 1
            if verbose and error_msg:
                cc = to_do_map[future] ⓮
                print(f'{cc} error: {error_msg}')

```

```
        return counter

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

- ❶ 重用 `flags2_sequential` 模块（见示例 20-14）中的 `download_one` 函数。
- ❷ 如果没有在命令行中指定 `-m/--max_req` 选项，则使用这个值作为并发请求数的最大值，也就是线程池的大小；真实的数量可能会比这少，例如下载的国旗数量较少。
- ❸ 不管要下载多少国旗，也不管 `-m/--max_req` 命令行选项的值是多少，`MAX_CONCUR_REQ` 会限制最大的并发请求数。这是一项安全措施，免得启动太多线程，消耗过多内存。
- ❹ 把 `max_workers` 设为 `concur_req`，创建 `executor`。`main` 函数会把下面这 3 个值中最小的那个赋给 `concur_req`：
`MAX_CONCUR_REQ`、`cc_list` 的长度、`-m/--max_req` 命令行选项的值。这样能避免创建过多的线程。
- ❺ 这个字典把各个 `Future` 实例（表示一次下载）映射到相应的国家代码上，在处理错误时使用。
- ❻ 按字母表顺序迭代国家代码列表。结果的顺序主要由 HTTP 响应的长短决定，不过，如果线程池的大小（由 `concur_req` 设定）比 `len(cc_list)` 小得多，那么可能会按字母表顺序批量下载。
- ❼ 每次调用 `executor.submit` 方法排定一个可调用对象的执行时间，返回一个 `Future` 实例。第一个参数是可调用对象，余下的参数是传给可调用对象的参数。
- ❽ 把返回的 `future` 和国家代码存储在字典中。

⑨ `futures.as_completed` 函数返回一个迭代器，在每个任务运行结束后产出 `future` 对象。

⑩ 如果不是详细模式，则把 `as_completed` 函数返回的结果传给 `tqdm` 函数，显示进度条；因为 `done_iter` 没有长度，所以我们必须通过 `total=` 参数告诉 `tqdm` 函数预期的项数，这样 `tqdm` 才能预计剩余的工作量。

⑪ 迭代运行结束后的 `future` 对象。

⑫ 在 `future` 对象上调用 `result` 方法，要么返回可调用对象的返回值，要么抛出可调用对象在执行过程中捕获的异常。这个方法可能会阻塞，等待确定结果；但是，在这个示例中不阻塞，因为 `as_completed` 函数只返回已经运行结束的 `future` 对象。

⑬ 处理可能出现的异常。这个函数余下的代码与依序下载版 `download_many` 函数一样（见示例 20-15），唯有下一点除外。

⑭ 为了给错误消息提供上下文，以当前的 `future` 为键，从 `to_do_map` 中获取国家代码。在依序下载版中无须这么做，因为那一版迭代的是国家代码，知道当前国家代码是什么，而这里迭代的是 `future` 对象。



示例 20-16 用到了一个对 `futures.as_completed` 函数特别有用的惯用法：构建一个字典，把各个 `future` 对象映射到 `future` 对象运行结束后可能有用的其他数据上。这里，在 `to_do_map` 中，我们把各个 `future` 对象映射到对应的国家代码上。这样，尽管 `future` 对象生成的结果顺序已经乱了，依然便于使用结果做后续处理。

Python 线程特别适合 I/O 密集型应用程序，`concurrent.futures` 包大大简化了某些使用场景下 Python 线程的用法。另外，使用 `ProcessPoolExecutor` 还可以利用多核解决 CPU 密集型问题——

如果是“高度并行”计算的话。我们对 `concurrent.futures` 基本用法的介绍到此结束。

20.6 本章小结

本章开头对两个并发版 HTTP 客户端和一个依序下载版客户端做了比较，结果是并发版比依序下载的脚本性能高很多。

分析过使用 `concurrent.futures` 实现的第一个示例后，我们深入探讨了 `future` 对象，即 `concurrent.futures.Future` 或 `asyncio.Future` 类的实例，着重说明了二者的共同点（区别在第 21 章详述）。我们说明了如何使用 `Executor.submit` 方法创建 `future` 对象，以及如何使用 `concurrent.futures.as_completed` 函数迭代运行结束的 `future` 对象。

然后，我们讨论了如何借助 `concurrent.futures.ProcessPoolExecutor` 类使用多个进程，以此绕开 GIL，使用多个 CPU 核简化第 19 章中的多核版素数检查程序。

在随后的一节中，我们深入分析了 `concurrent.futures.ThreadPoolExecutor` 类的运作方式。为了说明问题，我特意举了一个示例，创建几个任务，但是空闲几秒，什么也不做，只是显示带有时间戳的状态。

接下来，本章回到下载国旗的示例，增加了进度条和错误处理代码，并且进一步探讨了 `future.as_completed` 生成器函数。我们得知一个常见的做法：把 `future` 对象存储在一个字典中，提交时把 `future` 对象与相关的信息联系起来；这样，`as_completed` 迭代器产出 `future` 对象后，就可以使用那些信息。

20.7 延伸阅读

Brian Quinlan 是 `concurrent.futures` 包的贡献者，他在 PyCon Australia 2010 上所做的“The Future Is Soon!”演讲对这个包做了介绍。Quinlan 演讲时没用幻灯片，而是直接在 Python 控制台中输入代码，以此说明这个库的用途。这个库的正式介绍文件是“PEP 3148 - `futures - execute computations asynchronously`”。在这个 PEP 中，Quinlan 写道，`concurrent.futures` 库“深受 Java 的 `java.util.concurrent` 包影响”。

与 `concurrent.futures` 有关的其他资料，参阅第 19 章。19.9.1 节给出的与 Python 的 `threading` 和 `multiprocessing` 模块有关的资料同样适用于 `concurrent.futures`。

杂谈

远离线程

并发是计算机科学中最难的概念之一（通常最好别去招惹它）。

——David Beazley
Python 教练和科学狂人⁷

上面引自 David Beazley 的话与本章开头引自 Michele Simionato 的话明显矛盾，但我都同意。

我本科时学过一门并发课程，利用 POSIX 线程编程。学完之后，我得出一个结论：千万不要自己管理线程和锁，原因与不自己管理内存分配和释放一样。这些任务最好由懂行的系统程序员处理，他们有这种爱好，也有时间去管理（但愿如此）。公司发工资是让我开发应用程序，而不是开发操作系统，因此也用不着精细控制线程、锁、`malloc` 和 `free`。

因此，我觉得 `concurrent.futures` 包很棒，它把线程、进程和队列视作服务的基础设施，不用自己动手直接处理。当然，这个包针对的是简单的作业，也就是所谓的“高度并行”问题。可是，正如本章开头 Simionato 所说的那样，开发应用程序（而非操作系统或数据库服务器）时，我们遇到的大部分并发问题属于这一种。

对于并发程度不高的问题来说，线程和锁也不是解决之道。在操作系统层面，线程永远不会消失；不过，过去几年我觉得让人眼前一亮的编程语言（包括 Go、Elixir 和 Clojure）都对并发做了更好、更高层的抽象，不信你可以读一下《七周七并发模型》一书。Erlang（实现 Elixir 的语言）是典型示例，设计这门语言时彻底考虑到了并发。我对这门语言不感兴趣的原因很简单——句法丑陋。我被 Python 的句法宠坏了。

José Valim 以前是 Ruby on Rails 的核心贡献者，他设计的 Elixir 提供了友好而现代的句法。与 Lisp 和 Clojure 一样，Elixir 也实现了句法宏。这是一把双刃剑。使用句法宏能实现强大的 DSL（Domain-Specific Language，领域特定语言），可是衍生语言多起来之后，基准代码会出现兼容问题，社区会分裂。大量涌现的宏导致 Lisp 没落，因为各种 Lisp 实现都使用独特而难懂的方言。标准化后的 Common Lisp 才开始复苏。我希望 José Valim 能引领 Elixir 社区，不要重蹈覆辙。目前来看，他做得不错。数据库包装和查询生成库 Ecto 使用起来让人心情舒畅，这是一个使用宏的正面示例，创建的 DSL 非常灵活，而且对用户友好，与关系数据库或非关系数据库交互都不在话下。

与 Elixir 类似，Go 也是一门充满新意的现代语言。可是，与 Elixir 相比，某些方面有点保守。Go 语言不支持宏，句法倒是比 Python 简单。Go 语言也不支持继承和运算符重载，而且提供的元编程支持没有 Python 多。这些限制被认为是 Go 语言的特点，因为行为和性能更可预料。这对高并发来说是好事，而 Go 语言的重要使命是取代 C++、Java 和 Python。

虽然 Elixir 和 Go 语言在高并发领域是直接的竞争者，但是设计原理的不同吸引了不同的用户群。这两门语言都可能蓬勃发展。可是纵观编程语言的历史，保守的语言更能吸引程序员。

⁷摘自 PyCon 2009 上“A Curious Course on Coroutines and Concurrency”教程的第 9 张幻灯片。

第 21 章 异步编程

常规异步编程方案有一种“不成功便成仁”式的傲骨侠风，这就是问题所在。你孤注一掷重写代码，要么彻底避免阻塞，要么纯属浪费时间。

——Alvaro Videla 和 Jason J. W. Williams
《RabbitMQ 实战：高效部署分布式消息队列》

本章将厘清 3 个联系紧密的主要话题：

- Python 的 `async def`、`await`、`async with` 和 `async for` 结构；
- 支持这些结构的对象——原生协程和异步版上下文管理器、可迭代对象、生成器及推导式；
- `asyncio` 和其他异步库。

本章内容建立在可迭代对象和生成器（第 17 章，尤其是 17.13 节）、上下文管理器（第 18 章），以及并发编程一般概念（第 19 章）等基础之上。

我们将要分析的并发 HTTP 客户端与第 20 章中的类似，也使用 HTTPX 库，但我们将利用该库提供的异步 API，使用原生协程和异步上下文管理器来重写该客户端。我们还将说明如何把速度慢的操作委托给线程或进程执行器，以防止阻塞事件循环。

在 HTTP 客户端示例之后，我们将实现两个简单的异步服务器端应用程序，其中一个应用程序使用日益流行的 FastAPI 框架。然后，介绍随 `async/await` 关键字而来的其他语言结构，包括异步生成器函数、异步推导式和异步生成器表达式。为了指出这些语言功能没有限定只能使用 `asyncio` 库，我们将使用 Curio（David Beazley 开发的一个异步框架，不仅优雅，而且具有创新）重写一个示例。

最后，简要说明异步编程的优势和陷阱。

本章内容较多，只能举些简单的例子，不过都能体现各个功能的要点。



经 Yury Selivanov¹ 重新编排之后，`asyncio` 库的文档比以前好多了。现在，供应用程序开发人员使用的几个函数与供 Web 框架和数据库驱动包的编写人员使用的底层 API 已经区分开。

如果你想阅读专门讨论 `asyncio` 的书，我推荐 *Using Asyncio in Python* (Caleb Hattingh 著)。Caleb 是本书的技术审校之一。

¹Selivanov 实现了 Python 中的 `async/await`，还编写了相关的 PEP：492、525 和 530。

21.1 本章新增内容

撰写本书第 1 版时，`asyncio` 库尚未成型，`async/await` 关键字也没有出现。因此，第 2 版更新了本章的所有示例。另外，我还新增了一些示例，包括几个域名探测脚本、一个 FastAPI Web 服务，以及使用 Python 控制台新的异步模式所做的几个实验。

本章新增了几节，讲解第 1 版中不存在的语言功能。

21.13 节中的观点是从过往教训中总结出来的经验，能帮你绕开很多陷阱，我觉得任何使用异步编程的人都应该阅读——不管你是使用 Python 还是 Node.js。

最后，我删除了与 `asyncio.Futures` 有关的多段内容，因为它现在属于 `asyncio` 的底层 API。

21.2 一些定义

17.13 节开头提到，Python 3.5 及以上版本提供了如下 3 种协程。

原生协程

使用 `async def` 定义的协程函数。在原生协程内可以使用 `await` 关键字委托另一个原生协程，这类似于在经典协程中使用 `yield from`。`async def` 语句定义的始终是原生协程，即使主体中没有使用 `await` 关键字。`await` 关键字不能在原生协程外部使用。²

²这条规则有一个例外：如果使用 `-m asyncio` 选项运行 Python，在 `>>>` 提示符中可以直接使用 `await` 驱动原生协程。详见 21.10.1 节的“在 Python 异步控制台中实验”。

经典协程

一种生成器函数，在表达式中使用 `yield` 读取 `my_coro.send(data)` 调用发送的数据。经典协程可以使用 `yield from` 委托其他经典协程。经典协程不能由 `await` 驱动，而且 `asyncio` 库不再支持。

基于生成器的协程

一种使用 `@types.coroutine` (Python 3.5 引入) 装饰的生成器函数。使用这个装饰器的生成器与新增的 `await` 关键字兼容。

本章重点讨论原生协程和**异步生成器**。

异步生成器

一种使用 `async def` 定义，而且在主体中使用 `yield` 的生成器函数。返回一个提供 `__anext__` 方法（获取下一项）的异步生成器对象。



@asyncio.coroutine 没有前途³

供经典协程和基于生成器的协程使用的 `@asyncio.coroutine` 装饰器在 Python 3.8 中已被弃用，根据 43216 号工单，计划在 Python 3.11 中删除。不过，根据 36921 号工单，`@types.coroutine` 应该会得以保留。尽管 `asyncio` 已不再支持它，但是异步框架 Curio 和 Trio 的底层代码仍在使用。

³请原谅我的直白。

21.3 一个 asyncio 示例：探测域名

假设你想创建一个有关 Python 的博客，准备使用一个 Python 关键字注册后缀为 `.dev` 的域名，例如 `await.dev`。示例 21-1 中的脚本使用 `asyncio` 并发检查多个域名。

注意，域名不以特定的顺序出现。运行这个脚本，你会发现域名一个接一个地显示出来，延迟时间不等。`+` 符号表示你的设备能通过 DNS 解析相应的域名。不带 `+` 符号的域名无法解析，这意味着该域名或许可以注册。

`blogdom.py` 脚本通过原生协程对象探测 DNS。由于异步操作是交叉执行的，因此检查 18 个域名所需的时间比依序检查少很多。其实，总用时基本与最慢的 DNS 响应时长相当，而不是所有响应时间之和。

`blogdom.py` 脚本的代码如示例 21-1 所示。

示例 21-1 `blogdom.py`：为一个 Python 博客搜索域名

```
#!/usr/bin/env python3
import asyncio
import socket
from keyword import kwlist
```

```

MAX_KEYWORD_LEN = 4 ❶

async def probe(domain: str) -> tuple[str, bool]: ❷
    loop = asyncio.get_running_loop() ❸
    try:
        await loop.getaddrinfo(domain, None) ❹
    except socket.gaierror:
        return (domain, False)
    return (domain, True)

async def main() -> None: ❺
    names = (kw for kw in kwlist if len(kw) <= MAX_KEYWORD_LEN) ❻
    domains = (f'{name}.dev'.lower() for name in names) ❼
    coros = [probe(domain) for domain in domains] ❽
    for coro in asyncio.as_completed(coros): ❾
        domain, found = await coro ❿
        mark = '+' if found else ' '
        print(f'{mark} {domain}')

if __name__ == '__main__':
    asyncio.run(main()) ⓫

```

❶ 设置关键字的最大长度，因为域名越短越好。

❷ `probe` 返回一个元组，包含域名和一个布尔值。`True` 表示域名可解析。同时返回域名方便显示结果。

❸ 获取 `asyncio` 事件循环的引用，供后面使用。

❹ 协程方法 `loop.getaddrinfo(...)` 返回一个五元组，使用套接字连接指定的地址。在这个示例中，我们不需要返回的结果。如果有结果返回，则说明域名可解析；否则不可解析。

❺ `main` 必定是一个协程，因此可以在主体中使用 `await`。

❻ 一个生成器，产出长度不超过 `MAX_KEYWORD_LEN` 的 Python 关键字。

- ⑦ 一个生成器，产出后缀为 `.dev` 的域名。
- ⑧ 调用 `probe` 协程，传入各个 `domain`，构建一个协程对象列表。
- ⑨ `asyncio.as_completed` 是一个生成器，产出协程，按照传入的协程完成的顺序（不是协程的提交顺序）返回结果。作用类似于示例 20-4 中用过的 `futures.as_completed`。
- ⑩ 此时，我们知道协程已经结束，因为 `as_completed` 就是这个作用。因此，`await` 表达式不阻塞，但是我们需要从 `coro` 中获取结果。若 `coro` 抛出的异常未被处理，自然在这里重新抛出。
- ⑪ `asyncio.run` 启动事件循环，仅当事件循环退出后返回。使用 `asyncio` 的脚本经常这样做，即把 `main` 实现为协程，在 `if __name__ == '__main__':` 块中使用 `asyncio.run` 驱动。



`asyncio.get_running_loop` 函数在 Python 3.7 中新增，供协程内部使用，例如这里的 `probe`。如果没有运行中的循环，那么 `asyncio.get_running_loop` 抛出 `RuntimeError`。`asyncio.get_running_loop` 的实现比 `asyncio.get_event_loop` 更简单，速度也更快。`asyncio.get_event_loop` 在必要时会启动事件循环。从 Python 3.10 开始，`asyncio.get_event_loop` 已被弃用，最终将变成 `asyncio.get_running_loop` 的别名。

Guido 阅读异步代码的技巧

学习 `asyncio` 库要掌握的新概念很多，但是如果你采用 Guido van Rossum 自用的技巧阅读示例 21-1，那么整体逻辑不难理解。Guido 的技巧是，对 `await` 关键字视而不见，假装它不存在。如此一来，协程就像是依序运行的常规函数。

以下述协程为例。


```
async def probe(domain: str) -> tuple[str, bool]:
    loop = asyncio.get_running_loop()
    try:
        await loop.getaddrinfo(domain, None)
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

该协程的作用就像是下面的函数一样，只不过在魔法的作用下，永不阻塞。

```
def probe(domain: str) -> tuple[str, bool]: # 没有async
    loop = asyncio.get_running_loop()
    try:
        loop.getaddrinfo(domain, None) # 没有await
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

`await loop.getaddrinfo(...)` 句法能避免阻塞，因为 `await` 中止当前协程对象。例如，执行 `probe('if.dev')` 的时候，`getaddrinfo('if.dev', None)` 创建一个新协程对象，等待该协程对象启动底层 `addrinfo` 查询，控制权交还事件循环，而不是被中止的 `probe('if.dev')` 协程。然后，事件循环可以驱动其他待完成的协程对象，例如 `probe('or.dev')`。

事件循环得到 `getaddrinfo('if.dev', None)` 查询的响应后，对应的协程对象恢复执行，控制权交还 `probe('if.dev')`（在 `await` 处被中止），处理可能出现的异常，返回得到的元组。

目前，我们是在协程上使用 `asyncio.as_completed` 和 `await`。其实，它们可以处理任何**可异步调用**对象，详见 21.4 节。

21.4 新概念：可异步调用对象

`for` 关键字处理**可迭代对象**，`await` 关键字处理**可异步调用对象**。

作为 `asyncio` 库的终端用户，日常可见到以下两种可异步调用对象。

- **原生协程对象**，通过调用**原生协程函数**得到。
- `asyncio.Task`，通常由把协程对象传给 `asyncio.create_task()` 得到。

然而，终端用户编写的代码不一定要使用 `await` 处理 `Task`，还可以使用 `asyncio.create_task(one_coro())` 调度 `one_coro` 并发执行，不等待它返回。我们在 `spinner_async.py` 中的 `spinner` 协程内就是这么做的（见示例 19-4）。如果不打算取消或等待任务，则无须保存 `create_task` 返回的 `Task` 对象。仅仅创建任务就能调度协程运行。

相比之下，使用 `await other_coro()` 立即运行 `other_coro`，等待协程运行完毕，因为继续向下执行之前需要协程返回的结果。在 `spinner_async.py` 中，`supervisor` 协程使用 `res = await slow()` 执行 `slow` 并获得结果。

实现异步库，或者为 `asyncio` 库做贡献时，可能还要处理以下底层的可异步调用对象。

- 提供 `__await__` 方法、返回一个迭代器的对象；例如，`asyncio.Future` 实例（`asyncio.Task` 是 `asyncio.Future` 的子类）。
- 以其他语言编写的对象，使用 Python/C API，提供 `tp_as_async.am_await` 函数（类似于 `__await__` 方法），返回一个迭代器。

现存基准代码或许还有一种可异步调用对象——**基于生成器的协程对象**——正在走弃用流程。



PEP 492 指出, `await` 表达式“使用 `yield from` 实现, 外加验证参数步骤”, 而且“`await` 只接受一个可异步调用对象”。PEP 492 没有说明实现细节, 不过引用了引入 `yield from` 的 PEP 380。

下面分析下载指定国家国旗图像脚本的 `asyncio` 版本。

21.5 使用 `asyncio` 和 HTTPX 下载

`flags_asyncio.py` 脚本会下载 20 个国家的国旗。20.2 节提到过这个脚本, 现在运用新学的概念详细分析。

自 Python 3.10 起, `asyncio` 库仅直接支持 TCP 和 UDP, 而且标准库中没有异步 HTTP 客户端和服务包。在所有 HTTP 客户端示例中, 我将使用 HTTPX。

我们将从下往上分析 `flags_asyncio.py` 脚本, 先看设置操作的几个函数, 如示例 21-2 所示。



为了确保代码易于阅读, `flags_asyncio.py` 脚本没有处理错误。介绍 `async/await` 时, 最好先把注意力集中在“主逻辑”上, 理解常规函数和协程在程序中的作用。从 21.7 节开始, 示例中将加入错误处理和更多功能。

本章和第 20 章中的 `flags_*.py` 示例共用一些代码和数据, 因此我把相关示例都放在 `example-code-2e/20-executors/getflags` 目录中了。

示例 21-2 `flags_asyncio.py`: 设置操作的函数

```
def download_many(cc_list: list[str]) -> int:           ❶
    return asyncio.run(supervisor(cc_list))             ❷
```

```
async def supervisor(cc_list: list[str]) -> int:
    async with AsyncClient() as client:
        to_do = [download_one(client, cc)
                  for cc in sorted(cc_list)]
        res = await asyncio.gather(*to_do)

    return len(res)

if __name__ == '__main__':
    main(download_many)
```

❶ 这个函数应为常规函数（不是协程），以便传给 `flags.py` 模块（见示例 20-2）中的 `main` 函数调用。

❷ 执行事件循环，驱动 `supervisor(cc_list)` 协程对象，直到协程返回。在事件循环运行期间，这行代码导致阻塞。这行代码的结果是 `supervisor` 返回的内容。

❸ `httpx` 中的异步 HTTP 客户端是 `AsyncClient` 的方法。`AsyncClient` 还是异步上下文管理器，即提供了异步设置和清理方法的上下文管理器（详见 21.6 节）。

❹ 要下载每个国旗调用一次 `download_one` 协程，构建一个协程对象列表。

❺ 等待 `asyncio.gather` 协程。`asyncio.gather` 接受的参数是一个或多个可异步调用对象，等待全部执行完毕，以可异步调用对象的提交顺序返回结果列表。

❻ `supervisor` 返回 `asyncio.gather` 返回的列表长度。

现在来看 `flags_asyncio.py` 脚本的前半部分（见示例 21-3）。我调整了协程的顺序，使其与事件循环启动协程的顺序一致，方便阅读。

示例 21-3 `flags_asyncio.py`：导入语句和负责下载的函数

```
import asyncio

from httpx import AsyncClient ❶

from flags import BASE_URL, save_flag, main ❷

async def download_one(client: AsyncClient, cc: str): ❸
    image = await get_flag(client, cc)
    save_flag(image, f'{cc}.gif')
    print(cc, end=' ', flush=True)
    return cc

async def get_flag(client: AsyncClient, cc: str) -> bytes: ❹
    url = f'{BASE_URL}/{cc}/{cc}.gif'.lower()
    resp = await client.get(url, timeout=6.1,
                             follow_redirects=True) ❺

    return resp.read() ❻
```

❶ httpx 不在标准库中，必须自行安装。

❷ 重用 flags.py 中的代码（见示例 20-2）。

❸ download_one 必须是原生协程，这样才能使用 await 处理 get_flag（处理 HTTP 请求）。然后，显示下载的国旗对应的代码，并保存图像。

❹ get_flag 需要接收发起请求的 AsyncClient。

❺ httpx.AsyncClient 实例的 get 方法返回一个 ClientResponse 对象，这也是一个异步上下文管理器。

❻ 网络 I/O 操作以协程方法实现，可由 asyncio 事件循环异步驱动。



为了提升性能，download_one 中的 save_flag 调用应该异步执行，以免阻塞事件循环。但是，目前 asyncio 没有提供（类似 Node.js 那种）异步文件系统 API。

21.7.1 节将说明如何把 `save_flag` 委托给一个线程。

我们编写的代码使用 `await` 把操作显式委托给 `httpx` 协程，或者通过异步上下文管理器（例如 `AsyncClient` 和 `ClientResponse`，详见 21.6 节）的特殊方法隐式委托。

21.5.1 原生协程的秘密：默默无闻的生成器

17.13 节中的经典协程示例和 `flags_asyncio.py` 中原生协程的关键区别在于，后者没有一目了然的 `.send()` 调用或 `yield` 表达式。你的代码位于 `asyncio` 库和你使用的异步库（例如 `HTTPX`）之间，如图 21-1 所示。

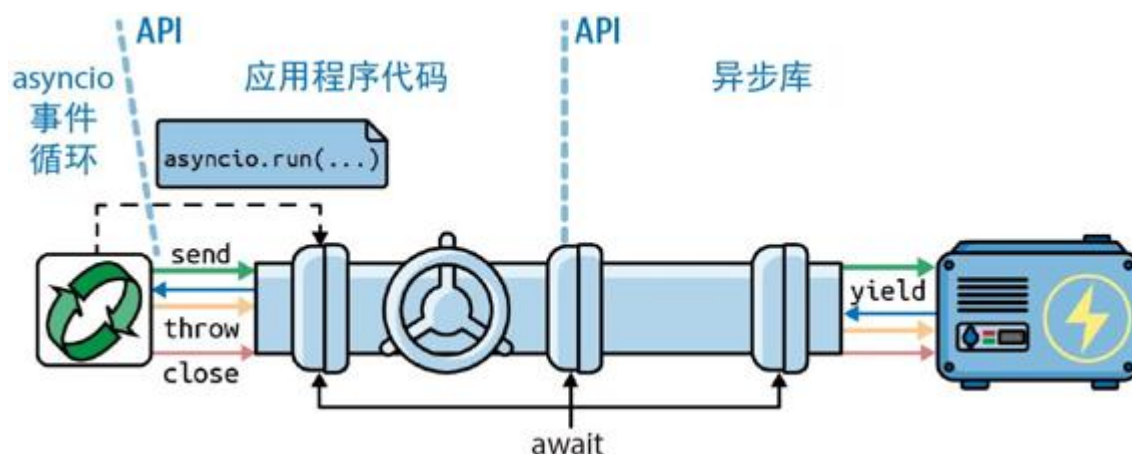


图 21-1：在异步程序中，用户编写的函数启动事件循环，使用 `asyncio.run` 调度最初的协程。用户编写的各个协程使用 `await` 表达式驱动下一步，构成一个通道，打通 `HTTPX` 等库与事件循环

`asyncio` 事件循环在背后调用 `.send` 驱动你的协程，而你的协程使用 `await` 等待其他协程，包括库提供的协程。前文说过，`await` 的实现大量借鉴 `yield from`，也调用 `.send` 驱动协程。

`await` 链最终到达一个底层可异步调用对象，返回一个生成器，由事件循环驱动，对计时器或网络 I/O 等事件做出响应。位于 `await` 链末

端的底层可异步调用对象深埋在库的实现中，不作为 API 开放，有可能是 Python/C 扩展。

使用 `asyncio.gather` 和 `asyncio.create_task` 等函数可以启动多个并发 `await` 通道，在单个线程内由单个事件循环驱动多个 I/O 操作并发执行。

21.5.2 “不成功便成仁”问题

注意，在示例 21-3 中不能重用 `flags.py`（见示例 20-2）中的 `get_flag` 函数。为了使用 HTTPX 提供的异步 API，必须将其重写为协程。为了充分发挥 `asyncio` 的性能，必须把执行 I/O 操作的每个函数替换为异步版本，使用 `await` 或 `asyncio.create_task` 激活，这样在函数等待 I/O 期间才能把控制权交还给事件循环。如果无法把导致阻塞的函数重写为协程，那就应该在单独的线程或进程中运行那个函数，详见 21.8 节。

正是因为这样，我才选择在本章开头引用那句话。请记住这个建议：“你孤注一掷重写代码，要么彻底避免阻塞，要么纯属浪费时间。”

出于同样的原因，也不能重用 `flags_threadpool.py`（见示例 20-3）中的 `download_one` 函数。示例 21-3 中的代码使用 `await` 驱动 `get_flag`，所以 `download_one` 也必须是一个协程。在 `supervisor` 中，每次请求创建一个 `download_one` 协程对象，这些对象全由 `asyncio.gather` 协程驱动。

接下来分析 `supervisor`（见示例 21-2）中的 `async with` 语句和 `get_flag` 协程（见示例 21-3）。

21.6 异步上下文管理器

在 18.2 节，我们了解到，在 `with` 块主体的前后可以使用一个对象运行代码，前提是那个对象所属的类提供了 `__enter__` 和 `__exit__` 方法。

下面来看示例 21-4。这段代码摘自兼容 `asyncio` 的 PostgreSQL 驱动 `asyncpg` 的事务文档。

示例 21-4 摘自 PostgreSQL 驱动 `asyncpg` 文档中的示例代码

```
tr = connection.transaction()
await tr.start()
try:
    await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
except:
    await tr.rollback()
    raise
else:
    await tr.commit()
```

数据库事务特别适合使用上下文管理器协议：事务务必启动，数据由 `connection.execute` 改动，然后根据改动的结果，必须回滚或提交。

在 `asyncpg` 这种异步驱动中，设置和清理需要由协程执行，好让其他操作有机会并发执行。然而，传统的 `with` 语句采用的实现方式不支持协程使用 `__enter__` 或 `__exit__` 执行相关操作。

鉴于此，“PEP 492—Coroutines with `async` and `await` syntax”引入了 `async with` 语句，用于实现异步上下文管理器，即一种以协程实现 `__aenter__` 和 `__aexit__` 方法的对象。

使用 `async with`，示例 21-4 可以写成如下形式（同样摘自 `asyncpg` 文档）。

```
async with connection.transaction():
    await connection.execute("INSERT INTO mytable VALUES (1, 2, 3)")
```


在 `asyncpg.Transaction` 类中，协程方法 `__aenter__` 使用 `await self.start()`，协程方法 `__aexit__` 根据有没有异常发生，异步等待私有协程方法 `__rollback` 或 `__commit`。使用协程把 `Transaction` 实现为异步上下文管理器，`asyncpg` 就能并发处理多个事务。



Caleb Hattingh 对 `asyncpg` 的点评

`asyncpg` 还有一个非常优秀的功能：实现了一个连接池，供 `Postgres` 内部连接使用，补足了 `PostgreSQL` 缺少高并发支持的劣势（`PostgreSQL` 在服务器端为每个连接创建一个进程）。

这意味着，我们不用像 `asyncpg` 文档所说的那样使用 `pgbouncer` 等额外工具。⁴

⁴这个提示栏直接引用技术审校 Caleb Hattingh 的一条点评，只字未改。谢谢 Caleb！

回到 `flags_asyncio.py`，`httpx` 提供的 `AsyncClient` 类是一个异步上下文管理器，因此可在特殊的协程方法 `__aenter__` 和 `__aexit__` 中使用可异步调用对象。



20.10.1 节的“异步生成器用作上下文管理器”将说明如何使用 Python 的 `contextlib` 创建异步上下文管理器，无须编写类。这些内容放在本章后半部分是因为在 21.10.1 节需要讲解一些预备知识。

接下来，我们将增强 `asyncio` 版国旗下载示例的功能，增加进度条。在这个过程中，我们将继续探索 `asyncio` API。

21.7 增强 `asyncio` 版下载脚本的功能

在 20.5 节, `flags2` 系列示例具有相同的命令行界面, 而且在下载过程中显示一个进度条。另外, 还有错误处理结构。



 建议你自己研究一下 `flags2` 系列示例，直观感受并发 HTTP 客户端的运作。使用 `-h` 选项查看示例 20-10 所示的帮助界面。使用 `-a`、`-e` 和 `-l` 命令行选项控制下载数量，使用 `-m` 选项设置并发下载数。LOCAL、REMOTE、DELAY 和 ERROR 服务器都测试一下，找出能最大程度发挥各个服务器吞吐量的最佳并发下载数。如果想调整各测试服务器的选项，请参考 20.5 节的“搭建测试服务器”提示栏。

例如，示例 21-5 使用 100 个并发请求（`-m 100`）尝试从 `ERROR` 服务器中获取 100 个国旗图像（`-al 100`），结果得到 48 个错误，要么是 HTTP 418，要么是超时错误——`slow_server.py` 的预期（不良）行为。

示例 21-5 运行 flags2 asyncio.py

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8002/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
100%|██████████████████████████████████████████████████████████████████████████████| 100/100
[00:03<00:00, 30.48it/s]
-----
    52 flags downloaded.
    48 errors.
Elapsed time: 3.31s
```



测试并发客户端时要有责任心

尽管线程版和 `asyncio` 版 HTTP 客户端总的下载用时相差无几，但是 `asyncio` 版发送请求的速度更快，不经意间就会对服务器发起 DoS 攻击。为了充分发挥这些并发客户端的潜力，请使

用本地 HTTP 服务器测试，详见 20.5 节的“搭建测试服务器”提示栏。

下面来看 `flags2_asyncio.py` 的实现。

21.7.1 使用 `asyncio.as_completed` 和一个线程

在示例 21-3 中，我们把几个协程传给 `asyncio.gather`，按照协程的提交顺序返回协程的结果构成的列表。这意味着，只有所有可异步调用对象都执行完毕后，`asyncio.gather` 才返回。然而，为了更新进度条，每有一个执行完毕就需要得到结果。

幸好，`asyncio` 也提供了 `as_completed` 生成器函数，作用与带进度条的线程池版示例（见示例 20-16）一样。

示例 21-6 是 `flags2_asyncio.py` 脚本的前半部分，定义 `get_flag` 和 `download_one` 协程。示例 21-7 是余下的代码，定义 `supervisor` 和 `download_many`。这个脚本比 `flags_asyncio.py` 长，因为增加了错误处理结构。

示例 21-6 `flags2_asyncio.py`：脚本的前半部分；余下的代码在示例 21-7 中

```
import asyncio
from collections import Counter
from http import HTTPStatus
from pathlib import Path

import httpx
import tqdm # type: ignore

from flags2_common import main, DownloadStatus, save_flag

# 默认的并发数较少，以免远程网站返回错误，
# 例如 503 - Service Temporarily Unavailable
DEFAULT_CONCUR_REQ = 5
MAX_CONCUR_REQ = 1000
```

```

async def get_flag(client: httpx.AsyncClient, ❶
                  base_url: str,
                  cc: str) -> bytes:
    url = f'{base_url}/{cc}/{cc}.gif'.lower()
    resp = await client.get(url, timeout=3.1,
follow_redirects=True) ❷
    resp.raise_for_status()
    return resp.content

async def download_one(client: httpx.AsyncClient,
                      cc: str,
                      base_url: str,
                      semaphore: asyncio.Semaphore,
                      verbose: bool) -> DownloadStatus:

    try:
        async with semaphore: ❸
            image = await get_flag(client, base_url, cc)
    except httpx.HTTPStatusError as exc: ❹
        res = exc.response
        if res.status_code == HTTPStatus.NOT_FOUND:
            status = DownloadStatus.NOT_FOUND
            msg = f'not found: {res.url}'
        else:
            raise
    else:
        await asyncio.to_thread(save_flag, image, f'{cc}.gif') ❺
        status = DownloadStatus.OK
        msg = 'OK'
    if verbose and msg:
        print(cc, msg)
    return status

```

❶ `get_flag` 与示例 20-14 中的顺序下载版相差不大。第一个区别：这里需要 `client` 参数。

❷ 第二个和第三个区别：`.get` 是 `AsyncClient` 提供的方法，而且是协程，因此需要使用 `await`。

❸ 把 `semaphore` 当作异步上下文管理器使用，防止整个程序出现阻塞。当信号量计数器为零时，只有这个协程中止。详见“Python 中的信号量”附注栏。

❹ 错误处理逻辑与示例 20-14 中的 `download_one` 一样。

⑤ 保存图像是 I/O 操作。为免阻塞事件循环，在一个线程中运行 `save_flag`。

所有网络 I/O 都使用 `asyncio` 库提供的协程处理，而文件 I/O 不能这么做，因为文件 I/O 也是“阻塞”操作——读写文件用时比读写 RAM 长上千倍。就算是网络附属存储（Network-Attached Storage），背后可能也涉及网络 I/O。

从 Python 3.9 开始，`asyncio.to_thread` 协程可以轻松地把文件 I/O 委托给 `asyncio` 提供的一个线程池。如果需要支持 Python 3.7 或 3.8，则需要多增加几行代码，详见 21.8 节。在此之前，我们要完成对这个 HTTP 客户端代码的分析。

21.7.2 使用信号量限制请求

像我们分析的这种网络客户端应该限流，以免对服务器发起过多并发请求。

信号量是同步原语，比时钟灵活。信号量可以配置最大数量，而且一个信号量可由多个协程持有，因此特别适合用于限制活动的并发协程数量。进一步说明见“Python 中的信号量”附注栏。

在 `flags2_threadpool.py` 中（见示例 20-16），我们在 `download_many` 中实例化 `ThreadPoolExecutor`，把必须提供的参数 `max_workers` 设为 `concur_req`，起到限流作用。在 `flags2_asyncio.py` 中，我们在 `supervisor` 函数（见示例 21-7）中创建 `asyncio.Semaphore`，作为 `semaphore` 参数传给 `download_one`（见示例 21-6）。

Python 中的信号量

信号量由计算机科学家 Edsger W. Dijkstra 在 20 世纪 60 年代发明，背后的思想很简单，但是十分灵活，其他多数同步对象（例如锁和栅栏）都可以在此基础上构建。Python 标准库中有 3 个

Semaphore 类, threading、multiprocessing 和 asyncio 中各有一个。这里讲的是最后一个。

asyncio.Semaphore 有一个内部计时器。每次使用 await 处理协程方法 .acquire(), 计时器递减; 每次调用 .release() 方法 (不是协程, 因为永不阻塞), 计时器递增。计时器的初始值在实例化 Semaphore 时设定。

```
semaphore = asyncio.Semaphore(concur_req)
```

若计时器大于零, 则使用 await 处理 .acquire() 方法没有延迟; 若计时器为零, 则 .acquire() 中止待处理的协程, 直到其他协程在同一个 Semaphore 实例上调用 .release(), 递增计时器。一般不直接调用这些方法, 把 semaphore 当作异步上下文管理器使用更安全。在示例 21-6 中, download_one 函数就是这么做的。

```
async with semaphore:
    image = await get_flag(client, base_url, cc)
```

协程方法 Semaphore.__aenter__ 异步等待 .acquire(), 协程方法 __aexit__ 调用 .release()。上述代码片段可以确保 et_flags 协程的数量在任意时刻都不超过 concur_req。

标准库中的各个 Semaphore 类均有一个 BoundedSemaphore 子类, 额外施加一个约束: 如果 .release() 操作数超过 .acquire() 操作数, 那么内部计时器的值不能比初始值大。⁵

⁵Guto Maia 在阅读本书第 1 版这一章的草稿时指出, 信号量这一概念没有给出解释。谢谢 Maia。

现在来看这个脚本的余下部分, 如示例 21-7 所示。

示例 21-7 flags2_asyncio.py: 接续示例 21-6

```

async def supervisor(cc_list: list[str],
                    base_url: str,
                    verbose: bool,
                    concur_req: int) -> Counter[DownloadStatus]:
    ❶
    counter: Counter[DownloadStatus] = Counter()
    semaphore = asyncio.Semaphore(concur_req) ❷
    async with httpx.AsyncClient() as client:
        to_do = [download_one(client, cc, base_url, semaphore,
verbose)

                    for cc in sorted(cc_list)] ❸
        to_do_iter = asyncio.as_completed(to_do) ❹
        if not verbose:
            to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list))
    ❺

        error: httpx.HTTPError | None = None ❻
        for coro in to_do_iter: ❼
            try:
                status = await coro ❸
            except httpx.HTTPStatusError as exc:
                error_msg = 'HTTP error {resp.status_code} -
{resp.reason_phrase}'
                error_msg = error_msg.format(resp=exc.response)
                error = exc ❹
            except httpx.RequestError as exc:
                error_msg = f'{exc} {type(exc)}'.strip()
                error = exc ❺
            except KeyboardInterrupt:
                break

            if error:
                status = DownloadStatus.ERROR ❶
                if verbose:
                    url = str(error.request.url) ❷
                    cc = Path(url).stem.upper() ❸
                    print(f'{cc} error: {error_msg}')
                counter[status] += 1

    return counter

def download_many(cc_list: list[str],
                 base_url: str,
                 verbose: bool,
                 concur_req: int) -> Counter[DownloadStatus]:
    coro = supervisor(cc_list, base_url, verbose, concur_req)
    counts = asyncio.run(coro) ❺

```

```
    return counts

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

- ❶ supervisor 接受的参数与 download_many 函数相同，但是不能由 main 直接调用，因为 download_many 是普通函数，而 supervisor 是协程。
- ❷ 创建一个 asyncio.Semaphore 实例，不允许使用这个信号量的活动协程数超过 concur_req。concur_req 的值由 flags2_common.py 中的 main 函数根据命令行选项和各个示例中设置的常量计算。
- ❸ 创建一个协程对象列表，一个元素对应一次 download_one 协程调用。
- ❹ 获取一个迭代器，返回处理完毕的协程对象。我没有把这个 as_completed 调用放在下方的 for 循环内，因为我需要根据用户选择的详细级别确定是否外套一层 tqdm 迭代器。
- ❺ 使用 tqdm 生成器函数包装 as_completed 迭代器，显示进度。
- ❻ 声明 error，初始化为 None。这个变量在 try/except 语句外部存储可能抛出的异常。
- ❼ 迭代已完成的协程对象；这个循环与示例 20-16 中 download_many 内的循环类似。
- ❽ 异步等待协程，获取结果。这一步不阻塞，因为 as_completed 只生产已完成的协程。
- ❾ 这个赋值有必要，因为 exc 变量的作用域限定在这个 except 子句中，而我需要存储它的值，供后面使用。

⑩ 同上。

⑪ 如果有错误，设置 `status`。

⑫ 在详细模式下，从抛出的异常中提取 URL……

⑬ ……再提取文件名称，在下一行显示国家代码。

⑭ `download_many` 实例化 `supervisor` 协程对象，通过 `asyncio.run` 传给事件循环，在事件循环结束后获得 `supervisor` 返回的计数器。

在示例 21-7 中，我们不能像示例 20-16 那样使用 `future` 对象到国家代码的映射，因为 `asyncio.as_completed` 返回的与传给 `as_completed` 调用的是同一批可异步调用对象。`asyncio` 的内部机制可能会把我们提供的可异步调用对象替换为其他可异步调用对象，但是最终的结果是一样的。⁶

⁶我在 `python-tulip` 群组中发起的一个话题中有详细讨论，题为“Which other futures may come out of `asyncio.as_completed`?”。Guido 在回复中剖析了 `as_completed` 的实现，还指出了在 `asyncio` 中，`future` 对象和协程存在紧密关系。



由于操作失败时不能以可异步调用对象为键从字典中获取国家代码，因此我不得不从异常中提取国家代码。为此，我把异常存储在 `try/except` 语句外部的 `error` 变量中。Python 语言不使用块级作用域，循环和 `try/except` 等语句不在它们管理的块中创建局部作用域。但是，如果 `except` 子句把异常绑定到变量上（例如刚才见到的 `exc` 变量），绑定的变量只存在于 `except` 子句所在的块内。

我们使用 `asyncio` 实现了与 `flags2_threadpool.py` 同等的功能。对这个示例的讨论到此结束。

下一个示例演示一个简单的模式：使用协程逐个执行异步任务。我觉得这个模式值得关注，因为有 JavaScript 经验的人都知道，为了在一个异步函数运行完毕后运行另一个异步函数，需要不断嵌套回调，由此产生的编程模式称为“死亡金字塔”。`await` 关键字可以根除这个模式。所以，Python 和 JavaScript 现在都提供了 `await` 关键字。

21.7.3 每次下载发起多个请求

假设保存各个国家的国旗时，除了国家代码之外，你还想加上国家名称。那么，针对每个国旗需要发起两个 HTTP 请求：一个请求下载国旗图像，另一个请求与图像在同一个目录中的 `metadata.json` 文件（记录着国家名称）。

在线程版脚本中很容易让同一个任务发起多个请求：先发起一个请求，再发起另一个请求，阻塞线程两次，把两部分数据（国家代码和名称）存储在局部变量中，在保存文件时使用。如果在异步脚本中使用回调实现同样的操作，那就需要嵌套函数，在闭包内存储国家代码和名称变量，如此才能在保存文件时使用，因为各个回调在不同的局部作用域内运行。`await` 关键字的出现拯救了我们：一个异步请求结束后驱动另一个请求，共用驱动协程的局部作用域。



使用现代 Python 做异步应用程序编程，如果大量使用回调，那就说明你使用的可能是旧的编程模式，已不适应现代 Python 环境。如果你编写的库需要兼容不支持协程的遗留或底层代码，那就没问题。Stack Overflow 中的“What is the use case for `future.add_done_callback()`?”问题解说了为什么底层代码需要回调，以及为什么如今的 Python 应用程序级别代码不太需要它了。

使用 `asyncio` 实现的第三版下载脚本有几处变化。

```
get_country
```



```

try:
    async with semaphore: ❶
        image = await get_flag(client, base_url, cc)
    async with semaphore: ❷
        country = await get_country(client, base_url, cc)
except httpx.HTTPStatusError as exc:
    res = exc.response
    if res.status_code == HTTPStatus.NOT_FOUND:
        status = DownloadStatus.NOT_FOUND
        msg = f'not found: {res.url}'
    else:
        raise
else:
    filename = country.replace(' ', '_') ❸
    await asyncio.to_thread(save_flag, image,
f'{filename}.gif')
    status = DownloadStatus.OK
    msg = 'OK'
    if verbose and msg:
        print(cc, msg)
    return status

```

❶ 持有 semaphore, 异步等待 get_flag.....

