O'REILLY®

Python编程之美
最佳实践指南

The Hitchhiker's Guide to Python

[美] Kenneth Reitz　Tanya Schlusser　著

夏永锋　廖邦杰　译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书是Python用户的一本百科式学习指南，由Python社区数百名成员协作奉献。

全书内容分为三大部分。第1部分是关于如何配置和使用Python编辑工具的；第2部分深入讲解地道Python风格的代码范例；第3部分研究Python社区常用的一些代码库。

本书适合有一定Python基础的人员学习，帮助你迅速从小工修炼成专家，编写出高质量的代码！

版权贸易合同登记号 图字：01-2016-9669

图书在版编目（CIP）数据

Python 编程之美：最佳实践指南 /（美）肯尼思·赖茨（Kenneth Reitz），（美）坦尼娅·胥卢瑟（Tanya Schlusser）著；夏永锋，廖邦杰译 . —北京：电子工业出版社，2018.9
书名原文：The Hitchhiker's Guide to Python
ISBN 978-7-121-34757-3
Ⅰ . ① P… Ⅱ . ①肯… ②坦… ③夏… ④廖… Ⅲ . ①软件工具－程序设计 Ⅳ . ① TP311.561
中国版本图书馆 CIP 数据核字 (2018) 第 161121 号

策划编辑：刘　皎
责任编辑：汪达文
封面设计：Randy Comer　张　健
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787×980　1/16　印张：20　字数：460千字
版　　次：2018年9月第1版
印　　次：2018年9月第1次印刷
定　　价：89.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始,O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来,而我们关注真正重要的技术趋势——通过放大那些"细微的信号"来刺激社会对新科技的应用。作为技术社区中活跃的参与者,O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的"动物书";创建第一个商业网站(GNN);组织了影响深远的开放源代码峰会,以至于开源软件运动以此命名;创立了Make杂志,从而成为DIY革命的主要先锋;公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖,共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择,O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程,每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

"O'Reilly Radar博客有口皆碑。"

——Wired

"O'Reilly凭借一系列(真希望当初我也想到了)非凡想法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、最广阔的视野并且切实地按照Yogi Berra的建议去做了:'如果你在路上遇到岔路口,走小路(岔路)。'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

推荐序一

我从 2011 年开始学习和使用 Python，印象里那个时候掌握 Python 语言基本是 BAT 等大型企业岗位要求里面的附加条件。由于 Python 语法简单很容易上手，再加上极好的代码可读性、丰富和强大的数据结构和内置标准库、良好的社区生态能极大地提升开发效率等优势，我能清晰感受到这些年 Python 越来越受到国内企业和开发者的青睐，而据我所知国内外知名的大型互联网公司或多或少都在使用 Python，甚至很多公司的主要技术栈是 Python。

作为一个 Python 开发者，我非常幸运。因为本书的社区开源项目“The Hitchhiker’s Guide to Python!”也是 2011 年由 Kenneth Reitz 发起的，虽然我 2012 年才知道这个学习指南，但是必须承认书中的内容对我学习和实践 Python 有非常大的帮助，直到现在我还是会时常翻阅本书。

和我学习 Python 时相比，现在的环境实在好得太多了：有很多 Python 书籍、网上资源、国外的视频课程等，但是其中由一线开发者编写、与实践结合、代码能称为“Pythonic”的书籍却凤毛麟角，而这本《Python 编程之美：最佳实践指南》就是这样的一本书。它是一本 Python 安装、配置和使用的最佳实践手册，涉及开发环境和部署、编写符合 Pythonic 品味的代码、各应用场景下主流的 Python 解决方案、学习资源推荐等多个方面，有足够的广度也有合适的深度。这是我非常喜欢的一种学习指南的写作风格，如果有最佳实践会直接告诉你应该这么做，不应该怎么做，或者会告诉你这个应用场景下都有哪些技术选型，它们各自的优缺点及选择建议。这本书会告诉你怎么用，但是更多的是引路，如果你希望深入学习还是要自己搜索相关资源。

我认为其中最有价值的内容之一是教你编写高质量 Python 代码这部分，书中有非常多的场景对应实践及建议，尤其是作者对“Python 之禅”的理解非常透彻，作者的理解和经验对于初学者养成良好的编程习惯，以及培养“Pythonic”品味是非常有意义的。

还有一个章节是教你阅读高质量的代码。成为一名优秀程序员的秘诀之一就是阅读、理解其他优质项目中好的代码，吸收并应用到自己的工作中来。书中列的几个开源项目都是质量非常高的开源项目，在这个章节中作者把自己阅读代码的思路、方法和技巧分享出来，告诉你为什么这么设计，这么做的优点是什么等，而不是只看代码。另外也清晰

地把项目结构图呈现出来，有极强的学习价值。

对初学者来说这是一本极佳的学习指南，但是也建议每个 Python 开发者都读这本书！最后我也期望国内会有越来越多的一线开发者写技术图书，分享自己使用 Python 的相关经验和技巧，让优质的 Python 书籍越来越多！

董伟明

豆瓣高级产品开发工程师、《Python Web 开发实战》作者

推荐序二

这本书的关键词是“实践”和“指南”。

我曾经被多次拷问：

- “已经学习了 Python 基础知识，后续应该做什么呢？”
- “Python 中有那么多模块和包，我应该选择使用哪一个呢？”
- “什么样的代码才是所谓‘优雅’的呢？”
- “怎样提高自己的编程水平？”
- “我不会阅读代码，怎么办？”
- ……

对于初学者而言，遇到上述问题很正常，关键是怎么解决问题。

《Python 编程之美：最佳实践指南》一书帮助开发者破解了一些常见疑惑，提供了具有实践价值的指南。

编程，是一个实践性很强的工作。学习了某种语言的基本知识之后，能够写出一些程序，但是否写得好，则是另外一个话题了，况且，实践中也很难确立“好”的标准，如何才能写得“更好”？

一要多写。业精于勤，荒于嬉。不论是各种研究结果还是个人经验，都认同这样的结论：实践性强的技能都要不断地，甚至是重复地做。所以，我经常唠叨“（代）码不离手”，有的人理解、有的人嫌弃、有的人漠视，至少我观察到身边的这三类人在几年之后，他们的“某些属性”的“值”差别不小。

二要思考。行成于思，毁于随。韩愈老先生的两句话，放到这里都很适合。有开发者，写代码多年，但习惯以“时间紧、任务重、给钱少”为思考原点，拒绝主动优化代码——通常老板也不给时间优化代码，你手不在敲代码，他就在亏钱。结果，本来是以智力活动为主的“开发者”，在主客观因素的裹挟下成了以机械操作为主的“搬砖工”，因此慨叹“程序员是吃青春饭的”。所幸，在本书中有“编写高质量的代码”“阅读高质量的代码”“交付高质量的代码”三方面的内容，为我们提供了一个“思考”的范例。高质量的代码绝非一朝一夕能够实现的，需要长期积淀。如此，“年龄”就不再是开发者的魔障了。

三要学习。学而不思则罔，思而不学则殆。学习的方式有多种，“读书”则是一种重要的方式，相比“碎片化”学习的一知半解，完整地读一本书则会让开发者在思维、知识、技能等维度有系统地提升。比如本书中的“高质量的代码”部分不仅仅是实践经验的总结，也应该是开发者的行动指南；而在第三部分“场景化指南”中，则为开发者较为系统地列举了各种应用场景中会用到的工具——名为“指南”，很恰当。

就个人来看，本书比较适合“入门”之后阅读，读者可以根据具体的应用场景循“指南”而深入。

齐伟

“跟老齐学 Python”系列图书作者

推荐序三

判断一门编程语言是不是流行，可以观察该语言相关图书的多寡，虽然市面上已有众多 Python 相关的书籍，但我仍有充分的理由来推荐本书，原因如下。

第一，本书的作者之一 Kenneth Reitz 先生是大名鼎鼎的 Requests 库的作者，仅凭此一点，本书的质量就有了充分的保证。

第二，根据本人多年从事软件开发和 Python 培训的经验，很多通过自学或培训、刚刚转行成功的程序员，虽然学会了用写代码来解决具体问题，但对下一步的学习和成长方向比较迷茫。如果在这个阶段能在公司遇到一位好的导师，那么三生有幸，他会引导你快速地成长，独立上手开发项目。但是，如果没有遇到这样的导师，又该怎么办呢？这本《Python 编程之美：最佳实践指南》刚好可以履行起这样一位导师的职责：书中丰富的内容、详尽的指导能让你快速补上运用 Python 开发项目的知识短板。

第三，本书用两章的篇幅着重阐述了如何编写、阅读高质量的代码，这对于每一位 Python 开发人员都是极有价值的。

在第 4 章“编写高质量的代码”中，作者从代码风格、组织好项目的结构、测试代码、文档、日志等方面全面讲解如何写出高质量的代码。刚踏入软件开发的朋友们，如果还没有养成写测试、日志的习惯，完全可以从这里完整地补充相关知识点。

在第 5 章“阅读高质量的代码”中，按照第 4 章编写高质量的代码的原则，引导读者阅读 6 个优秀的项目（HowDoI、Diamond、Tablib、Requests、Werkzeug、Flask），学习开源项目的架构设计、书写 Python 风格的代码——小到命名风格、大到实现特定需求的数据结构和算法等。

本书还简明扼要地阐述了软件交付、Web 应用、持续集成、分布式系统等方面的知识，虽然不是特别详细，但足以引导读者了解 Python 项目开发的方方面面，对 Python 的开发过程有全局、清晰的认识。此外，读者还可以借助本书提供的资源链接，运用搜索引擎来拓展自己的知识面。

综上，我认为本书值得每一位初级程序员拥有，如果读者能把从中学到的知识娴熟地应用到日常开发中，那么个人成长的价值就远远超出本书微薄的定价了。

黄哥

知乎专栏“通过 Python 学会编程”作者

好评袭来

本书是 Python 开发者的实操指引，Kenneth Reitz 出品，必属精品。

——刘志军

公众号“Python 之禅”出品人

最近十年，人工智能、金融科技等领域发展迅速，Python 在这些领域中的应用取得了巨大的成功。本书追根溯源，从代码风格、设计哲学、开源项目等各方面为我们详述了 Python 强大的生态系统，书名中的“最佳实践指南”名副其实。

——阿橙

公众号“Python 中文社区”主编

本书不仅仅是一本 Python 教程，更多的是通过一些优秀项目源码，向读者阐释了如何写出更优雅的 Python 代码……有助于 Python 学习者完成向 Pythonic 的转变。

——Crossin

公众号“Crossin 的编程教室”作者、码课创始人

《Python 编程之美：最佳实践指南》不仅仅是一本介绍 Python 代码编写最佳实践的书籍，更是一本 Python 开发相关的百科全书。书中介绍了 Python 在各大操作系统上的安装方法、各种用于 Python 开发的文本编辑器和集成开发环境（IDE），教你以“Python 之禅”为指导书写高质量的 Python 代码、阅读高质量的 Python 第三方库源码。除了代码编写，书中还介绍了一系列与 Python 开发工作相关的高效工具，例如持续集成工具、系统管理工具、代码交付工具和发布工具等。这本书对于已有一定 Python 基础的人来说是一本极佳的进阶教材和参考资料。应用书中介绍的最佳实践能够帮助我们编写更加优雅、Pythonic 的代码。学习书中介绍的各种工具能够帮助我们提高代码从开发到交付再到发布的效率。学习一门编程语言，从入门到进阶的一条捷径就是学习前辈专家的开发经验，而这本书正是这些专家经验的总结。

——杨学光

Django 中文社区发起人

《Python 编程之美：最佳实践指南》详略得当，一定会让你深入了解 Python 大世界。

——Raymond Hettinger
杰出的 Python 核心开发者

本书是 Python 开发者不可或缺的图书。对于新手来说，它是一份记录社区约定和最佳实践的学习资源，其价值不可估量。社区内大量专家一起精诚协作，将开源的精华结集出版，与世界同仁共享 Python 编程的一流理念和实践。

——Eric Holscher
Read the Docs 公司（readthedocs.org）的联合创始人

这真是一本令人惊叹的好书！它并不是教授 Python 语言本身的，而是假定你已经有了一定的编程基础。它阐述的是编程相关的知识点：何时、何地、怎样运用 Python，如何使用各类 Python 工具……这使得你可以高效地编写代码、运维程序，并与其他程序优雅交互。

我从中学到了太多的东西，尤其是那些我以前从来不知道自己不知道的东西，而且发现它们竟然是如此的有用！

书中所有的资源都提供了链接地址。如果你喜欢写代码并且用专业的方式呈现这些代码（毕竟从设计开始它们就是为这样的目的而诞生的），那么这本书是你的必读之作！

——读者
来自 amazon.com

译者序

从毕业至今，我在互联网行业从事软件研发工作已将近五年。这五年间，我做过后端开发、前端开发、大数据处理等，使用过的编程语言包括 Python、PHP、Go、Java、JavaScript 等。

虽说编程语言各异，但是我使用它们来写各种项目的代码始终坚持两点：代码可读性和自解释性 / 自文档性（self-documentation）。这很大程度上应该是受到 Python 语言设计哲学的影响——追求简单、易读、易懂的代码。

很多人可能会认为这两点其实均可归结为代码可读性一点，但我想做点区分。代码可读性突出对代码阅读者视觉上的影响，即在视觉效果上是否存在对阅读者不必要的理解干扰，比如必要的空行、变量定义与使用之间的距离、函数体 / 逻辑分支是否过长、逻辑表达是否直观等。可读性高的代码通常都非常漂亮，令人赏心悦目。自解释性代码则更侧重语义层面，比如变量名称、函数名称、类名是否恰当，函数、方法、API 职责是否单一，工程目录结构、包、模块拆分是否符合“高内聚、低耦合”原则等。写代码以这两点为原则，可以极大地提高个人以及团队的工作效率和工作质量。

本书作者 Kenneth Reitz 于 2011 年发布 Requests 这个 HTTP 请求工具库，提出“for humans”的理念，强调软件 / 工具库应该友好易用，这一理念本质上是对 Python 哲学（特别是上述两点）的一种引申和发扬。之后 Reitz 在一些 Python 大会上做技术分享，宣扬“for humans”的理念，对 Python 社区产生了巨大影响。我在第一次用 Requests 库之后，便很少使用 Python 标准库中的 urllib 和 urllib2 了，现在标准库文档中也特别建议开发者使用 Requests。

因为对“for humans”理念的认同，也因为我经常使用 Requests，所以当 Reitz 在 GitHub 上邀请我翻译 Requests 文档中文版时，我欣然接受，和本书的另一位译者邦杰共同翻译了 Requests 文档的首个官方中文版。

在 Reitz 发起“The Hitchhiker’s Guide to Python!”项目（也就是本书的社区开源版）后，我一直跟进阅读，收获巨大。后来得知这本开源书籍正式出版，欣喜若狂，辗转咨询多人，联系到刘皎老师，申请了本书的翻译工作。但是，后来发现翻译的工作量远远超出预估，除一些主观原因外，主要因为本书内容的广度和深度。

- 广度：本书由 Python 社区数百人共同创作而成，可以视作 Python 小百科全书。第 1~3 章指导读者按照自己的需求选择安装配置 Python 版本 / 发行版、开发环境等。第 7~11 章则针对不同的应用场景，从多个维度甄选并对比了大量的 Python 库，读者可以“按图索骥”地做出自己的选择，从而节约大量的时间精力。因为译者的 Python 开发经验主要集中在 Web 开发和数据处理上，对于很多应用场景下的 Python 库不太熟悉，所以翻译之前花费了大量时间来学习和理解。
- 深度：针对 Python 新手的核心需求，本书探讨了大量的最佳实践。其中第 4~5 章通过大量示例具体地阐释了“Python 之禅”：如何编写高质量的 Python 代码，并精选若干高质量的知名 Python 开源项目，详细介绍如何通过阅读源码来提升编程技术水平。虽说在 Python 社区几乎人人都知道“Python 之禅”，但如何落实在开发实践中估计极少有人能说得清楚。对照书中的实例阐释，译者反复推敲“Python 之禅”的译文，最终敲定的译文也不是特别令自己满意。

相比原计划，本书最终延期近一年才翻译完成。除了歉意，我内心满是感谢：感谢邦杰中途友情加入，帮忙翻译了第 4~6 章的初稿，这三章的难度和文字量都非常大；感谢编辑刘皎对我拖稿的次次容忍和耐心等待；感谢妻儿的理解，我欠了你们太多的陪伴。

虽然我已尽自己所能地保证译文质量，但是错误和瑕疵难免，在此也请读者原谅。希望你们阅读愉快！

夏永锋

写于上海

目录

第 2 部分　步入正题

第 3 部分　场景化指南

前言

Python 是一个大世界，大到让你难以置信！

本书不是教你如何学习 Python 语言的（我们引用了大量优秀资源供你学习），而是一份 Python 社区推荐工具和最佳实践的（有态度的）业内指南。本书的目标读者是初级到中级水平的 Python 程序员，他们可能有志于使用 Python 为开源项目做贡献、开启一段职业生涯或开创一家公司，不过临时用用 Python 的人也会发现第 1 部分和第 5 章的内容对自己颇有帮助。

本书的第 1 部分帮助读者选择适合各自场景的文本编辑器或交互式开发环境（例如，常用 Java 的读者可能偏爱 Eclipse，用它安装 Python 开发插件），并调研了其他可选择的解释器，这些解释器也许可以满足那些你还不知道的 Python 能够解决的需求（例如，MicroPython 是基于 ARM Cortex-M4 芯片的一个实现）。第 2 部分重点介绍开源社区公认的范例代码，展示地道的 Python 代码风格，希望能够鼓励读者进一步深入阅读和尝试开源代码。第 3 部分简要地调研了大量的 Python 社区常用库，让读者初步认识到目前 Python 涉及的领域。

本书纸质版的所有版税都捐赠给 Django Girls，这是一个充满欢乐的全球性组织，旨在组织免费的 Django 和 Python 讲习班，创建开源的在线教程，策划令人惊叹的技术体验活动。如果有意愿，那么你可以从 *http://docs.python-guide.org/en/latest/notes/contribute/* 上了解如何为本书的在线版本做贡献。

本书的使用约定

这个图标标识一个提示或建议。

这个图标标识一个一般性注解。

这个图标标识一个警告信息。

Safari® 图书在线

Safari® 图书在线是一个点播式电子图书馆，以图书和视频的形式展示来自技术和商业领域的世界权威作者的专业内容。

技术专家、软件开发者、Web 设计者及商业和创意专业人士都使用 Safari® 图书在线作为科研、解决问题、学习和认证培训的核心资源。

Safari® 图书在线为企业、政府部门、教育机构，以及个人提供一系列的购买计划。

其成员可以访问无数的书籍、培训视频及正式出版前的草稿，这些资源存放在一个完全可检索的数据库中，资源来源于各大出版商，如 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等 200 多家出版商。更多关于 Safari® 图书在线的信息，请访问 *https://www.safaribooksonline.com/*。

联系我们

请将对本书的评价和发现的问题通过如下地址通知出版社。

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

我们提供了本书网页，上面列出了勘误表、示例和其他信息。请通过 *http://bit.ly/restful-Web-clients* 访问该页。

要给出本书意见或者询问技术问题，请发送邮件到 *bookquestions@oreilly.com*。

更多有关书籍、课程、会议和新闻的信息，请见网站 *http://www.oreilly.com*。

在 Facebook 找到我们：*http://facebook.com/oreilly*。
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*。
在 YouTube 上观看：*http://www.youtube.com/oreillymedia*。

致谢

欢迎各位朋友阅读《Python 编程之美：最佳实践指南》。

据我所知，本书的成书方式是首创的：由一个作者（也就是我，Kenneth）设计策划，而大部分内容由来自世界各地的数百人免费提供。人类有史以来很少有过以这种技术方式达成这种规模的美妙协作。

本书得以完成，归功于以下 3 方的共同努力。

1. 社区

爱把我们凝聚在一起，克服了万千困难。

2. 软件项目

Python、Sphinx、Alabaster 和 Git。

3. 在线服务

GitHub 和 Read the Docs。

最后，我想对 Tanya 致以衷心的感谢，她克服了全部的艰难困苦，把这项工作转变成图书形式，并将一切准备妥当交付出版社正式出版。还有极其出色的 O'Reilly 团队：Dawn、Jasmine、Nick、Heather、Nicole、Meg，以及其他许许多多工作在幕后的人，感谢你们的付出，让本书如此完美。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/34757*

第 1 部分

起步

第 1 部分受到了 Stuart Ellis 的《Windows 环境下的 Python 指南》（*http://www.stuartellis.name/articles/python-development-windows/*）一文的启发，主要介绍如何搭建一个 Python 环境，包含以下几章。

第 1 章，选择一个解释器：对比 Python 2 和 Python 3，并讨论了 CPython 之外其他可选择的解释器。

第 2 章，恰当地安装 Python：演示如何获取 Python、pip 和 virtualenv。

第 3 章，搭建开发环境：描述 Python 开发者最喜爱的一些文本编辑器和集成开发环境。

第 1 章

选择一个解释器

Python 2 与 Python 3 的状况对比

选择 Python 解释器时始终存在一个重要的问题：选择 Python 2 还是 Python 3。答案并非像有些人想的那么显而易见（虽然大家对 Python 3 的关注度与日俱增）。

Python 状况的说明如下所示。

- 长久以来，Python 2.7 都是标准选择。
- Python 3 对语言做了重大改变，令有些开发者不开心[1]。
- 2020 年 Python 2.7 会进行必要的安全更新（*https://www.python.org/dev/peps/pep-0373/*）。
- Python 3 持续演进，就像过去那些年的 Python 2 那样。

至此你应该明白为什么选择 Python 2 还是 Python 3 不是一个容易的决定了。

建议

在我们看来，真正精明的人[2]会使用 Python 3。但如果你迫于无奈只能使用 Python 2，那也没什么，至少你仍然在用 Python，以下是我们的建议。

如果你满足下面 3 个条件之一，那么请使用 Python 3。

1 如果不做大量底层网络编程，那么除 print 语句变成一个函数之外，几乎感觉不到其他变化。否则，“不开心”只是一种客气且保守的表述，某些庞大且流行的 Web、socket 或网络编程库要处理 Unicode 和字节字符串，因为 Python 3 的重大变化，这些库的开发负责人已经（或者仍然在）对库做了大量更新。关于 Python 3 的变化细节，可以直接阅读 Python 3 问世后的首个简介（*http://bit.ly/text-vs-data*），从“关于二进制数据和 Unicode，你自以为理解的一切都已改变”开始阅读。

2 这里的“真正精明的人”是指那些知道自己应该把最大的价值投资在哪里的人。

- 你喜爱 Python 3。
- 你不知道选择哪一个。
- 你勇于拥抱变化。

如果你满足下面 3 个条件之一，那么请使用 Python 2。

- 你喜爱 Python 2，并且对 Python 3 的未来持悲观态度。
- 切换到 Python 3 会影响软件的稳定性[3]。
- 你所依赖的软件要求使用 Python 2。

那就选择 Python 3 吗

如果你正要选择一个 Python 解释器，也没什么偏见，那么就使用最新的 Python 3.x。每个版本都会带来新的标准库模块、安全补丁和问题修复。除非你有特殊原因，比如某个库为 Python 2 独有，Python 3 没有合适的替代库；要求使用某个特定的 Python 实现；或者 Python 2 让你（像我们中的某些人一样）欢欣鼓舞，否则都不要使用 Python 2。

参照《我能使用 Python 3 吗》（*https://caniusepython3.com/*），看看是否有任何你正依赖的 Python 项目阻拦你采用 Python 3。

《Python 2 还是 Python 3》（*http://bit.ly/python2-or-python3*），这篇文章清晰地说明了语言标准上不再向后兼容的一些原因，并提供了一些详细说明标准差异的文档链接。

如果你是一个初学者，相比 Python 版本之间的兼容性问题，还有很多更重要的事情要考虑，那么先使用 Python，再考虑版本兼容问题。

Python 的不同实现

大家说起 Python 这个词，通常不仅是指这门语言，也指代了 CPython 这个实现。Python 实际上是一门语言的一个标准规范，实现方式可以多种多样。

不同的实现也许对某些库的兼容性有所不同，或者运行效率存在差异。无论你选择哪种 Python 实现，纯 Python 实现的库应该都能正常工作，但那些 C 实现的库（例如，NumPy）则不然。本节简要介绍几种最流行的 Python 实现。

本书假定你正在使用 Python 3 的标准 CPython 实现，在涉及 Python 2 时通常也会添加一些说明。

3　这个链接是 Python 标准库变化点的一个简要列表（*http://python3porting.com/stdlib.html*）。

CPython

CPython（*http://www.python.org/*）是 Python 的参考性实现[4]。它用 C 语言编写，先将 Python 代码编译成中间字节码，然后在虚拟机上解释执行。CPython 对 Python 包和 C 扩展模块[5]的兼容性最好。

编写开源 Python 代码时，如果希望覆盖更多的潜在用户，那么就使用 CPython。如果 Python 包的功能依赖于 C 扩展，那么 CPython 是唯一的选择。

因为 CPython 是参考性实现，所以 Python 语言的所有版本都用 C 语言实现过。

Stackless

Stackless Python（*https://bitbucket.org/stackless-dev/stackless/wiki/Home*）在普通 CPython（因此 CPython 可以使用的库，它也能使用）的基础上打了一个补丁，它将 Python 解释器与调用栈解耦，从而在某些情况下可以改变代码的执行次序。Stackless 引入了 tasklet 的概念，它可以封装函数，将函数转变成“微线程”，这种微线程可以序列化到磁盘，以便将来执行和调度，默认是轮询执行。

greenlet 库为 CPython 用户实现了这种栈切换功能，在 PyPy 中也部分实现了这种功能。

PyPy

PyPy（*https://pypy.org/*）在是用 RPython 实现的一个 Python 解释器。RPython 是 Python 语言的一个有限子集，在运行时可以推断出变量类型，因此可以应用某些优化策略。PyPy 的特色是 JIT 编译器，它支持多种编译器后端，比如 C、通用中间语言（CIL）及 Java 虚拟机（JVM）字节码。

PyPy 的目标是在提升性能的同时尽可能最大程度地兼容 CPython 参考性实现。如果想提升 Python 代码的性能，那么 PyPy 值得一试。一组基准测试表明目前 PyPy 的执行速度至少是 CPython 的 5 倍。

PyPy 支持 Python 2.7，PyPy 3 则对应 Python 3。两个版本都可以从 PyPy 的下载页（*http://pypy.org/download.html*）上下载。

Jython

Jython（*https://www.jython.org/*）在解释器可以将 Python 代码编译成 Java 字节码，再由

4 参考性实现，准确地表达了语言的定义，其行为是其他实现应该表现的。
5 C 扩展模块用 C 语言编写，在 Python 代码中使用。

JVM 来执行。此外，它还能够像 Python 模块一样导入并使用任意 Java 类。

如果需要与已有的 Java 代码库进行交互，或者因为其他原因需要基于 JVM 编写 Python 代码，那么 Jython 是不二之选。

Jython 目前支持到 Python 2.7。

IronPython

IronPython（*https://ironpython.net/*）在是专为 .Net 框架准备的一个 Python 实现，它可同时使用 Python 和 .Net 框架的代码库，从而让 .Net 框架中的其他语言也可以调用 Python 代码。

Visual Studio 的 Python 工具集将 IronPython 直接集成到 Visual Studio 开发环境中，对于 Windows 平台上的开发者来说，IronPython 是一个理想选择。

IronPython 支持 Python 2.7。

PythonNet

PythonNet（*https://pythonnet.github.io/*）是一个 Python 程序包，它能够近乎无缝地将 .Net 中间语言在运行时（CLR）集成到 Python 环境中。它与 IronPython 的使用方式相反，意味着 PythonNet 和 IronPython 之间是互补而不是竞争关系。

与 Mono（*https://www.mono-project.com*）协同，PythonNet 使非 Windows 操作系统（例如，Mac OS X 和 Linux）上的 Python 能够深入 .Net 框架中进行操作。它可以与 IronPython 同时存在，不会相互冲突。

PythonNet 支持 Python 2.3 到 Python 2.7 版本。在 PythonNet 的自述文件页面可以看到安装说明。

Skulpt

Skulpt（*https://www.skulpt.org/*）是 Python 语言的一个 JavaScript 实现。它还没完全移植到 CPython 标准库。Skulpt 能实现的标准库中有模块 math、random、turtle、image、unittest 及 time、urllib、DOM 模块的部分，它主要用于教学。你也能自己添加模块（*http://www.skulpt.org/static/developer.html#adding-a-module*）。

值得一提的 Skulpt 应用案例有 IneractivePython（*http://interactivepython.org/*）和

CodeSkulptor（*http://www.codeskulptor.org/demos.html*）。

Skulpt 支持 Python 2.7 和 Python 3.3 的大部分特性。详细情况见 Skulpt 的 GitHub 页面（*https://github.com/skulpt/skulpt*）。

MicroPython

MicroPython（*https://micropython.org/*）是一个 Python 3 实现，为了在微控制器上运行进行了针对性优化，支持 Thumb v2 指令集的 32 位 ARM 处理器，例如，用于低功耗微控制器上的 Cortex-M 系列。它不仅包含一些源自 Python 标准库的模块，还提供了一些 MicroPython 特有的库来提供开发版的硬件细节、内存信息、网络访问，以及一个经过裁剪优化的 ctypes 版本。pyboard（*https://micropython.org/store/#/store*）使用 MicroPython 作为操作系统，不同于 Raspberry Pi，Raspberry Pi 是提供一个 Debian 或其他 C 语言实现的操作系统，系统中安装了 Python。

从现在起，假定我们使用的都是 CPython，不管操作系统是类 Unix、Mac OS X，还是 Windows。

马上开始学习安装 Python 了，请做好选择！

第 2 章

恰当地安装Python

本章会逐一讲述如何在 Mac OS X、Linux 及 Windows 平台上安装 CPython。其中关于打包工具（如 Setuptool 和 pip）的内容是重复的，因此可以直接跳到目标操作系统部分学习。

如果用一款商业化 Python 的发行版，比如 Anaconda、Canopy，那么应按照供应商的说明安装。本章的“商业化 Python 二次发行版”部分也有一小段说明。

如果你已经安装了 Python，那么绝对不要把指向 Python 可执行文件的符号链接改为指向其他文件，那会和大声读出沃贡人的诗歌[1]（设想一下这样一种情形，系统中的某些代码依赖于某个特定位置的一个特定 Python）一样糟糕。

在 Mac OS X 上安装 Python

El Capitan 是 Mac OS X 的最新版本，它自带一个 Mac 特有的 Python 2.7 实现。

因此，在 Mac OS X 系统上使用 Python，不需要安装或者配置任何其他东西。在开始构建实际的 Python 应用（例如，为协作项目做贡献）前，要先安装 Setuptools、pip 及 virtualenv。在 Setuptools、pip、virtualenv 三者中，Setuptools 始终应该最先安装，因为它能让使用其他第三方 Python 库更简便。

Mac OS X 系统自带的 Python 版本只适合学习，不太适合协作开发。因为 Mac OS X 系

1 该词出自英国作家道格拉斯·亚当斯所写的科幻小说《银河系漫游指南》，书中称沃贡人的诗歌是宇宙中第三糟糕的诗歌。

统自带的版本可能已过时，Python 官方最新发布的稳定版本才是适合生产环境的版本[2]。如果只是想私下编写一些脚本从某些网站抓取信息或处理数据，那么使用 Mac OS X 系统自带的 Python 即可。如果你正在为开源项目做贡献，或者和团队成员一起做项目，而其他成员可能使用不同的操作系统（或者将来会使用不同的操作系统[3]），那么使用 CPython 正式发布的版本才是正确的选择。

下载之前，请阅读下面的注意事项。在安装 Python 之前，需要先安装 GCC。通过下载安装 Xcode、较小的命令行工具集（有苹果账号才能下载它），或者更小的 osx-gcc-installer 安装包都可以得到 GCC。

如果已安装了 Xcode，那么就不要再安装 osx-gcc-installer 了，因为同时安装这两个软件可能会造成一些难以诊断的问题。

Mac OS X 虽然自带了大量的 Unix 实用工具，但熟悉 Linux 系统的人会发现它缺少一个关键组件——包管理器，Homebrew 填补了这一空白。

打开 Terminal 或者其他 Mac OS X 终端模拟器，执行下面的代码来安装 Homebrew。

```
$ BREW_URI=https://raw.githubusercontent.com/Homebrew/install/master/install
$ ruby -e "$(curl -fsSL ${BREW_URI})"
```

这个脚本在安装开始前会解释其将做一些什么变更，并提示用户。一旦安装了 Homebrew，记得将 Homebrew 的目录插在 PATH 环境变量的最前面[4]，可以通过在 ~/.profile 文件的末尾添加下面这行代码来实现。

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

安装 Python，在终端中执行下面的命令：

```
$ brew install python3
```

2 有人对此持不同看法。理由之一是 Mac OS X 系统上的 Python 实现与众不同，甚至包含一些 Mac OS X 专有库。Stupid Python Ideas 博客就此话题发表了一篇驳斥我们建议的文章（*http://bit.ly/sticking-with-apples-python*）。文章认为如果有人同时安装 Mac OS X 的 CPython 2.7 和官方的 CPython 2.7，那么可能会遇到某些名称冲突的问题。如果有此顾虑，完全可以使用虚拟环境。或者，至少可以对 Mac OS X 中的 Python 2.7 置之不理，在确保系统稳定运行的前提下，安装 CPython 实现的标准 Python 2.7，修改环境变量 PATH，绝不使用 Mac OS X 的 Python 版本，那么一切都能相安无事，包括那些依赖于 Mac OS X 专有 Python 版本的产品。

3 不过说实话，个人认为最佳方案是选择 Python 3，或者从一开始就使用虚拟环境，除 virtualenv（根据 Hynek Schlawack 的建议也许还应该安装 virtualenvwrapper）外，不要全局安装任何东西。

4 确保你使用的 Python 是 Homebrew 刚安装的，且没有改变系统原有的 Python。

或者安装 Python 2 ：

```
$ brew install python
```

Python 默认安装在 /usr/local/Cellar/python3/ 或 /usr/local/Cellar/python/ 目录，并在 /usr/local/python3 或 /usr/local/python 目录下创建解释器软链接[5]。如果执行 pip install 命令想使用 --user 选项，则需要绕过与 distutils 和 Homebrew 配置相关的一个 bug。推荐使用虚拟环境，相关信息会在 virtualenv 小节说明。

Setuptools 和 pip

在安装 Python 时，Homebrew 会把 Setuptools 和 pip 一并安装好。如果使用 Python 3，那么 pip 可执行程序实际指向 pip3 ；如果使用 Python 2，则指向 pip。

有了 Setuptools，就可以使用单个命令（easy_install）经由网络安装任何兼容的[6] Python 软件，开发者也可以轻松地在自己的 Python 软件里添加这种网络安装。

pip 的 pip 命令和 Setuptools 的 easy_install 命令都是安装管理 Python 程序包的工具。相比 easy_install 命令，笔者更推荐 pip 命令，因为它不仅可以卸载程序包，错误信息更容易理解，而且也不会出现程序包部分安装的情况（安装过程中途失败会把至此发生的操作全部回退）。更详细的讨论，请阅读《Python 程序打包用户指南》（它是最新打包信息的第一参考资料）中的“pip vs easy_install（*http://bit.ly/pip-vs-easy-install*）”这部分内容。

在 shell 中输入下面的命令来升级 pip ：

```
$ pip install --upgrade pip
```

virtualenv

virtualenv（*https://pypi.python.org/pypi/virtualenv*）用于创建隔离的 Python 环境。它会创建一个目录，其中包含一些可执行程序，Python 项目若依赖于其他 Python 程序包，就会用到这些可执行程序。有些人尊崇一个最佳实践 ：除 virtualenv 和 Setuptools 之外，不在系统全局安装任何东西，始终使用虚拟环境[7]。

若通过 pip 安装 virtualenv，则在终端 shell 的命令行中运行 pip 命令。

---

5 软链接是一个指向实际文件路径的指针，可以在命令行中输入命令（例如，ls -l /usr/local/bin/python3）确认软链接指向哪儿。

6 兼容 Setuptools 的包至少要提供足够的信息方便识别和获取所有的包依赖。更多信息请阅读文档打包分发 Python 项目（*https://packaging.python.org/en/latest/distributing.html*）、PEP 302（*https://www.python.org/dev/peps/pep-0302/*）和 PEP 241（*https://www.python.org/dev/peps/pep-0241/*）。

7 这一实践的拥趸者认为这是唯一可以确保不会因为某个库用新版本覆盖原有版本而破坏系统中存在版本依赖的其他代码的方式。

```
$ pip3 install virtualenv
```

如果使用的是 Python 2，则运行：

```
$ pip install virtualenv
```

无论使用 Python 2 还是 Python 3，一旦处于虚拟环境中，就可以始终使用 pip 命令，本书的余下部分都是这样操作的。虚拟环境一节会详细说明它的用法和动机。

在 Linux 上安装 Python

从 Wily Werewolf（Ubuntu 15.10）版本开始，Ubuntu 发行时仅预装 Python 3（Python 2 可通过 apt-get 安装）。详细说明可见 Ubuntu 官方的 Python wiki 页面（*https://wiki.ubuntu.com/Python*）。Fedora 的第 23 个发布版本首次仅预装 Python 3（在第 20 至第 22 个发布版本中 Python 2.7 和 Python 3 并存），但 Python 2.7 可通过软件包管理器进行安装。

如果同时安装 Python 2 和 Python 3，那么多数情况下都是将 Python 2 符号链接到 Python 2 解释器，将 Python 3 符号链接到 Python 3 解释器。如果决定使用 Python 2，类 Unix 系统中目前推荐（参考 Python 增强提案 PEP 394）在 shebang 符号中指定 Python 2，例如，文件首行为#!/usr/bin/env Python2，而不是依赖运行环境中 Python 指向期望的解释器位置。

在 PEP 394 中虽未提及，但将 pip2 和 pip3 分别符号链接到各自的 pip 包安装器已成为一种约定。

Setuptools 和 pip

虽然可以通过系统中的软件包管理器安装 pip 命令，但是为确保安装的是最新版本，也请遵从以下步骤[8]。

打开一个 shell 程序，输入：

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python3 get-pip.py
```

使用 Python 2 则输入：

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python get-pip.py
```

这个过程也会安装 Setuptools。

8 欲知更多细节，请阅读 pip 的安装说明（*https://pip.pypa.io/en/latest/installing.html*）。

使用随 Setuptools 安装的 easy_install 命令，可以经由网络下载安装任意兼容的[9] Python 软件，也让开发者可以轻松地在自己的 Python 软件里添加这种网络安装。

pip 是一个有助于简化 Python 软件包安装和管理的工具。相比 easy_install 命令，笔者更推荐 pip 命令，因为它不仅可以卸载程序包，错误信息更容易理解，而且也不会出现程序包部分安装的情况（安装过程中途失败会把至此发生的操作全部回退）。更详细的讨论，请阅读《Python 程序打包用户指南》（它是最新打包信息的第一参考资料）中的“pip vs easy_install（*http://bit.ly/pip-vs-easy-install*）”这部分内容。

开发工具

我们总会碰到这种情况：想要使用的 Python 库依赖于 C 扩展。有时软件包管理器可能已经预先构建好了 Python 库，我们可以先检查一下（使用 yum search 或 apt-cache search）。另外因为 Python 推出了更新的 wheels 打包格式（预先编译好特定平台的二进制文件），所以也可以使用 pip 直接从 PyPI 获取二进制安装文件。但如果将来要创建 C 扩展，或者正使用的 Python 库的维护者还没针对平台制作 wheels，则需要为 Python 准备一些开发工具：各种各样的 C 库、make 及 GCC 编译器。下面是一些常用的 Python 程序包，它们都使用了 C 库。

并发工具

- 线程库 threading（*https://docs.python.org/3/library/threading.html*）。
- 事件处理库 asyncio（*https://docs.python.org/3/library/asyncio.html*），它是 Python 3.4 及以上的版本才有的。
- 基于协程的网络库 curio（*https://curio.readthedocs.org/*）。
- 基于协程的网络库 gevent（*http://www.gevent.org/*）。
- 事件驱动的网络库 Twisted（*https://twistedmatrix.com/*）。

科学分析

- 线性代数库 NumPy（*http://www.numpy.org/*）。
- 数值处理工具包 SciPy（*http://www.scipy.org/*）。
- 机器学习库 scikit-learn（*http://scikit-learn.org/*）。
- 绘图库 Matplotlib（*http://matplotlib.org/*）。

数据 / 数据库接口

- HDF5 数据格式接口库 h5py（*http://www.h5py.org/*）。

9 兼容 Setuptools 的包至少要提供足够的信息方便识别和获取所有的包依赖。更多信息，请阅读文档打包分发 Python 项目（*https://packaging.python.org/en/latest/distributing.html*）、PEP 302（*https://www.python.org/dev/peps/pep-0302/*）和 PEP 241（*https://www.python.org/dev/peps/pep-0241/*）。

- PostgreSQL 数据库适配器 Psycopg（*http://initd.org/psycopg/*）。
- 数据库抽象和对象关系映射库 SQLAlchemy（*http://www.sqlalchemy.org/*）。

在 Ubuntu 上，在一个终端 shell 中输入：

```
$ sudo apt-get update --fix-missing
$ sudo apt-get install python3-dev # 对于 Python 3
$ sudo apt-get install python-dev # 对于 Python 2
```

或者在 Fedora 上，在一个终端 shell 中输入：

```
$ sudo yum update
$ sudo yum install gcc
$ sudo yum install python3-devel # 对于 Python 3
$ sudo yum install python-devel # 对于 Python 2
```

执行 pip3 install--user desired-package 就可以构建必须经过编译的 Python 库了（如果是 Python 2，则执行 pip install--user desired-package）。当然也需要先安装工具本身的客户端开发库（安装细节可参考 HDF5 的安装文档）。如果想在 Ubuntu 上安装 PostgreSQL 客户端开发库，那么要在终端 shell 中输入以下命令：

```
$ sudo apt-get install libpq-dev
```

在 Fedora 上则是：

```
$ sudo yum install postgresql-devel
```

virtualenv

virtualenv 命令随 virtualenv 软件包安装而来，用于创建隔离的 Python 环境。它会创建一个目录，其中包含一些可执行程序，Python 项目若依赖于其他 Python 程序包，就会用到这些可执行程序。

如果使用 Ubuntu 的包管理器来安装 virtualenv，则输入：

```
$ sudo apt-get install python-virtualenv
```

如果使用 Fedora，则输入：

```
$ sudo yum install python-virtualenv
```

如果通过 pip 安装，那么要在一个终端 shell 的命令行中运行 pip 命令，并使用 --user 选项安装到局部，而不是安装到系统全局：

```
$ pip3 install --user virtualenv
```

如果使用 Python 2，则输入：

```
$ sudo pip install --user virtualenv
```

无论使用 Python 2 还是 Python 3，一旦处于虚拟环境中，就可始终使用 pip 命令，本书的余下部分都是这样操作的。虚拟环境一节会详细说明它的用法和动机。

在 Windows 上安装 Python

与其他系统的 Python 开发者相比，Windows 用户遇到的麻烦会更多些，因为在 Windows 上编译任何东西都会更困难一点，并且许多 Python 库底层都使用了 C 扩展。幸亏有 wheels，我们可以使用命令 pip 直接从 PyPI 下载二进制安装包。

在 Windows 上安装 Python 有两个途径：选择一个商业化发行版（商业化 Python 二次发行版一节有相关论述）或者直接安装 CPython。商业化 Python 二次发行版 Anaconda 用起来很方便，特别是做科研工作时，几乎每个在 Windows 上做科学计算的人都推荐 Anaconda。但是如果你精通编译和链接，想为使用 C 代码的开源项目做贡献，或者只是不想使用一个商业化发行版，则可以考虑直接安装 CPython[10]。

随着时间的推移，越来越多依赖 C 库的程序包在 PyPI 上有了 wheels，因而也就能通过 pip 命令获取二进制安装包。但如果程序包依赖的 C 库没有打包在 wheels 中，那么麻烦又会出现。这是导致优先选择 Anaconda 这种商业化 Python 二次发行版的另一个原因。

如果你是以下 Windows 用户中的一种，那么请使用 CPython。

- 不需要依赖于 C 扩展的 Python 库。
- 拥有 Visual C++ 编译器（不是免费的那种）。
- 能安装 MinGW 的配置。
- 下载 Python 库的二进制安装包 [11] 并能轻松用命令 pip 安装它。

如果想用 Python 来替代 R 或者 MATLAB，或者只是想快速上手，之后如有必要再安装 CPython，那就使用 Anaconda[12]。

如果希望安装过程的交互方式尽可能是图形化的，或者如果 Python 是你学习的第一门编

10 如果想集成 Python 和 .Net 框架，则可以考虑 IronPython。如果你是一个初学者，那么目前它可能不适合你。此外，本书是围绕 CPython 进行论述的。

11 你至少必须知道要使用什么版本的 Python，以及选择 32 位还是 64 位的 Python。因为所有第三方 DLL 都提供 32 位的版本，但有些动态链接库（DLL）不提供 64 位的版本，所以我们推荐使用 32 位的。最广泛的获取编译过的二进制安装文件的途径是 Christoph Gohlke 的资源站点。对于 scikit-learn，Carl Kleffner 正在使用 MinGW 构建二进制安装包，准备最终发布在 PyPI 上。

12 Anaconda 会提供更多免费的资源，并与 Spyder（一个很好的 IDE）捆绑安装。如果使用 Anaconda，那么将发现 Anaconda 免费包索引和 Canopy 包索引大有帮助。

程语言，并且当前是第一次安装，那么就使用 Canopy。

如果你们团队已选定了其中某个方案，那么你也应该使用那个方案。

在 Windows 上安装标准的 CPython 实现，先要从官网上下载最新版 Python 3 或 Python 2.7。如果想确保正在安装的是最新版本或者想要 64 位的安装器 [13]，那就从 Python 的 Windows 发行版页面（*https://www.python.org/downloads/windows/*）下载需要的发行版。

Python 的 Windows 版本以 MSI 包格式提供，它让 Windows 管理员可以使用标准工具来自动安装。双击文件即可手动安装 MSI 包。

Python 会安装到目录名带版本数字的目录中（例如，Python 3.5 版本会安装在 *C:\Python35* 路径下），这是有意为之的。这样就能在同一个系统中安装多个版本的 Python。当然，只有一个解释器会成为 Python 文件类型的默认应用。安装器不会自动修改 PATH 环境变量 [14]，因此你始终能够决定运行哪个版本的 Python。

如果每次运行 Python 都要输入 Python 解释器的完整路径名，则很麻烦，因此把默认 Python 版本的目录加到 PATH 中就可以省去输入完整路径名的麻烦。如果想要使用的 Python 在 C:\Python35\ 路径中，那就把如下内容加到 PATH 中。

```
C:\Python35;C:\Python35\Scripts\
```

在 PowerShell[15] 中执行如下命令：

```
PS C:\> [Environment]::SetEnvironmentVariable(
    "Path",
    "$env:Path;C:\Python35\;C:\Python35\Scripts\",
    "User")
```

安装某些 Python 包时，相关可执行命令文件会放入第二个目录中（Scripts），因此这是一个非常有用的附加路径。有了它，不需要安装或配置任何其他东西，就可以使用 Python 了。

之前说过，在开发有实际用处的 Python 应用（例如，为协作式项目做贡献）之前，强烈推荐安装 Setuptools、pip 及 virtualenv，下面将进一步介绍它们的使用及安装方法。

13 这意味着要百分之百确定你需要的任意 DLL 和驱动程序都有可用的 64 位的版本。

14 操作系统查找可执行程序（包括 Python 和 Python 脚本，如 pip）的所有路径都会罗列在环境变量 PATH 值中，查找路径以分号分隔。

15 Windows PowerShell 提供一个命令行 shell 和一种脚本语言，非常类似于 Unix 上的 shell 程序，所有 Unix 用户无须阅读手册就知道怎么使用，不过 PowerShell 还提供了一些 Windows 的专有特性，它构建在 .Net 框架之上。欲知更多信息，请阅读微软提供的学习资源《使用 Windows PowerShell》。

Setuptools 和 pip

MSI 格式打包的安装器在安装 Python 的同时也会安装 Setuptools 和 pip，因此如果在本书的指导下完成了安装，那么就已经有这两个工具了。否则，最佳获取方式是将 Python 2.7 升级到最新发行版 [16]。对于 Python 3 的 3.3 版本及更早版本，可以下载脚本 get-pip.py[17] 并运行。打开一个 shell，将目录切换到 get-pip.py 所在位置，然后输入：

```
PS C:\> python get-pip.py
```