❷ 同样, 异步等待 get_country.

❸ 使用国家名称创建文件名。作为命令行用户, 我不喜欢在文件名中看到空格。

这比嵌套回调好多了!

我把 get_flag 和 get_country 分开放在两个使用 semaphore 控制的 with 块中, 因为持有信号量和锁的时间越短越好。

get_flag 和 get_country 也可以使用 asyncio.gather 并行调度, 但若 get_flag 抛出异常, 则没有图像需要保存, 此时运行 get_country 没有意义。不过, 有些时候应该使用 asyncio.gather 同时请求多个 API, 而不是等一个请求得到响应之后再发起下一个请求。

在 `flags3_asyncio.py` 中, `await` 句法出现 6 次, `async with` 出现 3 次。希望你已经掌握 Python 异步编程的要领。有些人可能不确定何时应该使用 `await`, 何时不应该使用。其实, 答案很简单: 协程和其他可异步调用对象, 例如 `asyncio.Task` 实例, 应该使用 `await`。但是, 有些 API 比较混乱, 把协程和常规函数混在一起, 没有章法可循, 比如示例 21-14 中用到的 `StreamWriter` 类。

分析过示例 21-9 之后, 我们对 `flags` 系列示例的讨论结束了。接下来探讨如何在异步编程中使用线程或进程执行器。

21.8 把任务委托给执行器

与 Python 相比, Node.js 在异步编程上的显著优势体现在 Node.js 标准库上: 为所有 I/O 提供了异步 API, 而不仅限于网络 I/O。在 Python 中, 如果你不够小心, 那么文件 I/O 可以明显降低异步应用程序的性能水平, 因为在主线程中读写存储器会阻塞事件循环。

在示例 21-6 中, `download_one` 协程使用下面这行代码把下载的图片保存在磁盘中。

```
await asyncio.to_thread(save_flag, image, f'{cc}.gif')
```

前文说过, `asyncio.to_thread` 在 Python 3.9 中增加。如果想支持 Python 3.7 或 3.8, 则需要把那一行替换成示例 21-10 中的三行。

示例 21-10 替换 `await asyncio.to_thread` 的三行

```
loop = asyncio.get_running_loop()           ❶  
loop.run_in_executor(None, save_flag,       ❷  
                        image, f'{cc}.gif')  ❸
```

❶ 获取事件循环的引用。

❷ 第一个参数是要使用的执行器。这里传入 `None`，使用在 `asyncio` 事件循环中始终可用的默认执行器 `ThreadPoolExecutor`。

❸ 可以通过位置参数传入要运行的函数，但是如果传入关键字参数，则需要利用 `functool.partial`，详见 `run_in_executor` 文档。

新增的 `asyncio.to_thread` 函数用起来更简单，也更灵活，因为它也接受关键字参数。

`asyncio` 自身的实现在背后多次使用 `run_in_executor`。例如，示例 21-1 中的 `loop.getaddrinfo(...)` 协程通过调用 `socket` 模块中的 `getaddrinfo` 函数实现，而这个函数要阻塞几秒钟才返回，具体时间取决于 DNS 解析速度。

异步 API 经常在内部使用 `run_in_executor` 把实现细节中的阻塞调用包装成协程。如此一来，协程的接口保持了一致，都使用 `await` 驱动，而且用到的线程被隐藏起来了，简单纯粹。MongoDB 的异步驱动 `Motor` 提供了与 `async/await` 兼容的 API，把与数据库服务器通信的线程版核心包装起来。`Motor` 的首席开发者 A. Jesse Jiryu Davis 在“Response to ‘Asynchronous Python and Databases’”一文中解释了背后的原因。内容剧透：Davis 发现对数据库驱动这个特定的使用场景来说，线程池的性能更好——尽管人们普遍认为对网络 I/O 来说，异步方案总是比线程的速度快。

把一个特定的执行器显式传给 `loop.run_in_executor`，主要是为了使用 `ProcessPoolExecutor` 在不同的 Python 进程中运行 CPU 密集型函数，以避免争用 GIL。由于启动进程的开销较大，因此最好在 `supervisor` 中启动 `ProcessPoolExecutor`，再把它传给需要用到的协程。

本书的技术审校之一 Caleb Hattingh (*Using Asyncio in Python* 一书的作者) 建议我添加下面的警告栏，指出在 `asyncio` 中使用执行器的注意事项。



Caleb 对 `run_in_executor` 的警告

使用 `run_in_executor` 可能导致难以调试的问题，因为撤销行为与大多数人的预期不符。使用执行器的协程给人一种可以撤销的假象，其实底层线程（假如使用的是 `ThreadPoolExecutor`）没有撤销机制。例如，在 `run_in_executor` 调用中创建的长时间运行的线程阻碍 `asyncio` 程序正常关闭：`asyncio.run` 等待执行器完全关闭才返回，如果执行器作业出于什么原因没有停止，那它将一直等下去。在我这个老头儿看来，那个函数应该命名为 `run_in_executor_uncancellable`。

现在，将我们的视线从客户端脚本上移开，转向使用 `asyncio` 编写服务器。

21.9 使用 `asyncio` 编写服务器

演示 TCP 服务器时通常使用回显服务器。我们将要构建的两个示例服务器有点好玩儿，用于搜索 Unicode 字符，第一个服务器使用 FastAPI 处理 HTTP，第二个服务器仅使用 `asyncio` 处理 TCP。

用户可以通过 `unicodedata` 模块（详见 4.9 节）中 Unicode 字符标准名称所包含的单词，在这两个服务器中查询 Unicode 字符。图 21-2 是在 `web_mojifinder.py`（我们要构建的第一个服务器）中查询的某次会话。

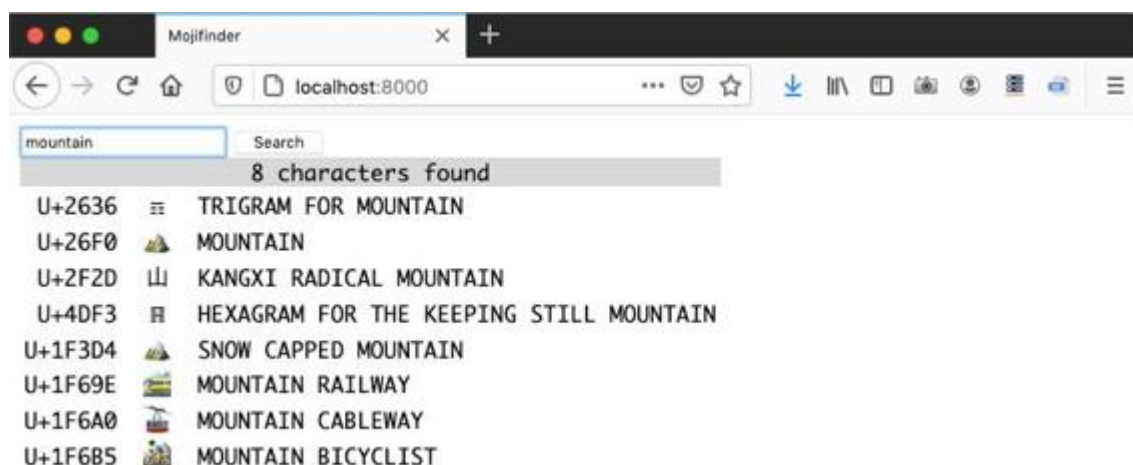


图 21-2：浏览器窗口中显示了在 web_mojifinder.py 服务器中搜索“mountain”得到的结果

这些示例中用到的 Unicode 搜索逻辑在 `InvertedIndex` 类中，我把它所在的模块 `charindex.py` 放到了本书代码中。这个模块内容不多，不涉及并发，下方附注栏做了简要说明，请根据实际情况选读。如果觉得没必要阅读，可以直接跳到 21.9.1 节，了解 HTTP 服务器的实现。

倒排索引简介

倒排索引通常把单词映射到包含相应单词的文档上。在 `mojifinder` 示例中，“文档”是 Unicode 字符。

`charindex.InvertedIndex` 类为 Unicode 数据库中各个字符名称中出现的每个单词建立索引，把倒排索引存储在一个 `defaultdict` 中。例如，索引 U+0037 字符 (DIGIT SEVEN) 时，`InvertedIndex` 初始化方法把字符 '7' 追加到 `defaultdict` 中，放在 'DIGIT' 和 'SEVEN' 这两个键名下。把 Python 3.9.1 自带的 Unicode 13.0.0 数据索引完毕后，'DIGIT' 对应 868 个字符，'SEVEN' 对应 143 个字符，包括 U+1F556 (CLOCK FACE SEVEN OCLOCK) 和 U+2790 (DINGBAT NEGATIVE CIRCLED SANS-SERIF DIGIT SEVEN，出现在本书多个代码清单中)。

查看 'CAT' 和 'FACE' 条目得到的结果如图 21-3 所示。⁷

```
>>> from charindex import InvertedIndex
>>> idx.entries['CAT']
{'😺', '😺', '찰', '☐', '😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺'}
>>> len(idx.entries['FACE'])
171
>>> idx.entries['FACE'] & idx.entries['CAT']
{'😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺'}
>>> idx.search('cat face')
{'😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺', '😺'}
>>>
```

图 21-3：在 Python 控制台中探索 InvertedIndex 类的 entries 属性和 search 方法

InvertedIndex.search 方法把查询词条拆分成一个个单词，返回各个单词对应条目的交集。所以，搜索“face”得到 171 个结果，搜索“cat”得到 14 个结果，而搜索“cat face”只得到 10 个结果。

这就是倒排索引背后的思想。倒排索引是信息检索的基石，是搜索引擎背后的理论基础。

⁷截图中的问号框不是你看的纸质书或电子书的问题，那原本是 U+101EC (PHAISTOS DISC SIGN CAT) 字符，我的终端使用的字体不含该字符，因此无法正常显示。费斯托斯圆盘 (Phaistos Disc) 是一件刻有象形文字的古代艺术品，发现于克里特岛。

21.9.1 一个 FastAPI Web 服务

接下来的示例 (web_mojifinder.py) 使用 FastAPI 编写，即 19.7.4 节中“异步服务器网关接口 (ASGI)”提示框提到的一个 Python ASGI Web 框架。图 21-2 是前端的一个截图。其实，这是一个特别简单的单页应用程序 (single page application, SPA)，HTML 下载完毕后，由客户端 JavaScript 与服务器通信更新 UI。

FastAPI 旨在为单页应用程序和移动应用程序实现后端，基本上都是返回 JSON 响应的 Web API 端点，不在服务器端渲染 HTML。借助装饰器、类型提示和代码内省，FastAPI 可以削减 Web API 的大量样

板代码。另外，FastAPI 还能为我们创建的 API 自动发布交互式 OpenAPI（即 Swagger）文档。图 21-4 是为 web_mojifinder.py 自动生成的 /docs 页面。

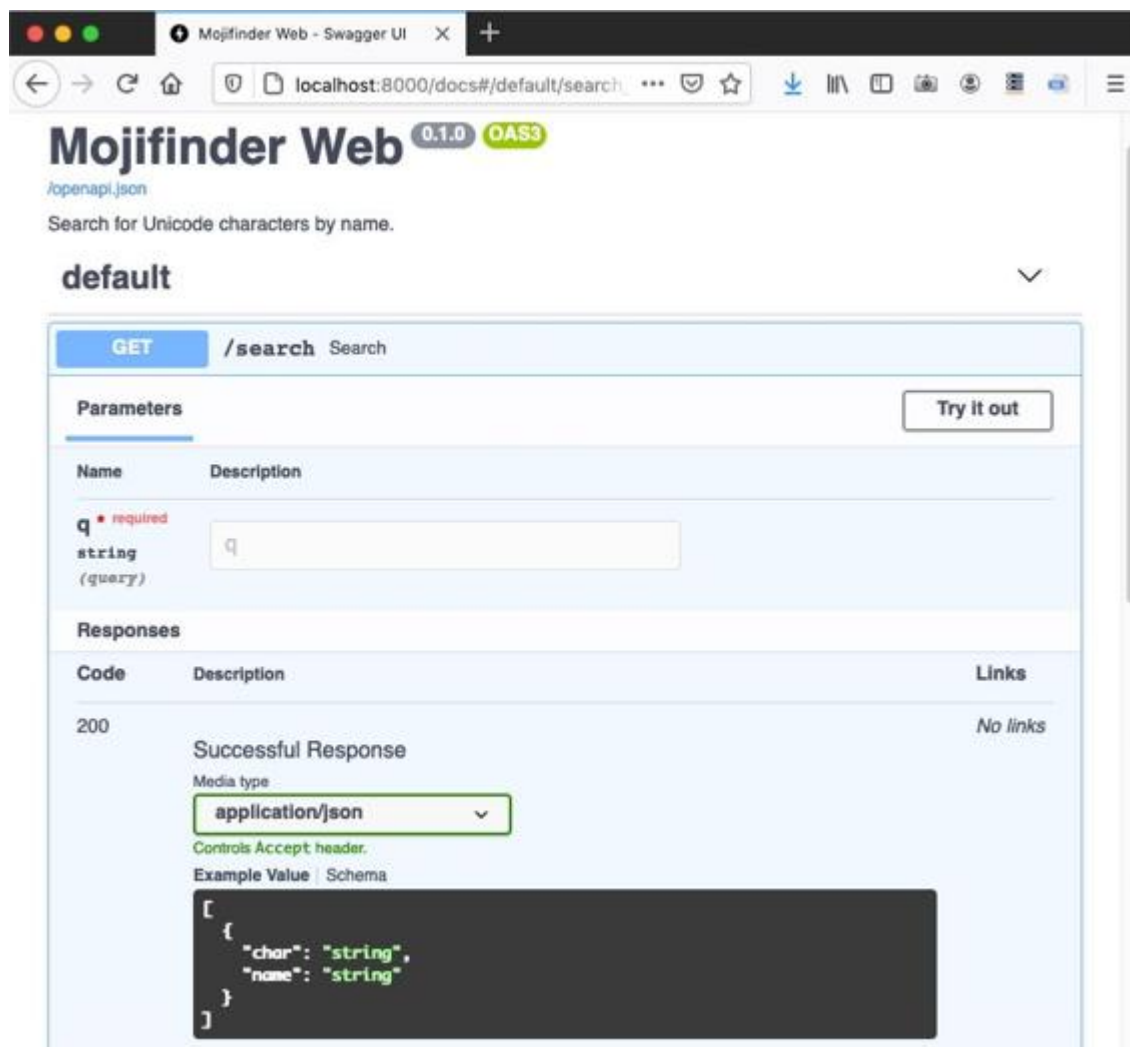


图 21-4：为 /search 端点自动生成的 OpenAPI 模式

web_mojifinder.py 的代码在示例 21-11 中，但是这里只有后端代码。访问根 URL/ 时，服务器发送 form.html 文件。这个文件有 81 行代码，其中 54 行是 JavaScript 代码，负责与服务器通信，把结果填入表格。如果你想阅读没有使用框架的纯 JavaScript 代码，请在本书代码中找到 21-async/mojifinder/static/form.html。

运行 `web_mojifinder.py` 之前，需要安装两个包及其依赖：FastAPI 和 uvicorn。⁸ 使用 uvicorn 在开发模式下运行示例 21-11 的命令如下所示。

⁸除了 uvicorn，使用其他 ASGI 服务器也可以，例如 hypercorn 或 Daphne。详见 ASGI 文档中罗列实现的页面。

```
$ uvicorn web_mojifinder:app --reload
```

传入的参数列举如下。

```
web_mojifinder:app
```

包名、一个冒号和包内定义的 ASGI 应用程序的名称（时常命名为 `app`）。

```
--reload
```

让 uvicorn 监控应用程序源文件的变化，自动重新加载。只在开发过程中有用。

下面分析 `web_mojifinder.py` 的源码。

示例 21-11 web_mojifinder.py：完整源码

```
from pathlib import Path
from unicodedata import name

from fastapi import FastAPI
from fastapi.responses import HTMLResponse
from pydantic import BaseModel

from charindex import InvertedIndex

STATIC_PATH = Path(__file__).parent.absolute() / 'static' ❶

app = FastAPI( ❷
    title='Mojifinder Web',
    description='Search for Unicode characters by name.',
)
```

```

class CharName(BaseModel): ❸
    char: str
    name: str

def init(app): ❹
    app.state.index = InvertedIndex()
    app.state.form = (STATIC_PATH / 'form.html').read_text()

init(app) ❺

@app.get('/search', response_model=list[CharName]) ❻
async def search(q: str): ❼
    chars = sorted(app.state.index.search(q))
    return ({'char': c, 'name': name(c)} for c in chars) ❽

@app.get('/', response_class=HTMLResponse,
include_in_schema=False)
def form(): ❾
    return app.state.form

# 没有主函数 ❿

```

❶ 与本章的话题无关，不过值得提一下：pathlib 重载了 / 运算符，写出的代码很优雅。⁹

⁹感谢技术审校 Miroslav Šedivý 指出在这个代码示例中适合使用 pathlib。

❷ 这一行定义 ASGI 应用程序。可以直接写成 `app = FastAPI()`。提供的参数是自动生成的文档的元数据。

❸ 一个 JSON 响应的 pydantic 模式，有 `char` 和 `name` 两个字段。¹⁰

¹⁰第 8 章讲过，pydantic 在运行时通过类型提示验证数据。

❹ 构建 `index`，加载静态 HTML 表单，都依附在 `app.state` 上，供后面使用。

❺ ASGI 服务器加载这个模块时运行 `init`。

- ⑥ `/search` 端点的路由。 `response_model` 使用前面定义的 `pydantic` 模型 `CharName` 描述响应格式。
- ⑦ FastAPI 假定函数或协程签名中不在路由路径内的参数都是 HTTP 查询字符串，例如 `/search?q=cat`。 `q` 没有默认值，如果查询字符串中缺少 `q`，则 FastAPI 返回 422 (Unprocessable Entity) 响应。
- ⑧ 返回由字典构成的可迭代对象，与 `response_model` 模式兼容，因此 FastAPI 根据 `@app.get` 装饰器中的 `response_model` 构建 JSON 响应。
- ⑨ 常规函数（即非异步运行）也可用于生成响应。
- ⑩ 这个模块没有主函数，由 ASGI 服务器（本例中是 `uvicorn`）加载和驱动。

示例 21-11 没有直接调用 `asyncio`。 FastAPI 构建在 ASGI 工具包 `Starlette` 之上，后者使用了 `asyncio`。

另外要注意， `search` 的主体中没有 `await`、 `async with` 或 `async for`，因此也可以定义为普通函数。我之所以把 `search` 定义为协程，是为了告诉你， FastAPI 知道如何处理。在真实的应用中，多数端点要查询数据库或请求远程服务器，因此 FastAPI（以及其他 ASGI 框架）务必支持协程，以便利用异步库处理网络 I/O。



我编写的 `init` 和 `form` 函数用于加载和伺服静态 HTML 表单，这其实是一种捷径，可以让示例保持简短、易于运行。推荐的最佳做法是在 ASGI 服务器前面放一个代理（或负载均衡程序），处理所有静态资源；如果可能的话，还应该使用内容分发网络（content delivery network, CDN）。 `Traefik` 就是一个代理（负载均衡程序），自称是一个“边缘路由器”，“代表你的系统接收请求，找出负责处理请求的组件”。 FastAPI 提供的项目生成脚本能为你准备好相关代码。

类型狂热人士或许注意到了，`search` 和 `form` 没有返回值类型提示。其实，FastAPI 依靠路由装饰器中的关键字参数 `response_model=` 判断类型。FastAPI 文档中的“Response Model”页面指出：

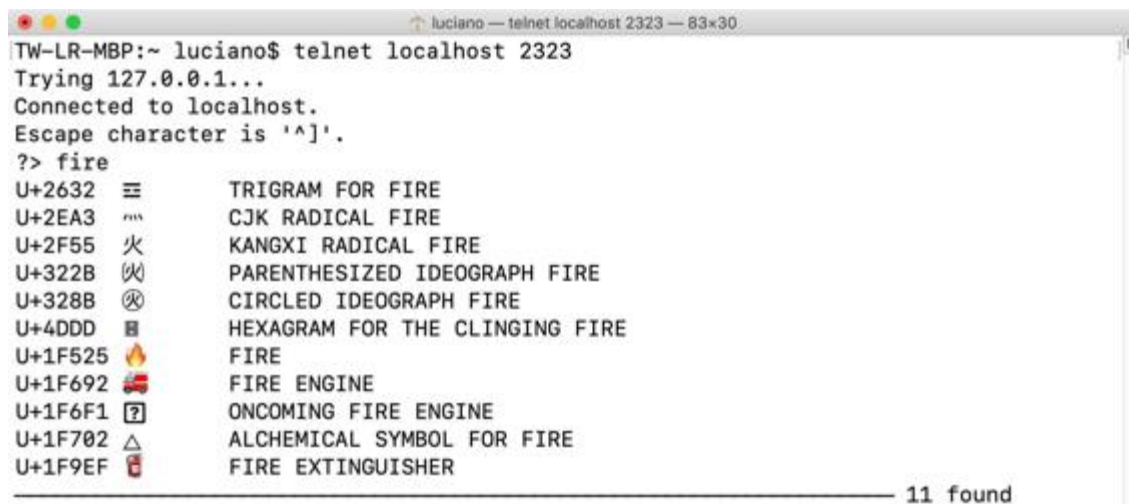
响应模型在这个参数中声明，而不通过函数的返回值类型注解，因为处理路径的函数可能根本不返回响应模型，而是返回一个字典、数据库对象或其他模型，然后使用 `response_model` 限定字段和序列化。

例如，在 `search` 中，我返回一个生成器，产出的项是字典，而不是 `CharName` 对象构成的列表，但是 FastAPI 和 `pydantic` 能正确验证数据，构建与 `response_model=list[CharName]` 相容的 JSON 响应。

接下来把注意力移到响应查询的 `tcp_mojifinder.py` 脚本上，效果如图 21-5 所示。

21.9.2 一个使用 `asyncio` 编写的 TCP 服务器

`tcp_mojifinder.py` 程序通过 TCP 与 Telnet 或 Netcat 等客户端通信，因此无须借助外部依赖，使用 `asyncio` 就能写出来，而且不用重新发明 HTTP。图 21-5 是 `tcp_mojifinder.py` 基于文本的 UI。



```
luciano — telnet localhost 2323 — 83x30
TW-LR-MBP:~ luciano$ telnet localhost 2323
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
?> fire
U+2632  ≡      TRIGRAM FOR FIRE
U+2EA3  𐄃      CJK RADICAL FIRE
U+2F55  火      KANGXI RADICAL FIRE
U+322B  𤇑      PARENTHEZIZED IDEOGRAPH FIRE
U+328B  𤇓      CIRCLED IDEOGRAPH FIRE
U+4DDD  𠩺      HEXAGRAM FOR THE CLINGING FIRE
U+1F525  🔥      FIRE
U+1F692  🚒      FIRE ENGINE
U+1F6F1  🔥      ONCOMING FIRE ENGINE
U+1F702  ⚗      ALCHEMICAL SYMBOL FOR FIRE
U+1F9EF  🧯      FIRE EXTINGUISHER
11 found
```

图 21-5: 在一个 Telnet 会话中向 tcp_mojifinder.py 服务器查询“fire”

这个程序的长度是 web_mojifinder.py 的 2 倍，将分成三部分讲解，分别是示例 21-12、示例 21-14 和示例 21-15。tcp_mojifinder.py 的前半部分（包含 import 语句）在示例 21-14 中，不过我决定先分析驱动程序运行的 supervisor 协程和 main 函数。

示例 21-12 tcp_mojifinder.py: 一个简单的 TCP 服务器；接续示例 21-14

```
async def supervisor(index: InvertedIndex, host: str, port: int) -> None:
    server = await asyncio.start_server(
        functools.partial(finder, index),
        host, port)

    socket_list = cast(tuple[TransportSocket, ...],
server.sockets)
    addr = socket_list[0].getsockname()
    print(f'Serving on {addr}. Hit CTRL-C to stop.')
    await server.serve_forever()

def main(host: str = '127.0.0.1', port_arg: str = '2323'):
    port = int(port_arg)
    print('Building index.')
    index = InvertedIndex()
    try:
        asyncio.run(supervisor(index, host, port))
    except KeyboardInterrupt:
        print('\nServer shut down.')

if __name__ == '__main__':
    main(*sys.argv[1:])
```

❶ await 很快得到一个 asyncio.Server 实例，即一个 TCP 套接字服务器。默认情况下，start_server 创建并启动服务器，随即就能接收连接。

❷ `start_server` 的第一个参数 `client_connected_cb` 是一个回调，在客户端发起新连接时运行。这个回调可以是普通函数，也可以是协程，不过必须接受两个参数，不多不少，一个是 `asyncio.StreamReader` 对象，另一个是 `asyncio.StreamWriter` 对象。但是，我编写的 `finder` 协程还需要获取 `index`，因此我使用 `functools.partial` 绑定该参数，得到一个接受 `asyncio.StreamReader` 和 `asyncio.StreamWriter` 对象的回调。为适应回调 API 而改造用户函数是 `functools.partial` 最常见的用途之一。

❸ `start_server` 的第二个和第三个参数是 `host` 和 `port`。完整的签名见 `asyncio` 文档。

❹ 这里有必要调用 `cast`，因为截至 2021 年 5 月，`typeshed` 为 `Server` 类的 `sockets` 特性提供的类型提示还是过时的。详见 `typeshed` 项目的 5535 号工单。¹¹

¹¹ 5535 号工单在 2021 年 10 月已关闭，但是那时 `Mypy` 没有发布新版，所以这个问题还在。

❺ 显示服务器第一个套接字的地址和端口。

❻ 虽然 `start_server` 是以并发任务启动服务器的，但这里还是要使用 `await` 处理 `server_forever` 方法，目的是让我实现的 `supervisor` 协程在这里中止。倘若没有这一行，`supervisor` 将立即返回，终止 `asyncio.run(supervisor(...))` 启动的循环，导致程序退出。`Server.serve_forever` 的文档指出，“服务器已经开始接收连接后，可以调用这个方法”。

❼ 构建倒排索引。¹²

¹² 技术审校 Leonardo Rochaël 指出，可以在 `supervisor` 协程中使用 `loop.run_with_executor()` 委托另一个线程构建索引。这样，一旦索引构建完毕，服

务器即可接收请求。没错，但是这个服务器唯一的工作就是查询索引，所以对这个示例来说收效不大。

⑧ 启动事件循环，运行 supervisor。

⑨ 捕获 KeyboardInterrupt，防止在终端按 Ctrl-C 停止服务器时输出太多调用跟踪，扰乱视线。

观察服务器端控制台生成的输出或许更容易理解 tcp_mojifinder.py 的控制权流动情况，如示例 21-13 所示。

示例 21-13 tcp_mojifinder.py：图 21-5 中的会话在服务器端产生的输出

```
$ python3 tcp_mojifinder.py
Building index. ❶
Serving on ('127.0.0.1', 2323). Hit Ctrl-C to stop. ❷
  From ('127.0.0.1', 58192): 'cat face' ❸
    To ('127.0.0.1', 58192): 10 results.
  From ('127.0.0.1', 58192): 'fire' ❹
    To ('127.0.0.1', 58192): 11 results.
  From ('127.0.0.1', 58192): '\x00' ❺
Close ('127.0.0.1', 58192). ❻
^C ❼
Server shut down. ❽
$
```

❶ 由 main 输出。在我的设备中，下一行显示之前有 0.6 秒延迟，这是构建索引的时间。

❷ 由 supervisor 输出。

❸ finder 中 while 循环的第一次迭代。TCP/IP 栈为我的 Telnet 客户端分配的端口是 58192。如果有多个客户端连接服务器，你会看到输出的端口各不相同。

❹ finder 中 while 循环的第二次迭代。

- ⑤ 我在客户端终端按下了 Ctrl-C, `finder` 中的 `while` 循环退出。
- ⑥ `finder` 协程显示这个消息后退出。此时, 服务器仍在运行, 准备为其他客户端提供服务。
- ⑦ 我在服务器终端按下了 Ctrl-C, `server.serve_forever` 被撤销, 终止 `supervisor` 和事件循环。
- ⑧ 由 `main` 输出。

`main` 构建索引之后启动事件循环, `supervisor` 很快显示 `Serving on...` 消息, 随即在 `await server.serve_forever()` 那一行中止。此时, 控制权流入事件循环, 保持不动, 偶尔回到 `finder` 协程, 但是需要等待网络发送或者接收数据时又把控制权交还事件循环。

在事件循环存续期间, 针对连接服务器的每一个客户端启动一个 `finder` 协程实例, 从而让这个简单的服务器可以并发处理多个客户端。这个过程一直持续, 直到服务器抛出 `KeyboardInterrupt`, 或者服务器进程被操作系统终止。

现在来看 `tcp_mojifinder.py` 的前半部分, 包括 `finder` 协程。

示例 21-14 `tcp_mojifinder.py`: 示例 21-12 之前的部分

```
import asyncio
import functools
import sys
from asyncio.trsock import TransportSocket
from typing import cast

from charindex import InvertedIndex, format_results ❶

CRLF = b'\r\n'
PROMPT = b'?> '

async def finder(index: InvertedIndex, ❷
                  reader: asyncio.StreamReader,
```

```

        writer: asyncio.StreamWriter) -> None:
    client = writer.get_extra_info('peername') ❸
    while True: ❹
        writer.write(PROMPT) # 不能使用await! ❺
        await writer.drain() # 必须使用await! ❻
        data = await reader.readline() ❼
        if not data: ❽
            break
        try:
            query = data.decode().strip() ❾
        except UnicodeDecodeError: ❿
            query = '\x00'
        print(f' From {client}: {query!r}') ⓫
        if query:
            if ord(query[:1]) < 32: ⓬
                break
            results = await search(query, index, writer) ⓭
            print(f'    To {client}: {results} results.') ⓮

    writer.close() ⓯
    await writer.wait_closed() ⓰
    print(f'Close {client}.') ⓱

```

❶ `format_results` 优化 `InvertedIndex.search` 的结果在基于文本的 UI（例如命令行或 Telnet 会话）中的显示效果。

❷ 为了把 `finder` 传给 `asyncio.start_server`，我使用 `functools.partial` 做了包装，因为服务器预期传入的协程或函数只接受 `reader` 和 `writer` 参数。

❸ 获取套接字连接的远程客户端地址。

❹ 这个循环处理一个对话，直到从客户端收到一个控制字符。

❺ `StreamWriter.write` 不是协程方法，只是普通函数。这一行发送 `?>` 提示符。

❻ `StreamWriter.drain` 刷新 `writer` 缓冲。这是一个协程，必须由 `await` 驱动。

- ⑦ `StreamWriter.readline` 是一个协程，返回 `bytes`。
- ⑧ 如果没有收到字节序列，则客户端关闭连接，从而退出循环。
- ⑨ 使用默认的 UTF-8 编码把 `bytes` 解码为 `str`。
- ⑩ 用户按下 Ctrl-C，以及 Telnet 客户端发送控制字节序列时，可能抛出 `UnicodeDecodeError`。如果遇到这个异常，则把查询替换为一个空字符，简化处理。
- ⑪ 把查询输出到服务器的控制台。
- ⑫ 如果收到控制字符或空字符，则退出循环。
- ⑬ 开始搜索。 `search` 协程的代码见下。
- ⑭ 把响应输出到服务器的控制台。
- ⑮ 关闭 `StreamWriter`。
- ⑯ 等待 `StreamWriter` 关闭。根据 `.close()` 方法的文档，这是推荐做法。
- ⑰ 把客户端会话终止的消息输出到服务器的控制台。

这个示例的最后一部分是 `search` 协程，如示例 21-15 所示。

示例 21-15 `tcp_mojifinder.py`: `search` 协程

```
async def search(query: str, ❶
                    index: InvertedIndex,
                    writer: asyncio.StreamWriter) -> int:
    chars = index.search(query) ❷
    lines = (line.encode() + CRLF for line ❸
              in format_results(chars))
    writer.writelines(lines) ❹
    await writer.drain() ❺
```

```
status_line = f'{"-" * 66} {len(chars)} found' ❹
writer.write(status_line.encode() + CRLF)
await writer.drain()
return len(chars)
```

❶ search 必须是协程，因为需要写入 StreamWriter，而且必须使用 StreamWriter 的协程方法 `.drain()`。

❷ 查询倒排索引。

❸ 这个生成器表达式产出字节字符串，使用 UTF-8 编码 Unicode 码点、字符本身、字符名称和 CRLF 序列，例如 `b'U+0039\t9\tDIGIT NINE\r\n'`。

❹ 发送 lines。奇怪的是，`writer.writelines` 不是协程。

❺ 但 `writer.drain()` 是协程。别把 `await` 忘了！

❻ 构建状态行，然后发送。

注意，在 `tcp_mojifinder.py` 中，所有网络 I/O 都是 `bytes` 类型；我们需要解码从网络接收到的 `bytes`，发送之前还要编码字符串。在 Python 3 中，默认编码是 UTF-8，本例中所有 `encode` 和 `decode` 调用也都假定使用这个编码。



注意，有些 I/O 方法是协程，必须使用 `await` 驱动，而另一些是普通函数。例如，`StreamWriter.write` 是普通函数，因为它写入缓冲；而 `StreamWriter.drain`（刷新缓冲，执行网络 I/O）是协程，`StreamReader.readline` 也是协程，但 `StreamWriter.writelines` 不是。写作本书第 1 版的过程中，`asyncio` API 文档已经改进，明确指出哪些是协程。

`tcp_mojifinder.py` 利用 `asyncio` 提供的高层 Streams API，我们只需实现一个处理函数（可以是普通回调或协程）就能得到一个可用的服

务器。此外，受 Twisted 框架对传输和协议抽象的启发，`asyncio` 也有底层 Transports and Protocols API，详见 `asyncio` 文档，文档中还使用底层 API 实现了 TCP 及 UDP 回显服务器和客户端。

下一个话题是 `async for` 及其背后的对象。

21.10 异步迭代和异步可迭代对象

21.6 节讲过，`async with` 可以处理实现了 `__aenter__` 和 `__aexit__` 方法的对象，二者返回可异步调用对象，通常是协程对象。

类似地，`async for` 处理**异步可迭代对象**，即实现了 `__aiter__` 的对象。然而，`__aiter__` 必须是常规方法（不是协程方法），而且必须返回一个**异步迭代器**。

异步迭代器提供 `__anext__` 协程方法，返回一个可异步调用对象，通常是一个协程对象。异步迭代器也应实现 `__aiter__`，往往返回 `self`。这与 17.5.2 节所讲的可迭代对象与迭代器之间的区别是一样的。

PostgreSQL 异步驱动 `aiopg` 的文档中有一个示例，演示了如何使用 `async for` 迭代一个数据库游标的各行。

```
async def go():
    pool = await aiopg.create_pool(dsn)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 1")
            ret = []
            async for row in cur:
                ret.append(row)
            assert ret == [(1,)]
```

这个示例中的查询只返回一行，而现实中，一个 `SELECT` 查询有可能返回上千行。对于大型响应，游标不一次性加载所有行。因此，

`async for row in cur:` 一定不能阻塞事件循环，妨碍游标等待后续行。`aiopg` 把游标实现为异步迭代器，每次调用 `__anext__` 时可以把控制权交还事件循环，当 PostgreSQL 发送更多行时再重获控制权。

21.10.1 异步生成器函数

若想实现异步迭代器，可以编写一个类，实现 `__anext__` 和 `__aiter__`。不过，还有更简单的方法：以 `async def` 声明一个函数，在主体中使用 `yield`。这与利用经典迭代器模式的生成器函数是一样的。

下面分析一个简单的示例，这个示例用到了 `async for`，还实现了一个异步生成器。示例 21-1 中的 `blogdom.py` 脚本用于探测域名。现在，假设我们发现那里定义的 `probe` 协程还有其他用途，于是决定把它放在新模块 `domainlib.py` 中。在这个模块中，我们还将新定义一个异步生成器 `multi_probe`，接受一组域名，产出相应的探测结果。

`domainlib.py` 的实现稍后再讲，先看如何在 Python 新增的异步控制台中使用。

01. 在 Python 异步控制台中实验

从 Python 3.8 开始，使用命令行选项 `-m asyncio` 运行解释器可以得到一个“异步 REPL”，这个 Python 控制台导入 `asyncio`，提供一个运行中的事件循环，支持在顶层提示符下使用 `await`、`async for` 和 `async with`——这些在原生协程外部使用原本会导致句法错误。¹³

在本书代码的本地副本中，进入 `21-async/domains/asyncio/` 目录，运行以下命令。

```
$ python -m asyncio
```

控制台启动，你会看到类似下面的输出。

```
asyncio REPL 3.9.1 (v3.9.1:1e5d33e9b9, Dec 7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more
information.
>>> import asyncio
>>>
```

头部信息指出，可以用 `await` 代替 `asyncio.run()`，驱动协程和其他可异步调用对象。另外，我没有输入 `import asyncio`。`asyncio` 模块是自动导入的，输出那一行是为了向用户明确表明该模块已导入。

现在导入 `domainlib.py`，使用其中的两个协程：`probe` 和 `multi_probe`（见示例 21-16）。

示例 21-16 运行 `python3 -m asyncio` 后试验 `domainlib.py`

```
>>> await asyncio.sleep(3, 'Rise and shine!') ❶
'Rise and shine!'
>>> from domainlib import *
>>> await probe('python.org') ❷
Result(domain='python.org', found=True) ❸
>>> names = 'python.org rust-lang.org golang.org no-
lang.invalid'.split() ❹
>>> async for result in multi_probe(names): ❺
...     print(*result, sep='\t')
...
golang.org      True ❻
no-lang.invalid False
python.org      True
rust-lang.org   True
>>>
```

❶ 简单试用 `await`，看看异步控制台的效果。提示：
`asyncio.sleep()` 接受一个可选的参数，设置等待的秒数。

- ❷ 驱动 probe 协程。
- ❸ domainlib 模块中的 probe 返回一个具名元组 Result。
- ❹ 构建一个域名列表。invalid 是保留顶级域名，用于测试。针对这种域名的 DNS 查询，DNS 服务器始终返回 NXDOMAIN 响应，表示“域名不存在”。¹⁴
- ❺ 使用 async for 迭代异步生成器 multi_probe，显示结果。
- ❻ 注意，结果的显示顺序与把域名传给 multi_probe 的顺序不同。返回一个 DNS 响应就显示一个域名。

示例 21-16 表明，multi_probe 是异步生成器，因为它能用 async for 迭代。接下来，再多做一些实验，如示例 21-17 所示。

示例 21-17 更多实验，接续示例 21-16

```
>>> probe('python.org') ❶
<coroutine object probe at 0x10e313740>
>>> multi_probe(names) ❷
<async_generator object multi_probe at 0x10e246b80>
>>> for r in multi_probe(names): ❸
...     print(r)
...
Traceback (most recent call last):
...
TypeError: 'async_generator' object is not iterable
```

- ❶ 调用原生协程得到一个协程对象。
- ❷ 调用异步生成器得到一个 async_generator 对象。
- ❸ 不能使用常规的 for 循环迭代异步生成器，因为异步生成器实现的是 __aiter__，而不是 __iter__。

异步生成器由 `async for` 驱动，可能导致阻塞（如示例 21-16 所示）。另外，异步推导式也使用 `async for`，详见下文。

02. 实现一个异步生成器

下面分析异步生成器 `multi_probe` 所在的 `domainlib.py` 模块，代码如示例 21-18 所示。

示例 21-18 `domainlib.py`：探测域名的函数

```
import asyncio
import socket
from collections.abc import Iterable, AsyncIterator
from typing import NamedTuple, Optional

class Result(NamedTuple): ❶
    domain: str
    found: bool

OptionalLoop = Optional[asyncio.AbstractEventLoop] ❷

async def probe(domain: str, loop: OptionalLoop = None) ->
Result: ❸
    if loop is None:
        loop = asyncio.get_running_loop()
    try:
        await loop.getaddrinfo(domain, None)
    except socket.gaierror:
        return Result(domain, False)
    return Result(domain, True)

async def multi_probe(domains: Iterable[str]) ->
AsyncIterator[Result]: ❹
    loop = asyncio.get_running_loop()
    coros = [probe(domain, loop) for domain in domains] ❺
    for coro in asyncio.as_completed(coros): ❻
```

```
result = await coro ⑦  
yield result ⑧
```