有了 Setuptools，就可以使用单个命令（easy_install）经由网络安装任何兼容的 [18]Python 软件，也让开发者可以轻松地在自己的 Python 软件里添加这种网络安装。

pip 的 pip 命令和 Setuptools 的 easy_install 命令都是安装管理 Python 程序包的工具。相比 easy_install 命令，笔者更推荐 pip 命令，因为它不仅可以卸载程序包，错误信息更容易理解，而且也不会出现程序包部分安装的情况（安装过程中途失败会把至此发生的操作全部回退）。更详细的讨论，请阅读《Python 程序打包用户指南》（它是最新打包信息的第一参考资料）中的“pip vs easy_install（*http://bit.ly/pip-vs-easy-install*）”这部分内容。

virtualenv

virtualenv（*https://pypi.python.org/pypi/virtualenv*）用于创建隔离的 Python 环境。它会创建一个目录，其中包含一些可执行程序，Python 项目若依赖于其他 Python 程序包，就会用到这些可执行程序。之后，当你在新目录中使用命令激活 Python 环境时，它会把该目录的路径前置在 PATH 环境变量中，这样该目录中的 Python 会第一个被找到，在其子目录中的包也会优先被查找使用。

在 PowerShell 终端的命令行内运行 pip 命令，通过 pip 命令来安装 virtualenv：

```
PS C:\> pip install virtualenv
```

虚拟环境一节会更详细地说明 virtualenv 的用法和动机。在 Mac OS X 和 Linux 上，由于系统自身或第三方软件的需要，可能会同时安装 Python 2 和 Python 3，因此必须区分 Python 2 和 Python 3 版本的 pip 命令。但在 Windows 上，则无此必要，因此当我们说 pip3 时，对于 Windows 用户而言就是指 pip。

16 安装器会询问你是否覆盖原来安装的版本，请选择“是”。次版本号相同的版本发布是向后兼容的。

17 欲知更多细节，请阅读 pip 的安装说明。

18 兼容 Setuptools 的包至少要提供足够的信息方便识别和获取所有的包依赖。更多信息，请阅读文档打包分发 Python 项目（*https://packaging.python.org/en/latest/distributing.html*）、PEP 302（*https://www.python.org/dev/peps/pep-0302/*）和 PEP 241（*https://www.python.org/dev/peps/pep-0241/*）。

商业化 Python 二次发行版

IT 部门或者课程助教可能会让你安装一个商业化二次发行版的 Python。这是为了简化组织内多用户环境一致的维护工作。以下罗列的商业化版本都是提供 C 实现的 Python（CPython）。

本章初稿的一个技术评审说我们低估了多数用户在 Windows 上使用普通 CPython 会遇到的困难，即使是使用 wheels，对大多数用户来说编译和链接外部 C 库都是一件痛苦的事情。虽然我们偏好使用普通的 CPython，但是事实上如果你只是库或包的消费者（而不是创建者或贡献者），那么应该下载一个商业化二次发行版，这样会容易很多。之后，在你想要做开源贡献时，再安装一个 CPython 普通发行版也不迟。

如果安装了特定供应商的 Python 版本，而不修改其默认配置，那么退到标准 Python 发行版会更容易一些。

以下是这些商业化发行版的优势说明。

Intel 公司的 Python 发行版

> 它用一个易于获取且免费的软件包提供高性能 Python。它不仅为一些 Python 程序包链接上原生库（例如，Intel 数学核心库（MKL）），而且增强了多线程能力（包含 Intel 线程构件库（TBB））从而提升了性能。这个发行版依赖于 Continuums 公司的包管理器 Conda 进行包管理，但也自带了 pip。用户可自行下载 Intel 公司的 Python 发行版，或者在 Conda 环境中从网站（*https://anaconda.org*）上下载安装 [19]。
>
> Intel 公司的 Python 发行版提供了 SciPy 技术栈，而且在发布说明中罗列了其他常用库。Intel Parallel Studio XE 的客户可以得到商业支持，其余客户则可以通过论坛寻求帮助。因此，这一选择会提供各种科学计算库，而标准 Python 发行版则不会。

Continuum Analytics 公司的 Anaconda

> Anacnda 基于 BSD 许可证发布，并在它的免费软件包索引网站（*https://repo.continuum.io/pkgs/*）上提供了大量预先编译好的科学计算和数学二进制库。它有一个与 pip 不同的包管理器，名为 Conda。Conda 也能管理虚拟环境，但操作起来更像 Buildout，能为用户管理库和其他外部依赖，而不像 virtualenv。因为 Conda 支

19 由于 Intel 和 Anaconda 有合作，所有 Intel 的加速包都只能通过 Conda 安装使用。不过，你始终可以用 Conda 安装 pip，然后使用 pip（或者用 pip 安装 Conda，然后使用 Conda）。

持的包格式与 pip 不兼容，所以没法从其他包索引地方下载包安装器。

Anacnda 自带 SciPy 技术栈及其他工具。Anaconda 的许可证最好，大部分东西都免费提供。如果你准备使用一个商业化发行版，特别是如果你已经习惯使用命令行，也喜欢 R 或 Scala（二者也已一并捆绑发行），那就选择 Anacnda。如果并不需要全部的附加东西，则可以使用 miniconda 发行版。客户可以选择不同级别的赔偿条款（涉及开源许可证，即什么人在什么时候可以使用什么，或谁会因为什么事情被起诉）、商业支持及额外的 Python 库。

ActiveState 公司的 ActivePython

ActivePython 基于 ActiveState 社区许可证发布，免费试用，试用结束后需要购买许可证。ActiveState 为 Perl 和 Tcl 提供解决方案。它的最大卖点是以宽泛的赔偿条款（同样与开源许可证相关）在其维护的包索引库中提供了 7000 多个程序包，这些程序包可以使用 ActiveStat 的 pypm 工具（pip 的一个替代品）来获取。

Enthought 公司的 Canopy

Canopy 基于 Canopy 软件许可证进行发布，自带一个包管理器 enpkg，用于代替 pip 访问 Canopy 的包索引库。

Enthought 公司为学生和教职工提供免费的学术许可证。Canopy 提供图形化 Python 交互工具，包括一个类似于 MATLAB 的自研 IDE、一个图形化包管理器、一个图形化调试器，以及一个图形化的数据处理工具。与其他商业化二次发行商一样，它不仅为客户提供更多的软件包，而且提供了保障和商业支持。

第 3 章

搭建开发环境

本章将综述目前 Python 开发流行的文本编辑器、集成开发环境（IDE），以及其他开发工具。

我们强烈推荐代码编辑器 Sublime Text 和集成开发环境 PyCharm/IntelliJ IDEA，但也得承认哪个是最佳选择取决于你在编写什么代码，以及在 Python 之外还使用其他什么语言。本章会列举许多最流行的开发工具，并说明选择的原因。

像 Make、Java 的 Ant 或 Maven 这类构建工具是解释型而非编译型[1]的，Python 与它们不同，因此在此我们不讨论构建工具。第 6 章会介绍如何使用 Setuptools 来打包项目并使用 Sphinx 来构建文档。

我们也不会涉及版本控制系统，因为其与编程语言无关，不过维护 Python C（参考性）实现的开发人员刚从 Mercurial 迁移到 Git，参见 PEP 512（*https://www.python.org/dev/peps/pep-0512/*）。PEP 374（*https://www.python.org/dev/peps/pep-0374/*）解释了原来使用 Mercurial 的理由，并比较了当今最流行的 4 大版本控制系统：Subversion、Bazaar、Git 及 Mercurial。

本章最后会简要评述当前管理多个不同解释器的几种方式，从而帮助你在编码时能针对不同的部署场景进行测试。

文本编辑器

虽然任何可以编辑纯文本的工具都可以写 Python 代码，但是选择合适的编辑器可以为你每周节省几个小时的时间。本节列举的所有文本编辑器都支持语法高亮，并可以通过插件扩展来使用静态代码检查器和调试器。

1 如果想要为 Python 构建 C 扩展，则可参考《使用 C 或 C++ 扩展 Python》（*https://docs.python.org/3/extending/extending.html*）这篇文章。欲知更多细节，推荐阅读 *Python Cookbook* 一书的第 15 章。

表 3-1 按推荐优先级从高到低罗列了我们最喜爱的文本编辑器，并说明了某种编辑器优于另一种编辑器的理由。本节概括介绍每种编辑器。如果有人想检查编辑器是否具备某些特性，推荐阅读 Wikipedia 上提供的文本编辑器详情对比图（*https://en.wikipedia.org/wiki/Comparison_of_text_editors*）。

表3-1 文本编辑器列表

工具	可用性	理由
Sublime Text	• 开放 API，提供免费试用版 • 支持 Mac OS X、Linux、Windows 平台	• 快速且内存占用少 • 可轻松处理大型文件（>2GB） • 扩展使用 Python 语言编写
Vim	• 开源并且欢迎捐赠 • 支持 Mac OS X、Linux、Windows、Unix 平台	• 喜爱 Vi/Vim • 在除 Windows 之外的操作系统上都会预装（或至少预装 Vi） • 可以在控制台中使用
Emacs	• 开源并且欢迎捐赠 • 支持 Mac OS X、Linux、Windows、Unix 平台	• 喜爱 Emacs • 扩展使用 Lisp 语言编写 • 可以在控制台中使用
TextMate	• 开源，但是需要一个许可证 • 仅支持 MacMac OS X 平台	• 界面很棒 • 自带几乎所有功能的接口（静态代码检查、调试、测试） • 支持苹果工具链，例如，执行 xcodebuild 的接口（通过 Xcode 捆绑包）
Atom	• 开源，免费 • 支持 Mac OS X、Linux、Windows 平台	• 扩展使用 JavaScript/HTML/CSS 编写 • GitHub 集成非常良好
Code	• 开放 API（最终会），免费 • 支持 Mac OS X、Linux、Windows 平台（但是 Visual Studio 对应的 IDE 仅支持 Windows）	• IntelliSense（代码自动补全）能力与微软的 VisualStudio 相当 • 适用于 Windows 开发，支持 .Net、C# 及 F# • 警告：尚不可扩展（即将可以）（译注：关于 Code 的描述内容已过时）

Sublime Text

推荐使用 Sublime Text 来写代码，标记文本、文章。人们在推荐它时，首先会提及它

的速度，其次是它的可用扩展包的数量（大于 3000 个）。

2008 年 Jon Skinner 发布了 Sublime Text 的首个版本。该编辑器用 Python 语言编写，对 Python 代码编写的支持极佳，并用 Python 来实现其包扩展 API。其工程特性允许用户添加 / 删除文件或目录，可通过跳至任意位置功能搜索文件目录，该功能可识别工程中包含搜索词的位置。

用 PackageControl 访问 Sublime Text 的扩展包仓库，流行的扩展包有：SublimeLinter，用户选择已安装静态代码检查器的一个接口；Emmett，提供 Web 开发代码片段（snippets）[2]；Sublime SFTP，通过 FTP 进行远程编辑。

Anaconda（与同名的商业化 Python 二次发行版无关）发布于 2013 年，能够自动地将 Sublime 转变成近乎一个 IDE，补充了静态代码检查功能、文档字符串检查功能、测试执行器，以及查找高亮对象的定义或位置等能力。

Vim

Vim 是一个基于控制台的文本编辑器（可选图形化用户界面），使用键盘快捷键进行编辑，而不是菜单或图标。由 Bram Moolenaar 于 1991 年首次发布，它的前身 Vi 由 Bill Joy 于 1976 年发布，两者皆以 C 语言实现。

Vim 可通过 vimscript（一种简单的脚本语言）进行扩展。也可以使用其他编程语言：如果想启用 Python 脚本功能，那么在源码构建前，设置构建配置标志 --enable-pythoninterp 或 --enable-python3interp；如果想检查是否已启用 Python 2 或 Python 3，那么输入 echo has(“python”) 或 echo has(“python3”)，如果已启用则结果为 1，否则为 0。

Vi（通常是 Vim）在除 Windows 之外的系统上都是开箱即用的，在 Windows 上使用 Vim 需要先下载安装一个可执行安装器。用户在忍受略显陡峭的学习曲线之后效率会得到极大提升，因而在多数其他编辑器和 IDE 上都有配置项对应基本的 Vi 键。

如果想在一个大公司内从事 IT 工作，熟知 Vi 的功能是必要的[3]。虽然 Vim 比 Vi 的功能特性更多，但 Vim 用户会在 Vi 里操作即可。

2 代码片段是一组经常输入的代码（比如 CSS 的风格或类定义），可以在输入若干字符并按 Tab 键后自动补充完成。

3 在命令行中输入 vi 或 vim 后按 Enter 键就能打开这个编辑器，一旦进入编辑器，输入 help 后按 Enter 键，就能访问其教程。

如果仅使用 Python 语言进行开发，则可以将缩进和自动换行默认配置为 PEP 8（*https://www.python.org/dev/peps/pep-0008/*）兼容的值。在 home 目录[4]下创建一个名为 .vimrc 的文件，并在其中添加以下内容：

```
set textwidth=79     " 长于 79 列的行会被自动折断
set shiftwidth=4     " >> 操作缩进 4 列；<< 操作取消缩进 4 列
set tabstop=4        " 硬制表符（hard TAB）以 4 列显示
set expandtab        " 点击制表符键时插入空格
set softtabstop=4    " 在点击制表符 / 回退键时，插入 / 删除 4 个空格
set shiftround       " 自动将缩进量设定为 shiftwidth 的倍数
set autoindent       " 将新行的缩进与前一行对齐
```

使用这些设定，每 79 个字符后就会插入新一行，缩进设定为每个制表符对应 4 个空格，并且如果当前在一个缩进语句中，下一行也会自动缩进为相同层级。

语法插件 python.vim 在 Vim 6.1 包含的语法文件基础上做了一些改进，插件 SuperTab 通过使用制表符键或任意其他自定义键使得代码补全更加简便。如果使用 Vim 来编写其他编程语言代码，则可以用名为 indent 的简便插件为 Python 源码文件处理缩进设置。

这些插件为 Python 开发提供了一个基础环境。如果你的 Vim 是以 +python 选项编译安装的（Vim 7 及更新的版本默认支持），则也可以使用插件 vim-flake8 在编辑器内进行静态代码检查，其提供函数 Flake8，执行 PEP8（*http://pypi.python.org/pypi/pep8/*）和 Pyflakes（*https://pypi.python.org/pypi/pyflakes/*），可被映射到 Vim 中的任意热键或期望动作。该插件会在屏幕底部显示错误信息，并提供一个简单的方式跳转到对应的代码行。

如果希望更简便些，那么可以在 .vimrc 中添加如下所示的一行内容，让 Vim 在每次保存 Python 文件时都调用 Flake8。

```
autocmd BufWritePost *.py call Flake8()
```

如果你已在使用 syntastic，那么可以设置让其在文件写入时运行 Pyflakes，并在 quickfix 窗口中显示错误和警告信息。下面是配置示例，同时状态栏中也展示了状态和警告信息。

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

Python-mode

Python-mode 是针对 Vim 中编写 Python 代码的一个复杂解决方案。如果需要下面罗列的

4 在 Windows 上，如果要定位 home 目录，那么打开 Vim 并输入 echo $HOME 即可。

任一特性，那就使用它（注意，它会让 Vim 启动更慢）。

- 异步的 Python 代码检查（pylint、pyflakes、pep8、mccabe），工具可任意组合。
- 基于 rope（*https://github.com/python-rope/rope*）进行代码重构和自动补全。
- 快速的 Python 代码折叠（可以隐藏 / 展示缩进块内的代码）。
- 支持 virtualenv。
- 能够搜寻 Python 文档，运行 Python 代码。
- PEP8（*http://pypi.python.org/pypi/pep8/*）错误自动修复。

Emacs

Emacs 是一个强大的文本编辑器，它虽然有图形化用户界面，但仍然可以直接在控制台运行，完全可编程，只要小小一点付出，就能构造成一个 Python IDE。“受虐狂”和 Raymond Hettinger[5] 都使用它。

Emacs 以 Lisp 语言编写实现，由 Richard Stallman、Guy L. Steele 和 Jr. 于 1976 年首次发布。Emacs 内置特性包括：远程编辑（通过 FTP）、日历、邮件发送 / 阅读，以及一个“心理医生”（点击 Esc 键，输入 x 字符，再输入 doctor 单词）。流行的插件包括：YASnippet，将自定义代码片段映射到按键；Tramp，用于调试。Emacs 用其自有的 Lisp 方言——elisp 进行扩展。

EmacsWiki 上的《使用 Emacs 进行 Python 编程》（*http://emacswiki.org/emacs/PythonProgrammingInEmacs*）一文为 Python 相关扩展包和配置提供了最佳建议，如果你是 Emacs 的老用户，那么推荐阅读。Emacs 新手则可以从 Emacs 官方教程（*https://www.gnu.org/software/emacs/manual/html_node/emacs/Intro.html*）开始学习。

目前，Emacs 的 Python 模式主要有三个。

- Fabián Ezequiel Gallina 的 python.el 现已和 Emacs 捆绑（版本 24.3 以上的），实现了语法高亮、缩进、移动、shell 交互，以及许多其他常见的 Emacs 编辑模式特性。
- Jorgen Schäfer 的 Elpy，目标是在 Emacs 内提供全功能的交互式开发环境，包括调试、语法检查（linter）及代码自动补全。
- Python 源码发行版的 Misc/python-mode.el 路径下有个模式文件可供选择（译注：Python 最新版本源码压缩包内已不提供该模式文件了，可以从 launchpad（*https://launchpad.net/python-mode*）上单独下载它）。其提供一些工具，支持通过语音、附加快捷键进行编程，并让用户能够建立一个完整的 Python IDE。

5 Raymond Hettinger 深受程序员的爱戴。如果人人都按照他推荐的方式写代码，那么这个世界就会美好得多。

TextMate

TextMate 是一个图形化用户界面的编辑器，根源于 Emacs，仅在 Mac OS X 系统上可用。其用户界面符合苹果公司软件的风格，以巧妙的方式展现了所有命令，既容易发现，又不突兀。

TextMate 以 C++ 语言编写实现，由 Allan Oddgard 和 Ciarán Walsh 首次发布。Sublime Text 可以直接导入 TextMate 的代码片段，微软的 Code 编辑器可以直接导入 TextMate 的语法高亮配置。

任何编程语言的代码片段都可以添加到捆绑组（bundled group）中，也可以使用 shell 脚本进行扩展：用户可以高亮某些文本作为标准输入，使用组合键 Cmd+|（管道符号）通过管道传递给脚本。脚本的输出会替换高亮部分的文本。

TextMate 为苹果的 Swift 和 Objectitve C 语言内置了语法高亮方案，并内置了 xcodebuild 接口（通过 Xcode 捆绑包）。TextMate 的资深用户使用该编辑器进行 Python 编程不会遇到什么问题。为苹果产品编程的新用户使用某些更新的跨平台编辑器可能会更好点，这些编辑器从 TextMate 上大量借鉴了其最受欢迎的特性。

Atom

按照 GitHub 上该项目创建者的说法，Atom 是一个面向 21 世纪的 hackable 文本编辑器。Atom 首次发布于 2014 年，用 CoffeeScript（JavaScript）和 Less（CSS）编写实现，构建在 Electron（前身为 Atom Shell）[6] 之上。Electron 是 GitHub 基于 io.js（现在已并入 node.js）和 Chromium 实现的一个应用程序外壳。

Atom 可通过 JavaScript 和 CSS 进行扩展，并且用户可添加任意编程语言的代码片段（包括 TextMate 风格的代码片段定义）。它良好地集成了 GitHub，自带本地扩展包管理功能，并提供了大量扩展包（大于两千个）。对于 Python 开发而言，推荐 Linter 扩展，配合 linter-flake8 使用。Web 开发者可能也会喜欢 Atom development server（*https://atom.io/packages/atom-development-server*），其运行一个小型 HTTP 服务，通过它开发者可以在 Atom 内预览 HTML。

Code

微软于 2015 年公布 Code 项目。它是 Visual Studio 家族内的一个免费闭源的文本编辑器，构建在 GitHub 的 Electron 之上，跨平台并具备类似 TextMate 的键绑定功能。

Code 自带一套扩展 API——用户可以在 VS Code 扩展市场（*https://code.visualstudio.*

6 Electron 是一个使用 HTML、CSS 和 JavaScript 构建跨平台桌面应用的平台。

com/docs/editor/extension-gallery）浏览已有扩展——并结合了 TextMate 和 Atom 的（Code 开发者认为的）精华部分，具备能与 VisualStudio 相媲美的代码自动补全功能，对 .Net、C# 和 F# 支持良好。

Visual Studio（Code 编辑器的姐妹款 IDE）仅在 Windows 上可用，而 Code 是跨平台的。

IDE

在开发大型复杂或者合作性项目时，许多开发者都会把文本编辑器切换成 IDE。表 3-2 介绍了一些流行 IDE 的显著特性。

表3-2　IDE一览

工具	可用性	使用理由
PyCharm/Intellij IDEA	• 开放 API/ 专业版本付费使用 • 开源 / 社区版本免费 • 支持 Mac OS X、Linux、Windows 平台	• 近乎完美的代码自动补全功能 • 对虚拟环境支持良好 • 在付费版本中，对 Web 框架支持良好
Aptana Studio 3/ Eclipse+ LiClipse+ PyDev	• 开源 / 免费 • 支持 Mac OS X、Linux、Windows 平台	• 你已爱上 Eclipse • 支持 Java 开发（LiClipse/Eclipse）
WingIDE	• 开放 API/ 免费试用 • 支持 Mac OS X、Linux、Windows 平台	• 卓越的调试器（基于 Web）——在此表罗列的 IDE 中是最好的 • 可使用 Python 进行扩展
Spyder	• 开源 / 免费 • 支持 Mac OS X、Linux、Windows 平台	• 对数据科学支持良好：IPython 集成并捆绑了 NumPy、SciPy 和 matplotlib • Anaconda、Python(x, y) 及 WinPython 等流行科学计算 Python 发行版的默认 IDE
NINJA-IDE	• 开源 / 欢迎赞助 • 支持 Mac OS X、Linux、Windows 平台	• 追求轻量 • 重点支持 Python 开发
Komodo IDE	• 开放 API/ 文本编辑器（Komodo Edit）开源 • 支持 Mac OS X、Linux、Windows 平台	• 支持 Python、PHP、Perl、Ruby、Node • 扩展基于 Mozilla 附加组件技术

续表

工具	可用性	使用理由
Eric（Eric Python IDE）	• 开源 / 欢迎捐赠 • 支持 Mac OS X、Linux、Windows	• 支持 Ruby 和 Python • 追求轻量 • 它是科学计算方面卓越的调试器，在一个线程调试的同时可以继续执行其他线程
Visual Studio（社区版）	• 开放 API/ 社区版本免费 • 专业或企业版本付费使用 • 仅支持 Windows 平台	• 与微软的编程语言和工具集成良好 • IntelliSense（代码自动补全）极其出色 • 提供项目管理和部署辅助功能，企业版本中还包括冲刺规划工具和清单模板 • 警告：只有企业版本（最昂贵）可以使用虚拟环境，其他的版本不能使用虚拟环境

全功能 IDE 不仅具有优秀的代码自动补全和调试工具，而且能在多个 Python 解释器之间（例如，从 Python 2 切换到 Python 3 再切换到 IronPython）快速切换，这个特性是全功能 IDE 备受欢迎的一个理由。表 3-2 中罗列的所有免费版 IDE，以及 Visual Studio 目前的所有版本都具有这一特性 [7]。

IDE 可能会免费提供的附加特性包括：与问题跟踪系统交互的工具、部署工具（例如，Heroku 和 Google 应用引擎）、协作工具、远程调试，以及配合使用 Web 开发框架（例如，Django）的额外特性。

PyCharm/IntelliJ IDEA

PyCharm 是深受开发者喜爱的 Python IDE。最主要原因是其近乎完美的代码自动补全工具，以及高质量的 Web 开发支持工具。科学计算社区的朋友们可能会推荐使用免费版本（不提供 Web 开发相关工具），因为对他们来说免费版本就够用了，但对科学计算来说更常见的选择是 Spyder。

PyCharm 由 JetBrains 公司开发，该公司也因 Intellij IDEA（一个与 Eclipse 竞争的专有 Java IDE）而知名。PyCharm（首次发布于 2010 年）和 IntelliJ IDE（首次发布于 2001 年）共享基础代码，PyCharm 的多数特性在 IntelliJ 中可通过免费的 Python 插件获得。

JetBrains 推荐使用 PyCharm 进行 Python 开发，因其 UI 更简单，但如果希望深入 Jython 函数内部进行跨语言代码浏览，或进行 Java 到 Python 的重构，则推荐使用 IntelliJ IDEA（PyCharm 可使用 Jython，但仅将其作为一个解释器选项，并不提供内省工具）。这两个

7 *https://github.com/Microso /PTVS/wiki/Features-Matrix*。

IDE 各自授权许可证，因此在购买之前先选择好。

IntelliJ 社区版和 PyCharm 社区版都已开源（Apache 2.0 许可证）并可免费使用。

Aptana Studio 3/Eclipse+LiClipse+PyDev

作为一个开放通用的 Java IDE，Eclipse 于 2001 年由 IBM 发布首个版本。Aleks Totic 于 2003 年发布支持 Python 开发的 Eclipse 插件，其后由 Fabio Zadrozny 接替负责开发工作。该插件是 Eclipse 上最流行的 Python 开发插件。

当许多人在论坛上鼓吹使用 IntelliJ IDEA 时，Eclipse 社区虽然没在网上反击，但 Eclipse 仍然是最常用的 Java IDE。许多流行工具（例如，Hadoop、Spark 以及一些专有工具）都为在 Eclipse 中开发提供操作指南和插件，某些 Python 开发者（有和 Java 编写的工具进行交互需求的开发者）对于这一点比较在意。

PyDev 的一个分支并入了 Aptana 的 Studio 3，该软件是 Eclipse 捆绑插件的一个开源套件，为 Python（及 Django）、Ruby（及 Rails）、HTML、CSS 及 PHP 开发提供 IDE。Aptanta 所有者 Appcelerator 的主要关注点是 Appcelerator Studio，它是以 HTML、CSS 和 JavaScript 进行应用开发的一个专有移动平台，要求按月授权许可证（只要你的应用还存活）。因此虽然对 PyDev 和 Python 的一般性支持还是有的，但不受重视。那也就是说，如果你是一个 JavaScript 开发者，喜爱 Eclipse，为移动平台开发应用，偶尔涉足 Python，特别是如果你在工作中使用 Appcelerator，那么 Aptana 的 Studio 3 是个不错的选择。

为了让 Eclipse 中多编程语言开发的体验更佳，并且容易获取完全的黑色系主题（例如，文本背景、菜单栏和边框都是黑色系的），LiClipse 出现了。它是一个 Eclipse 插件的专有套件，由 Zadrozny 开发；它的许可证的部分费用（可选的）用于保持 PyDev 完全免费和开源（EPL 许可证与 Eclipse 相同），其已捆绑 PyDev，因此 Python 用户不需要自己安装 PyDev。

WingIDE

WingIDE 是一个 Python 专用 IDE。它可能是仅次于 PyCharm 的第二大流行 Python IDE，可运行在 Linux、Windows 和 Mac OS X 平台上。

它的调试工具非常不错，并且包含 Django 模板的调试工具。它的调试器容易学习、内存占用小，这也是用户推荐 WingIDE 的理由。

Wingware 于 2000 年发布 Wing，其以 Python、C 和 C++ 编写而成。支持插件但尚未有

一个插件仓库，因此用户需要搜索其他人的博客或 GitHub 账号来查找已有的插件包。

Spyder

Spyder（Scientific python development environment），即科学计算 Python 开发环境，它特别适合与科学计算 Python 库打交道的开发工作。

Spyder 由 Carlos Córdoba 用 Python 编写而成，开源（MIT 许可证），提供代码自动补全、语法高亮、类和函数浏览器，以及对象内省特性。Spyder 的其他特性可通过社区插件获得。

Spyder 集成了 pyflakes、pylint 和 rope，并捆绑了 NumPy、SciPy、IPython 及 Matplotlib。而它自己又被流行的科学计算 Python 发行版 Anaconda、Python(x, y) 和 WinPython 捆绑发行。

NINJA-IDE

NINJA-IDE 名称来自递归缩写 Ninja-IDE is Not Just Another IDE，它是一个跨平台的 IDE，专为构建 Python 应用而设计，可运行在 Linux/X11、Mac OS X 和 Windows 平台上。可从 NINJA-IDE 的官网（*http://www.ninja-ide.org/*）下载适合各平台的安装器。

NINJA-IDE 以 Python 和 Qt 开发而成，开源（GPLv3 许可证），并追求轻量。它的特性有开箱即用、在运行静态代码检查器或调试时会高亮显示问题代码、能在浏览器里预览网页。NINJA-IDE 可以使用 Python 进行扩展，有一个插件库，它的理念是用户可以按需安装工具。

NINJA-IDE 邮件论坛的交流很活跃，社区里有很多西班牙语的发言者，包括核心开发团队。

Komodo IDE

ActiveState 公司开发的 Komodo IDE 是一个可运行在 Windows、Mac 和 Linux 上的商业化 IDE。该 IDE 的文本编辑器 KomodoEdit 是其开源替代方案（Mozilla 公共许可证）。

Komodo 由 ActiveState 公司于 2000 年首次发布，使用了 Mozilla 和 Scintilla 的代码库，可通过 Mozilla 的附加组件技术进行扩展。支持 Python、Perl、Ruby、PHP、Tcl、SQL、Smarty、CSS、HTML 及 XML。Komodo Edit 本身不带调试器，但可以通过插件获得。Komodo IDE 不支持虚拟环境，但允许用户自主选择使用 Python 解释器的类型。Komodo IDE 对 Django 的支持没有 WingIDE、PyCharm 和 Eclipse+PyDev 那么丰富。

Eric（Eric Python IDE）

Eric 是开源的，它已活跃开发十余年。Eric 用 Python 开发而成，基于 Qt GUI 工具包，集成 Scintilla 编辑器控件。Eric 以巨蟒剧团成员之一的 Eric Idle 的名字而命名，并向 Python 发行版捆绑发行的 IDLE IDE 致敬。

Eric 的特性包含源码自动补全、语法高亮、支持源码控制系统、支持 Python 3、集成一个 Web 浏览器、内置一个 Python shell、集成一个调试器，以及一个灵活的插件系统。不过它没有为 Web 框架提供额外工具。

与 NINJA-IDE 及 Komodo IDE 一样，Eric 也追求轻量。它的忠实用户认为它有最优秀的调试工具，能够在暂停调试一个线程的同时继续运行其他线程。如果想在该 IDE 中使用 Matplotlib 进行交互式绘图，则必须使用 Qt4 后端。

```
# 这两行必须先写
import matplotlib
matplotlib.use('Qt4Agg')

# 然后 pylot 将会使用 Qt4 后端
import matplotlib.pyplot as plt
```

Eric IDE 的最新文档的链接是 *http://eric-ide.python-projects.org/eric-documentation.html*。在 Eric IDE 相关网页上留下积极评论的用户几乎都来自科学计算社区（例如，天气建模或计算流体动力学）。

Visual Studio

在 Windows 上做微软产品开发工作的程序员会想用 Visual Studio。它是用 C++ 和 C# 语言编写的，首个版本于 1995 年发布。2014 年年底首个 Visual Studio 社区版向非商业开发者免费提供。

如果打算主要做企业软件的开发工作，并且使用微软产品如 C# 和 F#，那么 Visual Studio 就是最适合的 IDE。

请确保为 Visual Studio 安装了 Python 工具（PTVS），该工具是自定义安装选项列表中的一个复选框选项，默认为不勾选，与 Visual Studio 一同安装或者在 Visual Studio 之后安装皆可，安装说明见 PTVS 的 wiki 页面（*https://github.com/Microsoft/PTVS/wiki/PTVS-Installation*）。

增强型交互式工具

下面列举的工具可以提供增强体验的交互式 shell。集成开发与学习环境（IDLE，Integrated Development and Learning Environment）其实是一个 IDE，未包含在前一节，是因为多数人认为：与前面列举的其他 IDE 相比，它使用起来（对于企业级项目而言）功能还不够强大。然而，其用于教学则是极好的。Spyder 默认整合了 IPython，其他 IDE 也可以整合 IPython。这些增强型交互式工具不是要取代 Python 解释器，而是要让用户有更多的解释器 shell 可选择，并提供额外的工具和特性。

IDLE

IDLE（也是巨蟒剧团成员 Eric Idle 的姓氏），是 Python 标准库的一部分，随 Python 一起发行。

IDLE 由 Guido van Rossum（Python 的 BDFL，仁慈的独裁者）用 Python 编写而成，使用了 Tkinter GUI 工具包。虽然 IDLE 不适用于大型成熟的 Python 开发，但有助于试验短小的 Python 代码片段或尝试 Python 的各种特性。

IDLE 有以下特性。

- 一个 Python shell 窗口（解释器）。
- 一个多窗口的文本编辑器，可进行 Python 代码着色。
- 极简的调试能力。

IPython

IPython 有一个丰富的工具包，它帮助用户尽可能交互式地使用 Python。其核心组件有 5 个。

- 强大的 Python shell（基于终端和基于 Qt 的）。
- 基于 Web 的 Notebook，它的核心特性与终端 shell 一样，此外还支持富媒体、文本、代码、数学表达式、行内绘图。
- 支持交互式数据可视化（例如，经配置，Matplotlib 可以在新弹出窗口中绘图）以及图形化用户界面工具包的使用。
- 灵活、可嵌入的解释器，可载入用户自己的项目中使用。
- 提供高级交互式并行计算工具。

在终端 shell 或 Powershell 中输入以下命令来安装 IPython：

```
$ pip install ipython
```

bpython

bpython 是类 Unix 操作系统上 Python 解释器的一个备选界面，包含以下特性。

- 行内语法高亮。
- 自动缩进和自动代码补全。
- Python 函数参数列表提示。
- 提供重播功能，从内存中取出最后执行的那行代码并重新求值。
- 能够将输入的代码发送到 pastebin 从而将代码分享到网上。
- 能够将输入的代码保存到一个文件中。

在终端 shell 中输入以下命令来安装 bpython。

```
$ pip install bpython
```

环境隔离工具

本节将详细描述应用最广泛的环境隔离工具，从 virtualenv（用于隔离多个 Python 环境）开始，一直介绍到 Docker（用于虚拟化整个系统）。

这些工具为应用与其宿主环境之间提供不同级别的隔离，从而可以针对不同版本的 Python 和依赖库进行代码测试及调试，也可以实现一致的部署环境。

虚拟环境

Python 虚拟环境可以将不同项目的依赖环境保存在不同的位置。在同一个系统中为不同项目分别创建不同的 Python 环境，全局 site-packages 目录（用户全局安装的 Python 包保存于此）可以保持整洁，容易管理。并且用户可以同时开发维护两个项目，这两个项目依赖同一个第三方库，但版本可以不同，例如一个项目依赖 Django 1.3，另一个项目依赖 Django 1.0。

virtualenv 命令会创建一个独立目录，其中包含一个指向 Python 可执行文件的软链接、一个 pip 复制及一个 Python 库存放之处。一旦激活虚拟环境，独立目录的路径就会被添加到 PATH 环境变量中，虚拟环境解除之后，PATH 又会恢复到原来的状态值。通过命令行选项，也可以在虚拟环境中使用系统全局安装的 Python 版本和库。

虚拟环境一旦创建，便不可移动，可执行文件中的路径硬编码成当前虚拟环境 bin/ 目录中解释器的绝对路径。

创建并激活虚拟环境

在不同操作系统上，Python 虚拟环境的安装和激活方式有细微不同。

在 Mac OS X 和 Linux 上可以用先 python 参数来指定 Python 版本，然后用 activate 脚本来设置 PATH 环境变量，进入虚拟环境。

```
$ cd my-project-folder
$ virtualenv --python python3 my-venv
$ source my-venv/bin/activate
```

在 Windows 上应先设置系统的程序执行策略，允许运行本地创建的脚本[8]。以管理员身份打开 PowerShell，并输入以下命令完成设置。

```
PS C:\> Set-ExecutionPolicy RemoteSigned
```

对于出现的询问回复“Y”退出，再打开一个常规的 PowerShell，这样就能创建一个虚拟环境。

```
PS C:\> cd my-project-foler

PS C:\> virtualenv --python python3 my-venv
PS C:\> .\my-venv\Scripts\activate
```

向虚拟环境中添加依赖库

一旦激活了虚拟环境，搜索路径中找到的第一个 pip 可执行程序位于刚创建的 my-venv 目录中，它会将依赖库安装到下面的目录中。

- 在 POSIX[9] 兼容系统上：my-venv/lib/python3.4/site-packages。
- 在 Windows 上：my-venv/Lib/site-packages。

在激活的虚拟环境里，若要为其他人打包你的 Python 包或项目，可使用如下命令。

```
$ pip freeze > requirements.txt
```

这个操作会将当前已安装的所有 Python 包（也就是项目依赖）的名称及版本号写入文件 requirements.txt 中。在拿到 requirements.txt 文件后，协作者可以在他们的虚拟环境中输入以下命令安装所有依赖。

```
$ pip install -r requirements.txt
```

8 如果你愿意，那么也可以使用 Set-ExecutionPolicy AllSigned。

9 POSIX 是 Portable Operating System Interface of UNIX 的缩写，它是可移植操作系统接口。它包含一组 IEEE 标准，规定操作系统的行为应该如何：基础 shell 命令、I/O、线程及其他服务和工具的行为和接口。大多数 Linux 和 Unix 发行版都会尽可能兼容 POSIX，而 Darwin（Mac OS X 和 iOS 依赖的底层操作系统）自 Leopard（10.5）开始兼容它。通常所说的 POSIX 系统指的是尽可能兼容 POSIX 的系统。

pip 会安装罗列出来的依赖，如果存在冲突，则会覆盖子级包中的依赖说明。requirements.txt 中指定的依赖是用来设置整个 Python 环境的。在分发 Python 库时，为其设置依赖，最好是在 setup.py 文件中为 setup() 函数传递 install_requires 关键字参数。

注意，在虚拟环境外不要使用 pip install -r requirements.txt。如果这么干了，又恰好 requirements.txt 中有和系统中已安装的版本不同的依赖包，那么 pip 会使用 requirements.txt 中指定的版本覆盖已安装的库版本。

解除虚拟环境

输入以下命令可回到正常的系统设定。

```
$ deactivate
```

进一步的信息可查看虚拟环境文档（*https://github.com/kennethreitz/python-guide/blob/master/docs/dev/virtualenvs.rst*）、virtualenv 官方文档（*https://virtualenv.pypa.io/en/latest/userguide.html*）及 Python 打包的官方指南（*https://packaging.python.org/*）。因为 Python 3.3 及以上版本中随标准库分发的 pyenv 包不是要替代 virtualenv（实际上，它是 virtualenv 的一个依赖），所以上面这些操作指令对所有 Python 版本均有效。

pyenv

pyenv 允许用户同时使用多个版本的 Python 解释器，解决了不同项目要求不同版本 Python 的问题，但如果依赖库发生冲突（例如，要求不同的 Django 版本），那么还是需要使用虚拟环境。例如，可以安装 Python 2.7 来兼容某个项目，同时仍以 Python 3.5 为默认解释器。pyenv 不仅能用 CPython 的不同版本，而且能安装 PyPy、Anaconda、Miniconda、Stackless、Jython 及 IronPython 解释器。

pyenv 的工作原理是：将 Python 解释器和 pip、2to3 这类可执行程序的垫片版本填充到一个 shims 目录下。如果该目录被前置到 $PATH 环境变量中，那么可执行程序会被优先找到。垫片是一种传递函数，可以根据当前环境选择最恰当的函数来处理目标任务。例如，在系统寻找一个名为 python 的程序时，它会先到 shims 目录中查看，并使用垫片版本，垫片版本转而把命令传递给 pyenv。基于环境、*.python-version 文件，以及全局默认设置，pyenv 会决策出应该运行的 Python 版本。

虚拟环境中存在插件 pyenv-virtualenv，它可以自动创建不同环境，也允许我们使用已有的 pyenv 工具在不同环境之间切换。

Autoenv

Autoenv 提供一种轻量方案在 virtualenv 范围之外管理不同环境的配置。它会覆写 cd shell 命令，这样在切入一个包含 .env 文件（例如，设置 PATH 和数据库 URL 环境变量）的目录时，Autoenv 会自动激活环境。在切出这个目录时，又会取消之前应用的效果。不过它在 Windows Powershell 中不起作用。

在 Mac OS X 上可使用 brew 进行安装：

```
$ brew install autoenv
```

或者在 Linux 上安装：

```
$ git clone git://github.com/kennethreitz/autoenv.git ~/.autoenv
$ echo 'source ~/.autoenv/activate.sh' >> ~/.bashrc
```

打开一个新的终端 shell 就能看到效果了。

virtualenvwrapper

virtualenvwrapper 提供一组命令来扩展 Python 虚拟环境功能，方便控制管理。它将所有虚拟环境放在同一个目录下，并提供一些空的钩子脚本，可以在虚拟环境或者工程创建 / 激活之前或之后触发执行。例如，钩子可以通过执行某个目录下的 .env 文件设置一些环境变量。

对于这些钩子脚本，存在这样一个问题：用户必须以某种方式获取这些脚本，才能在另一台机器上完整地复制出整个环境。在一台共享开发服务器上，如果所有环境都放置在一个共享目录下供多个用户使用，那么这个问题就不存在了。

这里省略完整的 virtualenvwrapper 安装说明。在 Mac OS X 或 Linux 上安装完 virtualenv 后，在命令终端中输入以下命令：

```
$ pip3 install virtualenvwrapper
```

如果使用的是 Python 2，则执行 pip install virtualenvwrapper，并将下面这行内容添加到 ~/.profile 文件中。

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
```

将下面的代码添加到 ~/.bash_profile 文件或其他 shell 配置文件中。

```
source /usr/local/bin/virtualenvwrapper.sh
```

关闭当前终端窗口，再打开一个新的终端窗口来激活新配置文件，此时

virtualenvwrapper 就可以用了。

在 Windows 上，则以 virtualenvwrapper-win 替代 virtualenvwrapper。安装 virtualenv 之后输入：

```
PS C:\> pip install virtualenvwrapper-win
```

在两种平台上以下命令都是最常用的。

- mkvirtualenv my_venv，在 ~/.virtualenv/my_venv 文件夹内创建虚拟环境，或者在 Windows 上的命令行中输入 %USERPROFILE%\Envs 得到的目录路径，my_venv 即被创建在其内。通过环境变量 $WORKON_HOME 可以定制这个位置。
- workon my_venv，激活虚拟环境或者从当前环境切换到指定的虚拟环境。
- deactivate，停止使用虚拟环境。
- rmvirtualenv my_venv，删除虚拟环境。

virtualenvwrapper 对环境名称提供 Tab 键自动补全功能，在存在很多环境而且又不太能记得住环境名称的时候，这个功能确实很有所帮助。virtualenvwrapper 命令全列表记录了许多其他的实用功能（*http://virtualenvwrapper.readthedocs.io/en/latest/command_ref.html*）。

Buildout

Buildout 是一个 Python 框架，允许用户创建并组合构建工序（recipe）。构建工序是指可以包含任意代码的 Python 模块（通常是系统调用创建目录或者检查源代码并构建，以及将非 Python 部件加入项目，比如数据库或 Web 服务器）。使用 pip 安装 Buildout。

```
$ pip install zc.buildout
```

Buildout 项目在 requirements.txt 中包含 zc.buildout 以及需要的构建工序模块，或者直接将自定义的构建工序模块包含在源码内。项目的顶级目录中还会包含配置文件 buildout.cfg 和 bootstrap.py 脚本。如果输入 python bootstrap.py 执行脚本，那么它会读取配置文件来决定使用哪些构建工序模块，以及每个构建工序模块使用什么配置项（例如，特定的编译器标记和库连接标记）。

Buildout 为存在非 Python 部件的 Python 项目提供了可移植性，其他用户可以重建相同的环境。这与 Virtualenvwrapper 的钩子脚本不同，钩子脚本需要随 requirements.txt 文件一起复制传输才能重建相同的虚拟环境。

Buildout 包含用于安装 eggs[10] 的部件，更新版本的 Python 使用 wheels 替代 eggs，所以可以略过这些部件。详细信息推荐阅读 Buildout 教程（*http://www.buildout.org/en/latest/docs/tutorial.html*）。

Conda

Conda 类似于 pip、virtualenv 及 Buildout 的组合。它内置于 Anaconda Python 发行版中，是 Anaconda 的默认包管理器，可通过 pip 命令安装：

```
$ pip install conda
```

也可以通过 conda 命令来安装 pip：

```
$ conda install pip
```

两者默认从不同的仓库中提取软件包进行安装（pip 从 *http://pypi.python.org* 提取，而 Conda 则是从 *https://repo.continuum.io/* 提取），软件包使用的格式也不一样，因此这两个工具不能互换。

Continuum 公司（Anaconda 的创造者）创建了一个表，比较了 Conda、pip 和 virtualenv 的功能，参见 *https://conda.io/docs/_downloads/conda-pip-virtualenv-translator.html*。

conda-build 是 Continuum 公司提供的对等于 Buildout 的工具，在所有平台上都可以输入以下命令进行安装：

```
conda install conda-build
```

与 Buildout 类似，conda-build 配置文件格式名为构建工序，构建工序并不限于使用 Python 工具。与 Buildout 不同，conda-build 以 shell 脚本编写构建代码逻辑，而不是 Python，配置也是以 YAML[11] 格式设定，而不是 Python 的 ConfigParser 格式。

对于 Windows 用户而言，与 pip 和 virtualenv 相比，Conda 的最大优势是：以 C 扩展实现的 Python 库可能存在也可能不存在对应的 wheels，但在 Anaconda 包索引库里这些

10 eggs 文件是具有特定结构的一种 ZIP 文件，包含发布的内容。自 PEP 427（*https://www.python.org/dev/peps/pep-0427/*）公布开始，eggs 已被 wheels 取代。它们均由非常流行的（目前也是事实标准的）打包库 Setuptools 引入，Setuptools 为 Python 标准库的 distutils 包提供了便利的接口。可以通过阅读《Python 打包用户指南》的 Wheel vs Eggs（*https://packaging.python.org/discussions/wheel-vs-egg/*）部分的内容来了解两种打包格式的区别。

11 YAML 即 YAML 不是标记语言（YAML Ain't Markup Language）的递归缩写，是一种兼顾人类可读性和机器可读性的标记语言。

库几乎都存在。并且如果一个包通过 Conda 不可用，那么也许可以先安装 pip，然后从 PyPI 上获取并安装 Conda。

Docker

Docker 与 virtualenv、Conda、Buildout 一样，可以帮助实现环境隔离，但它不是提供一个虚拟环境，而是提供一个 Docker 容器。与虚拟环境相比，容器能提供更好的隔离。例如，可以同时运行多个容器，其中每个容器具有不同的网络接口、防火墙规则，以及不同的主机名。这些运行的容器由 Docker Engine 管理，协调对底层操作系统的访问。如果在 Mac OS X、Windows 或远程主机上运行 Docker 容器，则还需要 Docker Machine，其负责与运行 Docker Engine 的虚拟机 [12] 进行交互。

Docker 容器最初是基于 Linux 容器的，而 Linux 容器最初又与 shell 命令 chroot 相关。chroot 是一种系统级别的 virtualenv 命令版本：它能够让根目录（/）位于用户指定的一个路径上，而不是真正的根路径上，从而提供一个完全分离的用户空间。

不过 Docker 现在不再使用 chroot，甚至不再使用 Linux 容器（Docker 镜像的世界也因而可以包含 Citrix 和 Solaris 机器），但 Docker 容器所做的仍然是相同的事情。其配置文件名为 Dockerfile，基于它构建出来的 Docker 镜像可以托管在 Docker 包仓库 Docker Hub（类似于 PyPI）上。

与 Buildout 或 Conda 创建的环境相比，Docker 镜像经正确配置后占用的空间更小。因为 Docker 使用 AUFS 联合文件系统存储镜像的变化差异，而不是整个镜像，所以如果想要针对某个依赖的多个版本构建并测试软件包，则可以先创建一个基础 Docker 镜像，内含一个包含所有其他依赖的虚拟环境 [13]（或 Buildout 环境、Conda 环境），然后从这个基础镜像继承出所有其他镜像，在最后一层上仅添加单个变化的依赖。这样，所有派生容器共享基础镜像的内容，区别仅在包含了不同的新库。详细内容可以登录 *https://docs.docker.com/* 查看 Docker 文档。

12 虚拟机是一个模拟计算机系统的应用，在宿主计算机上，模拟目标硬件提供目标操作系统。

13 Docker 容器内部的虚拟环境会隔离你的 Python 环境，保留系统的 Python，以免破坏支撑应用的某些工具。建议不要通过 pip（或其他安装工具）在系统的 Python 目录中安装任何东西。

第 2 部分

步入正题

至此，Python 解释器、虚拟环境及编辑器或 IDE 都已准备好了，万事俱备步入正题。本部分不介绍 Python 的语言基础，附录 A 中的 Python 学习材料一节罗列了大量优秀的学习资源，这些资源都是教你学习语言基础的。相反，希望你读完这部分内容以后，感觉像一个真正的 Python 业内人士一样“天下我有”，学到社区中一些最优秀的 Python 高手才知道的窍门。这部分包含以下 3 章。

第 4 章，编写高质量的代码：从代码风格、约定、习语及陷阱等多方面简要介绍 Python 编程的最佳实践，助你快速晋级为 Python 高手。

第 5 章，阅读高质量的代码：从一些知名 Python 库中摘录部分源码，带你一起阅读学习，希望能够鼓舞你阅读更多的高质量代码。

第 6 章，交付高质量的代码：简要地介绍 Python 打包技术权威组织（PyPA），以及如何将自己的 Python 库打包发布到 PyPI 上，还会介绍一些构建交付可执行文件的技术方案。

第 4 章

编写高质量的代码

本章重点介绍编写高质量 Python 代码的最佳实践。此外，回顾一些编码风格约定（第 5 章会用到），简要介绍日志记录的最佳实践，并列出了常用开源许可证之间的一些主要差异。这些有助于你编写易用易扩展的代码。

代码风格

富有经验的 Python 开发者被称为 Pythonista，他们都庆幸存在 Python 这样一种如此易读易理解的编程语言，从未编写过代码的人阅读 Python 程序的源码，也能理解程序做了什么。可读性是 Python 语言的设计核心，因为通常情况下，阅读代码的需求比编写代码的需求更加频繁。

Python 代码易于理解的原因之一是它有相对完整的代码风格指南集（收录于 PEP 20 和 PEP 8 两篇 Python 增强提案中）和 Pythonic 风格。当 Pythonista 指出某段代码不符合 Pythonic 风格时，这通常意味着这些代码未遵循通用编码规范，也没使用最可读的方式来表达它的意图。当然，盲目地保持一致是头脑简单的表现 [1]。一味地遵循 PEP 可能会破坏代码的可读性和易懂性。

PEP 8

PEP 8 是 Python 代码风格事实上的标准指南，它涵盖命名约定、代码结构、空白（用制表符还是空格符），以及其他代码风格的知识点。

强烈推荐读者读一读 PEP 8。整个 Python 社区都在尽其所能地遵循 PEP 8 文档中的风格

1 出自爱默生的《自立》一书，PEP 8 中引用这句话来支持：是否遵循风格指南依赖于程序员自己的判断。例如，和项目中的其他代码及已有的约定保持一致性比遵循 PEP 8 更重要。

指南。有些项目可能会时不时地偏离它，而另一些项目（如 Requests）可能会建议做一些调整。编写符合 PEP 8 的 Python 代码通常是个好主意，它可以帮助大家在一起开发项目时最大程度保持代码风格的一致性。PEP 8 非常明确详细，可以编写程序来自动检查。pep8 是一个命令行程序，能够检查代码是否符合 PEP 8 风格。在终端中，运行如下命令来安装 pep8。

```
$ pip3 install pep8
```

运行 pep8 可能会看到的各种提示如下所示。

```
$ pep8 optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

PEP 8 描述的大多数问题的修复方法都非常简单明确。Requests 代码风格指南（*http://bit.ly/reitz-code-style*）给出了优雅代码和糟糕代码的示例。相对于 PEP 8 的风格，Requests 代码风格指南略有修改。

linter（代码风格静态检查工具）通常间接使用 pep8，因此，可以先安装其中一个，然后在编辑器或者 IDE 中检查代码风格。autopep8 可以把代码自动重新格式化为 PEP 8 风格，安装方式如下所示。

```
$ pip3 install autopep8
```

如下所示，使用 autopep8 来就地（覆盖原始文件）格式化文件。

```
$ autopep8 --in-place optparse.py
```

移除 --in-place 参数将使程序把修改后的代码直接输出到控制台（或者重定向到其他文件）。--aggressive 参数将执行更多的实质性修改，多次使用效果更佳。

PEP 20（又名 Python 之禅）

PEP 20 是编写 Python 程序的指导准则，在 Python shell 中输入 import this 就能看到。虽然名字是 PEP 20，但实际上它只包含了 19 条（最后一条还没写下来）。

Barry Warsaw 在博文 *Import This and the Zen of Python* 中记录了 Python 之禅的真实历史来源。

Tim Peter[2] 的 Python 之禅

优美胜于丑陋。

明确胜于隐晦。

简单胜于复杂。

复杂胜于难懂。

扁平胜于嵌套。

留白胜于紧凑。

可读性很重要。

特例也并不能特殊到可以违背这些原则。

虽然实用性胜过纯粹性。

错误不应被默默地忽略。

除非你明确地忽视。

面对歧义，不要尝试去猜测。

应该有一种——最好是仅有一种——明显的处理方式。

一开始那种方式并非显而易见，除非你是 Python 之父。

做好过不做。

不假思索就动手还不如不做。

如果实现很难解释，那就不是个好思路。

如果实现易于解释，则可能是个好思路。

命名空间是个绝妙的主意，我们要多多利用它！

对于每句箴言实际应用的例子，详见 Hunter Blank 的 *PEP 20(The Zen of Python) by Example* 一文。Raymond Hettinger 也在题为“Beyond PEP 8: Best Practices for Beautiful, Intelligible Code”的演讲中介绍了这些原则的绝妙应用。

一般性建议

本节包含了无争议且易于接受的风格理念，通常也适用于 Python 以外的编程语言。其中

2 Tim Peters 是一个忠实的 Python 用户，并最终成为持续高产的 Python 核心开发者之一（他设计实现了 Python 的排序算法 Timsort，参见 *https://en.wikipedia.org/wiki/Timsort*），他也经常现身网络。曾一度谣传他是 Richard Stallman AI 程序 stallman.el 的一个长期运行的 Python 移植版本。这一阴谋论在 20 世纪 90 年代后期首次出现在一个电子论坛上（*https://www.python.org/doc/humor/#the-other-origin-of-the-great-timbot-conspiracy-theory*）。

一些建议直接来自 Python 之禅，另一些则是常识。当面对多种 Python 表达方式时，这些风格理念有助于我们选择最浅显直白的方式来呈现代码。

明确胜于隐晦

虽然 Python 中存在各种编码方式，但是我们提倡用最直接明了的编码方式。

糟糕的写法	优雅的写法

`def make_dict(*args):`
`  x, y = args`
`  return dict(**locals())`
	`def make_dict(x, y):`
	`  return {'x': x, 'y': y}`

很显然，在上述优雅的代码中，x 和 y 由调用者传入，然后明确以字典形式返回。判断代码写得是否优雅的一个经验法则是：其他开发者是否能只阅读函数的首行和末行就能理解函数的作用。在糟糕的代码中则没有这么直观（当然，如果函数只有两行代码，也非常容易理解）。

留白胜于紧凑

一行只写一条语句。某些复合语句，比如列表解析，表达能力强，也允许并支持单行编写，不过对于相互独立的多条语句，建议分多行代码来写。这样一条语句在多个版本中的变更差异（diff）[3] 更容易理解。

糟糕的写法	优雅的写法
`print('one');print('two')`	`print('one')` `print('two')`
`if x == 1: print('one')`	`if x == 1:` `  print('one')`

`if (<complex comparison> and`
`  <other complex comparison>):`
`  # do something`
	`cond1 = <complex comparison>`
	`cond2 = <other complex comparison>`
	`if cond1 and cond2:`
	`  # do something`

对于 Pythonista 来说，可读性比代码多几个字节（两个 print 函数在同一行的语句）或多几微秒的运行时间（拆成多行后额外的条件语句执行）更重要。当一个团队为开源代码做贡献时，写得好的代码修改历史会更容易阅读，因为一行的修改只影响一个地方。

3 diff 是一个命令行工具，可用于识别并展示两个文件间的不同之处。

错误不应被默默地忽略，除非你明确地忽视

Python 使用 try 语句来处理异常。如下忽略异常的例子，来自 Ben Gleitzman 的 HowDoI 库：

```
def format_output(code, args):
    if not args['color']:
        return code
    lexer = None

    # 使用 Stack Overflow 的标签（tags）和查询（query）参数来尝试找到一个词法分析器
    for keyword in args['query'].split() + args['tags']:
        try:
            lexer = get_lexer_by_name(keyword)
            break
        except ClassNotFound:
            pass

    # 如果上面未找到任何词法分析器，则使用代码（code）来猜一个
    if not lexer:
        lexer = guess_lexer(code)

    return highlight(code,
                     lexer,
                     TerminalFormatter(bg='dark'))
```

HowDoI 软件包提供一个命令行脚本，在因特网上查询（默认在 Stackoverflow 上查询）如何完成一个特定的编程任务，然后在终端中输出结果。format_output() 函数先根据问题的标签查找到一个词法分析器（也称为分词器，python、java、bash 这类标签可以用来确定应该使用哪个词法分析器来切分代码并着色），然后应用语法高亮。如果查找失败，则从代码本身来推断是什么语言。当程序执行到 try 语句时，有三条可继续执行的路径。

- 执行流程进入 try 子句（try 和 except 之间的所有内容），成功找到词法分析器，跳出循环，函数返回词法分析器高亮后的代码。
- 没找到词法分析器，抛出 ClassNotFound 异常，但是异常被捕捉后什么也不做。循环继续执行直到正常结束或者找到词法分析器。
- 如果发生其他异常（比如 KeyboardInterrupt），而没有被捕捉到，则会被抛到顶层，终止执行。

Python 之禅中“错误不应被默默地忽略”不鼓励使用过度的错误捕捉。可以在一个单独的终端中尝试下面的代码，一旦理解了其中关键点，以后就不会再编写出这种难以关闭的程序了。

```
>>> while True:
```

```
...     try:
...         print("nyah", end=" ")
...     except:
...         pass
```

或者，还是不要尝试了吧。没有任何指定异常的 except 语句将会捕获所有异常，包括 KeyboardInterrupt（POSIX 终端中的 Ctrl+C 组合键），然后忽略它。因此它将忽略所有你尝试给出的关闭程序的中断指令。它也不仅仅是中断问题，宽泛的异常语句会掩盖 bug，然后造成一些难以诊断的问题。再次重申，不要让错误被默默地忽略：始终明确地指定要捕捉什么异常，然后仅仅处理这些异常。如果只是想记录或者确认异常然后重现它，像下面的代码片段这样，也没问题，只是别让错误被悄无声息地忽略（不处理或者不重新抛出）。

```
>>> while True:
...     try:
...         print("ni", end="-")
...     except:
...         print("An exception happened. Raising.")
...         raise
```

函数参数使用起来应该符合直觉

API 设计中你的选择将决定下游开发者的使用体验。将参数传递给函数有 4 种不同的方式：

```
                ❶           ❷          ❸          ❹
def func(positional, keyword=value, *args, **kwargs):
    pass
```

❶ 位置参数是强制性的，且没有默认值。

❷ 关键字参数是可选的，有默认值。

❸ 任意数量参数列表是可选的，没有默认值。

❹ 任意数量关键字参数字典是可选的，没有默认值。

下面是每一种参数传递方式的使用要点。

位置参数

> 如果函数参数不多，且是函数完整含义的一部分，那么就使用这种方式，参数顺序遵从语义顺序即可。例如，对于 send(message,recipient) 或 point(x,y) 函数，记住函数要求两个参数及参数顺序，对函数使用者而言并不困难。

用法反模式：调用函数时，可以使用参数名称，同时也可以改变参数顺序，例如，send(recipient="World", message="The answer is 42.") 和 point(y=2, x=1)，但是这样降低了可读性，增加了不必要的冗余。使用更直接的调用方式 send("The answer is 42", "World") 和 point(1, 2)。

关键字参数

如果一个函数的位置参数不止两三个，那么函数签名将难以记忆，带默认值的关键字参数就派上用场了。例如，一个功能更完整的 send 函数，签名为 send(message,to,cc=None,bcc=None)。其中 cc 和 bcc 可选，如果没有其他值传递给它们，它们的值就为 None。

用法反模式：调用函数时，可以只按照参数定义的顺序传参，而不显式地指定参数名，例如 send("42", "Frankie", "Benjy", "Trillian")，同时给 Trillian 发送密件。也可以指明参数名称但传参顺序不同于定义，例如 send("42", "Frankie", bcc= "Trillian", cc="Benjy")。如果没有其他充分的理由，那么最好还是使用最接近函数定义的形式 send("42", "Frankie", cc="Benjy", bcc="Trillian")。

不假思索就动手还不如不做

通常移除一个为了“以防万一”但看起来从未使用过的可选参数（及其在函数中的相关逻辑），比新增一个需要的可选参数及其相关逻辑更难！

任意数量参数列表

任意数量参数列表以 *args 结构来定义，表示一组可变数量的位置参数。在函数体中，args 是一个元组，包含所有额外传入的位置参数。例如，一个收件人一个参数这样来调用 send(message,*args) : send("42", "Frankie", "Benjy", "Trillian")。在函数体中，args 相当于 ("Frankie", "Benjy", "Trillian")。Python 的 print 函数就是此类型的一个好例子。

警告：如果函数接收的是一系列性质相同的参数，那么使用列表或其他任何序列结构作为参数，语义更加明确。在这里，如果 send 有多个接收者，那么将它定义成 send(message, recipients) 含义更明确，然后调用 send("42", ["Benjy", "Frankie", "Trillian"])。

任意数量关键字参数字典

任意数量关键字参数字典以 **kwargs 结构来定义，向函数传递一组任意的命名参数。

在函数体中，kwargs 是一个字典，包含所有传递给函数但没被函数签名中的关键字参数匹配到的命名参数。这个特性的一个应用例子是日志记录。不同级别的日志格式器可以随意按需获取信息，而不会对用户造成不便。

警告：与 *args 的理由类似，kwargs 这类强大的技术应该用在确实需要之处。如果函数的意图可以通过更简单更清晰的结构来充分表达，那么不应该使用这类技术。

如果有其他名称更符合变量含义，则可以而且应该替换 *args 和 **kwargs 这种变量名称。

哪些参数是位置参数、哪些参数是可选的关键字参数，以及决定是否使用更高级的任意数量参数传递技巧，全凭编写函数的程序员决定。虽然如此，应该有且最好仅有一种明确直白的方式来定义函数。当你编写的 Python 函数具备以下特点时，其他用户都会感谢你的工作。

- 易读（函数名称和参数都无须解释）。
- 易改（添加新的关键字参数不会破坏代码其他部分）。

如果实现很难解释，那就不是个好思路

对于技术高手而言，Python 是一个非常强大的工具，自带非常丰富的钩子和工具，允许你使用各种技巧。例如，Python 可以做如下事情。

- 改变对象创建和实例化的过程。
- 改变 Python 解释器导入模块的过程。
- 在 Python 中嵌入 C 语言程序。

当然，所有的这些做法各有各的弊端，使用最直截了当的方式来实现目标通常是更好的方法。使用这些语言结构最大的弊端是牺牲了可读性，因此只有在与代码可读性相比，还有更重要的目的时，才值得这样做。许多代码分析工具，比如 pylint 或者 pyflakes 都不能解析这种代码。

Python 开发者应该了解这些近乎无限的技术可能性，因为它能让大家相信没有解决不了的问题。然而，审时度势，行有所止也非常重要。

真正的 Pythonista，就像功夫大师那样，仅用一根手指就能杀人，但他们从不会那么做。

我们都是负责任的用户

如前所述，Python 中可用的技巧很多，其中一些会带来风险。举例来说，使用对象的代码可以重写对象的属性和方法，Python 里没有“private”关键字。这种哲学，用一句话来表达，即“我们都是负责任的用户”。Java 这样高度防御性的语言则截然相反，提供了很多机制来防止任何可能的误用。

不过，这并不意味着 Python 中没有私有属性，也不意味着无法进行恰当的封装。Python 社区更倾向于依赖一组约定来表明某些代码元素不应该被直接访问，而不是依赖不同开发者在各自的代码之间竖立起一堵一堵的墙。

私有属性和实现细节的主要约定是为所有“内部”变量名称加“_”前缀（例如，sys._getframe）。如果使用方代码打破了这个规则并访问了这些变量，那么任何由代码修改引发的不当行为或问题都由使用方负责。

任何不开放给外部使用的方法或属性，都应该带上下画线前缀。这样代码模块的职责划分更好，也方便以后对已有代码进行修改，随时都可以将私有属性公有化，但是把公有属性私有化可能会困难得多。

尽量仅在一处返回函数结果

复杂函数的函数体中存在多处返回语句的情况并不少见。然而，为了保持代码意图清晰，维持可读性，函数体中的返回点越少越好，并且返回值意义要明确。

函数退出有两种情况：出现错误或者函数正常执行结束后返回结果。当函数不能正确执行时，最好返回 False 或者 None。这种情况下，为保持函数结构扁平化，在检测到错误上下文后最好尽早从函数中返回，因错误而返回语句后的所有代码均是假设当前已满足后续计算函数结果的条件。因此有多个返回语句经常是必要的。

不过，还是尽可能保持单个退出点。在调试函数时，多个返回点将使你很难分清哪个是当前结果的返回点。强制函数出口点只有一个地方有助于提取一些代码路径，因为存在多个退出点也可能暗示了这里需要重构。下面的例子也不是什么糟糕的写法，不过如代码注释所言，它还可以写得更清楚一些。

```
def select_ad(third_party_ads, user_preferences):
    if not third_party_ads:
        return None # 抛出一个异常可能会更好
    if not user_preferences:
        return None # 抛出一个异常可能会更好
    # 一些复杂的代码，给定一些广告候选项和个人偏好
    # 计算出最佳广告
```

```
    # 抵住诱惑，不要在此处判断已成功获取最佳广告并返回
    if not best_ad:
        # 计算最佳广告的 B 计划
    return best_ad  # 仅在一处返回结果有助于维护代码
```

约定

约定，惠及众人，但也不是做事的唯一方法。下面展示的是一些使用最普遍的约定，推荐它们是因为它们更易阅读。

检查相等性的替代方法

当不必明确将一个值与 True、None 或者 0 做比较时，可以把这个值直接应用到 if 条件语句，例子如下（阅读这段真值测试文档，了解 Python 把哪些值视为假值）。

糟糕的写法	优雅的写法

`if attr == True:`
`    prit 'True!'`
	`# 直接检查值`
	`if attr:`
	`    print 'attr is truthy!'`
	
	`# 或检查条件相反`
	`if not attr:`
	`    print 'attr is falsey!'`
	
	`# 但如果只想值为 True`
	`if attr is True:`
	`    print 'attr is True'`

`if attr == None:`
`    print 'attr is None!'`
	`# 或显式检查值为 None`
	`if attr is None:`
	`    print 'attr is None!'`

访问字典元素

使用 x in d 语法而不是 dict.has_key 方法，或者给 dict.get() 传递一个默认值。

糟糕的写法	优雅的写法
`>>> d = {'hello': 'world'}`	`>>> d = {'hello': 'world'}`
`>>>`	`>>>`
`>>> if d.has_key('hello'):`	`>>> print d.get('hello', 'default_value')`
`...  print(d['hello']) # 输出 'world'`	`world`
`... else:`	`>>> print d.get('howdy', 'default_value')`
`...  print('default_value')`	`default_value`
`...`	`>>>`
`world`	`>>> # 或者`
	`... if 'hello' in d:`
	`...  print(d['hello'])`
	`...`
	`world`

操作列表

列表解析提供了一个强大而简明的方式来操作列表（详情请登录 *http://docs.python.org/tutorial/datastructures.html#list-comprehensions* 阅读 Python 官方教程中的相关内容）。map() 和 filter() 函数也可以使用一种不同且更简明的语法来操作列表。

标准循环	列表解析
`# 过滤出大于 4 的元素`	`# 列表解析，更易读易理解`
`a = [3, 4, 5]`	`a = [3, 4, 5]`
`b = []`	`b = [i for i in a if i > 4]`
`for i in a:`	
`  if i > 4:`	`# 或者`
`  b.append(i)`	`b = filter(lambda x: x > 4, a)`
`# 为所有列表元素值加上 3`	`# 这种情况，也更易读易理解`
`a = [3, 4, 5]`	`a = [3, 4, 5]`
`for i in range(len(a)):`	`b = [i + 3 for i in a]`
`  a[i] += 3`	
	`# 或者`
	`a = map(lambda i: i + 3, a)`

使用 enumerate() 来维护列表元素位置索引。相比手动创建计数器，enumerate() 更易读，而且对迭代器做了更好的优化。

```
>>> a = ["icky", "icky", "icky", "p-tang"]
>>> for i, item in enumerate(a):
...     print("{i}: {item}".format(i=i, item=item))
...
0: icky
1: icky
2: icky
3: p-tang
```

代码续行

当代码的逻辑行长度超过可接受的物理行长度限制[4]时，需要将它分成多个物理行。如果代码行的最后一个字符是反斜线，那么 Python 解释器会把这些连续的行拼接在一起，有时这一写法有用。因为形式上的脆弱性，即反斜线后添加空白字符可能会破坏代码，产生不可预料的结果，所以通常应避免使用。

更好的方法是将代码元素包含在圆括号内。左侧以一个未闭合的括号开头，Python 解释器会将接下来的所有行连接在一起，直到遇到闭合括号。

糟糕的写法	优雅的写法

`french_insult = \`
`"Your mother was a hamster, and \`
`your father smelt of elderberries!"`
	`french_insult = (`
	`"Your mother was a hamster, and"`
	`"your father smelt of elderberries!"`
	`)`

`from some.deep.module.in.a.module \`
`import a_nice_function, \`
`another_nice_function, \`
`yet_another_nice_function`
	`from some.deep.module.in.a.module import (`
	`a_nice_function,`
	`another_nice_function,`
	`yet_another_nice_function`
	`)`

通常情况下，分割长逻辑行意味着此行代码同时做了太多的事，这可能会降低可读性。

4 根据 PEP 8，行的最大长度为 80 个字符，也有其他规范将行的最大长度限制为 100 个字符。对你来说，听你老板的就行了。不过说实话，任何人如果曾站在服务器机架旁使用终端调试过代码，都会毫不犹豫地支持 80 个字符的长度限制（这样，在终端中代码不会折行）。在 Vi 中，考虑到行号，行的长度限制一般为 75~77 个字符。

习语

虽然，通常有且只有一种显然的方式来编写地道（或者说 Pythonic）的代码，但是对于 Python 初学者而言这些方式并不显然（除非他们是荷兰人[5]）。因此，良好的习语必须刻意习得。

解包

如果知道一个列表或元组的长度，则可以利用解包来为其中的元素分配名称。例如，在 split() 和 rsplit() 中可以指定字符串分割次数，一个赋值语句的右侧可以只分割一次（例如，分割成文件名和扩展名），左侧可以按照正确的顺序同时包含两个目标变量，如下所示。

```
>>> filename, ext = "my_photo.orig.png".rsplit(".", 1)
>>> print(filename, "is a", ext, "file.")
my_photo.orig is a png file.
```

也可以使用解包来交换变量值。

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

也可以嵌套解包。

```
a, (b, c) = 1, (2, 3)
```

在 Python 3 中，PEP 3132（*https://www.python.org/dev/peps/pep-3132/*）引入了一个新的加强版解包方法。

```
a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]

a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4
```

忽略一个值

如果解包时需要赋值而又不需要其中某个值，那么可以使用双下画线（__）。

```
filename = 'foobar.txt'
basename, __, ext = filename.rpartition('.')
```

许多 Python 风格指南都建议使用单下画线（_）来抛弃值，而不是此处推荐的双下画线（__）。单下画线的问题在于它通常被用作 gettext.gettext() 函数的别名，同时在交互模式下用来保存上一次操作的值。使用双下画线同样清晰方便，还避免了在任何情况下意外覆盖单下画线变量造成的风险。

5 引自 Python 之禅第 14 句。荷兰人 Guide 是仁慈的独裁者。

创建一个包含 N 个相同对象的列表

使用 Python 列表的 * 操作符来创建一个包含相同不可变元素的列表。

```
>>> four_nones = [None] * 4
>>> print(four_nones)
[None, None, None, None]
```

但是注意可变对象。因为列表是可修改的，* 操作符将创建一个包含 *N* 个指向同一列表的列表，这可能不是你想要的，那么可以使用列表解析。

糟糕的写法	优雅的写法
`>>> four_lists = [[]] *4`	`>>> four_lists = [[] for __ in range(4)]`
`>>> four_lists[0].append("Ni")`	`>>> four_lists[0].append("Ni")`
`>>> print(four_lists)`	`>>> print(four_lists)`
`>>> [['Ni'], ['Ni'], ['Ni'], ['Ni']]`	`>>> [['Ni'], [], [], []]`

创建字符串的一种常见惯用法是在空字符串上使用 str.join()。这个方法同样适用于列表和元组。

```
>>> letters = ['s', 'p', 'a', 'm']
>>> word = ''.join(letters)
>>> print(word)
spam
```

有时需要通过数据集查找。看看对于列表和集合这两个数据结构如何做查找，以如下代码为例：

```
>>> x = list(('foo', 'foo', 'bar', 'baz'))
>>> y = set(('foo', 'foo', 'bar', 'baz'))
>>>
>>> print(x)
['foo', 'foo', 'bar', 'baz']
>>> print(y)
{'foo', 'bar', 'baz'}
>>>
>>> 'foo' in x
True
>>> 'foo' in y
True
```

对于列表和集合成员的布尔测试，虽然看起来区别不大，但是由于 foo in y 借助了哈希

表特性(Python 中集合以及字典以哈希表来实现)[6],因此这两个例子的查找性能是不同的。对于列表，Python 将逐个遍历匹配其中的元素，非常耗时（集合越大，时间越长），而集合使用哈希查询可以很快找到键值。另外，集合和字典会丢弃重复项，这就是为什么字典不能有两个相同的键。更多详情请阅读 Stack OverFlow 网上关于 list 与 dict 的讨论（*http://stackoverflow.com/questions/513882*）。

异常安全的上下文

在异常可能出现时，通常使用 try/finally 语句管理资源，比如文件或者线程锁。PEP 343（*https://www.python.org/dev/peps/pep-0343/*）为 Python 引入了 with 语句和上下文管理协议（Python 2.5 或更高版本）来替代 try/finally 语句，with 语句和上下文管理协议使得代码可读性更高。该协议包含两个方法：__enter__() 和 __exit__()，如果一个对象实现了这两个方法，就能用在 with 语句中，如下所示。

```
>>> import threading
>>> some_lock = threading.Lock()
>>>
>>> with some_lock:
...     print(
...         "Look at me: I design coastlines.\n"
...         "I got an award for Norway."
...     )
...
```

没有 with 语句时按照下面这样写。

```
>>> import threading
>>> some_lock = threading.Lock()
>>> some_lock.acquire()
>>> try:
...     print(
...         "Look at me: I design coastlines.\n"
...         "I got an award for Norway."
...     )
... finally:
...     some_lock.release()
```

标准库模块 contextlib 提供了额外的工具来帮助将函数变成上下文管理器，强制执行对象的 close() 方法，抑制异常（Python3.4 或更高版本），重定向标准输出和错误流

6 这也就是为什么只有可哈希的对象才能存入集合或者用作字典的键。要想自己定义的 Python 对象可哈希，需要定义一个 object.__hash__(self) 成员函数，函数返回一个整数。比较结果为相等的对象必须具有相同的哈希值。更多详情，请阅读 Python 文档（*https://docs.python.org/3/reference/datamodel.html#object.__hash__*）。

（Python3.4、Python3.5 或更高版本）。下面是使用 contextlib.closing() 的一个例子。

```
>>> from contextlib import closing
>>> with closing(open("outfile.txt", "w")) as output:
...     output.write("Well, he's...he's, ah...probably pining for the
fjords.")
...
56
```

因为处理文件 I/O 的对象已经定义了 __enter__() 和 __exit__() 方法[7]，所以可以直接用在 with 语句中，不需要 closing。

```
>>> with open("outfile.txt", "w") as output:
    output.write("PININ' for the FJORDS?!?!?!? "
        "What kind of talk is that?, look, why did he fall "
        "flat on his back the  moment I got 'im home?\n"
    )
...
123
```

常见陷阱

从大部分语言特性来看，Python 的目标是成为一门简洁一致的语言，然而，还是有一些情况会困扰新手。

其中某些情况是有意为之，但可能令人惊讶，另一些可以说是语言的缺陷。不过，总的来说，下面这些取巧的行为初看起来很奇怪，但只要你了解了背后的成因，这些行为通常都是合乎情理的。

可变的默认参数

Python 编程新手通常会感到惊讶的可能是 Python 对于函数定义中可变默认参数的处理。

当你写下：

```
def append_to(element, to=[]):
    to.append(element)
    return to
```

你期望的可能是：

```
my_list = append_to(12)
print(my_list)
```

7 此处，__exit__() 方法只是调用 I/O 包装器的 close() 方法来关闭文件描述符。在许多系统上，对于打开的文件描述符都有最大数量限制，因此 I/O 完成后立即释放文件描述符是一个很好的做法。

```
my_other_list = append_to(42)
print(my_other_list)
```

你可能认为如果不提供第二个参数，那么函数每次调用时都会创建一个新的列表，因此输出的是：

```
[12]
[42]
```

然而实际上输出的是：

```
[12]
[12, 42]
```

新列表仅会在函数定义时被创建一次，后续每次函数调用都使用同一个列表。在函数被定义时而不是在每次函数调用时（如 Ruby 中那样），就会计算 Python 的默认参数。这意味着如果使用一个可变默认参数并修改它，那么也就修改了后续调用该函数时使用的那个对象。

你应该这样做：

调用函数时，使用一个默认参数值表示没有提供参数（通常会使用 None），每次都创建一个新的对象。

```
def append_to(element, to=None):
    if to is None:
        to=[]
    to.append(element)
    return to
```

这个陷阱何时不再是陷阱？

有时你可以利用这个行为在多次函数调用之间维持状态，这个行为通常用在编写缓存函数时（将结果保存在内存中），例如：

```
def time_consuming_function(x, y, cache={}):
    args = (x, y)
    if args in cache:
        return cache[args]
    # 否则是首次出现这些参数
    # 做一些耗时的操作，然后缓存结果
    cache[args] = result
    return result
```

延迟绑定的闭包

另一个常见困扰来源于 Python 在闭包（或外部全局作用域）中绑定变量的方式。

当你写下：

```
def create_multipliers():
    return [lambda x : i * x for i in range(5)]
```

你期望的可能是：

```
for multiplier in create_multipliers():
    print(multiplier(2), end=" ... ")
print()
```

一个列表包含五个函数，每个函数有自己的封闭变量 i，i 乘以函数的参数得到：

```
0...2...4...6...8...
```

而实际上是：

```
8...8...8...8...8...
```

创建了 5 个函数，不过它们全部都是用 4 乘以 *x*。因为 Python 的闭包是延迟绑定的。这意味着闭包中用到的变量值是在函数被调用时才查找获得的。

不论返回的任一函数何时被调用，i 的值都是调用时在外部作用域中查找获得的。那时，循环已经完成，i 的最终值是 4。

关于这个陷阱，存在一个比较严重的误解：认为它和 Python 的 lambda 表达式有关。由 lambda 表达式创建的函数其实并无特别之处，事实上同样的问题也会出现在普通 def 函数上：

```
def create_multipliers():
    multipliers = []

    for i in range(5):
        def multiplier(x):
            return i*x
        multipliers.append(multiplier)

    return multipliers
```

你应该这样做：

大多数通用解决方案都有点 hack 的味道。由于 Python 存在前述行为，预先计算函

数默认参数值，可以在创建闭包时，使用默认参数值来绑定参数：

```
def create_multipliers():
    return [lambda x, i=i : i * x for i in range(5)]
```

或者使用 functools.partial() 函数：

```
from functools import partial
from operator import mul

def create_multipliers():
    return [partial(mul, i) for i in range(5)]
```

这个陷阱何时不再是陷阱？

当你希望闭包这样表现时，这个陷阱就不再是陷阱了。在大多数情况下，延迟绑定都是一个良好的行为（例如，在 Diamond 项目中，闭包的应用示例（此时陷阱不再是陷阱））。不幸的是，循环创建唯一性函数恰好是延迟绑定会造成小麻烦的一个情况。

组织好项目的结构

使用“结构”一词的意思是你做的每一个决定都关乎项目如何以最佳的形式达成其目标。我们要尽可能利用 Python 的特性来编写高效的代码。具体来说，即代码和文件目录结构中的逻辑和依赖都应该是清晰明了的。

哪个函数应该放到哪个模块？项目中的数据流是怎样的？什么样的特性和函数应该组合在一起或相互分离？广义来说，回答诸如此类的问题就是规划最终产品形态的开始。

Python Cookbook 中的《模块与包》一章详细描述了 __import__ 语句和包的工作方式。本节首先概述 Python 模块和导入系统的机制，这是强化项目结构的核心方式，然后就如何构建可扩展可测试的代码来讨论各种观点。

组织好一个 Python 项目的结构相对比较简单，这归功于 Python 对依赖导入和模块的处理方式：约束限制很少，模块导入的模型也很好理解。因此，你需要做的是一些纯粹的架构任务，即精心规划好项目的各个部分及它们之间的交互关系。

模块

模块是 Python 的主要抽象层之一，也可能是最自然的一层。抽象层让程序员可以将代码划分成多个部分，各自控制相关数据和功能。

例如，如果项目的其中一层负责处理用户行为交互，同时另一层负责处理底层数据操作，那么划分这两层最自然的方式就是将所有接口功能重组到一个文件中，所有底层操作重组到另一个文件中。这个分组将代码逻辑分成两个单独的模块。接口文件可以使用 import module 或者 from module import attribute 语句导入底层操作文件。

只要使用了 import 语句，也就使用了模块，模块可以是内置模块（例如，os 和 sys）、环境中安装的第三方包（例如，Requests 或 NumPy）或者项目的内部模块。如下代码展示了 import 语句的一些例子，同时也证实导入的模块是一个 Python 对象，具有自己的数据类型。

```
>>> import sys # 内置模块
>>> import matplotlib.pyplot as plt # 第三方模块
>>>
>>> import mymodule as mod # 项目内部模块
>>>
>>> print(type(sys), type(plt), type(mod))
<class 'module'> <class 'module'> <class 'module'>
```

遵循 *https://www.python.org/dev/peps/pep-0008/* 上的风格指南，模块名称要简短，使用小写字母且避免使用特殊符号，比如点（.）、问号（?），这些会干扰 Python 模块的查找方式。my.spam.py[8] 这样的文件名也应该避免，因为 Python 会误以为在 my 文件夹中查找 spam.py 文件。Python 官方文档（*http://docs.python.org/tutorial/modules.html#packages*）更加详细地介绍了点号的使用方法，推荐阅读。

导入模块

除一些命名限制外，将 Python 文件用作模块不但没有其他特殊要求，而且有助于理解模块导入机制。首先，import modu 语句会在调用者所在的当前目录下查找 modu.py 文件，如果存在则直接调用它。如果没找到，那么 Python 解释器将在 Python 搜索路径（*https://docs.python.org/2/library/sys.html#sys.path*）中递归查找该文件，若仍然没找到，则抛出 ImportError 异常。搜索路径与平台相关，包含环境变量 $PYTHONPATH（或 Windows 中的 %PYTHONPATH%）中用户或者系统定义的任意文件目录，可以在 Python 会话中检查或修改搜索路径。

```
import sys
>>> sys.path
[ '', '/current/absolute/path', 'etc']
# 实际的路径列表包含导入库时搜索过的所有路径，次序按搜索的顺序
```

8 如果愿意，可以把模块命名为 my_spam.py，不过即使下画线也不应该常用于模块名中（下画线会让人以为这是一个变量名）。

一旦找到 modu.py 文件，Python 解释器将在隔离作用域执行这个模块。modu.py 中的所有顶层语句都会被执行，包括可能存在的其他模块导入。函数和类的定义保存在模块字典中。

模块的变量、函数和类都将通过模块的命名空间提供给调用者。命名空间是编程的一个核心概念，在 Python 中非常有用。命名空间提供一个作用域，作用域之内的命名属性彼此可见，但在命名空间之外不能直接访问它。

在许多编程语言中，预处理器会按照文件包含指令高效地把被包含文件的内容复制到调用者的代码中。但在 Python 中并非如此：被包含的代码会被隔离放在模块命名空间中。import modu 语句将在全局空间里引入一个名为 modu 的模块对象，然后可以通过点号来访问模块内定义的属性，比如，modu.sqrt 即是定义在 modu.py 中的 sqrt 对象。这意味着通常不必担心导入的代码会有什么负面效果，例如，覆盖了已经存在的同名函数。

命名空间工具

函数 dir()、globals() 和 locals() 可以帮助我们快速探查命名空间内部。

- dir(object)：返回通过该对象可以获取到的属性的一个列表。
- globals()：返回全局命名空间中当前存在的属性的一个字典，包含对应的属性值。
- locals()：返回当前局部命名空间（比如，在一个函数内）存在的属性的一个字典，包含对应的属性值。

更多信息请参考 Python 官方文档中关于数据模型的内容（*https://docs.python.org/3/reference/datamodel.html*）。

使用 import 语句的特殊语法 from modu import * 可以模拟更多的标准行为。然而，这通常是不好的做法，使用 import * 的代码更难阅读，依赖也难以区分，也可能因为导入模块中新定义的对象而覆盖已有的同名对象。

使用 from modu import func 只把需要的属性导入全局命名空间中。与 from modu import * 相比，使用 from modu import func 的坏处更少，因为它明确指明了在全局空间中导入了什么。与 import modu 相比，它的优点是使用模块时你可以少打字。表 4-1 展示了从其他模块导入定义的不同方式。

表4-1　对比从其他模块导入定义的不同方式

非常糟糕的方式（会让读者感到困惑）	较好的方式（明确知道在全局命名空间中引入的新名字）	最好的方式（属性来自哪里一目了然）
`from modu import *`	`from modu import sqrt`	`import modu`
`x = sqrt(4)`	`x = sqrt(4)`	`x = modu.sqrt(4)`
sqrt 是 modu 的一部分？是内置函数？还是之前定义的	在导入和调用之间，sqrt 被修改或被重新定义过吗？还是说仍是 modu 中的那个	sqrt 显然是 modu 命名空间的一部分

如代码风格一节所言，良好的代码可读性是 Python 的核心特点之一。代码易读，则无须过多的文本说明，也不会给读者造成不必要的理解困难。明确声明一个类或函数来自哪，像 modu.func() 这样，可以大幅度提升各种项目的代码可读性和易懂性，当然最简单的单文件项目除外。

结构是关键

按照个人意愿组织项目结构时要注意避免一些陷阱。

1. 多重混乱的循环依赖

举个例子，furn.py 文件中类 Table 和类 Chair 需要从 worker.py 文件中导入类 Carpenter 来回答某个问题，如 table.is_done_by()。而类 Carpenter 又需要导入类 Table 和类 Chair，以便回答 carpenter.what_do()。这样就会出现循环依赖，furn.py 依赖于 workers.py，workers.py 又依赖于 furn.py。要解决循环依赖问题，只能求助于一些不那么可靠的技巧，比如，在方法内部使用 import 语句，从而避免 ImportError 异常产生。

2. 暗藏的耦合关系

如果每次变更类 Table 的具体实现都会导致不相关测试用例大量测试失败（因为变更破坏了 Carpenter 的代码，其代码必须非常小心地进行相应的调整），这就意味着 Carpenter 的代码中对类 Table 的实现做了过多不应该的假设。

3. 重度使用全局状态或上下文

类 Table 和类 Carpenter 的具体实现可能会依赖于一些全局变量，而不是显式地相互传参（高度、宽度、类型等），全局变量可能会被多方随意修改，那么为了

弄清楚为什么长方形的桌子变成了正方形的，你得细细检查这些全局变量的全部访问途径，然后发现远程模板代码竟然也在修改这个上下文，弄错了桌子的尺寸。

4. 意面式代码

代码中嵌套的 if 句子和 for 循环长达几个显示页，包含大量复制粘贴的过程式代码没有正常的分段，这类代码即是臭名昭著的意面式代码。不过所幸 Python 中的缩进具有程序语义（这是 Python 最有争议的语言特性），开发者很难维护意面式代码，所以也就不太常见。

5. 馄饨式代码

相比意面式代码，馄饨式代码在 Python 中更容易出现。馄饨式代码包含大量相似的逻辑小片段，通常是类或者对象，没有恰当的组织结构。如果对于手头的任务，你完全不记得应该使用 FurnitureTable、AssetTable、Table，还是 TableNew，那么你可能就身陷馄饨式代码中了。Diamond、Requests 及 Werkzeug 都把有用但不相关的逻辑片段统统收到一个 utils.py 模块或一个 utils 包中，以便在项目中复用，避免产生馄饨式代码。

包

Python 的包管理系统，将模块机制扩展到目录，非常简单易懂。

任何包含 __init__.py 文件的目录都会被视作一个 Python 包。包含 __init__.py 的顶级目录是根包（root package）[9]。导入包里的各个模块和导入普通模块的方式类似，不过 __init__.py 文件用于收集所有包范围（package-wide）的定义。

pack 目录下的 modu.py 文件可以通过 import pack.modu 语句导入。首先，解释器会在 pack 目录下查找 __init__.py 文件并执行其中的所有顶级语句。然后，查找文件 pack/modu.py 并执行其中的所有顶级语句。这些操作执行后，modu.py 中定义的任何变量、函数或者类都能在 pack.modu 命名空间中找到。

__init__.py 文件中的代码过多是一个常见问题。随着项目越来越复杂，目录结构可能会很复杂，也就出现了子包和多层子包。在这种情况下，从多层子包中导入文件需要执行

9 根包结构现在有一个替代方案，称为命名空间包，这归功于 PEP 420（Python 3.3 已实现该特性）。命名空间包不能有 __init__.py 文件，但是它可以分散在 sys.path 环境变量指定的多个目录中。Python 会把所有分散的代码模块收集在一起，作为一个包提供给用户。

目录树，遍历过程中遇到的所有 __init__.py 文件。

如果包内的模块和子包不需要共享任何代码，那么 __init__.py 文件留空是一个常规做法，甚至可以说是一个最佳实践。HowDoI 和 Diamond 项目，在它们的 __init__.py 文件中除版本号外都没写任何代码。Tablib、Requests 和 Flask 项目都在 __init__.py 文件中包含一个顶层文档字符串，以及一些 import 语句，向外暴露一些有意提供的 API，Werkzeug 项目同样也在 __init__.py 文件中向外暴露它的顶级 API，不过 Werkzeug 项目使用了延迟加载（编写了一些额外的代码，按需向命名空间中添加内容，这样可以缩短初始 import 语句的时间耗用）。

导入多层嵌套包有一个便利的语法 import very.deep.module as mod，可以使用 mod 来代替冗长的 very.deep.module 语句。

面向对象编程

Python 有时也被描述为面向对象的编程语言。这可能有几分误导，因此有必要澄清一下。

在 Python 中，一切都能当作对象来处理。这就是人们常说的函数是一等对象的原因。函数、类、字符串甚至类型在 Python 中都是对象：它们都有类型，都能被当成函数的参数传递，同时也可能有自己的方法和属性。从这个角度理解，Python 是一种面向对象的编程语言。

与 Java 不同，Python 并没将面向对象编程作为主要的编程范式。以非面向对象的方式编写一个 Python 项目也是完全可行的，即不使用（或很少使用）类定义、类继承及其他面向对象编程特有的机制。这些特性，Pythonista 可以使用却又不强制使用。此外，Python 处理模块和命名空间的方式为开发者提供了一种自然的方式，来确保各个抽象层的封装和分离（使用面向对象的最大理由），而又无须使用类。

函数式编程（一种编程范式，在其最纯粹的形式中，没有赋值操作符，没有副作用，主要通过串接函数调用来完成任务）的支持者们认为如果一个函数的行为依赖于系统的外部状态，则更容易滋生 bug 和错误。例如，使用一个全局变量来指示一个用户是否登录。Python 虽然不是一种纯粹的函数式编程语言，但也有一些工具可以帮助实现函数式编程（*http://www.oreilly.com/programming/free/functional-programming-python.csp*），这样，我们可以只在想要把状态和功能合在一起的情况下才使用自定义类。

在某些架构中（以 Web 应用为典型），应用会派生出多个 Python 进程来响应外部请求，请求可能会同时发生。在这种情况下，如果实例化对象中保持某些状态，记录当前环境的某些静态信息容易形成竞态条件（从对象状态初始化到通过对象的某个方法来实际使用该对象状态之间的某个时间点，对象的状态已经发生改变），在 Python 中常通过 Class.__init__() 方法来实现。

例如，一个请求是，在内存中载入一个商品，并将其标记为已添加到用户的购物车里。与此同时，另一个请求是将该商品卖给另一个用户，这就可能产生售卖操作实际发生在商品载入内存之后的情况，那么我们就是在尝试售卖已经标记为已出售状态的商品，这类问题导致我们倾向于使用无状态函数。

我们建议：如果正在编写的代码依赖于某些持久化的上下文或全局状态（多数 Web 应用都存在这个情况），那么函数或者过程的隐式上下文和副作用应该越少越好。函数的隐式上下文是指在函数内部能访问到的任何全局变量或持久层数据。副作用是指函数对其隐式上下文做的变更。如果函数在全局变量或持久层中保存或者删除数据，即称该函数存在副作用。

Python 中的自定义类用于从具有逻辑的函数中仔细分离出具有上下文和副作用的函数，这被称作纯函数。纯函数是确定的，即给一个固定输入，输出同样也是固定的。这是因为纯函数不依赖上下文，没有副作用。例如 print() 函数，不是纯函数，因为它的副作用是返回写入数据到标准输出中。下面是使用纯函数和独立函数的好处。

- 当需要重构或者优化时，纯函数更容易修改或者替换。
- 纯函数更容易进行单元测试，很少需要复杂的上下文设置和之后的数据清理工作。
- 纯函数更容易操作、修饰（同时使用多个装饰器）和分发。

总之，对于某些架构而言，相比类和对象，纯函数是更高效的程序构建块，因为它们没有上下文和副作用。例如，在 Tablib 库中，每种文件格式（tablib/formats/.py*）对应的 I/O 函数都是纯函数，不是类的一部分，因为它们做的仅仅是将数据读到一个单独的 Dataset 对象中，或者将 Dataset 数据写到一个文件里。但 Requests 库的 Session 对象是一个类，因为它需要保存 HTTP 会话中交换的 cookie 和认证信息。

面向对象在很多情况下大有用处，甚至是必需的。例如，在开发图形化桌面应用或游戏时，用户可以操作的物件（如窗口、按钮、头像、车辆）在计算机内存中的生命周期相对较长。这也是对象关系映射（将数据库中的行记录映射成代码中的对象）背后的动机之一。

装饰器

从 Python 2.4 版开始，Python 加入了装饰器，PEP 318（*https://www.python.org/dev/peps/pep-0318/*）中有详细的定义和讨论。装饰器是一个函数或者类的方法，用来包装或者修饰另一个函数或者方法，被装饰的函数或者方法将会替代原来的函数或方法。由于 Python 中函数是第一等对象，因此可以手动对函数进行包装，但使用 @decorator 语法更清晰明了，如下示例演示了如何使用装饰器。

```
>>> def foo():
...     print("I am inside foo.")
...
...
...
>>> import logging
>>> logging.basicConfig()
>>>
>>> def logged(func, *args, **kwargs):
...     logger = logging.getLogger()
...     def new_func(*args, **kwargs):
...         logger.debug("calling {} with args {} and kwags {}".format(
...                      func.__name__, args, kwargs))
...         return func(*args, **kwargs)
...     return new_func
...
>>>
>>>
... @logged
... def bar():
...     print("I am inside bar.")
...
>>> logging.getLogger().setLevel(logging.DEBUG)
>>> bar()
DEBUG:root:calling bar with args () and kwargs {}
I am inside bar.
>>> foo()
I am inside foo.
```

这个机制对于隔离函数或者方法的核心逻辑有很大帮助。例如：制表（memoization）或缓存（将耗时函数的结果存入一个查找表中，后续需要时直接查找使用，而不是重新计算）使用装饰器来实现效果更佳。这显然不属于函数逻辑的一部分。如 PEP 3129（*https://www.python.org/dev/peps/pep-3129/*）所述，从 Python 3 开始，装饰器可以直接应用于类。

动态类型

Python 是动态类型语言。相对于静态类型语言，动态类型语言意味着变量没有一个固定的类型。变量被实现为指向对象的指针，因此可以先让变量指向值 42，然后指向字符串“thanks for all the fish”，再指向一个函数。

Python 使用的动态类型常常被认为是一个缺点，因为它增大了代码复杂性和调试难度。如果一个名为 a 的变量可以指向很多不同的事物，那么开发者或者维护者必须在代码中跟踪这个变量名称，以确保它不会被指向一个毫不相关的对象。表 4-2 展示了使用变量名称时一些好的例子和坏的例子。