- ❶ 使用 `NamedTuple`, `probe` 的结果更易于阅读和调试。
- ❷ 这个类型别名是为了防止下一行太长, 超出打印范围。
- ❸ `probe` 现在接受一个可选的 `loop` 参数, 免得在 `multi_probe` 驱动这个协程的过程中不断调用 `get_running_loop`。
- ❹ 异步生成器函数产生一个异步生成器对象, 可以注解为 `AsyncIterator[SomeType]`。
- ❺ 构建一个 `probe` 协程对象列表, 对应各个域名。
- ❻ 不能使用 `async for`, 因为 `asyncio.as_completed` 是传统生成器。
- ❼ 异步等待协程对象, 获取结果。
- ❽ 产出 `result`。有这一行的存在, `multi_probe` 才是异步生成器。



示例 21-18 中的 `for` 循环可以进一步精简:

```
for coro in asyncio.as_completed(coros):  
    yield await coro
```

Python 把主体解析为 `yield (await coro)`, 能达到预期效果。

我觉得在本书第一个异步生成器示例中使用简写形式可能会让人不解其意, 所以分成了两行。

有了 domainlib.py, 就可以演示在 domaincheck.py 中如何使用异步生成器 multi_probe 了。domaincheck.py 脚本搜索 Python 短关键字与指定后缀构成的域名。

在 domainlib 模块的支持下, domaincheck.py 的代码简单直观, 如示例 21-19 所示。

示例 21-19 domaincheck.py: 使用 domainlib 模块探测域名

```
#!/usr/bin/env python3
import asyncio
import sys
from keyword import kwlist

from domainlib import multi_probe

async def main(tld: str) -> None:
    tld = tld.strip('.')
    names = (kw for kw in kwlist if len(kw) <= 4) ❶
    domains = (f'{name}.{tld}'.lower() for name in names) ❷
    print('FOUND\t\tNOT FOUND') ❸
    print('=====\t\t=====')
    async for domain, found in multi_probe(domains): ❹
        indent = ' ' if found else '\t\t' ❺
        print(f'{indent}{domain}')

if __name__ == '__main__':
    if len(sys.argv) == 2:
        asyncio.run(main(sys.argv[1])) ❻
    else:
        print('Please provide a TLD.', f'Example:
{sys.argv[0]} COM.BR')
```

- ❶ 生成长度不超过 4 的关键字。
- ❷ 使用指定的顶级域名生成一组域名。
- ❸ 格式化表格输出的表头。

- ④ 异步迭代 `multi_probe(domains)`。
- ⑤ 把 `indent` 设为零个或两个制表符，把结果放入相应的列中。
- ⑥ 运行 `main` 协程，传入命令行参数。

生成器还有一个与迭代无关的用途：上下文管理器。对于异步上下文也是如此。

03. 异步生成器用作上下文管理器

自己编写异步上下文管理器并不常见，万一需要，可以考虑使用 Python 3.7 在 `contextlib` 模块中增加的 `@asynccontextmanager` 装饰器。这与 18.2.2 节讨论的 `@contextmanager` 装饰器非常相似。

Using Asyncio in Python (Caleb Hattingh 著) 一书中有一个有趣的示例，结合了 `@asynccontextmanager` 和 `loop.run_in_executor`。示例 21-20 是 Caleb 给出的代码，我只做了一处改动，并添加了标号。

示例 21-20 `@asynccontextmanager` 和 `loop.run_in_executor` 用法示例

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def web_page(url): ①
    loop = asyncio.get_running_loop() ②
    data = await loop.run_in_executor( ③
        None, download_webpage, url)
    yield data ④
    await loop.run_in_executor(None, update_stats, url) ⑤
```

```
async with web_page('google.com') as data: ❹
    process(data)
```

- ❶ 被装饰的函数必须是异步生成器。
- ❷ 稍微改动 Caleb 的代码，用轻量级的 `get_running_loop` 换掉 `get_event_loop`。
- ❸ 假设 `download_webpage` 是一个阻塞型函数，使用 `requests` 库实现。在单独的线程中运行，防止阻塞事件循环。
- ❹ 这个 `yield` 表达式前面的所有行将变成装饰器构建的异步上下文管理器的 `__aenter__` 协程方法。`data` 的值将绑定下方 `async with` 语句中 `as` 子句的 `data` 变量。
- ❺ `yield` 表达式后面的行将变成 `__aexit__` 协程方法。这里，我们把另一个阻塞调用委托给线程执行器。
- ❻ 使用 `async with` 结构调用 `web_page`。

这与顺序执行代码的 `@contextmanager` 装饰器非常相似。详细说明见 18.2.2 节，包括如何在 `yield` 那一行处理错误。`contextlib` 文档中也有一个 `@asynccontextmanager` 示例。

最后，对比一下异步生成器函数与原生协程。

04. 异步生成器与原生协程

异步生成器函数与原生协程之间的主要同异点如下。

- 都使用 `async def` 声明。

- 异步生成器的主体中肯定有一个 `yield` 表达式——这才是生成器。原生协程绝对不含 `yield`。
- 原生协程可能返回 `None` 之外的值。异步生成器只能使用空 `return` 语句。
- 原生协程是可异步调用对象，可由 `await` 表达式驱动，也可以传给 `asyncio` 库中众多接受可异步调用对象的函数，例如 `create_task`。异步生成器不是可异步调用对象，而是异步可迭代对象，由 `async for` 或异步推导式驱动。

下面讨论异步推导式。

¹³它就像 Node.js 控制台一样，方便实验。感谢 Yury Selivanov 为 Python 异步编程所做的另一项卓越贡献。

¹⁴见“RFC 6761—Special-Use Domain Names”。

21.10.2 异步生成器表达式和异步推导式

“PEP 530—Asynchronous Comprehensions”为推导式和生成器表达式引入 `async for` 和 `await` 句法，从 Python 3.6 开始可以使用。

PEP 530 定义的结构只有异步生成器表达式可以出现在 `async def` 主体外部。

01. 定义和使用一个异步生成器表达式

对于示例 21-18 中的异步生成器 `multi_probe`，我们可以再编写一个异步生成器，只返回找到的域名。下面在以 `-m asyncio` 选项启动的异步控制台中演示。

```
>>> from domainlib import multi_probe
>>> names = 'python.org rust-lang.org golang.org no-
lang.invalid'.split()
>>> gen_found = (name async for name, found in
multi_probe(names) if found) ❶
>>> gen_found
```

```
<async_generator object <genexpr> at 0x10a8f9700> ❷  
>>> async for name in gen_found: ❸  
...     print(name)  
...  
golang.org  
python.org  
rust-lang.org
```

❶ 使用 `async for`，表明这是一个异步生成器表达式。可在 Python 模块的任意位置定义。

❷ 异步生成器表达式构建一个 `async_generator` 对象，与异步生成器函数（例如 `multi_probe`）返回的对象是一种类型。

❸ 异步生成器对象由 `async for` 语句驱动，而该语句只能出现在 `async def` 主体内，或者在异步控制台中使用（像本例这样）。

综上，异步生成器表达式可在程序的任何位置定义，但是只能在原生协程或异步生成器函数内使用。

PEP 530 引入的其他结构只能在原生协程或异步生成器函数内定义和使用。

02. 异步推导式

PEP 530 的作者 Yury Selivanov 通过 3 段简短的代码证明了异步推导式的必要性，下面直接摘录。

不得不说，以下代码片段：

```
result = []  
async for i in aiter():  
    if i % 2:  
        result.append(i)
```


如果能写成下面这样多好：

```
result = [i async for i in aiter() if i % 2]
```

另外，对于原生协程 `fun`，我们可以这样编写代码：

```
result = [await fun() for fun in funcs]
```



在列表推导式中使用 `await`，作用类似于 `asyncio.gather`。不过，`gather` 接受一个可选的 `return_exceptions` 参数，可以进一步处理异常。Caleb Hattingh 建议始终设置 `return_exceptions=True`（默认值为 `False`）。详见 `asyncio.gather` 文档。

回到异步控制台中。

```
>>> names = 'python.org rust-lang.org golang.org no-  
lang.invalid'.split()  
>>> names = sorted(names)  
>>> coros = [probe(name) for name in names]  
>>> await asyncio.gather(*coros)  
[Result(domain='golang.org', found=True),  
Result(domain='no-lang.invalid', found=False),  
Result(domain='python.org', found=True),  
Result(domain='rust-lang.org', found=True)]  
>>> [await probe(name) for name in names]  
[Result(domain='golang.org', found=True),  
Result(domain='no-lang.invalid', found=False),  
Result(domain='python.org', found=True),  
Result(domain='rust-lang.org', found=True)]  
>>>
```

注意，我把域名存储在列表中是想表明结果的顺序与提交的顺序一致——两种方式都是如此。

PEP 530 允许在列表推导式中使用 `async for` 和 `await`，在词典和集合推导式中也可以。例如，下面在异步控制台中使用词典推导式存储 `multi_probe` 的结果。

```
>>> {name: found async for name, found in multi_probe(names)}  
{'golang.org': True, 'python.org': True, 'no-lang.invalid':  
False,  
'rust-lang.org': True}
```

在 `for` 或 `async for` 子句前面，以及 `if` 子句后面，可以使用 `await` 关键字。下面在异步控制台中使用集合推导式收集找到的域名。

```
>>> {name for name in names if (await probe(name)).found}  
{'rust-lang.org', 'python.org', 'golang.org'}
```

由于 `__getattr__` 运算符 `.`（点号）的优先级较高，因此要在 `await` 表达式两侧再加一层括号。

再次强调，这些推导式只能出现在 `async def` 主体内，或者在施了魔法的异步控制台中使用。

接下来会讨论 `async` 语句、`async` 表达式及其创建的对象具有的一个非常重要的性质。这些结构通常使用 `asyncio` 库处理，而事实上它们不限于使用特定的库。

21.11 `asyncio` 之外的异步世界： Curio

Python 语言的 `async/await` 结构不限于特定的事件循环或库。¹⁵ 特殊方法提供的 API 具有扩展性，任何有足够动机的人都可以自己编写异步运行时环境和框架，驱动原生协程、异步生成器等。

¹⁵相比之下，JavaScript 中的 `async/await` 结构仅限于内置事件循环和运行时环境，即浏览器、Node.js 或 Deno。

David Beazley 开发的 Curio 项目就是这样做的。他想重新审视这些语言新功能，因此从零开始构建了一个框架。我们知道，`asyncio` 在

Python 3.4 中发布，使用 `yield from`，而不是 `await`，因此 `asyncio` 的 API 不能使用异步上下文管理器、异步迭代器，以及其他依托 `async/await` 关键字的结构。因此，与 `asyncio` 相比，Curio 的 API 更简洁，实现也更简单。

示例 21-21 使用 Curio 重写 `blogdom.py` 脚本（见示例 21-1）。

示例 21-21 `blogdom.py`：使用 Curio 重写示例 21-1

```
#!/usr/bin/env python3
from curio import run, TaskGroup
import curio.socket as socket
from keyword import kwlist

MAX_KEYWORD_LEN = 4

async def probe(domain: str) -> tuple[str, bool]: ❶
    try:
        await socket.getaddrinfo(domain, None) ❷
    except socket.gaierror:
        return (domain, False)
    return (domain, True)

async def main() -> None:
    names = (kw for kw in kwlist if len(kw) <= MAX_KEYWORD_LEN)
    domains = (f'{name}.dev'.lower() for name in names)
    async with TaskGroup() as group: ❸
        for domain in domains:
            await group.spawn(probe, domain) ❹
    async for task in group: ❺
        domain, found = task.result
        mark = '+' if found else ' '
        print(f'{mark} {domain}')

if __name__ == '__main__':
    run(main()) ❻
```

❶ `probe` 无须获取事件循环，因为.....

❷`getaddrinfo` 是 `curio.socket` 的顶层函数，不像 `asyncio` 那样，是 `loop` 对象的方法。

③ `TaskGroup` 是 `Curio` 的核心概念，用于监控和控制多个协程，能确保协程全部执行并得到清理。

④ 协程使用 `TaskGroup.spawn` 启动，由特定的 `TaskGroup` 实例管理。协程包装在 `Task` 对象中。

⑤ 使用 `async for` 迭代 `TaskGroup`，在任务完成后产出 `Task` 实例。这对应于示例 21-1 中 `for ... as_completed(...)`：那几行。

⑥ `Curio` 开创了这种在 `Python` 中启动异步程序的明智方式。

展开最后一点：看一下本书第 1 版中的 `asyncio` 代码示例，你会看到以下几行重复出现。

```
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

`Curio` 中的 `TaskGroup` 是异步上下文管理器，取代了 `asyncio` 的多个特有 API 和编程模式。前面的示例就指出，现在我们可以迭代 `TaskGroup`，无须调用 `asyncio.as_completed(...)` 函数。另外，以下摘自“Task Groups”文档的代码片段从任务组中收集全部任务的结果，不使用特殊的 `gather` 函数。

```
async with TaskGroup(wait=all) as g:
    await g.spawn(corol)
    await g.spawn(coroz)
    await g.spawn(coroz3)
print('Results:', g.results)
```

任务组支持**结构化并发**，这种并发编程形式限制异步任务组中的所有活动只有一个出入口。这类似于结构化编程，避开了 `GOTO` 命令，引入了块语句，限制循环和子例程的出入口。用作异步上下文管理器的 `TaskGroup`，确保在退出所在的块时，内部派生的所有任务都已完成或被撤销，异常也已抛出。



在即将发布的 Python 版本中，`asyncio` 可能会采用结构化并发。获准在 Python 3.11 中实现的“PEP 654–Exception Groups and `except*`”就是明显迹象。“Motivation”一节提到了 Trio 的“nurseries”（对任务组的称呼）：“参照 Trio nurseries，为 `asyncio` 实现更好的任务派生 API，这是本 PEP 的主要动机。”

Curio 的另一个重要特色是对在同一份基准代码中使用协程和线程编程提供了更好的支持——这对大部分重要的异步程序来说是必不可少的。以 `await spawn_thread(func, ...)` 启动进程，返回一个 `AsyncThread` 对象，接口与 `Task` 类似。借助特殊的 `AWAIT(coro)` 函数（全大写，因为现在 `await` 是关键字），线程可以调用协程。

Curio 还提供了一个 `UniversalQueue`，可用于协调线程、Curio 协程和 `asyncio` 协程之间的工作。没错，Curio 和 `asyncio` 可以在同一个进程内的不同线程中运行，通过 `UniversalQueue` 和 `UniversalEvent` 通信。这些“通用”类的 API 在协程内部和外部是相同的，但在协程中，需要在调用前面加上 `await`。

在我写下这些文字的 2021 年 10 月，`HTTPX` 是第一个兼容 Curio 的 HTTP 客户端库，但我尚未听说是否有任何异步数据库代码库支持 Curio。Curio 仓库中有大量网络编程示例，有一个使用 `WebSocket`，还有一个实现“RFC 8305—Happy Eyeballs”并发算法，连接 IPv6 端点，可在需要时快速回退到 IPv4。

Curio 的设计产生了很大影响。Nathaniel J. Smith 创建的 Trio 框架就很大程度上受到了 Curio 的启发。Curio 可能还促使 Python 贡献者对 `asyncio` API 的可用性做出了改进。例如，在早期发布的 `asyncio` 中，用户需要频繁获取和传递 `loop` 对象，因为一些重要函数要么是 `loop` 对象的方法，要么需要 `loop` 参数。在 Python 的最近几个版本中，不像以往那样经常需要访问循环了，而且多个接受可选 `loop` 参数的函数也开始弃用那个参数了。

下一个话题是异步类型的类型注解。

21.12 异步对象的类型提示

原生协程返回的类型是原生协程函数主体内出现在 `return` 语句中的对象类型，指明使用 `await` 处理协程时得到的是什么。¹⁶

¹⁶这与经典协程的注解不同，详见 17.13.3 节。

本章很多示例为原生协程添加了注解，例如示例 21-21 中的 `probe`。

```
async def probe(domain: str) -> tuple[str, bool]:
    try:
        await socket.getaddrinfo(domain, None)
    except socket.gaierror:
        return (domain, False)
    return (domain, True)
```

注解接受一个协程对象的参数，使用以下泛型。

```
class typing.Coroutine(Awaitable[V_co], Generic[T_co, T_contra,
V_co]):
    ...
```

那个类型，以及下述几个类型，在 Python 3.5 和 Python 3.6 中引入，用于注解异步对象。

```
class typing.AsyncContextManager(Generic[T_co]):
    ...
class typing.AsyncIterable(Generic[T_co]):
    ...
class typing.AsyncIterator(AsyncIterable[T_co]):
    ...
class typing.AsyncGenerator(AsyncIterator[T_co], Generic[T_co,
T_contra]):
    ...
class typing.Awaitable(Generic[T_co]):
    ...
```

对于 Python 3.9 及以上版本，应使用 `collections.abc` 中的对应类型。

关于这些泛型，我想强调以下 3 点。

首先，第一个类型参数，即异步对象产出的项的类型，均是协变的。正如 15.7.4 节的“型变经验法则”第一条所说：

如果一个形式类型参数定义的是从对象中获取的数据类型，该形式类型参数可能是协变的。

其次，`AsyncGenerator` 和 `Coroutine` 的第二个到最后一个参数是逆变的。这些是底层 `.send()` 方法的参数类型，事件循环调用该方法驱动异步生成器和协程。因此，属于“输入”类型。所以，根据“型变经验法则”第二条，可以是逆变的。

如果一个形式类型参数定义的是对象初始化之后向对象中输入的数据类型，该形式类型参数可能是逆变的。

最后，与 `typing.Generator` 不同（见 17.13.3 节），`AsyncGenerator` 没有返回值类型。17.13 节讲过，通过抛出 `StopIteration(value)` 返回一个值是一种特殊处理方式，为的是把生成器用作协程，以及支持 `yield from`。异步对象之间没有这种重叠：`AsyncGenerator` 对象不返回值，与原生协程对象（使用 `typing.Coroutine` 注解）完全不沾边。

本章最后简单讨论一下异步编程的优缺点。

21.13 异步原理与陷阱

本章即将结束，我们抛开所用的语言和库，讨论一下异步编程相关的总体思想。

我们先说明异步编程吸引人的首要因素，然后指出一个常见误区，以及如何绕开陷阱。

21.13.1 阻塞型调用导致漫长等待

Ryan Dahl 在介绍 Node.js 项目的理念时说：“我们对待 I/O 的方式大错特错。”¹⁷ 他把文件或网络 I/O 相关的函数定义为**阻塞型函数**，认为我们不能像非阻塞型函数那样对待阻塞型函数。为了说明原因，他给出了表 21-1 中第二列内的数值。

¹⁷“Introduction to Node.js”视频，4:55 处。

表 21-1：使用现代计算机从不同设备中读取数据的延迟；第三列是以一定比例换算的时间，方便人类理解

设备	CPU 循环	换算成“人类”时间
L1 缓存	3	3 秒
L2 缓存	14	14 秒
RAM	250	250 秒
磁盘	41 000 000	1.3 年
网络	240 000 000	7.6 年

为了理解表 21-1，要知道，现代 CPU 的时钟以 GHz 为单位，每秒可运行数十亿次循环。假设有一个 CPU 每秒正好运行 10 亿次循环。在 1 秒内，这个 CPU 可以读取 3.33 亿次 L1 缓存，或者读取 4 次网

络。表 21-1 中的第三列把第二列乘以一个常量因子，进行换算。因此，在另一个宇宙中，如果读取 L1 缓存耗时 3 秒，那么读取网络耗时 7.6 年。

表 21-1 表明，规范的异步编程方法可以提升服务器的性能。而我们面临的挑战是如何实现规范方法。第一步是认清一个事实，即“I/O 密集型系统”只是一个幻觉。

21.13.2 I/O 密集型系统的误区

人们经常说，异步编程适合“I/O 密集型系统”。而我从惨痛的教训中认识到，其实根本没有所谓的“I/O 密集型系统”。你遇到的可能是 I/O 密集型函数。系统中的函数，大多数算是 I/O 密集型的：与处理数据相比，用在等待 I/O 上的时间更多。在等待的过程中，函数把控制权交给事件循环，驱动其他挂起的任务。任何非平凡系统都有一些部分是 CPU 密集型的，这是不可避免的。即使普通系统也承受一定压力。在本章的“杂谈”中，我会讲述一个故事：两个异步程序受 CPU 密集型函数牵连，拖慢事件循环，严重影响性能。

既然任何非平凡系统都有 CPU 密集型函数，那么正确处理它们就是异步编程成功的关键。

21.13.3 绕开 CPU 密集型陷阱

大规模使用 Python，应该备有自动化测试，专门检测性能衰退，尽早发现性能问题。这对异步代码至关重要，而且因为有 GIL，这对 Python 多线程代码也不可或缺。如果等速度慢到影响开发团队的工作，那就太晚了，可能需要做出伤筋动骨的改动。

发现 CPU 占用出现瓶颈后，可以采取以下措施。

- 把任务委托给 Python 进程池。
- 把任务委托给外部任务队列。

- 使用 Cython、C、Rust，或者可编译成机器码、能与 Python/C API 交互的其他语言（最好能释放 GIL）重写相关代码。
- 如果判断自己可以承受性能损失，那可以什么都不做，但要把决定记录下来，方便以后重新审视。

项目启动伊始就应该选定并集成外部任务队列，需要时拿来即用，不给团队成员添麻烦。

最后一项措施是什么也不做，但这容易留下技术债务。

并发编程是一个迷人的话题，我也想多写一些内容。但这不是本书的关注重点，而且本章篇幅已经够长了。就此打住吧！

21.14 本章小结

常规异步编程方案有一种“不成功便成仁”式的傲骨侠风，这就是问题所在。你孤注一掷重写代码，要么彻底避免阻塞，要么纯属浪费时间。

——Alvaro Videla 和 Jason J. W. Williams
《RabbitMQ 实战：高效部署分布式消息队列》

本章开篇引用这段话有两个原因。总体来看，是为了提醒大家不要阻塞事件循环，速度慢的任务应委托其他处理单元——从简单的线程到分布式任务队列，不一而足。具体而言，也是对大家的警告：一旦编写第一个 `async def`，毫无疑问，程序中将有越来越多的 `async def`、`await`、`async with` 和 `async for`，再想使用非异步库，难度可想而知。

第 19 章简单实现了几个旋转指针示例，现在我们关注的重点是使用原生协程做异步编程。本章首先分析了 DNS 探测示例 `blogdom.py`，随后介绍了**可异步调用对象**概念。在阅读 `flags_asyncio.py` 源码的过程中，我们第一次见到**异步上下文管理器**。

国旗下载程序的高级版本用到了两个强大的函数：

`asyncio.as_completed` 生成器和 `loop.run_in_executor` 协程。我们还学习了信号量概念，了解到通过它可以限制并发下载数——行为端正的 HTTP 客户端理应如此。

我们通过 `mojifinder` 示例讲解了服务器端异步编程。我们实现了一个 FastAPI Web 服务器和 `tcp_mojifinder.py`——后者只使用 `asyncio` 和 TCP 协议。

接下来的话题是异步迭代和异步可迭代对象，介绍了 `async for`、Python 的异步控制台、异步生成器、异步生成器表达式和异步推导式。

本章最后一个示例是使用 Curio 框架重写的 `blogdom.py`，以此指出 Python 的异步功能不限于只能使用 `asyncio` 包。Curio 框架还运用了**结构化并发**概念，对业界可能有一定影响，写出的并发代码更加清晰易懂。

最后，21.13 节讨论了异步编程吸引人的首要因素、对“I/O 密集型系统”的误解，以及如何处理程序中不可避免的 CPU 密集型部分。

21.15 延伸阅读

David Beazley 在 PyOhio 2016 上的主题演讲“Fear and Awaiting in Async”十分精彩，通过现场编程介绍了 Yury Selivanov 为 Python 3.5 贡献的 `async/await` 关键字的潜力。在此期间，Beazley 抱怨 `await` 不能在列表推导式中使用。Selivanov 在“PEP 530—Asynchronous Comprehensions”中做出修正，同一年发布的 Python 3.6 已经实现。除此之外，Beazley 演讲的内容现在看来也不过时，他没有借助任何框架，只用一个简单的 `run` 函数通过 `.send(None)` 驱动协程，就揭示了本章讲到的异步对象的工作原理。Beazley 在演讲最后才提到 Curio。这是他在那一年开始的实验，他想抛开回调和 `future` 对象，试试只用协程做异步编程能走多远。结果表明，能走得

很远——Curio 一直有新版发布，还启发 Nathaniel J. Smith 创造了 Trio。Curio 文档中有更多 Beazley 对这个话题的演讲。

除了创造 Trio 之外，Nathaniel J. Smith 还有两篇深度文章，强烈推荐阅读。一篇是“Some thoughts on asynchronous API design in a post-async/await world”，对比 Curio 和 `asyncio` 的设计；另一篇是“Notes on structured concurrency, or: Go statement considered harmful”，探讨结构化并发。Smith 还在 Stack Overflow 上对“What is the core difference between asyncio and trio?”问题做出了详尽解答。

如果想深入学习 `asyncio` 包，我在本章开头已经提到了最好的文字资料：经 Yury Selivanov 于 2018 年修订后的官方文档，以及 *Using Asyncio in Python*（Caleb Hattingh 著）一书。务必阅读官方文档中的“Developing with asyncio”一文，这里记载了 `asyncio` 调试模式，还讨论了常见误区及避开方法。

Miguel Grinberg 在 PyCon 2017 上的演讲“Asynchronous Python for the Complete Beginner”，介绍了异步编程通用知识，也涉及 `asyncio`。此次演讲只有 30 分钟，通俗易懂。Michael Kennedy 的演讲“Demystifying Python's Async and Await Keywords”，也是很好的入门资料。我从中学到了很多，比如说我了解到 `unsync` 库提供了一个装饰器，可把协程、I/O 密集型函数和 CPU 密集型函数委托给 `asyncio`、`threading` 或 `multiprocessing` 执行。

Lynn Root 是 Spotify 高级工程师，也是 PyLadies 全球领袖，她根据自己使用 Python 的经验，在 EuroPython 2019 上做了一场精彩的演讲，题为“Advanced asyncio: Solving Real- world Production Problems”。

2020 年，Łukasz Langa 录制了讲解 `asyncio` 的系列视频，第一个是“Learn Python's AsyncIO #1—The Async Ecosystem”。Langa 还为 PyCon 2020 录制了“AsyncIO + Music”视频，立意新颖。在这个视频中，他不仅说明了如何在具体的面向事件领域中运用 `asyncio`，而且还讲解了基础知识。

另一个由面向事件编程主导的领域是嵌入式系统。所以，Damien George 为针对微控制器的 MicroPython 添加了 `async/await` 支持。在 PyCon Australia 2018 上，Matt Trentini 演示了 `uasyncio`。这是 MicroPython 标准库中的一个包，是 `asyncio` 的子集。

Tom Christie 写的“Python async frameworks—Beyond developer tribalism”一文，对 Python 异步编程做了更高层次的思考。

最后，推荐阅读 Bob Nystrom 写的“What Color Is Your Function?”一文。这篇文章讨论了 JavaScript、Python、C# 等语言中常规函数和异步函数（即协程）不相容的执行模型。内容剧透：Nystrom 得出结论，Go 语言的做法是正确的，即所有函数的颜色相同。我喜欢 Go 语言的这种做法。但是，我也觉得 Nathaniel J. Smith 在“Go statement considered harmful”一文中所说的有道理。世间不完美，并发编程始终是难题。

杂谈

一个速度慢的函数差点儿毁掉 uvloop 的基准测试

2016 年，Yury Selivanov 发布了 uvloop。这是“一个事件循环，可以直接取代 `asyncio` 内置的事件循环，速度更快”。当年宣布这个库问世的博客文章提供了基准测试，表现让人赞叹。他写道：“（uvloop）至少比 Node.js、gevent，以及其他 Python 异步框架快 2 倍。基于 uvloop 的 `asyncio`，性能接近 Go 程序。”

然而，那篇文章也指出，若想与 Go 语言的性能相当，需要满足两个条件。

01. 配置 Go 语言使用单个线程。这样 Go 语言运行时的行为才与 `asyncio` 类似：通过同一个线程中由一个事件循环驱动的多协程实现并发。¹⁸
02. 使用 Python 3.5 编写代码，而且除了 uvloop 自身，还要使用 `httptools`。

Selivanov 使用 aiohttp（早期使用 `asyncio` 实现的全功能 HTTP 库之一）对 uvloop 做了基准测试，之后编写了 httptools。他解释了这么做的原因：

然而，我们发现 aiohttp 的性能瓶颈是它的 HTTP 解析器，速度太慢了，即使底层 I/O 库再快又有什么用呢？为了得到更准确的测试结果，我们为 http-parser（Node.js HTTP 解析器库，使用 C 语言实现，最初为 NGINX 开发）创建了 Python 绑定。这个库名为 httptools，GitHub 和 PyPI 上都有。

Selivanov 所做的 HTTP 性能测试用到一个简单的回显服务器，使用不同的语言和库编写，由基准测试工具 wrk 发送大量请求。在多数开发者看来，一个简单的回显服务器应该是“I/O 密集型系统”，对吧？其实，HTTP 首部解析是 CPU 密集型操作，而在 Selivanov 做基准测试的 2016 年，以 Python 实现的 aiohttp 解析首部的速度很慢。使用 Python 编写的函数解析首部，事件循环受到阻塞。由此产生的影响太大，Selivanov 不得不动手编写了 httptools。倘若不优化 CPU 密集型代码，则事件循环对性能的提升将被抵消。

积羽沉舟

回显服务器简单了点儿，那我们来看一个复杂的 Python 系统——代码上万行，用到很多外部库，而且一直在演进。几年前，有人找我诊断一个系统的性能问题。那个系统就很复杂，使用 Python 2.7 编写，用的框架是 Twisted——一个可靠的库，基本上算是 `asyncio` 的前身。

那个系统使用 Python 构建外层 Web UI，集成使用其他语言编写的库和命令行工具提供的功能，但是没有考虑到并发执行。

那个项目雄心勃勃，已经持续开发超过一年，但是始终没有上线。¹⁹ 随着时间的推移，开发人员发现，整个系统的性能每况愈

下，而瓶颈何在却毫无头绪。

问题出在哪里呢？每增加一个功能，就有更多 CPU 密集型代码拖慢 Twisted 的事件循环。把 Python 作为胶水语言使用，意味着需要大量解析数据和转换格式。瓶颈不是出现在一个点上，问题分散在几个月的开发过程中不断添加的无数小型函数上。要解决问题，就需要重新思考系统架构，并重写大量代码。也许需要任务队列，可能还要使用微服务，或者以更适合并发处理 CPU 密集型操作的语言编写库。然而，项目方不准备继续投资，那个项目不久后被取消了。

我把这个故事告诉了 Twisted 项目的创始人 Glyph Lefkowitz，他说，准备开发一个异步编程项目时，他的首要任务之一是决定使用哪些工具来完成 CPU 密集型任务。受这次与 Glyph 的谈话启发，我撰写了 21.13.3 节。

¹⁸直到 Go 1.5 发布，默认设置才是使用单个线程。在此之前的几年，由于可实现高度并发的网络系统，Go 语言名声大噪。这再一次证明，并发不一定要靠多线程或多个 CPU 核。

¹⁹抛开技术选择，那个项目最大的问题应该是项目方没有采取 MVP 方法，即尽早交付一个最简可用产品（minimum viable product），然后稳步增加功能。

第五部分 元编程

- 第 22 章 动态属性和特性
- 第 23 章 属性描述符
- 第 24 章 类元编程

第 22 章 动态属性和特性

特性之所以重要，是因为它的存在使得开发者可以非常安全并且确定可行地将公共数据属性作为类的公共接口的一部分开放出来。

——Martelli、Ravenscroft 和 Holden
“特性的重要性”¹

¹出自 *Python in a Nutshell*, 3rd ed. (Alex Martelli、Anna Ravenscroft 和 Steve Holden 著)。

在 Python 中，数据属性和方法统称**属性** (attribute)。其实，方法是**可调用的属性**。**动态属性** (dynamic attribute) 的接口与数据属性一样 (`obj.attr`)，不过按需计算。这与 Bertrand Meyer 所说的**统一访问原则** (Uniform Access Principle) 相符。

不管服务是由存储还是计算实现的，一个模块提供的所有服务应统一表示法。²

²出自 *Object-Oriented Software Construction*, 2nd ed. (Bertrand Meyer 著)。

在 Python 中，实现动态属性的方式有好几种。本章涵盖最简单的方式：`@property` 装饰器和 `__getattr__` 特殊方法。

用户定义的类通过 `__getattr__` 方法可以实现一种我称之为**虚拟属性** (virtual attribute) 的动态属性。虚拟属性在类的源码中没有显式声明，也不出现在实例属性 `__dict__` 中，但是我们随时都能访问，而且当用户读取不存在的属性（例如 `obj.no_such_attr`）时，还能即时计算。

创建动态属性和虚拟属性是一种元编程，框架的作者经常这么做。然而，在 Python 中，相关的基础技术十分简单，在日常数据转换任务中也能用到。下面我们以这种任务开启本章的话题。

22.1 本章新增内容

本章的改动大多是为了讨论 `@functools.cached_property` (Python 3.8 引入)，以及 `@property` 和 `@functools.cache` (Python 3.9 新增) 的综合运用。受此影响，22.3 节中 `Record` 类和 `Event` 类的代码有所变动。我还利用“PEP 412—Key-Sharing Dictionary”提出的优化措施重构了一下。

为了把重点放在相关功能上，同时保持示例简单易懂，我改动了一些不重要的代码：把旧的 `DbRecord` 类并入 `Record`，把 `shelve.Shelve` 替换为 `dict`，还删除了下载 OSCON 数据集的逻辑——相关示例现在从本书代码的一个本地文件内读取数据。

22.2 使用动态属性转换数据

接下来的几个示例将利用动态属性处理 O'Reilly 为 OSCON 2014 大会发布的 JSON 数据集。示例 22-1 是那个数据集中的 4 个记录。³

³由于一些原因，OSCON (O'Reilly Open Source Conference) 近几年没有举办。这几个示例使用的那个 744 KB 的 JSON 文件在 2021 年 1 月 10 日已不可在线访问。本书代码中有一份副本，在 `osconfeed.json` 文件中。

示例 22-1 `osconfeed.json` 文件中的记录示例（节略了部分字段的内容）

```
{ "Schedule":
  { "conferences": [{"serial": 115 }],
    "events": [
      { "serial": 34505,
        "name": "Why Schools Don't Use Open Source to Teach
```

```

Programming",
    "event_type": "40-minute conference session",
    "time_start": "2014-07-23 11:30:00",
    "time_stop": "2014-07-23 12:10:00",
    "venue_serial": 1462,
    "description": "Aside from the fact that high school
programming...",
    "website_url":
"http://oscon.com/oscon2014/public/schedule/detail/34505",
    "speakers": [157509],
    "categories": ["Education"] }
],
"speakers": [
{ "serial": 157509,
  "name": "Robert Lefkowitz",
  "photo": null,
  "url": "http://sharewave.com/",
  "position": "CTO",
  "affiliation": "Sharewave",
  "twitter": "sharewaveteam",
  "bio": "Robert 'r0ml' Lefkowitz is the CTO at Sharewave, a
startup..." }
],
"venues": [
{ "serial": 1462,
  "name": "F151",
  "category": "Conference Venues" }
]
}
}

```

那个 JSON 文件中有 895 条记录，示例 22-1 只列出了 4 条。整个数据集是一个 JSON 对象，里面有一个键，名为 "Schedule"，这个键对应的值又是一个映像，有 4 个

键："conferences"、"events"、"speakers" 和 "venues"。这 4 个键中每一个键对应的值都是一个记录列表。在完整的数据集中，列表 "events"、"speakers" 和 "venues" 中有几十上百条记录，不过 "conferences" 列表中只有示例 22-1 中那一条记录。每条记录都有一个 "serial" 字段，这是列表中各条记录的唯一标识符。

为了了解这个数据集的结构，我在 Python 控制台中进行了探索，如示例 22-2 所示。

示例 22-2 在交互式控制台中探索 osconfeed.json

```
>>> import json
>>> with open('data/osconfeed.json') as fp:
...     feed = json.load(fp) ❶
>>> sorted(feed['Schedule'].keys()) ❷
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed['Schedule'].items()):
...     print(f'{len(value):3} {key}') ❸
...
   1 conferences
 484 events
 357 speakers
   53 venues
>>> feed['Schedule']['speakers'][-1]['name'] ❹
'Carina C. Zona'
>>> feed['Schedule']['speakers'][-1]['serial'] ❺
141590
>>> feed['Schedule']['events'][40]['name']
'There *Will* Be Bugs'
>>> feed['Schedule']['events'][40]['speakers'] ❻
[3471, 5199]
```

❶ feed 的值是一个字典，里面嵌套着字典和列表，存储着字符串和整数值。

❷ 列出 "Schedule" 键中的 4 个记录合集。

❸ 显示各个合集中的记录数量。

❹ 深入嵌套的字典和列表，获取最后一个演讲者的名字。

❺ 获取那位演讲者的编号。

❻ 每个事件都有一个 'speakers' 字段，该字段列出了零个或多个演讲者的编号。

22.2.1 使用动态属性访问 JSON 类数据

示例 22-2 十分简单，不过，`feed['Schedule']['events'][40]['name']` 这种句法很冗长。在 JavaScript 中，可以使用 `feed.Schedule.events[40].name` 获取那个值。在 Python 中，可以实现一个近似字典的类（网上有大量实现），⁴ 达到同样的效果。我自己实现了 `FrozenJSON` 类，该类比大多数实现简单，因为它只支持读取，即只能访问数据。不过，`FrozenJSON` 类能递归，自动处理嵌套的映射和列表。

⁴比如说 `AttrDict` 和 `addict`。

示例 22-3 演示了 `FrozenJSON` 类的用法，源码在示例 22-4 中。

示例 22-3 示例 22-4 定义的 `FrozenJSON` 类不仅能读取属性（例如 `name`），还能调用方法（例如 `.keys()` 和 `.items()`）

```
>>> import json
>>> raw_feed = json.load(open('data/osconfeed.json'))
>>> feed = FrozenJSON(raw_feed) ❶
>>> len(feed.Schedule.speakers) ❷
357
>>> feed.keys()
dict_keys(['Schedule'])
>>> sorted(feed.Schedule.keys()) ❸
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed.Schedule.items()): ❹
...     print(f'{len(value):3} {key}')
...
  1 conferences
484 events
357 speakers
  53 venues
>>> feed.Schedule.speakers[-1].name ❺
'Carina C. Zona'
>>> talk = feed.Schedule.events[40]
>>> type(talk) ❻
<class 'explore0.FrozenJSON'>
>>> talk.name
'There *Will* Be Bugs'
>>> talk.speakers ❼
[3471, 5199]
>>> talk.flavor ❽
```

```
Traceback (most recent call last):
...
KeyError: 'flavor'
```

- ❶ 传入嵌套的字典和列表组成的 `raw_feed`，创建一个 `FrozenJSON` 实例。
- ❷ `FrozenJSON` 实例能使用属性表示法遍历嵌套的字典，这里我们获取演讲者列表的长度。
- ❸ 也可以使用底层字典的方法（例如 `.keys()`）获取记录合集的名称。
- ❹ 使用 `items()` 方法获取各个记录合集及其内容，然后显示各个记录合集的长度。
- ❺ 列表（例如 `feed.Schedule.speakers`）仍是列表，但是，如果里面的项是映射，则转换成 `FrozenJSON` 对象。
- ❻ `events` 列表中的第 40 项是一个 JSON 对象，现在则变成了一个 `FrozenJSON` 实例。
- ❼ 事件记录中有一个 `speakers` 列表，其中列出了演讲者的编号。
- ❽ 读取不存在的属性抛出 `KeyError` 异常，而不是常规的 `AttributeError` 异常。

`FrozenJSON` 类的关键是 `__getattr__` 方法。12.6 节的 `Vector` 示例中用过这个方法，那时是为了通过字母获取 `Vector` 对象的分量，例如 `v.x`、`v.y`、`v.z` 等。有一点必须记住，仅当无法通过常规方式获取属性（例如，在实例、类或超类中找不到指定的属性）时，解释器才调用特殊方法 `__getattr__`。

示例 22-3 的最后一行揭露了我的代码中的一个小问题：尝试读取不存在的属性应该抛出 `AttributeError` 异常，而不是 `KeyError` 异

常。其实，一开始我对这个异常做了处理，但是 `__getattr__` 方法的代码量增加了一倍，而且偏离了我最想展示的重要逻辑。鉴于用户知道 `FrozenJSON` 建立在映射和列表之上，我想 `KeyError` 也不至于太让人困惑。

示例 22-4 `explore0.py`: 把 JSON 数据集转换成嵌套着 `FrozenJSON` 对象、列表和简单类型的 `FrozenJSON` 对象

```
from collections import abc

class FrozenJSON:
    """一个只读接口，该接口使用属性表示法访问JSON类对象"""

    def __init__(self, mapping):
        self.__data = dict(mapping) ❶

    def __getattr__(self, name): ❷
        try:
            return getattr(self.__data, name) ❸
        except AttributeError:
            return FrozenJSON.build(self.__data[name]) ❹

    def __dir__(self): ❺
        return self.__data.keys()

    @classmethod
    def build(cls, obj): ❻
        if isinstance(obj, abc.Mapping): ❼
            return cls(obj)
        elif isinstance(obj, abc.MutableSequence): ❽
            return [cls.build(item) for item in obj]
        else: ❾
            return obj
```

❶ 使用 `mapping` 参数构建一个字典。这么做是为了确保传入的映射或其他对象可以转换成字典。`__data` 前面有两条下划线，是私有属性。

❷ 仅当未指定名称 (`name`) 的属性时才调用 `__getattr__` 方法。

③ 如果 `name` 匹配 `__data` 字典的某个属性，就返回对应的属性。
`feed.keys()` 调用就是这样处理的：`keys` 方法是 `__data` 字典的一个属性。

④ 否则，从 `self.__data` 中获取 `name` 键对应的项，返回调用 `FrozenJSON.build()` 方法得到的结果。⁵

⁵这一行中的 `self.__data[name]` 表达式可能抛出 `KeyError` 异常。理想情况下，应该处理这个异常，并抛出 `AttributeError` 异常，因为这才是 `__getattr__` 方法应该抛出的异常种类。建议勤奋的读者实现错误处理代码，将此当作一个练习。

⑤ 实现为内置函数 `dir()` 提供支持的 `__dir__` 方法，进而支持在 Python 标准控制台，以及 IPython、Jupyter Notebook 等中自动补全。这个方法的代码很简单，将基于 `self.__data` 中的键实现递归自动补全，因为 `__getattr__` 方法能即时构建 `FrozenJSON` 实例——方便采用交互方式探索数据。

⑥ 这是一个备选构造方法，是 `@classmethod` 装饰器的常见用途。

⑦ 如果 `obj` 是一个映射，那么就构建一个 `FrozenJSON` 对象。这里利用了**大鹅类型**（详见 13.5 节）。

⑧ 如果是一个 `MutableSequence` 对象，则必然是列表，⁶ 因此，把 `obj` 中的每一项递归都传给 `.build()` 方法，构建一个列表。