表4-2　避免为不同的事物使用相同的变量名

建议	坏的例子	好的例子

使用短小函数或方法，以降低不相关事物同名的风险	`a = 1`
	`a = 'answer is {}'.format(a)`
		`def get_answer(a):`
		`    return 'answer is {}'.format(a)`
		`a = get_answer(1)`

相关事物，类型不同，也应使用不同的名称	`# 一个字符串`
	`items = 'a b c d'`
	`# 不要这样写，这是一个列表`
	`items = items.split(' ')`
	`# 不要这样写，这是一个集合`
	`items = set(items)`
		`items_string = 'a b c d'`
		`items_list = items.split(' ')`
		`items = set(items_list)`

复用变量名称对代码效率提升并无帮助：赋值时同样会创建一个新的对象。随着代码越来越复杂，在同一变量名多次赋值之间被其他代码行（包括分支和循环）分隔开，将导致确定给定变量的类型更加困难。

在一些编码实践中，比如函数式编程，建议不要对同一变量进行多次赋值。在 Java 中，使用 final 关键字可以强制变量赋值后保持值不变。Python 没有 final 关键字，而且这有悖于 Python 的设计哲学。但是一个变量只赋值一次通常是一个良好的编程原则，有助于强化可变类型和不可变类型的概念。

如果对一个变量，重复赋值为不同的类型，Pylint（*https://www.pylint.org/*）会输出警告提示你。

可变类型和不可变类型

Python 内置的或用户自定义的[10]类型分为两类。

```
# 列表可变
my_list = [1, 2, 3]
my_list[0] = 4
print my_list # [4, 2, 3] <- 同一个列表，内容改变了
```

10　如何使用 C 语言定义自己的 Python 类型，可参考 Python 扩展文档（*https://docs.python.org/3/extending/newtypes.html*）。

```
# 整数不可变
x = 6
x = x + 1 # 这个新的 x，占据了一个不同的内存位置
```

1. 可变类型

允许就地（in-place）修改对象的内容。例如，列表和字典，提供了 list.append() 或 dict.pop() 内容修改方法。

2. 不可变类型

不提供修改自身内容的方法。例如，变量 x 赋值为整数 6，没有自增的方法，计算 x+1 必须创建另一个整数并为其命名。

行为上的差异导致的一个结果就是可变类型不能用作字典的键，因为值一旦改变，就不能哈希到相同的值，字典在键存储时使用了哈希[11]。与列表对等的不可变类型是元组，使用圆括号来创建，例如 (1,2)。元组的内容无法修改，因此可以作为字典的键。

对希望修改的对象使用恰当的可变类型，比如 my_list=[1, 2, 3]。对希望有固定值的对象使用不可变类型，比如 islington_phone=(“220”,“7946”,“0347”)，以此向其他开发者明确地表明代码的意图。

Python 的一个特性可能会让新手感到惊奇：字符串是不可变类型，尝试修改会产生一个类型错误。

```
>>> s = "I'm not mutable"
>>> s[1:7] = "am"
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

这意味着若想从多个部分构造出一个字符串，先将各个部分积聚到一个列表中，然后将列表元素拼接成一个完整的字符串效率更高，因为列表是可变的。此外，相比循环调用 append() 来构造列表，Python 的列表解析（在输入上迭代创建列表的速记语法）更好且更快。表 4-3 展示了从一个可迭代对象创建列表的不同方式。

11 一个简单的哈希算法的例子：首先将一个数据项的字节序列转换成一个整数，然后对某个数求余。这是 memcached 跨多个机器分布其数据键的方式。

表4-3　从一个可迭代对象创建列表的不同方式

糟糕的方式	较好的方式	最好的方式
`>>> s =""`	`>>> s = []`	`>>> r = (97, 98, 99)`
`>>> for c in (97,98,98):`	`>>> for c in (97,98,99):`	`>>> s = [unichr(c) for c in r]`
`...  s += unichr(c)`	`...  s.append(unichr(c))`	`>>> print("".join(s))`
`...`	`...`	`abc`
`>>> print(s)`	`>>> print("".join(s))`	
`abc`	`abc`	

关于此类优化，可以阅读 Python 官网上的一个讨论页面（*https://www.python.org/doc/essays/list2str/*）。

如果一次字符串拼接中涉及的元素个数是已知的，相比先创建一个列表然后使用“”.join() 来创建字符串，直接的字符串加法更快。如下所示创建 cheese 变量的格式化方案所做的都是一样的[12]。

```
>>> adj = "Red"
>>> noun = "Leicester"
>>>
>>> cheese = "%s %s" % (adj, noun) # 这种格式化风格已废弃（见 PEP 3101）
>>> cheese = "{} {}".format(adj, noun) # 自 Python 3.1 开始应该都可以使用
>>> cheese = "{0} {1}".format(adj, noun) # 其中的数字可以复用
>>> cheese = "{adj} {noun}".format(adj=adj, noun=noun) # 这种风格最佳
>>> print(cheese)
Red Leicester
```

管理依赖

某些 Python 程序包会在源码中包含其外部依赖（第三方库），通常是存放在名为 vendor 或 packages 的目录中。关于这个话题，*http://bit.ly/on-vendorizing* 上一篇非常优秀的博文列举了 Python 包开发者可能会这样做的主要理由（基本上都是为了避免各种依赖问题），并讨论了其他的一些可选方案。

几乎在所有情况下，最好将依赖分离开都是共识，否则会添加不必要的内容到代码库中。使用 setup.py（更为推荐，特别是如果你的包是一个库时）或者 requirements.txt（如果使用 requirements.txt，在与 setup.py 发生依赖冲突时，requirements.txt 会覆盖 setup.py 中指定的依赖）结合虚拟环境，可以将依赖限制为已知的可正常工作的版本。

12 我们应该承认：根据 PEP 3101，虽然这种百分号风格的格式化方式（%s、%d、%f）至今废弃十年有余，但是多数老的 Python 开发者仍在使用，并且 PEP 460 还引入这种方式来格式化 bytes 或 bytearray 对象。

如果这些措施还不够，那么可以联系依赖库的开发者，通过更新他们的包来解决问题（例如，你的库可能依赖于依赖包即将发布的版本，或者可能需要添加一个特定的新特性），这些更新很可能让整个社区都受益。

注意，如果你提交的合并请求中含有大量的代码变更，那么在未来新的建议和请求出现时，可能需要你来维护这些更新。出于此原因，Tablib 和 Requests 都尽可能少地包含了某些依赖库的源码（译注：Requests 在最新版中移除了依赖库的源码）。随着社区逐步全面采用 Python 3，希望这些最紧迫的问题更少些。

测试代码

测试代码非常重要。能如期工作的项目只有先确定下来，大家才更有可能使用。

2001 年发行的 Python 2.1 首次包含 doctest 和 unittest，支持测试驱动开发（TDD）。在测试驱动开发流程中，开发人员首先编写测试用例，定义函数的主要操作和边界情况，然后编写函数来通过这些测试用例。自 Python 包含这两个库后，TDD 已被广泛接受并用于商业和开源的 Python 项目中，同时编写测试代码和实际运行的代码是个不错的主意。若能正确使用，TDD 可以帮助你精确定义你的代码意图，架构的模块化也会更好。

测试要点

测试代码大概是开发者能写的最大规模的有用代码。我们在此总结一些要点。

每个测试只做一件事情。一个测试用例应该聚焦于一个小的功能点，并证明它的正确性。

必须独立且无依赖。每个测试用例必须完全独立：能单独执行，也能在测试集中执行，而不用考虑调用的顺序。这个规则暗指每个测试用例都必须加载一个新的数据集，并在完成之后做一些可能的清理工作。这些工作通常由 setUp() 和 tearDown() 方法来完成。

精确好过简约。为测试函数使用长的描述性名称。这条准则与实际运行的代码稍有不同，实际运行的代码更倾向使用简短的名字。测试函数绝不会被显式地调用。在实际运行的代码中，square() 甚至命名成 sqr() 都行，但在测试代码中，应该使用 test_square_of_number_2() 或者 test_square_negative_number() 这样的名称。由于测试失败时会显示这些函数名称，因此函数名称应该尽可能多地提供描述性信息。

速度很重要。尽量让测试快速运行。如果一个测试用例的运行时间超过毫秒级，则会明显拖慢开发效率，或者使得测试用例不能如期运行。在某些情况下，测试用例需要处理复杂的数据结构，并且每次运行时都要加载，因此运行速度不会很快。把这些繁重的测试用例放到另外的测试集中，通过一些定时任务来运行，然后按需执行其他测试用例。

要阅读手册啊，朋友！首先了解你的工具，学习怎样运行单个测试或测试用例。然后在模块中开发一个功能，经常运行这个功能的测试用例，最好是当你保存代码时自动执行测试。

在开发开始时测试一切，并在开发结束时再次测试一切。在每次编码之前和之后，都要运行全部测试集。这样才能确信没有破坏其余代码中的任何东西。

版本控制自动化钩子是极好的。实现一个钩子，在代码推送到共享代码库前运行所有测试是一个好主意。可以直接在版本控制系统中添加钩子，一些 IDE 也在其环境中提供了一些可以更简便地做到这一点的方式。下面是当前流行的版本控制系统的文档链接，这些文档将带你深入了解如何去做。

- GitHub（*https://developer.github.com/webhooks/*）。
- Mercurial（*http://bit.ly/mercurial-handling-repo*）。
- Subversion（*http://bit.ly/svn-repo-hook*）。

编码过程中断时编写一个破坏性测试。当处于开发中间阶段但不得不中断工作时，对将要开发的功能编写一个破坏性单元测试是个好主意。这样当你回来工作时，可以更快地回到原先离开的地方继续工作。

面对歧义，使用测试来调试。调试代码的第一步是编写一个新的测试来定位 bug。虽然并不总是能这样做，但测试捕捉到的 bug 通常都位于项目最有价值的代码片段中。

如果测试很难解释，那么是否能找到合作者就只能祝你好运了。在某些功能出错了或者不得不做出改变时，如果代码有一套良好的测试用例，那么你或者其他维护者将主要依赖测试集来修复问题或者修改给定的行为。因此，测试代码被阅读的次数通常会多过实际运行的代码，甚至更多。在这种情况下，目的不明确的单元测试就没多大用处了。

如果测试代码很容易解释，那么测试代码几乎总是一个好想法。测试代码的另一个用处是作为新进开发人员的入门介绍。当有人需要在现有的代码库上工作时，运行和阅读相关的测试代码通常是一个最好的做法。他们会发现热点代码（这也是大多数理解难点出现的地方）以及边界情况。如果他们需要添加一些功能，那么第一步应该是添加一个测试用例，以此确保新功能在未实现接口之前不会是一个起作用的路径。

更重要的是，别慌张。这是开源的，全世界都会鼎力相助。

测试的基础知识

本节会列举一些测试的基础知识，并给出一些例子，这些实例选自第 5 章中的 Python 项目。关于 Python 测试驱动开发，市面上已经有一整本书来讲解了，这里不再重复了，请

阅读 O'Reilly 出版的《Python Web 开发：测试驱动方法》。

unittest

unittest 是 Python 标准库自带的测试模块，使用过 Junit(Java)/nUnit(.Net)/CppUnit(C/C++) 系列工具的人都熟悉其 API。

通过继承 unittest.TestCase 来创建测试用例。MyTest 中测试函数仅是定义为一个普通的新方法，示例代码如下所示。

```
# test_example.py
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test_that_fun_adds_one(self):
        self.assertEqual(fun(3), 4)

class MySecondTest(unittest.TestCase):
    def test_that_fun_fails_when_not_adding_number(self):
        self.assertRaises(TypeError, fun, "multiply six by nine")
```

测试方法的名称必须以字符串 test 开始，否则不会被执行。测试代码模块（文件）名称默认需要满足 test*.py 模式，但也可以从命令行中提供 --pattern 关键字参数来指定任意模式。

打开一个终端，进入测试文件所在目录，在命令行中调用 Python 的 unittest 模块，执行 TestClass 的所有测试。

```
$ python -m unittest test_example.MyTest
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

或者，如果要执行一个文件内的所有测试，那么需要指定文件名称：

```
$ python -m unittest test_example
.
----------------------------------------------------------------------
```

```
Ran 2 tests in 0.000s

OK
```

mock（在 unittest 模块中）

自 Python 3.3 开始，标准库提供 unittest.mock 模块，允许你在测试中使用模拟的对象替换系统的某些部分，并对它们的使用方式做出断言。

例如，可以打一个猴子补丁（猴子补丁是在运行时修改或替换其他已有代码的代码），代码如下所示。在如下代码中，我们创建了一个名为 instance 的实例，将其已有的方法 ProductionClass.method 替换为一个新的对象 MagicMock，调用时始终返回 3，并计算它接收到的方法调用次数，记录它被调用的签名，还包含一些用于测试的断言方法。

```
from unittest.mock import MagicMock

instance = ProductionClass()
instance.method = MagicMock(return_value=3)
instance.method(3, 4, 5, key='value')

instance.method.assert_called_with(3, 4, 5, key='value')
```

若要在测试代码模块中模拟类或者对象，可以使用 patch 装饰器。在如下示例中，使用一个始终返回相同结果的模拟对象替代一个外部的搜索系统（正如示例中那样，这个补丁只在测试时使用）。

```
import unittest.mock as mock

def mock_search(self):
    class MockSearchQuerySet(SearchQuerySet):
        def __iter__(self):
            return iter(["foo", "bar", "baz"])
    return MockSearchQuerySet()

# 此处 SearchForm 指的是已导入的类引用 myapp.SearchForm
# 修改的是这个实例，而不是原本定义的 SearchForm 自身类的代码
@mock.patch('myapp.SearchForm.search', mock_search)
def test_new_watchlist_activities(self):
    # get_search_results runs a search and iterates over the result
    self.assertEqual(len(myapp.get_search_results(q="fish")), 3)
```

mock 模块还有许多其他方式可以用来对其进行配置，控制其行为。unittest.mock 的 Python 官方文档（*http://docs.python.org/3/library/unittest.mock.html*）中有详细介绍。

doctest

doctest 模块先在文档字符串中搜索类似于交互式 Python 会话的文本段，然后执行这些会话来验证它们是否如所示的那样正常工作。

与恰当的单元测试目的不同，doctest 测试代码通常不太详细，不能捕获一些特殊情况的 bug 或者隐蔽的回归测试 bug。相反，它们作为模块及其组件主要用法的描述性文档非常有用。不过，每次执行完整的测试集时，doctest 测试也应该自动执行。

一个简单的函数 doctest 测试。

```
def square(x):
    """Squares x.

    >>> square(2)
    4
    >>> square(-2)
    4
    """
    return x * x

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

在命令行中运行某个模块时，比如 python module.py，doctest 测试将会运行，如果结果和文档字符串中描述的不一致则会报错。

举例说明

本节将从 Python 程序包中摘录部分代码，以实际代码来突显最佳的测试实践。测试集要求的额外依赖库不是包含在程序包中（例如，Requests 使用 Flask 来模拟一个 HTTP 服务器），而是包含在项目的 requirements.txt 文件中。

所有这些例子，第一步都是打开一个终端，切换到开源项目的工作目录，复制源码库，并设置一个虚拟环境，如下所示。

```
$ git clone https://github.com/username/projectname.git
$ cd projectname
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ pip install -r requirements.txt
```

例子：在 Tablib 中的测试

Tablib 使用 Python 标准库中的 unittest 模块进行测试。其软件包没有包含测试集，需要复制 GitHub 上的代码库来获取相关文件。如下是摘录的代码，重要部分在后面做了注解。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Tests for Tablib."""

import json
import unittest
import sys
import os
import tablib
from tablib.compat import markup, unicode, is_py3
from tablib.core import Row


class TablibTestCase(unittest.TestCase):    ❶
    """Tablib test cases."""

    def setUp(self):                            ❷
        """Create simple data set with headers."""

        global data, book

        data = tablib.Dataset()
        book = tablib.Databook()

        #
        # ... skip additional setup not used here ...
        #

    def tearDown(self):                         ❸
        """Teardown."""
        pass

    def test_empty_append(self):                ❹
        """Verify append() correctly adds tuple with no headers."""
        new_row = (1, 2, 3)
        data.append(new_row)

        # Verify width/data
        self.assertTrue(data.width == len(new_row))
        self.assertTrue(data[0] == new_row)
```

```
    def test_empty_append_with_headers(self):    ❺
        """Verify append() correctly detects mismatch of number of
        headers and data.
        """
        data.headers = ['first', 'second']
        new_row = (1, 2, 3, 4)

        self.assertRaises(tablib.InvalidDimensions, data.append, new_row)
```

❶ 使用 unittest 需要继承 unittest.TestCase，并编写名称以 test 为开头的测试方法。TestCase 提供了一些断言方法来检测相等性、真值、数据类型、是否是集合成员，以及是否引发异常，更多详情请参考 *http://bit.ly/unittest-testcase*。

❷ TestCase.setUp() 会在 TestCase 中每个测试方法之前执行。

❸ TestCase.tearDown() 会在 TestCase 中每次测试方法之后执行。[13]

❹ 所有测试方法的名称都必须以 test 为开头，否则不会被执行。

❺ 单个 TestCase 可以有多个测试方法，但每个方法都应该只做一件事情。

如果想为 Tablib 贡献代码，那么复制代码后首先应该做的是运行测试集，确保没有错误出现，如下所示。

```
(venv)$ ### 在项目的顶级目录 tablib/ 中
(venv)$ python -m unittest test_tablib.py
..........................................................
----------------------------------------------------------------------
Ran 62 tests in 0.289s

OK
```

自 Python 2.7 开始，unittest 包含了自己的测试发现机制，在命令行使用 discover 选项。

```
(venv)$ ### 在项目顶级目录 tablib/ 之外
(venv)$ python -m unittest discover tablib/
..........................................................
----------------------------------------------------------------------
Ran 62 tests in 0.234s

OK
```

13 注意：如果因为代码发生错误而退出，那么 unittest.TestCase.tearDown 不会被执行。如果你用过 unittest.mock 模块中的特性来修改代码的实际行为，那么你对此可能会感到惊讶。Python 3.1 中添加了方法 unittest.TestCase.addCleanup()，它会将一个清理函数及其参数放入一个调用栈中，在 unittest.TestCase.tearDown() 或其他方法调用之后，调用栈中的函数会被逐个调用执行，无论 tearDown() 是否会被调用。更多详情请阅读 unittest.TestCase.addCleanup() 的相关文档（*http://docs.python.org/3/library/unittest.html#unittest.TestCase.addCleanup*）。

在确认所有测试用例运行通过后，你应该找到与你修改部分相关的测试用例，在修改代码时需要经常执行它，也可以针对正在增加的特性或者追踪的 bug 编写一个新的测试用例，然后在修改代码时经常执行它，例如：

```
(venv)$ ### 在项目顶级目录 tablib/ 中
(venv)$ python -m unittest test_tablib.TablibTestCase.test_empty_append
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

一旦代码完成，在推送代码到代码库之前最好执行全部的测试集。因为你要经常运行这些测试，所以它们的运行速度应该尽可能快。更多关于 unittest 的使用细节，请参考 *http://bit.ly/unittest-library* 上的标准库 unittest 文档。

例子：在 Requests 中的测试

Requests 使用 py.test 进行测试。实战操作方法是，打开一个终端，切换到一个临时目录，复制 Requests 库，安装其依赖，并执行 py.test，如下所示。

```
$ git clone -q https://github.com/kennethreitz/requests.git
$
$ virtualenv venv -q -p python3 # -q 选项开启静默模式
$ source venv/bin/activate
(venv)$
(venv)$ pip install -q -r requests/requirements.txt # 再次开启静默模式
(venv)$ cd requests
(venv)$ py.test
========================== test session starts ==============================
===
platform darwin -- Python 3.4.3, pytest-2.8.1, py-1.4.30, pluggy-0.3.1
rootdir: /tmp/requests, inifile:
plugins: cov-2.1.0, httpbin-0.0.7
collected 219 items

tests/test_requests.py ....................................................
...
X.......................................
tests/test_utils.py ..s.....................................................

========= 217 passed, 1 skipped, 1 xpassed in 25.75 seconds =================
```

其他流行工具

下面列举的测试工具虽然使用者相对较少，但是依然很流行，值得一提。

pytest

pytest 是 Python 标准 unittest 模块的替代方案，但无须编写一些样板代码，这意味着它不需要测试类的脚手架，甚至可能不需要设置和数据清理的方法。使用 pip 命令安装它：

```
$ pip install pytest
```

虽然 pytest 的功能完备，并且可以拓展，但它的语法很简单。创建一个测试集和编写一个带多个函数的模块一样容易。

```
# test_sample.py 文件内容
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

相对于 unittest 模块的同等功能，运行 py.test 命令需要做的工作少得多。

```
$ py.test
========================== test session starts ============================
platform darwin --Python 2.7.1 -- pytest-2.2.1
collecting ... collected 1 items

test_sample.py F

================================ FAILURES =================================
_______________________________ test_answer _______________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        + where 4 = func(3)

test_sample.py:5: AssertionError
======================== 1 failed in 0.02 seconds =========================
```

Nose

Nose 扩展了 unittest，使得测试更加容易：

```
$ pip install nose
```

Nose 提供了测试自动化发现机制，减少了手动创建测试集的烦恼。它也提供了大量的特性插件。例如，xUnit 插件可以兼容测试输出、代码覆盖率报告和测试选集。

tox

tox 是一个针对多解释器配置环境的自动化测试环境管理和测试的工具。

```
$ pip install tox
```

tox 允许通过简单的 ini 风格配置文件来配置复杂的多参数测量模型。

老版本 Python 的测试方案

如果你不能决定使用哪个版本的 Python，但仍想使用这些测试工具，下面是一些替代方案。

unittest2 是 Python2.7 unittest 模块的一个向后移植版本，相比以前老版本 Python 中的单元测试模块，其提供了改进过的 API 和更好的断言。

如果使用 Pyhton 2.6 或者更低的版本（这意味着你可能是在大型银行或者财富 500 强的公司工作），则可以通过 pip 安装它：

```
pip install unittest2
```

如果想要以 unittest 名称导入这个模块，使得将来把代码移植到模块新版本时更加容易，那么可以执行以下操作。

```
import unittest2 as unittest

class MyTest(unittest.TestCase):
    ...
```

使用这种方式，如果以后要转到新版 Python 并且不再需要 unittest2 模块，那么可以简单地在测试代码模块中修改导入语句，而不需要改变任何其他代码。

如果喜欢 mock（在 unittest 模块中），但使用的 Python 版本低于 3.3，那么仍然可以导入一个独立的库来使用 unittest.mock。

```
$ pip install mock
```

fixture 提供工具来简化测试中数据库后端的设置和清理。它可以在使用 SQLAlchemy、SQLObject、Google Datastore、Django ORM 或 Storm 时导入模拟数据集。目前仍然有新版本发布，但只在 Python 2.4 到 Python 2.6 的版本上测试过。

Lettuce 和 Behave

Lettuce 和 Behave 是 Python 中实现行为驱动开发（BDD）的包。BDD 是 21 世纪早期从 TDD 中衍生出来的一种敏捷开发方法，它希望以行为来替代测试驱动开发中的测试，以便帮助新手解决开头难的问题，掌握 TDD。Dan North 在 2003 年首次使用了 BDD 这个名称，于 2006 年在 *Better Software* 杂志的一篇题为 *Introducing BDD* 的文章正式向世界介绍了这一技术，文中同时还介绍了配套的 Java 工具 JBehave。Dan North 的博客（*https://dannorth.net/introducing-bdd/*）转载了这篇文章。

2011 年 *The Cucumber Book (Pragmatic Bookshelf)* 一书出版之后（译注：中文版已出，书名为《Cucumber：行为驱动开发指南》），BDD 迅速火爆。这本书主要介绍 Ruby 的一个 Behave。受此鼓舞的 Python 社区成员 Gabriel Falco 开发了 Lettuce，Peter Parente 开发了 Behave。

我们使用名为 Gherkin 的语法以普通文本来描述行为，该语法人类可读而且机器也能处理。下面这些教程可能会对掌握 Gherkin 语法有用。

- Gherkin tutorial（*https://github.com/cucumber/cucumber/wiki/Gherkin*）。
- Lettuce tutorial（*http://lettuce.it/tutorial/simple.html*）。
- Behave tutorial（*http://tott-meetup.readthedocs.org/en/latest/sessions/behave.html*）。

文档

可读性是 Python 开发者的主要关注点，包括项目和代码文档。本节描述的最佳实践可以为开发者节约很多时间。

项目文档

API 文档适用于项目用户，对于想为项目做贡献的人来说，他们需要的是附加的项目文档。本节就是关于如何编写附加项目文档的。

项目根目录下的 README 文件应该为项目的用户和维护者提供一些基本信息，应该是纯文本格式，或者以某种极易阅读的标记语言来编写，比如 reStructured Text（推荐，因为它是目前 PyPI 唯一可以理解的格式[14]）或者 Markdown，应该包含几行解释项目或者库的目的（不能假设用户对项目有所了解）、软件主要安装源的 URL 和一些基本的致谢信息。该文件是代码阅读者的主要入口。

INSTALL 文件对于 Python 来说不太必要（不过可能有助于遵守 GPL 这类许可证的要求）。

14 若感兴趣，可以阅读 *https://bitbucket.org/account/signin/?next=/pypa/pypi/issues/148/support-markdown-for-readmes* 上关于为 PyPI 上的 README 文件添加 Markdown 支持的一些讨论。

安装步骤通常会简化为一个命名，如 pip install module 或者 python setup.py install，添加到 README 文件中。

应该始终提供 LICENSE 文件，并且指定软件是基于什么许可证开放使用的。

TODO 文件或 README 中的 TODO 小节应该列出代码的开发计划。

CHANGELOG 文件或 README 中的 CHANGELOG 小节应该简要概述项目最近一些版本中代码库的变更点。

项目配套发行文档

文档可能包含如下的部分或者全部内容，实际情况由项目决定。

- 项目入门（introduction）应该使用一到两个极简的用例，简要概述产品能够用来做什么。这就是项目的 30 秒自我陈述。
- 项目教程（tutorial）应该更详细地介绍一些主要用例。读者能够跟着一步一步地搭建一个可以工作的原型。
- API 参考，通常从代码中生成，列出所有公共可用的接口、参数和返回值。
- 开发者文档是为潜在的贡献者准备的，可以包含项目的代码约定和通用设计策略。

Sphinx

Sphinx 无疑是最流行 [15] 的 Python 文档工具。它能将 reStructured Text 标记语言文本转成多种输出格式，包括 HTML、LaTeX（可进一步转换成可打印的 PDF 版本）、帮助手册和普通文本。

Sphinx 文档还可以免费托管在 Read the Docs 平台（*http://readthedocs.org/*）上。它是一个非常好的平台上，可以为你的源码库配置提交钩子，自动构建文档。

Sphinx 因项目 API 文档生成特性而著名，但同样也可以用于生成一般的项目文档。本书的在线版本即是使用 Sphinx 构建的，并托管在 Read the Docs 上。

reStructured Text

Sphinx 使用的是 reStructured Text 标记语法，几乎所有的 Python 文档都是用它编写的。如果 setuptools.setup() 的 long_description 参数内容以 reStructured Text 标记语法编写，

15 你可能见过其他工具，例如 Pycco、Ronn、Epydoc（现在已停止开发和维护）及 MkDocs。几乎所有人都使用 Sphinx，推荐你也用。

那么它可以在 PyPI 上渲染成 HTML，如果是其他格式则以文本形式展示。它就像是内建所有可选扩展的 Markdown。如下是一些语法学习资源。

- reStructuredText Primer（*http://sphinx.pocoo.org/rest.html*）。
- reStructuredText 快速参考（*http://docutils.sourceforge.net/docs/user/rst/quickref.html*）。

读者也可以立即开始为喜爱的 Python 包文档做贡献，边读边学。

文档字符串与块注释

文档字符串和块注释不可互换。两者都能用于函数或类，如下示例使用了两者。

```
# This function slows down program execution for some reason.      ❶
def square_and_rooter(x):
    """Return the square root of self times self."""     ❷
    ...
```

❶ 开头的注释块是程序员的一个说明。

❷ 文档字符串描述的是这个函数或者类的操作，在交互式 Python 会话中，当用户输入 help(square_and_rooter) 时将展示文档字符串。

放在模块开头或 __init__.py 文件顶部的文档字符串也会出现在 help() 中。 Sphinx 的 autodoc 特性也可以使用恰当地格式化好的文档字符串来自动生成文档。Sphinx 教程（*http://www.sphinx-doc.org/en/stable/tutorial.html#autodoc*）中介绍了如何自动生成文档，以及如何为 autodoc 格式化文档字符串。进一步了解文档字符串，请阅读 PEP 257（*https://www.python.org/dev/peps/pep-0257/*）。

日志

自 Python 2.3 版本开始，logging 模块已是 Python 标准库的一部分。PEP 282（*https://www.python.org/dev/peps/pep-0282/*）对其做了简要描述。除基础 logging 教程（*https://docs.python.org/3/howto/logging.html#logging-basic-tutorial*）以外，其文档都晦涩难读。

日志服务于两个目的。

1. 诊断日志

   诊断日志是记录与应用程序操作相关的事件。例如，用户来电报告一个错误，则可以搜索日志找到上下文。

2. 审计日志

审计日志是为商业分析记录事件。从审计日志可以提取用户的事务（比如，一个点击事件流）并结合用户的其他详情（比如，最终购买行为）来生成报告或者优化商业目标。

Logging 与 print

只有在一个命令行应用展示帮助信息时，与 logging 相比，print 才是一个更好的选择。在其他情况下，使用 logging 比 print 更好的原因如下所示。

- 每个日志事件创建的日志记录都会包含一些易用的诊断信息，比如，文件名、完整路径、函数，以及日志记录事件的代码行号。
- 项目包含的模块记录的事件都能通过根日志记录器输出到应用的日志流，除非你将它们过滤掉。
- 可以通过 logging.Logger.setLevel() 方法有选择性地记录日志，或者设置属性 logging.Logger.disabled 为 True 来禁用日志。

在库中使用 logging

配置日志模块的说明在 *http://bit.ly/configuring-logging* 的日志指南中。另一个日志使用的好资源在第 6 章的标准库中。在一个日志事件发生时，如何处理是由用户决定的，而不是库，所以下面的忠告值得一再强调。

强烈建议除 NullHandler 以外，不要向你的库的日志记录器中加入任何其他处理器。

NullHandler 正如其名：它什么也不做。否则用户如果不需要记录日志，就必须特意关闭日志记录才行。

在库中实例化日志记录器时，最佳实践是使用全局变量 __name__，logging 模块使用点号（dot）创建层级日志记录器，因此，使用 __name__ 可以确保不会出现名字冲突。

下面这个最佳实践的例子取自 Requests 源码（*https://github.com/kennethreitz/requests*），在项目的顶级 __init__.py 文件中也可以加入这段代码。

```
# Set default logging handler to avoid "No handler found" warnings.
import logging

try: #Python 2.7+
```

```
    from logging import NullHandler
except ImportError:
    class NullHandler(logging.Handler):
        def emit(self, record):
            pass

logging.getLogger(__name__).addHandler(NullHandler())
```

在应用中使用 logging

Twelve-Factor App（*http://12factor.net/*）是应用开发最佳实践的一份权威参考，包含一节关于 logging 最佳实践的内容（*http://12factor.net/logs*），它将日志事件视为一个事件流并将事件流发送到标准输出中，由应用环境来处理。

配置日志记录器至少有三种方式。

	优点	缺点
使用 INI 格式文件	如果使用函数 logging.config.listen() 在一个套接字上监听变化，则可以在运行时更新配置	相对于在代码中配置日志记录器，你可以做的控制更少（例如，自定义子类化过滤器或记录器）
使用字典或者 JSON 格式文件	除了能在运行时更新，从 Python 2.6 开始，还能通过标准库中的 json 模块从文件中加载配置	相对于在代码中配置日志记录器，可以做的控制更少
使用源码	可以完全控制配置	任何配置的变更都需要修改源码

通过 INI 文件进行配置的例子

关于 INI 文件格式的更多细节，可以阅读 logging 教程中的 logging 配置（*https://docs.python.org/howto/logging.html#configuring-logging*）一节。一个简化的配置文件如下所示。

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
```

```
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

asctime、name、levelname 和 message 都是 logging 库的可选属性。选项的完整列表及定义。假设日志配置文件名 logging_config.ini，在代码中使用 logging.config.fileConfig 方法加载配置来设置日志记录器。

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

通过字典进行配置的例子

从 Python 2.7 开始，可以使用字典进行详细配置。PEP 391（*https://www.python.org/dev/peps/pep-0391*）罗列了配置字典中的必选项和可选项，下例是一个简单的实现。

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
        },
   handlers = {
       'h': {'class': 'logging.StreamHandler',
             'formatter': 'f',
             'level': logging.DEBUG}
       },
       loggers = {
            'root': {'handlers': ['h'],
                     'level': logging.DEBUG}
            }
```

```
)

dictConfig(logging_config)
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

直接在源码中进行配置的例子

最后举例说明一下直接在代码中配置 logging，实现方式如下所示。

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
        '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

选择许可证

在美国，如果源码作品没有指定许可证，那么用户无权下载、修改或分发，用户也不能为该项目做贡献，除非你告诉他们应该遵从何种规则。因此，项目需要一个许可证。

上游许可证

如果项目衍生自其他项目，那么可以选择的许可证由上游许可证决定。例如，Python 软件基金会（PSF）要求所有 Python 源码贡献者都签订一个贡献者协议，基于两种许可证中的一种[16] 正式把他们的代码授权给 PSF（保留他们自己的版权）。

因为这两种许可证都允许用户在不同的条款下再次授权许可，所以 PSF 就可以根据自己的许可证——Python 软件基金会许可证——自由分发 Python 源码。PSF 许可证常见问题页面（*https://wiki.python.org/moin/PythonSoftwareFoundationLicenseFaq*）以通俗易懂的（非法律的）语言详细说明了用户可以做什么、不可以做什么。这个许可证除授权 PSF 发行 Python 以外别无他用。

16 在写作此段内容时，这两种许可证是 Academic Free License 2.1 和 Apache 2.0。具体如何运作的完整描述请参考 *https://www.python.org/psf/contrib/*。

许可证选项

有许多许可证可供选择。PSF 建议从开放源码促进会（OSI）认可的许可证中选择一个使用。如果你希望最终将代码贡献给 PSF，那么选择贡献页面（*https://www.python.org/psf/contrib/*）上指定的许可证将会让这个过程变得更容易一些。

注意：记得修改许可证模板中的占位符文本，确保文本真实反映你的信息。例如，MIT 许可证模板在其第二行包含 Copyright (c) <year> <copyright holders>。Apache 许可证 2.0 版则不需要任何修改。

开源许可证可以分为两类[17]。

1. 宽松许可证

宽松许可证，又叫 BSD 许可证，更关注用户使用软件的自由，如下所示。

- Apache 许可证：目前是 Apache 2.0 版本，经过修改以便人们可以在任何项目里不加修改地包含它，可以在项目中引用许可证名称而不必在每个文件里都列出许可证内容，并且在 GPLv3 许可证下也可以使用 Apache 2.0 许可的代码。
- BSD 2- 条款许可证和 BSD 3- 条款许可证：BSD 3- 条款许可证在 BSD 2- 条款许可证的基础上，额外限制了对发行者商标的使用。
- MIT 许可证：Expat 许可证和 X11 许可证都是使用相应流行产品命名的 MIT 许可证。
- 互联网软件系统联盟（ISC）许可证：除了几行现在被认为不相关的内容，它几乎等同于 MIT 许可证。

2．Copyleft 许可证

Copyleft 许可证，或称为较不宽松许可证，它更关注代码本身，确保源码及对其做出的变更，对外都能使用。其中最知名的是 GPL 家族，最新版本是 GPLv3。

GPLv2 许可证与 Apache 2.0 许可证不兼容，因此使用 GPLv2 许可证的代码不能与使用 Apache 2.0 许可证的代码混合。Apache 2.0 许可证授权的项目可以用于 GPLv3 授权的项目，但之后必须全部是 GPLv3。

17 此处描述的所有许可证都是 OSI 认可的，可参考 OSI 许可证主页（*https://opensource.org/licenses*）进一步了解。

满足 OSI 标准的许可证在不同限制和要求下都允许商业使用、软件修改及下游分发。表 4-4 中罗列的所有许可证都限制了发行者的责任义务，要求用户在任何下游分发中保留原始版权和许可证。

表4-4　流行许可证相关的议题

许可证家族	限制	准许	要求
BSD	保护发行者的商标（BSD 3- 条款许可证）	允许担保（BSD 2- 条款许可证和 3- 条款许可证）	—
MIT（X11 或 Expat）、ISC	保护发行者的商标（ISC 和 MIT/X11 许可证）	允许以不同的许可证再次授权	—
Apache 2.0	保护发行者的商标	允许再次授权，允许用于专利	必须声明对原始代码做出的变更
GPL	禁止再次许可使用不同的许可证	允许担保，以及用于专利（仅 GPLv3）	必须声明对原始代码做出变更并包含源码

软件许可相关的学习资源

Van Lindberg 的《知识产权与开源》一书是开源软件法律问题方面的优秀学习资源，不仅有助于你了解许可证，还可以了解与开源相关的其他知识产权话题的法律问题，例如，商标、专利和版权。如果你不关心法律问题，只想快速地做出某些选择，那么下面这些网站可能对你有所帮助。

- GitHub 提供了一个方便的指南（*http://choosealicense.com/*），用简单的几句话对各种许可证进行概述和对比。
- TLDRLegal18（*http://tldrlegal.com/*）快速地罗列了每个许可证条款下能够做的、不能做的及必须做的事情。
- OSI 认可的许可证列表（*http://opensource.org/licenses*）包含了所有通过许可证审查过程的许可证全文，这些许可证均符合开源定义（允许自由使用、修改和共享软件）。

第 5 章

阅读高质量的代码

程序员通常会阅读大量的代码。Python 背后的核心设计理念之一就是代码可读性，而成为一名优秀程序员的秘诀之一就是阅读、理解并领会高质量的代码。这些高质量代码通常都会遵循代码风格一节概括的风格指南，并尽力清晰地向读者表达代码的意图。

本章会从一些非常易读的 Python 项目中摘录部分代码进行展示和讲解，这些项目用实例充分阐明了第 4 章涵盖的主题。在介绍这些项目的同时，我们也会分享一些代码阅读的技巧[1]。

本章重点介绍的项目清单，按照出场次序排列如下。

- HowDoI（*https://github.com/gleitz/howdoi*）：控制台应用程序，搜索因特网以回答编程问题，使用 Python 语言编写。
- Diamond（*https://github.com/python-diamond/Diamond*）：Python 后台程序[2]，负责收集系统指标度量数据（metrics）并发送到 Graphite 或其他后端。它能够收集 CPU、内存、网络、I/O、负载及磁盘等指标的度量数据。此外，它还特别提供了一个 API 来实现自定义收集器，几乎可以从任意来源收集指标度量数据。
- Tablib（*https://github.com/kennethreitz/tablib*）：一个无关格式的表格数据集库。
- Requests（*https://github.com/kennethreitz/requests*）：超文本传输协议库（90% 的人只是想要这样一个 HTTP 客户端：能够自动处理密码认证，并遵循多个协议标准（*https://www.w3.org/Protocols/*）执行一些工作，例如，以一个函数调用执行分块文件上传）。
- Werkzeug（*https://github.com/mitsuhiko/werkzeug*）：起初只是简单地为 Web 服务

1 推荐阅读 Serge Demeyer、Stéphane Ducasse 和 Oscar Nierstrasz 合著的《面向对象的软件再工程模式》一书，该书包含了作者数十年的代码阅读和重构经验。

2 后台程序是指作为后台守护进程运行的计算机程序。

网关接口（WSGI）应用提供各种工具，现在已经演变成最强大的 WSGI 工具模块之一。

- Flask（*https://github.com/mitsuhiko/flask*）：基于 Werkzeug 和 Jinja2 实现的一个 Web 微框架，适用于快速搭建简单的 Web 页面服务。

相比本章提到的内容，所有这些项目值得一说的东西还有很多很多，我们真心希望通过阅读本章，能激发你下载并深入阅读其中至少一到两个项目的热情（甚至可以在用户组里介绍你在这个过程中学到的东西）。

共同特征

某些特征是所有这些项目共同具备的：对每个项目的一个快照进行详细分析，结果显示平均每个函数的代码行数非常少（少于 20，不包括空白行和注视），空白行很多。更大更复杂的项目使用的文档字符串或注释也会更多；通常代码库中超过五分之一的内容是某类文档。但从 HowDoI 可以看到，如果代码足够直白易懂，注释则不再必需（HowDoI 没有文档字符串，因为它不是交互式使用的）。表 5-1 展示了这些项目共同的一些实践。

表5-1　示例项目的共同特征

Python 包	许可证	行数	档字符串（行数占比）	注释行数占比	空白行（行数占比）	平均函数长度
HowDoI	MIT 许可证	262	0%	6%	20%	13 行代码
Diamond	MIT 许可证	6021	21%	9%	16%	11 行代码
Tablib	MIT 许可证	1802	19%	4%	27%	8 行代码
Requests	Apache 2.0 许可证	4072	23%	8%	19%	10 行代码
Flask	BSD 3- 条款许可证	10163	7%	12%	11%	13 行代码
Werkzeug	BSD 3- 条款许可证	25822	25%	3%	13%	9 行代码

在接下来的每节中，首先，我们使用一种不同的代码阅读技巧来理解项目是关于什么的。然后，挑选一些代码片段来论证本书其他某处提到的观点（如果我们在介绍某个项目时没有强调某些东西并不意味着这些东西在这个项目中不存在；我们只是想各有侧重地基于这些示例项目来论证概念）。本章将通过示例让你理解是什么造就了好代码，也让你学习一些理念。读完本章，对于阅读代码，你会更加自信。

HowDoI

Benjamin Gleitzman 的 HowDoI 项目，不超过 300 行的代码，是我们开始代码阅读冒险

之旅的绝佳选择。

阅读单文件脚本

脚本通常有一个明确的运行起始点、配置选项和运行结束点，因此相比提供 API 或者提供框架的库，更容易阅读理解。

从 GitHub 上获取 HowDoI 模块[3]：

```
$ git clone https://github.com/gleitz/howdoi.git
$ virtualenv -p python3 venv # 或使用 mkvirtualenv，随你选择
$ source venv/bin/activate
(venv)$ cd howdoi/
(venv)$ pip install --editable .
(venv)$ python test_howdoi.py # 运行单元测试
```

现在应该已经在 venv/bin 目录下安装了 howdoi 可执行文件。在命令行中执行 cat 'which howdoi' 就能看到该文件的内容了。这个文件在你执行 pip install 时会自动生成。

阅读 HowDoI 文档

HowDoI 的文档在 HowDoI GitHub 仓库中的 README.rst 文件里。它是一个小型的命令行应用，允许用户在因特网上搜索编程问题的答案。

在终端 shell 中输入 howdoi --help 可以查看使用帮助。

```
(venv)$ howdoi --help
 usage: howdoi [-h] [-p POS] [-a] [-l] [-c] [-n NUM_ANSWERS] [-C] [-v]
               [QUERY [QUERY ...]]
  instant coding answers via the command line
positional arguments:
  QUERY                 the question to answer
optional arguments:
  -h, --help  show this help message and exit
  -p POS, --pos POS  select answer in specified position (default: 1)
  -a, --all     display the full text of the answer
  -l, --link      display only the answer link
  -c, --color           enable colorized output
  -n NUM_ANSWERS, --num-answers NUM_ANSWERS
                        number of answers to return
  -C, --clear-cache clear the cache
  -v, --version         displays the current version of howdoi
```

3 如果使用 lxml 时遇到麻烦，要求更新版本的 libxml2 共享库，则只需执行 pip uninstall lxml;pip install lxml == 3.5.0 命令安装更早版本的 lxml 即可。

总结一下，从文档中我们了解到 HowDoI 从因特网上为编程问题获取答案，从用法说明我们了解到用户可以选择特定位置的答案，程序可以对输出进行着色，程序可以获取多个答案并缓存起来，之后可以清除。

使用 HowDoI

可以通过实际使用来确认我们对 HowDoI 的理解，示例如下：

```
(venv)$ howdoi --num-answers 2 python lambda function list comprehension
--- Answer 1 ---
[(lambda x: x*x)(x) for x in range(10)]

--- Answer 2 ---
[x() for x in [lambda m=m: m for m in [1,2,3]]]
 # [1, 2, 3]
```

我们已经安装了 HowDoI，阅读了其文档，也会使用它了，那么开始阅读实际的代码吧！

阅读 HowDoI 代码

howdoi/ 目录里包含两个文件：一个是 __init__.py 文件，它只包含一行定义版本号的代码；还有一个是 howdoi.py 文件，我们打开阅读。

浏览 howdoi.py 文件，我们看到每个新定义的函数都会在下一个函数中用到，这样非常易于跟踪阅读。每个函数都只做一件事情，而且是一件如函数名称所言的事情。主函数 command_line_number() 定义在 howdoi.py 文件末尾。

这里我们不翻印 HowDoI 的源码，而是以调用关系图来阐明程序的调用结构，如图 5-1 所示。这个调用关系图使用 Python 函数调用关系图模块（*https://pycallgraph.readthedocs.io/*）生成，该工具可以对 Python 脚本运行时的函数调用进行可视化展示。因为命令行应用程序一般都是单个运行起始点，代码中的分支路径相对较少，所以非常适合可视化分析（注意，为了清晰明了地展示，我们从渲染好的图片中手动删除了不属于 HowDoI 项目的函数，并稍微调整了着色和格式）。

这些代码本来可能就是一个庞大费解的意面式函数。不过，我们可以有目的地组织代码结构，将代码划分成若干个名字直观的函数。简要描述一下图 5-1 描绘的执行流程：首先，command_line_runner() 解析输入并将用户标记和查询传递给 howdoi()；然后，howdoi() 将 _get_instructions() 包装在一个 try/except 语句中，以便捕捉链接错误并打印一条合理的错误信息（因为应用代码不应该在异常上终止执行）。

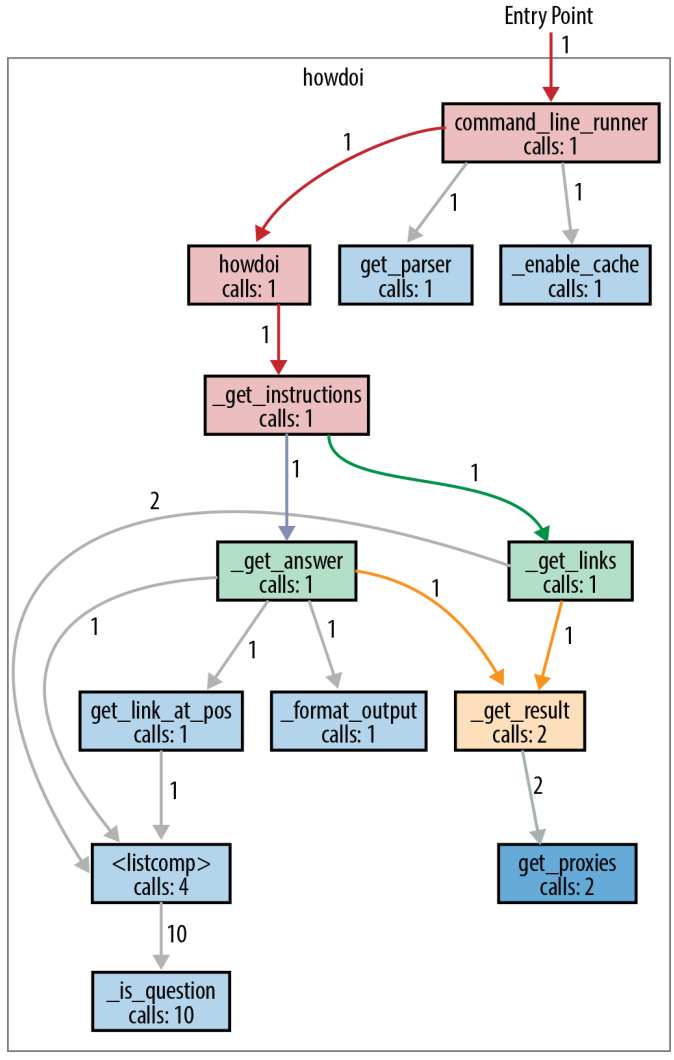


图5-1　HowDoI调用关系图

主要功能都在 _get_instructions() 函数中：首先，调用 _get_links() 使用 Google 搜索 Stack Overflow 上命中查询的链接；然后，对每个结果链接调用一次 _get_answer()（这取决于用户在命令行中指定的答案数目，默认是仅一个答案）。

_get_answer() 函数首先向 Stack Overflow 访问一个链接，从答案中提取代码进行着色，然后返回给 _get_instructions() 函数。该函数将所有的答案合成一个字符串然后返回。_get_links() 和 _get_answer() 都通过调用 _get_result() 来实际执行 HTTP 的请求，_get_links() 进行 Google 查询，_get_answer() 处理 Google 查询返回的结果链接。

_get_results() 所做的就是在 try/except 语句中封装 requests.get() 以便可以捕获 SSL 错误，打印错误信息并重新抛出异常，这样，顶层 try/except 可以捕获这个异常然后退出程序。对应用程序来说，在退出之前捕获所有的异常是最佳实践。

HowDoI 的打包

HowDoI 项目的 setup.py 文件在 howdoi/ 目录上，是一个不错的安装设置模块（setup module）示例，除了常规的包安装，它还会安装一个可执行文件（打包命令行工具时可以参考）。setuptools.setup() 函数使用关键字参数来定义所有的配置项。标识可执行文件的部分与关键字参数 entry_points 相关。

```
setup(
    name='howdoi',
    ## 省略了其他典型的配置项
    entry_points={
        'console_scripts': [                    ❶
            'howdoi = howdoi.howdoi:command_line_runner',    ❷
        ]
    },
## 省略了依赖列表
)
```

❶ 列出控制台脚本的关键字参数 console_scripts。

❷ 此处声明名为 howdoi 的可执行文件实际是调用目标函数 howdoi.howdoi.command_line_runner()。因此，在之后阅读代码时，我们就知道 command_line_runner() 是运行整个程序的入口点。

取自 HowDoI 的结构示例

因为 HowDoI 是一个非常小的库，所以本章侧重讲述工程结构，只对 HowDoI 做几点评注。

一个函数只做一件事情

我们已经强调多次，把 HowDoI 的逻辑拆分成多个内部函数，每个函数只完成一件事情，这对读者来说大有帮助。此外，还有一些函数唯一的目的就是使用 try/except 语句来包装其他函数。唯一一个例外是 format_output() 函数，它借助 try/except 子句为语法高亮识别正确的编码语言，而不是异常处理。

利用系统提供的数据

HowDoI 会检查并使用相关的系统值（比如 urllib.request.getproxes()）来处理代理服务器的使用（这种情况可能出现在像学校这样的机构中，这些机构通常会使用一台中间服务器来过滤互联网链接），或者用于如下代码片段：

```
XDG_CACHE_DIR = os.environ.get(
    'XDG_CACHE_HOME',
    os.path.join(os.path.expanduser('~'), '.cache')
)
```

如何知道这些变量是否存在呢？从 requests.get() 的可选参数中可以明显地看出对 urllib.request.getproxies() 的需求，因此，这些变量的一部分信息是来自对调用库 API 的理解。环境变量通常针对特定应用，如果一个库打算使用一个特定的数据库或者其他姊妹程序，那么其文档会列出相关的环境变量。对于普通的 POSIX 系统，若想了解其环境变量，一个不错的开始是 Ubuntu 默认环境列表（*https://help.ubuntu.com/community/EnvironmentVariables*），或者是 POSIX 规范中的环境变量基本列表（*http://bit.ly/posix-env-variables*），它会链接到很多相关的其他列表。

取自 HowDoI 的风格示例

HowDoI 主要遵循 PEP 8 的代码风格，但并非教条式遵循，在 PEP 8 限制代码可读性时也不一定会遵循。例如，HowDoI 虽然把 import 语句写在代码文件顶部，但是混合地引入标准库和外部模块。由于 HowDoI 没法自然地对字符串断行，因此 USER_AGENTS 中的字符串常量虽然超过了 80 个字符，但是也保持不变。

接下来这些代码摘录将重点说明我们在第 4 章推崇的其他风格选择。

带下画线前缀的函数名（我们都是负责任的用户）

HowDoI 中几乎每个函数的名称都带下画线前缀，这样将函数标识为仅供内部使用。因为对于其中大部分函数来说，调用它们可能会遇到未捕获的异常，任何调用 _get_result() 的函数都存在这个风险，howdoi() 函数能处理这些可能的异常。

其余内部函数（_format_output()、_is_question()、_enable_cache() 及 _clear_cache()）如此标识只是为了不让它们在包外使用。测试脚本 howdoi/tset_howdoi.py 仅调用了没有下画线前缀的函数，通过给顶层函数 howdoi.howdoi() 提供用于着色的命令行参数来检查格式化功能是否正常，而不是直接将代码提供给 howdoi._format_output()。

仅在一处地方处理兼容性问题（可读性很重要）

在主代码之前处理好可能存在的不同依赖版本的差异，这样读者就知道主代码不会存在依赖问题，并且版本检查的代码也不会到处都是。因为HowDoI是作为命令行工具交付的，在代码中把兼容性问题处理好，就能避免强迫用户改变他们的 Python 环境来适应这个工具，给用户提供了便利。兼容性问题变通方案的代码片段如下所示。

```
try:
    from urllib.parse import quote as url_quote
except ImportError:
    from urllib import quote as url_quote

try:
    from urllib import getproxies
except ImportError:
    from urllib.request import getproxies
```

通过创建函数 u(X) 来模拟 Python 3 或仅用 7 行代码就解决了 Python 2 和 Python 3 处理 Unicode 的差异。此外，为了遵循 Stack Overflow 的新增引用准则，还在代码中加上了引用的原始地址。

```
# Handle Unicode between Python 2 and 3
# http://stackoverflow.com/a/6633040/305414
if sys.version < '3':
    import codecs
    def u(x):
        return codecs.unicode_escape_decode(x)[0]
else:
    def u(x):
        return x
```

符合 Python 风格的写法（优美胜于丑陋）

下面的代码片段取自 howdoi.py，它的写法符合 Python 风格。函数 get_link_at_pos() 在没有结果时返回 False，或者从 links 参数中识别出指向 Stack Overflow 问题的链接，然后返回一个期望位置上的链接（如果没有足够链接，则返回最后一个）。

```
def _is_question(link):                                    ❶
    return re.search('questions/\d+/', link)

# 省略了一个函数

def get_link_at_pos(links, position):
    links = [link for link in links if _is_question(link)]  ❷
```

```
    if not links:
        return False                ❸

    if len(links) >= position:
        link = links[position-1]    ❹
    else:
        link = links[-1]                    ❺
    return link                             ❻
```

❶ 函数 _is_question() 单独一行，为一个原本比较晦涩的正则表达式搜索指定了清晰的语义。

❷ 列表解析读起来就像一个句子，这归功于单独定义的 _is_question() 和富有意义的变量名称。

❸ 返回的 return 语句减少了代码的逻辑嵌套，使代码扁平化。

❹ 这里额外一步给变量 link 赋值。

❺ 和上面一样给变量 link 赋值，而不是各自使用不带命名变量的 return 语句，以明确的变量名强调了函数 get_link_at_pos() 的目的。这样代码是自描述（self-documenting）的。

❻ 此处单个 return 语句，代码缩进仅一级，明确显示：代码的所有执行路径或者因为没找到相应的链接而立即退出，或者在函数末尾返回一个链接。我们的简单经验法则派上用场了：通过阅读这个函数的首行和末行就能理解它是干什么的（给定多个链接和一个位置，get_link_pos() 返回给定位置上的一个链接）。

Diamond

Diamond 是一个后台程序（作为后台守护进程一直运行的应用程序），收集系统的指标度量数据，并发布到下游程序，例如，MySQL、Graphite（*http://graphite.readthedocs.org/*）（Orbitz 是在 2008 年开源的一个平台，可以存储、检索并（可选）进行数值时序数据绘图）等。我们将探讨什么是优秀的包结构，因为 Diamond 是一个多文件应用程序，比 HowDoI 大得多。

阅读一个更大的应用程序

和 HowDoI 一样，Diamond 是一个命令行程序，也有着明确的运行起始点和清晰的执行路径，尽管实现的代码分散到了多个文件。

从 GitHub 上获取 Diamond 代码（文档说它只支持 CentOS 和 Ubuntu，但是从 setup.py 的代码看它似乎支持所有平台。不过，默认收集器用来监控内存、磁盘空间和其他系统指标的一些命令不支持 Windows），截至本文撰稿时，它依然使用 Python 2.7。

```
$ git clone https://github.com/python-diamond/Diamond.git
$ virtualenv -p python2 venv # 目前还不兼容 Python 3
$ source venv/bin/activate
(venv)$ cd Diamond/
(venv)$ pip install --editable .
(venv)$ pip install mock docker-py # 这是用于测试的依赖
(venv)$ pip install mock # 这也是一个用于测试的依赖
(venv)$ python test.py # 运行测试
```

和 HowDoI 库一样，Diamond 的安装设置脚本会在 venv/bin 下安装 diamond 和 diamond-setup 两个可执行文件。它们不是自动生成的，而是项目的 Diamond/bin 目录下预先写好的脚本。文档指出 diamond 工具用于启动服务器，diamond-setup 则是一个可选工具，可以让用户通过交互的方式修改配置文件中关于收集器的设定。

项目代码中有很多附加目录，diamond 包源码在 Diamond/src 目录下。我们将阅读 Diamond/src（包含核心代码）、Diamond/bin（包含可执行文件 diamond），以及 Diamond/conf（包含配置文件示例）目录下的文件。

阅读 Diamond 文档

首先，通过浏览 Diamond 的在线文档（*http://diamond.readthedocs.io/*）大致了解这个项目是什么，以及能做什么。Diamond 的目标是让收集集群机器的系统指标度量数据变得更容易。最初由 BrightCove 公司于 2011 年开源此项目，现在贡献者超过 200 个。

Diamond 的安装方法是：修改配置文件示例（即项目源码中的 conf/diamond.conf.example 文件），把它放到默认路径 /etc/diamond/diamond.conf 下，或者命令行中指定的一个路径下，这样就完成配置设定了。Diamond 官方 wiki 页面（*https://github.com/BrightcoveOS/Diamond/wiki/Configuration*）上有一个关于配置的章节，可以帮到你。

在命令行中执行 diamond --help 可以获得用法说明。

```
(venv)$ diamond --help
    Usage: diamond [options]
    Options:
      -h, --help            show this help message and exit
      -c CONFIGFILE, --configfile=CONFIGFILE
                            config file
      -f, --foreground      run in foreground
```

```
-l, --log-stdout      log to stdout
-p PIDFILE, --pidfile=PIDFILE
                      pid file
-r COLLECTOR, --run=COLLECTOR
                      run a given collector once and exit
-v, --version         display the version and exit
--skip-pidfile        Skip creating PID file
-u USER, --user=USER  Change to specified unprivileged user
-g GROUP, --group=GROUP
                      Change to specified unprivileged group
--skip-change-user    Skip changing to an unprivileged user
--skip-fork           Skip forking (damonizing) process
```

从用法说明可以看到：Diamond 会用到一个配置文件，默认在后台运行，会记录日志，可以为它指定一个 PID（进程 ID）文件，可以测试收集器，可以更改进程的用户和组，默认会派生进程从而成为守护进程[4]。

使用 Diamond

为了更好地理解 Diamond，我们可以尝试运行它。创建一个目录 Diamond/tmp，在该目录下放一个经过修改的配置文件。在 Diamond 目录下，执行如下代码。

```
(venv)$ mkdir tmp
(venv)$ cp conf/diamond.conf.example tmp/diamond.conf
```

编辑 tmp/diamond.conf。

```
### Options for the server
[server]
# Handlers for published metrics.                    ❶
handlers = diamond.handler.archive.ArchiveHandler
user =                                               ❷
group =
# Directory to load collector modules from           ❸
collectors_path = src/collectors/

### Options for handlers                             ❹
[handlers]
[[default]]
```

4 把一个前台进程转变为后台守护进程，需要先从进程中派生出子进程，从子进程中分离原有会话 ID，然后子进程再派生子进程，这样进程就与正在运行它的终端断开连接（非守护进程在终端关闭时会退出。在列出所有当前正在运行的进程之前，你可能会看到警告消息“确定要关闭此终端吗？关闭它会杀死以下进程……”）。即使在终端窗口关闭后，守护进程还会运行。守护进程（daemon）这个名称源自 Maxwell 的 daemon（一个聪明而不邪恶的妖）。

```
[[ArchiveHandler]]
log_file = /dev/stdout

### Options for collectors
[collectors]
[[default]]
# Default Poll Interval (seconds)
interval = 20

### Default enabled collectors
[[CPUCollector]]
enabled = True

[[MemoryCollector]]
enabled = True
```

从示例配置文件中可知：

❶ 存在多种处理器，可以通过类名进行选择。

❷ 可以控制守护进程以什么用户和组来运行（为空表示使用当前的用户和组）。

❸ 可以指定一个路径来查找收集器模块。这样 Diamond 就知道在哪可以找到自定义的 Collector 子类：直接在配置文件中声明了查找路径。

❹ 也可以分开配置处理器。

设置日志输出到 /dev/stdout（使用默认格式配置），设置应用程序在前台运行，设置跳过 PID 文件写入，并使用我们的新配置文件来运行 Diamond。

```
(venv)$ diamond -l -f --skip-pidfile --configfile=tmp/diamond.conf
```

若要终止进程，可按组合键 Ctrl+C，直到命令提示符重新出现。日志输出显示了收集器和处理器做的事情：收集器收集不同的指标度量数据（例如，MemoryCollector 收集内存总量、当前可用量、空闲量及交换内存大小），处理器格式化这些度量数据，并发送到多种目的地，例如 Graphite、MySQL，在测试用例中作为日志消息输出到 /dev/stdout。

阅读 Diamond 代码

在阅读大型项目时，IDE 可以在源代码中快速定位类和函数的原始定义；给出一个定义，IDE 可以找出项目中所有用到它的地方。请将 IDE 的 Python 解释器设置为虚拟环境中的解释器，以便使用这个功能。[5]

5 在 PyCharm 中，从菜单栏中找到 PyCharm -> 首选项 -> 项目 :Diamond -> 项目解释器，选择当前虚拟环境的 Python 解释器路径。

不像在 HowDoI 中那样追踪每个函数，图 5-2 只追踪 import 语句，只显示了 Diamond 中哪些模块导入了其他模块。这样的草图有助于对大型项目形成一个宏观理解，因为隐藏了细节才能看到整体。从左上的 diamond 可执行文件开始，追踪 Diamond 项目中的模块导入。除 diamond 可执行文件外，每个矩形框都表示 src/diamond 目录下的一个文件（模块）或者一个目录（包）。

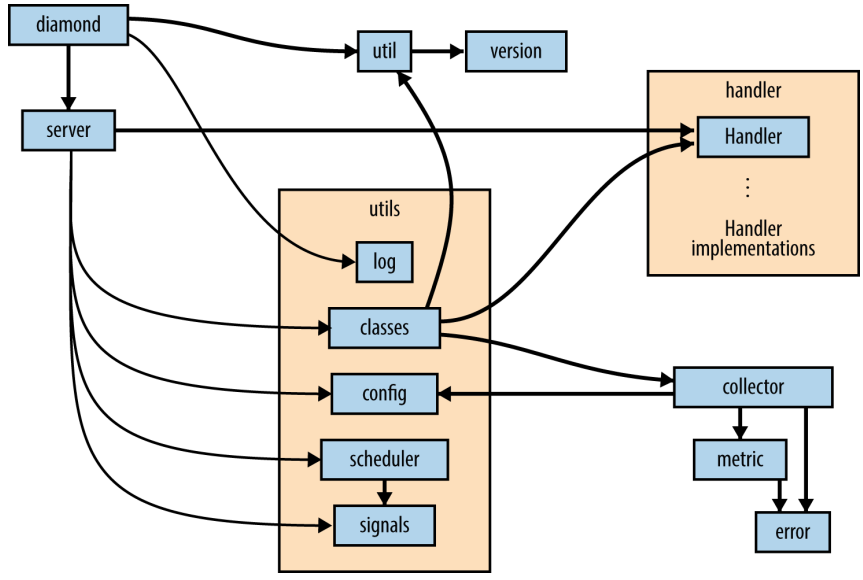


图5-2　Diamond的模块导入结构图

Diamond 项目结构组织合理，模块命名恰当，因此仅从图 5-2 中就能大致了解代码逻辑：diamond 从 util 模块中获取版本信息，然后使用 utils.log 模块来设置日志记录，并使用 server 模块来启动一个 Server 类实例。Server 从 utils 包中导入几乎所有的模块，使用 utils.classes 模块来访问 handler 模块和 collectors 目录中的处理器，使用 utils.config 模块来读取配置文件并获取收集器的设置（以及用户自定义收集器的额外路径），使用 utils.scheduler 模块和 utils.signals 模块来设置收集器的轮询间隔以计算它们的指标度量值，并设置和启动处理器来处理指标度量数据队列，将处理结果发送到各个目的地。

图 5-2 中没有包含特定收集器中使用的帮助模块 convertor.py 和 gmetric.py，也没有包含 handler 子包中定义的 20 多个处理器的实现，以及项目的 Diamond/src/collectors/ 目录（如果按照上文所示的方式进行安装，则应该是安装在其他地方，例如使用 PyPI 或者 Linux 包分发而不是源码方式进行安装）下定义的 100 多个收集器的实现。所有这些模块都是通过 diamond.classes.load_dynamic_class() 导入的，diamond.classes.load_dynamic_

class() 则是根据配置文件中指定的字符串名，调用 diamond.util.load_class_from_name() 函数来载入这些类，所以 import 语句无须显式地指明它们。

若要理解为什么同时有 utils 包和 util 模块，则必须深入阅读实际的代码。util 模块提供的函数更多的是与 Diamond 打包相关，而不是与其操作相关。一个函数从 version.__VERSION__ 获取版本号，另两个函数用来解析字符串，将其识别为模块或者类并导入。

Diamond 中的日志记录

当启动守护进程时，diamond 可执行文件中的 main() 函数会调用 src/diamond/utils/log.py 中的 diamond.utils.log.setup_logging() 函数，如下所示。

```
# Initialize logging
log = setup_logging(options.configfile, options.log_stdout)
```

如果 options.log_stdout 为 True，那么 setup_logging() 会以默认格式器设置一个日志记录器，将 DEBUG 级别及以上的日志记录到标准输出中。具体配置实现摘录如下：

```
##  省略其他代码

def setup_logging(configfile, stdout=False):
    log = logging.getLogger('diamond')

    if stdout:
        log.setLevel(logging.DEBUG)
        streamHandler = logging.StreamHandler(sys.stdout)
        streamHandler.setFormatter(DebugFormatter())
        streamHandler.setLevel(logging.DEBUG)
        log.addHandler(streamHandler)
    else:
        ##  省略此处代码
```

否则，它使用 Python 标准库中的 logging.config.file.fileconfg() 函数来解析配置文件。下面是该函数的调用，代码行多层缩进，因为它位于上面代码段的 if/else 语句中，还以一个 try/except 块包装着。

```
            logging.config.fileConfig(configfile,
                                      disable_existing_loggers=False)
```

日志配置会忽略配置文件中与日志记录无关的关键字。这也就是为什么 Diamond 可以使用同一个配置文件来为自己和日志配置提供配置参数。示例配置文件 Diamond/conf/diamond.conf.example 从其他 Diamond 处理器中把日志处理器标识了出来。

```
### Options for handlers
[handlers]

# daemon logging handler(s)
keys = rotated_file
```

在配置文件后面的“日志记录配置项”标题下定义了示例记录器，详细信息推荐阅读关于日志配置文件的文档（*http://bit.ly/config-file-format*）。

取自 Diamond 的结构示例

Diamond 不仅是可执行程序，也是一个库，为用户提供一种方式来创建和使用自定义收集器。

首先着重讲述关于整体包结构的东西，然后深入探究 Diamond 作为一个应用究竟如何导入并使用外部定义的收集器。

将不同的功能划分到不同的命名空间（命名空间是个绝妙的主意）

图 5-2 展示了项目中 server 模块与其他三个模块的交互：diamond.handler、diamond.collector 和 diamond.utils。utils 子包原本也可以将其所有的类和函数都包含在一个单独的、大型的 util.py 模块中，但恰好也适合使用命名空间将代码切分成相关的分组，开发小组也这样做了。

所有的处理器实现都包含在 diamond/handler 目录下，但收集器的组织结构却与此不同。没有目录，只有一个 diamond/collector.py 模块，其中定义了 Collector 和 ProcessCollector 基类。而所有的收集器实现都定义在 Diamond/src/collectors/ 目录下，如果从 PyPI（推荐）而不是从 GitHub（如上文为了阅读源码是这样做的）安装，那么这些收集器实现会被安装在虚拟环境的 venv/share/diamond/collectors 目录下。这样便于用户创建新的收集器实现：将所有收集器放置在同一位置，应用更容易找到它们，库的用户也可以以这些收集器实现为参考来自定义收集器。

Diamond/src/collectors 目录中的每个收集器实现都在自己的目录中（而不是在单个文件中），这样可以把每个收集器实现的测试分开。

用户可拓展自定义类（复杂胜于难懂）

添加新的收集器实现非常简单：只要继承 diamond.collector.Collector 抽象基类[6]，实现 Collector.collect() 方法，然后将新的收集器实现放在 venv/src/collectors 下自己的目录中。

应用加载调用自定义收集器的底层实现是复杂的，但是用户看不到，也无须关心。本节内容同时展现了 Diamond 收集器 API 面向用户部分的简单性，以及支撑这个用户接口的底层代码实现的复杂性。

复杂与难懂。我们可以把复杂代码关联的用户体验比作体验一只瑞士手表——就是好用。瑞士手表内部有很多制作精良的小零件，它们之间配合很好，从而创造了轻松愉悦的用户体验。但是，使用难懂的代码就像驾驶一架飞机，为避免飞机撞毁燃烧[7]，必须彻底理解你在做的每一件事情。我们不想生活在一个没有飞机的世界里，但也希望不必成为制表大师就能让手表正常工作。无论如何，不太复杂难懂的用户界面是一件好事。

简单的用户接口。为了创建一个自定义数据收集器，用户必须先继承抽象类 Collector，再通过配置文件提供新收集器的路径。如下这个新收集器定义的例子摘自 Diamond/src/collectors/cpu/cpu.py。当 Python 查找 collect() 方法时，会先在 CPUCollector 中查找方法定义，如果没找到，则会使用 diamond.collector.Collector.collect()，抛出 NotImplemented Error 异常。

最简化的收集器代码如下所示。

```
# coding=utf-8
import diamond.collector
import psutil

class CPUCollector(diamond.collector.Collector):

    def collect(self):
        # 在 Collector 中，该方法只包含一行 raise(NotImplementedError)
        metric_name = "cpu.percent"
        metric_value = psutil.cpu_percent()
        self.publish(metric_name, metric_value)
```

收集器定义的默认存放位置位于 venv/share/diamond/collectors/ 目录中，但可以在配置文件中通过 collectors_path 参数值指定任意存放位置。类名 CPUCollector 已经在示例配置

6 Python 中的抽象基类是指存在某些方法未定义的类，期望开发人员在子类中定义这些方法。调用抽象基类中的这些未定义方法会产生 NotImplementedError 异常。一个替代方式是使用 Python 的抽象基类模块 abc（在 Python 2.6 中首次实现），这种方式会在构造一个尚不完整的类时报错，而不是在尝试访问类的未事先方法时报错。完整的规约定义见 PEP 3119（*https://www.python.org/dev/peps/pep-3119/*）。

7 这一句改述自斯坦福大学名誉教授 Larry Cuban 针对某话题发表的博文《复杂和难懂之间的区别至关重要》（*http://bit.ly/complicated-vs-complex*）。

文件中列出。如下示例所示，除了在全局默认值（配置文件中对应位置文本的下面）或单个收集器的覆盖配置项中添加一个 hostname 或 hostname_method 配置参数，无须进行任何其他修改。

```
[CPUCollector]]
enabled = True
hostname_method = smart
```

更复杂的内部代码实现。在内部，Server 会使用 collectors_path 中指定的路径来调用 utils.load_collectors()，如下是该函数的大部分代码。

```
def load_collectors(paths=None, filter=None):
    """Scan for collectors to load from path"""
    # Initialize return value
    collectors = {}
    log = logging.getLogger('diamond')

    if paths is None:
        return

    if isinstance(paths, basestring):      ❶
        paths = paths.split(',')
        paths = map(str.strip, paths)

    load_include_path(paths)     ❷

    for path in paths:
        ## 省略了确认 path 是否存在的代码行
        for f in os.listdir(path):

            # Are we a directory? If so, process down the tree
            fpath = os.path.join(path, f)
            if os.path.isdir(fpath):
                subcollectors = load_collectors([fpath])    ❸
                for key in subcollectors:      ❹
                    collectors[key] = subcollectors[key]

            # Ignore anything that isn't a .py file
            elif (os.path.isfile(fpath)
                  ## 省略了确认 fpath 是否是一个 Python 模块的条件判断代码
                  ):
                ## 省略了被过滤路径的代码段
                modname = f[:-3]

                try:
```

```
                    # Import the module
                    mod = __import__(modname, globals(), locals(), ['*'])❺
                except (KeyboardInterrupt, SystemExit), err:
                    ## Log the exception and quit
                except:
                    ## Log the exception and continue

                # Find all classes defined in the module
                for attrname in dir(mod):
                    attr = getattr(mod, attrname)    ❻
                    # Only attempt to load classes that are subclasses
                    # of Collectors but are not the base Collector class
                    if (inspect.isclass(attr)
                            and issubclass(attr, Collector)
                            and attr != Collector):
                        if attrname.startswith('parent_'):
                            continue
                        # Get class name
                        fqcn = '.'.join([modname, attrname])
                        try:
                            # Load Collector class
                            cls = load_dynamic_class(fqcn, Collector)   ❼
                            # Add Collector class
                            collectors[cls.__name__] = cls   ❽
                        except Exception:
                            ## log the exception and continue
    # Return Collector classes
    return collectors
```

❶ 分解字符串（第一个函数调用），否则 paths 是用户自定义 Collector 子类所在路径的列表。

❷ 递归向下遍历给定的路径，将遍历到的每个路径插入 sys.path 中，以便这些路径下的收集器随后能被导入。

❸ 此处是递归调用，load_collectors() 调用自身。[8]

❹ 在加载了各个子目录下的收集器后，以来自这些子目录的新收集器更新原有的自定义收集器字典。

❺ 自 Python 3.1 起，Python 标准库中的 importlib 模块提供了一个完成该功能的首选方法（通过 importlib.import_module 实现；importlib.import_module 部分功能也移植到了 Python 2.7 中）。这里演示了：给定模块的字符串名称，如何通过编程方式导入一

8 Python 默认对递归调用深度有限制（允许函数调用自身的最大次数），主要是为了防止过度使用递归。通过 import sys;sys.getrecursionlimit() 可以查看所在 Python 环境的递归限制。

个模块。

❻ 这行演示了：仅给定属性的字符串名称，如何通过编程方式访问一个模块中的属性。

❼ 实际上，此处 load_dynamic_class 也许不是必要的。它重新导入模块，检查指定的类是否确实是一个类，并检查它是否确实是一个收集器。如果是，则返回新加载的类。大型团队编写的开源代码中有时会出现重复冗余的代码。

❽ 此处演示了如何获取类名，这样以后仅给定类的字符串名称就可以应用配置选项了。

取自 Diamond 的风格示例

Diamond 中有一个不错的闭包使用范例，适合解释延迟绑定的闭包，闭包的延迟绑定通常是一个大家期望的行为。

闭包的使用示例（此时陷阱不再是陷阱）

闭包是一种函数，它可以利用在局部作用域中可用但在函数调用时不可用的变量。在其他语言中闭包可能很难实现和理解，但在 Python 中并不难实现，因为 Python 对待函数就像对待其他对象一样[9]。例如，函数可以作为参数进行传递，或者从其他函数中返回。

如下例子摘录自 diamond 可执行文件，演示了在 Python 中如何实现一个闭包。

```
##~~ 省略了 import 语句                    ❶
def main():
   try:
       ##~~ 省略了创建命令行解析器的代码

       # Parse command-line Args
       (options, args) = parser.parse_args()

       ##~~ 省略了解析配置文件的代码
       ##~~ 省略了设置日志记录器的代码

   # Pass the exit upstream rather then handle it as an general exception
   except SystemExit, e:
       raise SystemExit

   ##~~ 省略了处理其他设置相关异常的代码
   try:
       # PID MANAGEMENT      ❷
       if not options.skip_pidfile:
```

9 这样的编程语言大家通常说它具有一等函数，即函数和其他对象一样，被作为“一等公民”对待。

```
    # Initialize PID file
    if not options.pidfile:
        options.pidfile = str(config['server']['pid_file'])

    ##~~ 省略了这块代码逻辑：如果 PID 文件存在，则打开并读取内容
    ##~~ 如果其内容不是一个 PID，则删除该文件
    ##~~ 或者如果已有一个进程正在运行，则退出


##~~ 省略了设置进程用户 ID 和用户组的代码
##~~ 以及变更 PID 文件权限的代码

##~~ 省略了这块代码逻辑：检测是否以后台守护进程的方式来运行程序
##~~ 如果是，则分离进程


# PID MANAGEMENT    ❸
if not options.skip_pidfile:
    # Finish initializing PID file
    if not options.foreground and not options.collector:
        # Write PID file
        pid = str(os.getpid())
        try:
            pf = file(options.pidfile, 'w+')
        except IOError, e:
            log.error("Failed to write child PID file: %s" % (e))
            sys.exit(1)
        pf.write("%s\n" % pid)
        pf.close()
        # Log
        log.debug("Wrote child PID file: %s" % (options.pidfile))

# Initialize server
server = Server(configfile=options.configfile)

def sigint_handler(signum, frame):    ❹
    log.info("Signal Received: %d" % (signum))
    # Delete PID file
    if not options.skip_pidfile and os.path.exists(options.pidfile): ❺
        os.remove(options.pidfile)
        # Log
        log.debug("Removed PID file: %s" % (options.pidfile))
    sys.exit(0)
```

```
        # Set the signal handlers
        signal.signal(signal.SIGINT, sigint_handler)    ❻
        signal.signal(signal.SIGTERM, sigint_handler)

        server.run()

    # Pass the exit upstream rather then handle it as a general exception
    except SystemExit, e:
        raise SystemExit

    ##~~ 省略了处理其他异常的代码
    ##~~ 以及脚本的所有剩余部分
```

❶ 当我们省略代码时，缺失的部分将使用两个波浪符号开头的注释来替代（像 ##~~ 这样）。

❷ 使用 PID[10] 文件是为了确保后台守护进程只有一个（也即，不会意外地启动两次），为了将关联的进程 ID 快速地告知其他脚本，以及为了在异常终止时有证据可以证明（因为在这个脚本中，PID 文件在正常终止时会被删除）。

❸ 这段代码只是为了在闭包之前提供上下文。此时，进程要么作为守护进程正在运行（并且现在它的进程 ID 与之前的不同），要么跳过这部分，因为它已经将正确的 PID 写入 PID 文件中。（译注：译者认为此处叙述可能有误！）

❹ 此处 sigint_handler() 函数就是闭包。它定义在 main() 函数内部，而不是在所有函数之外的顶层，因为它需要知道是否要去查找 PID 文件，以及如果需要，又应该去哪里查找。

❺ 直到调用 main() 后，它才能从命令行选项中获取这些信息。这意味着所有与 PID 文件相关的配置选项都是 main 命名空间中的局部变量。

❻ 闭包（函数 sigint_handler()）被发送到信号处理器中，用于处理信号 SIGINT 和 SIGTERM。

Tablib

Tablib 是一个 Python 库，可以进行不同格式之间的数据转换，转换期间数据存放在一个 Dataset（数据集）对象中，或者存为一个 Databook 对象中的多个数据集。它可以从 JSON、YAML、DBF 或 CSV 等文件格式导入数据集，也可以把数据集导出成 XLSX、

10 PID 表示进程标识符。每个进程都有一个唯一标识符，在 Python 中，使用标准库中 os 块下的 os.getpid() 可以获取。

XLS、ODS、JSON、YAML、DBF、CSV、TSV 或 HTML 等格式的文件。Kenneth Reitz 于 2010 年首次发布 Tablib，它具有 Reitz 项目典型的直观 API 设计风格。

阅读一个小型库

Tablib 是一个库而非一个应用程序，因此不像 HowDoI 和 Diamond 那样有一个明显的单一运行入口点。

从 GitHub 上获取 Tablib ：

```
$ git clone https://github.com/kennethreitz/tablib.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd tablib
(venv)$ pip install --editable .
(venv)$ python test_tablib.py # 运行单元测试
```

阅读 Tablib 文档

Tablib 文档参见 *http://docs.python-tablib.org/*，Tablib 文档开篇提供了一个使用案例，然后详细描述了其能力：它提供一个 Dataset 对象，该对象包含数据行、数据标题、数据列属性。它可以在 Dateset 对象与多种格式的文件之间进行输入和输出。文档的高级用法部分提到：它可以为表行打上标签，也可以创建动态派生数据列，由其他数据列通过函数计算而来）。

使用 Tablib

Tablib 是一个库，而不是像 HowDoI 或 Diamond 那样的可执行应用程序。因此可以打开一个 Python 交互式会话，并使用 help() 函数来探索其 API。如下示例演示了 tablib.Dataset 类、多种数据格式的使用，以及如何进行 I/O 操作。

```
>>> import tablib
>>> data = tablib.Dataset()
>>> names = ('Black Knight', 'Killer Rabbit')
>>>
>>> for name in names:
...     fname, lname = name.split()
...     data.append((fname, lname))
...
>>> data.dict
[['Black', 'Knight'], ['Killer', 'Rabbit']]
>>>
>>> print(data.csv)
Black,Knight
```

```
Killer,Rabbit

>>> data.headers=('First name', 'Last name')
>>> print(data.yaml)
- {First name: Black, Last name: Knight}
- {First name: Killer, Last name: Rabbit}
>>> with open('tmp.csv', 'w') as outfile:
...     outfile.write(data.csv)
...
64
>>> newdata = tablib.Dataset()
>>> newdata.csv = open('tmp.csv').read()
>>> print(newdata.yaml)
- {First name: Black, Last name: Knight}
- {First name: Killer, Last name: Rabbit}
```

阅读 Tablib 代码

tablib/ 目录下的文件结构如下所示。

```
tablib
|--- __init__.py
|--- compat.py
|--- core.py
|--- formats/
|--- packages/
```

tablib/formats/ 和 tablib/package/ 这两个目录将在后面讨论。

除了之前已经介绍过的文档字符串（一个字符串字面量，即函数、类或类方法中的第一个语句），Python 还支持模块级的文档字符串。对于如何描述一个模块，Stack Overflow 上提供了一个不错的建议，参见 *http://stackoverflow.com/a/2557196*。于我们而言，这意味着探索源代码的另一种方法是，在该 Python 包的顶级目录下，从终端中输入 head *.py 来一次性显示所有模块的文档字符串可以得到：

```
(venv)$ cd tablib
(venv)$ head *.py
==> __init__.py <==                          ❶
""" Tablib. """

from tablib.core import (
    Databook, Dataset, detect, import_set, import_book,
    InvalidDatasetType, InvalidDimensions, UnsupportedFormat,
    __version__
)
```

```
==> compat.py <==                    ❷
# -*- coding: utf-8 -*-

"""
tablib.compat
~~~~~~~~~~~~~
Tablib compatiblity module.
"""


==> core.py <==                      ❸
# -*- coding: utf-8 -*-
"""
    tablib.core
    ~~~~~~~~~~~

    This module implements the central Tablib objects.
    :copyright: (c) 2014 by Kenneth Reitz.
    :license:  MIT, see LICENSE for more details.
"""
```

从中我们了解到：

❶ 顶层 API（__init__.py 的内容在 import tablib 语句之后，可以从 tablib 中访问到的 API）只有 9 个入口点：文档中提到的 Databook 和 Dataset 类、用于识别格式的 detect 函数、用于导入数据的 import_set 和 import_book 等。最后 3 个类——InvalidDatasetType、InvalidDimensions 和 UnsupportedFormat——看似异常类（如果代码遵循 PEP 8 规范，则可以从名字的大小写分辨出哪些是自定义类）。

❷ tablib/compat.py 是一个兼容性模块。快速浏览一下会发现它以类似于 HowDoI 的方式来处理 Python 2/Python 3 的兼容性问题，即通过将不同的位置和名字解析成相同的符号以便 tablib/core.py 使用。

❸ tablib/core.py 实现了 Dataset 和 Databook 这类核心的 Tablib 对象。

Tablib 的 Sphinx 文档

Tablib 的文档提供了一个非常好的 Sphinx 使用范例（*http://www.sphinx-doc.org/en/stable/extensions.html*），因为该文档比较小，却使用了大量的 Sphinx 扩展。

该文档的最新 Sphinx 构建版本参见 Tablib 的官方文档（*http://docs.python-tablib.org/*）。如果想构建该文档（Windows 用户需要使用 make 命令），那么可以按照如下步骤进行。

```
(venv)$ pip install sphinx
(venv)$ cd docs
(venv)$ make html
(venv)$ open _build/html/index.html # 查看构建结果
```

Sphinx 自带一些默认配置的布局模板和 CSS 主题，还提供大量的主题配置项（*http://www.sphinx-doc.org/en/stable/theming.html*）。Tablib 文档左边栏上两个说明块的模板在 docs/_templates/ 目录下。它们的名字不是随意取的，都会在 basic/layout.html 中使用。可以在 Sphinx 主题目录（在命令行中输入如下命令可以找到该目录的位置）下找到该文件。

```
(venv)$ python -c 'import sphinx.themes;print(sphinx.themes.__
path__)'
```

高级用户也可以看一看 docs/_themes/kr/ 目录下的内容，这是一个对基本布局进行扩展的自定义主题。通过将 _themes/ 目录加入系统路径中，并在 docs/conf.py 中设置 html_theme_path = ['_themes'] 及 html_theme = 'kr' 即可选用该主题。

若想为代码从文档字符串自动生成 API 文档，可以使用 autoclass::。为了实际生效，请严格按照 Tablib 的 API 文档源码中的格式来写，如下所示。

```
.. autoclass:: Dataset
    :inherited-members:
```

为了启用该功能，在执行 sphinx-quickstart 创建新 Sphinx 工程时，在提示是否包含 autodoc Sphinx 扩展时请回答是。:inherited-members: 指令也会为从父类继承来的属性添加相关文档。

取自 Tablib 的结构示例

对于 Tablib，我们想强调的一个重点是：tablib/formats/ 目录下的模块没有使用类，是“不要过度使用类”这条规范的完美示范。我们从 Tablib 中摘录代码，展示 Tablib 如何使用装饰器语法和 property 类来创建数据集的高度和宽度等派生属性，以及如何动态注册文件格式来避免重复为每种格式类型（CSV、YAML 等）编写一些样板代码。

最后两个小节有点晦涩。先看看 Tablib 如何管理依赖，然后讨论一下新式类对象的 __slots__ 属性。跳过这两节，也无伤大雅。

无须面向对象代码来实现对各种文件格式的支持（使用命名空间来组织函数）

formats 目录下包含了为输入输出定义的所有文件格式。_csv.py、_tsv.py、_json.py、_yaml.py、_xls.py、_xlsx.py、_ods.py 及 _xls.py 等模块名称都带一个下画线前缀，用于向用户表明这些模块不应该直接被使用。切换到 formats 目录下可以搜索其中的类和函数。使用 grep ^class formats/*.py 命令会发现里面没有任何类定义，再使用 grep ^def formats/*.py 命令会发现每个模块都包含下列部分或全部的函数。

- detect(stream) 根据流内容推断文件格式。
- dset_sheet(dataset, ws) 格式化 Excel 表格单元。
- export_set(dataset) 将数据集导出为给定的格式，并返回一个用新格式格式化后的字符串（或者，对于 Excel，返回一个 bytes 对象；在 Python 2 中则是返回一个二进制格式化的字符串）。
- import_set(dset, in_stream, headers=True) 以输入流的内容替换数据集的内容。
- export_book(databook) 将 Databook 中的数据集导出为指定格式，返回一个字符串或者 bytes 对象。
- import_book(dbook, in_stream, headers=True) 以输入流的内容替换 Databook 的内容。

这些模块的代码示范了如何使用模块作为命名空间来分离函数，而不是使用不必要的类。对于其中的每个函数，我们都能顾名思义。例如，formats._csv.import_set()、formats._tsv.import_set() 和 formats._json.import_set() 分别是从 CSV、TSV 和 JSON 格式的文件中导入数据集。其他函数则是为每种 Tablib 支持的格式进行数据导出和文件格式检测。

描述符和属性装饰器（制造不可变性以实现更好的 API）

Tablib 是本书介绍的第一个使用 Python 装饰器语法的库。该语法在一个函数名称前加一个 @ 符号，然后直接放在另一个函数定义的上面。它会修改（或者说“装饰”）下方紧邻的函数。在如下摘录的代码片段中，property 装饰器将函数 Dataset.height 和 Dataset.width 转变成了描述符，至少定义了属性值获取方法（getter）__get()__、_ 属性设置方法（setter）_set__() 和属性删除方法（delete）__delete__() 三个方法中一个方法的类。例如，属性查找操作 Dataset.height 会触发调用属性值获取方法、属性设置方法或属性删除方法，具体调用哪个方法依赖于上下文中如何使用属性。注意，只有当前讨论的新式类才具备这种行为。更多详情，请阅读 Python 描述符的官方教程（*https://docs.python.org/3/howto/descriptor.html*）。

```
class Dataset(object):
    #
    # ... 省略了类定义的其余部分
    #

    @property                                    ❶
    def height(self):
        """The number of rows currently in the :class:'Dataset'.
           Cannot be directly modified.    ❷
        """
        return len(self._data)


    @property
    def width(self):
        """The number of columns currently in the :class:'Dataset'.
           Cannot be directly modified.
        """
        try:
            return len(self._data[0])
        except IndexError:
            try:
                return len(self.headers)
            except TypeError:
                return 0
```

❶ 此处演示了如何使用一个装饰器。在例子中，property 将 Dataset.height 从行为表现上修改为属性而不是一个绑定的方法，它只能使用在类方法上。

❷ 当 property 被用作一个装饰器时，height 属性将返回 Dataset 的高度，但不能调用 Dataset.height 为 Dataset 赋值一个高度。

height 和 width 属性的使用效果如下所示。

```
>>> import tablib
>>> data = tablib.Dataset()
>>> data.header = ("amount", "ingredient")
>>> data.append(("2 cubes", "Arcturan Mega-gin"))
>>> data.width
2
>>> data.height
1
>>>
>>> data.height = 3
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

因此，data.height 可以像属性一样访问，但不可设置。因为它是从数据中计算出来的，所以总是最新的。这符合人体工程学的 API 设计：data.height 比 data.get_height() 更容易输入；data.height 的含义也很明确，并且因为它是从数据中计算得到的（属性不可设置，只定义了 getter 功能），所以不存在未同步正确值的风险。

property 装饰器只能用于类的属性，而且只能用于从基类 object 继承来的类（例如，class MyClass(object)），不过在 Python 3 中都是从 object 继承的。

Tablib 在为各种文件格式创建数据导入和导出 API 时也使用了同样的工具。Tablib 不保存 CSV、JSON 及 YAML 等各种格式的字符串输出值。和前面例子中的 Dataset.height、Dataset.width 一样，Dataset 的 CSV、JSON 及 YAML 都是属性，它们调用一个函数从已保存在内存中的数据生成结果，或者解析输入格式，然后替换核心数据，但自始至终内存中的数据集都只有一份。

当 data.csv 在等号左边时，将调用属性的 setter 函数，从 CSV 格式数据中解析出数据集。当 data.yaml 在等号右边或者单独使用时，将调用属性的 getter 函数，从内部数据集创建一个指定格式的字符串，如下所示。

```
>>> import tablib
>>> data = tablib.Dataset()
>>>
>>> data.csv = "\n".join((                          ❶
...     "amount,ingredient",
...     "1 bottle,Ol' Janx Spirit",
...     "1 measure,Santraginus V seawater",
...     "2 cubes,Arcturan Mega-gin",
...     "4 litres,Fallian marsh gas",
...     "1 measure,Qalactin Hypermint extract",
...     "1 tooth,Algolian Suntiger",
...     "a sprinkle,Zamphuor",
...     "1 whole,olive"))
>>>
>>> data[2:4]
[('2 cubes', 'Arcturan Mega-gin'), ('4 litres', 'Fallian marsh gas')]
>>>
>>> print(data.yaml)                                ❷
- {amount: 1 bottle, ingredient: Ol Janx Spirit}
- {amount: 1 measure, ingredient: Santraginus V seawater}
- {amount: 2 cubes, ingredient: Arcturan Mega-gin}
- {amount: 4 litres, ingredient: Fallian marsh gas}
```

```
    - {amount: 1 measure, ingredient: Qalactin Hypermint extract}
- {amount: 1 tooth, ingredient: Algolian Suntiger}
- {amount: a sprinkle, ingredient: Zamphuor}
- {amount: 1 whole, ingredient: olive}
```

❶ data.csv 位于等号（赋值运算符）左侧，会调用 formats.csv.import_set()，以 data 作为第一个参数，含“漱爆破液（Gargle Blaster）”（译注：Gargle Blaster 一词出自科幻小说《银河系漫游指南》）的原料作为第二个参数。

❷ 单独使用 data.yaml 会调用 formats.yaml.export_set()，以 data 作为参数，为 print() 函数输出经过格式化的 YAML 字符串。

getter、setter 及 delete 函数可以通过使用 property 绑定到同一个属性上。其函数签名是 property(fget=None, fset=None, fdel=None, doc=None)，其中 fget 参数指定 getter 函数（如 formats.csv.import_set()），fset 参数指定 setter 函数（如 formats.csv.export_set()），fdel 参数指定 delete 函数（以 None 来占位）。接下来，我们将阅读以编程方式设置格式化属性的代码。

以编程方式注册文件格式（不要重复你自己）

Tablib 将所有文件格式化相关的代码都放在 formats 子包中。这一工程结构选择使得 core.py 主模块更简单，也使得整个包的模块化更好，易于添加新的文件格式。虽然这样也可能会导致粘贴一些几近相同的代码块，并且要分别导入每个文件格式的导入和导出行为，但实际上 Tablib 中所有格式都以编程方式自动加载到 Dataset 类的属性上，属性名对应格式名称。

在如下代码示例中，我们将 formats/__init__.py 文件的全部内容都打印出来，因为这个文件并不太大，而且我们想要展示一下 formats.available 定义的位置。

```
# -*- coding: utf-8 -*-    ❶

""" Tablib - formats
"""
from . import _csv as csv
from . import _json as json
from . import _xls as xls
from . import _yaml as yaml
from . import _tsv as tsv
from . import _html as html
from . import _xlsx as xlsx
from . import _ods as ods
```

```
    available = (json, xls, yaml, csv, tsv, html, xlsx, ods)    ❷
```

❶ 这一行明确告诉 Python 解释器文件编码是 UTF-8[11]。

❷ 此处是 formats.available 的定义。通过 dir(tablib.formats) 也能看到，但是这个显式的列表更容易理解。

在 core.py 中，不是针对所有格式重复定义大约 20 个函数（这样既不美观，也难以维护），而是在 Dataset 的 __init__() 方法结尾通过调用 self._register_formats() 方法以编程方式导入每个格式。此处只摘录了 Dataset._register_formats() 的代码。

```
class Dataset(object):
    #
    #  省略了文档和一些定义
    #

    @classmethod    ❶
    def _register_formats(cls):
        """Adds format properties."""
        for fmt in formats.available:    ❷
            try:
                try:
                    setattr(cls, fmt.title,
                    property(fmt.export_set, fmt.import_set))    ❸
                except AttributeError:    ❹
                    setattr(cls, fmt.title, property(fmt.export_set))    ❺
            except AttributeError:
                pass    ❻
#
#  省略了更多的定义
#

@property  ❼
def tsv():
    """A TSV representation of the :class:'Dataset' object. The top
    row will contain headers, if they have been set. Otherwise, the
    top row will contain the first row of the dataset.

    A dataset object can also be imported by setting
    the :class:'Dataset.tsv' attribute. ::

        data = tablib.Dataset()
        data.tsv = 'age\tfirst_name\tlast_name\n90\tJohn\tAdams'  ❽
```

11 Python 2 中默认是 ASCII 编码，Python 3 中默认是 UTF-8 编码。PEP 263 中列举了多种允许使用的文件编码指定方案，你可以选择最适合常用文本编辑器的方案。

```
        Import assumes (for now) that headers exist.
        """
        pass
```

❶ @classmethod 是一个装饰器，它修改了 _register_formats() 方法，使得该方法的第一个参数是传递对象的类（Dataset）而不是对象的实例。