⁶数据源是 JSON 格式，而在 JSON 中，只有字典和列表是容器类型。

⑨ 如果既不是字典也不是列表，那么原封不动返回项。

`FrozenJSON` 实例的私有实例属性 `__data` 存储在 `__FrozenJSON__data` 名下（详见 11.10 节）。尝试通过其他名称获取属性将触发 `__getattr__` 方法。这个方法会先查看 `self.__data` 字典有没有指定名称的属性（不是键），这样 `FrozenJSON` 实例便可以处理 `dict` 的方法，例如把 `items` 方法委

托给 `self.__data.items()`。如果 `self.__data` 没有指定名称的属性，那么 `__getattr__` 方法就会以那个名称为键，从 `self.__data` 中获取一项，传给 `FrozenJSON.build` 方法。这样便能深入 JSON 数据的嵌套结构，使用类方法 `build` 把每一层嵌套都转换成一个 `FrozenJSON` 实例。

注意，`FrozenJSON` 不转换或缓存原始数据集。在遍历数据的过程中，`__getattr__` 方法不断创建 `FrozenJSON` 实例。对这个体量的数据集来说，这么做没问题，而且这个脚本只用于访问或转换数据。

从随机源中生成或仿效动态属性名的脚本必须处理一个问题：原始数据中的键可能不适合作为属性名。22.2.2 节将解决这个问题。

22.2.2 处理无效属性名

`FrozenJSON` 类不能处理与 Python 关键字同名的属性名。例如，对于以下对象：

```
>>> student = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```

无法读取 `student.class`，因为在 Python 中 `class` 是保留关键字。

```
>>> student.class
File "<stdin>", line 1
    student.class
            ^
SyntaxError: invalid syntax
```

当然可以这样做：

```
>>> getattr(student, 'class')
1982
```

但是，`FrozenJSON` 类的目的是便于访问数据，因此更好的方法是检查传给 `FrozenJSON.__init__` 方法的映射中是否有键的名称为关

键字，如果有，就在键名后加上 `_`，通过下述方式读取。

```
>>> student.class_  
1982
```

为此，可以把示例 22-4 中只有一行代码的 `__init__` 方法改成示例 22-5 中的版本。

示例 22-5 `explore1.py`：在名称为 Python 关键字的属性后面加上 `_`

```
def __init__(self, mapping):  
    self.__data = {}  
    for key, value in mapping.items():  
        if keyword.iskeyword(key): ❶  
            key += '_'  
        self.__data[key] = value
```

❶ `keyword.iskeyword(...)` 正是我们所需的函数。为了使用它，必须导入 `keyword` 模块（这个代码片段中没有列出）。

如果 JSON 对象中的键不是有效的 Python 标识符，那么也会遇到类似的问题。

```
>>> x = FrozenJSON({'2be': 'or not'})  
>>> x.2be  
File "<stdin>", line 1  
    x.2be  
      ^  
SyntaxError: invalid syntax
```

这种有问题的键在 Python 3 中易于检测，`str` 类提供的 `s.isidentifier()` 方法能根据语言的语法判断 `s` 是否为有效的 Python 标识符。但是，把无效的标识符变成有效的属性名没那么容易。一种解决方案是实现 `__getitem__` 方法，使用类似 `x['2be']` 的表示法访问属性。为简单起见，我将忽略这个问题。

对动态属性的名称做了一些处理之后，下面来分析一下 FrozenJSON 类的另一个重要功能——类方法 build 的逻辑。__getattr__ 方法根据访问的属性是什么样的值，使用 FrozenJSON.build 构建不同类型的对象，把嵌套结构转换成 FrozenJSON 实例或 FrozenJSON 实例列表。

这样的逻辑除了在类方法中可以实现，还可以在特殊方法 __new__ 中实现，详见 22.2.3 节。

22.2.3 使用 __new__ 方法灵活创建对象

我们通常把 __init__ 称为构造方法，这是从其他语言借鉴过来的术语。在 Python 中，__init__ 的第一个参数是 self，可见在解释器调用 __init__ 时，对象已经存在。另外，__init__ 方法什么也不能返回。所以，__init__ 其实是初始化方法，不是构造方法。

调用类创建实例时，为了构建实例，Python 调用的特殊方法是 __new__。这是一个类方法，以特殊方式对待，因此不必使用 @classmethod 装饰器。Python 会把 __new__ 返回的实例传给 __init__ 的第一个参数 self。我们几乎不需要自己编写 __new__ 方法，因为从 object 类继承的实现已经可以满足大多数情况。

如有必要，__new__ 方法也可以返回其他类的实例。此时，解释器不调用 __init__ 方法。也就是说，Python 构建对象的过程可以使用以下伪代码概括。

```
# 构建对象的伪代码
def make(the_class, some_arg):
    new_object = the_class.__new__(some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

# 以下两个语句的作用基本等同
x = Foo('bar')
x = make(Foo, 'bar')
```

示例 22-6 是 FrozenJSON 类的另一个版本，把之前类方法 build 中的逻辑移到了 `__new__` 方法中。

示例 22-6 explore2.py: 使用 `__new__` 方法取代 build 方法，构建可能是也可能不是 FrozenJSON 实例的新对象

```
from collections import abc
import keyword

class FrozenJSON:
    """一个只读接口，该接口使用属性表示法访问JSON类对象
    """
    def __new__(cls, arg): ❶
        if isinstance(arg, abc.Mapping):
            return super().__new__(cls) ❷
        elif isinstance(arg, abc.MutableSequence): ❸
            return [cls(item) for item in arg]
        else:
            return arg

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if keyword.iskeyword(key):
                key += '_'
            self.__data[key] = value

    def __getattr__(self, name):
        try:
            return getattr(self.__data, name)
        except AttributeError:
            return FrozenJSON(self.__data[name]) ❹

    def __dir__(self):
        return self.__data.keys()
```

❶ `__new__` 是类方法，第一个参数是类本身，余下的参数与 `__init__` 方法一样，只不过没有 `self`。

❷ 默认行为是委托给超类的 `__new__` 方法。这里调用的是 `object` 基类的 `__new__` 方法，传入的唯一参数是 `FrozenJSON`。

③ `__new__` 方法中余下的代码与原先的 `build` 方法完全一样。

④ 之前这里调用的是 `FrozenJSON.build` 方法，现在只需调用 `FrozenJSON` 类，Python 将调用 `FrozenJSON.__new__` 来处理该类。

`__new__` 方法的第一个参数是类，因为创建的对象通常是那个类的实例。所以，在 `FrozenJSON.__new__` 方法中，`super().__new__(cls)` 表达式将调用 `object.__new__(FrozenJSON)`，`object` 类构建的实例其实是 `FrozenJSON` 实例。新实例的 `__class__` 属性将存储对 `FrozenJSON` 类的引用。不过，真正的构建操作由解释器调用 C 语言实现的 `object.__new__` 方法执行。

OSCON JSON 数据集的结构不太适合交互式探索。例如，索引为 40 的事件（名为 'There *Will* Be Bugs' 的那个）有两位演讲者，分别是 3471 和 5199，但想找到他们的名字不那么容易，因为提供的是编号，而 `Schedule.speakers` 列表没有使用编号建立索引。因此，如果想找出各个演讲者的名字，就必须迭代 `Schedule.speakers` 列表，直至找到编号对应的记录。我们的下一项任务是调整数据结构，为自动获取所链接的记录做好准备。

22.3 计算特性

我们第一次见到 `@property` 装饰器是在 11.7 节。在示例 11-7 中，我在 `Vector2d` 中用到了两个特性，目的只是把 `x` 和 `y` 变成只读属性。本节介绍可计算值的特性，顺带讨论如何缓存计算得到的值。

在 OSCON JSON 数据中，'events' 列表中的记录有一些整数编号，指向 'speakers' 列表和 'venues' 列表中的记录。例如，下面是大会中一次演讲的记录（描述信息有节略）。

```
{ "serial": 33950,
  "name": "There *Will* Be Bugs",
  "event_type": "40-minute conference session",
  "time_start": "2014-07-23 14:30:00",
  "time_stop": "2014-07-23 15:10:00",
  "venue_serial": 1449,
  "description": "If you're pushing the envelope of
programming...",
  "website_url":
"http://oscon.com/oscon2014/public/schedule/detail/33950",
  "speakers": [3471, 5199],
  "categories": ["Python"] }
```

我们将实现一个 `Event` 类，通过 `venue` 特性和 `speakers` 特性自动获取链接的数据，免去使用编号查找的麻烦。给定一个 `Event` 实例，示例 22-7 是我们想要实现的行为。

示例 22-7 读取 `venue` 和 `speakers` 返回 `Record` 对象

```
>>> event ❶
<Event 'There *Will* Be Bugs'>
>>> event.venue ❷
<Record serial=1449>
>>> event.venue.name ❸
'Portland 251'
>>> for spkr in event.speakers: ❹
...     print(f'{spkr.serial}: {spkr.name}')
...
3471: Anna Martelli Ravenscroft
5199: Alex Martelli
```

- ❶ 给定一个 `Event` 实例.....
- ❷ 读取 `event.venue` 返回一个 `Record` 对象，而不是编号。
- ❸ 现在轻易就能获取 `venue` 的名称。
- ❹ `event.speakers` 特性返回一系列 `Record` 实例。

一如往常，一步步构建。先从 `Record` 类和一个读取 JSON 数据、返回 `Record` 实例字典的函数开始。

22.3.1 第 1 步：数据驱动属性创建

指引第 1 步的 doctest 如示例 22-8 所示。

示例 22-8 测试驱动 schedule_v1.py (参见示例 22-9)

```
>>> records = load(JSON_PATH) ❶
>>> speaker = records['speaker.3471'] ❷
>>> speaker ❸
<Record serial=3471>
>>> speaker.name, speaker.twitter ❹
('Anna Martelli Ravenscroft', 'annaraven')
```

- ❶ load 函数加载一份字典形式的 JSON 数据。
- ❷ records 中的键是由记录类型和编号构成的字符串。
- ❸ speaker 是示例 22-9 中定义的 Record 类的实例。
- ❹ 原 JSON 数据中的字段可通过 Record 实例属性获取。

schedule_v1.py 的代码如示例 22-9 所示。

示例 22-9 schedule_v1.py：重新组织 OSCON 的日程数据

```
import json

JSON_PATH = 'data/osconfeed.json'

class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs) ❶

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>' ❷

def load(path=JSON_PATH):
    records = {} ❸
    with open(path) as fp:
```

```

    raw_data = json.load(fp) ❹
    for collection, raw_records in raw_data['Schedule'].items():
❺
        record_type = collection[:-1] ❻
        for raw_record in raw_records:
            key = f'{record_type}.{raw_record["serial"]}' ❼
            records[key] = Record(**raw_record) ❽
    return records

```

- ❶ 根据关键字参数构建带属性的实例经常这样简写（详细说明如下）。
- ❷ 使用 `serial` 字段自定义示例 22-8 所示那种 `Record` 的字符串表示形式。
- ❸ `load` 函数最终返回一个 `Record` 实例字典。
- ❹ 解析 JSON 数据，返回 Python 原生对象：列表、字典、字符串、数值等。
- ❺ 迭代 4 个顶级列表，即 `'conferences'`、`'events'`、`'speakers'` 和 `'venues'`。
- ❻ `record_type` 是列表名称去掉最后一个字符得到的结果，例如 `speakers` 变成 `speaker`。在 Python 3.9 及以上版本中，可以使用 `collection.removesuffix('s')`（参见“PEP 616—String methods to remove prefixes and suffixes”），明确表明意图。
- ❼ 构建 `'speaker.3471'` 格式的 `key`。
- ❽ 创建一个 `Record` 实例，存在 `records` 中的 `key` 名下。

`Record.__init__` 方法用到了一个古老的 Python 编程技巧。11.11 节讲过，一个对象的 `__dict__` 存储着对象的属性，除非类声明了 `__slots__`。因此，使用一个映射更新实例的 `__dict__` 可以快速为实例创建一大批属性。⁷

⁷顺便说一下，Bunch（大批）是 Alex Martelli 在 2001 年分享的技巧中使用的类名，该技巧名为“The simple but handy ‘collector of a bunch of named stuff’ class”。



在某些应用程序中，Record 类可能需要像 22.2.2 节所讲的那样处理不可作为属性名称的键。不过，如果分心处理该问题，就偏离了这个示例的主题，而且对目前读取的数据集来说，也没有这样的问题。

示例 22-9 定义的 Record 类非常简单，你或许会想，为什么我一开始不这样做，而是使用更为复杂的 FrozenJSON。原因有二。其一，FrozenJSON 要递归转换嵌套的映射和列表，而 Record 则不需要这样做，因为转换后的数据集没有嵌套映射的映射或列表，记录中只有字符串、整数、字符串列表和整数列表。其二，FrozenJSON 允许访问 `__data` 的字典属性（用以调用 `.keys()` 等方法），而现在不再需要这个功能了。



Python 标准库提供了一些与 Record 功能类似的类，可通过传给 `__init__` 的关键字参数构建属性，包括 `types.SimpleNamespace`、`argparse.Namespace` 和 `multiprocessing.managers.Namespace`。我选择自己编写更简单的 Record 类，是为了强调一个重要思想：在 `__init__` 中更新实例属性 `__dict__`。

重新组织日程数据集之后，可以增强 Record 类，自动获取 event 记录中引用的 venue 和 speaker。接下来的几个示例将使用特性实现这个想法。

22.3.2 第 2 步：通过特性获取链接的记录

下一版的目标是，给定一个 event 记录，读取 venue 特性得到一个 Record 对象。这与 Django ORM 访问 ForeignKey 字段的行为类似，得到的不是键，而是链接的模型对象。

从 venue 特性开始。示例 22-10 中的交互演示了我们想实现的效果。

示例 22-10 摘自 schedule_v2.py 的 doctest

```
>>> event = Record.fetch('event.33950') ❶
>>> event ❷
<Event 'There *Will* Be Bugs'>
>>> event.venue ❸
<Record serial=1449>
>>> event.venue.name ❹
'Portland 251'
>>> event.venue_serial ❺
1449
```

❶ Record.fetch 静态方法从数据集中获取一个 Record 或 Event 对象。

❷ 注意，event 是 Event 类的实例。

❸ 访问 event.venue，返回一个 Record 实例。

❹ 现在轻易就能获取 event.venue 的名称。

❺ Event 实例还根据 JSON 数据创建了一个 venue_serial 属性。

Event 是 Record 的子类，它增加了一个 venue 特性，用于获取链接的记录，还定制了 __repr__ 方法。

本节的代码在本书代码的 schedule_v2.py 模块中。这个示例的代码接近 60 行，我将分几部分给出，示例 22-11 中是增强后的 Record 类。

示例 22-11 schedule_v2.py：增加 fetch 方法的 Record 类

```
import inspect ❶
import json
```

```

JSON_PATH = 'data/osconfeed.json'

class Record:

    __index = None ❷

    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        return f'<{self.__class__.__name__} serial={self.serial!r}>'

    @staticmethod ❸
    def fetch(key):
        if Record.__index is None: ❹
            Record.__index = load()
        return Record.__index[key] ❺

```

- ❶ inspect 在示例 22-13 列出的 load 函数中使用。
- ❷ __index 私有类属性最终存放一个对 load 函数返回的字典的引用。
- ❸ 使用 staticmethod 装饰 fetch 方法，以此强调其效果不受调用它的实例或类的影响。
- ❹ 如有必要，请填写 Record.__index。
- ❺ 通过 Record.__index 获取指定 key 对应的记录。



这个示例展示了 staticmethod 的合理用途。fetch 方法始终处理类属性 Record.__index，即使是在子类中调用，例如 Event.fetch()（稍后说明）。声明为类方法会让人误解，因为用不到第一个参数 cls。

下面在 Event 类中使用一个特性，如示例 22-12 所示。

示例 22-12 schedule_v2.py: Event 类

```
class Event(Record): ❶

    def __repr__(self):
        try:
            return f'<{self.__class__.__name__} {self.name!r}>' ❷
        except AttributeError:
            return super().__repr__()

    @property
    def venue(self):
        key = f'venue.{self.venue_serial}'
        return self.__class__.fetch(key) ❸
```

❶ Event 扩展 Record。

❷ 如果实例有 `name` 属性，就用它来定制字符串表示形式。否则，委托给 Record 的 `__repr__` 方法。

❸ `venue` 特性根据 `venue_serial` 属性构建一个 `key`，传给从 Record 继承的类方法 `fetch`（使用 `self.__class__` 的原因稍后说明）。

在示例 22-12 中，`venue` 方法的第二行返回了 `self.__class__.fetch(key)`。为什么不直接调用 `self.fetch(key)` 呢？这种更简单的形式适用于这个 OSCON 数据集，因为任何事件记录都没有 `'fetch'` 键。但是，如果有一个事件记录有名为 `'fetch'` 的键，那么在那个 Event 实例中，`self.fetch` 获取的就是那个字段，而不是 Event 从 Record 继承的类方法 `fetch`。这个 bug 不易察觉，容易逃过测试，因为它的行为取决于所用的数据集。



从数据中创建实例属性的名称时，很容易遮盖类属性（例如方法），导致 bug，或者覆盖现有的实例属性，导致数据丢失。或许正是由于存在这些问题，Python 字典与 JavaScript 对象有着本质区别。

如果 Record 类的行为更像是映射，实现了动态的 `__getitem__` 方法，而不是动态的 `__getattr__` 方法，那么就不会受到覆盖或遮盖的影响，导致 bug。其实，把 Record 实现为映射或许更符合 Python 风格。然而，倘若真那样做，就无法研究动态属性编程的技巧和陷阱了。

这个示例的最后一部分是重写的 `load` 函数，如示例 22-13 所示。

示例 22-13 schedule_v2.py: `load` 函数

```
def load(path=JSON_PATH):
    records = {}
    with open(path) as fp:
        raw_data = json.load(fp)
        for collection, raw_records in raw_data['Schedule'].items():
            record_type = collection[:-1] ❶
            cls_name = record_type.capitalize() ❷
            cls = globals().get(cls_name, Record) ❸
            if inspect.isclass(cls) and issubclass(cls, Record): ❹
                factory = cls ❺
            else:
                factory = Record ❻
            for raw_record in raw_records: ❼
                key = f'{record_type}.{raw_record["serial"]}'
                records[key] = factory(**raw_record) ❽
    return records
```

❶ 目前，与 `schedule_v1.py`（参见示例 22-9）相比，`load` 函数没有改动。

❷ 把 `record_type` 的首字母变成大写，尝试获取类名，例如，`'event'` 变成 `'Event'`。

- ③ 从模块全局作用域中获取对应名称的对象。如果不存在相应的对象，就获取 `Record` 类。
- ④ 如果得到的对象是类，而且是 `Record` 的子类.....
- ⑤ 就绑定 `factory` 名称。这意味着，`factory` 可以是 `Record` 的任何子类，具体取决于 `record_type`。
- ⑥ 否则，把 `factory` 名称绑定为 `Record`。
- ⑦ 创建 `key` 和保存记录的 `for` 循环与之前一样，只是.....
- ⑧`records` 中存储的对象由 `factory` 构建。根据 `record_type`，`factory` 可能是 `Record` 或其子类，例如 `Event`。

注意，唯一有自定义类的 `record_type` 是 `Event`。如果也定义了 `Speaker` 或 `Venue` 类，那么 `load` 函数将自动使用这些类构建和保存记录，不再使用默认的 `Record` 类。

下面采用同样的方式在 `Events` 类中添加 `speakers` 特性。

22.3.3 第 3 步：用特性覆盖现有属性

示例 22-12 中的 `venue` 属性的名称不对应 "events" 记录合集集中的任何一个字段名称，其数据来自 `venue_serial` 字段。相比之下，`events` 合集集中的每个记录都有一个列出多个编号的 `speakers` 字段。我们打算通过 `Event` 实例的 `speakers` 特性对外开放这部分信息，返回一组 `Record` 实例。如此一来，名称就出现了冲突，需要特别处理，如示例 22-14 所示。

示例 22-14 `schedule_v3.py`: `speakers` 特性

```
@property
def speakers(self):
    spkr_serials = self.__dict__['speakers'] ❶
```

```
fetch = self.__class__.fetch
return [fetch(f'speaker.{key}')]
        for key in spkr_serials] ❷
```

❶ 我们需要的数据存储 `speakers` 属性中，因此必须从实例属性 `__dict__` 中直接获取，以免递归调用 `speakers` 特性。

❷ 获取键与 `spkr_serials` 中的数值匹配的所有记录，构成一个列表返回。

在 `speakers` 方法中，尝试读取 `self.speakers` 将调用相应的特性，从而很快就会抛出 `RecursionError`。然而，通过 `self.__dict__['speakers']` 读取同样的数据，Python 将绕过获取属性的常规算法，不调用特性，因此也就没有递归调用问题。鉴于此，直接通过对象的 `__dict__` 属性读写数据是 Python 元编程常见的技巧。



求解 `obj.my_attr` 时，解释器首先检查 `obj` 的类，如果类有名为 `my_attr` 的特性，那么同名实例属性就会被遮盖。22.5.1 节将举例说明。第 23 章将揭露，特性以描述符实现——一种更强大且更一般的抽象。

编写示例 22-14 中的列表推导式时，程序员的直觉告诉我：“这个操作消耗的资源估计不少。”其实不然，因为 OSCON 数据集中的事件没有多少个演讲者，所以把问题复杂化属于过早优化。然而，我们经常需要缓存特性——但是有一些注意事项。接下来的几个示例将演示具体做法。

22.3.4 第 4 步：自己实现特性缓存

特性经常需要缓存，因为普遍预期 `event.venue` 之类的表达式不消耗什么资源。⁸ 如果 `Event` 中的特性内部用到的 `Record.fetch` 方法需要查询数据库或 Web API，则缓存必不可少。

⁸其实，这是本章开头提到的 Meyer 提出的统一访问原则的一个缺点。如果对相关讨论感兴趣，请阅读本章的“杂谈”。

在本书第 1 版中，我自己动手为 `speakers` 方法实现了缓存逻辑，如示例 22-15 所示。

示例 22-15 使用 `hasattr` 禁用键共享优化，自己实现缓存逻辑

```
@property
def speakers(self):
    if not hasattr(self, '__speaker_objs'): ❶
        spkr_serials = self.__dict__['speakers']
        fetch = self.__class__.fetch
        self.__speaker_objs = [fetch(f'speaker.{key}')
                               for key in spkr_serials]
    return self.__speaker_objs ❷
```

❶ 如果实例没有名为 `__speaker_objs` 的属性，就获取演讲者对象，存入该属性名下。

❷ 返回 `self.__speaker_objs`。

示例 22-15 中实现的缓存很简单，不过在实例初始化之后创建属性违背了“PEP 412—Key-Sharing Dictionary”提出的优化措施（详见 3.9 节）。如果数据集体量较大，那么内存用量上的差异还是很大的。

自己实现缓存方案时，如果想兼顾键共享优化，则需要在 `Event` 类中定义 `__init__` 方法，把 `__speaker_objs` 初始化为 `None`，然后在 `speakers` 方法中再做 `hasattr` 检查，如示例 22-16 所示。

示例 22-16 在 `__init__` 中定义存储数据的属性，利用键共享优化措施

```
class Event(Record):

    def __init__(self, **kwargs):
        self.__speaker_objs = None
```



```
        super().__init__(**kwargs)

# 省略15行...
    @property
    def speakers(self):
        if self.__speaker_objs is None:
            spkr_serials = self.__dict__['speakers']
            fetch = self.__class__.fetch
            self.__speaker_objs = [fetch(f'speaker.{key}')]
            for key in spkr_serials]
        return self.__speaker_objs
```

示例 22-15 和示例 22-16 用到的缓存技术在遗留的 Python 基准代码中十分常见。然而，在多线程程序中，像这样自己实现的缓存容易引入竞争条件，导致数据损坏。如果两个线程同时读取一个尚未缓存的特性，那么第一个线程需要把计算得到的数据存入缓存属性（本例中的 `__speaker_objs`）名下，当第二个线程从缓存中读取数据时，计算过程可能还未结束。

幸好，Python 3.8 引入了对线程安全的

`@functools.cached_property` 装饰器。然而，使用这个装饰器时有一些注意事项，22.3.5 节将详述。

22.3.5 第 5 步：使用 `functools` 缓存特性

`functools` 模块提供了 3 个用于缓存的装饰器。9.9.1 节已经讲过 `@cache` 和 `@lru_cache`，还有一个是 Python 3.8 引入的 `@cached_property`。

`functools.cached_property` 装饰器把方法的结果缓存在同名实例属性中。例如，在示例 22-17 中，`venue` 方法计算得到的值存储在 `self` 的 `venue` 属性中。缓存之后，当客户代码尝试读取 `venue` 时，数据来自新创建的实例属性 `venue`，不再调用 `venue` 方法。

示例 22-17 `@cached_property` 的简单用法

```
@cached_property
def venue(self):
    key = f'venue.{self.venue_serial}'
    return self.__class__.fetch(key)
```

22.3.3 节讲过，特性会遮盖同名实例属性。既然如此，`@cached_property` 是如何运作的呢？如果特性覆盖了实例属性，那么 `venue` 属性将被忽略，始终调用 `venue` 方法，每次都计算 `key` 并运行 `fetch`。

答案可能会让你感到意外：`cached_property` 用词不当。`@cached_property` 装饰器不创建完整的特性，而是创建一个**非覆盖型描述符**（nonoverriding descriptor）。描述符是一种对象，负责管理如何访问另一个类的属性，第 23 章将深入探讨。`property` 装饰器是一个高级 API，用于创建**覆盖型描述符**（overriding descriptor）。第 23 章将全面说明覆盖型描述符与非覆盖型描述符之间的区别。

现在，暂且把底层实现搁在一旁，将重点放在从用户的视角来看，`cached_property` 和 `property` 之间有什么区别。Raymond Hettinger 在 Python 文档中已经解释得很清楚了。

`cached_property()` 的机制与 `property()` 不太一样。常规特性阻止属性写入，除非定义了设值方法。与之相反，`cached_property` 允许写入。

`cached_property` 装饰器仅在执行查找且不存在同名属性时运行。一旦运行，`cached_property` 就会写入同名属性。后续的属性读取和写入操作优先于 `cached_property` 方法，行为就像普通的属性一样。

缓存的值可通过删除属性清空。如此一来，`cached_property` 方法将再次运行。⁹

⁹来源于 `@functools.cached_property` 文档。我之所以知道这些说明出自 Raymond Hettinger 之手，是因为这几段话出现在他对我发起的工单“bpo42781—`functools.cached_property` docs should explain that it is non-overriding”的回复中。Hettinger 是 Python 官方文档和标准库的主要贡献者。他还编写了“Descriptor HowTo Guide”，这是第 23 章的主要参考资料。

回到 `Event` 类：鉴于 `@cached_property` 的这种行为，不太适合用于装饰 `speakers`，因为该方法依赖一个同名的现有属性，而这个属性已经存在，用于存储事件演讲者的编号。



`@cached_property` 有一些不可忽略的局限。

- 如果被装饰的方法已经依赖一个同名实例属性，则不能直接替代 `@property`。
- 不能在定义了 `__slots__` 的类中使用。
- 违背对实例属性 `__dict__` 的键共享优化，因为需要在初始化之后创建一个实例属性。

尽管有一些局限，但是 `@cached_property` 以简单的方式解决了一类常见问题，而且它是线程安全的。通过 Python 源码可以看到，`@cached_property` 用到了**重入锁**。

`speakers` 特性可以使用 `@cached_property` 文档推荐的另一种方案：叠放装饰器 `@property` 和 `@cache`，如示例 22-18 所示。

示例 22-18 叠放 `@property` 和 `@cache`

```
@property ❶
@cache ❷
def speakers(self):
    spkr_serials = self.__dict__['speakers']
    fetch = self.__class__.fetch
    return [fetch(f'speaker.{key}')]
            for key in spkr_serials]
```

❶ 顺序不能错：`@property` 在上.....

②@cache 在下。

根据 9.9.1 节的“叠放装饰器”提示栏对叠放句法的说明，示例 22-18 前 3 行代码等同于如下内容。

```
speakers = property(cache(speakers))
```

@cache 应用在 speakers 上，返回一个新函数。这个新函数又被 @property 装饰，替换成一个新构建的特性。

借助 OSCON 数据集对只读特性和缓存装饰器的讨论到此结束。22.4 节将通过一系列新示例创建读写特性。

22.4 使用特性验证属性

除了计算属性值，特性还可用于实施业务规则，把公开属性变成受读值方法和设值方法保护的属性，而客户代码不受影响。本节将通过一个详尽的示例进行说明。

22.4.1 LineItem 类第 1 版：表示订单中商品的类

假设有个销售散装有机食物的电商应用程序，客户可以按量订购坚果、干果或杂粮。在这个系统中，每个订单中有一系列商品，各个商品可以使用示例 22-19 中类的实例表示。

示例 22-19 bulkfood_v1.py: 最简单的 LineItem 类

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

这个类很精简，或许太简单了。示例 22-20 揭示了一个问题。

示例 22-20 重量为负值时，金额小计为负值

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> raisins.subtotal()
69.5
>>> raisins.weight = -20    # 无效输入...
>>> raisins.subtotal()      # 无效输出...
-139.0
```

这个示例像玩具一样，但是没有想象中那么好玩。下面是亚马逊早期的一个真实故事。

我们发现顾客买书时可以把数量设为负数！然后，我们把金额打到顾客的信用卡上，苦苦等待他们把书寄出（想得美）。

——Jeff Bezos
亚马逊创始人和 CEO¹⁰

¹⁰摘自《华尔街日报》的文章“Birth of a Salesman”（2011 年 10 月 15 日），这是 Jeff Bezos 的原话。

这个问题怎么解决呢？可以修改 `LineItem` 类的接口，使用读值方法和设值方法管理 `weight` 属性。这是 Java 采用的方式，完全可行。

但是，如果能直接设定商品的 `weight` 属性，则会显得更自然。此外，系统可能在生产环境中，而其他部分已经直接访问 `item.weight` 了。此时，符合 Python 风格的做法是，把数据属性换成特性。

22.4.2 `LineItem` 类第 2 版：能验证值的特性

实现特性之后，可以使用读值方法和设值方法，但是 `LineItem` 类的接口保持不变（例如，设置 `LineItem` 对象的 `weight` 属性依然写成 `raisins.weight = 12`）。

示例 22-21 给出了可读写的 `weight` 特性的代码。

示例 22-21 `bulkfood_v2.py`: 定义了 `weight` 特性的 `LineItem` 类

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight ❶
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property ❷
    def weight(self): ❸
        return self.__weight ❹

    @weight.setter ❺
    def weight(self, value):
        if value > 0:
            self.__weight = value ❻
        else:
            raise ValueError('value must be > 0') ❼
```

❶ 这里已经使用特性的设值方法了，请确保所创建实例的 `weight` 属性不能为负值。

❷ `@property` 装饰读值方法。

❸ 实现特性的所有方法，其名称与公开属性的名称一样——`weight`。

❹ 真正的值存储在私有属性 `__weight` 中。

❺ 被装饰的读值方法有一个 `.setter` 属性，这个属性也是装饰器——把读值方法和设值方法绑定在一起。

⑥ 如果值大于零，就设置私有属性 `__weight`。

⑦ 否则，抛出 `ValueError` 异常。

注意，现在不能创建重量为无效值的 `LineItem` 对象了。

```
>>> walnuts = LineItem('walnuts', 0, 10.00)
Traceback (most recent call last):
...
ValueError: value must be > 0
```

现在，禁止用户为 `weight` 属性提供负值。虽然买家通常不能设置商品的价格，但是工作人员可能犯错，应用程序也可能存在 bug，导致 `LineItem` 对象的 `price` 属性为负值。为了防止出现这种情况，也可以把 `price` 属性变成特性，但是这样我们的代码中就会存在一些重复。

还记得第 17 章引述 Paul Graham 的那句话吗？他说：“当我发现自己的程序中用到了模式，我就觉得这表明某个地方出错了。”去除重复的良方是抽象。抽象特性的定义有两种方式：一是使用特性工厂函数，二是使用描述符类。后者更灵活，第 23 章将全面讨论。其实，特性本身就是使用描述符类实现的。不过，这里要继续探讨特性，实现一个特性工厂函数。

但是，在实现特性工厂函数之前，需要对特性有深入理解。

22.5 特性全解析

虽然内置的 `property` 经常用作装饰器，但它其实是一个类。在 Python 中，函数和类通常可以互换，因为二者都是可调用对象，而且 Python 没有实例化对象的 `new` 运算符，调用构造函数与调用工厂函数没有区别。此外，只要能返回新的可调用对象，取代被装饰的函数，二者都可以用作装饰器。

property 构造函数的完整签名如下所示。

```
property(fget=None, fset=None, fdel=None, doc=None)
```

所有参数都是可选的，如果没有把函数传给某个参数，那么得到的特性对象就不允许执行相应的操作。

property 类型在 Python 2.2 中引入，但是直到 Python 2.4 才出现 @ 装饰器句法，因此有那么几年，若想定义特性，只能把存取函数传给前两个参数。

不使用装饰器定义特性的“经典”句法如示例 22-22 所示。

示例 22-22 bulkfood_v2b.py：效果与示例 22-21 一样，只不过未使用装饰器

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    def get_weight(self): ❶
        return self.__weight

    def set_weight(self, value): ❷
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')

    weight = property(get_weight, set_weight) ❸
```

❶ 普通的读值方法。

❷ 普通的设值方法。

③ 构建 `property` 对象，赋值给一个公开类属性。

在某些情况下，这种经典形式比装饰器句法好，稍后讨论的特性工厂函数就是一例。但是，如果在方法众多的类主体中使用装饰器，则一眼就能看出哪些是读值方法，哪些是设值方法，而不用约定一种惯例，在方法名的前面加上 `get` 和 `set`。

类中的特性能影响实例属性的寻找方式，但一开始这种方式可能会让人觉得意外。22.5.1 节将详细说明。

22.5.1 特性覆盖实例属性

特性是类属性，但是特性管理的其实是实例属性的存取。

如 11.12 节所述，如果实例和所属的类有同名数据属性，那么实例属性就覆盖（或称遮盖）类属性——至少通过实例读取属性时是这样。示例 22-23 阐明了这一点。

示例 22-23 实例属性遮盖类属性 `data`

```
>>> class Class: ❶
...     data = 'the class data attr'
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>> obj = Class()
>>> vars(obj) ❷
{}
>>> obj.data ❸
'the class data attr'
>>> obj.data = 'bar' ❹
>>> vars(obj) ❺
{'data': 'bar'}
>>> obj.data ❻
'bar'
>>> Class.data ❼
'the class data attr'
```

- ❶ 定义 Class 类，这个类有两个类属性：data 属性和 prop 特性。
- ❷ vars 函数返回 obj 的 __dict__ 属性，可以看到，没有实例属性。
- ❸ 读取 obj.data，获取的是 Class.data 的值。
- ❹ 为 obj.data 赋值，创建一个实例属性。
- ❺ 审查实例，查看实例属性。
- ❻ 现在读取 obj.data，获取的是实例属性的值。从 obj 实例中读取属性时，实例属性 data 遮盖类属性 data。
- ❼ Class.data 属性的值完好无损。

下面尝试覆盖 obj 实例的 prop 特性。接着前面的控制台会话，输入示例 22-24 中的代码。

示例 22-24 实例属性不遮盖类特性（接续示例 22-23）

```
>>> Class.prop ❶
<property object at 0x1072b7408>
>>> obj.prop ❷
'the prop value'
>>> obj.prop = 'foo' ❸
Traceback (most recent call last):
...
AttributeError: can't set attribute
>>> obj.__dict__['prop'] = 'foo' ❹
>>> vars(obj) ❺
{'data': 'bar', 'prop': 'foo'}
>>> obj.prop ❻
'the prop value'
>>> Class.prop = 'baz' ❼
>>> obj.prop ❽
'foo'
```

- ❶ 直接从 `Class` 中读取 `prop` 特性，获取的是特性对象本身，不运行特性的读值方法。
- ❷ 读取 `obj.prop` 执行特性的读值方法。
- ❸ 尝试设置 `prop` 实例属性，结果失败了。
- ❹ 但是可以直接把 `'prop'` 存入 `obj.__dict__`。
- ❺ 可以看到，`obj` 现在有两个实例属性：`data` 和 `prop`。
- ❻ 然而，读取 `obj.prop` 时仍会运行特性的读值方法。特性未被实例属性遮盖。
- ❼ 覆盖 `Class.prop` 特性，销毁特性对象。
- ❽ 现在，`obj.prop` 获取的是实例属性。`Class.prop` 不是特性了，因此不再覆盖 `obj.prop`。

最后再举一个例子，为 `Class` 类增加一个特性，覆盖实例属性。示例 22-25 接续示例 22-24。

示例 22-25 新添的类特性遮盖现有的实例属性（接续示例 22-24）

```
>>> obj.data ❶
'bar'
>>> Class.data ❷
'the class data attr'
>>> Class.data = property(lambda self: 'the "data" prop value') ❸
>>> obj.data ❹
'the "data" prop value'
>>> del Class.data ❺
>>> obj.data ❻
'bar'
```

- ❶ `obj.data` 获取的是实例属性 `data`。

- ❷ `Class.data` 获取的是类属性 `data`。
- ❸ 使用新特性覆盖 `Class.data`。
- ❹ 现在 `obj.data` 被 `Class.data` 特性遮盖了。
- ❺ 删除特性。
- ❻ 现在恢复原样，`obj.data` 获取的是实例属性 `data`。

本节的主要观点是，像 `obj.data` 这样的表达式不会从 `obj` 而是从 `obj.__class__` 开始寻找 `data`，而且仅当类中没有名为 `data` 的特性时，Python 才会在 `obj` 实例中寻找。这条规则适用于全体覆盖型描述符，包括特性。第 23 章将进一步讨论描述符。

现在回到特性。各种 Python 代码单元（模块、函数、类和方法）都可以有文档字符串。下一个主题是如何为特性添加文档。

22.5.2 特性的文档

当控制台中的 `help()` 函数或 IDE 等工具需要显示特性的文档时，会从特性的 `__doc__` 属性中提取信息。

如果使用经典调用句法，那么 `property` 对象的文档通过 `doc` 参数设置。

```
weight = property(get_weight, set_weight, doc='weight in kilograms')
```

读值方法（有 `@property` 装饰器的方法）的文档字符串作为一个整体，变成了特性的文档。图 22-1 显示的是从示例 22-26 的代码中生成的帮助界面。



图 22-1：在 Python 控制台中执行 `help(Foo.bar)` 命令和 `help(Foo)` 命令时的截图，源码在示例 22-26 中

示例 22-26 一个特性的文档

```
class Foo:

    @property
    def bar(self):
        """The bar attribute"""
        return self.__dict__['bar']

    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

至此，特性相关的基础知识已经介绍完毕。下面回过头来解决前面遇到的问题：保护 `LineItem` 对象的 `weight` 属性和 `price` 属性，只允许设为大于零的值，但是不用手动实现两对几乎一样的读值方法和设值方法。

22.6 定义一个特性工厂函数

我们将定义一个名为 `quantity` 的特性工厂函数。之所以采用这个名称，是因为在这个应用程序中要管理的属性表示不能为负数或零的量。示例 22-27 是 `LineItem` 类的简洁版，用到了 `quantity` 特性

的两个实例：一个用于管理 `weight` 属性，另一个用于管理 `price` 属性。

示例 22-27 `bulkfood_v2prop.py`：使用特性工厂函数
`quantity`

```
class LineItem:
    weight = quantity('weight') ❶
    price = quantity('price') ❷

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight ❸
        self.price = price

    def subtotal(self):
        return self.weight * self.price ❹
```

- ❶ 使用工厂函数把第一个自定义的特性 `weight` 定义为类属性。
- ❷ 第二次调用，构建另一个自定义的特性，即 `price`。
- ❸ 这里，特性已经激活，确保不能把 `weight` 设为负数或零。
- ❹ 这里也用到了特性，使用特性获取实例中存储的值。

如前所述，特性是类属性。构建各个 `quantity` 特性对象时，要传入 `LineItem` 实例属性的名称，让特性管理。非常不幸，这一行要输入单词 `weight` 两次。

```
weight = quantity('weight')
```

这里很难避免重复输入，因为特性根本不知道要绑定哪个类属性。记住，赋值语句的右侧先求值，因此调用 `quantity()` 时，`weight` 类属性还不存在。



如果想改进 `quantity` 特性，避免用户重复输入属性名，那么这对元编程来说是个挑战。第 23 章将解决这个问题。

示例 22-28 给出了 `quantity` 特性工厂函数的实现。¹¹

¹¹这段代码改编自《Python Cookbook（第 3 版）中文版》一书的 9.21 节“避免出现重复的属性方法”。

示例 22-28 `bulkfood_v2prop.py`: `quantity` 特性工厂函数

```
def quantity(storage_name): ❶

    def qty_getter(instance): ❷
        return instance.__dict__[storage_name] ❸

    def qty_setter(instance, value): ❹
        if value > 0:
            instance.__dict__[storage_name] = value ❺
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter) ❻
```

❶ `storage_name` 参数确定各个特性的数据存储在哪里。对 `weight` 特性来说，存储的名称是 `'weight'`。

❷ `qty_getter` 函数的第一个参数可以命名为 `self`，但是这么做很奇怪，因为 `qty_getter` 函数不在类主体中。`instance` 指代要把属性存储其中的 `LineItem` 实例。

❸ `qty_getter` 引用了 `storage_name`，把它保存在这个函数的闭包里。值直接从 `instance.__dict__` 中获取，以绕过特性，防止无限递归。

❹ 定义 `qty_setter` 函数，第一个参数也是 `instance`。

⑤ 将 `value` 直接存入 `instance.__dict__`，这也是为了绕过特性。

⑥ 构建一个自定义的特性对象，然后将其返回。

示例 22-28 中值得仔细分析的代码是与 `storage_name` 变量相关的部分。当使用传统方式定义特性时，用于存储值的属性名硬编码在读写方法和设值方法中。但是，这里的函数 `qty_getter` 和 `qty_setter` 是通用的，要依靠 `storage_name` 变量来判断从 `__dict__` 中获取哪个属性，或者设置哪个属性。每次调用 `quantity` 工厂函数构建属性时，都要把 `storage_name` 参数设为独一无二的值。

在工厂函数的最后一行，使用 `property` 对象包装函数 `qty_getter` 和 `qty_setter`。当需要运行这两个函数时，它们会从闭包中读取 `storage_name`，确定从哪里获取属性的值，或者在哪里存储属性的值。

在示例 22-29 中，我创建并审查了一个 `LineItem` 实例，以说明存储值的是哪个属性。

示例 22-29 `bulkfood_v2prop.py`：探索特性和存储值的属性

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price ❶
(8, 13.95)
>>> nutmeg.__dict__ ❷
{'description': 'Moluccan nutmeg', 'weight': 8, 'price': 13.95}
```

❶ 通过特性读取 `weight` 和 `price`，这会遮盖同名实例属性。

❷ 使用 `vars` 函数审查 `nutmeg` 实例，查看真正用于存储值的实例属性。

注意，工厂函数构建的特性利用了 22.5.1 节所述的行为：weight 特性覆盖了 weight 实例属性，因此对 self.weight 或 nutmeg.weight 的每个引用都由特性函数处理，只有直接存取 `__dict__` 属性才能绕过特性的处理逻辑。

示例 22-28 中的代码有点儿难理解，不过很简洁，与示例 22-21 中使用装饰器声明读值方法和设值方法的代码行数一样，但是那里只定义了 weight 特性。示例 22-27 中定义的 `LineItem` 类没有干扰人的读值方法和设值方法，看起来舒服多了。

在真实的系统中，分散在多个类中的多个字段可能要做同样的验证，因此最好把 `quantity` 工厂函数放在实用工具模块中，以便重复使用。最终，在第 23 章，我们将重构这个简单的工厂函数，改成更易扩展的描述符类，使用专门的子类执行不同的验证。

接下来分析删除属性的问题，以此结束对特性的讨论。

22.7 处理属性删除操作

`del` 语句不仅可以删除变量，也可以删除属性。

```
>>> class Demo:
...     pass
...
>>> d = Demo()
>>> d.color = 'green'
>>> d.color
'green'
>>> del d.color
>>> d.color
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Demo' object has no attribute 'color'
```

其实，Python 编程不常删除属性，通过特性删除属性则更少见。但是，Python 支持这么做，我可以虚构一个示例，演示删除方式。

定义特性时，可以使用 `@my_property.deleter` 装饰器包装一个方法，负责删除特性管理的属性。下面我来兑现承诺，虚构一个示例。这个示例的灵感来自电影《巨蟒与圣杯》中的黑衣骑士角色，如示例 22-30 所示。

示例 22-30 blackknight.py

```
class BlackKnight:

    def __init__(self):
        self.phrases = [
            ('an arm', "'Tis but a scratch."),
            ('another arm', "It's just a flesh wound."),
            ('a leg', "I'm invincible!"),
            ('another leg', "All right, we'll call it a draw.")
        ]

    @property
    def member(self):
        print('next member is:')
        return self.phrases[0][0]

    @member.deleter
    def member(self):
        member, text = self.phrases.pop(0)
        print(f'BLACK KNIGHT (loses {member}) -- {text}')
```

blackknight.py 脚本的 doctest 在示例 22-31 中。

示例 22-31 blackknight.py: 示例 22-30 的 doctest (黑衣骑士从不屈服)

```
>>> knight = BlackKnight()
>>> knight.member
next member is:
'an arm'
>>> del knight.member
BLACK KNIGHT (loses an arm) -- 'Tis but a scratch.
>>> del knight.member
BLACK KNIGHT (loses another arm) -- It's just a flesh wound.
>>> del knight.member
BLACK KNIGHT (loses a leg) -- I'm invincible!
>>> del knight.member
```

```
BLACK KNIGHT (loses another leg) -- All right, we'll call it a draw.
```

在不使用装饰器的经典调用句中，`fdel` 参数用于设置删值函数。例如，在 `BlackKnight` 类的主体中可以像下面这样创建 `member` 特性。

```
member = property(member_getter, fdel=member_deleter)
```

如果不使用特性，则还可以实现底层特殊方法 `__delattr__`，处理删除属性的操作（参见 22.8.3 节）。以下练习留给喜欢拖延的读者：虚构一个类，定义 `__delattr__` 方法。

特性是一个强大的功能，不过有时更适合使用简单的或底层的替代方案。22.8 节将概览 Python 为动态属性编程提供的部分核心 API。

22.8 处理属性的重要属性和函数

本章及本书前面的章节多次用到 Python 为处理动态属性而提供的内置函数和特殊方法。由于这些函数和方法的文档散布在官方文档中，因此本章专门用一节集中介绍。

22.8.1 影响属性处理方式的特殊属性

22.8.2 节和 22.8.3 节中的很多函数和特殊方法的行为受以下 3 个特殊属性的影响。

`__class__`

对象所属类的引用（例如，`obj.__class__` 与 `type(obj)` 的作用相同）。Python 的某些特殊方法（例如 `__getattr__`），只在对象的类中而不在实例中寻找。

`__dict__`

存储对象或类的可写属性的映射。有 `__dict__` 属性的对象，任何时候都能随意设置新属性。对于有 `__slots__` 属性的类，其实例可能没有 `__dict__` 属性。参见下面对 `__slots__` 属性的说明。

`__slots__`

类可以定义这个属性，节省内存。`__slots__` 属性的值是一个字符串元组，列出了允许有的属性。¹² 如果 `__slots__` 中没有 `'__dict__'`，那么该类的实例就没有 `__dict__` 属性，而只允许有 `__slots__` 中列出的属性。详见 11.11 节。

¹² Alex Martelli 指出，`__slots__` 属性的值虽然可以是一个列表，但是最好始终使用元组，因为处理完类的主体之后再修改 `__slots__` 列表没有任何作用。因此使用可变的序列容易让人误解。

22.8.2 处理属性的内置函数

以下 5 个内置函数会对对象的属性做读、写和内省操作。

`dir([object])`

列出对象的大多数属性。官方文档说，`dir` 函数的目的是交互式使用，因此没有提供完整的属性列表，只列出了一组“重要的”属性名。`dir` 函数能审查有或没有 `__dict__` 属性的对象。`dir` 函数不列出 `__dict__` 属性本身，但会列出其中的键。`dir` 函数也不列出类的几个特殊属性，例如 `__mro__`、`__bases__` 和 `__name__`。实现特殊方法 `__dir__` 可以自定义 `dir` 函数的输出，如示例 22-4 所示。如果没有指定可选的 `object` 参数，则 `dir` 函数会列出当前作用域中的名称。

`getattr(object, name[, default])`

从 `object` 对象中获取 `name` 字符串对应的属性。主要用于获取事先不知道名称的属性（或方法）。获取的属性可能来自对象所属的

类或超类。如果指定的属性不存在，则 `getattr` 函数会抛出 `AttributeError` 异常，或者返回 `default` 参数的值（如果提供了的话）。在标准库中，`cmd` 包的 `Cmd.onecmd` 方法就很好地利用了 `getattr`，用于获取并执行用户定义的命令。

```
hasattr(object, name)
```

如果 `object` 对象中存在指定的属性，或者能以某种方式（例如继承）通过 `object` 对象获取指定的属性，那么就返回 `True`。Python 标准库文档指出：“这个函数的实现方法是调用 `getattr(object, name)` 函数，看看是否抛出 `AttributeError` 异常。”

```
setattr(object, name, value)
```

把 `object` 对象指定属性的值设为 `value`，前提是 `object` 对象能接受提供的值。这可能创建一个新属性，也可能覆盖现有属性。

```
vars([object])
```

返回 `object` 对象的 `__dict__` 属性。如果实例所属的类定义了 `__slots__` 属性且实例没有 `__dict__` 属性，那么 `vars` 函数就不能处理（相反，`dir` 函数能处理）这样的实例。如果没有指定参数，那么 `vars()` 函数的作用与 `locals()` 函数一样：返回表示本地作用域的字典。

22.8.3 处理属性的特殊方法

在用户定义的类中，以下特殊方法用于获取、设置、删除和列出属性。

使用点号表示法或内置的函数 `getattr`、`hasattr` 和 `setattr` 存取属性都会触发本节后面列出的相应的特殊方法。但是，直接通过实

例的 `__dict__` 属性读写属性不会触发这些特殊方法——必要时，这是绕过特殊方法的常用方式。

《Python 语言参考手册》中的 3.3.11 节“特殊方法查找”警告说：

对用户自己定义的类来说，如果隐式调用特殊方法，那么仅当特殊方法在对象所属的类型上而不是在对象的实例字典中定义时，才能确保调用成功。

也就是说，要假定特殊方法从类上获取，即便操作目标是实例也是如此。因此，特殊方法不被同名实例属性遮盖。

在以下示例中，假设有一个名为 `Class` 的类，`obj` 是 `Class` 类的实例，`attr` 是 `obj` 的属性。

不管使用点号表示法存取属性，还是使用 22.8.2 节列出的某个内置函数，都会触发以下特殊方法中的一个。例如，`obj.attr` 和 `getattr(obj, 'attr', 42)` 都会触发 `Class.__getattr__(obj, 'attr')` 方法。

```
__delattr__(self, name)
```

只要使用 `del` 语句删除属性，就会调用这个方法。例如，`del obj.attr` 语句触发 `Class.__delattr__(obj, 'attr')`。如果 `attr` 是一个特性，而且类实现了 `__delattr__` 方法，则永不调用特性的删值方法。

```
__dir__(self)
```

在对象上调用 `dir` 函数时会调用这个方法，以列出属性。例如，`dir(obj)` 触发 `Class.__dir__(obj)`。所有现代的 Python 控制台 Tab 补全时也使用该方法。

```
__getattr__(self, name)
```

仅当获取指定的属性失败，搜索过 `obj`、`Class` 及其超类之后会调用这个方法。表达式 `obj.no_such_attr`、`getattr(obj, 'no_such_attr')` 和 `hasattr(obj, 'no_such_attr')` 可能触发 `Class.__getattr__(obj, 'no_such_attr')`，但是，仅当在 `obj`、`Class` 及其超类中找不到指定的属性时才触发。

```
__getattribute__(self, name)
```

在 Python 代码中尝试直接获取指定名称的属性时始终调用这个方法 [某些情况下（例如获取 `__repr__` 方法时）解释器可能会绕过该方法]。点号表示法与内置函数 `getattr` 和 `hasattr` 会触发这个方法。`__getattr__` 仅在 `__getattribute__` 之后，而且仅当 `__getattribute__` 抛出 `AttributeError` 时调用。为了在获取 `obj` 实例的属性时不导致无限递归，`__getattribute__` 方法的实现要使用 `super().__getattribute__(obj, name)`。

```
__setattr__(self, name, value)
```

尝试设置指定名称的属性时总会调用这个方法。点号表示法和内置函数 `setattr` 会触发这个方法。例如，`obj.attr = 42` 和 `setattr(obj, 'attr', 42)` 都会触发 `Class.__setattr__(obj, 'attr', 42)`。



其实，不管怎样，特殊方法 `__getattribute__` 和 `__setattr__` 都会调用，几乎影响每一次属性存取，因此比 `__getattr__` 方法（只处理不存在的属性名）更难正确使用。与定义这些特殊方法相比，使用特性或描述符相对不易出错。

对特性、特殊方法和其他动态属性编程技术的讨论到此结束。

22.9 本章小结

本章的话题是动态属性编程。我们首先举了几个实例，定义了几个简单的类，以简化处理 JSON 数据集的方式。第一个示例是 `FrozenJSON` 类，它可以把嵌套的字典和列表转换成嵌套的 `FrozenJSON` 实例和实例列表。`FrozenJSON` 类的代码展示了如何使用特殊方法 `__getattr__` 在读取属性时即时转换数据结构。`FrozenJSON` 类的最后一版展示了如何使用 `__new__` 构造方法把一个类转换成灵活的对象工厂函数，不受实例本身的限制。

然后，我们把 JSON 数据集转换成一个字典，以存储 `Record` 类的实例。第 1 版的 `Record` 类只有几行代码，介绍了“集束”惯用法：使用传给 `__init__` 方法的关键字参数，调用 `self.__dict__.update(**kwargs)` 构建任意属性。第 2 版增加了 `Event` 类，通过特性自动获取链接的记录。有些时候，计算特性的值需要缓存，所以本章介绍了几种缓存方式。

认识到 `@functools.cached_property` 并不总是适用之后，我们采用了一种替代方案：结合 `@property` 和 `@functools.cache`，把前者放在上方。

接下来，继续讨论特性。我们定义的 `LineItem` 类中有一个特性，以确保 `weight` 属性的值不能是对业务没有意义的负数或零。然后，我们深入说明了特性的句法和语义。随后又创建了一个特性工厂函数，在不定义多个读值方法和设值方法的前提下，对 `weight` 属性和 `price` 属性做相同的验证。那个特性工厂函数用到了几个精妙的概念（例如闭包和被特性覆盖的实例属性），提供了优雅的通用方案，代码行数与手动编写用来验证单个属性的特性一样多。

最后，我们简要说明了如何使用特性处理删除属性的操作，随后概览了 Python 核心语言为支持属性元编程而提供的重要的特殊属性、内置函数和特殊方法。

22.10 延伸阅读

属性处理和内置的内省函数的官方文档在 Python 标准库文档的第 2 章中，题为“Built-in Functions”。相关的特殊方法和特殊属性

`__slots__` 在《Python 语言参考手册》的 3.3.2 节“自定义属性访问”中说明。调用特殊方法会跳过实例的语义原因在《Python 语言参考手册》的 3.3.11 节“特殊方法查找”中说明。在 Python 标准库文档中，第 4 章“Built-in Types”中有一节“4.13. Special Attributes”说明了 `__class__` 属性和 `__dict__` 属性。

《Python Cookbook（第 3 版）中文版》一书中有几个经典实例涉及本章的话题，不过这里要重点提出 3 个：8.8 节“在子类中扩展特性”，解决了在继承自超类的特性中覆盖方法这个棘手问题；8.15 节“委托属性的访问”，实现了一个代理类，展示了本书 22.8.3 节所列的大多数特殊方法；还有出色的 9.21 节“避免出现重复的特性方法”，示例 22-28 中定义的特性工厂函数就以该节为基础。

Python in a Nutshell, 3rd ed. (Alex Martelli、Anna Ravenscroft 和 Steve Holden 著) 一书严谨而客观。讲到特性时，只用了 3 页，但这是由于该书采用了符合逻辑的行文方式：之前的 15 页已经对 Python 的类做了详尽说明，包括描述符，而特性背后就是使用描述符实现的。因此在讲到特性时，3 位作者可以在 3 页的篇幅中发表很多见解，例如本章开篇引用的那句话。

本章开头引用的统一访问原则定义出自 Bertrand Meyer，他提出了契约式设计（Design by Contract）方法论，设计了 Eiffel 语言，还撰写了优秀著作 *Object-Oriented Software Construction, 2nd ed.*。该书前 6 章对面向对象分析和设计相关概念的介绍是我见过最好的之一。第 11 章介绍了契约式设计，第 35 章阐述了他对几门重要的面向对象语言的评价，包括 Simula、Smalltalk、CLOS（Common Lisp Object System）、Objective-C、C++ 和 Java，还对其他语言做了简要评述。直到该书的最后一页他才披露，他使用的极具可读性的“表示法”看似伪代码，其实出自 Eiffel 语言。

杂谈

从美学的角度来看，Meyer 提出的统一访问原则很吸引人。作为使用 API 的程序员，我不应该关心 `product.price` 只是获取数据属性还是执行计算。但是，作为消费者和公民，我应该关心：在电子商务如此发达的今天，`product.price` 的值通常取决于这个问题由谁提出，因此它绝不仅仅是一个数据属性。其实，如果查询来自网店外部（例如比价引擎），那么价格通常会低一些。显然，这对喜欢浏览特定网店的忠实消费者来说，利益受到了损害。但是我不同意。

上一段内容离题了，可是提出了一个与编程有关的问题：虽然统一访问原则在理想的世界中完全合理，但在现实中，API 的用户可能需要知道读取 `product.price` 是否太耗资源或时间。一般来说，这是编程抽象问题：抽象之后很难推算求解表达式的运行时成本。另外，抽象能让用户用更少的代码完成更多的工作。这是一种权衡。Ward Cunningham 的维基对软件工程方面的话题有很多独到的见解，他对统一访问原则的功过也做了富有洞察力的论述。

在面向对象编程语言中，是否遵守统一访问原则通常体现在句法上：究竟是读取公开的数据属性，还是调用读值方法和设值方法。

Smalltalk 和 Ruby 使用简单而优雅的方式解决这个问题：根本不支持公开的数据属性。在这两门语言中，所有实例属性都是私有的，必须通过方法来存取。不过，这两门语言的句法把这个过程变得毫不费力：在 Ruby 中，`product.price` 调用读值方法 `price`；在 Smalltalk 中，只需使用 `product price`。

Java 采用的是另一种方式，即让程序员在 4 种访问级别修饰符中选择——包括没有名称的默认级别，Java 官方教程将其称为“包级私有”。

不过，普通大众并不认同 Java 设计者制定的这种句法。Java 世界的人都认为，属性应该是私有的，但是每一次都要写出

`private`，因为这不是默认的访问级别。如果所有属性都是私有的，那么从类外部访问属性就必须使用存取方法。一些 Java IDE 提供了自动生成存取方法的快捷方式。但是，当 6 个月后不得不阅读代码时，IDE 并没有多大帮助。我们要在众多什么也没做的存取方法中找出所需的那一个，添加实现某些业务逻辑所需的值。

Alex Martelli 把存取方法称为“愚蠢的惯用法”，这道出了 Python 社区中大多数人的心声。他举了下面两个例子，虽然二者外观差异很大，但是作用相同。¹³

```
someInstance.widgetCounter += 1
# 而不用...
someInstance.setWidgetCounter(someInstance.getWidgetCounter()
+ 1)
```

设计 API 时，我有时会想，能否把没有参数（除了 `self`）、返回一个值（除了 `None`）的纯函数（没有副作用）替换成只读特性。在本章中，`LineItem.subtotal` 方法（参见示例 22-27）就可以替换成只读特性。当然，用于修改对象的方法（例如 `my_list.clear()`）不在此列。把这样的方法变成特性是个糟糕的想法，因为直接访问 `my_list.clear` 就会删除列表中的内容。

在 GPIO 库 Pingo（参见 3.5.2 节）中，大多数用户级别的 API 基于特性实现。例如，为了读取模拟针脚的当前值，用户会编写 `pin.value`；为了设置数字针脚的模式，则会编写 `pin.mode = OUT`。在背后，读取模拟针脚的值或设置数字针脚的模式可能涉及大量代码，这取决于具体的主板驱动。我们决定在 Pingo 中使用特性，是因为想让 API 用起来舒服，即便在 Jupyter Notebook 等交互环境中也是如此，而且觉得 `pin.mode = OUT` 看起来和输入起来都比 `pin.set_mode(OUT)` 容易。

我觉得 Smalltalk 和 Ruby 的处理方式很简洁，但也认为 Python 的处理方式比 Java 更合理。一开始，可以从简单的方式入手，把数据成员定义为公开的属性，因为我们知道这些属性始终可以使用特性（或第 23 章讨论的描述符）包装。

`__new__` 方法比 `new` 运算符好

在 Python 中还有一处体现了统一访问原则（或者它的变体）：函数调用和对象实例化使用相同的句法——`my_obj = foo()`，其中 `foo` 是类或其他可调用对象。

其他受 C++ 句法影响的语言提供了 `new` 运算符，致使实例化不像是调用。大多数时候，API 的用户不关心 `foo` 是函数还是类。直到最近，我才意识到，`property` 是一个函数。在常规的用法中，这没什么区别。

把构造方法替换成工厂函数有很多充足的理由。¹⁴ 一个重要的原因是，通过返回之前构建的实例，限制实例的数量（体现了单例模式）。有个相关的功能是，缓存构建过程开销大的对象。此外，有时便于根据提供的参数返回不同类型的对象。

定义构造方法较为简单；提供工厂方法虽然增加了灵活性，但是要编写更多的代码。在有 `new` 运算符的语言中，API 的设计者必须提前决定，究竟是坚持使用简单的构造方法，还是投入工厂方法的怀抱。如果一开始选择错了，那么修正的代价可能很大——这一切都因为 `new` 是运算符。

有时可能更适合走另一条路，把简单的函数换成类。

在 Python 中，很多情况下类和函数可以互换。这不仅是因为 Python 没有 `new` 运算符，还因为有特殊的 `__new__` 方法，可以把类变成工厂函数，生成不同类型的对象（参见 22.2.3 节），或者返回事先构建好的实例，而不是每次都创建一个新实例。

如果“PEP 8—Style Guide for Python Code”没有推荐类名使用驼峰式 (CamelCase)，那么函数与类的对偶性更易于使用。不过，标准库中有很多类的名称是小写的（例如 `property`、`str`、`defaultdict` 等）。因此，使用小写的类名可能是一种特色，而不是 bug。但是，不管怎么看，Python 标准库在类名大小写上的不一致会导致可用性问题。

虽然调用函数与调用类没有区别，但是最好知道哪个是哪个，因为类还有一个功能：子类化。因此，我编写的每个类都使用驼峰式名称，而且希望 Python 标准库中的所有类也使用这一约定。

`collections.OrderedDict` 和
`collections.defaultdict`，我在盯着你们呢。

¹³参见《Python 技术手册（第 2 版）》。

¹⁴我将要提到的原因出自 Jonathan Amsterdam 发布在 *Dr. Dobbs Journal* 中的一篇文章，题为“Java's new Considered Harmful”，以及《Effective Java 中文版（原书第 3 版）》一书中的第一条“考虑用静态工厂方法代替构造函数”。

第 23 章 属性描述符

学会描述符之后，不仅有更多的工具集可用，还能对 Python 的运作方式有更深入的理解，不得不由衷赞叹 Python 设计的优雅。

——Raymond Hettinger
Python 核心开发者和专家¹

¹摘自 Raymond Hettinger 写的“Descriptor HowTo Guide”。

描述符是对多个属性运用相同存取逻辑的一种方式。例如，Django ORM 和 SQLAlchemy 等 ORM 中的字段类型就是描述符，其把数据库记录中字段里的数据与 Python 对象的属性对应了起来。

描述符是实现了动态协议的类，这个协议包括 `__get__` 方法、`__set__` 方法和 `__delete__` 方法。`property` 类实现了完整的描述符协议。通常，动态协议可以部分实现。其实，我们在真实的代码中见到的大多数描述符只实现了 `__get__` 方法和 `__set__` 方法，还有很多只实现了其中一个方法。

描述符是 Python 独有的功能，不仅在应用程序层中使用，在语言的基础设施中也会用到。用户定义的函数就是描述符。我们将看到，描述符协议可以把方法变成绑定方法或非绑定方法，这取决于方法的调用方式。

如果想晋升 Python 高手，则势必要理解描述符。本章将为你向上进阶铺平道路。

本章将重构 22.4 节中的散装食物示例，把特性替换为描述符，方便在不同的类中重用属性验证逻辑。我们将分辨覆盖型描述符和非覆盖型

描述符，并揭示 Python 函数其实就是描述符。最后，本章会给出一些实现描述符的建议。

23.1 本章新增内容

由于 Python 3.6 为描述符协议增加了特殊方法 `__set_name__`，因此 23.2.2 节中的 `Quantity` 描述符示例得以大大简化。

23.2.2 节原有的特性工厂函数示例删除了，因为现在用不到了。我当初使用特性工厂函数是为了展示解决 `Quantity` 问题的另一种方式，而增加 `__set_name__` 方法之后，描述符方案变得简单多了。

23.2.3 节原有的 `AutoStorage` 类也删除了，因为 `__set_name__` 方法的出现把它淘汰了。

23.2 描述符示例：属性验证

如 22.6 节所述，特性工厂函数借助函数式编程模式避免重复编写读值方法和设值方法。特性工厂函数是高阶函数，在闭包中存储 `storage_name` 等设置，由参数决定创建哪些存取函数，再使用存取函数构建自定义的特性实例。解决这种问题的面向对象方式是描述符类。

这里继续 22.6 节中的 `LineItem` 系列示例，把 `quantity` 特性工厂函数重构成 `Quantity` 描述符类，简化用法。

23.2.1 `LineItem` 类第 3 版：一个简单的描述符

本章开篇讲过，实现了 `__get__` 方法、`__set__` 方法或 `__delete__` 方法的类是描述符。描述符的用法是，创建一个实例，作为另一个类的类属性。

本章将定义一个 Quantity 描述符，LineItem 类用到两个 Quantity 实例：一个用于管理 weight 属性，另一个用于管理 price 属性。示意图有助于理解，如图 23-1 所示。

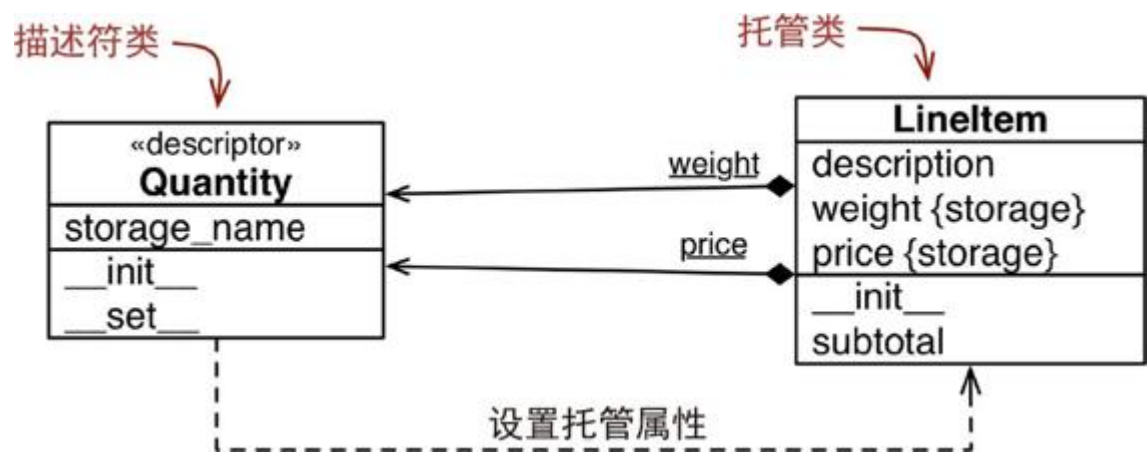


图 23-1：LineItem 的 UML 类图，用到了名为 Quantity 的描述符类。UML 中带下划线的属性是类属性。注意，weight 和 price 是依附在 LineItem 类上的 Quantity 类的实例，不过 LineItem 实例也有自己的 weight 属性和 price 属性，存储着相应的值

注意，在图 23-1 中，“weight”这个词出现了两次，因为其实有两个不同的属性都叫 weight：一个是 LineItem 的类属性，另一个是各个 LineItem 对象的实例属性。price 也是如此。

描述符相关术语

实现和使用描述符涉及多个组件，各个组件的命名务必准确。在说明本章中的示例时，我将使用以下术语和定义。看到具体代码后，对这些术语的理解会更为清晰。这里选择先给出定义是为了方便参考。

描述符类

实现描述符协议的类。图 23-1 中的 Quantity 是描述符类。

托管类

把描述符实例声明为类属性的类。图 23-1 中的 `LineItem` 是托管类。

描述符实例

描述符类的各个实例，声明为托管类的类属性。在图 23-1 中，各个描述符实例使用箭头和带下划线的名称表示（在 UML 中，下划线表示类属性）。与黑色菱形块接触的 `LineItem` 类包含描述符实例。

托管实例

托管类的实例。在这个示例中，`LineItem` 实例是托管实例（未在类图中展示）。

储存属性

托管实例中存储托管属性的属性。在图 23-1 中，`LineItem` 实例的 `weight` 属性和 `price` 属性是储存属性。这种属性与描述符属性不同，后者始终是类属性。

托管属性

托管类中由描述符实例处理的公开属性，值存储在储存属性中。也就是说，描述符实例和储存属性为托管属性建立了基础。

务必要理解，`Quantity` 实例是 `LineItem` 的类属性。图 23-2 中的机器和小怪兽强调了这一点。

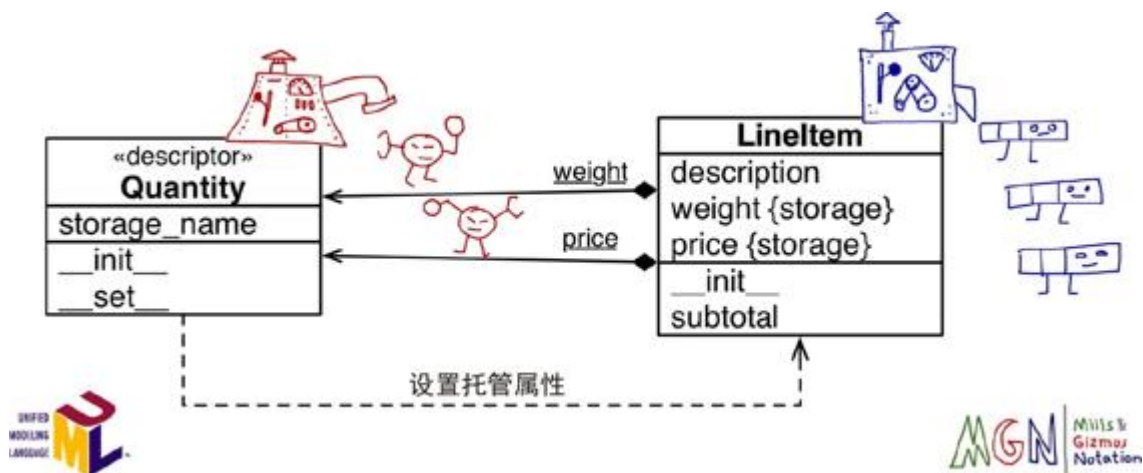


图 23-2：带有 MGN（Mills & Gizmos Notation，机器和小怪兽图示法）注解的 UML 类图：类是机器，用于生产小怪兽（实例）。Quantity 机器生产了两个圆头小怪兽，依附在 LineItem 机器上，即 weight 和 price。LineItem 机器生产的是方头小怪兽，有自己的 weight 属性和 price 属性，存储着相应的值

机器和小怪兽图示法介绍

我以前经常使用 UML 解说描述符，但是后来发现 UML 无法很好地展现类与实例之间的关系，例如托管类与描述符实例之间的关系。² 所以，我自己发明了一门“语言”——机器和小怪兽图示法（Mills & Gizmos Notation, MGN），使用它注解 UML 图。

MGN 的目的是明确区分类和实例，如图 23-3 所示。在 MGN 中，类被画成了“机器”，这是一种复杂的设备，用于生产小怪兽。类（机器）是有操控杆和刻度盘的设备。小怪兽是实例，外观更简洁，与生产它的机器具有相同的颜色。



图 23-3: MGN 简图表示, `LineItem` 类生产了 3 个实例, `Quantity` 类生产了两个实例。其中一个 `Quantity` 实例从一个 `LineItem` 实例中获取存储的值

在这个示例中, 我把 `LineItem` 实例画成了表格中的行, 各有 3 个单元格, 表示 3 个属性 (`description`、`weight` 和 `price`)。 `Quantity` 实例是描述符, 因此有个放大镜, 用于获取值 (`__get__`), 还有一个手抓, 用于设置值 (`__set__`)。讲到元类时, 你会感谢我画了这些涂鸦。

²在 UML 类图中, 类和实例都被画成了方框。虽然视觉上有区别, 但是类图中很少出现实例, 开发者可能认不出。

先把涂鸦放在一边, 来看代码: 示例 23-1 是 `Quantity` 描述符类; 示例 23-2 给出了新版 `LineItem` 类, 其中用到两个 `Quantity` 实例。

示例 23-1 `bulkfood_v3.py`: `Quantity` 描述符不接受负值

```
class Quantity: ❶

    def __init__(self, storage_name):
        self.storage_name = storage_name ❷

    def __set__(self, instance, value): ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value ❹
        else:
```

```
msg = f'{self.storage_name} must be > 0'
raise ValueError(msg)

def __get__(self, instance, owner): ❸
    return instance.__dict__[self.storage_name]
```

❶ 描述符基于协议实现，无须子类化。

❷ `Quantity` 实例有一个 `storage_name` 属性，这是托管实例中用于存储值的储存属性的名称。

❸ 尝试为托管属性赋值时，调用 `__set__` 方法。这里，`self` 是描述符实例（`LineItem.weight` 或 `LineItem.price`），`instance` 是托管实例（`LineItem` 实例），`value` 是要设定的值。

❹ 必须把属性的值直接存入 `__dict__`。调用 `setattr(instance, self.storage_name)` 将再次触发 `__set__` 方法，导致无限递归。

❺ 需要实现 `__get__` 方法，因为托管属性的名称可能与 `storage_name` 不同。`owner` 参数稍后解释。

`__get__` 方法有必要实现，因为用户可能编写如下代码。

```
class House:
    rooms = Quantity('number_of_rooms')
```

在这个 `House` 类中，托管属性是 `rooms`，而储存属性是 `number_of_rooms`。对于一个名为 `chaos_manor` 的 `House` 实例，读写 `chaos_manor.rooms` 都经过依附在 `rooms` 上的 `Quantity` 描述符，但是读写 `chaos_manor.number_of_rooms` 会绕过该描述符。

注意，`__get__` 方法接受 3 个参数：`self`、`instance` 和 `owner`。`owner` 参数是对托管类（例如 `LineItem`）的引用，在希望描述符支

持获取类属性时会用到——比如说模拟 Python 获取类属性，但是在实例中未找到指定名称的属性时的默认行为。

如果通过类获取托管属性（例如 `LineItem.weight`），那么描述符的 `__get__` 方法收到的 `instance` 参数值为 `None`。

为了支持内省和其他元编程技巧，当通过类存取托管属性时，`__get__` 方法最好返回描述符实例。为此，应像下面这样编写 `__get__` 方法。

```
def __get__(self, instance, owner):
    if instance is None:
        return self
    else:
        return instance.__dict__[self.storage_name]
```

示例 23-2 在 `LineItem` 中使用了 `Quantity`。

示例 23-2 `bulkfood_v3.py`: 在 `LineItem` 中使用 `Quantity` 描述符管理属性

```
class LineItem:
    weight = Quantity('weight') ❶
    price = Quantity('price') ❷

    def __init__(self, description, weight, price): ❸
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ 第一个描述符实例管理 `weight` 属性。

❷ 第二个描述符实例管理 `price` 属性。

❸ 类主体中余下的代码与 `bulkfood_v1.py` 脚本（参见示例 22-19）中的代码一样简洁。

示例 23-2 中的代码能像预期那样运作，禁止以 0 美元销售松露。³

³1 磅（1 磅≈ 0.45 千克）白松露价值几千美元。以下练习留给有进取心的读者：不准以 0.01 美元销售松露。我认识一个人，他以 18 美元买到了价值 1800 美元的统计学百科全书，因为那个网店（不是亚马逊）有漏洞。

```
>>> truffle = LineItem('White truffle', 100, 0)
Traceback (most recent call last):
...
ValueError: value must be > 0
```



编写描述符的 `__get__` 方法和 `__set__` 方法时，要记住 `self` 参数和 `instance` 参数的意思：`self` 是描述符实例，`instance` 是托管实例。管理实例属性的描述符应该把值存储在托管实例中。因此，Python 才为描述符中的方法提供了 `instance` 参数。

你可能想把各个托管属性的值直接存在描述符实例中，但这种做法是错误的。也就是说，在 `__set__` 方法中，应该像下面这样写：

```
instance.__dict__[self.storage_name] = value
```

而不能试图使用下面这种错误的写法：

```
self.__dict__[self.storage_name] = value
```

为了理解出错的原因，可以想想 `__set__` 方法前两个参数（`self` 和 `instance`）的意思。这里，`self` 是描述符实例，它其实是托管类的类属性。同一时刻，内存中可能有几千个 `LineItem` 实例，不过只会有两个描述符实例，即类属性 `LineItem.weight` 和 `LineItem.price`。因此，存储在描述符实例中的数据其实会变成 `LineItem` 的类属性，从而由全部 `LineItem` 实例共享。

示例 23-2 有个缺点，即在托管类的主体中实例化描述符时要重复输入属性的名称。如果 `LineItem` 类能像下面这样声明就好了。

```
class LineItem:
    weight = Quantity()
    price = Quantity()

    # 余下的方法不变
```

对此，示例 23-2 必须明确指明各个 `Quantity` 实例的名称。这样不仅麻烦，还很危险：如果程序员直接复制并粘贴代码，忘了编辑名称，比如写成 `price = Quantity('weight')`，那么程序的行为将大错特错，设置 `price` 的值时会覆盖 `weight` 的值。

可问题是，正如第 6 章所述，赋值语句右侧的表达式会先执行，而此时变量还不存在。`Quantity()` 表达式的求值结果是创建描述符实例，而此时 `Quantity` 类中的代码无法猜出要把描述符绑定给哪个变量（例如，`weight` 或 `price`）。

幸好，描述符协议现在支持名称贴切的特殊方法 `__set_name__`。接下来会介绍这个方法的用途。



为描述符的储存属性自动命名曾经是一个棘手的问题。在本书第 1 版中，本章和第 24 章用了好几页篇幅和多行代码探讨了不同的解决方案，涉及类装饰器和元类（参见第 24 章）。这个问题在 Python 3.6 中得到了极大的简化。

23.2.2 `LineItem` 类第 4 版：为储存属性自动命名

为了避免在描述符实例中重复输入属性名，我们将实现 `__set_name__` 方法，设置各个 `Quantity` 实例的 `storage_name`。特殊方法 `__set_name__` 在 Python 3.6 中加入了描述符协议。解释器会在 `class` 主体中找到的每个描述符上调用 `__set_name__` 方法，当然前提是描述符实现了该方法。⁴

⁴更准确地说，调用 `__set_name__` 的是 `type.__new__`——表示类的对象的构造方法。内置的 `type` 其实是一个元类，是用户定义的类的默认类。这一点一开始很难理解，但请放

心，第 24 章会专门讨论类的动态配置，包括元类概念。

在示例 23-3 中，`LineItem` 的描述符类不需要 `__init__` 方法了，`__set_name__` 方法负责保存储存属性的名称。

示例 23-3 `bulkfood_v4.py`: `__set_name__` 设置各个 `Quantity` 描述符实例的名称

```
class Quantity:

    def __set_name__(self, owner, name): ❶
        self.storage_name = name ❷

    def __set__(self, instance, value): ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value
        else:
            msg = f'{self.storage_name} must be > 0'
            raise ValueError(msg)

    # 不需要__get__ ❹

class LineItem:

    weight = Quantity() ❺
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ `self` 是描述符实例（不是托管实例），`owner` 是托管类，`name` 是在 `owner` 的类主体中把描述符实例赋给的那个属性的名称。

❷ 与示例 23-1 中 `__init__` 方法的做法一样。

❸ 这里的 `__set__` 方法与示例 23-1 中一模一样。

❷ 不需要实现 `__get__` 方法，因为储存属性的名称与托管属性的名称一致。表达式 `product.price` 直接从 `LineItem` 实例中获取 `price` 属性。

❸ 现在，不用把托管属性的名称传给 `Quantity` 构造函数。这正是这一版的目标。

看着示例 23-3，你或许会想，仅仅为了管理两个属性，有必要编写这么多代码吗？但是，要知道，现在描述符的逻辑抽象到单独的代码单元中了：`Quantity` 类。通常，我们不在使用描述符的模块中定义描述符，而是在一个单独的实用工具模块中定义，以方便在整个应用程序中重用——如果是在开发库或框架，那么甚至可以在多个应用程序中使用。

了解这一点之后即可推知，示例 23-4 是描述符的常规用法。

示例 23-4 `bulkfood_v4c.py`：整洁的 `LineItem` 类，`Quantity` 描述符类现在位于导入的 `model_v4c` 模块中

```
import model_v4c as model ❶

class LineItem:
    weight = model.Quantity() ❷
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ 导入实现 `Quantity` 的 `model_v4c` 模块。

❷ 使用 `model.Quantity` 描述符。

Django 用户会发现，示例 23-4 非常像模型定义。这不是巧合：Django 模型的字段就是描述符。

由于描述符通过类实现，因此可以利用继承重用部分代码来创建新描述符。详见 23.2.3 节。

23.2.3 LineItem 类第 5 版：一种新型描述符

我们虚构的有机食物网店遇到一个问题：不知怎么回事儿，有个商品的描述信息为空，导致无法下单。为了避免出现这个问题，我们将再创建一个描述符 `NonBlank`。在设计 `NonBlank` 的过程中，我们发现它与 `Quantity` 描述符很像，只是验证逻辑不同。

据此，我们判断需要重构，定义一个抽象类 `Validated`，覆盖 `__set__` 方法，调用必须由子类实现的 `validate` 方法。

然后，重写 `Quantity` 类，通过继承 `Validated` 类并只编写 `validate` 方法来实现 `NonBlank` 类。

`Validated`、`Quantity` 和 `NonBlank` 这 3 个类之间的关系体现了《设计模式》一书中提出的**模板方法**（template method）。

模板方法用一些抽象的操作定义算法，而子类将重定义这些操作以提供具体的行为。

在示例 23-5 中，`Validated.__set__` 是模板方法，`self.validate` 是抽象操作。

示例 23-5 model_v5.py：抽象基类 `Validated`

```
import abc

class Validated(abc.ABC):

    def __set_name__(self, owner, name):
        self.storage_name = name
```

```

def __set__(self, instance, value):
    value = self.validate(self.storage_name, value) ❶
    instance.__dict__[self.storage_name] = value ❷

@abc.abstractmethod
def validate(self, name, value): ❸
    """返回通过验证的值，或者抛出ValueError"""

```

❶ `__set__` 方法把验证操作委托给 `validate` 方法.....

❷ 然后使用返回的 `value` 更新存储的值。

❸ `validate` 是一个抽象方法，即模板方法。

Alex Martelli 喜欢把这个设计模式称为**自委托**（Self-Delegation），我也觉得这个名称更具描述性：`__set__` 方法的第 1 行把职责委托给自身的 `validate` 方法。⁵

⁵Alex Martelli 的题为“Python Design Patterns”的演讲，第 50 张幻灯片。强烈建议观看。

在这个示例中，`Quantity` 和 `NonBlank` 是 `Validated` 的具体子类，如示例 23-6 所示。

示例 23-6 `model_v5.py`: `Validated` 的具体子类 `Quantity` 和 `NonBlank`

```

class Quantity(Validated):
    """数值要大于零"""

    def validate(self, name, value): ❶
        if value <= 0:
            raise ValueError(f'{name} must be > 0')
        return value

class NonBlank(Validated):
    """字符串至少要包含一个非空字符"""

    def validate(self, name, value):

```

```
value = value.strip()
if not value: ❷
    raise ValueError(f'{name} cannot be blank')
return value ❸
```

❶ 实现抽象方法 `Validated.validate` 要求的模板方法。

❷ 去除头尾的空白后，如果什么也没剩下，则拒绝提供的值。

❸ 具体的 `validate` 方法必须返回通过验证的值，防止后续需要清理、转换或规范化接收到的数据。这里，返回的是去除头尾的空白之后的 `value`。

`model_v5.py` 脚本的用户不需要知道全部细节。用户只需知道，可以使用 `Quantity` 和 `NonBlank` 自动验证实例属性。详见示例 23-7 中的最新版 `LineItem` 类。

示例 23-7 `bulkfood_v5.py`: 使用描述符 `Quantity` 和 `NonBlank` 的 `LineItem` 类

```
import model_v5 as model ❶

class LineItem:
    description = model.NonBlank() ❷
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ 导入 `model_v5` 模块，指定一个更友好的名称。

❷ 使用 `model.NonBlank` 描述符，其余的代码没变。

本章所举的几个 `LineItem` 示例演示了描述符的典型用途，即管理数据属性。`Quantity` 这种描述符叫作覆盖型描述符，因为描述符的 `__set__` 方法使用托管实例中的同名属性覆盖（插手接管）了要设置的属性。除此之外，还有非覆盖型描述符。23.3 节将详述两种描述符之间的区别。

23.3 覆盖型描述符与非覆盖型描述符对比

如前所述，Python 处理属性的方式特别不对等。通过实例读取属性时，通常返回的是实例中定义的属性。但是，如果实例中没有指定的属性，则会获取类属性。而为实例中的属性赋值时，往往会在实例中创建属性，根本不影响类。

这种不对等的处理方式对描述符也有影响。其实，根据是否实现 `__set__` 方法，描述符可分为两大类。实现 `__set__` 方法的类是覆盖型描述符，未实现 `__set__` 方法的类则是非覆盖型描述符。分析以下几个示例之后，你将对这两个术语有深入的理解。