❷ formats.available 定义在 formats/__init__.py 文件中，包含了所有支持的格式化方案。

❸ 此行中，setattr 将一个值赋值给一个属性，属性名为 fmt.title 的值（例如，Dataset.csv 或 Dataset.xls）。它赋的值比较特殊，property(fmt.export_set, fmt.import_set) 可以将 Dataset.csv 转变成一个属性。

❹ 如果 fmt.import_set 未定义，则会抛出 AttributeError 异常。

❺ 如果没有导入功能，则尝试只赋值导出行为。

❻ 如果既没有导出也没有导入功能，则什么都不干。

❼ 此处对应每个文件格式分别定义一个属性，附带描述性的文档字符串。在标签 3 或 5 处调用 property() 为属性赋值额外行为时，文档字符串会被保留。

❽ \t 和 \n 分别表示 Tab 字符和换行的字符串转义序列，Python 官方的字符串字面量文档（*https://docs.python.org/3/reference/lexical_analysis.html#index-18*）中罗列了所有的转义字符。

我们都是负责任的用户

@property 装饰器的这些用法不像 Java 中类似工具的用法，Java 中类似工具的目标是控制用户对数据的访问，这违背了 Python 哲学——我们都是负责任的用户。@property 目的是将数据相关的视图函数与数据（例如，当前案例中的高度、宽度及各种存储格式）相分离。如果不需要预处理或者后处理的 getter 或 setter 函数，那么更符合 Python 风格的方式是直接给一个常规属性赋值数据，让用户直接与其交互。

在软件包中捆绑依赖（关于如何捆绑依赖的一个例子）

Tablib 的依赖目前是源码包含的（意思是依赖是与项目源码一起捆绑交付的，在当前例子中依赖源码放在 packages 目录下），不过未来可能迁移实现为一个插件系统（译注：

Tablib 最新实现中将大部分依赖库都改成了外部依赖，放在 requirements.txt 文件中）。packages 目录下包含了第三方依赖包的源码，以保证兼容性，而不是在 setup.py 文件中指定依赖版本，在安装 Tablib 时下载安装它。Tablib 的这种做法减少了用户必须下载的依赖的数量，并且因为某些依赖针对 Python 2 和 Python 3 提供不同的软件包，所以 Tablib 中把依赖的两个对应版本都包含了。在 tablib/compat.py 中根据 Python 版本导入恰当的那个依赖版本，并将依赖的函数设置为共同的名字。这样 Tablib 只需要一个代码库而不是两个分别对应 Python 2 和 Python 3 的代码库。因为这些依赖各自有许可证，所以项目的顶级目录中添加了一个 NOTICE 文档，列出每个依赖的许可证。

使用 __slots__ 来节约内存（明智而审慎地进行优化）

相较于运行速度，Python 更在意代码的可读性。由于 Python 整体设计、禅语以及早期受到教学型编程语言（如 ABC 语言）的影响（*http://bit.ly/abc-to-python*）等原因，因此 Python 把用户友好看得比性能更重。

因为此处性能优化至关重要，所以 Tablib 中使用 __slots__。这个优化方案可能有点令人费解，并且只适用于新式类，但是必须说明一下必要之时这个方案是可以优化 Python 代码的。这种优化仅当存在大量非常小的对象时才有效果，其通过为每个类的实例减少一个字典大小的内存空间占用来实现优化（对于大型对象而言，节省这一点点内存空间，无关痛痒；对于少量的对象而言，又不值得做这个优化）。如下解释摘自 __slots__ 文档（*http://bit.ly/__slots__-doc*）：

> 在默认情况下，类的实例都有一个保存属性的字典。如果类仅有很少的实例变量，就有点浪费内存空间了。在创建大量的实例时，内存消耗将变得非常严重。
>
> 通过在类中定义 __slots__ 可以覆盖默认行为。__slots__ 声明指定一个实例变量序列，在每个实例中只保留足够的空间来保存每个变量的值。因为每个实例都没有创建 __dict__，所以就节省了内存空间。

通常，这不是需要关心的事情。注意 __slots__ 没有出现在 Dataset 和 Databook 类中，它只出现在 Row 类中。因为可能有成千上万行的数据，所以使用 __slots__ 是个好主意。Row 类没有暴露在 tablib/__init__.py 中，因为它是 Dataset 的辅助类，对每行数据实例化一次。如下所示是 Row 类定义的开始部分，其中可以看到 __slot__ 的声明。

```
class Row(object):
    """Internal Row object. Mainly used for filtering."""

    __slots__ = ['_row', 'tags']

    def __init__(self, row=list(), tags=list()):
```

```
            self._row = list(row)
            self.tags = list(tags)

        #
        #  省略
        #
```

现在的问题是 Row 的实例中不再有 __dict__ 属性了，但是 pickle.dump() 函数（用于对象序列化）默认使用 __dict__ 来序列化对象，除非定义了 __getstate__() 方法。同样，在 pickle 反序列化过程中（读取经过序列化的字节序列并在内存中重建对象），如果没有定义 __setstate()__ 方法，pickle.load() 会把实例变量都导入对象的 __dict__ 属性中。解决方法如下所示。

```
    class Row(object):
        #
        # 省略了其他定义
        #

        def __getstate__(self):

            slots = dict()

            for slot in self.__slots__:
                attribute = getattr(self, slot)
                slots[slot] = attribute
            return slots

        def __setstate__(self, state):
            for (k, v) in list(state.items()):
                setattr(self, k, v)
```

更多关于 __getstate__()、__setstate__() 及 pickle 序列化的信息，请阅读 __getstate__ 文档（*http://bit.ly/__getstate__-doc*）。

取自 Tablib 的风格示例

本节仅介绍一个取自 Tablib 的风格示例。运算符重载将深入 Python 数据模型的细节当中，为类定制行为以便于别人使用你的 API 编写出优美的代码。

运算符重载（优美胜于丑陋）

下面的代码使用 Python 的运算符重载对数据集的行或列进行操作。第一段示例代码展示了针对数字索引和列名称交互式地使用括号运算符（[]）。第二段示例代码展示了使用这种行为的完整代码。

```
>>> data[-1]   ❶
('1 whole', 'olive')
>>>
>>> data[-1] = ['2 whole', 'olives']   ❷
>>>
>>> data[-1]
('2 whole', 'olives')   ❸
>>>
>>> del data[2:7]  ❹
>>>
>>> print(data.csv)
amount,ingredient  ❺
1 bottle,Ol' Janx Spirit
1 measure,Santraginus V seawater
2 whole,olives

>>> data['ingredient']  ❻
["Ol' Janx Spirit", 'Santraginus V seawater', 'olives']
```

❶ 如果使用数字，通过括号运算符（[]）访问数据时会给出指定位置上的数据行。

❷ 这个赋值操作使用了括号运算符。

❸ 从原来的一个橄榄变成了两个橄榄。

❹ 使用一个分片进行删除操作，2:7 表示数字 2、3、4、5、6，但不包括 7。

❺ 可以看到之后这个配方变小了很多。

❻ 也可以通过名称来访问数据列。

如下所示是 Dataset 中定义括号操作符行为的代码段，展示了对于通过列名或行号进行数据访问是怎么处理的。

```
class Dataset(object):
    #
    #  省略了其余定义
    #

    def __getitem__(self, key):
        if isinstance(key, str) or isinstance(key, unicode):   ❶
            if key in self.headers:   ❷
                pos = self.headers.index(key) # get 'key' index from each
data
                return [row[pos] for row in self._data]
            else:  ❸
```

```
                raise KeyError
            else:
                _results = self._data[key]
                if isinstance(_results, Row):  ❹
                    return _results.tuple
                else:
                    return [result.tuple for result in _results]  ❺

        def __setitem__(self, key, value):  ❻
            self._validate(value)
            self._data[key] = Row(value)

        def __delitem__(self, key):
            if isinstance(key, str) or isinstance(key, unicode):  ❼
                if key in self.headers:
                    pos = self.headers.index(key)
                    del self.headers[pos]

                    for row in self._data:
                        del row[pos]
                else:
                    raise KeyError
            else:
                del self._data[key]
```

❶ 首先，检查当前正在找一个列（如果 key 是一个字符串，则为 True）还是一个行（如果 key 是一个整数或分片，则为 True）。

❷ 此处代码检查 key 参数值是否在 self.headers 里。

❸ 如果不在，则显式地抛出 KeyError 异常，以便像访问字典一样通过列名来访问。整个 if/else 块对于该函数的操作而言不是必需的，如果忽略了它，当 key 不在 self.headers 中时，通过 self.headers.index(key) 访问仍然会触发一个 ValueError 异常。加上这个检查的唯一目的是为库的使用者提供一个信息更丰富的错误。

❹ 从这一行可以看出代码是如何判断 key 是一个数字还是一个分片的（例如，2:7）。如果是分片，那么 _results 是一个列表，而不是一个 Row。

❺ 此处代码负责处理分片。因为数据行作为元组返回，其中的值是实际数据的一个不可变副本，这样数据集的值（实际是存储为列表）不会被赋值操作意外破坏。

❻ __setitem__() 方法可以修改一行数据但不能修改一列，这是有意为之的。并没有提供方法来修改一整列的内容，出于数据完整性的考虑，这可能是一个不错的选择。用户可以随时对数据列进行转换，也可以使用 insert_col()、lpush_col() 或 rpush_col() 中

的一种方法在任意位置上插入数据列。

- __delitem__() 方法可以删除一行或一列，和 __getitem__() 的使用逻辑一样。

关于其他操作符重载及特殊方法，若想了解更多，请参阅 Python 官方文档中特殊方法名一节（*http://bit.ly/special-method-names*）。

Requests

2011 年的情人节，Kenneth Reitz 向 Python 社区发布了 Requests 库。这个 Python 库采用了直观的 API 设计风格（意即 API 足够直接明了，用户几乎不需要阅读文档）。

阅读一个更大的库

Requests 是一个比 Tablib 更大的库，有更多的模块，不过我们仍然以同样的方式阅读它，即首先阅读其文档，然后追踪代码中的 API。

从 GitHub 获取 Requests ：

```
$ git clone https://github.com/kennethreitz/requests.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd requests
(venv)$ pip install --editable .
(venv)$ pip install -r requirements.txt # 单元测试需要
(venv)$ py.test tests # 运行单元测试
```

某些测试用例可能会失败，例如，网络服务提供商拦截 404 错误给你返回广告页面，那将不会得到 ConnectionError 异常。

阅读 Requests 文档

因为 Requests 是一个更大的包，所以先浏览下 Requests 文档的章节标题（*http://docs.python-requests.org/*）。Requests 对 Python 标准库中的 urllib 和 httplib 进行扩展，提供一些执行 HTTP 请求的方法。该库支持国际化域名、URL、自动解压缩、内容编码自动识别、浏览器风格的 SSL 验证、HTTP(S) 代理支持及其他功能特性，互联网工程任务组（IETF）HTTP 标准 RFC 7230 到 7235 中定义了这些特性 [12]。

Requests 力求只使用少量函数、一些关键字参数及若干个特性丰富的类来覆盖全部的 IETF HTTP 规范。

12 如果需要更新自己的词汇表，推荐阅读 RFC 7231 - HTTP 语义文档（*http://bit.ly/http-semantics*）。仅需浏览其目录并阅读简介部分，便足以了解其定义范围，确定其中是否包含自己需要的定义，以及从何处找到它。

使用 Requests

和 Tablib 一样，即使没有实际阅读在线文档，文档字符串也已提供足够的信息来帮助用户上手 Requests。如下是一个简短的交互式使用。

```
>>> import requests
>>> help(requests) # 显示用法说明，并告知查看 requests.api
>>> help(requests.api) # 显示 API 的详细描述
>>>
>>> result = requests.get('https://pypi.python.org/pypi/requests/json')
>>> result.status_code
200
>>> result.ok
True
>>> result.text[:42]
'{\n "info": {\n "maintainer": null'
>>>
>>> result.json().keys()
dict_keys(['info', 'releases', 'urls'])
>>>
>>> result.json()['info']['summary']
'Python HTTP for Humans.'
```

阅读 Requests 代码

如下是 Requests 包的内容。

```
$ ls
__init__.py     cacert.pem  ❶   exceptions.py sessions.py
adapters.py     certs.py    hooks.py status_codes.py
api.py compat.py models.py structures.py
auth.py cookies.py packages/  ❷   utils.py
```

❶ cacert.pem 在检查 SSL 证书时默认使用的证书包。

❷ Requests 工程结构扁平，包含外部依赖库 chardet 和 urllib3 源码的 packages 目录（译注：Requests 最新实现中已去掉 packages 目录，在 setup.py 中声明外部依赖库及版本）。这两个依赖以 requests.packages.chardet 和 requests.packages.urllib3 名字导入，因此程序员依然可以访问 chardet 和标准库中的 urllib3。

从上述 Requests 包内容，我们就能基本清楚每个模块干了些什么，因为这些模块的名称都是经过精心选择的。不过欲知更多信息，可以在项目的顶级目录下执行 head *.py 查看模块的文档字符串。如下列表展示了这些模块的文档字符串，略有删节，没有展示 compat.py，因为从其名字我们就能知道它负责处理 Python 2 和 Python 3 的兼容性问题，特别是因为其名字和 Tablib 库中的那个相同。

1. api.py

实现 Requests API。

2. hooks.py

提供 Requests 钩子系统（hooks system）的能力。

3. models.py

包含支撑 Requests 的一些主要对象。

4. sessions.py

提供一个 Session 对象来管理和保持跨请求的一些设置（cookie、认证、网络代理）。

5. auth.py

包含一些 Requests 身份认证处理器。

6. status_codes.py

状态名到状态码的映射查找表。

7. cookies.py

兼容性代码，让请求能够使用 cookielib.CookieJar。

8. adapters.py

包含传输适配器，Requests 用它来定义和维护链接。

9. exceptions.py

包含所有的 Requests 异常。

10. structures.py

支撑 Requests 的一些数据结构。

11. certs.py

返回首选的默认 CA 证书包，其中罗列了受信任的 SSL 证书。

12. utils.py

提供一些 Requests 内部使用的工具函数，当然也适用于外部使用。

阅读这些模块的头部文档后，可以获得以下信息。

- 存在一个钩子系统，暗示着用户可以修改 Requests 的工作方式。对此不做深入讨论，因为这会让我们偏离主题太远。
- 主要模块是 models.py，因为其包含支撑 Requests 的一些主要对象。
- sessions.Session 存在的理由是跨多个请求保持 cookie（例如，身份认证过程中可能会出现此需求）。
- 实际的 HTTP 连接是由 adapters.py 中的对象完成的。
- 其余模块的用途则显而易见：auth.py 用于身份认证，status_codes.py 包含状态码，cookies.py 用于添加或删除 cookie，exceptions.py 提供一些自定义异常类，structures.py 包含一些数据结构（比如，不区分大小写的字典），utils.py 则包含一些工具函数。

将通信逻辑单独放在 adapters.py 中，这意味着 models.Request、models.PreparedRequest 及 models.Response 实际上不做任何事情，它们只存储数据，也可能为了展示、pickle 序列化或编码而做点数据操作。请求的相关动作由独立的类来处理，这些类专门用来执行某一个动作，比如，认证或者通信。每个类只做一件事情，每个模块包含处理相似事情的类。这种符合 Python 风格的方式，我们的函数定义基本都已遵循。

Requests 的文档字符串兼容 Sphinx

如果正开始一个新项目，使用 Sphinx 及其 autodoc 扩展，那么你写的文档字符串需要满足一定的格式，以便 Sphinx 可以解析。

Sphinx 文档并不总是易于搜索关键字的正确格式。如果想得到正确的格式，那么推荐做法是复制 Requests 中的文档字符串，而不是在 Sphinx 文档中查找指令。例如，requests/api.py 中 delete() 函数的定义如下所示：

```
def delete(url, **kwargs):
    """Sends a DELETE request.
    :param url: URL for the new :class:'Request' object.
    :param \*\*kwargs: Optional arguments that "request" takes.
    :return: :class:'Response <Response>' object
    :rtype: requests.Response
    """
    return request('delete', url, **kwargs)
```

Sphinx autodoc 对这个定义的渲染结果见在线 API 文档（*http://docs.python-requests.org/en/master/api/#requests.delete*）。

取自 Requests 的结构示例

大家都很喜欢 Requests 的 API，因为 Requests 的 API 容易记住，也有助于用户编写简单漂亮的代码。本节首先讨论一种设计偏好 requests.api 模块，为了提供更易于理解的错误信息和易于记忆的 API，这一点我们认为 requests.api 模块已经做到，然后探索 requests.Request 和 urllib.request.Request 对象之间的区别，最后对 requests.Requests 何以存在提出一种见解。

顶级 API（最好是仅有一种明显的实现方式）

除 request() 外，定义在 api.py 中的函数均以 HTTP 请求方法[13] 命名。除了方法名和对外暴露的关键字参数，每个请求方法都是相同的，因此我们将 requests/api.py 的内容在 get() 函数之后截断，摘录了如下代码：

```
"""
requests.api
~~~~~~~~~~~~

This module implements the Requests API.

:copyright: (c) 2012 by Kenneth Reitz.
:license: Apache2, see LICENSE for more details.

"""

from . import sessions

def request(method, url, **kwargs):  ❶
    """Constructs and sends a :class:'Request <Request>'.

    :param method: method for the new :class:'Request' object.
    :param url: URL for the new :class:'Request' object.
    :param params: (optional) Dictionary or bytes to be sent in the query
string
                    for the :class:'Request'.

    # 省略了其余关键字参数的文档          ❷

    :return: :class:'Response <Response>' object
    :rtype: requests.Response
```

13 这些方法都定义在超文本传输协议 RFC 最新版本的第 4.3 节中（*https://tools.ietf.org/html/rfc7231#section-4.3*）。

```
    Usage::

        >>> import requests
        >>> req = requests.request('GET', 'http://httpbin.org/get')
        <Response [200]>
    """

    # By using the 'with' statement, we are sure the session is closed, thus
we
    # avoid leaving sockets open which can trigger a ResourceWarning in some
    # cases, and look like a memory leak in others.
    with sessions.Session() as session:  ❸
        return session.request(method=method, url=url, **kwargs)


def get(url, params=None, **kwargs): ❹
    """Sends a GET request.

    :param url: URL for the new :class:'Request' object.
    :param params: (optional) Dictionary or bytes to be sent in the query
string
                    for the :class:'Request'.
    :param \*\*kwargs: Optional arguments that "request" takes.
    :return: :class:'Response <Response>' object
    :rtype: requests.Response
    """

    kwargs.setdefault('allow_redirects', True) ❺
    return request('get', url, params=params, **kwargs) ❻
```

❶ request() 函数在其签名中包含了一个 **kwargs。这意味着无关的关键字参数不会引发异常，也对用户隐藏了一下参数选项。

❷ 此处省略的文档是描述每个具有关联动作的关键字参数。如果在函数签名里使用了 **kwargs，那么除了看代码，用户只有通过这种方式获知 **kwargs 中应该使用什么。

❸ with 语句是 Python 支持运行时上下文的方式。它可以用于任何定义了 __enter__() 和 __exit()__ 方法的对象。一旦进入 with 语句便会调用 __enter__()，并在退出时调用 __exit__()，无论是正常退出还是由于异常退出，都会调用这两个方法。

❹ get() 函数特意把 params=None 关键字参数抽出来，对其应用默认值 None。params 关键字参数与 get 相关性比较大是因为它是用于传递参数生成 HTTP 查询字符串的。通过剩下的 **kwargs 暴露选择好的关键字参数，为高级用户提供足够的灵活性，同时，对于 99% 的不需要高级参数项的用户而言，API 的用法又足够明显。

❺ 因为 request() 函数默认不允许请求重定向，所以这一步将它设置为 True，用户也可以自己设置。

❻ get() 函数只是简单地调用一下 request() 函数，调用时将第一个参数设置为 get。定义 get 函数相比使用 request("get", ...) 有两个优点。第一，即使没有文档，也能知道 API 支持哪些 HTTP 方法。第二，如果用户输入了错误的方法名，则会在更早的时候抛出 NameError 异常，相比错误发生在代码深处，问题追溯起来受干扰的可能性更小。

requests/api.py 中没有添加新功能，其存在就是为了给用户提供简单的 API。此外，将 HTTP 方法名直接作为函数名意味着方法名的任何书写错误都能及早捕捉和识别，例如：

```
>>> requests.foo('http://www.python.org')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'foo'
>>>
>>> requests.request('foo', 'http://www.python.org')
<Response [403]>
```

Request 和 PreparedRequest 对象（我们都是负责任的用户）

__init__.py 将 models.py 中的 Request、PreparedRequest 及 Response 对象作为主 API 的一部分，那么 models.Request 存在的意义是什么呢？标准库里已经有一个 urllib.requests.Request，并且 cookies.py 里还特意定义了一个包装 models.Request 的 MockRequest 对象，这样，对 http.cookiejar[14] 来说，使用起来就和 urllib.requests.Request 一样。这意味着请求对象与这个 cookie 操作库交互所需要的任何方法都故意从 requests.Request 中排除了，那么所有这些额外工作的意义又是什么呢？

cookie 操作库使用 MockRequest（为 cookie 操作库模拟 urllib.request.Request）中额外添加的方法来管理 cookie。其中除了 get_type() 函数（使用 Requests 时该函数通常是返回 http 或 https）和 unverifiable 属性，其他方法或属性都与 URL 或者请求头相关。

1. 与请求头相关的方法 / 属性

add_unredirected_header()：向请求头中添加一对键值。

get_header()：指定请求头名称，从请求头字典中获取对应值。

get_new_headers()：返回包含新增请求头（由 cookielib 添加）的字典。

has_header()：检查请求头字典中是否存在某个请求头名称。

14 http.cookiejar 模块对应以前 Python 2 中的 cookielib，urllib.request.Request 对应以前 Python 2 中的 urllib2.Request。

2. 与 URL 相关的方法 / 属性

get_full_url() ：返回整个 url。

host 和 origin_req_host ：分别通过调用 get_host() 和 get_origin_req_host() 方法设置的属性。

get_host() ：从 URL 中提取 host（例如，*https://www.python.org/dev/peps/pep-0008/* 的 host 是 *www.python.org*）。

get_origin_req_host() ：直接调用 get_host()[15]。

除了 MockRequest.add_unredirected_header()，其他都是访问函数。MockRequest 的文档字符串指出原始的请求对象只读。

在 requests.Request 中，数据属性是直接暴露的。这样就不需要访问器函数了 ：直接访问 request-instance.headers 即可获取或者设置请求头。request-instance.headers 其实就是个字典。同样，用户可以这样获取或者修改 URL 字符串 request-instance.url。

PreparedRequest 对象初始化时数据属性都是空，需要调用 prepared-request-instance.prepare() 来填充相关数据（这些数据通常来自对 Request 对象的调用）。也正是在此时应用正确的大小写和编码。一旦准备好，对象的内容就会被发送到服务器上，但每个属性依然是直接对外暴露的。甚至 PreparedRequest._cookies 也是对外暴露的，虽然其前缀下画线善意地提醒 ：该属性不适合在类外使用，但也不禁止这样的访问。

将对象暴露给用户修改，使得程序更具可读性，并且 PreparedRequest 中也做了一点额外的工作来纠正大小写问题，允许使用字典代替 CookieJar 对象，参看如下代码中的 if isinstance()/else 语句。

```
#
#  取自 models.py 文件
#

class PreparedRequest():
    #
    #  省略了其他代码
    #

    def prepare_cookies(self, cookies):
        """Prepares the given HTTP cookie data.
```

15 该方法能够处理跨域请求（如获取托管在第三方网站上的一个 JavaScript 库）。该方法应该返回请求的原始 host，具体定义见 IETF RFC 2965（*http://bit.ly/http-state-management*）。

```
        This function eventually generates a "Cookie" header from the
        given cookies using cookielib. Due to cookielib's design, the header
        will not be regenerated if it already exists, meaning this function
        can only be called once for the life of the
        :class:'PreparedRequest <PreparedRequest>' object. Any subsequent
calls
        to "prepare_cookies" will have no actual effect, unless the "Cookie"
        header is removed beforehand."""
        if isinstance(cookies, cookielib.CookieJar):
            self._cookies = cookies
        else:
            self._cookies = cookiejar_from_dict(cookies)

        cookie_header = get_cookie_header(self._cookies, self)
        if cookie_header is not None:
            self.headers['Cookie'] = cookie_header
```

这些事情看起来没什么大不了的，但像这样的细微权衡让 API 更加直观易用。

取自 Requests 的风格示例

对于 Requests 的风格示例，我们会先看一个不错的集合使用示例（我们认为应该多使用集合），然后看一下 requests.status_codes 模块，因为这个模块的存在，避免了在项目代码中到处硬编码 HTTP 状态码，其余代码的风格也就更加简单了。

集合和集合运算（一个优雅的符合 Python 风格的习语）

到目前为止我们还没实战演示过 Python 集合的使用示例。Python 集合的行为类似于数学中的集合，可以进行差集、并集（or 运算符）和交集（and 运算符）运算。

```
>>> s1 = set((7,6))
>>> s2 = set((8,7))
>>> s1
{6, 7}
>>> s2
{8, 7}
>>> s1 - s2 # 差集
{6}
>>> s1 | s2 # 并集
{8, 6, 7}
>>> s1 & s2 # 交集
{7}
```

如下所示代码段取自 cookies.py 文件，这个函数的结尾（标 ❷ 处）处使用了集合运算。

```
    #
    #  取自 cookies.py 文件
    #

    def create_cookie(name, value, **kwargs):  ❶
        """Make a cookie from underspecified parameters.

        By default, the pair of 'name' and 'value' will be set for the domain ''
        and sent on every request (this is sometimes called a "supercookie").
        """
        result = dict(
            version=0,
            name=name,
            value=value,
            port=None,
            domain='',
            path='/',
            secure=False,
            expires=None,
            discard=True,
            comment=None,
            comment_url=None,
            rest={'HttpOnly': None},
            rfc2109=False,)

        badargs = set(kwargs) - set(result)  ❷
        if badargs:
            err = 'create_cookie() got unexpected keyword arguments: %s'  ❸
            raise TypeError(err % list(badargs))  ❸

        result.update(kwargs) ❹
        result['port_specified'] = bool(result['port'])  ❺
        result['domain_specified'] = bool(result['domain'])
        result['domain_initial_dot'] = result['domain'].startswith('.')
        result['path_specified'] = bool(result['path'])

        return cookielib.Cookie(**result)  ❻
```

❶ **kwargs 允许用户为一个 cookie 提供任意数量的关键字配置项。

❷ 集合运算，符合 Python 风格，简单易读。集合是由标准库提供的类型，对字典使用 set() 会返回键的一个集合。

❸ 此处很好地示范了如何将代码由一个长行切分成两个短行，有助于阅读理解。额外的 err 变量并没有什么坏处。

❹ result.update(kwargs) 使用 kwargs 字典中的键值对更新 result 字典，替换已存在的键值对，如果不存在则创建一个新的。

❺ 此处调用了 bool()，如果对象是真值，则强制地将它转变为 True（意思是演算结果为 True，在当前案例中，如果 result['port'] 不是 None 并且不是一个空容器，那么 bool(result['port']) 的演算结果为 True）。

❻ 初始化 cookielib.Cookie 的签名实际上是 18 个位置参数和一个关键字参数（rfc2109 默认为 False）。对于普通人来说，不可能记得住哪个位置上应该是哪个值，而 Python 可以使用关键字参数基于参数名称为位置参数赋值，这里 Requests 利用了这一特性，向初始化方法传递了整个字典。

状态码（可读性很重要）

整个 status_codes.py 就是创建了一个对象，可以通过对象的属性查找对应状态码。首先展示 status_codes.py 中查找字典的定义，然后展示 sessions.py 中使用了这个查找字典的代码片段。

```
#
#  摘录自 requests/status_codes.py 文件
#

_codes = {

    # Informational.
    100: ('continue',),
    101: ('switching_protocols',),
    102: ('processing',),
    103: ('checkpoint',),
    122: ('uri_too_long', 'request_uri_too_long'),
    200: ('ok', 'okay', 'all_ok', 'all_okay', 'all_good', '\\o/', '✓'),❶
    201: ('created',),
    202: ('accepted',),
    #
    #  省略
    #

    # Redirection.
    300: ('multiple_choices',),
    301: ('moved_permanently', 'moved', '\\o-'),
```

```
        302: ('found',),
        303: ('see_other', 'other'),
        304: ('not_modified',),
        305: ('use_proxy',),
        306: ('switch_proxy',),
        307: ('temporary_redirect', 'temporary_moved', 'temporary'),
        308: ('permanent_redirect',
              'resume_incomplete', 'resume',), # These 2 to be removed in 3.0  ❷

        #
        #  省略其余代码
        #
    }

    codes = LookupDict(name='status_codes')  ❸

    for code, titles in _codes.items():
        for title in titles:
            setattr(codes, title, code)   ❹
            if not title.startswith('\\'):
                setattr(codes, title.upper(), code)  ❺
```

❶ 除高兴人表情符号（\o/）和复选标记（✓）以外，所有的 OK 状态名称都会变成查找字典中的键。

❷ 废弃的值单独在一行，以便删除时在版本控制里看起来比较干净明了。

❸ LookupDict 允许使用点号访问它的元素。

❹ 这样设置属性后，则可以使用键 codes.ok == 200 和 codes.okay == 200。

❺ 这样也可以使用键 codes.OK == 200 和 codes.OKAY == 200。

为状态码做的所有工作都是在构建一个查找字典 codes，将所有的状态码数字都放到同一个文件中，非常容易阅读，而不是在代码中到处硬编码整数，这样非常容易拼写错误。因为代码中使用的所有状态码都源于一个以状态码为键的字典，状态码整数只会存在一次，因此拼写错误的可能性远远小于将一堆全局变量手动嵌到一个命名空间里。

将键转换成属性而不是将它们作为字典里的字符串来使用，再次降低了拼写错误的风险。如下例子取自 sessions.py 文件，从中可以看出单词相比数字阅读起来要轻松得多。

```
    #
    #  取自 sessions.py 文件
    #  截断了代码，只展示相关内容
    #
```

```
from .status_codes import codes  ❶


class SessionRedirectMixin(object):  ❷
    def resolve_redirects(self, resp, req, stream=False, timeout=None,
                          verify=True, cert=None, proxies=None,
                          **adapter_kwargs):
        """Receives a Response. Returns a generator of Responses."""
        i = 0
        hist = [] # keep track of history

        while resp.is_redirect:  ❸
            prepared_request = req.copy()

            if i > 0:
                # Update history and keep track of redirects.
                hist.append(resp)
                new_hist = list(hist)
                resp.history = new_hist
            try:
                resp.content # Consume socket so it can be released
            except (ChunkedEncodingError, ContentDecodingError,
RuntimeError):
                resp.raw.read(decode_content=False)

            if i >= self.max_redirects:
                raise TooManyRedirects(
                        'Exceeded %s redirects.' % self.max_redirects
                )

            # Release the connection back into the pool.
            resp.close()

            #
            #  省略了部分内容
            #

            # http://tools.ietf.org/html/rfc7231#section-6.4.4
            if (resp.status_code == codes.see_other and   ❹
                    method != 'HEAD'):
                method = 'GET'

            # Do what the browsers do, despite standards...
            # First, turn 302s into GETs.
            if resp.status_code == codes.found and method != 'HEAD':  ❺
```

```
            method = 'GET'

        # Second, if a POST is responded to with a 301, turn it into a GET.
        # This bizarre behavior is explained in Issue 1704.
        if resp.status_code == codes.moved and method == 'POST':  ❺
            method = 'GET'

        #
        # ... 还有其他代码 ...
        #
```

❶ 此处导入了状态码查找字典 codes。

❷ 混入类为 Session 核心类提供了重定向方法，Session 类也定义在这个文件中，不过没有一起节选出来。

❸ 此处进入一个循环，跟随重定向来获取我们希望得到的内容。节选的代码段中删除了整个循环逻辑。

❹ 相比难以记忆的整数，状态码作为文本阅读起来要容易得多。codes.see_other 在此处即是 303。

❺ codes.found 即 302，codes.moved 即 301，这类代码是自描述的；我们能从变量名中理解其含义，并且使用了点号而不是字典查询（例如，使用 codes.found 而不是 codes['found']），避免了因拼写错误而弄乱代码。

Werkzeug

为了阅读 Werkzeug，我们需要稍微了解一下 Web 服务器是如何与应用程序通信的。

WSGI 是 Python 中 Web 应用程序与服务器程序之间交互的接口，定义见 PEP 333（Phillip J.Eby 于 2003 年所写）[16]，其规定了 Web 服务器（例如 Apache）如何与 Python 应用程序或框架进行通信。

1. 服务器程序每次接收到 HTTP 请求（例如 GET 或 POST）时都会调用应用程序。
2. 应用程序返回一个可迭代的字节字符串（bytestring），服务器程序以这个字节字符串来响应 HTTP 请求。

16 PEP 3333（*https://www.python.org/dev/peps/pep-3333/*）取代 PEP 333 对规范做了一些更新，包含一些 Python 3 特有的细节。推荐阅读 Ian Bicking 的《WSGI 教程》（*http://pythonpaste.org/do-it-yourself-framework.html*），该教程容易理解而且讲得非常全面。

3. 这个规范还说明应用程序将会接收两个参数，例如 webapp(environ, start_reponse)。environ 参数包含与请求相关联的所有数据，start_reponse 参数则是一个函数或其他可被调用的对象，用于向服务器程序返回响应头部（例如 'Content-type', 'text/plain'）和状态信息（例如 200 OK）。

以上概述省略了不少细节信息。在 PEP 333 的中间有如下这样一段雄心勃勃的内容，论述了在新标准下 Web 框架可能实现模块化。

> 如果中间件既简单又健壮，并且服务器程序和框架广泛采用 WSGI，那么就可能产生一种全新的 Python Web 应用框架：由松耦合的中间件组件构成。甚至已有框架的作者都会选择重构他们的框架，以这种方式来提供功能服务，从而变得更像采用 WSGI 的库，而不那么像单体大框架。如此，应用开发者就可以针对特定功能选择最佳组件，而无须纠结于单个框架的优缺点。
>
> 当然，那一天毫无疑问还相当遥远。不过，在此期间，实现框架与服务器程序之间任意搭配使用，则是一个足够短期的目标。

2007 年 Armin Ronacher 发布了 Werkzeug，目的是为了满足大家的迫切需求——一个可以用于构建 WSGI 应用和中间件组件的 WSGI 库。

Werkzeug 是本章进行源码阅读的项目中最大的一个，因此我们只会重点介绍一些它的设计选择。

阅读一个工具包的代码

软件工具包是相互兼容的工具程序的一个集合。对于 Werkzeug，其中的工具程序都与 WSGI 相关。理解各种工具程序及其用途，最好的方式是看它们的单元测试，这也是我们阅读 Werkzeug 代码的方式。

从 GitHub 上获取 Werkzeug：

```
$ git clone https://github.com/pallets/werkzeug.git
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv)$ cd werkzeug
(venv)$ pip install --editable.
(venv)$ py.test tests # 运行单元测试
```

阅读 Werkzeug 文档

Werkzeug 文档（*http://werkzeug.pocoo.org/*）列出了它所提供的一些主要的东西，对 WSGI 1.0（PEP 333，参见 *https://www.python.org/dev/peps/pep-0333/*）规范的一个实现、

一个URL路由系统、解析和转储HTTP请求的能力、表示HTTP请求及HTTP响应的对象、会话和cookie支持、文件上传，以及其他实用功能和社区插件。此外，还提供一个全功能的调试器。

Werkzeug官方教程写得很好，不过我们将使用API文档来进一步了解这个库的组件。下面的内容取自Werkzeug的包装器文档（*http://werkzeug.pocoo.org/docs/latest/wrappers/*）和路由文档（*http://werkzeug.pocoo.org/docs/latest/routing/*）。

使用 Werkzeug

Werkzeug为WSGI应用程序开发提供了一些实用工具，所以要了解Werkzeug提供了什么，可以先从开发一个WSGI应用程序入手，然后使用一些Werkzeug的实用工具。第一个应用程序取自PEP 333，稍做修改，没使用Werkzeug。第二个程序与第一个用途一样，但使用了Werkzeug：

```
def wsgi_app(environ, start_response):
    headers = [('Content-type', 'text/plain'), ('charset', 'utf-8')]
    start_response('200 OK', headers)
    yield 'Hello world.'

# 这个应用和上面那个做的事情相同
response_app = werkzeug.Response('Hello world!')
```

Werkzeug实现了一个werkzeug.Client类，用于做一次性测试时替代真实的Web服务器程序。这个客户端的响应类型即是response_wrapper参数的类型。在如下代码中，我们创建客户端，并使用它们来调用上面实现的WSGI应用。首先，使用普通的WSGI应用（但是响应也会解析成werkzeug.Response对象）。

```
>>> import werkzeug
>>> client = werkzeug.Client(wsgi_app, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> type(resp)
<class 'werkzeug.wrappers.Response'>
>>> resp.status
'200 OK'
>>> resp.content_type
'text/plain'
>>> print(resp.data.decode())
Hello world.
```

然后，使用werkzeug.Response返回WSGI应用。

```
>>> client = werkzeug.Client(response_app, response_wrapper=werkzeug.
```

```
Response)
>>> resp=client.get("?answer=42")
>>> print(resp.data.decode())
Hello world!
```

werkzeug.Request 类以更易用的形式来提供环境字典的内容（上面代码中传递给 wsgi_app() 的 environ 参数），还提供一个装饰器来转换函数，这个函数获取一个 werkzeug.Request 类型参数并返回一个 erkzeug.Response 类型的结果。

```
>>> @werkzeug.Request.application
... def wsgi_app_using_request(request):
...     msg = "A WSGI app with:\n method: {}\n path: {}\n query: {}\n"
...     return werkzeug.Response(
...         msg.format(request.method, request.path, request.query_string))
...
```

使用时有如下行为：

```
>>> client = werkzeug.Client(
...         wsgi_app_using_request, response_wrapper=werkzeug.Response)
>>> resp=client.get("?answer=42")
>>> print(resp.data.decode())
A WSGI app with:
    method: GET
    path: /
    query: b'answer=42'
```

现在我们知道如何使用 werkzeug.Request 和 werkzeug.Response 对象了。文档重点介绍的另一个特性是路由。如下代码使用了路由特性，其中的编号标识了路由模式和匹配结果。

```
>>> import werkzeug
>>> from werkzeug.routing import Map, Rule
>>>
>>> url_map = Map([     ❶
...     Rule('/', endpoint='index'),      ❷
...     Rule('/<any("Robin","Galahad","Arthur"):person>', endpoint='ask'), ❸
...     Rule('/<other>', endpoint='other')    ❹
... ])
>>> env = werkzeug.create_environ(path='/shouldnt/match')     ❺
>>> urls = url_map.bind_to_environ(env)
>>> urls.match()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "[...path...]/werkzeug/werkzeug/routing.py", line 1569, in match
```

```
    raise NotFound()
werkzeug.exceptions.NotFound: 404: Not Found
```

❶ werkzeug.Routing.Map 类提供了一些核心的路由函数。规则匹配是按顺序完成的，第一条匹配到的规则即是被选中的那个。

❷ 如果规则的占位符字符串中没有尖括号匹配项，则只进行精确匹配，并且 url.match() 返回的第二个结果是一个空字典：

```
        >>> env = werkzeug.create_environ(path='/')
       >>> urls = url_map.bind_to_environ(env)
    >>> urls.match()
('index', {})
```

❸ 否则，第二个结果条目是一个将规则中命名匹配项映射到对应值的字典，例如将 'person' 映射到值 'Galahad'：

```
       >>> env = werkzeug.create_environ(path='/Galahad?favorite+color')
       >>> urls = url_map.bind_to_environ(env)
>>> urls.match()
      ('ask', {'person': 'Galahad'})
```

❹ 注意，Galahad 本可以匹配到名为 other 的路由，但实际并没有，而下面的 Lancelot 匹配到了，因为实际选中的是匹配到模式的第一条规则。

```
>>> env = werkzeug.create_environ(path='/Lancelot')
   >>> urls = url_map.bind_to_environ(env)
   >>> urls.match()
('other', {'other': 'Lancelot'})
```

❺ 如果没有匹配到规则列表中的任何一个，则会抛出一个异常。

```
>>> env = werkzeug.test.create_environ(path='/shouldnt/match')
   >>> urls = url_map.bind_to_environ(env)
      >>> urls.match()
   Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
   File "[...path...]/werkzeug/werkzeug/routing.py", line 1569, in match
raise NotFound()
werkzeug.exceptions.NotFound: 404: Not Found
```

可以使用映射将一个请求路由到合适的处理端点（endpoint）。如下代码接着前面的例子实现了这个逻辑。

```
@werkzeug.Request.application
def send_to_endpoint(request):
```

```
    urls = url_map.bind_to_environ(request)
    try:
        endpoint, kwargs = urls.match()
        if endpoint == 'index':
            response = werkzeug.Response("You got the index.")
        elif endpoint == 'ask':
            questions = dict(
                Galahad='What is your favorite color?',
                Robin='What is the capital of Assyria?',
                Arthur='What is the air-speed velocity of an unladen
swallow?')
            response = werkzeug.Response(questions[kwargs['person']])
        else:
        response = werkzeug.Response("Other: {other}".format(**kwargs))
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        response = werkzeug.Response(
            'You may not have gone where you intended to go,\n'
            'but I think you have ended up where you needed to be.',
            status=404
        )
    return response
```

再次使用 werkzeug.Client 来测试它。

```
>>> client = werkzeug.Client(send_to_endpoint, response_wrapper=werkzeug.
Response)
>>> print(client.get("/").data.decode())
You got the index.
>>>
>>> print(client.get("Arthur").data.decode())
What is the air-speed velocity of an unladen swallow?
>>>
>>> print(client.get("42").data.decode())
Other: 42
>>>
>>> print(client.get("time/lunchtime").data.decode()) # no match
You may not have gone where you intended to go,
but I think you have ended up where you needed to be.
```

阅读 Werkzeug 代码

如果测试覆盖良好，那么通过查看单元测试就能了解一个库的作用。不过要告诫你一点：从单元测试入手，你是在探索原本用来确保代码不会破坏的隐晦的测试用例，而不是查

找模块之间的相互关联。对于 Werkzeug 这样的工具包来说，这样没有问题，因为工具包通常都是由一些模块化的、松耦合的组件构成的。

因为我们已经熟悉路由以及请求和响应包装器是如何工作的，所以现在阅读 werkzeug/test_routing.py 和 werkzeug/test_wrappers.py 对我们来说是不错的选择。

在首次打开 werkzeug/test_routing.py 时，通过在整个文件中搜索导入的对象可以快速地查找模块间的相互关联。如下是所有的导入语句：

```
import pytest                                          ❶

import uuid                                            ❷

from tests import strict_eq                                    ❸
from werkzeug import routing as r                              ❹
from werkzeug.wrappers import Response                         ❺
from werkzeug.datastructures import ImmutableDict,             ❻
from werkzeug.test import create_environ                       ❼
```

❶ 当然，此处 pytest 是用来做测试的。

❷ uuid 模块只在 test_uuid_converter() 函数中用到，用于确认从字符串到一个 uuid.UUID 对象（全局唯一标识符字符串可以唯一性地标识互联网上的对象）的转换如期工作。

❸ 定义在 werkzeug/tests/__init__.py 中的 strict_eq() 函数经常用于做测试。只因为在 Python 2 里，Unicode 字符串和字节字符串之间会隐式地进行类型转换，但在 Python 3 中不会，所以需要这个函数。

❹ werkzeug.routing 是被测试的模块。

❺ Reponse 对象只在 test_dispatch() 函数中用到，用于确认 werkzeug.routing.MapAdapter.dispatch() 是否将正确的信息传递给 WSGI 应用。

❻ 这两个字典对象每个只使用了一次，ImmutableDict 用于确认 werkzeug.routing.Map 中一个不可修改的字典确实是不可修改的，MultiDict 则用于为 URL 构建器提供多个键相同的值，并确认构建器仍然会构建出正确的 URL。

❼ create_environ() 函数用于测试，无须使用一个真正 HTTP 请求就能创建一个 WSGI 环境（请求信息）。

进行快速搜索的要点是快速查看模块间的相互关联。如此我们发现仅有 werkzeug.routing 导入了一些特殊的数据结构。其余单元测试则表明了 routing 模块的适用范围。例如，可

以使用非 ASCII 字符。

```
def test_environ_nonascii_pathinfo():
    environ = create_environ(u'/лошадь')
    m = r.Map([
        r.Rule(u'/', endpoint='index'),
        r.Rule(u'/лошадь', endpoint='horse')
    ])
    a = m.bind_to_environ(environ)
    strict_eq(a.match(u'/'), ('index', {}))
    strict_eq(a.match(u'/лошадь'), ('horse', {}))
    pytest.raises(r.NotFound, a.match, u'/барсук')
```

有些测试用例是针对构建和解析 URL 的，甚至是针对一些工具代码的（例如，在不存在实际匹配项时找到最接近的可用匹配项）。我们在处理路径和 URL 字符串类型转换 / 解析时，甚至可以做各种自定义处理。

```
def test_converter_with_tuples():
    '''
    Regression test for https://github.com/pallets/werkzeug/issues/709
    '''
    class TwoValueConverter(r.BaseConverter):

        def __init__(self, *args, **kwargs):
            super(TwoValueConverter, self).__init__(*args, **kwargs)
            self.regex = r'(\w\w+)/(\w\w+)'

        def to_python(self, two_values):
            one, two = two_values.split('/')
            return one, two

        def to_url(self, values):
            return "%s/%s" % (values[0], values[1])

    map = r.Map([
        r.Rule('/<two:foo>/', endpoint='handler')
    ], converters={'two': TwoValueConverter})
    a = map.bind('example.org', '/')
    route, kwargs = a.match('/qwert/yuiop/')
    assert kwargs['foo'] == ('qwert', 'yuiop')
```

werkzeug/test_wrappers.py 也没有导入许多模块。通读其中的测试代码可以大致了解 Request 对象的可用功能覆盖的方面：cookie、编码、认证、安全、缓存超时及多语言编码。

```
def test_modified_url_encoding():
    class ModifiedRequest(wrappers.Request):
        url_charset = 'euc-kr'

    req = ModifiedRequest.from_values(u'/?foo=정상처리'.encode('euc-kr'))
    strict_eq(req.args['foo'], u'정상처리')
```

总而言之，阅读项目的测试代码提供了一种方式来仔细查看目标库都提供了什么。一旦了解了 Werkzeug 是什么，就可以接着往下读。

Werkzeug 中对 Tox 的使用

Tox 是一个使用虚拟环境来运行测试用例的 Python 命令行工具。只要你正在使用的 Python 解释器已安装好，就可以在电脑上运行它（在命令行中运行 Tox）。它已经和 GitHub 集成，因此如果在项目的顶级路径中有一个 tox.ini 文件，例如 Werkzeug 项目，那么每次提交代码时，Tox 会自动运行测试。

如下是 Werkzeug tox.ini 配置文件的全部内容。

```
[tox]
envlist = py{26,27,py,33,34,35}-normal, py{26,27,33,34,35}-uwsgi

[testenv]
passenv = LANG
deps=
# General
    pyopenssl
    greenlet
    pytest
    pytest-xprocess
    redis
    requests
    watchdog
    uwsgi: uwsgi
# Python 2
    py26: python-memcached
    py27: python-memcached
    pypy: python-memcached
# Python 3
    py33: python3-memcached
    py34: python3-memcached
    py35: python3-memcached
```

```
    whitelist_externals=
        redis-server
        memcached
        uwsgi

    commands=
        normal: py.test []
        uwsgi: uwsgi
            --pyrun {envbindir}/py.test
               --pyargv -kUWSGI --cache2=name=werkzeugtest,items=20 –
    master
```

取自 Werkzeug 的风格示例

第 4 章介绍的大多数核心风格要点前面都已经论述过了。本节选择的第一个风格示例展示了一种从一个字符串猜测数据类型的优雅方法，第二个示例建议在定义一个长正则表达式时使用 VERBOSE 参数选项，这样其他人无须花时间思考就能明白这个正则表达式的作用。

猜测数据类型的优雅方法（如果实现易于解释，则可能是个好思路）

我们中的大多数人肯定都曾解析过文本文件并将内容转换成各种数据类型，如下方案特别符合 Python 风格。

```
_PYTHON_CONSTANTS = {
    'None': None,
    'True': True,
    'False': False
}


def _pythonize(value):
    if value in _PYTHON_CONSTANTS: ❶
        return _PYTHON_CONSTANTS[value]
    for convert in int, float:  ❷
        try: ❸
            return convert(value)
        except ValueError:
            pass
    if value[:1] == value[-1:] and value[0] in '"\'': ❹
        value = value[1:-1]
    return text_type(value) ❺
```

❶ 对 Python 字典进行键查找会使用哈希映射，就像集合查找一样。Python 没有 switch case 语句（PEP 3103 曾提议添加，但因不够普及而被否决）。Python 用户使用 if/elif/else 或字典查找方案来代替 switch case 语句。

❷ 注意，在尝试转换成 float 类型之前，先尝试转换成限制性更大的类型 int。

❸ 使用 try/except 语句来推断类型也非常符合 Python 风格。

❹ 这部分是必要的，因为代码位于 werkzeug/routing.py 文件中，被解析的字符串是 URL 的一部分。这部分代码用于检查字符串两头是否存在引号和反引号。

❺ text_type 以 Python 2 和 Python 3 都兼容的方式将字符串转换成 Unicode 编码。它基本上和 HowDol 一节中介绍的 u() 函数一样。

正则表达式（可读性很重要）

如果在代码中使用冗长的正则表达式，则请使用 re.VERBOSE[17] 参数选项，这样其他人更容易理解这个正则表达式，如下代码片段选自 werkzeug/routing.py。

```
import re

_rule_re = re.compile(r'''
    (?P<static>[^<]*)                       # static rule data
    <
    (?:
    (?P<converter>[a-zA-Z_][a-zA-Z0-9_]*)   # converter name
    (?:\((?P<args>.*?)\))?                  # converter arguments
    \:                                      # variable delimiter
    )?
    (?P<variable>[a-zA-Z_][a-zA-Z0-9_]*)    # variable name
    >
''', re.VERBOSE)
```

取自 Werkzeug 的结构示例

前两个例子展示了符合 Python 风格的动态类型利用方式。动态类型一节告诫读者不要为一个变量多次赋不同类型的值，但尚未提过动态类型的优点。动态类型的一个优点是无论对象是什么类型，只要其行为符合预期，就可以使用，即所谓的鸭子类型。鸭子类型以如下观点来描述数据类型：如果它看着像鸭子[18]并且叫起来也像鸭子，那么它就是一只鸭子。

17 re.VERBOSE 会改变空白字符的处理方式，并且允许编写注释，这样开发者可以写出可读性更好的正则表达式。请阅读 re 模块文档了解更多详情。

18 如果对象可被调用，或可被迭代，或者定义了正确的方法。

这两个例子以不同的方式演示了对象不是函数也可以被调用的情况：cached_property.__init__() 让一个类实例在初始化后就能够以类似普通函数调用的方式来使用它，而 Response.__call__() 方法是让 Response 实例自己调用起来像个函数。

最后一段代码借助 Werkzeug 中一些混合类的实现（每个混合类都定义了 Werkzeug Request 对象功能的一个子集）来讨论为什么它们是一个绝妙的主意。

基于类的装饰器（一种符合 Python 风格的动态类型使用方式）

Werkzeug 利用鸭子类型来实现 @cached_property 装饰器。在介绍 Tablib 项目时讨论过 property，我们讨论它就像在讨论一个函数。装饰器通常都是函数，但是因为没有类型上的强制性，所以装饰器可以是任意可调用的对象。property 实际上是一个类（因为其类名并没有如 PEP 8 要求的那样以大写字母开头，所以可以断定其本意就是要像函数一样使用）。在像函数调用一样实例化 property 类（property()）时，property.__init__() 会被调用来做初始化操作并返回一个 property 实例。一个类，如果其 __init__() 方法恰当定义，那么使用起来就可以像一个可调用的对象。

下面摘录的代码包含 cached_property 的完整定义，它继承自 property 类。cached_property 定义中的文档已做了自我描述。如果将它用于装饰 BaseRequest.form，那么 instance.form（译注：这里的 instance 代表 BaseRequest/Request 类的实例）将具有 cached_property 类型，并且因为其定义了 __get__() 和 __set__() 方法，在用户看来其行为就像一个字典。第一次访问 BaseRequest.form 时，它会读取一次表单数据（如果存在），然后将数据存储在 instance.form.__dict__ 中以备后用。

```
class cached_property(property):

    """A decorator that converts a function into a lazy property. The
    function wrapped is called the first time to retrieve the result,
    and then that calculated result is used the next time you access
    the value::

        class Foo(object):

            @cached_property
            def foo(self):
                # calculate something important here
                return 42

    The class has to have a '__dict__' in order for this property to
    work.
    """
```

```
    # implementation detail: A subclass of Python's built-in property
    # decorator, we override __get__ to check for a cached value. If one
    # choses to invoke __get__ by hand, the property will still work as
    # expected because the lookup logic is replicated in __get__ for
    # manual invocation.

    def __init__(self, func, name=None, doc=None):
        self.__name__ = name or func.__name__
        self.__module__ = func.__module__
        self.__doc__ = doc or func.__doc__
        self.func = func

    def __set__(self, obj, value):
        obj.__dict__[self.__name__] = value

    def __get__(self, obj, type=None):
        if obj is None:
            return self
        value = obj.__dict__.get(self.__name__, _missing)
        if value is _missing:
            value = self.func(obj)
            obj.__dict__[self.__name__] = value
        return value
```

如下是实战应用：

```
>>> from werkzeug.utils import cached_property
>>>
>>> class Foo(object):
...     @cached_property
...     def foo(self):
...         print("You have just called Foo.foo()!")
...         return 42
...
>>> bar = Foo()
>>>
>>> bar.foo
You have just called Foo.foo()!
42
>>> bar.foo
42
>>> bar.foo # 注意此处不再有 print 输出
42
```

Response.__call__

Response 类就是将一些特性混入 BaseResponse 类，就像 Request 那样。我们将重点介绍其用户接口，不会展示其实际代码，仅通过解说 BaseResponse 的文档字符串来解释用法细节。

```
class BaseResponse(object):

    """Base response class. The most important fact about a response object
    is that it's a regular WSGI application. It's initialized with a couple
    of response parameters (headers, body, status code, etc.) and will start
a
    valid WSGI response when called with the environ and start response
    callable.

    Because it's a WSGI application itself, processing usually ends before
the
    actual response is sent to the server. This helps debugging systems
    because they can catch all the exceptions before responses are started.

    Here is a small example WSGI application that takes advantage of the
    response objects::

        from werkzeug.wrappers import BaseResponse as Response

        def index(): ❶
            return Response('Index page')

        def application(environ, start_response):  ❷
            path = environ.get('PATH_INFO') or '/'
            if path == '/':
                response = index()  ❸
            else:
                response = Response('Not Found', status=404)  ❹
            return response(environ, start_response) ❺
    """
    #  省略其余部分
```

❶ 在文档字符串的示例中，index() 是用来响应 HTTP 请求的函数。响应将会是字符串 Index page。

❷ 这是对一个 WSGI 应用的签名要求，正如 PEP 333/PEP 3333 中的规定。

❸ 因为 Response 继承自 BaseResponse，所以 Response 是 BaseResponse 的一个实例。

❹ 可以看到 404 响应仅要求设置 status 关键字参数。

❺ Response 实例自身是可调用的，并且将所有相关的响应头部和一些细节参数都设置成合理的默认值（或者在路径不是“/”时覆盖一些默认值）。

那么，类的实例怎样才是可调用的？ Response 实例可调用是因为 BaseResponse 类中定义了 __call__ 方法。如下代码示例中展示了该方法。

```
class BaseResponse(object):
    #
    #  省略了所有其他代码
    #

    def __call__(self, environ, start_response):  ❶
        """Process this response as WSGI application.

        :param environ: the WSGI environment.
        :param start_response: the response callable provided by the WSGI
        server.
        :return: an application iterator
        """
        app_iter, status, headers = self.get_wsgi_response(environ)
        start_response(status, headers)   ❷
        return app_iter   ❸
```

❶ 正是这个方法签名让 BaseResponse 实例变成可调用的。

❷ 这里调用了 start_response 函数来满足规范对 WSGI 应用的要求。

❸ 此处返回一个字节迭代序列。

从这里我们学到：如果一个方案在语言层面是可能的，那么为什么不用呢？在意识到可以为任何对象增加一个 __call__() 方法，从而将其变成可调用的，我们应该认真阅读 Python 数据模型的原始文档（*https://docs.python.org/3/reference/datamodel.html*）。

混入类（也是一个绝妙的主意）

Python 中的混入类是指专门用于向一个类添加特定功能（一组相关属性）的类。与 Java 不同，Python 允许多重继承。这意味着：支持同时继承自多个不同的类的范式，可以模块化地将不同的功能封装到不同的类中。

像这样的模块化，在 Werkzeug 工具库中大有用处，因为这样就是在告诉用户哪些函数是相关的哪些是不相关的。开发者可以确信一个混入类中的属性不会被另一个混入类中的任何函数修改。

在 Python 中，混入类没有任何特殊的标识，但按照约定一般会在混入类名称的最后添加一个 Mixin。这意味着：如果不想在方法解析顺序上浪费精力，那么所有混入类的方法都应当具有不同的名称。

在 Werkzeug 中，某些混入类中的某些方法会要求存在某些属性。这样的要求通常在混入类的文档字符串中会有说明。

```
    # ... in werkzeug/wrappers.py


    class UserAgentMixin(object):  ❶

        """Adds a 'user_agent' attribute to the request object which contains
        the parsed user agent of the browser that triggered the request as a
        :class:'~werkzeug.useragents.UserAgent' object.
        """

        @cached_property
        def user_agent(self):
            """The current user agent."""
            from werkzeug.useragents import UserAgent
            return UserAgent(self.environ)  ❷

    class Request(BaseRequest, AcceptMixin, ETagRequestMixin,
                  UserAgentMixin, AuthorizationMixin,      ❸
                  CommonRequestDescriptorsMixin):

        """Full featured request object implementing the following mixins:

        - :class:'AcceptMixin' for accept header parsing
        - :class:'ETagRequestMixin' for etag and cache control handling
        - :class:'UserAgentMixin' for user agent introspection
        - :class:'AuthorizationMixin' for http auth handling
        - :class:'CommonRequestDescriptorsMixin' for common headers
        """
        ❹
```

❶ UserAgentMixin 继承自 object，虽然在 Python 3 中默认如此，不过为了兼容 Python 2，强烈推荐这样显式地声明，因为明确胜过隐晦。

❷ UserAgentMixin.user_agent 假设存在一个 self.environ 属性。

❸ 如果 UserAgentMixin 包含在 Request 继承的基类列表中，那么它提供的属性就可以通过 Request(environ).user_agent 来访问。

❹ 没有任何其他代码，这就是 Request 类定义的全部。所有功能通过基类或混入类来提供。它具有模块化、可插拔的优点，和福特 • 大老爷一样棒（译注：福特 • 大老爷原文为 Ford Prefect，是科幻小说《银河系漫游指南》中一个角色的名字，而这个角色名又引自福特汽车公司的著名高端车型系列）。

新式类和对象

object 基类添加了一些其他内置特性所依赖的默认属性。不是继承自 object 的类被称为老式类或经典类，老式类在 Python 3 中已移除。Python 3 中默认都是继承自 object 基类，这意味着所有的 Python 3 类都是新式类。Python 2.7 中可以使用新式类（自 Python 2.3 起，新式类就具备现在的行为了），不过必须显式地声明新式类的继承关系，并且不管在 Python 2 中还是 Python 3 中都应该始终显式地写出来，避免兼容性问题。

欲知更多详情，推荐阅读 Python 官方的新式类文档（*https://www.python.org/doc/newstyle/*），以及 *http://www.python-course.eu/classes_and_type.php* 和 *http://tinyurl.com/history-new-style-classes* 两个网站，通过它们可以了解新式类技术的创造史。

新式类与老式类的区别如下所示（特指在 Python 2.7 中的区别；在 Python 3 中所有类都是新式的）。

```
>>> class A(object):
...     """New-style class, subclassing object."""
...
>>> class B:
...     """Old-style class."""
...
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__',
'__getattribute__', '__hash__', '__init__', '__module__', '__
new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__
sizeof__',
'__str__', '__subclasshook__', '__weakref__']
>>>
```

```
>>> dir(B)
['__doc__', '__module__']
>>>
>>> type(A)
<type 'type'>
>>> type(B)
<type 'classobj'>
>>>
>>> import sys
>>> sys.getsizeof(A()) # 大小单位为字节
64
>>> sys.getsizeof(B())
72
```

Flask

Flask 是一个整合了 Werkzeug 和 Jinja2 的 Web 微型框架（Werkzeug 和 Jinja2 的创始人都是 Armin Ronacher），它于 2010 年愚人节发布，很快成了 Python 最受欢迎的 Web 框架之一。2007 年 Armin Ronacher 发布了 Werkzeug，号称 Python Web 开发的瑞士军刀，但没多少人采用。Werkzeug 的思路是将 WSGI 与其他任何东西解耦，这样开发者就可以使用自己选择的实用中间件程序。他不知道大家有多欣赏 rails 这样的框架[19]。

阅读一个框架的代码

软件框架就像物理框架一样，提供构建 WSGI[20] 应用的底层结构：库的使用者为 Flask 应用程序主体提供待运行的组件。阅读框架源码的目标是理解框架结构及明确框架提供了哪些特性。

从 GitHub 获取 Flask 代码。

```
$ git clone https://github.com/pallets/flask.git
$ virtualenv venv    # Python 3 环境下也可用，但现在还不推荐
$ source venv/bin/activate
(venv)$ cd flask
(venv)$ pip install --editable .
(venv)$ pip install -r test-requirements.txt # 单元测试需要
```

19 这里 rails 指代 Ruby on Rails，一个非常流行的 Web 框架，它与包罗万象的 Django 风格，而不是几乎啥都不包含（除非你自己添加插件）的 Flask 风格，更相近。如果 Django 能满足所有需求，那么它是一个非常好的选择。Django 原本就是为搭建在线新闻站点而设计的，因此非常适合用来搭建类似站点服务。

20 WSGI 是一个 Python 标准，定义在 PEP 333 和 PEP 3333 中，规定了应用程序如何与 Web 服务器程序相互通信。

```
(venv)$ py.test tests # 运行单元测试
```

阅读 Flask 文档

Flask 的在线文档（*http://flask.pocoo.org/*）一开始就以 7 行代码实现了一个 Web 应用程序，然后对 Flask 进行概述：它是一个基于 Unicode 的 WSGI 兼容的框架，该框架使用 Jinja2 进行 HTML 模板渲染，使用 Werkzeug 来提供 WSGI 实用工具（如：URL 路由等），也内置了一些便于开发测试的工具。因为文档中也包含了详细教程，所以进一步的学习会比较轻松。

使用 Flask

从 GitHub 下载的 Flask 源码中包含一个 flaskr 示例项目，我们可以运行它，它实现的是一个小型博客站点。在 flask 顶级目录下进行如下操作：

```
(venv)$ cd examples/flaskr/
(venv)$ py.test test_flaskr.py  # 所有测试用例都应通过
(venv)$ export FLASK_APP=flaskr
(venv)$ flask initdb
(venv)$ flask run
```

阅读 Flask 代码

因为 Flask 的目标是创建 Web 应用程序，所以它与 Diamond 和 HowDoI 这样的命令行应用程序大不相同。这一次我们不是绘制一张流程图来追踪代码中的函数调用路径，而是通过运行 flaskr 示例应用，使用一个调试器来单步调试 Flask，我们将使用 pdb（标准库中提供的 Python 调试器）。

首先，在 flaskr.py 中添加一个断点，在代码运行到这个点时就会激活，触发交互式会话进入调试器。

```
@app.route('/')
def show_entries():
    import pdb; pdb.set_trace() ## 本行设置断点
    db = get_db()
    cur = db.execute('select title, text from entries order by id desc')
    entries = cur.fetchall()
    return render_template('show_entries.html', entries=entries)
```

然后，关闭文件并在命令行中执行 python 进入一个交互式会话。我们不会真的启动一个服务，而是使用 Flask 的内部测试工具来模拟一个 HTTP GET 请求访问路径，正是在这个路径的处理流程中我们放置了调试器。

```
>>> import flaskr
>>> client = flaskr.app.test_client()
>>> client.get('/')
> /[... 这里截断了路径 ...]/flask/examples/flaskr/flaskr.py(74)show_entries()
-> db = get_db()
(Pdb)
```

最后三行来自 pdb：我们可以看到暂停位置所在文件路径（flaskr.py 文件的绝对路径）、行号（74），以及方法名（show_entries()）。-> db = get_db 这一行显示的是在调试器中往前一步将会执行的语句。另外，提示符（pdb）提醒我们正在使用 pdb 调试器。

我们可以通过在命令提示符后输入 u 或者 d 分别向上或向下查看调用[21]。可以在 pdb 文档的调试器命令一节（*https://docs.python.org/3/library/pdb.html*）查看完整的命令列表。输入变量名也可以查看变量值，也可以输入任何其他 Python 命令，我们甚至可以在继续执行代码之前将变量设置为不同的值。

如果对栈向上回溯一步，那么将看到是谁调用了 show_entries() 函数（函数内包含我们刚刚设置的断点）：它是一个 flask.app.Flask 对象，其中包含一个名为 view_funcions 的查找字典，该字典将字符串名称（例如 show_entries）映射到函数。我们也看到 show_entries() 函数是用 **req.view_args 参数来调用的。我们可以在交互式调试器命令行输入 req.view_args 查看其值（一个空字典 - {}，意味着没有参数）。

```
(Pdb) u
> /[ ... 这里截断了路径 ...]/flask/flask/app.py(1610)dispatch_request()
-> return self.view_functions[rule.endpoint](**req.view_args)
(Pdb) type(self)
<class 'flask.app.Flask'>
(Pdb) type(self.view_functions)
<type 'dict'>
(Pdb) self.view_functions
{'add_entry': <function add_entry at 0x108198230>,
'show_entries': <function show_entries at 0x1081981b8>, [... truncated ...]
'login': <function login at 0x1081982a8>}
(Pdb) rule.endpoint
'show_entries'
(Pdb) req.view_args
{}
```

如果需要，那么可以同时用多个会话追踪代码，打开相应文件，进入上面声明的断点行。

21 Python 调用栈包含当前 Python 解释器正在执行的指令。因此，如果函数 f() 调用了函数 g()，那么函数 f() 会先入栈，待 g() 被调用时则会入栈压到 f() 上面。当 g() 返回时，它从栈中弹出（即从栈中移除），f() 则会从原来中断之处继续执行。之所以被称为栈，是因为从概念上讲，它的工作原理和洗碗机洗盘子一样，新的盘子会放到最上面，而洗碗机总是先处理最上面的盘子。

如果继续向上回溯调用栈，那么可以看到 WSGI 应用是在哪调用的。

```
(Pdb) u
> /[ ... 这里截断了路径 ...]/flask/flask/app.py(1624)full_dispatch_request()
-> rv = self.dispatch_request()
(Pdb) u
> /[ ... 这里截断了路径 ...]/flask/flask/app.py(1973)wsgi_app()
-> response = self.full_dispatch_request()
(Pdb) u
> /[ ... 这里截断了路径 ...]/flask/flask/app.py(1985)__call__()
-> return self.wsgi_app(environ, start_response)
```

如果再次输入 u，就会在测试模块中结束，测试模块就是用来创建模拟客户端的模块，我们已经在调用栈上回溯到顶了。从这个过程中我们了解到：请求是在一个 flask.app.Flask 类实例中，flask/flask/app.py 中第 1985 行的那个函数分发到 flaskr 应用，如下是该函数：

```
class Flask:
    ## ~~ ... 省略了大量定义的内容 ...

    def wsgi_app(self, environ, start_response):
        """The actual WSGI application. ... 省略了其他文档 ...
        """
        ctx = self.request_context(environ)
        ctx.push()
        error = None
        try:
            try:
                response = self.full_dispatch_request()  ❶
            except Exception as e:
                error = e
                response = self.make_response(self.handle_exception(e))
            return response(environ, start_response)
        finally:
            if self.should_ignore_error(error):
                error = None
            ctx.auto_pop(error)

        def __call__(self, environ, start_response):
            """Shortcut for :attr:'wsgi_app'."""
            return self.wsgi_app(environ, start_response)  ❷
```

❶ 这是第 1793 行，上面调试器会话中有标明。

❷ 这是第 1986 行，上面调试器会话中有标明。WSGI 服务器程序会接受这个 Flask 类

的实例作为一个应用，并在每次接收到 HTTP 请求时调用一次。通过使用调试器，我们发现了代码的这个运行入口点。

我们使用调试器的方式和阅读 HowDoI 时使用调用流程图的方式相同，即追踪函数调用，直接通读代码也是如此方式。使用调试器的优势是可以避免被一些不必要的代码分散注意力或迷惑。注意，使用对你来说最高效的方式就好。

在使用 u 向上追溯调用栈后，我们可以使用 d 向下追溯调用栈，最终会在断点处结束，输出标记 *** Newest frame。

```
> /[ ... truncated path ...]/flask/examples/flaskr/flaskr.py(74)show_entries()
-> db = get_db()
(Pdb) d
*** Newest frame
```

从那开始，我们可以使用 n（next）命令来推进一个函数调用，或使用 s（step）命令来单步向前运行程序。

```
(Pdb) s
--Call--
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(55)get_db()
-> def get_db():
(Pdb) s
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(59)get_db()
-> if not hasattr(g, 'sqlite_db'):                    ❶
##~~
##~~ ... 执行了许多步骤来创建并返回数据库连接 ...
##~~
-> return g.sqlite_db
(Pdb) n
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(75)show_entries()
-> cur = db.execute('select title, text from entries order by id desc')
(Pdb) n
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(76)show_entries()
-> entries = cur.fetchall()
(Pdb) n
> /[ ... truncated path ... ]/flask/examples/flaskr/flaskr.py(77)show_entries()
-> return render_template('show_entries.html', entries=entries)          ❷
(Pdb) n
--Return--
```

还有很多，但是一一演示则会显得有些乏味。我们从中可以了解到：

❶ 注意到其中涉及一个 Flask.g 对象。它是全局（实际上是局限在 Flask 实例中的）上下文。它的存在是为了存放数据库连接和其他一些持久性数据（例如，cookie），这

类持久性数据需要在 Flask 类中方法的生命周期之外继续存在。像这样使用一个字典在 Flask 应用命名空间之外保持变量值，可以避免可能发生的命名冲突。

❷ 函数 render_template() 位于 flaskr.py 模块中函数定义的末尾，意味着核心应用逻辑已运行结束。该函数的返回值会一路返回到 Flask 实例中调用的那个函数上。其余部分我们就略过不提了。

对于检查代码中的局部位置，精确查明用户选择的固定断点前后发生的事情，调试器大有帮助。调试器的一大特性是可以即时改变变量值（调试器中任何 Python 代码都正常执行）然后继续运行代码。

Flask 中的日志记录

Diamond 默认配置一个日志记录器，示例了在应用中应该如何进行日志记录，而 Flask 是在库中提供一个日志记录模块。如果你想要的仅是避免“未发现日志记录器”的警告，则可以在 Request 库里面搜索 logging（结果在 requests/requests/__init__.py 中）。但是如果想要为库或者框架提供一些日志记录支持，Flask 中的日志记录则是一个值得学习的好例子。

Flask 特有的日志记录方式实现可以参见 flask/flask/logging.py 文件。它为生产运行（以 ERROR 级别记录日志）和开发调试（以 DEBUG 级别记录日志）定义了对应的日志记录格式字符串，并且遵循 Twelve-Factor App（*http://12factor.Net/*）提出的建议：将日志记录到输出流（输出流指向 wsgi.errors 的其中一个或者 sys.stderr，具体取决于上下文环境）。

如下所示，flask/flask/app.py 将日志记录器添加到核心 Flask 应用上（代码片段省略了文件中其他不相干的代码）。

```
    # a lock used for logger initialization
    _logger_lock = Lock()  ❶


    class Flask(_PackageBoundObject):

        ##~~ ... 省略其他定义

        #: The name of the logger to use. By default the logger name
    is the
        #: package name passed to the constructor.
        #:
```

```
    #: .. versionadded:: 0.4
    logger_name = ConfigAttribute( 'LOGGER_NAME' ) ❷
    def __init__(self, import_name, static_path=None, static_url_
path=None,
                 ##~~ ... 省略其余的参数 ...
                 root_path=None):
        ##~~ ... 省略其余的初始化操作
        # Prepare the deferred setup of the logger.
        self._logger = None ❸
        self.logger_name = self.import_name

    @property
    def logger(self):
        """A :class:'logging.Logger' object for this application. The
        default configuration is to log to stderr if the application
is
         in debug mode. This logger can be used to (surprise) log
messages.
        Here some examples::

            app.logger.debug('A value for debugging')
            app.logger.warning('A warning occurred (%d apples)', 42)
            app.logger.error('An error occurred')

        .. versionadded:: 0.3
        """
           if self._logger and self._logger.name == self.logger_
name:
            return self._logger  ❹
        with _logger_lock:  ❺
            if self._logger and self._logger.name == self.logger_
name:
                return self._logger
            from flask.logging import create_logger
            self._logger = rv = create_logger(self)
            return rv
```

❶ 这个锁会在代码片段的最后使用。锁是在同一时间只能被一个线程控制的对象。在它被使用时，任何其他需要它的线程都必须阻塞直到锁被释放。

❷ 和 Diamond 一样，Flask 使用配置文件来设置日志记录器的名称（也会默认进行合理的设置，这里没有展示出来，因此用户可以什么都不管就能得到一个合理的日志记录设置）。

❸ Flask 应用的日志记录器初始设置为 None，以便可以在之后创建。

❹ 如果日志记录器已存在，则返回它。像本章之前所述，property 装饰器用于防止用户不小心修改了日志记录器。

❺ 如果日志记录器还不存在（初始设置为 None），那就使用标记 ❶ 中创建的锁，并创建日志记录器。

取自 Flask 的风格示例

第 4 章提出的风格大多数都已经举例讨论过了，因此对于 Flask，我们只讨论一个风格示例——优雅又简单的 Flask 路由装饰器及其实现。

Flask 路由装饰器（优美胜于丑陋）

Flask 中的路由装饰器是将 URL 路由添加到目标函数上，如下所示。

```
@app.route('/')
def index():
    pass
```

在分发一个请求时，Flask 应用会使用 URL 路由来找到正确的函数，生成响应结果。装饰器语法将路由代码逻辑分离在目标函数之外，保持函数结构扁平（译注：这里的扁平是指如果把路由代码逻辑放到目标函数之外，不可避免地需要添加一些条件判断语句，导致函数缩进层次更多，代码更长），使用起来也直观。

但也不是必须使用它，它的存在只是为用户提供 API 特性的。如下是其代码实现，在 flask/flask/app.py 文件内 Flask 主类定义的一个方法。

```
class Flask(_PackageBoundObject):  ❶
    """The flask object implements a WSGI application ...
        ... 省略了文档字符串中的其他内容 ...
    """
    ##~~ ... 忽略其他部分，只剩 route() 方法

    def route(self, rule, **options):
        """A decorator that is used to register a view function for a
        given URL rule. This does the same thing as :meth:'add_url_rule'
        but is intended for decorator usage::

            @app.route('/')
            def index():
```

```
            return 'Hello World'

        #... 省略了其余的文档字符串 ...
        """
        def decorator(f):  ❷
            endpoint = options.pop('endpoint', None)
            self.add_url_rule(rule, endpoint, f, **options)  ❸
            return f
        return decorator
```

❶ 基于各种内容文件的路径配置，_PackageBoundObject 会设置文件目录结构以便后续导入 HTML 模板、静态文件和其他内容。内容文件的路径都是以相对于应用模块（例如 app.py 文件）的相对路径进行配置的。

❷ 直接命名为 decorator，名副其实。

❸ 这才是实际将 URL 添加到映射表（包含所有路由规则）的函数。Flask.route 唯一的目的是为库的用户提供一个便捷的装饰器。

取自 Flask 的结构示例

取自 Flask 的两个结构示例，主题都是模块化。Flask 的结构有意设计成非常易于扩展和修改的，从 JSON 字符串的编码解码方式（Flask 在标准库提供的 JSON 处理能力之外补充了 datetime 和 UUID 对象的编码解码能力）到 URL 路由匹配时使用的类。

应用特定的默认值（简单胜于复杂）

Flask 和 Werkzeug 都有一个 wrappers.py 模块。其目的是为了给 Flask 添加恰当的默认值。Flask 是一个 Web 应用开发框架，其底层依赖是 Werkzeug 专为 WSGI 应用开发而设计的更通用的工具库。Flask 对 Werkzeug 的 Request 和 Response 对象进行继承，增加了一些 Web 应用程序相关的特性。例如，flask/flask/wrappers.py 中的 Response 对象如下所示。

```
from werkzeug.wrappers import Request as RequestBase, Response as
ResponseBase
##~~ ... 省略了其他内容 ...

class Response(ResponseBase):   ❶
    """The response object that is used by default in Flask. Works like the
    response object from Werkzeug but is set to have an HTML mimetype by
    default. Quite often you don't have to create this object yourself
because
    :meth:'~flask.Flask.make_response' will take care of that for you.
    If you want to replace the response object used you can subclass this and
    set :attr:'~flask.Flask.response_class' to your subclass.  ❷
```

```
    """
    default_mimetype = 'text/html'  ❸
```

❶ Werkzeug 的 Response 类导入为 ResponseBase，这是一个不错的风格细节，使得导入类的角色更明显，还允许新的 Response 子类以父类的原名来命名。

❷ 文档字符串中突出地说明了开发者可以继承 flask.wrappers.Response 并替换之，以及如何实现。如果项目中实现了这样的特性，切记要在文档中进行说明，否则用户都不知道这个特性的存在。

❸ 这是 Response 类中唯一的变化。Request 类的变化更多一些，为控制本章的篇幅，在这里就不进行展示了。

下面这个简短的交互式会话展示了 Flask 和 Werkzeug 的 Response 类之间都有哪些改变。

```
>>> import werkzeug
>>> import flask
>>>
>>> werkzeug.wrappers.Response.default_mimetype
'text/plain'
>>> flask.wrappers.Response.default_mimetype
'text/html'
>>> r1 = werkzeug.wrappers.Response('hello', mimetype='text/html')
>>> r1.mimetype
u'text/html'
>>> r1.default_mimetype
'text/plain'
>>> r1 = werkzeug.wrappers.Response('hello')
>>> r1.mimetype
'text/plain'
```

改变 mimetype 默认值的目的仅仅是让 Flask 用户在构建包含 HTML 的响应对象（Flask 期望的用法）时可以少输入一些字符。对于一般用户而言，明智的默认值使得代码更易于使用。

明智的默认值很重要

有时默认值的重要性不仅仅在于易于使用。例如，Flask 默认将会话流程和安全通信的密钥设定为 Null。当密钥为空时，如果应用程序尝试启动一个安全会话，则会抛出错误。强制抛出这个错误意味着用户将不得不使用他们自己的密钥，其他的处理方式要么是默默地允许使用一个空的会话密钥和不安全的会话流程，要么是提供一个类似 mysecretkey 的默认密钥，许多人通常不会去改变默认值（因此在生产部署中也使用了默认值）。

模块化（也是一个绝妙的主意）

flask.wrappers.Response 的文档字符串让用户了解到可以继承 Response 对象并在 Flask 主对象中使用新定义的类。

如下代码摘自 flask/flask/app.py，我们借此介绍一下 Flask 中的另一些模块化特点。

```
class Flask(_PackageBoundObject):
    """ ... 省略了文档字符串 ...
    """
    #: The class that is used for request objects. See :class:'~flask.Request'
    #: for more information.
    request_class = Request  ❶

    #: The class that is used for response objects. See
    #: :class:'~flask.Response' for more information.
    response_class = Response  ❷

    #: The class that is used for the Jinja environment.
    #:
    #: .. versionadded:: 0.11
    jinja_environment = Environment ❸

    ##~~ ... 省略了一些其他定义 ...

    url_rule_class = Rule
    test_client_class = None
    session_interface = SecureCookieSessionInterface()

    ##~~ .. 省略了其他内容 ..   ❹
```

❶ 这个属性值可以用自定义 Request 类来取代。

❷ 这个属性值可以设置成自定义 Response 类。这些是 Flask 类的类属性（而不是实例属性），并且以一种清晰的方式进行命名，使其用途更明显。

❸ Environment 类是 Jinja2 Environment 类的一个子类，能够理解 Flask 蓝图，这样就能够构建更大的多文件 Flask 应用。

❹ 还有其他一些模块化配置项没有展示，因为那样会显得重复。

如果你搜遍 Flask 类的定义，就能找到这些类都是在哪实例化在哪使用的。虽然这些类定义并非必须暴露给用户，但是 Flask 明确地做了这个模块化结构选择，以此方便库用户更好地控制 Flask 的行为。

当人们讨论 Flask 的模块化时，他们不只是在说可以使用任何想要的数据库后端，而且是在讨论 Flask 提供的这种插入并使用不同类的能力。

现在你已经见过一些写得很好且非常符合 Python 之禅的代码了。

我们强烈建议将这里讨论过的每个程序的代码都完整地看一遍，因为成为一个优秀程序员的最佳途径就是阅读高质量的代码。每当写代码遇到困难时，那就使用源码吧，卢克！（译注：这句话改编自《星球大战》中的“使用原力，卢克！”，体现了程序员式幽默。）

第 6 章

交付高质量的代码

本章重点介绍打包和分发 Python 代码方面的最佳实践。开发者要么是想创建一个 Python 库提供给其他开发者使用，要么是想创建一个独立的应用程序供他人使用，像 pytest 那样。

Python 打包相关的生态系统在过去的几年间变得简单易用得多了，这归功于 Python 打包技术权威组织（PyPA）[1] 的付出。该组织的成员负责维护 pip、Python 包索引（PyPI）及大量 Python 打包相关的基础设施。因为他们编写打包技术文档绝对一流，所以打包你的代码一节介绍的轮子我们没必要重造，不过我们会简要介绍从私有站点托管包的两种方法，并且讨论如何将代码上传到 Anaconda.org（一个类似 PyPI 的商业化包索引平台，由 Continuum Analytics 公司运营）上。

通过 PyPI 或其他包仓库发布代码的缺点是安装者必须理解如何安装要求的 Python 版本，还能够并且乐意去使用 pip 这类工具去安装代码的其他依赖。如果你是把代码分发给其他开发者，这样是没有问题的，但如果是将应用程序发布给非程序员的最终用户，这个方法就不太适合了。对此，可以使用冻结你的代码一节中介绍的工具。

那些为 Linux 系统制作 Python 包的开发者可能还会考虑构建一个 Linux 发行版安装包（例如，Debian/Ubuntu 上的 .deb 文件。Python 官方文档中称之为构建发行版）。这一方式的维护工作量很大，不过 Linux 已构建分发包的打包技术一节还是为你提供了一些可选方案。这个方式类似代码冻结，但是却可以从包里面移除 Python 解释器。

可执行的 ZIP 文件一节将分享一个强大的技巧：如果代码的 ZIP 压缩包带有一个特定的头部，则可以直接执行这个 ZIP 文件。如果你知道目标受众已安装了 Python 解释器，并且你的项目是纯 Python 代码，那么这是一个不错的方案。

1 他们更喜欢幽默地将其称为“安装事务内阁（Ministry of Installation）”。Nick Coghlan，在打包技术相关 PEP 方面代表着 BDFL，几年前在他的博客上写过一篇深度文章，介绍了整个组织系统、历史渊源及前进方向。

有用的词汇和概念

在 PyPA 组织形成之前，Python 社区实际上不存在任何一种明显的打包方式（从 *http://stackoverflow.com/questions/6344076* 上的历史讨论可以看出这点）。如下是本章讨论中最重要的一些词汇，PyPA 术语表（*https://packaging.python.org/en/latest/glossary/*）里有更多的词汇定义。

依赖

> Python 包会在 requirements.txt 文件（可用于测试或应用部署）中列出其 Python 依赖库，或者在 setup.py 文件中调用 setuptools.setup() 时提供 install_require 参数中列出其 Python 依赖库。
>
> 某些项目可能还有其他依赖，例如，Postgres 数据库、C 编译器或者 C 语言库共享对象。Python 包可能不会显式声明这些依赖，但如果缺失将会中断构建。如果你要构建这样的库，那么 Paul Kehrer 的《分发要求编译的模块》可能对你有所帮助。

已构建的分发包

> Python 包的一种发布格式（还有可选的其他资源和元数据），是一种安装后即可运行的分发包形式，无须进一步编译。

Egg

> Egg 是一种已构建的分发格式。它是一种特定结构的 ZIP 文件，包含安装要求的元数据。它由 Setuptools 库引入，曾经多年都是事实上的标准分发形式，但从来都不是 Python 官方的打包格式。自 PEP 247 发布后，Egg 已被 Wheel 所代替。在 Python 打包用户手册中可以读到 Wheel 和 Egg 格式之间的所有不同。

Wheel

> Wheel 也是一种已构建的分发格式，目前是已构建 Python 库分发包的标准。这种分发包是内含某些元数据的 ZIP 文件，pip 可以利用这些元数据来安装或卸载包。通常，这种文件的扩展名是 .whl，并遵从一种特定的命名约定，以此特意告知用户此分发包应用于哪个平台、是哪个构建版本、使用哪个解释器。

除了要求安装 Python，纯 Python 语言写成的常规 Python 包，只要求可以从 PyPI（或者 WareHouse，即将到来的 PyPI 新地址）上下载其他 Python 库。如果 Python 库依赖于 Python 以外的东西（例如，C 库或者系统可执行程序），那么困难就来了（在第 2 章中我们已经尝试以额外的安装步骤来克服这个困难）。如果分发过程遇到的复杂问题连

Wheel 格式都无法处理，那么 Buildout 和 Conda 这种工具应该能派上用场。

打包你的代码

为分发而打包代码，即创建必要的文件结构、添加要求的问题并定义恰当的变量以满足，遵照相关的 PEP 和 Python 打包指南[2] 中打包并分发项目教程（*https://packaging.python.org/en/latest/distributing/*）描述的当前最佳实践，或者遵照其他包仓库（*http://anaconda.org/*）的打包要求。

> **包、分发包与安装包**
>
> 我们用“包”这个词表示了这么多事物，可能会令人迷惑。分发包包含（常规的 Python）软件包、模块和定义一次发布需要的其他文件。有时我们会把库称为“安装包”，包含一整个库的顶级包目录。简单的“包”一词，意指任意包含 __init__py 文件和其他模块（*.py 文件）的目录。PyPA 维护了一个打包相关术语的术语表，参见 *http://anaconda.org/*。

Conda

即使你已经安装 Anaconda 的 Python 二次发行版，也可以使用 pip 和 PyPI，不过默认包管理器是 Conda，并且默认的包仓库地址是 *http://anaconda.org/*。我们建议按照 *http://bit.ly/building-conda* 上的教程来构建包，该教程最后介绍了如何将分发包上传到 Anaconda.org。

大量学术界、商业产品开发及使用 Windows 系统的开发者都选择使用 Anaconda 来获取易用的二进制分发包，如果你正在为科学或统计学应用开发一个库，那么即使你自己不用 Anaconda，为了吸引他人使用你的库，你应该也想为库构建一个 Anaconda 分发包。

PyPI

PyPI 和 pip 这类工具的生态系统建立得非常好，其他开发者因为临时性实验或者开发大型专业性系统，下载安装你开发的包，非常轻松方便。

2 当前似乎存在 *https://python-packaging-user-guide.readthedocs.org* 和 *https://packaging.python.org/* 两个网址可以访问到相同的内容。

如果你正在编写一个开源 Python 模块，那么 PyPI 正是最佳的包托管平台[3]。如果你的代码不是打包放在 PyPI 上的，那么其他开发者会更难找到它并用在他们的项目中。或者他们会怀疑你的项目管理不善、还没有正式发布或已废弃。

关于 Python 打包技术，最新且正确的权威信息来源是 PyPA 维护的 Python 打包指南（*https://packaging.python.org/en/latest/*）。

测试使用 testPyPI，正式使用 PyPI

如果只是想测试打包设置，或教某人使用 PyPI，则可以使用 testPyPI，并且在推送一个正式版本到 PyPI 之前记得运行单元测试。和使用 PyPI 一样，在每次推送一个新文件之前必须改变版本号。

示例项目

PyPA 的示例项目演示了当前打包一个 Python 项目的最佳实践。setup.py 模块中的注释给出了建议，并且标识了相关 PEP 控制的配置选项。整个文件按要求组织结构，并附带了大量注释，帮助理解每个文件的目的，以及应当包含什么内容。

项目的 README 文件包含链接指向打包指南（*https://packaging.python.org/*）和关于打包分发的一个教程（*https://packaging.python.org/en/latest/distributing.html*）。

使用 pip，而不是 easy_install

关于 Python 库分发、打包及安装的经典方法，社区中有值得关注的困惑（*http://stackoverflow.com/questions/6344076*）和讨论（*http://stackoverflow.com/questions/3220404*），自 2011 年起，PyPA 做了大量工作来解决这些问题。在 PEP 453（*https://www.python.org/dev/peps/pep-0453/*）中选择了 pip 作为 Python 的默认包安装器，Python 3.4（首次发布于 2014 年）及之后的版本都会默认安装 pip[4]。

pip 和 esay_instal 有许多不同的用途，并且一些旧系统可能仍然需要 easy_install。PyPA 提供了对比 pip 和 esay_install 的表格，参见 *http://packaging.python.org/en/latest/pip_easy_install/*。

做开发时使用 pip install --editable 来安装项目代码，就能持续编辑代码而不用一次次重复安装。

3 PyPI 正处于逐步切换到 Warehouse 系统的过程中。Warehouse 现在处于评测阶段。据我们所知，他们还正在调整 UI，但不会调整 API。PyPA 开发者之一是 Nicole Harris，她写过一篇关于 Warehouse 的简介（*http://whoisnicoleharris.com/warehouse/*）。

4 如果你安装了 Python 3.4 或更高版本，但没有 pip，那么可以在命令行中使用 python -m ensurepip 进行安装。

个人的 PyPI 服务

如果想从 PyPI 之外的来源安装包（例如：一个内部的工作服务器，存储公司专有软件包或被安全和法律团队检查通过的软件包），那么可以搭建一个简单的 HTTP 服务器来实现，在包含安装包的目录下运行服务。

例如，安装一个名为 MyPackage.tar.gz 的包，目录结构如下所示。

```
.
|--- archive/
    |--- MyPackage/
        |--- MyPackage.tar.gz
```

可以在 shell 中输入如下命令，从 archive 目录下运行一个 HTTP 服务器。

```
$ cd archive
$ python3 -m SimpleHTTPServer 9000
```

这会在 9000 端口上运行一个简单的 HTTP 服务器，服务会列出所有的包（当前例子中是 MyPackage）。你可以使用任何一个 Python 包安装器来安装 MyPackage。在命令行里使用 pip 命令，你需要这样写：

```
$ pip install --extra-index-url=http://127.0.0.1:9000/ MyPackage
```

这里存在一个和包名一样的文件夹，如果你觉得 MyPackage/MyPackage.tar.gz 结构比较多余，则可以将软件包放在这个目录外面，并使用直接路径进行安装。

```
$ pip install http://127.0.0.1:9000/MyPackage.tar.gz
```

pypiserver

pypiserver 是一个极简的 PyPI 兼容的服务器，可用于为 easy_install 或 pip 提供安装包服务。它包含一些辅助特性，例如管理命令（-U），它将所有包更新到 PyPI 上可以找到的最新版本。

托管在 S3 上的 PyPI 服务

搭建个人的 PyPI 服务器，另一个方案是把安装包托管在亚马逊的简单存储服务（Amazon S3）上。首先你必须拥有一个亚马逊 Web 服务（AWS）账号，该账号还要有一个 S3 存储桶（bucket）。确保遵从存储桶的命名规则，S3 允许创建违反命名规则的存储桶，但无法访问它。要使用存储桶，先要创建一个虚拟环境，并且从 PyPI 或其他源安装项目的所有依赖包，再安装 pip2pi。

```
$ pip install git+https://github.com/wolever/pip2pi.git
```

按照 pip2pi README 文件来使用 pip2pi 和 dir2pi 命令，可以这样做：

```
$ pip2tgz packages/ YourPackage+
```

或者使用如下两个命令：

```
$ pip2tgz packages/ -r requirements.txt
$ dir2pi packages/
```

现在，上传文件。使用 Cyberduck 之类的客户端将整个 packages 目录同步到你的 S3 存储桶。确保上传了 packages/simple/index.html 及所有新的文件和目录。

在默认情况下，上传到 S3 存储桶的新文件只有用户本身有访问权限。如果在尝试安装一个包之时遇到 HTTP 403 错误码，那么请确保已经正确地设置了权限：使用亚马逊 Web 控制台将文件的 READ 权限设置给 EVERYONE。现在团队将能够使用如下命令来安装你的包。

```
$ pip install \
  --index-url=http://your-s3-bucket/packages/simple/ \
  YourPackage+
```

pip 对 VCS 的支持

可以使用 pip 直接从版本控制系统获取代码。为了做到这点，请遵照这些说明。这是搭建个人 PyPI 服务的另一个可选方案。使用 pip 安装一个 GitHub 项目的示例命令如下所示：

```
$ pip install git+git://git.myproject.org/MyProject#egg=MyProject
```

其中，Egg 不一定是一个 Egg 分发包，它是项目中你想要安装的目录的名字。

冻结你的代码

冻结你的代码是指创建一个可独立运行的程序包，最终用户不需要在他们的电脑上安装 Python 就能运行这个程序包，它同时包含了应用程序代码和 Python 解释器。

像 Dropbox、星战前夜（Eve Online）、文明 4（Civilization IV）及 BitTorrent 客户端这些主要使用 Python 编写的应用程序，均是作为独立运行的程序包进行分发。

以这种方式进行分发的优势是应用安装后即能正常运行，即使用户并没有安装要求的（或任意的）Python 版本。在 Windows 上，甚至在许多 Linux 发行版和 Mac OS X 上，可能都还没安装与应用匹配的 Python 版本。此外，应该始终以某种可执行格式将软件分发给

最终用户。名称以 .py 结尾的文件是给软件工程师或系统管理员使用的。

代码冻结的一个缺点是发布包的大小会增大 2MB~12MB。而且，在 Python 新增安全漏洞补丁之时，也由你负责为应用发布更新版本。

当使用 C 库时，请检查许可证

你应当针对所有操作系统检查整个项目依赖树上每个包的使用许可。不过在此我们想特别强调一下 Windows 系统，因为所有 Windows 版本都需要在目标机器上安装 MS Visual C++ 动态链接库。你可能没有权限来重新发布特定的库，因此必须在发布应用前检查你的许可证权限（更多信息参见 *http://bit.ly/visual-cplusplus* 上微软关于 Visual C++ 文件的法律公报）。你也可以选择使用 MinGW 编译器（MinGW，Minimalist GNU for Windows），因为这是个 GNU 项目，所以使用许可又可能会被反向限制（必须开源而且免费）。

另外，因为 MinGW 和 Visual C++ 编译器并不完全相同，所以在项目更换到另一个编译器时，应当检查单元测试是否仍然按照期望运行。如果你不在 Windows 上频繁编译 C 代码，那么就忽略这些。注意，在 MinGW 环境编译使用 NumPy 仍然会有一些问题（*https://github.com/numpy/numpy/issues/5479*）。NumPy 官方的 wiki 上有一篇文章推荐使用自带静态工具链的 MinGW 构建版本（*https://github.com/numpy/numpy/wiki/Mingw-static-toolchain*）。

表 6-1 中比较了一些流行的冻结工具。它们都与 Python 标准库里的 distuils 模块配合工作。因为它们不能做跨平台的交叉冻结 [5]，所以必须在各个目标平台上分别构建。

在表 6-1 中，PyInstaller 和 cx_Freeze 适用于所有平台，py2app 仅能在 Mac OS X 上工作，py2exe 仅能在 Windows 上工作。bbFreeze 可以在类 UNIX 和 Windows 系统上工作，但不支持 Mac OS X，并且还没有移植到 Python 3。不过，它可以生成 egg 格式的分发包，对于一些遗留的老系统，可能需要这个能力。

表6-1　冻结工具

	pyInstaller	cx_Freeze	py2app	py2exe	bbFreeze
Python 3	支持	支持	支持	支持	
许可证	经过修改的 GPL	经过修改的 PSF	MIT	MIT	Zlib

5　PyInstall 1.4 尝试过在 Linux 上将 Python 代码冻结打包成一个 Windows 可执行程序，但是在 PyInstall 1.5 版本中放弃了，因为这一特性在非纯 Python 编写的程序上无法正常工作（因此，无法冻结打包 GUI 应用）。

续表

	pyInstaller	cx_Freeze	py2app	py2exe	bbFreeze
Windows	支持	支持	—	支持	支持
Linux	支持	支持	—	—	支持
Mac OS X	支持	支持	支持	—	—
Egg 分发包格式	支持	支持	支持	—	支持
pkg_resourcesa 支持 [a]	—	—	支持	—	支持
单文件模式 [b]	支持	—	—	支持	—

a pkg_resources 是与 Setuptools 捆绑在一起的一个单独的模块，可用于动态查找依赖。当冻结代码时，这是一个挑战，因为很难发现从静态代码中动态加载的依赖。例如，PyInstaller 只声称当依赖分析基于 Egg 文件时可以正常工作。

b 单文件模式是指在 Windows 上将应用程序及其所有依赖捆绑打包成一个可执行文件的方案选项。InnoSetup 和 NullSoft 脚本安装系统（NSIS）都是创建安装器的流行工具，可以将代码打包成单个 .exe 文件。

PyInstaller

PyInstaller 可用于为 Mac OS X、Windows 和 Linux 系统创建应用程序。其主要目标是开箱即用、兼容第三方包，这样代码冻结均能正常工作[6]。PyIntsllaer 为支持的包维护了一个列表，其中支持的图形库包括 Pillow、pygame、PyOpenGL、PyGTK、PyQT4、PYQT5、PySide（QT 插件除外）及 wxPython，支持的科学工具包括 NumPy、Matplotlib 和 SciPy。

PyInstaller 使用一个经过修改的 GPL 许可证（*https://github.com/pyinstaller/pyinstaller/wiki/License*），由于有一个需要特殊考虑的情况：允许任何人使用 PyInstaller 构建分发非免费的程序（包括商业程序），因此你必须遵从的许可证取决于代码开发使用的库。他们团队甚至为那些制作商业程序或者想防止别人修改代码的人介绍了如何隐藏源代码。但是，如果为了隐藏源码，你要修改依赖库的源码来构建你的应用，那么请一定要读一下许可证（如果这一点很重要，则咨询一下律师，如果不太重要，则可以参考 *https://tldrlegal.com/*），因为也许会要求你共享代码变更。

PyInstaller 手册详略得当。记得查看 PyInstaller 依赖要求页面来确认你的系统是否兼容。

6 下文讨论其他安装器时将看到：代码冻结的挑战不仅在于为一个 Python 库的特定版本查找并捆绑兼容的 C 库，还在于发现相关的配置文件、精灵图片或特殊图像及其他文件，通过分析源码，冻结工具没法发现这些文件。

对于Windows系统，需要XP及之后的新版本；对于Linux系统，需要若干终端应用程序（文档里面列出了可以在哪里找到它们）；对于 Mac OS X 系统，需要 10.7（Lion）及之后的新版本。在 Linux 或 Mac OS X 系统上可以使用 Wine（一个 Windows 模拟器）来交叉编译 Windows 程序。

在构建应用的虚拟环境中，使用 pip 来安装 PyInstaller ：