为了观察这两类描述符的差异，需要用到几个类。我们将使用示例 23-8 中的代码作为接下来几节的试验台。



在示例 23-8 中，每个 `__get__` 方法和 `__set__` 方法都调用了 `print_args` 函数，目的是以一种更易读的方式显示调用过程。没必要理解 `print_args` 函数及辅助函数 `cls_name` 和 `display`，不要花心思研究它们。

示例 23-8 `descriptorkinds.py`：用于研究描述符覆盖行为的几个简单的类

```
### 辅助函数，仅用于显示 ###
```

```

def cls_name(obj_or_cls):
    cls = type(obj_or_cls)
    if cls is type:
        cls = obj_or_cls
    return cls.__name__.split('.')[-1]

def display(obj):
    cls = type(obj)
    if cls is type:
        return f'<class {obj.__name__}>'
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return f'<{cls_name(obj)} object>'

def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
    print(f'-> {cls_name(args[0])}.__{name}__({pseudo_args})')

```

对这个示例重要的类

```

class Overriding: ❶
    """也叫数据描述符或强制描述符"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner) ❷

    def __set__(self, instance, value):
        print_args('set', self, instance, value)

```

```

class OverridingNoGet: ❸
    """没有`__get__`方法的覆盖型描述符"""

    def __set__(self, instance, value):
        print_args('set', self, instance, value)

```

```

class NonOverriding: ❹
    """也叫非数据描述符或遮盖型描述符"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)

```

```

class Managed: ❺
    over = Overriding()
    over_no_get = OverridingNoGet()

```

```
non_over = NonOverriding()

def spam(self): ❹
    print(f'-> Managed.spam({display(self)})')
```

- ❶ 有 `__get__` 方法和 `__set__` 方法的覆盖型描述符。
- ❷ 在这个示例中，各个描述符方法都调用了 `print_args` 函数。
- ❸ 没有 `__get__` 方法的覆盖型描述符。
- ❹ 没有 `__set__` 方法，所以这是一个非覆盖型描述符。
- ❺ 托管类，使用各个描述符类的一个实例。
- ❻ 将 `spam` 方法放在这里是为了对比，因为方法也是描述符。

接下来的 23.3.1 节、23.3.2 节和 23.3.3 节将分析对 `Managed` 类及其实例做属性读写时的行为，还会讨论这里定义的几个描述符。

23.3.1 覆盖型描述符

实现 `__set__` 方法的描述符属于**覆盖型描述符**，因为虽然描述符是类属性，但是实现 `__set__` 方法的话，描述符将覆盖对实例属性的赋值操作。示例 23-3 就是这样实现的。特性也是覆盖型描述符：如果没有提供设值函数，那么 `property` 类中的 `__set__` 方法就会抛出 `AttributeError` 异常，表明那个属性是只读的。可以使用示例 23-8 中的代码测试覆盖型描述符的行为，如示例 23-9 所示。

示例 23-9 一个覆盖型描述符的行为

```
>>> obj = Managed() ❶
>>> obj.over ❷
-> Overriding.__get__(<Overriding object>, <Managed object>,
<class Managed>)
>>> Managed.over ❸
-> Overriding.__get__(<Overriding object>, None, <class Managed>)
```

```
>>> obj.over = 7 ❹
-> Overriding.__set__(<Overriding object>, <Managed object>, 7)
>>> obj.over ❺
-> Overriding.__get__(<Overriding object>, <Managed object>,
<class Managed>)
>>> obj.__dict__['over'] = 8 ❻
>>> vars(obj) ❼
{'over': 8}
>>> obj.over ❽
-> Overriding.__get__(<Overriding object>, <Managed object>,
<class Managed>)
```

- ❶ 创建 Managed 对象，供测试使用。
- ❷ obj.over 触发描述符的 __get__ 方法，传入的第二个参数是托管实例 obj。
- ❸ Managed.over 触发描述符的 __get__ 方法，传入的第二个参数 (instance) 是 None。
- ❹ 为 obj.over 赋值，触发描述符的 __set__ 方法，传入的最后一个参数是 7。
- ❺ 读取 obj.over 仍会调用描述符的 __get__ 方法。
- ❻ 绕过描述符，直接通过 obj.__dict__ 属性设置。
- ❼ 确认值在 obj.__dict__ 属性中，在 over 键名下。
- ❽ 然而，即使是名为 over 的实例属性，Managed.over 描述符仍会覆盖读取 obj.over 操作。



讨论这些概念时，不同的 Python 贡献者和作者会使用不同的术语。“覆盖型描述符”是我从 *Python in a Nutshell* 一书中学到的说法。Python 官方文档使用“数据描述符”，不过“覆盖型描述

符”更能凸显它的特殊行为。覆盖型描述符也叫“强制描述符”。非覆盖型描述符也叫“非数据描述符”或“遮盖型描述符”。

23.3.2 没有 `__get__` 方法的覆盖型描述符

特性和其他覆盖型描述符（例如 Django 模型字段）既实现 `__set__` 方法，也实现 `__get__` 方法，不过也可以只实现 `__set__` 方法，如示例 23-2 所示。此时，只有写操作由描述符处理。通过实例读取描述符会返回描述符对象本身，因为没有处理读操作的 `__get__` 方法。如果直接通过实例的 `__dict__` 属性创建同名实例属性，那么以后再设置那个属性时，仍由 `__set__` 方法插手接管，但是读取那个属性的话，会直接从实例中返回新赋予的值，而不返回描述符对象。也就是说，实例属性会遮盖描述符，不过只有读操作是如此。如示例 23-10 所示。

示例 23-10 没有 `__get__` 方法的覆盖型描述符

```
>>> obj.over_no_get ❶
<__main__.OverridingNoGet object at 0x665bcc>
>>> Managed.over_no_get ❷
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.over_no_get = 7 ❸
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed
object>, 7)
>>> obj.over_no_get ❹
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.__dict__['over_no_get'] = 9 ❺
>>> obj.over_no_get ❻
9
>>> obj.over_no_get = 7 ❼
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed
object>, 7)
>>> obj.over_no_get ❽
9
```

❶ 这个覆盖型描述符没有 `__get__` 方法，因此，`obj.over_no_get` 从类中获取描述符实例。

- ❷ 直接从托管类中读取描述符实例也是如此。
- ❸ 为 `obj.over_no_get` 赋值，调用描述符的 `__set__` 方法。
- ❹ 因为 `__set__` 方法没有修改属性，所以再次读取 `obj.over_no_get` 获取的仍是托管类中的描述符实例。
- ❺ 通过实例的 `__dict__` 属性设置名为 `over_no_get` 的实例属性。
- ❻ 现在，`over_no_get` 实例属性遮盖描述符，但是只有读操作如此。
- ❼ 为 `obj.over_no_get` 赋值，仍然通过描述符的 `__set__` 方法处理。
- ❽ 但是读取时，只要有同名的实例属性，描述符就会被遮盖。

23.3.3 非覆盖型描述符

没有实现 `__set__` 方法的描述符是非覆盖型描述符。如果设置了同名的实例属性，那么描述符就会被遮盖，致使其无法处理那个实例的那个属性。所有方法和 `@functools.cached_property` 是以非覆盖型描述符实现的。示例 23-11 展示了对一个非覆盖型描述符的操作。

示例 23-11 一个非覆盖型描述符的行为

```
>>> obj = Managed()
>>> obj.non_over ❶
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>,
<class Managed>)
>>> obj.non_over = 7 ❷
>>> obj.non_over ❸
7
>>> Managed.non_over ❹
```

```
-> NonOverriding.__get__(<NonOverriding object>, None, <class
Managed>)
>>> del obj.non_over ❸
>>> obj.non_over ❹
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>,
<class Managed>)
```

❶ `obj.non_over` 触发描述符的 `__get__` 方法，传入的第二个参数是 `obj`。

❷ `Managed.non_over` 是非覆盖型描述符，没有干涉赋值操作的 `__set__` 方法。

❸ 现在，`obj` 有个名为 `non_over` 的实例属性，会遮盖 `Managed` 类的同名描述符属性。

❹ `Managed.non_over` 描述符依然存在，会通过类截获这次访问。

❺ 如果把 `non_over` 实例属性删除了.....

❻ 那么读取 `obj.non_over` 时会触发类中描述符的 `__get__` 方法。但要注意，第二个参数的值是托管实例。

在上述几个示例中，我们为几个与描述符同名的实例属性赋了值，结果依描述符是否有 `__set__` 方法而有所不同。

依附在类上的描述符无法控制为类属性赋值的操作。其实，这意味着为类属性赋值能覆盖描述符属性，详见 23.3.4 节。

23.3.4 覆盖类中的描述符

不管描述符是不是覆盖型，为类属性赋值都能覆盖描述符。这是一种猴子补丁技术，不过示例 23-12 把描述符替换成了整数，这其实会导致依赖描述符的类不能正常执行操作。

示例 23-12 通过类可以覆盖任何描述符

```
>>> obj = Managed() ❶
>>> Managed.over = 1 ❷
>>> Managed.over_no_get = 2
>>> Managed.non_over = 3
>>> obj.over, obj.over_no_get, obj.non_over ❸
(1, 2, 3)
```

❶ 为后面的测试新建一个实例。

❷ 覆盖类中的描述符属性。

❸ 描述符真的不见了。

示例 23-12 揭示了读写属性的另一种不对等：类属性读取操作可以由依附在托管类上定义有 `__get__` 方法的描述符处理，但是类属性写入操作不由依附在托管类上定义有 `__set__` 方法的描述符处理。



要想控制设置类属性的操作，需要把描述符依附在类的类上，即依附在元类上。默认情况下，对于用户定义的类，元类是 `type`，不能向 `type` 添加属性。我们将在第 24 章自己创建元类。

下面调转话题，探讨 Python 是如何使用描述符实现方法的。

23.4 方法是描述符

在类中定义的函数，如果在实例上调用，就会变成绑定方法，因为用户定义的函数都有 `__get__` 方法，在依附到类上后，就相当于描述符。示例 23-13 演示了从示例 23-8 定义的 `Managed` 类中读取 `spam` 方法。

示例 23-13 方法是非覆盖型描述符

```
>>> obj = Managed()
>>> obj.spam ❶
```

```
<bound method Managed.spam of <descriptorkinds.Managed object at
0x74c80c>>
>>> Managed.spam ❷
<function Managed.spam at 0x734734>
>>> obj.spam = 7 ❸
>>> obj.spam
7
```

❶ `obj.spam` 获取一个绑定方法对象。

❷ 但是 `Managed.spam` 获取的是一个函数。

❸ 为 `obj.spam` 赋值，遮盖类属性，导致无法通过 `obj` 实例访问 `spam` 方法。

函数没有实现 `__set__` 方法，因此是非覆盖型描述符，如示例 23-13 中的最后一行所示。

从示例 23-13 中还可以看出一个重要信息：`obj.spam` 和 `Managed.spam` 获取的是不同的对象。与描述符一样，通过托管类访问时，函数的 `__get__` 方法返回自身的引用。但是，通过实例访问时，函数的 `__get__` 方法返回的是绑定方法对象：一种可调用对象，里面包装着函数，并把托管实例（例如 `obj`）绑定给函数的第一个参数（`self`），这与 `functools.partial` 函数的行为一致（参见 7.8.2 节）。为了深入理解这种机制，请看示例 23-14。

示例 23-14 `method_is_descriptor.py`: `Text` 类，衍生自 `UserString` 类

```
import collections

class Text(collections.UserString):

    def __repr__(self):
        return 'Text({!r})'.format(self.data)

    def reverse(self):
        return self[::-1]
```

下面来分析 `Text.reverse` 方法，如示例 23-15 所示。

示例 23-15 测试一个方法

```
>>> word = Text('forward')
>>> word ❶
Text('forward')
>>> word.reverse() ❷
Text('drawrof')
>>> Text.reverse(Text('backward')) ❸
Text('drawkcab')
>>> type(Text.reverse), type(word.reverse) ❹
(<class 'function'>, <class 'method'>)
>>> list(map(Text.reverse, ['repaid', (10, 20, 30),
Text('stressed')])) ❺
['diaper', (30, 20, 10), Text('desserts')]
>>> Text.reverse.__get__(word) ❻
<bound method Text.reverse of Text('forward')>
>>> Text.reverse.__get__(None, Text) ❼
<function Text.reverse at 0x101244e18>
>>> word.reverse ❽
<bound method Text.reverse of Text('forward')>
>>> word.reverse.__self__ ❾
Text('forward')
>>> word.reverse.__func__ is Text.reverse ❿
True
```

❶ `Text` 实例的字符串表示形式是一个类似 `Text` 构造函数调用的字符串，可用于创建相同的实例。

❷ `reverse` 方法返回反向拼写的单词。

❸ 在类上调用方法相当于调用函数。

❹ 注意类型是不同的，一个是 `function`，一个是 `method`。

❺ `Text.reverse` 相当于函数，甚至可以处理 `Text` 实例之外的其他对象。

- ❹ 函数都是非覆盖型描述符。如果在函数上调用 `__get__` 方法时传入实例，则得到的是绑定到那个实例上的方法。
- ❺ 调用函数的 `__get__` 方法时，如果 `instance` 参数的值是 `None`，那么得到的是函数本身。
- ❻ `word.reverse` 表达式其实会调用 `Text.reverse.__get__(word)`，返回对应的绑定方法。
- ❼ 绑定方法对象有一个 `__self__` 属性，其值是调用该方法的实例引用。
- ❽ 绑定方法的 `__func__` 属性是依附在托管类上那个原始函数的引用。

绑定方法对象还有 `__call__` 方法，用于处理实际调用过程。这个方法会调用 `__func__` 属性引用的原始函数，传入的第一个参数是方法的 `__self__` 属性。这就是形参 `self` 的隐式绑定方式。

函数会变成绑定方法，这是 Python 语言底层使用描述符的最好例证。

深入了解描述符和方法的运作方式之后，下面讨论用法方面的一些实用建议。

23.5 描述符用法建议

下面根据刚刚探讨的描述符特征给出一些实用的结论。

使用 `property` 以保持简单

内置的 `property` 类创建的其实是实现了 `__set__` 方法和 `__get__` 方法的覆盖型描述符，即使没有定义设值方法。⁶ 特性的

`__set__` 方法会默认抛出 `AttributeError: can't set attribute`，因此创建只读属性最简单的方式是使用特性，这能避免下一条所述的问题。

⁶`property` 装饰器还提供了 `__delete__` 方法，即使没有定义删值方法。

只读描述符必须有 `__set__` 方法

使用描述符类实现只读属性时要记住，`__get__` 和 `__set__` 这两个方法必须都定义，否则，实例的同名属性会遮盖描述符。只读属性的 `__set__` 方法只需抛出 `AttributeError` 异常，并提供合适的错误消息。⁷

⁷Python 为此类异常提供的错误消息不一致。如果试图修改 `complex` 数值的 `c.real` 属性，那么得到的错误消息是 `AttributeError: readonly attribute`。但是，如果通过 `c.conjugate` (`complex` 对象的方法) 修改，则得到的错误消息是 `AttributeError: 'complex' object attribute 'conjugate' is read-only`。连“read-only”的拼写方式都不一样。

用于验证的描述符可以只有 `__set__` 方法

在仅用于验证的描述符中，`__set__` 方法应该检查 `value` 参数获得的值，如果有效，就使用描述符实例的名称作为键，直接在实例的 `__dict__` 属性中设置。这样，从实例中读取同名属性的速度很快，因为不用经过 `__get__` 方法处理。参见示例 23-3 中的代码。

仅有 `__get__` 方法的描述符可以实现高效缓存

如果只编写了 `__get__` 方法，那么得到的是非覆盖型描述符。这种描述符可用于执行某些耗费资源的计算，然后为实例设置同名属性，缓存结果。⁸ 同名实例属性会遮盖描述符，因此后续访问直接从实例的 `__dict__` 属性中获取值，不再触发描述符的 `__get__` 方法。`@functools.cached_property` 装饰器创建的其实就是非覆盖型描述符。

⁸然而，前面讲过，在 `__init__` 方法运行之后创建实例属性违背键共享内存优化（详见 3.9 节）。

非特殊的方法可以被实例属性遮盖

函数和方法只实现了 `__get__` 方法，属于非覆盖型描述符。像 `my_obj.the_method = 7` 这样简单赋值之后，后续通过该实例访问 `the_method`，得到的是数值 7——但是不影响类或其他实例。然而，特殊方法不受这个问题的影响。解释器只在类中寻找特殊方法，也就是说，`repr(x)` 执行的其实是 `x.__class__.__repr__(x)`，因此 `x` 的 `__repr__` 属性对 `repr(x)` 方法调用没有影响。出于同样的原因，实例的 `__getattr__` 属性不会破坏常规的属性访问规则。

实例的非特殊方法可以轻易被覆盖，这听起来不太可靠且容易出错，可是在我使用 Python 的 20 多年间从未受此困扰。然而，如果要创建大量动态属性，且属性名称从不受自己控制的数据中获取（像本章前面那样），那么就应该知道这种行为。或许可以实现某种机制，筛选或转义动态属性的名称，以维持数据的健全性。



示例 22-5 中的 `FrozenJSON` 类不会出现实例属性遮盖方法的问题，因为那个类只有几个特殊方法和一个 `build` 类方法。只要通过类访问，类方法就是安全的，在示例 22-5 中我就是这么调用 `FrozenJSON.build` 方法的，在示例 22-6 中又换成了 `__new__` 方法。22.3 节中的 `Record` 类和 `Event` 类也是安全的，因为二者只实现了特殊方法、静态方法和特性。特性是覆盖型描述符，不能被实例属性遮盖。

在结束本章之前，我们来讲讲这里讨论的描述符还未涉及的在讨论特性时讲的两个功能：文档和对删除托管属性的处理。

23.6 描述符的文档字符串和覆盖删除操作

描述符类的文档字符串也用作托管类中各个描述符实例的文档。图 23-4 中的截图是示例 23-6 和示例 23-7 中带有描述符 `Quantity` 和 `NonBlank` 的 `LineItem` 类的帮助界面。

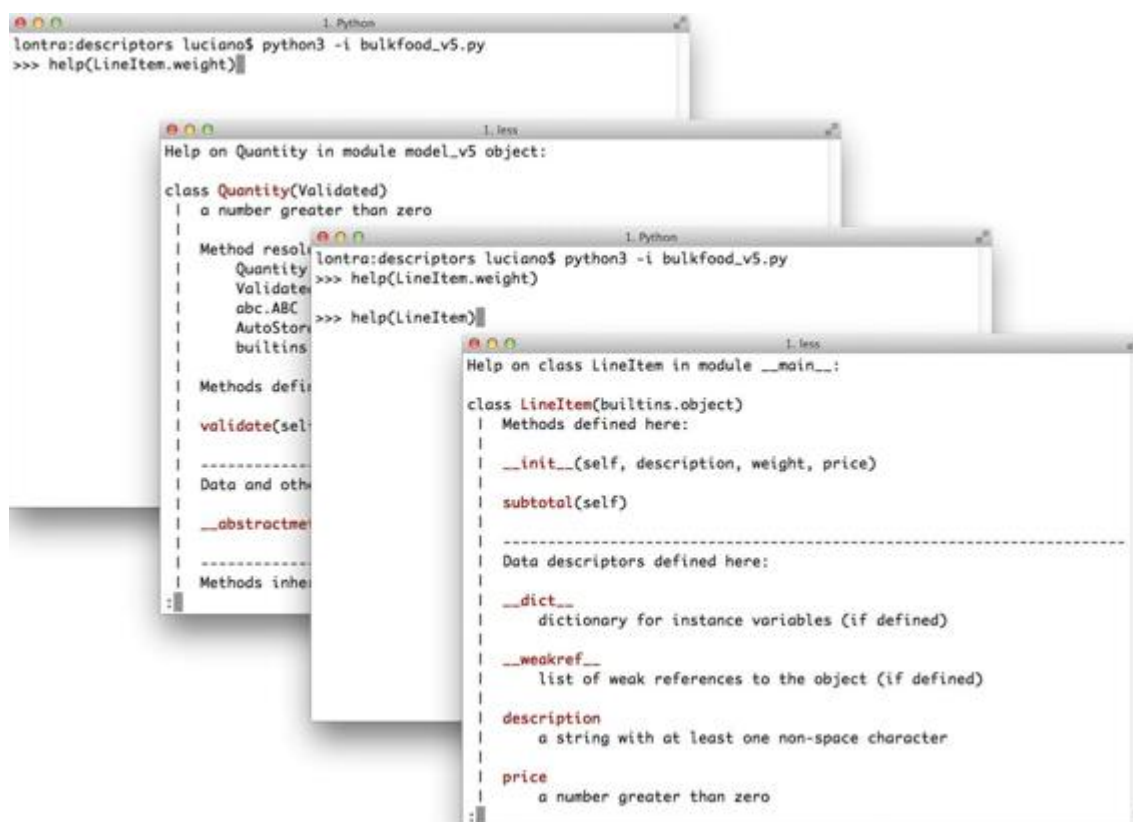


图 23-4: 在 Python 控制台中执行 `help(LineItem.weight)` 命令和 `help(LineItem)` 命令时的界面截图

提供的信息有点儿不足。对 `LineItem` 类来说，如果能说明 `weight` 必须以千克为单位就好了。这对特性来说是“小菜一碟”，因为各个特性只处理特定的托管属性。可是对描述符来说，`weight` 和 `price` 使用的都是 `Quantity` 描述符类。⁹

⁹定制各个描述符实例的帮助文本特别难。有一种方法是为各个描述符实例动态构建包装类。

我们在讨论特性时还讲了一个细节，而本章讨论的描述符没有涉及，那就是对删除托管属性的处理。在描述符类中，除了实现常规的 `__get__` 方法和 `__set__` 方法，还可以实现 `__delete__` 方法，或者只实现 `__delete__` 方法，处理删除托管属性操作。我故意没有讲解 `__delete__` 方法，因为觉得现实中很少用到。如果需要使用，请阅读 Python 数据模型文档中的“Implementing Descriptors”一节。时间充足的读者可以编写一个没有实际作用描述符类来实现 `__delete__` 方法，以此当作练习。

23.7 本章小结

本章的第一个示例接续第 22 章的 `LineItem` 系列示例。在示例 23-2 中，我们把特性替换成了描述符。我们了解到，描述符类的实例能用作托管类的属性。为了讨论这个机制，本章引入了几个特殊的术语，例如**托管实例**和**储存属性**。

在 23.2.2 节中，我们把声明 `Quantity` 描述符所需的 `storage_name` 参数去掉了，因为那个参数多余且容易出错。采用的新方案是在 `Quantity` 中实现特殊方法 `__set_name__`，把托管特性的名称保存为 `self.storage_name`。

23.2.3 节讲解了如何通过子类化抽象描述符类以共享代码，同时构建具有部分共通功能的专用描述符。

然后，本章分析了有或没有 `__set__` 方法时，描述符的行为有何不同，了解了覆盖型描述符和非覆盖型描述符之间的重要差异。通过详细的测试，我们揭示了描述符何时接管，以及何时被遮盖、被绕过或被覆盖。

接下来，本章分析了非覆盖型描述符的一种具体类型：方法。通过控制台中的测试可知，通过实例访问依附在类上的函数时，经由描述符协议的处理，这个函数变成了方法。

最后，23.5 节对描述符的用法给出了一些实用建议，23.6 节还简要说明了如何为描述符添加文档。



23.1 节讲过，在 Python 3.6 为描述符协议添加特殊方法 `__set_name__` 之后，本章的几个示例简化了很多。这是语言进化的好处。

23.8 延伸阅读

除了《Python 语言参考手册》中必读的第 3 章“数据模型”，Raymond Hettinger 写的“Descriptor HowTo Guide”也值得一读——这是 Python 官方文档 HowTo 合集中的一篇。

对 Python 对象模型相关的话题来说，*Python in a Nutshell, 3rd ed.* (Martelli、Ravenscroft 和 Holden 著) 一书仍是权威资料，客观公正。Martelli 还做了一次题为“Python's Object Model”的演讲，深入探讨了特性和描述符。



注意，在 2016 年采纳 PEP 487 之前，对描述符的讲解中用到的示例有可能比现在复杂，因为在 3.6 版之前，Python 不支持 `__set_name__` 方法。

如果想学习更具实践意义的示例，可以阅读《Python Cookbook (第 3 版) 中文版》。该书中有许多演示描述符用法的经典实例，推荐阅读 6.12 节“读取嵌套型和大小可变的二进制结构”、8.10 节“让属性具有惰性求值的能力”、8.13 节“实现一种数据模型或类型系统”和 9.9 节“把装饰器定义成类”。最后一个经典实例解决了函数装饰器、描述符和方法之间相互作用的深层次问题，说明了如何使用有 `__call__`

方法的类实现函数装饰器。如果既想装饰方法又想装饰函数，则还要实现 `__get__` 方法。

“PEP 487—Simpler customization of class creation”引入了特殊方法 `__set_name__`，还提供了一个验证描述符示例。

杂谈

`self` 的设计

在 Python 中，方法的第一个参数必须显式声明为 `self`，这个设计决定具有一定争议。使用这门语言 23 年后，我已经习惯了。我觉得这个决定体现了“变糟更好”（worse is better）原则。这是计算机科学家 Richard P. Gabriel 在“The Rise of Worse is Better”一文中提出的设计原则。这个原则的第一要义是“简单”。对此，Gabriel 说道：

设计方式必须简单，对实现和接口来说都应如此。简单的实现比简单的接口更重要。简单是设计过程中最重要的考虑因素。

Python 强制要求声明 `self` 参数正是这个原则的体现。这样，实现是简单了（甚至也优雅了），但牺牲了用户接口：方法的签名，例如 `def zfill(self, width):`，在外观上与 `label.zfill(8)` 调用不匹配。

这种做法由 Modula-3 语言（同样使用 `self` 标识符）创造，但是与 Python 有重要差异：在 Modula-3 中，接口的声明与实现是分开的，而且在接口声明中会省略 `self` 参数，因此对用户来说，接口声明中的方法显示的参数数量与调用时传入的参数数量完全一致。

随着时间的推移，Python 为方法参数提供的错误消息越来越清晰。对于除了 `self` 之外接受一个参数的用户定义的方法，如果

用户调用 `obj.meth()`，那么 Python 2.7 将抛出如下内容。

```
TypeError: meth() takes exactly 2 arguments (1 given)
```

而 Python 3 中则不再提及让人困惑的参数数量，而且指明了缺少的参数名称。

```
TypeError: meth() missing 1 required positional argument: 'x'
```

除了要明确把 `self` 作为参数，限制必须通过 `self` 访问实例属性也备受批评。例如，可以参考 A. M. Kuchling 发表的著名文章“Python Warts”（存档）。Kuchling 自己并不讨厌 `self` 限定符，但是他提到了这一点——可能是为了呼应 `comp.lang.python` 邮件列表中的观点。我自己并不介意输入 `self` 限定符，这样便于把局部变量和属性区分开。我介意的是在 `def` 语句中使用 `self`。

如果讨厌 Python 要求显式使用 `self`，可以想想 JavaScript 中隐式 `this` 那变幻莫测的语义，这样就会感觉好多了。像这样使用 `self` 有一些合理之处，Guido 在他的博客 *The History of Python* 中写了一篇文章，题为“Adding Support for User-Defined Classes”，说明了相关原因。

第 24 章 类元编程

所有人都知道调试程序的难度是写程序的两倍。所以如果你写程序的时候尽可能地“精明”，那么未来将如何调试它？

——Brian W. Kernighan 和 P. J. Plauger
《编程格调》

类元编程指在运行时创建或定制类的技艺。在 Python 中，类是一等对象，因此任何时候都可以使用函数新建类，无须使用 `class` 关键字。类装饰器也是函数，不过能够审查、修改，甚至把被装饰的类替换成另一个类。最后，元类是类元编程最高级的工具：使用元类可以创建具有某种特质的全新类，例如前面讲过的抽象基类。

元类功能强大，但是难以掌握。类装饰器能使用更简单的方式解决很多同类问题。此外，Python 3.6 实现的“PEP 487—Simpler customization of class creation”提供了一些特殊方法，可以完成以前需要元类或类装饰器才能完成的任务。¹

¹并不是说 PEP 487 会破坏使用那些功能的代码。只是在 Python 3.6 以前使用类装饰器或元类编写的代码，现在可以使用普通的类重构，写出的代码更简单且更高效。

本章按照复杂度从易到难讲解类元编程技术。



这是一个令人兴奋的话题，很容易让人忘乎所以。因此，我必须提出以下警告。

为了可读性和可维护性，或许应该避免在应用程序代码中使用本章讲解的技术。

反之，如果想编写下一个引起轰动的 Python 框架，那么肯定离不开这些工具。

24.1 本章新增内容

本书第 1 版“类元编程”一章中的所有代码依然能正常运行。然而，与 Python 3.6 新增的功能相比，以前的示例有一些已经不是最简单的方案了。

我把那些示例换掉了，有些是为了讲解 Python 新增的元编程功能，有些还增加了新需求，以便使用更高级的技术。新示例中有一些利用类型提示实现与 `@dataclass` 装饰器和 `typing.NamedTuple` 类似的类构建器。

24.10 节是新增的，站在一定高度上探讨了实际运用元类时需要注意的事项。



最好的重构方式之一是把换用更新、更简单的方案之后多出的代码删除——线上代码和图书均是如此。

首先研究 Python 数据模型为所有类定义的属性和方法。

24.2 身为对象的类

与 Python 中的很多程序实体一样，类也是对象。Python 数据模型为每个类定义了很多属性，参见 Python 标准库文档中“Built-in Types”一章的“4.13. Special Attributes”一节。这些属性中有 3 个属性已在本书中出现多次：`__class__`、`__name__` 和 `__mro__`。此外，还有以下标准属性。

```
cls.__bases__
```

由类的基类构成的元组。

```
cls.__qualname__
```


类或函数的限定名称，即从模块的全局作用域到类的点分路径。在一个类中定义另一个类时会用到这个属性。例如，在 Django 模型类 `Ox` 内部有一个名为 `Meta` 的类。`Meta` 类的 `__qualname__` 是 `Ox.Meta`，但 `__name__` 只是 `Meta`。这个属性的规范是“PEP 3155—Qualified name for classes and functions”。

```
cls.__subclasses__()
```

这个方法会返回包含类的直接子类的列表，其实现使用弱引用，以防止在超类和子类之间出现循环引用（子类在 `__bases__` 属性中储存指向超类的强引用）。这个方法返回的列表中是内存里现存的子类，不含尚未导入的模块中的子类。

```
cls.mro()
```

构建类时，如果需要获取储存在类属性 `__mro__` 中的超类元组，那么解释器就会调用这个方法。元类可以覆盖这个方法，定制要构建的类解析方法的顺序。



`dir(...)` 函数不列出本节提到的任何一个属性。

既然类是对象，那么类的类是什么呢？

24.3 type：内置的类工厂函数

我们通常认为 `type` 是一个函数，会返回对象所属的类，即 `type(my_object)` 返回 `my_object.__class__`。

然而，`type` 是一个类，在调用时会传入 3 个参数，创建一个新类。

以下面这个简单的类为例。

```
class MyClass(MySuperClass, MyMixin):  
    x = 42
```

```
def x2(self):
    return self.x * 2
```

使用 `type` 构造函数，可以在运行时创建 `MyClass` 类，如下所示。

```
MyClass = type('MyClass',
               (MySuperClass, MyMixin),
               {'x': 42, 'x2': lambda self: self.x * 2},
               )
```

这个 `type` 调用与前面的 `class MyClass...` 语句块在功能上是等同的。

Python 读取 `class` 语句时会调用 `type` 构建类对象，传入的参数如下所示。

`name`

`class` 关键字后的标识符，例如 `MyClass`。

`bases`

类标识符后面圆括号内提供的超类元组，如果 `class` 语句没有提到超类，则为 `(object,)`。

`dict`

属性名称到值的映射。可调用对象变成方法（参见 23.4 节），其他值变成类属性。



`type` 构造函数还接受一些可选的关键字参数，`type` 自身忽略这些参数，但是会原封不动地传给 `__init_subclass__` 方法，该方法必须使用它们。24.5 节将研究这个特殊方法，但是不涉及关键字参数。详见“PEP 487—Simpler customization of class creation”。

`type` 类是一个**元类**，即构建类的类。也就是说，`type` 类的实例还是类。标准库还提供了一些其他元类，不过 `type` 是默认的元类。

```
>>> type(7)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type(NotFoundError)
<class 'type'>
>>> class Whatever:
...     pass
...
>>> type(Whatever)
<class 'type'>
```

我们将在 24.8 节自己构建元类。

接下来使用内置函数 `type` 定义一个用于构建类的函数。

24.4 类工厂函数

本书多次提到标准库中的一个类工厂函数——`collections.namedtuple`。第 5 章还介绍过 `typing.NamedTuple` 和 `@dataclass`。这些类构建器都用到了本章涵盖的技术。

首先，我们将构建一个特别简单的工厂函数，用于创建可变对象的类——算是 `@dataclass` 最简单的替代品。

假设我在编写一个宠物店应用程序，想以简单的记录存储狗的数据。但是，我不想编写像下面这样的样板代码。

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

无趣.....各个字段名称出现了 3 次。写了这么多样板代码，甚至字符串表示形式都不友好。

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

参考 `collections.namedtuple`，下面创建一个 `record_factory` 函数，即时创建 `Dog` 这种简单的类。这个函数的用法如示例 24-1 所示。

示例 24-1 测试 `record_factory` 函数，一个简单的类工厂函数

```
>>> Dog = record_factory('Dog', 'name weight owner') ❶
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex ❷
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex ❸
>>> name, weight
('Rex', 30)
>>> "{2}'s dog weighs {1}kg".format(*rex) ❹
"Bob's dog weighs 30kg"
>>> rex.weight = 32 ❺
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
>>> Dog.__mro__ ❻
(<class 'factories.Dog'>, <class 'object'>)
```

❶ 工厂函数可以像 `namedtuple` 那样调用：先写类名，后跟一个字符串，列出属性名称，以空格分开。

❷ 友好的字符串表示形式。

❸ 实例是可迭代对象，因此赋值时可以便利地拆包。

❹ 传给 `format` 等函数时也可以拆包。

❺ 记录实例是可变的对象。

❸ 新建的类继承自 `object`，与我们的工厂函数没有关系。

`record_factory` 函数的代码在示例 24-2 中。²

²感谢我的朋友 J. S. O. Bueno 向我提供这个示例。

示例 24-2 `record_factory.py`：一个简单的类工厂函数

```
from typing import Union, Any
from collections.abc import Iterable, Iterator

FieldNames = Union[str, Iterable[str]] ❶

def record_factory(cls_name: str, field_names: FieldNames) ->
    type[tuple]: ❷

    slots = parse_identifiers(field_names) ❸

    def __init__(self, *args, **kwargs) -> None: ❹
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self) -> Iterator[Any]: ❺
        for name in self.__slots__:
            yield getattr(self, name)

    def __repr__(self): ❻
        values = ', '.join(f'{name}={value!r}'
                           for name, value in zip(self.__slots__, self))
        cls_name = self.__class__.__name__
        return f'{cls_name}({values})'

    cls_attrs = dict( ❼
        __slots__=slots,
        __init__=__init__,
        __iter__=__iter__,
        __repr__=__repr__,
    )

    return type(cls_name, (object,), cls_attrs) ❽

def parse_identifiers(names: FieldNames) -> tuple[str, ...]:
```

```
if isinstance(names, str):
    names = names.replace(',', ' ').split() ❸
if not all(s.isidentifier() for s in names):
    raise ValueError('names must all be valid identifiers')
return tuple(names)
```

❶ 用户可以使用一整个字符串或者产出字符串的可迭代对象提供字段名称。

❷ 接受的参数与 `collections.namedtuple` 的前两个参数类似，返回一个 `type`，即一个行为类似元组的类。

❸ 使用属性名构建一个元组，这将成为新建类的 `__slots__` 属性的值。

❹ 这个函数将成为新建类的 `__init__` 方法，接受位置参数和关键字参数。³

³我没有为参数添加类型提示，因为真正的类型是 `Any`。但是，我为返回值添加了类型提示，如若不然，`Mypy` 不会检查方法内部。

❺ 按照 `__slots__` 设定的顺序产出字段值。

❻ 迭代 `__slots__` 和 `self`，生成友好的字符串表示形式。

❼ 组建类属性字典。

❽ 调用 `type` 构造函数，构建新类，然后将其返回。

❾ 把以空格或逗号分隔的 `names` 转换成字符串列表。

示例 24-2 是我们第一次在类型提示中见到 `type`。如果只使用 `-> type` 注解，则表示 `record_factory` 会返回一个类——这也没错。但是，`-> type[tuple]` 注解更准确，它指明了返回的类是 `tuple` 的子类。

在示例 24-2 中, `record_factory` 函数的最后一行构建了一个类, 类的名称是 `cls_name` 的值, 唯一的直接基类是 `object`, 而且在命名空间中加载了 `__slots__`、`__init__`、`__iter__` 和 `__repr__`, 其中后 3 个是实例方法。

我们本可以把 `__slots__` 类属性的名称改成其他值, 但是如果那样做, 则必须实现 `__setattr__` 方法, 以验证在为属性赋值时属性的名称, 因为对于这种记录类, 我们希望属性始终是固定的那几个, 而且顺序相同。然而 11.11 节说过, `__slots__` 属性的主要特色是在处理数百万个实例时节省内存, 不过也有一些缺点。



`record_factory` 函数创建的类的实例不能序列化, 即不能使用 `pickle` 模块里的 `dump` 函数导出。示例 24-2 是为了说明如何使用 `type` 类满足简单的需求, 因此不会解决这个问题。如果想了解完整的方案, 请分析 `collections.namedtuple` 的源码, 搜索“pickling”这个词。

接下来介绍如何模拟 `typing.NamedTuple` 等更现代化的类构建器, 为用户使用 `class` 语句定义的类自动增加功能。

24.5 引出 `__init_subclass__`

`__init_subclass__` 和 `__set_name__` 均由“PEP 487—Simpler customization of class creation”提议增加。23.2.2 节讲过描述符的特殊方法 `__set_name__`, 本节探讨 `__init_subclass__`。

读过第 5 章, 我们了解到, 借助 `typing.NamedTuple` 和 `@dataclass`, 程序员可以使用 `class` 语句为新类指定属性, 然后由类构建器自动添加 `__init__`、`__repr__`、`__eq__` 等基本的方法, 增强新类的功能。

这两个类构建器均读取用户在 `class` 语句中添加的类型提示，以增强类的功能。静态类型检查工具还能通过那些类型提示验证用于设置或获取属性的代码。然而，`NamedTuple` 和 `@dataclass` 在运行时不能利用类型提示验证属性。接下来的示例中的 `Checked` 类则可以这样做。



运行时类型检查不可能涵盖所有静态类型提示，或许正是因为如此，`typing.NamedTuple` 和 `@dataclass` 没有勉为其难。然而，某些同时还是具体类的类型可以在 `Checked` 中使用，包括常用作字段内容的简单类型，例如 `str`、`int`、`float` 和 `bool`，以及由这些类型的元素构成的列表。

示例 24-3 展示了如何使用 `Checked` 构建一个 `Movie` 类。

示例 24-3 `initsub/checkedlib.py`: 衍生 `Checked` 创建 `Movie` 类的 `doctest`

```
>>> class Movie(Checked): ❶
...     title: str ❷
...     year: int
...     box_office: float
...
>>> movie = Movie(title='The Godfather', year=1972,
box_office=137) ❸
>>> movie.title
'The Godfather'
>>> movie ❹
Movie(title='The Godfather', year=1972, box_office=137.0)
```

❶ `Movie` 继承自 `Checked`（稍后在示例 24-5 中定义）。

❷ 使用构造函数注解各个属性。这里用的是内置类型。

❸ `Movie` 实例必须使用关键字参数创建。

❹ 字符串表示形式十分友好。

属性的类型提示所使用的构造函数可以是任何可调用对象，该对象接受零个或一个参数，返回一个符合字段类型的值，或者抛出 `TypeError` 或 `ValueError`，拒绝传入的参数。

在示例 24-3 中使用内置类型注解意味着类型的构造函数必须能够接受提供的值。也就是说，对于 `int`，不管传入的 `x` 是什么，`int(x)` 必须返回一个 `int` 值；对于 `str`，运行时可接受任何值，因为在 Python 中，`str(x)` 可处理任何 `x` 值。⁴

⁴的确如此，除非 `x` 所属的类覆盖了从 `object` 继承的 `__str__` 或 `__repr__` 方法，实现方式有误。

如果调用时没有传入参数，那么构造函数应该返回相应类型的默认值。⁵

⁵这个方案避免使用 `None` 作为默认值。避免空值是有原因的。一般来说，很难做到这一点，但是在某些情况下很简单。在 Python 以及 SQL 中，我喜欢使用空字符串而不是 `None` 或 `NULL` 来表示某个文本字段缺少数据。据我所知，Go 语言就强制施行了这个做法：在 Go 语言中，原始类型的变量和结构体字段默认使用“零值”（zero value）来初始化。如果感到好奇，可以看一下 *Tour of Go* 在线教程中的“Zero values”。

Python 内置类型构造函数的标准行为如下所示。

```
>>> int(), float(), bool(), str(), list(), dict(), set()
(0, 0.0, False, '', [], {}, set())
```

对于像 `Movie` 这样的 `Checked` 的子类，在创建实例时，如果缺少参数，则对应的值为字段构造函数返回的默认值。

```
>>> Movie(title='Life of Brian')
Movie(title='Life of Brian', year=0, box_office=0.0)
```

在实例化期间以及为实例直接设定属性时，使用构造函数进行验证。

```
>>> blockbuster = Movie(title='Avatar', year=2009,
box_office='billions')
Traceback (most recent call last):
```

```
...
TypeError: 'billions' is not compatible with box_office:float
>>> movie.year = 'MCMLXXII'
Traceback (most recent call last):
...
TypeError: 'MCMLXXII' is not compatible with year:int
```



Checked 的子类和静态类型检查

对于 `Movie` 类（在示例 24-3 中定义）的一个实例 `movie`，Mypy 在检查所在的 `.py` 源码文件时，将报告以下赋值语句有类型错误。

```
movie.year = 'MCMLXXII'
```

然而，Mypy 发现不了下面的构造函数调用中有类型错误。

```
blockbuster = Movie(title='Avatar', year='MMIX')
```

这是因为 `Movie` 继承了 `Checked.__init__`，而为了构建任意的用户定义的类，该方法的签名必须接受一切关键字参数。

如果在 `Checked` 的子类中使用类型提示 `list[float]` 声明一个字段，那么在赋予不兼容的内容时，Mypy 就会报错，但是 `Checked` 将忽略类型参数，把 `list[float]` 视作 `list`。

下面来看 `checkedlib.py` 的实现。第一个类是 `Field` 描述符，如示例 24-4 所示。

示例 24-4 `initsub/checkedlib.py`: `Field` 描述符类

```
from collections.abc import Callable ❶
from typing import Any, NoReturn, get_type_hints

class Field:
    def __init__(self, name: str, constructor: Callable) -> None:
```

❷

```

        if not callable(constructor) or constructor is type(None):
③
            raise TypeError(f'{name!r} type hint must be
callable')
        self.name = name
        self.constructor = constructor

    def __set__(self, instance: Any, value: Any) -> None:
        if value is ...: ④
            value = self.constructor()
        else:
            try:
                value = self.constructor(value) ⑤
            except (TypeError, ValueError) as e: ⑥
                type_name = self.constructor.__name__
                msg = f'{value!r} is not compatible with
{self.name}:{type_name}'
                raise TypeError(msg) from e
            instance.__dict__[self.name] = value ⑦

```

① 注意，从 Python 3.9 开始，注解中的 `Callable` 类型是 `collections.abc` 中的抽象基类，不是已经弃用的 `typing.Callable`。

② 这是一个极简的 `Callable` 类型提示，`constructor` 的参数类型和返回值类型均隐含为 `Any`。

③ 使用内置函数 `callable` 做运行时检查。^⑥ 针对 `type(None)` 的测试不可省略，因为从类型中读取的 `None` 是 `NoneType`，即 `None` 的类（因此可以调用），但是无用的构造函数只返回 `None`。

^⑥我认为 `callable` 应当可以用作类型提示。截至 2021 年 5 月 6 日，相应的工单还处于待解决状态。

④ 如果 `Checked.__init__` 把 `value` 设为 `...`（内置对象 `Ellipsis`），就调用无参数的 `constructor`。

⑤ 否则，传入提供的 `value`，调用 `constructor`。

❹ 如果 `constructor` 抛出其中一个异常，那么我们就抛出 `TypeError`，并提供一条包含字段和构造函数名称的有用消息，例如 `'MMIX' is not compatible with year:int`。

❺ 如果没有异常抛出，就把 `value` 存入 `instance.__dict__`。

在 `__set__` 方法中，需要捕获 `TypeError` 和 `ValueError`，因为根据传入的参数，内置构造函数有可能抛出其中一个。例如，`float(None)` 抛出 `TypeError`，而 `float('A')` 抛出 `ValueError`。但是，`float('8')` 不抛出错误，而是返回 `8.0`。我特意指出这一点是为了告诉你，对这个简单的示例来说，这个行为是可以接受的，不是 bug。



23.2.2 节介绍过对描述符有一定用处的特殊方法 `__set_name__`。Field 类用不到该方法，因为描述符不在客户源码中实例化，用户声明的类型是构造函数，如 `Movie` 类所示（参见示例 24-3）。Field 描述符实例在运行时由 `Checked.__init_subclass__` 方法创建，详见示例 24-5。

现在开始定义 `Checked` 类。我把源码分成了两个代码清单。示例 24-5 是 `Checked` 类的前半部分，包含最重要的几个方法。余下的部分在示例 24-6 中。

示例 24-5 `initsub/checkedlib.py`: `Checked` 类最重要的几个方法

```
class Checked:
    @classmethod
    def _fields(cls) -> dict[str, type]: ❶
        return get_type_hints(cls)

    def __init_subclass__(subclass) -> None: ❷
        super().__init_subclass__() ❸
        for name, constructor in subclass._fields().items(): ❹
            setattr(subclass, name, Field(name, constructor)) ❺
```

```

def __init__(self, **kwargs: Any) -> None:
    for name in self._fields():
        value = kwargs.pop(name, ...)
        setattr(self, name, value)
    if kwargs:
        self.__flag_unknown_attrs(*kwargs)

```

❶ 编写这个类方法的目的是对类的余下部分隐藏

`typing.get_type_hints` 调用。如果只需要支持 Python 3.10 及以上版本，那么直接调用 `inspect.get_annotations` 即可。这两个函数的问题见 15.5.1 节。

❷ `__init_subclass__` 方法在定义当前类的子类时调用。该方法的第一个参数是新定义的子类，因此我把参数命名为 `subclass`，而不是以往的 `cls`。详见后面的“`__init_subclass__` 不是常规的类方法”附注栏。

❸ 严格来说，不需要调用 `super().__init_subclass__()`，但是为了与同一继承图中实现了 `__init_subclass__()` 方法的其他类“和谐共处”，则应该调用它。详见 14.4 节。

❹ 迭代各个字段的 `name` 和 `constructor`.....

❺ 在 `subclass` 上创建一个名为 `name` 的属性，绑定以 `name` 和 `constructor` 参数化的 `Field` 描述符。

❻ 遍历类字段中的各个 `name`.....

❼ 从 `kwargs` 中获取对应的 `value`，并把 `value` 从 `kwargs` 中删除。以 `...` (`Ellipsis` 对象) 作为 `value` 的默认值是为了区分传入 `None` 值和未提供值两种情况。⁷

⁷19.6.3 节的“循环、哨符和毒药丸”附注栏中介绍过，使用 `Ellipsis` 对象做哨符值，省事又安全。`Ellipsis` 对象存在很长时间了，不过最近人们为它找到了更多用途，例如类型提示和 `NumPy` 就经常用到。

- ⑧ setattr 调用触发 Checked.__setattr__ (参见示例 24-6)。
- ⑨ 如果 kwargs 中还有余项, 则说明存在与声明的字段不匹配的名称, __init__ 将执行失败。
- ⑩ 错误由 __flag_unknown_attrs 方法 (参见示例 24-6) 报告。该方法接受的参数 *names 是那些未知的属性名称。*kwargs 中星号的作用是把键作为序列传给方法。

`__init_subclass__` 不是常规的类方法

`__init_subclass__` 不需要使用 @classmethod 装饰器, 这没什么好意外的, 特殊方法 `__new__` 也不用 @classmethod 装饰器, 但是它的行为与类方法一样。Python 传给 `__init_subclass__` 方法的第一个参数是一个类, 但不是实现 `__init_subclass__` 方法的类, 而是那个类的子类。这一点与 `__new__` 以及我所知道的其他类方法不同。因此, 我觉得 `__init_subclass__` 不是一般意义上的类方法, 倘若把第一个参数命名为 cls, 则容易让人误解。`__init_subclass__` 的文档把第一个参数命名为 cls, 不过有附加说明: “.....在实例化当前类时调用。cls 是新定义的子类。”

下面来看 Checked 类余下的方法 (接续示例 24-5)。注意, `_fields` 和 `_asdict` 这两个方法的名称开头都有 `_`, 原因与 `collections.namedtuple` API 一样: 减少与用户定义的字段名称发生冲突的概率。

示例 24-6 initsub/checkedlib.py: Checked 类余下的方法

```
def __setattr__(self, name: str, value: Any) -> None: ❶
    if name in self._fields(): ❷
        cls = self.__class__

        descriptor = getattr(cls, name)
```

```

        descriptor.__set__(self, value) ❸
    else: ❹
        self.__flag_unknown_attrs(name)

    def __flag_unknown_attrs(self, *names: str) -> NoReturn: ❺
        plural = 's' if len(names) > 1 else ''
        extra = ', '.join(f'{name!r}' for name in names)
        cls_name = repr(self.__class__.__name__)
        raise AttributeError(f'{cls_name} object has no
attribute{plural} {extra}')

    def _asdict(self) -> dict[str, Any]: ❻
        return {
            name: getattr(self, name)
            for name, attr in self.__class__.__dict__.items()
            if isinstance(attr, Field)
        }

    def __repr__(self) -> str: ❼
        kwargs = ', '.join(
            f'{key}={value!r}' for key, value in
self._asdict().items()
        )
        return f'{self.__class__.__name__}({kwargs})'

```

❶ 截获一切设置实例属性的操作。为了避免设置未知属性，需要这个方法。

❷ 如果属性名称是已知的，就获取对应的描述符。

❸ 一般无须显式调用描述符的 `__set__` 方法。但是这里需要，因为一切设置实例属性的操作均被 `__setattr__` 方法截获了，即使存在诸如 `Field` 之类的覆盖型描述符。⁸

⁸覆盖型描述符的具体概念见 23.3.1 节。

❹ 否则，属性名称是未知的，由 `__flag_unknown_attrs` 抛出异常。

❺ 构建一个详细的错误消息，列出所有不接受的参数，抛出 `AttributeError`。这里用到了不常见的特殊类型 `NoReturn`，详见

8.5.12 节。

❹ 根据 `Movie` 对象的属性创建一个字典。我本打算把这个方法命名为 `_as_dict`，但是按照 `collections.namedtuple` 的命名约定，我最终使用了 `_asdict`。

❺ 本例中定义 `_asdict` 方法的主要目的是提供一个友好的字符串表示形式。

`Checked` 示例演示了当实现阻碍实例化之后随意设置属性的 `__setattr__` 方法时如何处理覆盖型描述符。对于这个示例，是否需要实现 `__setattr__` 方法，存在争议。如果不实现，那么 `movie.director = 'Greta Gerwig'` 可以成功，但是 `director` 属性无法以任何形式检查，而且不会出现在字符串表示形式中，也不会出现在 `_asdict` 返回的字典中。

在 `record_factory.py`（参见示例 24-2）中，我使用类属性 `__slots__` 解决了这个问题。然而，这里不适用这种更简单的方案，原因参见接下来的内容。

为什么 `__init_subclass__` 不能配置 `__slots__`

`__slots__` 属性仅在作为类命名空间中的一个条目传给 `type.__new__` 时才有用。为现有的类添加 `__slots__` 属性没有效果。Python 只在已经构建类之后调用 `__init_subclass__` 方法，此时配置 `__slots__` 属性为时已晚。类装饰器也不能配置 `__slots__` 属性，因为它比 `__init_subclass__` 应用得还晚。这里涉及的时序问题将在 24.7 节探讨。

如果想在运行时配置 `__slots__` 属性，那么客户代码必须自己构建类命名空间，作为最后一个参数传给 `type.__new__`。为此，需要编写一个类工厂函数（类似于 `record_factory.py`），或者采用尖端技术，实现元类。24.8 节将说明如何动态配置 `__slots__` 属性。

Python 3.7 实现的 PEP 487 简化了使用 `__init_subclass__` 创建自定义类的过程，在此之前，类似的功能只能使用类装饰器实现，详见 24.6 节。

24.6 使用类装饰器增强类的功能

类装饰器是一种可调用对象，行为与函数装饰器类似：以被装饰的类为参数，返回一个类，取代被装饰的类。类装饰器通常返回被装饰的类，不过会通过属性赋值注入更多方法。

把更简单的 `__init_subclass__` 放在一旁不用，选择使用类装饰器，最常见的原因应该是想避免妨碍其他类功能，例如继承和元类。⁹

⁹“PEP 557–Data Classes”在摘要中就以这个原因解释为什么使用类装饰器实现。

本节分析 `checkeddeco.py` 模块，它提供的服务与 `checkedlib.py` 相同，不过以类装饰器实现。一如往常，先看用法示例。示例 24-7 摘自 `checkeddeco.py` 中的 `doctest`。

示例 24-7 `checkeddeco.py`：创建使用 `@checked` 装饰的 `Movie` 类

```
>>> @checked
... class Movie:
...     title: str
...     year: int
...     box_office: float
...
>>> movie = Movie(title='The Godfather', year=1972,
box_office=137)
>>> movie.title
'The Godfather'
>>> movie
Movie(title='The Godfather', year=1972, box_office=137.0)
```

示例 24-7 与示例 24-3 之间唯一的区别是 `Movie` 类的声明方式不同：示例 24-7 使用 `@checked` 装饰，而不是子类化 `Checked`。除此

之外，对外行为是一样的，包括 24.5 节中示例 24-3 后面的类型验证和默认值赋值。

下面来分析 checkeddeco.py 的实现。导入语句和 Field 类与 checkedlib.py 一样，详见示例 24-4。checkeddeco.py 中没有其他类，只有几个函数。

之前在 `__init_subclass__` 方法中实现的逻辑，现在被放在 checked 函数中。类装饰器 checked 的实现如示例 24-8 所示。

示例 24-8 checkeddeco.py: 类装饰器 checked

```
def checked(cls: type) -> type: ❶
    for name, constructor in _fields(cls).items(): ❷
        setattr(cls, name, Field(name, constructor)) ❸

    cls._fields = classmethod(_fields) # type: ignore ❹

    instance_methods = ( ❺
        __init__,
        __repr__,
        __setattr__,
        __asdict__,
        __flag_unknown_attrs,
    )
    for method in instance_methods: ❻
        setattr(cls, method.__name__, method)

    return cls ❼
```

❶ 注意，类是 type 的实例。这些类型提示充分表明这是一个类装饰器：接受一个类，返回一个类。

❷ `_fields` 是一个顶层函数，在该模块后面定义（参见示例 24-9）。

❸ 把 `_fields` 返回的各个属性替换为一个 Field 描述符实例，这与示例 24-5 中 `__init_subclass__` 方法的逻辑一样。不过，这里还

要多做些工作.....

④ 把 `_fields` 构建为类方法，添加到被装饰的类中。 `type: ignore` 注释不可省略，否则 Mypy 将抱怨 `type` 没有 `_fields` 属性。

⑤ 将变成被装饰的类的实例方法的模块级函数。

⑥ 把 `instance_methods` 中的各个函数添加到 `cls` 中。

⑦ 返回被装饰的类 `cls`，满足类装饰器的基本合约。

在 `checkeddeco.py` 模块中，除装饰器 `checked` 之外，所有顶层函数的名称都以下划线开头。采用这种命名约定的原因如下。

- `checked` 函数是 `checkeddeco.py` 模块的公开接口，其他函数则不是。
- 示例 24-9 中的函数将注入被装饰的类，以 `_` 开头可以减少与用户在被装饰的类中定义的属性和方法出现名称冲突的概率。

`checkeddeco.py` 模块余下的内容如示例 24-9 所示。这些模块级函数的代码与 `checkedlib.py` 模块中 `Checked` 类相应的方法一样，具体说明见示例 24-5 和示例 24-6。

示例 24-9 checkeddeco.py: 注入被装饰的类的方法

```
def _fields(cls: type) -> dict[str, type]:
    return get_type_hints(cls)

def __init__(self: Any, **kwargs: Any) -> None:
    for name in self._fields():
        value = kwargs.pop(name, ...)
        setattr(self, name, value)
    if kwargs:
        self.__flag_unknown_attrs(*kwargs)

def __setattr__(self: Any, name: str, value: Any) -> None:
    if name in self._fields():
        cls = self.__class__
```

```

        descriptor = getattr(cls, name)
        descriptor.__set__(self, value)
    else:
        self.__flag_unknown_attrs(name)

def __flag_unknown_attrs(self: Any, *names: str) -> NoReturn:
    plural = 's' if len(names) > 1 else ''
    extra = ', '.join(f'{name!r}' for name in names)
    cls_name = repr(self.__class__.__name__)
    raise AttributeError(f'{cls_name} has no attribute{plural}{extra}')

def _asdict(self: Any) -> dict[str, Any]:
    return {
        name: getattr(self, name)
        for name, attr in self.__class__.__dict__.items()
        if isinstance(attr, Field)
    }

def __repr__(self: Any) -> str:
    kwargs = ', '.join(
        f'{key}={value!r}' for key, value in
self._asdict().items()
    )
    return f'{self.__class__.__name__}({kwargs})'

```

注意，`_fields` 函数在 `checkeddeco.py` 模块中有两个职责：一是在 `checked` 装饰器的第一行中用作常规函数，二是作为类方法注入被装饰的类。

`checkeddeco.py` 模块实现了一个简单但用处不小的类装饰器。Python 的 `@dataclass` 装饰器功能更丰富，支持很多配置选项，可为被装饰的类添加更多方法，与被装饰的类中用户定义的方法出现冲突时还能自行处理或发出警告，甚至还能遍历 `__mro__`，收集用户在被装饰的类的超类中声明的属性。在 Python 3.9 中，`dataclasses` 包的源码长达 1200 多行。

对类做元编程时，必须知晓在构造类的过程中 Python 解释器何时求解各个代码块，详见 24.7 节。

24.7 导入时和运行时比较

Python 程序员会区分“导入时”和“运行时”，不过这两个术语没有严格的定义，而且二者之间存在灰色地带。

在导入时，解释器执行以下操作。

01. 从上到下一次性解析完 .py 模块的源码。此时可能抛出 `SyntaxError`。
02. 编译生成用于执行的字节码。
03. 执行编译后的模块中的顶层代码。

如果本地的 `__pycache__` 文件夹中有最新的 .pyc 文件，则解释器会跳过解析和编译步骤，因为已经有供运行的字节码了。

解析和编译肯定是“导入时”活动，不过那个时期还会做些其他事，因为 Python 中的语句大部分是可执行的，也就是说语句可能会运行用户代码，修改用户程序的状态。

需要特别注意的是，`import` 语句不只是声明而已，¹⁰ 首次把模块导入进程时，所导入模块中的全部顶层代码都将运行。后续再导入相同的模块将使用缓存，唯一需要做的事情是把导入的对象绑定到客户模块中的名称上。顶层代码可以做任何事，包括通常在“运行时”做的事，例如写入日志或连接数据库。¹¹ 因此，“导入时”与“运行时”之间的界线是模糊的：`import` 语句可以触发各种“运行时”行为。反过来，“导入时”也可能深埋在运行时中，因为任何常规函数中都可以使用 `import` 语句和内置函数 `__import__()`。

¹⁰相比之下，Java 中的 `import` 语句只是声明，用于告知编译器需要特定的包。

¹¹此处并不是说导入模块时应该连接数据库，只是指出来可以做到这一点。

以上说明模糊又抽象，下面通过实验具体分析。

求解时间实验

假设有一个 `evaldemo.py` 脚本，用到了 `builderlib.py` 模块中定义的一个类装饰器、一个描述符和一个基于 `__init_subclass__` 实现的类构建器。这两个模块中有多个 `print` 调用，以展示背后执行的操作。除此之外，没有其他用途。这些实验的目的是观察 `print` 调用以什么顺序执行。



对一个类同时使用类装饰器和基于 `__init_subclass__` 实现的类构建器，应该算是过度设计了，风险很大。这里不按常理出牌是为了实验，观察类装饰器和 `__init_subclass__` 对类所做的改动发生在什么时刻。

先看 `builderlib.py` 模块。我把这个模块的内容一分为二，分别放在示例 24-10 和示例 24-11 中。

示例 24-10 `builderlib.py`：模块的前半部分

```
print('@ builderlib module start')

class Builder: ❶
    print('@ Builder body')

    def __init_subclass__(cls): ❷
        print(f'@ Builder.__init_subclass__({cls!r})')

        def inner_0(self): ❸
            print(f'@ SuperA.__init_subclass__:inner_0({self!r})')

        cls.method_a = inner_0

    def __init__(self):
        super().__init__()
        print(f'@ Builder.__init__({self!r})')

def deco(cls): ❹
    print(f'@ deco({cls!r})')
    def inner_1(self): ❺
        print(f'@ deco:inner_1({self!r})')
```

```
cls.method_b = inner_1
return cls ❹
```

- ❶ 这是一个类构建器.....
- ❷ 实现了 `__init_subclass__` 方法。
- ❸ 定义一个函数，通过下面的赋值语句添加到子类中。
- ❹ 一个类装饰器。
- ❺ 要添加到被装饰的类中的函数。
- ❻ 返回通过参数传入的类。

builderlib.py 模块的后半部分如示例 24-11 所示。

示例 24-11 builderlib.py: 模块的后半部分

```
class Descriptor: ❶
    print('@ Descriptor body')

    def __init__(self): ❷
        print(f'@ Descriptor.__init__({self!r})')

    def __set_name__(self, owner, name): ❸
        args = (self, owner, name)
        print(f'@ Descriptor.__set_name__({args!r})')

    def __set__(self, instance, value): ❹
        args = (self, instance, value)
        print(f'@ Descriptor.__set__({args!r})')

    def __repr__(self):
        return '<Descriptor instance>'

print('@ builderlib module end')
```

- ❶ 一个描述符类.....

- ❷ 演示描述符实例何时创建.....
- ❸ 以及在构造 owner 类的过程中, `__set_name__` 何时调用。
- ❹ 与其他方法一样, `__set__` 方法除了显示接收的参数之外什么也没做。

在 Python 控制台中导入 `builderlib.py`, 将看到如下输出。

```
>>> import builderlib
@ builderlib module start
@ Builder body
@ Descriptor body
@ builderlib module end
```

注意, `builderlib.py` 打印的行以 `@` 开头。

现在来看 `evaldemo.py` 脚本, 该脚本将触发 `builderlib.py` 模块中的特殊方法, 如示例 24-12 所示。

示例 24-12 `evaldemo.py`: 使用 `builderlib.py` 模块做实验的脚本

```
#!/usr/bin/env python3

from builderlib import Builder, deco, Descriptor

print('# evaldemo module start')

@deco ❶
class Klass(Builder): ❷
    print('# Klass body')

    attr = Descriptor() ❸

    def __init__(self):
        super().__init__()
        print(f'# Klass.__init__({self!r})')

    def __repr__(self):
        return '<Klass instance>'
```



```

def main(): ❹
    obj = Klass()
    obj.method_a()
    obj.method_b()
    obj.attr = 999

if __name__ == '__main__':
    main()

print('# evaldemo module end')

```

❶ 应用装饰器。

❷ 子类化 Builder, 触发 `__init_subclass__`。

❸ 实例化描述符。

❹ 仅当以主程序运行该模块时才调用这个函数。

evaldemo.py 脚本中 `print` 调用显示的行以 `#` 开头。再次打开控制台, 导入 evaldemo.py, 看到的输出如示例 24-13 所示。

示例 24-13 在控制台中使用 evaldemo.py 做实验

```

>>> import evaldemo
@ builderlib module start ❶
@ Builder body
@ Descriptor body
@ builderlib module end
# evaldemo module start
# Klass body ❷
@ Descriptor.__init__(<Descriptor instance>) ❸
@ Descriptor.__set_name__(<Descriptor instance>,
    <class 'evaldemo.Klass'>, 'attr') ❹
@ Builder.__init_subclass__(<class 'evaldemo.Klass'>) ❺
@ deco(<class 'evaldemo.Klass'>) ❻
# evaldemo module end

```

❶ 前 4 行是 `from builderlib import...` 的结果。如果没有关闭前一次实验打开的控制台, 则看不到这部分输出, 因为已经加载了 builderlib.py。

- ❷ 这表明 Python 开始读取 `Klass` 的主体。此时，类对象尚不存在。
- ❸ 创建描述符实例，绑定到 Python 传给默认类对象构造方法 `type.__new__` 的命名空间中的 `attr` 上。
- ❹ 此时，Python 内置的 `type.__new__` 方法已经创建 `Klass` 对象，在提供该方法的描述符类的各个实例上调用 `__set_name__`，把 `Klass` 传给 `owner` 参数。
- ❺ 然后，`type.__new__` 在 `Klass` 的超类上调用 `__init_subclass__`，传入的第一个参数是 `Klass`。
- ❻ `type.__new__` 返回类对象后，Python 应用装饰器。在这个示例中，`deco` 返回的类绑定到模块命名空间中的 `Klass` 上。

`type.__new__` 用 C 语言实现。我描述的行为见于《Python 语言参考手册》中的 3.3.3.6 节“创建类对象”。

注意，控制台会话没有执行 `evaldemo.py` 中的 `main()` 函数（参见示例 24-12），因此没有创建 `Klass` 实例。我们看到的所有操作均由“导入时”触发：导入 `builderlib`，定义 `Klass`。

如果把 `evaldemo.py` 当作脚本运行，则输出的前几行与示例 24-13 一样，后面还多出几行。多出的几行是运行 `main()` 函数的结果，如示例 24-14 所示。

示例 24-14 把 `evaldemo.py` 当作程序运行

```
$ ./evaldemo.py
[..... 省略9行 .....]
@ deco(<class '__main__.Klass'>) ❶
@ Builder.__init__(<Klass instance>) ❷
# Klass.__init__(<Klass instance>)
@ SuperA.__init_subclass__:inner_0(<Klass instance>) ❸
@ deco:inner_1(<Klass instance>) ❹
@ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999)
```

⑤

```
# evaldemo module end
```

- ❶ 前 10 行（包括这一行）与示例 24-13 相同。
- ❷ 由 `Klass.__init__` 中的 `super().__init__()` 触发。
- ❸ 由 `main` 中的 `obj.method_a()` 触发，`method_a` 由 `SuperA.__init_subclass__` 注入。
- ❹ 由 `main` 中的 `obj.method_b()` 触发，`method_b` 由 `deco` 注入。
- ❺ 由 `main` 中的 `obj.attr = 999` 触发。

实现了 `__init_subclass__` 的基类和类装饰器都是强大的工具，但是仅适用于背后已由 `type.__new__` 构建的类。如果想调整传给 `type.__new__` 的参数——十分少见，则需要使用元类。本章以及本书最后一个话题就是元类。

24.8 元类入门

（元类）是深奥的知识，99% 的用户无须关注。如果想知道是否需要使用元类，那么我告诉你，不需要（真正需要使用元类的人确信他们需要，无须解释原因）。

——Tim Peters

Timsort 算法的发明者，活跃的 Python 贡献者¹²

¹²摘自 `comp.lang.python` 邮件列表中对“Acrimony in c.l.p.”话题的回复。前言中引述的那句话也出自发布于 2002 年 12 月 23 日的这则消息。TimBot 在那天获得了灵感。

元类是制造类的工厂，不过不是函数（例如示例 24-2 中的 `record_factory`），而是类。也就是说，元类也是类，其实例还是

类。图 24-1 使用机器和小怪兽图示法描述了元类，可以看出，元类是生产机器的机器。

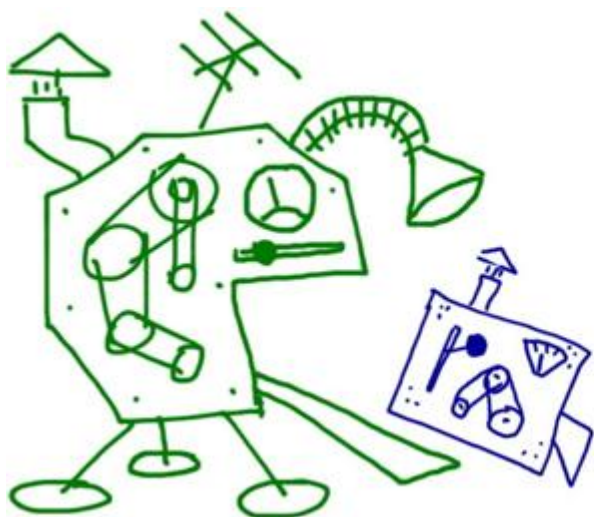


图 24-1：元类是用于构建类的类

根据 Python 对象模型，类是对象，因此类肯定是另外某个类的实例。默认情况下，Python 中的类是 `type` 的实例。也就是说，`type` 是大多数内置的类和用户定义的类的元类。

```
>>> str.__class__
<class 'type'>
>>> from bulkfood_v5 import LineItem
>>> LineItem.__class__
<class 'type'>
>>> type.__class__
<class 'type'>
```

为了避免无限回溯，`type` 的类是其自身，如上述代码最后一行所示。

注意，我没有说 `str` 或 `LineItem` 是 `type` 的子类。我的意思是，`str` 和 `LineItem` 是 `type` 的实例。这两个类是 `object` 的子类。图 24-2 或许有助于你厘清这个奇怪的现象。



object 类和 type 类之间的关系很独特：object 是 type 的实例，而 type 是 object 的子类。这种关系很“神奇”，无法使用 Python 代码表述，因为定义其中一个之前另一个必须存在。type 是自身的实例这一点也很神奇。

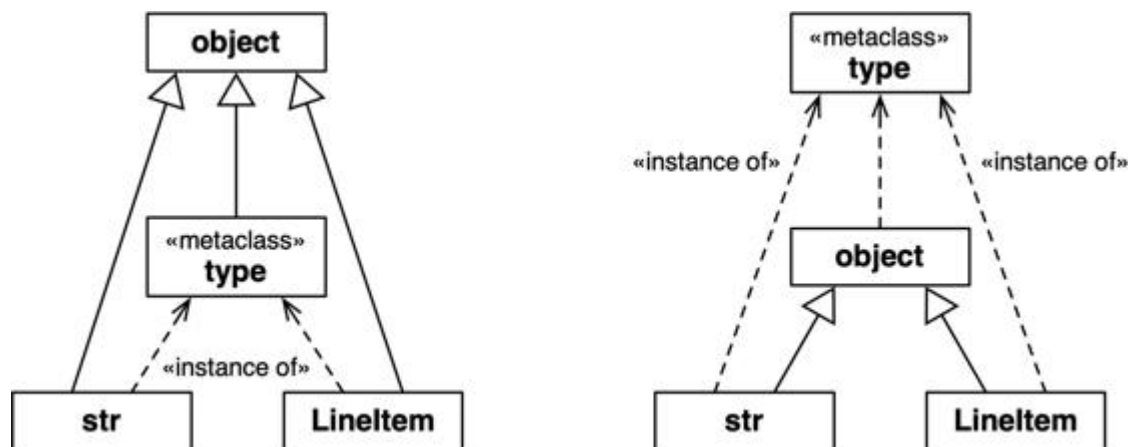


图 24-2：两个示意图都是正确的。左图强调 str、type 和 LineItem 是 object 的子类。右图则清楚地表明 str、object 和 LineItem 是 type 的实例，因为它们都是类

以下代码片段表明，collections.Iterable 所属的类是 abc.ABCMeta。注意，Iterable 是抽象类，而 ABCMeta 是具体类——然而，Iterable 是 ABCMeta 的实例。

```
>>> from collections.abc import Iterable
>>> Iterable.__class__
<class 'abc.ABCMeta'>
>>> import abc
>>> from abc import ABCMeta
>>> ABCMeta.__class__
<class 'type'>
```

向上追溯，ABCMeta 最终所属的类也是 type。所有类都直接或间接地是 type 的实例，不过只有元类同时也是 type 的子类。如果想理解元类，那么一定要知道这种关系：元类（例如 ABCMeta）从 type 类继承了构造类的能力。图 24-3 对这种至关重要的关系做了图解。

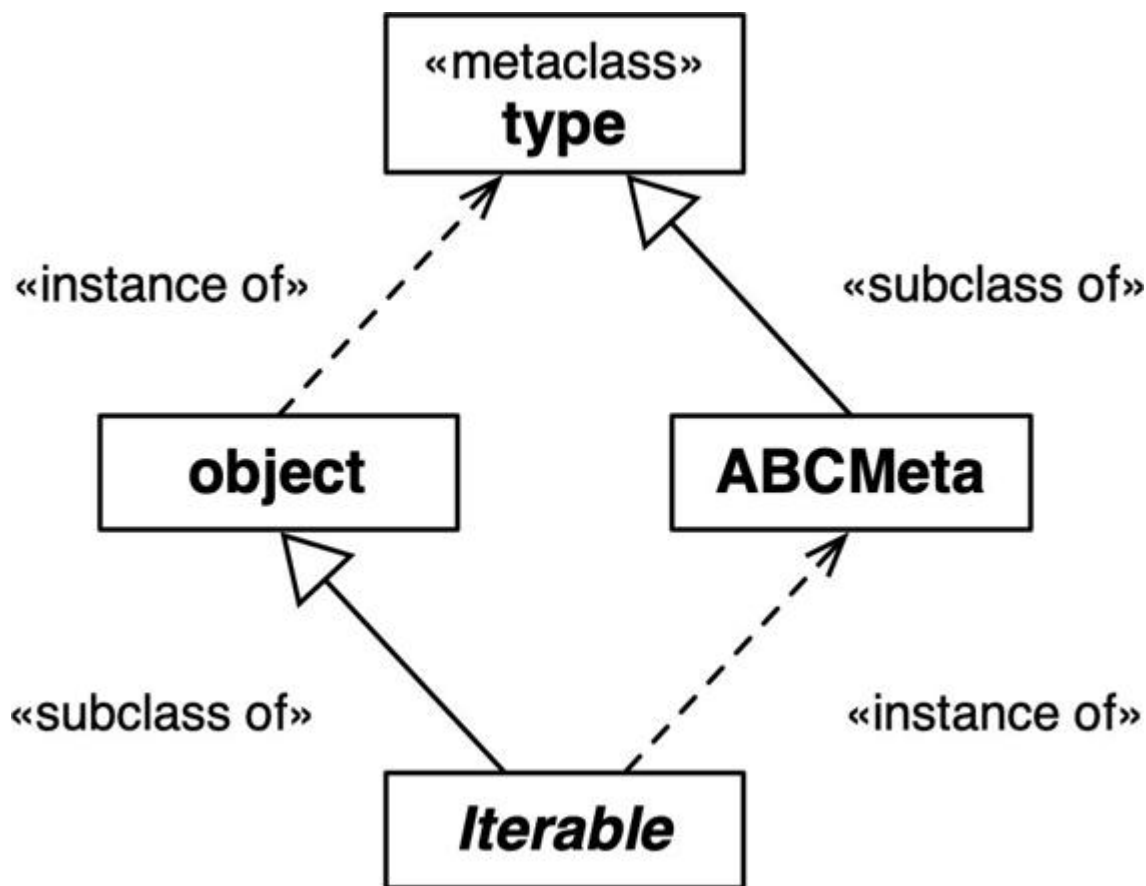


图 24-3: Iterable 既是 object 的子类，也是 ABCMeta 的实例。object 和 ABCMeta 都是 type 的实例，但是这里的重要关系是，ABCMeta 还是 type 的子类，因为 ABCMeta 是元类。示意图中只有 Iterable 是抽象类

我们要抓住的重点是，元类是 type 的子类，因此可以作为制造类的工厂。元类通过实现特殊方法定制实例，详见 24.8.1 节。

24.8.1 元类如何定制类

使用元类之前，务必理解 `__new__` 方法对一个类的作用（详见 22.2.3 节）。

元类创建的实例是类，因此同样的机制也适用于“元”层。以下面的声明为例。

```
class Klass(SuperKlass, metaclass=MetaKlass):
    x = 42
    def __init__(self, y):
        self.y = y
```

为了处理这个 `class` 语句，Python 调用 `MetaKlass.__new__`，传入以下参数。

`meta_cls`

元类自身 (`MetaKlass`)，因为 `__new__` 被当作类方法使用。

`cls_name`

字符串 `Klass`。

`bases`

只有一个元素的元组 (`SuperKlass,`)，如果是多重继承则有多
个元素。

`cls_dict`

类似下面的映射。

```
{x: 42, '__init__': <function __init__ at 0x1009c4040>}
```

实现 `MetaKlass.__new__` 时，可以对参数进行审查和修改，然后传给 `super().__new__`，最终调用 `type.__new__` 创建新的类对象。

`super().__new__` 返回之后，还可以进一步处理新创建的类，返回给 Python。随后，Python 调用

`SuperKlass.__init_subclass__`，传入新创建的类，如果有类装饰器的话，还会应用类装饰器。最后，Python 把类对象绑定给所在

命名空间中的名称——`class` 语句是顶层语句时，所在的命名空间通常是模块全局命名空间。

在元类的 `__new__` 方法中，最常执行的操作是向 `cls_dict` 中添加项，或者替换其中的项。`cls_dict` 是一个映射，表示待构造的类的命名空间。例如，调用 `super().__new__` 之前，可以向 `cls_dict` 中添加函数，为待构造的类注入方法。然而，请注意，方法也可以在构建类之后添加，不然 `__init_subclass__` 和类装饰器就失去存在意义了。

有一个属性必须在运行 `type.__new__` 之前添加到 `cls_dict` 中，即 `__slots__`，详见 24.5 节中的“为什么 `__init_subclass__` 不能配置 `__slots__`”。元类的 `__new__` 方法是配置 `__slots__` 的理想位置。24.8.2 节将说明具体做法。

24.8.2 一个友好的元类示例

本节的 `MetaBunch` 元类改编自 *Python in a Nutshell, 3rd ed.* (Alex Martelli、Anna Ravenscroft 和 Steve Holden 著) 一书中第 4 章的最后一个示例。¹³ 那个示例兼顾 Python 2.7 和 Python 3.5，但是我假设你使用 Python 3.6 或以上版本，因此代码得以大大简化。

¹³几位作者很友善，授权我使用他们的示例。`MetaBunch` 首次出现在 Martelli 于 2002 年 7 月 7 日发布在 `comp.lang.python` 群组中的一则消息里，题为“a nice metaclass example (was Re: structs in python)”。那个话题讨论的是在 Python 中如何使用记录类型的数据结构，Martelli 发布的消息是其中一个回复。Martelli 最初编写的代码针对 Python 2，不过只需改动一处，依然可以运行：在 Python 3 中使用元类，必须在类声明中使用 `metaclass` 关键字参数，例如 `Bunch(metaclass=MetaBunch)`，而旧时的约定是添加一个类级属性 `__metaclass__`。

首先，看一下 `Bunch` 基类实现的功能。

```
>>> class Point(Bunch):
...     x = 0.0
...     y = 0.0
...     color = 'gray'
```



```

...
>>> Point(x=1.2, y=3, color='green')
Point(x=1.2, y=3, color='green')
>>> p = Point()
>>> p.x, p.y, p.color
(0.0, 0.0, 'gray')
>>> p
Point()

```

还记得吗？Checked 基于类变量的类型提示，把名称分配给子类中的 Field 描述符，其实不是类的属性，因为没有值。

而 Bunch 的子类则不同，类属性是有值的，这些值将变成实例属性的默认值。生成的 `__repr__` 方法忽略等于默认值的属性。

MetaBunch (Bunch 的元类) 根据用户在自己定义的类中声明的类属性为新类生成 `__slots__`。如此一来，实例化和后续赋值都不能使用未声明的属性。

```

>>> Point(x=1, y=2, z=3)
Traceback (most recent call last):
...
AttributeError: No slots left for: 'z'
>>> p = Point(x=21)
>>> p.y = 42
>>> p
Point(x=21, y=42)
>>> p.flavor = 'banana'
Traceback (most recent call last):
...
AttributeError: 'Point' object has no attribute 'flavor'

```

MetaBunch 的代码十分优雅，如示例 24-15 所示。

示例 24-15 metabunch/from3.6/bunch.py: MetaBunch 元类和 Bunch 类

```

class MetaBunch(type): ❶
    def __new__(meta_cls, cls_name, bases, cls_dict): ❷

        defaults = {} ❸

```

```

def __init__(self, **kwargs): ❷
    for name, default in defaults.items(): ❸
        setattr(self, name, kwargs.pop(name, default))
    if kwargs: ❹
        extra = ', '.join(kwargs)
        raise AttributeError(f'No slots left for:
{extra!r}')

def __repr__(self): ❺
    rep = ', '.join(f'{name}={value!r}'
                    for name, default in defaults.items()
                    if (value := getattr(self, name)) !=
default)
    return f'{cls_name}({rep})'

new_dict = dict(__slots__=[], __init__=__init__,
__repr__=__repr__) ❻

for name, value in cls_dict.items(): ❼
    if name.startswith('__') and name.endswith('__'): ❽
        if name in new_dict:
            raise AttributeError(f"Can't set {name!r} in
{cls_name!r}")
        new_dict[name] = value
    else: ❾
        new_dict['__slots__'].append(name)
        defaults[name] = value
    return super().__new__(meta_cls, cls_name, bases,
new_dict) ❿

class Bunch(metaclass=MetaBunch): ⓫
    pass

```

❶ 继承 `type`, 创建新元类。

❷ `__new__` 被当作类方法使用, 但是我们创建的是元类, 因此我喜欢把第一个参数命名为 `meta_cls` (也常用 `mcs`)。余下 3 个参数与直接调用 `type()` 创建类时传入的 3 个参数一样。

❸ `defaults` 用于存放属性名称到默认值的映射。

❹ 这个方法将被注入新类。

⑤ 读取 `defaults`，把相应的实例属性设为从 `kwargs` 中取出的值或默认值。

⑥ 如果 `kwargs` 中有余项，则说明没有位置可以安放关键字参数。我们坚信应该遵从**快速失败**原则，因此不能默不作声，直接忽略多余的项。一种简单有效的方案是从 `kwargs` 中取出一项，尝试在实例上设置，故意触发 `AttributeError`。

⑦ `__repr__` 返回一个类似构造函数调用的字符串（例如 `Point(x=3)`），忽略值为默认值的关键字参数。

⑧ 初始化新类的命名空间。

⑨ 迭代用户定义的类的命名空间。

⑩ 如果 `name` 是带双下划线的名称，就把对应的项复制到新类的命名空间中，除非该项已经存在。这是为了防止用户覆盖 `__init__`、`__repr__`，以及 Python 设置的其他属性，例如 `__qualname__` 和 `__module__`。

⑪ 如果 `name` 不是带双下划线的名称，就追加到 `__slots__` 中，并把对应的值存入 `defaults`。

⑫ 构建并返回新类。

⑬ 提供一个基类，因此用户无须关注 `MetaBunch`。

`MetaBunch` 行之有效，因为它能在调用 `super().__new__` 构建最终的类之前配置 `__slots__`。做元编程时，一定要理解各个操作的执行顺序。下面再做一次求解时间实验，这一次针对元类。

24.8.3 元类求解时间实验

本节在前面的“求解时间实验”的基础上加入元类。builderlib.py 模块与之前一样，不过主脚本现在换成了 evaldemo_meta.py，如示例 24-16 所示。

示例 24-16 evaldemo_meta.py：使用元类做实验

```
#!/usr/bin/env python3

from builderlib import Builder, deco, Descriptor
from metalib import MetaKlass ❶

print('# evaldemo_meta module start')

@deco
class Klass(Builder, metaclass=MetaKlass): ❷
    print('# Klass body')

    attr = Descriptor()

    def __init__(self):
        super().__init__()
        print(f'# Klass.__init__({self!r})')

    def __repr__(self):
        return '<Klass instance>'

def main():
    obj = Klass()
    obj.method_a()
    obj.method_b()
    obj.method_c() ❸
    obj.attr = 999

if __name__ == '__main__':
    main()

print('# evaldemo_meta module end')
```

❶ 从 metalib.py 中导入 MetaKlass（参见示例 24-18）。

❷ 把 Klass 声明为 Builder 的子类以及 MetaKlass 的实例。

❸ 这个方法由 `MetaKlass.__new__` 注入，后面会看到。



出于实验目的，示例 24-16 无视所有理由，在 `Klass` 上运用了 3 种不同的元编程技术：一个装饰器、一个使用 `__init_subclass__` 实现的基类和一个自定义的元类。不要尽信书，在生产代码中不能这么做。这个实验的目标也是观察 3 种技术在类构造过程中的执行顺序。

与前面的求解时间实验一样，这个示例也没做什么实际操作，只是打印一些消息，揭示执行流程。示例 24-17 是 `metalib.py` 的前半部分，余下的内容在示例 24-18 中。

示例 24-17 `metalib.py`: `NosyDict` 类

```
print('% metalib module start')

import collections

class NosyDict(collections.UserDict):
    def __setitem__(self, key, value):
        args = (self, key, value)
        print(f'% NosyDict.__setitem__ {args!r}')
        super().__setitem__(key, value)

    def __repr__(self):
        return '<NosyDict instance>'
```

我编写 `NosyDict` 类是为了覆盖 `__setitem__` 方法，在设置字典中的项时显示 `key` 和 `value`。`MetaKlass` 元类将使用 `NosyDict` 实例存放待构造类的命名空间，进一步揭示 Python 的内部机制。

`metalib.py` 的主要内容是示例 24-18 中定义的元类。这个元类实现了特殊方法 `__prepare__`，这是 Python 只在元类上调用的一个类方法。`__prepare__` 方法是影响创建新类过程最早的机会。



用。

编写元类时，我发现采用以下约定命名特殊方法的参数很有用。

- 实例方法中的 `self` 换成 `cls`，因为得到的实例是类。
- 类方法中的 `cls` 换成 `meta_cls`，因为定义类的是元类。注意，即使没有 `@classmethod` 装饰器，`__new__` 的行为也等同类方法。

示例 24-18 metalib.py: MetaKlass 元类

```
class MetaKlass(type):
    print('% MetaKlass body')

    @classmethod ❶
    def __prepare__(meta_cls, cls_name, bases): ❷
        args = (meta_cls, cls_name, bases)
        print(f'% MetaKlass.__prepare__ {args!r}')
        return NosyDict() ❸

    def __new__(meta_cls, cls_name, bases, cls_dict): ❹
        args = (meta_cls, cls_name, bases, cls_dict)
        print(f'% MetaKlass.__new__ {args!r}')
        def inner_2(self):

            print(f'% MetaKlass.__new__:inner_2({self!r})')

            cls = super().__new__(meta_cls, cls_name, bases,
cls_dict.data) ❺

            cls.method_c = inner_2 ❻

            return cls ❼

        def __repr__(cls): ❽
            cls_name = cls.__name__
            return f"<class {cls_name!r} built by MetaKlass>"

print('% metalib module end')
```

- ❶ `__prepare__` 应声明为类方法。`__prepare__` 不是实例方法，因为当 Python 调用它时待构造的类尚不存在。
- ❷ Python 在元类上调用 `__prepare__` 方法，获取存储待构造的类的命名空间的映射。
- ❸ 返回用作命名空间的 `NosyDict` 实例。
- ❹ `cls_dict` 参数的值是 `__prepare__` 方法返回的 `NosyDict` 实例。
- ❺ `type.__new__` 的最后一个参数必须是真正的字典，因此传入 `NosyDict` 从 `UserDict` 继承的 `data` 属性。
- ❻ 在新创建的类中注入一个方法。
- ❼ 一同往常，`__new__` 方法必须返回刚创建的对象——这里是新创建的类。
- ❽ 在元类中定义 `__repr__` 方法，方便定制类对象的字符串表示形式。

在 Python 3.6 之前，`__prepare__` 方法的主要作用是提供一个 `OrderedDict` 对象，以存放待构造类的属性，这样元类的 `__new__` 方法才能按照属性在用户编写的类定义代码中出现的顺序处理它们。现在 `dict` 能保留插入顺序了，因此很少需要定义 `__prepare__` 方法。24.11 节将介绍 `__prepare__` 方法的一个创意用途。

在 Python 控制台中导入 `metalib.py`，结果没什么让人意外的。注意，以 `%` 开头的行由这个模块输出。

```
>>> import metalib
% metalib module start
% MetaKlass body
% metalib module end
```

导入 evaldemo_meta.py，发生的事情较多，如示例 24-19 所示。

示例 24-19 在控制台中使用 evaldemo_meta.py 做实验

```
>>> import evaldemo_meta
@ builderlib module start
@ Builder body
@ Descriptor body
@ builderlib module end
% metalib module start
% MetaKlass body
% metalib module end
# evaldemo_meta module start ❶
% MetaKlass.__prepare__(<class 'metalib.MetaKlass'>, 'Klass', ❷
                        (<class 'builderlib.Builder'>,))
% NosyDict.__setitem__(<NosyDict instance>, '__module__',
'evaldemo_meta') ❸
% NosyDict.__setitem__(<NosyDict instance>, '__qualname__',
'Klass')
# Klass body
@ Descriptor.__init__(<Descriptor instance>) ❹
% NosyDict.__setitem__(<NosyDict instance>, 'attr', <Descriptor
instance>) ❺
% NosyDict.__setitem__(<NosyDict instance>, '__init__',
                        <function Klass.__init__ at ...>) ❻
% NosyDict.__setitem__(<NosyDict instance>, '__repr__',
                        <function Klass.__repr__ at ...>)
% NosyDict.__setitem__(<NosyDict instance>, '__classcell__', <cell
at ...: empty>)
% MetaKlass.__new__(<class 'metalib.MetaKlass'>, 'Klass',
                    (<class 'builderlib.Builder'>,), <NosyDict
instance>) ❼
@ Descriptor.__set_name__(<Descriptor instance>,
                        <class 'Klass' built by MetaKlass>,
'attr') ❽
@ Builder.__init_subclass__(<class 'Klass' built by MetaKlass>)
@ deco(<class 'Klass' built by MetaKlass>)
# evaldemo_meta module end
```

❶ 这一行前面的几行是导入 builderlib.py 和 metalib.py 的结果。

❷ Python 调用 __prepare__，开始处理 class 语句。

③ 解析类主体之前，Python 把 `__module__` 和 `__qualname__` 添加到待构造类的命名空间中。

④ 创建描述符实例.....

⑤ 绑定给类命名空间中的 `attr`。

⑥ 定义 `__init__` 方法和 `__repr__` 方法，把二者添加到命名空间中。

⑦ 处理完类主体之后，Python 调用 `MetaKlass.__new__`。

⑧ 元类的 `__new__` 方法返回新构造的类之后，依序调用 `__set_name__`、`__init_subclass__` 和装饰器。

如果作为脚本运行 `evaldemo_meta.py`，则 `main()` 函数将被调用，并会发生以下几件事，如示例 24-20 所示。

示例 24-20 把 `evaldemo_meta.py` 当作程序运行

```
$ ./evaldemo_meta.py
[... 省略20行 ...]
@ deco(<class 'Klass' built by MetaKlass>) ❶
@ Builder.__init__(<Klass instance>)
# Klass.__init__(<Klass instance>)
@ SuperA.__init_subclass__:inner_0(<Klass instance>)
@ deco:inner_1(<Klass instance>)
% MetaKlass.__new__:inner_2(<Klass instance>) ❷
@ Descriptor.__set__(<Descriptor instance>, <Klass instance>, 999)
# evaldemo_meta module end
```

❶ 前 21 行（包括这一行）与示例 24-19 相同。

❷ 由 `main` 中的 `obj.method_c()` 触发，`method_c` 由 `MetaKlass.__new__` 注入。

下面回到前面定义的带 `Field` 描述符的 `Checked` 类。这个类实现了运行时类型检查，24.9 节将说明如何使用元类实现同样的功能。

24.9 使用元类实现 `Checked` 类

我不想鼓励过早优化和过度设计，这里会虚构一个适合使用 `__slots__` 重写 `checkedlib.py` 的场景，借此运用元类。本节可以随意跳过。

一点儿背景

我们使用 `__init_subclass__` 实现的 `checkedlib.py` 非常成功，全公司都在用。任何时刻，生产服务器中都有上百万个 `Checked` 子类的实例驻留内存。

经过论证，我们发现使用 `__slots__` 能减少云托管费用，原因有以下两点。

- 内存用量较少，各个 `Checked` 实例无须单独维护 `__dict__`。
- 删除 `__setattr__` 之后，性能更高。当初定义这个方法是为了阻断预期之外的属性，但是实例化以及调用 `Field.__set__` 之前所有的属性设置操作都触发该方法。

下面将要分析的 `metaclass/checkedlib.py` 模块可以直接替代 `initsub/checkedlib.py`。两个模块中的 `doctest` 一模一样，而且使用 `pytest` 编写的 `checkedlib_test.py` 文件也没变。

`checkedlib.py` 中复杂的操作均做了抽象，用户无须直面。下面是用到了这个包的脚本的源码。

```
from checkedlib import Checked

class Movie(Checked):
```

```

    title: str
    year: int
    box_office: float

if __name__ == '__main__':
    movie = Movie(title='The Godfather', year=1972,
box_office=137)
    print(movie)
    print(movie.title)

```

Movie 类的定义非常简短，背后用到了 Field 验证描述符的 3 个实例、一个 `__slots__` 配置、从 Checked 继承的 5 个方法，以及把各方面综合起来的一个元类。checkedlib 对外唯一可见的部分是 Checked 基类。

在图 24-4 中，我在 UML 类图上使用机器和小怪兽图示法画了一些补注，类与实例之间的关系更明显了。

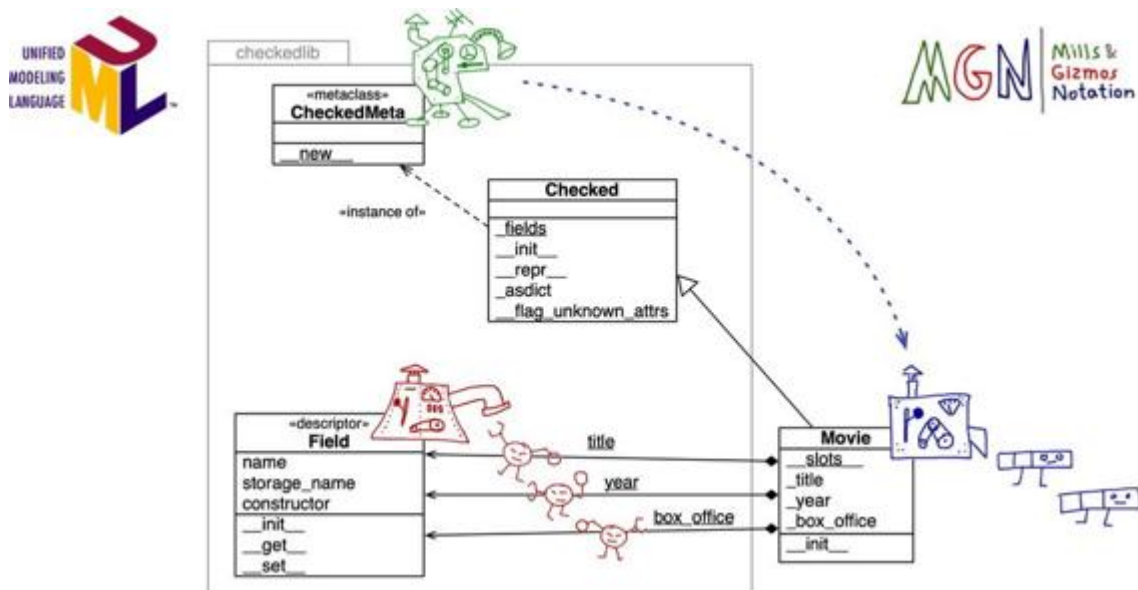


图 24-4：使用 MGN 注解的 UML 类图。CheckedMeta 元机器构建 Movie 机器。Field 机器构建描述符 title、year 和 box_office——这些都是 Movie 的类属性。各个实例的字段数据存储在 Movie 的实例属性 _title、_year 和 _box_office 中。注意 checkedlib 包的边界。Movie 的开发人员无须熟谙 checkedlib.py 的内部机制

例如，使用新版 `checkedlib.py` 的 `Movie` 类既是 `CheckedMeta` 的实例，也是 `Checked` 的子类。另外，`Movie` 的类属性 `title`、`year` 和 `box_office` 是 3 个独立的 `Field` 实例。每个 `Movie` 实例都有自己的 `_title` 属性、`_year` 属性和 `_box_office` 属性，存储相应字段的值。

下面来分析代码，从 `Field` 类开始。示例 24-21 是 `Field` 的源码，标号只说明这一版的改动。

示例 24-1 `metaclass/checkedlib.py`: 有 `storage_name` 属性和 `__get__` 方法的 `Field` 描述符

```
class Field:
    def __init__(self, name: str, constructor: Callable) -> None:
        if not callable(constructor) or constructor is type(None):
            raise TypeError(f'{name!r} type hint must be
callable')
        self.name = name
        self.storage_name = '_' + name ❶
        self.constructor = constructor

    def __get__(self, instance, owner=None):
        if instance is None: ❷
            return self
        return getattr(instance, self.storage_name) ❸

    def __set__(self, instance: Any, value: Any) -> None:
        if value is ...:
            value = self.constructor()
        else:
            try:
                value = self.constructor(value)
            except (TypeError, ValueError) as e:
                type_name = self.constructor.__name__
                msg = f'{value!r} is not compatible with
{self.name}:{type_name}'
                raise TypeError(msg) from e
            setattr(instance, self.storage_name, value) ❹
```

❶ 根据 `name` 参数计算 `storage_name`。

❷ 如果 `__get__` 得到的 `instance` 参数的值是 `None`，则描述符从托管类自身中读取，而不是托管实例。因此，我们返回描述符。

❸ 否则，返回存储在名为 `storage_name` 的属性中的值。

❹ `__set__` 现在使用 `setattr` 设置或更新托管属性。

`Field` 描述符类现在有点儿不一样。在前面的示例中，各个 `Field` 描述符实例把值存在托管实例的同名属性中。例如，在 `Movie` 类中，`title` 描述符把字段的值存在托管实例的 `title` 属性中。因此，`Field` 不能提供 `__get__` 方法。

然而，使用 `__slots__` 之后，对于 `Movie` 这样的类，类属性和实例属性不能同名。每个描述符实例都是一个类属性，因此现在需要让各个实例单独存储属性。我们选择在描述符名称前添加一个 `_`。因此，各个 `Field` 实例都有自己的 `name` 属性和 `storage_name` 属性，可以实现 `Field.__get__`。

示例 24-22 是驱动示例 24-21 的元类的代码。

示例 24-22 metaclass/checkedlib.py: `CheckedMeta` 元类

```
class CheckedMeta(type):

    def __new__(meta_cls, cls_name, bases, cls_dict): ❶
        if '__slots__' not in cls_dict: ❷
            slots = []
            type_hints = cls_dict.get('__annotations__', {}) ❸
            for name, constructor in type_hints.items(): ❹
                field = Field(name, constructor) ❺
                cls_dict[name] = field ❻
                slots.append(field.storage_name) ❼

            cls_dict['__slots__'] = slots ❽

        return super().__new__(
            meta_cls, cls_name, bases, cls_dict) ❾
```

- ❶ `__new__` 是 `CheckedMeta` 唯一实现的方法。
- ❷ 仅当类的 `cls_dict` 中不含 `__slots__` 时才增强类的功能。如果存在 `__slots__`，则假设它是 `Checked` 基类，而不是用户定义的子类，并原封不动构建类。
- ❸ 在前面的示例中，为了获取类型提示，使用的是 `typing.get_type_hints`，但是该函数的第一个参数必须是现有的类。现在，我们配置的类尚不存在，因此需要直接从 `cls_dict` 中获取 `__annotations__`。`cls_dict` 是待构造类的命名空间，Python 把它作为最后一个参数传给元类的 `__new__` 方法。
- ❹ 迭代 `type_hints`.....
- ❺ 为每个带注解的属性构建一个 `Field` 实例.....
- ❻ 使用 `Field` 实例覆盖 `cls_dict` 中相应的项.....
- ❼ 把字段的 `storage_name` 追加到一个列表中.....
- ❽ 用于填充 `cls_dict`（待构造类的命名空间）中的 `__slots__` 项。
- ❾ 最后，调用 `super().__new__`。

`metaclass/checkedlib.py` 的最后一部分是 `Checked` 基类。这个库的用户通过子类化该基类增强自己定义的类，例如 `Movie`。

这一版 `Checked` 类，除了以下 3 处改动之外，其他代码与 `initsub/checkedlib.py` 中一样（参见示例 24-5 和示例 24-6）。

01. 添加一个空的 `__slots__` 属性，告诉 `CheckedMeta.__new__`，这个类不需要特殊处理。

02. 删除 `__init_subclass__`。相关工作现在交给

`CheckedMeta.__new__`。

03. 删除 `__setattr__`。为用户定义的类添加 `__slots__` 之后，无法设置未声明的属性了，因此这个方法也就多余了。

示例 24-23 是最终版 `Checked` 类的完整代码。

示例 24-23 metaclass/checkedlib.py: `Checked` 基类

```
class Checked(metaclass=CheckedMeta):
    __slots__ = () # 跳过CheckedMeta.__new__的处理

    @classmethod
    def _fields(cls) -> dict[str, type]:
        return get_type_hints(cls)

    def __init__(self, **kwargs: Any) -> None:
        for name in self._fields():
            value = kwargs.pop(name, ...)
            setattr(self, name, value)
        if kwargs:
            self.__flag_unknown_attrs(*kwargs)

    def __flag_unknown_attrs(self, *names: str) -> NoReturn:
        plural = 's' if len(names) > 1 else ''
        extra = ', '.join(f'{name!r}' for name in names)
        cls_name = repr(self.__class__.__name__)
        raise AttributeError(f'{cls_name} object has no attribute{plural} {extra}')

    def _asdict(self) -> dict[str, Any]:
        return {
            name: getattr(self, name)
            for name, attr in self.__class__.__dict__.items()
            if isinstance(attr, Field)
        }

    def __repr__(self) -> str:
        kwargs = ', '.join(
            f'{key}={value!r}' for key, value in
            self._asdict().items()
        )
        return f'{self.__class__.__name__}({kwargs})'
```

带有验证描述符的类构建器的第三次实现到此结束。

24.10 节将讨论与元类有关的一些一般性问题。

24.10 元类的实际运用

元类功能强大，想正确使用却不容易。着手实现元类之前，请考虑以下几点。

24.10.1 可简化或代替元类的现代功能

随着新语言功能的推出，元类的多个常见用途已显多余。

类装饰器

比元类更易于理解，而且导致基类与元类产生冲突的可能性更小。

```
__set__ name__
```

无须自定义元类逻辑就能自动设置描述符的名称。¹⁴

¹⁴本书第 1 版使用元类定义了一版更高级的 `LineItem` 类，仅仅是为了设置储存属性的名称。详见本书第 1 版代码中 `bulkfood` 示例的元类。

```
__init__ subclass__
```

提供一种自定义类创建过程的方式，对终端用户透明，而且比装饰器更简单，但是遇到复杂的类层次结构可能产生冲突。

内置的 `dict` 保留键的插入顺序

不使用 `__prepare__` 的首要原因。以前定义 `__prepare__` 是为了使用 `OrderedDict` 存储待构造类的命名空间。Python 只在元类上调用 `__prepare__`，因此如果想按照源码中的出现顺序处理类命名空间，那么在 Python 3.6 之前只能使用元类。

截至 2021 年，所有积极维护的 CPython 版本都支持以上全部功能。

我始终推荐使用这些功能，因为我见过业内很多人无意中把问题复杂化了，而元类就是罪魁祸首。

24.10.2 元类是稳定的语言功能

元类与所谓的“新式类”、描述符和特性一起在 2002 年发布的 Python 2.2 中引入。

令人吃惊的是，最初由 Alex Martelli 在 2002 年 7 月发布的 `MetaBunch` 示例，在 Python 3.9 中还能正常运行，唯一需要改动的地方是指定使用元类的方式：在 Python 3 中，使用的句法是 `class Bunch(metaclass=MetaBunch) :`。

24.10.1 节提到的新功能没有破坏使用元类的现有代码。但是，以前使用元类的代码通常可以借助新功能简化，尤其是不再支持 Python 3.6 之前版本（不再维护）的代码。

24.10.3 一个类只能有一个元类

如果声明的类涉及两个或更多元类，则你会看到以下令人不解的错误消息。

```
TypeError: metaclass conflict: the metaclass of a derived class
must be a (non-strict) subclass of the metaclasses of all its
bases
```

即使不涉及多重继承，也有可能见到这条消息。例如，下面的声明就会导致这样的 `TypeError`。

```
class Record(abc.ABC, metaclass=PersistentMeta):
    pass
```

我们知道, `abc.ABC` 是 `abc.ABCMeta` 元类的实例。如果 `Persistent` 元类不是 `abc.ABCMeta` 的子类, 就会出现元类冲突。

处理这个错误的方式有两种。

- 寻找其他方式解决手头遇到的问题, 至少把用到的元类去掉一个。
- 自己编写 `PersistentABCMeta` 元类, 作为 `abc.ABCMeta` 和 `PersistentMeta` 的子类 (利用多重继承), 仅以 `PersistentABCMeta` 为 `Record` 的元类。¹⁵

¹⁵如果厘不清元类对多重继承的影响, 实属正常。我也宁愿远离这种方案。



可以想象, 有些人为了赶进度, 会在截止时间之前使用两个基元类。根据我的经验, 元类编程的时间总是比预期的要长, 如果在截止时间之前匆匆使用这种方案, 那么就会有一定风险。为了赶进度而这么做, 代码中难免埋下难以察觉的 bug。即使有意避免已知的 bug, 也应把这种方案视为技术债务, 因为写出的代码难以理解和维护。

24.10.4 元类应作为实现细节

除了 `type`, 整个 Python 3.9 标准库中仅有 6 个元类。最为人熟知的元类应该是 `abc.ABCMeta`、`typing.NamedTupleMeta` 和 `enum.EnumMeta`。但是, 在用户的代码中, 不应该显式使用其中任何一个。应把元类当作实现细节。

尽管使用元类可以做一些古怪的元编程, 但是最好严守“最小惊讶”原则, 让大多数用户真正把元类当作实现细节。¹⁶

¹⁶我常年以编写 Django 代码为生, 多年后才开始研究 Django 的模型字段是如何实现的。直到那时我才听说描述符和元类。

近些年，Python 标准库中的某些元类换成了其他机制，而相应包的公开 API 没有变化。为了不让 API 过时，最简单的方法是提供一个常规的类供用户子类化，让用户使用元类提供的功能——就像前面的示例中那样做。

结束对类元编程的讨论之前，我想与你分享一个元类小示例。这是我在研究如何更新本章时发现的，酷炫十足。

24.11 使用元类的 `__prepare__` 方法实现新颖的构思

针对本书第 2 版更新本章时，我想找一个简单又具启发性的示例，代替从 Python 3.6 开始不再需要元类的 `LineItem` 示例。

João S. O. Bueno（在巴西 Python 社区中人称 JS）给我提供的想法最简单且最有趣。利用他的想法可以创建一个类，自动生成多个常量。

```
>>> class Flavor(AutoConst):
...     banana
...     coconut
...     vanilla
...
>>> Flavor.vanilla
2
>>> Flavor.banana, Flavor.coconut
(0, 1)
```

是的，就是这样用的！这其实是 `autoconst_demo.py` 中的 `doctest`。

下面是对用户友好的 `AutoConst` 基类，以及背后的元类在 `autoconst.py` 中的实现。

```
class AutoConstMeta(type):
    def __prepare__(name, bases, **kwargs):
```

```
        return WilyDict()

class AutoConst(metaclass=AutoConstMeta):
    pass
```

就这么简单。

显然，神奇之处隐藏在 WilyDict 中。

Python 处理用户定义的类的命名空间和读取 banana 时，在 `__prepare__` 提供的映射（一个 WilyDict 实例）中查找名称。WilyDict 实现了 `__missing__` 方法（详见 3.5.2 节）。最初，WilyDict 实例没有 'banana' 键，因此触发 `__missing__` 方法，即时创建 'banana' 键，把值设为 0，返回该值。Python 得到回应后，尝试获取 'coconut'。WilyDict 立即添加相应的项，返回值 1。获取 'vanilla' 也是同样的过程，对应的值是 2。

我们之前已经见过 `__prepare__` 和 `__missing__`。JS 真正的创意之处是把二者结合起来使用。

WilyDict 的源码也在 `autoconst.py` 中，如下所示。

```
class WilyDict(dict):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__next_value = 0

    def __missing__(self, key):
        if key.startswith('__') and key.endswith('__'):
            raise KeyError(key)
        self[key] = value = self.__next_value
        self.__next_value += 1
        return value
```

实验过程中，我发现 Python 会在待构造类的命名空间中查找 `__name__`，导致 WilyDict 添加 `__name__` 项，递增 `__next_value` 的值。因此，我在 `__missing__` 方法中增加了一个 if 语句，当查找的键是带双下划线的属性时抛出 `KeyError`。

autoconst.py 包要求你精通 Python 动态构建类的机制，也演示了这个机制的强大之处。

我还为 AutoConstMeta 和 AutoConst 添加了更多功能，那段时光让人回味。我所做的实验就不与你分享了，那种欢快留给你自己吧！请你自己体验 JS 新颖的构思。

下面给你一些思路。

- 通过值获取常量名称。例如，`Flavor[2]` 返回 `'vanilla'`。为此，需要在 `AutoConstMeta` 中实现 `__getitem__` 方法。从 Python 3.9 开始，可以在 `AutoConst` 中实现 `__class_getitem__` 方法。
- 在元类中实现 `__iter__` 方法，支持迭代类。我会让 `__iter__` 方法产出 `(name, value)` 形式的常量。
- 实现全新的 Enum 变体。这是一项艰巨的任务，因为 `enum` 包充满陷阱，包括有数百行代码的 `EnumMeta` 元类和一个重要的 `__prepare__` 方法。

尽情享受吧！



特殊方法 `__class_getitem__` 在 Python 3.9 中添加，目的是支持泛型，包含在“PEP 585—Type Hinting Generics In Standard Collections”中。得益于 `__class_getitem__`，Python 核心开发人员不用再为内置类型编写新的元类，实现编写 `list[int]` 之类的泛化类型提示所需的 `__getitem__` 方法。这个功能看似作用不大，却为元类打开了一片新天地：可实现在类层级上操作的运算符和其他特殊方法，例如把类自身变成可迭代对象，就像 `Enum` 的子类一样。

24.12 小结

元类，以及类装饰器和 `__init_subclass__` 的用途如下：

- 注册子类；
- 验证子类结构；
- 把装饰器一次性应用到多个方法上；
- 序列化对象；
- 映射对象关系；
- 持久存储对象；
- 在类层级实现特殊方法；
- 实现其他语言特有的功能，例如性状和面向方面程序设计。

某些情况下，使用类元编程还能把运行时重复执行的任务放到导入时执行，解决性能问题。

最后，来回顾一下 Alex Martelli 在 13.5 节的“水禽和抽象基类”附注栏中给出的建议。

此外，**不要**在生产代码中定义抽象基类（或元类）……如果很想这样做，我打赌可能是因为你“找碴儿”。刚拿到新工具的人都有大干一场的冲动。如果能避开这些深奥的概念，那么你（以及未来的代码维护人员）的生活将更愉快，因为代码会变得简洁明了。

我认为，Martelli 的建议不仅适用于抽象基类和元类，也适用于类层次结构、运算符重载、函数装饰器、描述符、类装饰器和使用 `__init_subclass__` 实现的类构建器。

这些强大的工具存在的目的是为库和框架开发提供支持。应用程序自然应该**使用**Python 标准库或外部包提供的工具。但是，在应用程序代码中自己**实现**这些工具，往往表明抽象不够成熟。

好的框架是扩展而来的，不是发明出来的。 ¹⁷

——David Heinemeier Hansson
Ruby on Rails 创造者

¹⁷这句话经常被人引用。我发现 2005 年 DHH 在一篇博客文章中对这句话的引用比较早。

24.13 本章小结

本章首先概述了类对象的属性，例如 `__qualname__` 方法和 `__subclasses__()` 方法。我们了解到，使用内置函数 `type` 可以在运行时构造类。

然后介绍了特殊方法 `__init_subclass__`。我们定义了第 1 版 `Checked` 基类，把用户定义的子类中的属性类型提示替换为 `Field` 实例，借助构造函数在运行时限定属性的类型。

我们还使用 `@checked` 类装饰器实现了同样的功能。与 `__init_subclass__` 类似，类装饰器也可以为用户定义的类添加功能。我们发现，`__init_subclass__` 和类装饰器均不能动态配置 `__slots__`，因为它们只在创建类之后发挥作用。

我们通过实验区分了“导入时”和“运行时”：打印输出模块、描述符、类装饰器和 `__init_subclass__` 在 Python 代码中的执行顺序。

随后讨论了元类。首先简要说明作为元类的 `type`，以及用户定义的元类如何通过 `__new__` 方法定制要构建的类。然后自己定义了第一个元类，即使用 `__slots__` 的经典 `MetaBunch` 示例。最后又做了一个求解时间实验，结果表明元类的 `__prepare__` 方法和 `__new__` 方法均在 `__init_subclass__` 和类装饰器之前调用，这为深入定制类提供了机会。

接下来定义了 `Checked` 类构建器的第 3 版。用到了 `Field` 描述符，还自己配置了 `__slots__`。之后为元类的实际运用提供了几个一般性考虑事项。

最后介绍了 João S. O. Bueno 提出的 `AutoConst` 构思：巧妙使用元类的 `__prepare__` 方法，返回一个实现了 `__missing__` 方法的映

射。autoconst.py 的代码不到 20 行，却展现了各项 Python 元编程技术综合在一起的强大功能。

我没有发现任何一门语言能像 Python 这样，对初学者来说简单、对专业人士来说实用、对黑客来说令人振奋。感谢 Guido van Rossum 和 Python 背后的每一个人。

24.14 延伸阅读

本书技术审校 Caleb Hattingh 编写的 autoslot 包提供了一个元类，通过审查 `__init__` 的字节码，找出所有对 `self` 的属性赋值，自动为用户定义的类创建 `__slots__` 属性。这个包用处很大，也值得研究：autoslot.py 只有 74 行代码，其中 20 行是注释，用于说明最难解决的部分。

为了深入学习本章讨论的知识，一定要阅读《Python 语言参考手册》中的 3.3.3 节“自定义类创建”，该节涵盖 `__init_subclass__` 和元类。Python 标准库文档中“Built-in Functions”一章的 `type` 类的文档，以及“Built-in Types”一章的“4.13. Special Attributes”一节也是必读的。

在 Python 标准库文档中，`types` 模块的文档说明了 Python 3.3 引入的可以简化类元编程的两个新函数：`types.new_class` 和 `types.prepare_class`。

类装饰器的规范是“PEP 3129—Class Decorators”，作者是 Collin Winter，参考实现由 Jack Diederich 提供。Jack Diederich 在 PyCon 2009 大会上做了一场题为“Class Decorators: Radically Simple”的演讲，对这个功能做了简单介绍。除了 `@dataclass`，Python 标准库中还有一个有趣的类装饰器（而且更简单），即 `functools.total_ordering`，可生成比较对象的特殊方法。

元类的主要参考资料是引入特殊方法 `__prepare__` 的“PEP 3115—Metaclasses in Python 3000”。

Python in a Nutshell, 3rd ed. (Alex Martelli、Anna Ravenscroft 和 Steve Holden 著) 一书具有权威性，但是写于“PEP 487—Simpler customization of class creation”之前。该书中主要的元类示例，即 `MetaBunch`，现在仍不过时，因为没有更简单的机制出现。

《Effective Python：编写高质量 Python 代码的 90 个有效方法（原书第 2 版）》一书中有几个构建类的示例用到了最新的技术，包括元类。

如果想了解 Python 类元编程的源起，推荐阅读 Guido van Rossum 在 2003 年发表的论文“Unifying types and classes in Python 2.2”。这篇论文也适用于现代的 Python，谈到了后来称为“新式类”的语义（Python 3 默认的语义），包括描述符和元类。Guido 这篇论文的参考文献之一是 *Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming* (Ira R. Forman 和 Scott H. Danforth 著)。他在亚马逊上给该书打了 5 颗星，还写了如下评价。

这本书促成 Python 2.2 实现了元类

可惜，这本书已经绝版了。Python 通过 `super()` 函数实现了协作式多重继承，谈到这方面的难题时，我总会提到这本书。据我所知，这本书是这方面最好的教程。¹⁸

¹⁸我买了一本二手书，发现它很难读懂。

如果对元编程着迷，那么你可能希望 Python 提供基本的元编程功能：Lisp 语言家族提出，而后被 Elixir 和 Rust 采用的句法宏。与 C 语言中原始的代码置换宏相比，句法宏更强大，出错的可能性更低。句法宏是一种特殊的函数，在编译步骤之前使用自定义的句法把源码改写成标准代码，开发人员无须改动编译器就能引入新的语言结构。与运算符重载一样，句法宏也可能被滥用。但是，只要社区能理解并管控缺点，利用句法宏就能实现强大且对用户友好的抽象，例如 DSL

(Domain-Specific Language, 领域特定语言)。2020 年 9 月, Python 核心开发人员 Mark Shannon 发布了“PEP 638—Syntactic Macros”, 提议实现句法宏。一年后, PEP 638 仍在草案阶段, 讨论热度已经消失。显然, 这不是 Python 核心开发人员的首要任务。我希望看到有人进一步讨论 PEP 638, 并且最终获批实现。借助句法宏, Python 社区可以在核心语言做出改动之前先行试验有争议的新特性, 例如海象运算符 (PEP 572)、模式匹配 (PEP 634) 和求解类型提示的替代规则 (PEP 563 和 PEP 649)。目前, 可以使用 MacroPy 包体验句法宏。

杂谈

这是本书最后一篇杂谈了。首先我要从 Brian Harvey 与 Matthew Wright 合写的著作中引述一大段文字。Harvey 和 Wright 是加州大学 (伯克利分校和圣巴巴拉分校) 的计算机科学教授, 二人在合著的 *Simply Scheme: Introducing Computer Science* 一书中写道:

计算机科学的教学方式分成两个流派, 可以描述如下。

01. **保守派** 计算机程序已经变得极其大而复杂, 超过了人类思维所能承载的限度。因此, 计算机科学教育的任务是训练平庸的程序员, 这样 500 个人合作便能开发出恰好满足需求的程序。
02. **激进派** 计算机程序已经变得极其大而复杂, 超过了人类思维所能承载的限度。因此, 计算机科学教育的任务是教人如何拓展思维, 打破常规, 学习以更广博、更强大和更灵活的方式思考, 让思维超越程序。编程思想的各个方面在程序中必会得到充分体现。

——Brian Harvey 和 Matthew Wright
Simply Scheme 前言 ¹⁹

这是 Harvey 和 Wright 对计算机科学教育的夸张描述，不过同样适用于编程语言的设计。现在，你应该能猜到，我赞成“激进派”，我认为 Python 也是以这种态度设计的。

为了稳扎稳打，Java 从一开始使用的就是存取方法，而且众多 Java IDE 都提供了生成读值方法和设值方法的快捷键，与此相比，特性算是一大进步。特性的主要优点是，一开始编写程序时可以先将属性设为公开的（遵照“KISS 原则”），因为公开的属性无须大幅改动，随时都能变成特性。不过，描述符更进一步，提供了去除存取方法中逻辑重复的机制。这种机制特别有效，因此基本的 Python 结构在背后也用到了描述符。

另一个强大的想法是，把函数当作一等对象，这为高阶函数铺平了道路。描述符和高阶函数合在一起实现，使得函数和方法的统一成为可能。函数的 `__get__` 方法能把实例绑定到 `self` 参数上，即时生成方法对象。这种做法相当优雅。²⁰

最后，Python 中的类也是一等对象。作为一门对初学者友好的语言，Python 能提供类构建器、类装饰器，以及允许用户定义功能完整的元类，这些强大的抽象真是太棒了。最棒的是，这些高级功能没有拖累日常编程（其实无形中提供了帮助）。Django 和 SQLAlchemy 等框架用起来这么方便，发展得这么成功，很大程度上归功于元类。近些年，Python 中的类元编程越来越简单，至少常见场景是变简单了。最优秀的语言功能能让所有人受益，而 Python 用户甚至不知道它们的存在。当然，用户可以学习，去创建下一个伟大的库。

期待你对 Python 社区和生态系统的贡献！

¹⁹ *Simply Scheme* (Brian Harvey 和 Matthew Wright 著，MIT 出版社，1999)，第 xvii 页。加州大学伯克利分校的网站中有此书全文。

²⁰ *Machine Beauty: Elegance and the Heart of Technology* (David Gelernter 著) 一书对工程作品（从桥梁到软件）的优雅和美学做了阐述。

结语

Python 是给法定成年人使用的语言。

——Alan Runyan
Plone 联合创始人

Alan 的精辟定义道出了 Python 最好的特质之一：它不妨碍你，让你做你该做的事。这也意味着，它不会给你提供工具，让你限制其他人能对你的代码和代码所构建的对象做什么。

Python 已经 30 岁了，受欢迎程度依然不减。当然，Python 不完美。对我来说，最无法接受的是，Python 在标准库中混用驼峰式和蛇底式，或者直接把单词连在一起。但是，语言的定义和标准库只是生态系统的一部分。用户和贡献者组成的社区才是 Python 生态系统最重要的部分。

有一个例子可以说明社区的好处。撰写本书第 1 版 `asyncio` 包相关的内容时，我感到很沮丧，因为那个包的 API 有很多函数，其中有些是协程，可是协程必须使用 `yield from` 调用（现在使用 `await`），而常规的函数不能这么做。虽然这在 `asyncio` 包的文档中有说明，但是有时阅读几段文字之后才能确定某个函数是不是协程。因此，我给 `python-tulip` 邮件列表发了一则消息，题为“Proposal: make coroutines stand out in the asyncio docs”。`asyncio` 包的核心开发者 Victor Stinner、`aiohttp` 包的主要作者 Andrew Svetlov、Tornado 的首席开发者 Ben Darnell，以及 Twisted 的发明者 Glyph Lefkowitz 都加入了讨论。Darnell 提出了一种方案，Alexander Shorin 解说了如何在 Sphinx 中实现，Stinner 添加了所需的配置和标记。我提出这个问题不到 12 小时，整个 `asyncio` 包的线上文档都更新了，添加了今天你所看到的“coroutine”标签。

在排外的社区中绝不可能发生这种事。任何人都能加入 python-tulip 邮件列表，我编写那个提议之前只发布过几次消息而已。这件事表明，Python 社区特别开放，广纳新想法和新成员。Guido van Rossum 以前经常出现在 python-tulip 邮件列表中，时常回答一些基本的问题。

还有一个例子能说明 Python 的开放：Python 软件基金会（Python Software Foundation, PSF）一直在努力提升 Python 社区的多样性，而且已经取得一些令人欣喜的成果。2013—2014 年，PSF 董事会首次选出了女性董事——Jessica McKellar 和 Lynn Root。2015 年在蒙特利尔举办的 PyCon North America 大会（Diana Clarke 主持），约 1/3 的演讲者是女性。PyLadies 运动广布全球，巴西的 PyLadies 分会数量众多——真是让人感到骄傲。

如果你是 Python 程序员，但尚未加入社区，建议你尽快加入。寻找你所在地区的 PyLadies 或 Python 用户组（Python Users Group, PUG）。如果没有，那就创办一个。任何地方都有人使用 Python，你并不孤独。如果可能的话，参加别处举办的会议。参加线上活动也可以。来参加 Python Brasil 大会吧，多年以来这个大会都有来自世界各地的演讲者。与 Python 程序员面对面交流有很多好处，例如获得工作机会和真正的友谊。

我知道，如果没有多年来在 Python 社区中结交的朋友的帮助，我不可能写出本书。

我的父亲 Jairo Ramalho 经常说，“Só erra quem trabalha”。这是葡萄牙语，意思是“只有真正做事的人才会犯错”。这个建议很棒，能让你不再害怕失败，迈步向前。撰写本书的过程中，我难免犯错。审校、编辑和抢读版的读者帮我找出了很多错误。第 1 版抢读版刚发布几个小时，就有一位读者在本书的勘误页面报告了拼写错误。其他读者报告了更多错误，我的朋友还直接联系我，提供建议和更正。我写完本书后，O'Reilly 的文字编辑会在出版过程中找出其他错误。如果还有任何错误和词不达意的表述，责任都在我，在此向各位读者致歉。

终于写完本书的第 2 版了，我特别高兴，无论有没有错误，都十分感激一路上给我帮助的每一个人。

希望很快就能在线上活动中见到你。如果见到我，请过来打声招呼。

延伸阅读

在本书的最后，我要介绍一些“Python 风格”的参考资料——这正是本书试图解决的主要问题。

Brandon Rhodes 是一位出色的 Python 教师，他的演讲“A Python Aesthetic: Beauty and Why I Python”很精彩，从标题中使用的 Unicode 字符 U+00C6 (LATIN CAPITAL LETTER AE) 开始谈起。另一位出色的教师 Raymond Hettinger，在 2013 年的 PyCon US 大会上谈了 Python 之美：“Transforming Code into Beautiful, Idiomatic Python”。

Ian Lee 在 Python-ideas 邮件列表中发起的“Evolution of Style Guides”话题值得一读。Lee 是 `pep8` 包的维护者，这个包的作用是检查 Python 源码是否符合 PEP 8 的要求。检查本书中的代码时，我用的是 `flake8`。这个包融合了 `pep8`、`pyflakes` 和 Ned Batchelder 开发的复杂度插件 McCabe。

除了 PEP 8，谷歌的 Python 风格指南和 Pycocoo 风格指南也有很大的影响。Pycocoo 团队为我们开发了 Flask、Sphinx、Jinja 2 和其他优秀的 Python 库。

The Hitchhiker's Guide to Python! 由多人维护，教你如何编写符合 Python 风格的代码。为这个项目贡献最多内容的是 Kenneth Reitz，他因开发特别符合 Python 风格的 `requests` 包而被社区视为英雄。David Goodger 在 2008 年举办的 PyCon US 大会上办了一场教学活动，题为“Code Like a Pythonista: Idiomatic Python”。如果打印出来，这个教程的教案有 30 页。

`reStructuredText` 和 `docutils` 都是 Goodger 的作品。这两个工具是 Sphinx 的基础。Sphinx 是优秀的 Python 文档系统（顺便提一下，MongoDB 和很多其他项目的官方文档系统都是 Sphinx）。

Martijn Faassen 直接回答了“什么是 Python 风格”这个问题。python-list 邮件列表中也有一个相同标题的话题。Martijn 的文章是 2005 年写的，那个话题是 2003 年讨论的，不过 Python 风格的思想没怎么变化，Python 语言本身也是如此。“Pythonic way to sum n-th list element?”话题对 Python 风格做了深入讨论，第 12 章的“杂谈”中有大量引用。

“PEP 3099—Things that will Not Change in Python 3000”解释了为何经过 Python 3 的大幅度调整之后，许多东西仍是现在的样子。长久以来，Python 3 有个昵称——Python 3000，不过诞生时间早了几个世纪，让一些人失望了。PEP 3099 的作者是 Georg Brandl，他收集了仁慈的“独裁者”（Guido van Rossum）的很多观点。“Python Essays”页面列出了 Guido 自己写的多篇论文。

作者简介

卢西亚诺·拉马略 (Luciano Ramalho) 在 1995 年 Netscape 首次公开募股以前就是一名 Web 开发人员了，先后用过 Perl 和 Java，1998 年开始使用 Python。他于 2015 年加入 Thoughtworks，在圣保罗办事处担任首席咨询师。他经常在世界各地举办的 Python 活动中做主题演讲、讲座和辅导，也参加过 Go 和 Elixir 技术大会，主要关注语言设计话题。拉马略是 Python 软件基金会成员，还是巴西第一个众创空间 Garoa Hacker Clube 的联合创始人。

关于封面

本书封面上的动物是纳马沙蜥（学名：Pedioplanis namaquensis），分布于纳米比亚全境的干旱稀树草原和半荒漠地区。

纳马沙蜥通体黑色，身披 4 条白纹；四肢棕色，带白点；腹白，尾长呈红棕色。它们是运动速度最快的日间活动蜥蜴之一，以小昆虫为食，栖息在草木稀疏的沙砾平地。在 11 月，雌性产蛋 3~5 枚。冬季，它们在灌木丛边挖穴休眠。

纳马沙蜥目前的濒危等级是“无危”。O'Reilly 出版的图书，封面上的很多动物濒临灭绝。这些动物是地球的至宝。

封面图片由 Karen Montgomery 根据 Wood 的 *Natural History* 中的一幅黑白版画绘制。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
 - 微博 @图灵社区：电子书和好文章的消息
 - 微博 @图灵新知：图灵教育的科普小组
 - 微信 图灵访谈：ituring_interview，讲述码农精彩人生
 - 微信 图灵教育：turingbooks
-