```
$ pip install pyinstaller
```

从名为 script.py 的模块创建一个标准的可执行程序，使用 ：

```
$ pyinstaller script.py
```

若要创建一个带窗口的 MacMac OS X 或 Windows 应用程序，则在命令行里使用 --windowed 选项 ：

```
$ pyinstaller --windowed script.spec
```

这样会在执行 pyinstaller 命令的目录中创建两个新的文件夹和一个文件。

- 一个 .spec 文件，PyInstaller 可以再次运行它来重新构建。
- 一个 build 文件夹，其中保存一些日志文件。
- 一个 dist 文件夹，其中保存着主执行文件及其依赖的一些 Python 库。

因为 PyInstaller 会将应用程序用到的所有 Python 库都放入 dist 文件夹，所以在分发可执行文件时，需要分发整个 dist 文件夹。

可以编辑 script.spec 实现自定义构建，其提供配置选项。

- 将数据文件与可执行文件捆绑打包在一起。
- 包含 PyInstaller 不能自动推断出来的运行时库（.dll 或 .so 文件）。
- 向可执行文件添加 Python 运行时选项。

这一特性大有用处，可以将文件保存到版本控制中，这样未来的构建会更加简单。PyInstaller 的 wiki 页面包含一些常见应用的构建方法（*https://github.com/pyinstaller/pyinstaller/wiki/Recipes*），包括 Django、PyQt4 及针对 Windows 和 Mac OS X 的代码签名。它是 PyInstaller 目前最新的快速教程集。现在，编辑过的 script.spec 文件可以作为 pyinstaller 的一个参数来运行（而不再是使用 script.py）。

```
$ pyinstaller script.spec
```

在为 PyInstaller 传递一个 .spec 文件时，它会从文件内容里获取所有配置选项，并忽略命令行选项，除非命令行选项是 --upx-dir=、--distpath=、--workpath=、--noconfirm 及 --ascii。

cx_Freeze

和 PyInstaller 一样，cx_Freeze 可以在 Linux、Mac OS X 和 Windows 系统上冻结 Python 项目。然而，cx_Freeze 团队不建议使用 Wine 环境来编译 Windows 应用程序，因为这种方式需要手动复制一些文件才能让应用程序正常工作。使用 pip 安装 cx_Freeze。

```
$ pip install cx_Freeze
```

制作可执行程序的最简单方法是从命令行中运行 cxfreeze，但是如果写一个 setup.py 脚本，则可以使用更多的配置选项（而且可以使用版本控制）。setup.py 脚本和 Python 标准库中 disutils 模块使用的相同。cx_freeze 扩展了 disutils，提供额外的命令（并修改了另一些命令）。可以在命令行、设置脚本或者一个 setup.cfg 配置文件中提供这些配置选项。

cxfreeze-quickstart 脚本会创建一个基本的 setup.py 文件，可以修改该文件并提交版本控制方便未来构建。如下这个示例会话是为脚本 hello.py 构建可执行程序。

```
$ cxfreeze-quickstart
Project name: hello_world
Version [1.0]:
Description: "This application says hello."
Python file to make executable from: hello.py
Executable file name [hello]:
(C)onsole application, (G)UI application, or (S)ervice [C]:
Save setup script to [setup.py]:

Setup script written to setup.py; run it as:
    python setup.py build
Run this now [n]?
```

现在我们有了一个设置脚本 setup.py，并且可以修改它来满足应用的需要。cx_freeze 文档的 distutils setup sctipts 部分（*https://cx-freeze.readthedocs.org/en/latest/distutils.html*）中提供了 setup.py 文件中可以使用的配置选项。在 cx_freeze 源码（*https://bitbucket.org/anthony_tuininga/cx_freeze/src*）的 samples/ 目录（从项目顶层目录开始路径为 cx_Freeze/cx_Freeze/samples/）下还有一些 setup.py 示例脚本和最简化的应用演示了如何冻结使用了 PyQT4、Tkinter、wsPython、Malplotlib、Zope 或其他库的应用。已安装好的 cx_free 库中也捆绑了自己的源码，可以输入如下命令获取其安装路径。

```
$ python -c 'import cx_Freeze; print(cx_Freeze.__path__[0])'
```

一旦编辑完 setup.py 文件，就可以通过如下命令来构建应用程序。

```
$ python setup.py build_exe     ❶
$ python setup.py bdist_msi     ❷
$ python setup.py bdist_rpm     ❸
$ python setup.py bdist_mac     ❹
$ python setup.py bdist_dmg     ❺
```

❶ 这是构建命令行可执行程序的命令选项。

❷ 这个命令选项是 cx_Freeze 在 distutils 原有命令的基础上修改而成的，也是用于处理 Windows 可执行程序及其依赖的。

❸ 这个命令选项对 distutils 原有命令进行了修改，以确保基于适合当前平台的架构创建 Linux 包。

❹ 这个命令选项会创建一个独立的带窗口的 Mac OS X 应用程序包（.app），其中包含依赖和可执行程序。

❺ 这个命令选项也会创建一个 .app 应用程序包，然后将其打包到一个 DMG 磁盘镜像中。

py2app

py2app 专为 Mac OS X 系统构建可执行程序。和 cx_Freeze 一样，它对 disutils 进行扩展，增加了新命令 py2app。使用 pip 来安装它：

```
$ pip install py2app
```

使用 py2applet 命令自动生成一个 setup.py 脚本：

```
$ py2applet --make-setup hello.py
Wrote setup.py
```

这会生成一个基本的 setup.py 文件，可以按需要进行修改。py2app 源码中包含一些示例，每个示例都包含最简化却可工作的代码及对应 setup.py 脚本，这些示例使用的库包括 PyObjC、PyOpenGL、pygame、PySide、PyQT、Tkinter 和 wxPython。从项目顶层目录进到 py2app/examples 目录可以找到它们。

使用 py2pp 命令执行 setup.py 会创建两个目录：build 和 dist。在重新构建之前确保清理这两个目录，命令如下：

```
$ rm -rf build dis
```

```
$ python setup.py py2app
```

更多文档请查阅 py2app 教程（*https://py2app.readthedocs.io/en/latest/tutorial.html*）。构建过程可能遇到一个 AttributeError 异常而退出。如果真的这样，那么请阅读 py2app 使用教程（*http://bit.ly/py2app-tutorial*）中关于 _scan_code 和 _load_module 的部分。

py2exe

py2exe（*https://pypi.python.org/pypi/py2exe*）专为 Windows 系统构建可执行程序。它非常流行，Windows 版本的 BitTorrent 客户端就是使用 py2exe 制作的。像 cx_Freeze 和 py2app 一样，它也扩展了 disutils，这一次增加的是 py2exe 命令。如果需要在 Python 2 环境下使用它，请从 sourceforge（*https://sourceforge.net/projects/py2exe/*）下载老版本。否则，对于 Python 3.3，使用 pip ：

```
$ pip install py2exe
```

py2exe 官方教程（*http://www.py2exe.org/index.cgi/Tutorial*）写得非常棒（使用 wiki 风格而不是源码控制方式维护的文档似乎都很好）。其最基本的 setup.py 文件如下所示。

```
from distutils.core import setup
import py2exe

setup(
    windows=[{'script': 'hello.py'}],
)
```

官方文档中列出了 py2exe 的所有配置选项（*http://www.py2exe.org/index.cgi/ListOfOptions*），并且详尽说明了如何（可选的）包含图标及如何创建一个单文件的可执行程序（*http://www.py2exe.org/index.cgi/SingleFileExecutable*）。能否随同代码分发 Microsoft Visual C++ 运行时的动态库，取决于你持有的 Microsoft Visual C++ 许可证。如果可以，则阅读官网的文档，其中有介绍如何随 .exe 文件一起发布 Microsoft Visual C++DLL（*http://www.py2exe.org/index.cgi/Tutorial#Step52*）；否则，你可以为你的应用用户提供一种方式来下载安装 Microsoft Visual C++ 2008 可再发行组件包（*http://bit.ly/ms-visual-08*）或 Microsoft Visual C++ 2010 可再发行组件包（如果使用 Python 3.3 或更新的版本，参见 *http://bit.ly/ms-visual-10*）。

修改好设置文件 setup.py，可以输入如下命令在 dist 目录下来生成 .exe 文件。

```
$ python setup.py py2exe
```

bbFreeze

bbFreeze 库目前无人维护，并且尚未移植到 Python 3，但它仍然被频繁下载。像 cx_Freeze、py2app 和 py2exe 一样，它对 distutils 进行扩展，增加了 bbfreeze 命令。事实上，旧版本的 bbFreeze 就是基于 cx_Freeze 实现的。有些人在维护遗留系统或者想要打包构建 egg 格式的发布包以便用于基础设施，此处内容正是为了满足他们的需求。使用 pip 安装 bbFreeze：

```
$ pip install bbfreeze  # bbFreeze 不能在 Python 3 下工作
```

其文档比较缺乏，但针对 Flup、Django、Twisted、Metplotlib、GTK、Tkinter 及其他项目编写了构建套路的代码（*https://github.com/schmir/bbfreeze/blob/master/bbfreeze/recipes.py*）。使用 bbfreeze 命令来制作可执行二进制文件。

```
$ bbfreeze hello.py
```

它会在 bbfreeze 运行的路径下创建一个 dist 目录，其中包含一个 Python 解释器和一个与脚本同名的可执行文件（在这个例子里是 hello.py）。

使用新的 disutils 命令来生成 egg 格式分发包：

```
$ python setup.py bdist_bbfreeze
```

还有其他一些命令选项，比如快照版本构建或每日构建。使用标准的 --help 选项可以获取更多的用法信息：

```
$ python setup.py bdist_bbfreeze --help
```

若想精细化配置调整，则可以使用 bbfreeze.Freezer 类，这是首选的 bbfreeze 使用方式。它提供了一些标志属性，可以用来设置是否在创建的 ZIP 文件中使用压缩、是否包含 Python 解释器或者需要包含哪些脚本。

Linux 已构建分发包的打包技术

创建 Linux 已构建分发包可能是在 Linux 上分发代码的一个正确方式：一个已构建分发包就像一个冻结的包，但它不包含 Python 解释器，所以相比使用代码冻结，下载和安装都要小 2MB 左右[7]。并且，如果一个 Linux 发行版发布了一个新的 Python 安全更新，你的应用程序会自动使用 Python 新版本。

7 某些人可能听说过这种分发包被称为二进制包（binary package）或者安装器（installer）；Python 官方将其命名为已构建分发包（built distribution），涵盖 RPM 包、Debian 软件包及 Windows 二进制程序安装器。wheel 格式也是一类已构建分发包，由于《wheel 分发格式微议》（*http://bit.ly/python-on-wheels*）中提到的多种原因，制作平台专有的 Linux 软件分发包更好。

Python 标准库中 distutils 模块提供的 bist_rpm 命令使得生成 rpm 文件（RedHat 和 SUSE 等 Linux 发行版使用的软件包格式）更方便。

关于 Linux 发行版软件分发包的警告

为每种软件分发包格式（例如 *.deb 之于 Debian/Ubuntu、*.rpm 之于 Red Hat/Fedora 等）创建和维护要求的不同配置，工作量相当庞大。如果你的应用代码计划分发到 Linux 之外的其他平台，那么你必须创建和维护 Windows 和 Mac OS X 冻结代码各自所需的配置。简单地为冻结你的代码一节中描述的工具创建和维护单份配置文件，工作量要小得多，基于一份配置就可以分别为所有 Linux 发行版、Windows 和 Mac OS X 生成一个独立的可执行程序。

如果你的代码依赖于当前 Linux 发行版并不支持的 Python 版本，那么创建 Linux 发行版软件分发包也是有问题的。这样必须告诉某些 Ubuntu 版本的最终用户在安装你的 .deb 文件之前，需要使用 sudo apt-repository 命令增加 dead-snakes PPA，这将造成不愉快的用户体验。不仅如此，你还需要为每个 Linux 发行版维护一份对应的说明，更糟糕的是，还需要让你的用户阅读、理解并按照它来实际操作。

上面说了这么多，下面是针对一些流行 Linux 发行版的 Python 打包说明文件。

- Fedora（*https://fedoraproject.org/wiki/Packaging:Python*）。
- Debian 和 Ubuntu（*http://bit.ly/debian-and-ubuntu*）。
- Arch（*https://wiki.archlinux.org/index.php/Python_Package_Guidelines*）。

如果你想要一种更快的方式来为各类 Linux 发行版打包代码，也许应该尝试一下工具包管理（fpm）。它以 Ruby 和 shell 写成，可以将多种源类型（包括 Python）的代码打包成目标平台格式，包括 Debian (.deb)、RedHat (.rpm)、MacMac OS X(.pkg)、Solaris 等。这是一个不错的方法，但不提供依赖树，所以包维护者可能会不赞成使用它。Debian 用户可以尝试 Alien，它可以将 Debian、RedHat、Stampede（.slp）及 Slackware（.tgz*）软件包文件格式相互转换的一个 Perl 程序，但是该项目的代码从 2014 年开始就停止更新了，而且维护者也已退出。

关于实际工作中如何在 Debian 系统上部署服务器应用，Rob McQueen 发表过一些深刻见解（*https://nylas.com/blog/packaging-deploying-python*），如果感兴趣，那么可以读一读。

可执行的 ZIP 文件

从 Python 2.6 开始，Python 能够直接执行包含一个 __main__.py 文件的 ZIP 文件。这是打包纯 Python 应用程序（应用不依赖于一些平台特定的二进制文件）的一个绝佳方式。输入如下所示的单个文件 __main__.py。

```
if __name__ == '__main__':
    try:
        print 'ping!'
    except SyntaxError: # Python 3
        print('ping!')
```

并在命令行输入如下命令，创建一个包含它的 ZIP 文件。

```
$ zip machine.zip __main__.py
```

那么你可以将 ZIP 文件发送给其他人，只要他们安装了 Python 环境，就能像下面这样在命令行里直接执行它。

```
$ python machine.zip
ping!
```

或者如果你想要把它变成一个可执行文件，则可以在 ZIP 文件内容的起始加入一行 POSIX 的 shebang(#!)，ZIP 文件格式允许这样做。你现在就有了一个自包含的应用程序（通过 shebang 里指定的路径访问系统提供的 Python）。如下示例继续上面的代码。

```
$ echo '#!/usr/bin/env python' > machine
$ cat machine.zip >> machine
$ chmod u+x machine
```

现在它是一个可执行文件了。

```
$ ./machine
ping!
```

从 Python 3.5 开始，Python 标准库中新增了一个 zipapp 模块，使用这个模块来创建这类 ZIP 文件也变得更方便，并且它更加灵活，主文件也不再需要命名为 __main__.py。

如果你将依赖库的源码放到项目当前目录进行管理，并调整相关的 import 语句，则可以制作一个包含所有依赖的 ZIP 可执行文件。所以，如果目录结构看上去像这样：

```
.
|--- archive/
    |--- __main__.py
```

并且当前处于一个虚拟环境中，其中只安装了依赖，则可以在 shell 里输入如下命令来包含你的依赖。

```
$ cd archive
$ pip freeze | xargs pip install --target=packages
$ touch packages/__init__.py
```

xargs 命令从标准输入中获取 pip freeze 的运行结果，并将结果转换成 pip 的参数列表，--target=packages 选项指定将依赖安装到一个新文件夹 packages 中。touch 命令在目标文件不存在时会创建这个文件，否则，它会把文件的时间戳更新为当前时间。现在目录结构看起来像下面这样。

```
.
|--- archive/
    |--- __main__.py
    |--- packages/
        |--- __init__.py
        |--- dependency_one/
        |--- dependency_two/
```

如果你这样做了，那么请确保也改变相关的 import 语句来使用刚刚创建的 packages 目录。

```
#import dependency_one # 不要这句
import packages.dependency_one as dependency_one
```

使用 zip -r 命令在新 ZIP 文件中递归包含所有目录，像这样：

```
$ cd archive
$ zip machine.zip -r *
$ echo '#!/usr/bin/env python' > machine
$ cat machine.zip >> machine
$ chmod ug+x machine
```

第 3 部分

场景化指南

读到这里相信你已经安装了 Python、选择了一个编辑器、知道 Pythonic 意味着什么、阅读了一些优秀的 Python 代码，也能够将代码分享给世界上的其他人。第 3 部分将针对特定的编码场景，分享社区中最常见的方法，帮助你选择 Python 库用于项目中。

第 7 章，用户交互：涉及各类用户交互的 Python 库，从控制台应用到图形化用户界面应用，再到 Web 应用。

第 8 章，代码管理和改进：介绍系统管理工具，C 和 C++ 库接口工具，以及提升 Python 速度的方式。

第 9 章，软件接口：概述多个网络库，包括异步库和序列化及加密解密算法库。

第 10 章，数据操作：纵览符号数值算法库、绘图库，以及图像音频处理工具。

第 11 章，数据持久化：介绍与数据库交互的各个流行 ORM 库之间的一些区别。

第 7 章

用户交互

本章涉及的 Python 库可以帮助开发者编写与最终用户交互的代码。首先简述 Jupyter 项目，它非常独特，然后介绍更典型的命令行和图形化用户界面库（GUI），最后探讨 Web 应用的相关工具。

Jupyter Notebooks 项目

Jupyter 是一个 Web 应用，用户可以在上面交互式地展示和执行 Python 代码。考虑到它是一种连接用户的方式，因此简要介绍它。

用户在客户端机器的 Web 浏览器中查看 Jupyter 的客户端界面，这个界面以 CSS、HTML 和 JavaScript 编写而成，与 Python（或多种其他编程语言）实现的一个内核相互通信，内核执行一些代码块并向客户端响应结果。用户输入的内容在服务器上存储为 notebook（*.nb）格式，它是一种纯文本的 JSON，划分为一连串的单元（cell），每个单元包含 HTML、Markdown（一种人类可读的标记性语言，类似于 wiki 页面上用的那种）、原始笔记或可执行代码。服务器端可以在本地（用户自己的笔记本电脑）或远程机器上（例如 *https://try.jupyter.org/* 上的示例 notebooks）。

Jupyter 服务器端要求 Python 3.3 及以上版本，不过也同时兼容 Python 2.7。一些商业化 Python 二次发行版（例如 Canopy 和 Anaconda）的最新版本会捆绑安装 Jupyter，因此无须额外的安装步骤。对于常规安装，在设置好 Python 环境后，可在命令行中使用 pip 安装 Jupyter。

```
$ pip install jupyter
```

关于在课堂上使用 Jupyter 的一个近期调研（*https://arxiv.org/pdf/1505.05425v3.pdf*）发现：对于初学编程的学生而言，Jupyter 创建的交互式讲义非常高效，因而广受欢迎。

命令行应用

命令行（也称为控制台）应用是专为文本界面（例如，shell）设计的计算机程序，可以是简单的命令（例如，pep8 或 virtualenv），也可以是交互式命令，像 python 解释器或 ipython 那样。有些命令还有子命令，比如 pip install、pip uninstall 及 pip freeze，这些子命令又有自己的命令选项。所有命令行应用通常都从一个 main() 函数开始运行。

下面以一个 pip 调用示例来介绍调用命令行应用时可以指定的各个部分。

```
    ❶    ❷       ❸
$ pip install --user -r requirements.txt
```

❶ 命令是被调用的可执行程序的名称。

❷ 参数跟在命令后边，不以破折号开始，也被称为参量或子命令。

❸ 选项以一个连字符（用于单字符，例如 -h）或者两个连字符（用于单词，例如 --help）开始。它也可能会被称为标记或开关。

表 7-1 中的库都提供了各种选项来解析命令行参数（译注：这里的“参数”包含上面的“参数”和“选项”），或为命令行应用提供其他有用的工具。

表7-1　命令行工具

库	许可证	使用理由
argparse	PSF 许可证	• 标准库内置 • 提供标准的参数和选项解析功能
docopt	MIT 许可证	• 开发者需要控制帮助信息的版式 • 根据 POSIX 标准中定义的工具约定（*http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html*）解析命令行参数
plac	BSD 3- 条款许可证	• 从一个已有函数签名自动生成帮助信息 • 首先解析命令行参数，然后将参数直接传递给函数
click	BSD 3- 条款许可证	• 提供装饰器来构造帮助信息和解析器（与 plac 类似） • 允许组合使用多个子命令 • 可与其他 Flask 插件交互（click 独立于 Flask，但最初用于帮助用户组合使用来自不同 Flask 插件的命令行工具，由于它不会破坏什么，因此其实它已被用于 Flask 生态）

库	许可证	使用理由
clint	互联网软件联盟（ISC）许可证	• 为文本输出提供格式化特性，例如，着色、缩进及纵向展示 • 也为交互式输入提供类型检查（例如,检查是否是一个正则表达式、整数或路径） • 允许直接访问参数列表，也提供简单的过滤和分组工具
cliff	Apache 2.0 许可证	• 为具有多个子命令的大型 Python 项目提供一个结构化框架 • 构建一个交互式环境来使用子命令，但不需要额外编程

一般来说，你应该优先尝试使用 Python 标准库提供的工具，仅当标准库没有你想要的而其他库提供时才添加其他库。

下面几个小节将详细描述表 7-1 中列举的命令行工具。

argparse

Python 标准库中的 argparse 模块（替代现已废弃的 optparse）用于帮助解析命令行选项。HowDoI 项目提供的命令行接口使用了 argparse，构建命令行接口时可以参考它。

下面是生成解析器的代码。

```
import argparse
#
#  跳过多行代码
#

def get_parser():
  parser = argparse.ArgumentParser(description='...truncated for brevity...')
parser.add_argument('query', metavar='QUERY', type=str, nargs='*',
                            help='the question to answer')
  parser.add_argument('-p','--pos',
                           help='select answer in specified position (default:
1)',
                            default=1, type=int)
  parser.add_argument('-a','--all', help='display the full text of the
answer', action='store_true')
  parser.add_argument('-l','--link', help='display only the answer link',
                            action='store_true')
  parser.add_argument('-c', '--color', help='enable colorized output',
                            action='store_true')
  parser.add_argument('-n','--num-answers', help='number of answers to
return',
                            default=1, type=int)
```

```
    parser.add_argument('-C','--clear-cache', help='clear the cache',
                                action='store_true')
    parser.add_argument('-v','--version',
                                help='displays the current version of howdoi',
                                action='store_true')
    return parser
```

这个解析器会解析命令行，创建一个字典，将每个参数分别映射到一个值。比如其中action='store_true' 表明该选项是一个标识，如果命令行中存在该选项，那么解析器的字典中会将其值存为 True。

docopt

docopt 的核心哲学是文档应该漂亮并且容易理解。其提供一个主函数 docopt.docopt()，附加一些为高级用户准备的便捷函数和类。函数 docopt.docopt() 获取开发者编写的 POSIX 风格命令使用说明，基于使用说明来解析用户的命令行参数，并返回一个字典，包含从命令行解析出来的所有参数和选项。docopt 也会恰当地处理 --help 和 --version 选项。

在下面的例子中，变量 arguments 的值是一个字典，包含键 name、--capitalize 和 --num_repetitions。

```
#!/usr/bin env python3
"""Says hello to you.
  Usage:
    hello <name>... [options]
    hello -h | --help | --version

    -c, --capitalize  whether to capitalize the name
    -n REPS, --num_repetitions=REPS number of repetitions [default: 1]
"""

__version__ = '1.0.0' # --version需要

def hello(name, repetitions=1):
  for rep in range(repetitions):
    print('Hello {}'.format(name))

if __name__ == '__main__':
  for docopt import docopt
  arguments = docopt(__doc__, version=__version__)
  name = ' '.join(arguments['<name>'])
  repetitions = arguments['--num_repetitions']
  if arguments['--capitalize']:
```

```
    name = name.upper()
  hello(name, repetitions=repetitions)
```

自 0.6.0 版以来，docopt 可用于创建带子命令的复杂程序，这个程序具有类似 git 命令或 Subversions 的 svn 命令功能。甚至可用于以不同语言编写的子命令。推荐学习 git 命令的一个仿造实现（*https://github.com/docopt/docopt/tree/master/examples/git*）。

Plac

Plac 的哲学是：解析一个命令调用，其需要的全部信息都在目标函数的签名中。它对 Python 标准库的 argparse 做了一个轻量（大约 200 行）的封装，提供一个主函数 plac.plac()，用于从目标函数签名推断出参数解析器，解析命令行，然后调用函数。

这个库原本命名为命令行参数解析器（clap），但在作者选择这个名称一段时间之后，就没人使用这个名称了，最后改成了 Plac，也就是 clap 反着写了。虽说命令的用法声明通常信息量不大，但下面这个例子只用了几行代码。

```
# hello.py

def hello(name, capitalize=False, repetitions=1):
  """Says hello to you."""
  if capitalize:
    name = name.upper()
  for rep in range(repetitions):
    print('Hello {}'.format(name))

if __name__ == '__main__':
  import plac
  plac.call(hello)
```

其用法声明看起来是这样的：

```
$ python hello.py --help
usage: hello.py [-h] name [capitalize] [repetitions]

Says hello to you.

positional arguments:
  name
  capitalize [False]
  repetitions [1]

optional arguments:
  -h, --help  show this help message and exit
```

如果想在参数传递给函数之前对其进行类型转换（转换成正确的类型），那么可使用 annotations 装饰器。

```
import plac

@plac.annotations(
  name = plac.Annotation("the name to greet", type=str),
  capitalize = plac.Annotation("use allcaps", kind="flag", type=bool),
  repetitions = plac.Annotation("total repetitions", kind="option", type=int)
)
def hello(name, capitalize=False, repetitions=1):
  """Says hello to you."""
  if capitalize:
    name = name.upper()
  for rep in range(repetitions):
    print('Hello {}'.format(name))
```

另外，plac.Interpreter 提供了一种轻量方式来构建快速交互式的命令行应用。Plac 的交互式模式文档中有相关示例，参见 h*ttps://github.com/kennethreitz-archive/plac/blob/master/doc/plac_adv.txt*。

Click

Click（命令行接口构建工具集）的主要目的是帮助开发者以尽可能少的代码创建可组合的命令行接口。Click 的文档澄清了其与 docopt 的关系。

> Click 的目标是构建可组合的系统，而 docopt 的目标是构建最漂亮的手工命令行接口。在一些细微之处，这两个目标存在冲突。Click 为了实现统一的命令行接口，主动防止使用者实现某些模式。举例来说，开发者很难大幅度调整帮助信息页的格式。

其默认设置能满足大多数开发者的需求，同时对于高级用户又高度可配置。和 Plac 一样，其使用装饰器将解析器定义与函数相关联，始终在函数之外管理命令行参数。

Click 的 hello.py 应用如下所示。

```
import click

@click.command()
@click.argument('name', type=str)
@click.option('--capitalize', is_flag=True)
@click.option('--repetitions', default=1, help="Times to repeat the
greeting.")
def hello(name, capitalize, repetitions):
```

```
    """Say hello, with capitalization and a name."""
    if capitalize:
      name = name.upper()
    for rep in range(repetitions):
      print('Hello {}'.format(name))

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

Click 从命令的文档字符串中解析出命令描述，并使用一个定制解析器（衍生自 Python 标准库中现已废弃的 optparse）来创建命令的帮助信息，相比 argparse，这个定制解析器与 POSIX 标准的兼容性更好[1]。帮助信息如下所示。

```
$ python hello.py --help
Usage: hello.py [OPTIONS] NAME

  Say hello, with capitalization and a name.

Options:
  --capitalize
  --repetitions   INTEGER Times to repeat the greeting.
  --help          Show this message and exit.
```

Click 的真正价值在于其模块可组合性，添加一个外层分组函数，项目中其他 Click 装饰的函数就都成了这个顶级命令的子命令。

```
import click

@click.group()                                    ❶
@click.option('--verbose', is_flag=True)
@click.pass_context                               ❷
def cli(ctx, verbose):
    ctx.obj = dict(verbose = verbose)             ❸
    if ctx.obj['verbose']:
        click.echo('Now I am verbose.')

# 此处 hello 函数同前面所示

if __name__ == '__main__':
    cli()                                         ❹
```

❶ group() 装饰器构建一个顶级命令，先于被调用子命令执行。

❷ pass_context 装饰器（可选）决定了如何将对象从分组命令传递给子命令。传递给子

1 docopt 既不使用 optparse 也不使用 argparse，而是依赖正则表达式来解析文档字符串。

命令的首个参数是一个 click.core.Context 对象。

❸ 这个对象有个特殊属性 ctx.obj，可被传递给使用了 @click.pass_context 装饰器的子命令。

❹ 现在不是调用函数 hello()，而是调用 @click.group() 装饰的函数，在这个用例中是 cli()。

Clint

Clint（Command-line interface tools）是一套命令行接口工具集。提供的特性包括：命令行着色和缩进、一个简单而强大的列式打印器、基于迭代器的进度条，以及隐式参数处理。下面这个例子演示了着色和缩进工具。

```
"""Usage string."""
from clint.arguments import Args
from clint.textui import colored, columns, indent, puts

def hello(name, capitalize, repetitions):
  if capitalize:
    name = name.upper()
  with indent(5, quote=colored.magenta(' ~*~', bold=True)):                ❶
    for i in range(repetitions):
      greeting = 'Hello {}'.format(colored.green(name))                    ❷
      puts(greeting)                                  ❸


if __name__ == '__main__':
  args = Args()                                       ❹
  # 先检查，条件满足则展示帮助信息
  if len(args.not_flags) == 0 or args.any_contain('-h'):
    puts(colored.red(__doc__))
    import sys
    sys.exit(0)

  name = " ".join(args.grouped['_'].all)              ❺
  capitalize = args.any_contain('-c')
  repetitions = int(args.value_after('--reps') or 1)
  hello(name, capitalize=capitalize, repetitions=repetitions)
```

❶ Clint 的缩进特性是一个上下文管理器，用在 with 语句中很直观。quote 选项为每行添加一个洋红色粗体 ~*~ 前缀。

❷ colored 模块有 8 个着色函数，并提供一个选项来关闭着色。

❸ puts() 函数类似 print()，但还处理缩进和引用格式。

❹ Args 为参数列表提供一些简单的过滤工具，并返回另一个 Args 对象，这样可以串接过滤器。

❺ 此处演示如何使用 Args() 产生的 args。

cliff

cliff（命令行接口形式化框架）是一个用于构建命令行程序的框架。该框架就是为了用于创建类似 svn（Subversion）或 git 那样的多级命令，以及类似 Cassandra 或 SQL shell 那样的交互式程序。

cliff 的功能被分成若干抽象基类，对于每个子命令都必须先实现一次 cliff.command.Command，然后 cliff.commandmanager.CommandManager 负责代理到正确的命令。下面是一个最精简的 hello.py 实现。

```
import sys

from argparse import ArgumentParser                          ❶
from pkg_resources import get_distribution

from cliff.app import App
from cliff.command import Command
from cliff.commandmanager import CommandManager

__version__ = get_distribution('HelloCliff').version        ❷


class Hello(Command):
  """Say hello to someone."""

  def get_parser(self, prog_name):                           ❸
    parser = ArgumentParser(description="Hello command", prog=prog_name)
    parser.add_argument('--num', type=int, default=1, help='repetitions')
    parser.add_argument('--capitalize', action='store_true')
    parser.add_argument('name', help='person\'s name')
    return parser

  def take_action(self, parsed_args):                        ❹
    if parsed_args.capitalize:
      name = parsed_args.name.upper()
    else:
      name = parsed_args.name
```

```
        for i in range(parsed_args.num):
            self.app.stdout.write('Hello from cliff, {}.\n'.format(name))


    class MyApp(App):                                                    ❺
        def __init__(self):
            super(MyApp, self).__init__(
                description='Minimal app in Cliff',
                version=__version__,
                command_manager=CommandManager('named_in_setup_py'),    ❻
            )


    def main(argv=sys.argv[1:]):
        myapp = MyApp()
        return myapp.run(argv)
```

❶ cliff 直接将 argparse.ArgumentParser 应用到其命令行接口。

❷ 从 setup.py（之前 pip install 安装依赖包执行过的）中获取版本。

❸ get_parser() 是抽象基类要求的，它应返回一个 argparse.ArgumentParser。

❹ take_action() 也是抽象基类要求的，在 Hello 命令被调用时会运行。

❺ 主应用是 cliff.app.App 的子类，负责设置日志输出、I/O 流，以及任何其他会应用到所有子命令的全局配置。

❻ CommandManager 管理所有 Command 类，其根据 setup.py 中 entry_points 的内容找到命令名称。

图形化用户界面应用

本节先罗列一些窗口部件的库，提供按钮、滚动条、进度条及其他预建组件的工具包和框架，最后列举一些游戏开发相关的库。

窗口部件库

在图形化用户界面（GUI）开发的语境中，窗口部件是指按钮、滑动条、滚动条及其他常用的用户界面控制和展示元素。使用它们，你不再需要处理低层次的编程细节，例如在鼠标点击时识别鼠标下面是哪个按钮，甚至是更低层次的任务，例如每种操作系统所使用的不同的窗口 API。

在刚接触图形化用户界面开发时，你最期望的应该是易于使用的工具库，这样就能快速上手实现图形化用户界面。为此，我们推荐 Tkinter，它内置于 Python 标准库中。你可能还关心支撑 Python 库的底层工具包的结构和功能，因此我们按照底层工具包对 Python 库进行归类，按流行度先后排序，如表 7-2 所示。

表7-2　图形化用户界面部件库

底层库（语言）	Python 库	许可证	使用理由
Tk(Tcl)	tkinter	Python 软件基金会许可证	• 所有依赖都已捆绑进 Python 发行版 • 提供标准的 UI 部件，如按钮、滚动条、文本框及画布
SDL2(C)	Kivy	MIT 或 LGPL3（1.7.2 版本之前）	• 可用于实现 Android 应用 • 支持多点触控特性 • 尽可能优化成 C，并使用 GPU
Qt(C++)	PyQt	GNU 通用公共许可证（GPL）或商业许可证	• 提供跨平台一致的界面外观 • 许多应用和库依赖于 Qt（例如，Eric IDE、Spyder 及 Matplotlib），因此也许系统已安装 Qt 了 • Qt5（不能与 Qt4 同时使用）提供开发 Android 应用的工具
Qt(C++)	PySide	GNU 宽通用公共许可证（LGPL）	• 一个许可证更宽容的 PyQt 替代品
GTK(C)(GIMP 工具包)	PyGObject(PyGi)	GNU 宽通用公共许可证（LGPL）	• 提供 GTK+ 3 的 Python 绑定 • 对于那些在 GNOME 桌面系统上做过开发的人来说应该比较熟悉
GTK(C)	PyGTK	GNU 宽通用公共许可证（LGPL）	• 应该仅当项目已使用 PyGTK 时才使用；最好把老的 PyGTK 代码移植到 PyGObject 上
wxWindows（C++）	wxPython	wxWindows 许可证（一个经过修改的 LGPL）	• 通过为不同平台直接暴露不同的视窗库提供原生的界面外观 • 这意味着部分代码会因平台而异
Objective-C	PyObjC	MIT 许可证	• 提供 Objective-C 访问接口 • 为 Mac OS X 项目提供原生外观 • 无法在其他平台上使用

下面几小节将进一步介绍表 7-2 中列举的 Python GUI 编程库，根据每个库依赖的底层工

具包进行归类。

Tk

Python 标准库中的模块 Tkinter 是 Tk 之上的一个面向对象轻量封装层，Tk 则是 Tcl 语言编写的一个 GUI 部件库（通常两者一起写成 Tcl/Tk[2]）。因为标准库内置 Tkinter，所以它是这个列表中使用最方便、兼容性最好的 GUI 工具包。Tk 和 Tkinter 在多数 Unix 平台、Windows 和 Mac OS X 上都可用。

推荐阅读 TkDocs，它是一份不错的 Tk 教程，其中包含多种编程语言的示例，也可以通过 Python 的 wiki 页面（*https://wiki.python.org/moin/TkInter*）进一步学习。

如果安装了 Python 标准发行版，那么系统上应该有 IDLE（一个完全用 Python 编写的图形化用户界面的交互式编程环境），它是 Python 标准库的一部分。在命令行中输入“idle”来启动它，也可以查看其全部源码。在 shell 中输入以下内容可以找到其安装路径。

```
$ python -c"import idlelib; print(idlelib.__path__[0])"
```

在该目录下有很多文件，IDLE 主应用从 PyShell.py 模块启动。

turtle 模块的源码是绘图接口 tkinter.Canvas 的一个用例。在 shell 中输入以下内容可以找到它。

```
$ python -c"import turtle; print(turtle.__file__)"
```

Kivy

Kivy 是一个用于开发多点触控富媒体应用的 Python 库。社区正在积极地开发 Kivy，它使用一个宽松的类 BSD 许可证，能够运行在所有主流平台（Linux、Mac OS X、Windows，以及 Android）上。Kivy 以 Python 语言编写，没有使用任何底层视窗工具包，而是直接与 SDL2（简易直控媒体层）交互。SDL2 是一个 C 库，提供对用户输入设备[3]及音频的低层次访问，还使用 OpenGL（或者 Windows 上的 Direct3D）进行 3D 渲染。Kivy 有一些窗口部件（在模块 kivy.uix 中），但没有最流行的 Qt 和 GTK 的窗口部件数量多。如果你正在开发一个传统的桌面商业应用，那么 Qt 或者 GTK 可能更合适。

若要安装 Kivy，可前往 Kivy 的下载页（*https://kivy.org/#download*）找到你的操作系统，为你的 Python 版本下载正确的 ZIP 文件，并按照对应操作系统的安装说明进行安装。Kivy 的代码中有个目录包含大量示例，演示了各种 API 的使用方式。

2 Tcl 是一种轻量的编程语言，由 John Ousterhout 于 20 世纪 90 年代初为集成电路设计而创建。

3 除了支持常规鼠标，还可以处理触控输入：TUIO，一个开源的触控与手势协议和 API；任天堂的 Wii remote；Windows 的触控 APIWM_TOUCH()；苹果产品使用 HidTouch 驱动的 USB 触屏输入，以及其他输入方式。

Qt

Qt 是一个跨平台的应用框架，广泛用于开发图形化用户界面的软件，也可以用于开发无图形化用户界面的应用。它还有一个适用于 Android 平台的 Qt5 版本（*http://doc.qt.io/qt-5/android-support.html*）。如果你已安装 Qt（当你使用 Spyder、Eric IDE、Matplotlib 或者其他使用 Qt 的工具时），那么可以在命令行中输入以下命令查看 Qt 的版本。

```
$ qmake -v
```

Qt 基于 LGPL 许可证发行，允许开发者发行使用 Qt 的二进制程序，只要不修改 Qt 即可。商业许可证则可以让你获得数据可视化这类附加工具和售卖型应用。Qt 是一个框架，为不同类型应用提供了一些预建的脚手架，其 Python 接口库 PyQt 和 PySide 的文档都不太好，最好参考 Qt 自己的 C++ 文档（*http://doc.qt.io/*）。下面简要描述一下这两个接口库。

PyQt

英国河岸计算有限公司主持开发的 PyQt 比 PySide 更流行一些（PySide 还没有 Qt5 的版本）。安装 PyQt 时，请按照 PyQt4（*http://pyqt.sourceforge.net/Docs/PyQt4/installation.html*）或 PyQt5（*http://pyqt.sourceforge.net/Docs/PyQt5/installation.html*）安装文档的要求进行。PyQt4 仅能使用 Qt4，PyQt5 则仅能使用 Qt5（如果必须同时使用两者来开发，那么请使用 Docker，它是一个用户空间隔离工具，这样就不用反复改变依赖库的路径了）。

河岸计算有限公司还发布了 pyqtdeploy——一个基于 PyQt5 实现的图形化用户界面工具，可生成平台相关的 C++ 代码，用于构建二进制发行文件。更多信息可查看这些 PyQt4 教程和 PyQt5 示例。

PySide

在诺基亚还拥有 Qt 时，由于没法说服河岸计算有限公司将 PyQt 的许可证从 GPL 改成 LGPL，因此诺基亚发布了 PySide。PySide 项目意在成为 PyQt 的直接替代品，但在开发进度上落后于 PyQt。wiki 页面描述了 PySide 和 PyQt 之间的不同之处（*http://bit.ly/differences-pyside-pyqt*）。

若要安装 PySide，可按照 Qt 文档（*https://wiki.qt.io/Setting-up-PySide*）中的安装说明进行。*https://wiki.qt.io/Hello-World-in-PySide* 能帮助你编写第一个 PySide 应用。

GTK+

GTK+ 工具箱（GTK 是 GIMP[4] 工具箱的缩写）提供 API 来访问 GNOME 桌面环境核心功能。有些程序员或许因为更喜欢 C 并且对于必要时候可以深入 GTK 源码觉得更舒服，或许因为之前编写过 GNOME 应用并且更习惯于 GTK+ 的 API，而选择 GTK+ 而不是 Qt。对 GTK+ 进行 Python 绑定的两个库是 PyGTK 和 PyGObject。

PyGTK

> 虽然 PyGTK 为 GTK+ 提供 Python 绑定，但是目前仅支持 GTK 2.x API（不支持 GTK+ 3+）。该项目已停止开发，其开发团队建议不要把 PyGTK 用于新项目，同时建议将已有应用从 PyGTK 移植到 PyGObject。

PyGObject（又称 PyGI）

> PyGObject 提供的 Python 绑定支持访问整个 GNOME 软件平台的功能。PyGObject 又名为 PyGI，因为它借助了 GObject 自省项目（*https://wiki.gnome.org/Projects/GObjectIntrospection*），并为其提供了一套 Python API。GObject 自省项目是其他编程语言和 GNOME 核心 C 库 GLib 之间的一套桥接 API，不过它们都要遵从 GObject 定义的约定。PyGObject 完全兼容 GTK+ 3。Python GTK+ 3 教程（*http://python-gtk-3-tutorial.readthedocs.io/en/latest/*）是一份不错的入门资料。
>
> 若要安装 PyGObject，可从 PyGObject 的下载站点 *http://bit.ly/pygobject-download* 获取二进制安装文件。在 Mac OS X 上，可使用命令 brew install pygobject 通过 homebrew 来安装。

wxWidgets

wxWidgets 背后的设计哲学是：对于一个应用而言，要想实现原生界面外观，最佳方式就是使用每个操作系统的原生 API。现在 Qt 和 GTK+ 在底层也可以使用 X11 之外的其他视窗库，但 Qt 是对不同视窗库进行抽象，而 GTK+ 则是让它们用起来看似在做 GNOME 编程。使用 wXWidgets 的好处是直接和各个平台进行交互，并且其许可证也更加宽松。但问题是你必须对各个平台进行不同的处理。

wxPython 是为 Python 用户封装 wxWidgets 的 Python 扩展模块。可能因为使用原生界面工具的哲学，其一度是最流行的 Python 视窗库，但现在 Qt 和 GTK+ 的变通方法看起来已经足够好了。同样，如果要安装 wxPython，那么请登录 *https://www.wxpython.org/download.php#stable* 下载对应操作系统的安装包，并学习 wxPython 教程（*http://bit.ly/*

4 GIMP 是 GNU 图像处理程序（GNU Image Manipulation Program）的缩写。GTK+ 原本用来支持 GIMP 内绘图，随着它的流行，有人想用它来构建一个完整的桌面窗口环境，因此也就有了 GNOME。

wxpython-getting-started）。

Objective-C

Objective-C 是苹果公司用于 Mac OS X 和 iOS 操作系统的专有编程语言，为 Mac OS X 上应用开发提供对 Cocoa 框架的访问。Objective-C 不是跨平台的，它只能用于苹果的产品。

PyObjC 是 Mac OS X 上 Objective-C 语言和 Python 语言之间的一个双向桥接方案，这意味着 Mac OS X 上的应用开发，不仅可以通过 Python 访问 Cocoa 框架，也可以通过 Objective-C 访问 Python[5]。

Cocoa 框架仅在 Mac OS X 上可用，因此如果编写跨平台应用，那么就不要选择 Objective-C，也不要通过 PyObjC 使用 Objective-C。

使用 PyObjC 需要先安装 Xcode，因为 PyObjC 需要一个编译器。并且，PyObjC 仅能和标准 CPython 发行版一起使用，其他发行版，例如 PyPy 或者 Jython 都不行。我们推荐使用 Mac OS X 提供的 Python 可执行文件，因为它被苹果公司修改过，是专为在 Mac OS X 上使用而配置的。

使用系统的 Python 解释器创建虚拟环境，应使用完整路径来调用 Python 解释器。如果不想以超级用户身份安装虚拟环境管理包，则可以使用 --user 开关选项进行安装，这样 Python 库会保存在 $HOME/Library/Python/2.7/lib/python/site-packages/ 路径下。

```
$ /usr/bin/python -m pip install --upgrade --user virtualenv
```

首先激活并进入虚拟环境，然后安装 PyObjC。

```
$ /usr/bin/python -m virtualenv venv
$ source venv/bin/activate
(venv)$ pip install pyobjc
```

这要花点时间。PyObjC 捆绑自带 py2app，这是 Mac OS X 特有的工具，用于创建可发行的独立应用二进制文件。在 PyObjC 示例页面（*http://pythonhosted.org/pyobjc/examples/index.html*）上可以看到一些应用样例。

5 不过 Swift 的出现可能减少了这种需求。Swift 像 Python 一样易用，如果只是针对 Mac OS X 编写程序，那为什么不用 Swift 原生实现一切呢（NumPy 和 Pandas 这类科学计算库在纯计算应用方面具有明显优势）？

游戏开发

Kivy 现在已风靡四方，但相比本节罗列的库，它的体量要大得多。本书将其归为工具包，是因为虽然 Kivy 经常被用于构建游戏，但其提供的是部件和按钮。Pygame 社区搭建了一个 Python 游戏开发者网站（*http://www.pygame.org/hifi.html*），无论是否使用 Pygame，都欢迎游戏开发者参与。以下是一些最流行的游戏开发库。

cocos2d

> cocos2d 以 BSD 许可证发行，构建在 pyglet 上，提供一个框架，将游戏组织成通过自定义工作流连接的一组场景，所有场景都归一个导演管理。如果喜欢文档（*http://python.cocos2d.org/doc/programming_guide/basic_concepts.html#scenes*）中描述的“场景—导演—工作流”，或者想使用 pyglet 来绘图，辅之 SDL2 来控制操作杆和音频，那就用这个库。我们可以使用 pip 安装 cocos2d。至于 SDL2，先看看包管理器是否能安装它，如果不能，则从 SDL2 官网（*https://www.libsdl.org/*）下载。最佳入门方式是学习它们的 cocos2d 示例应用（*https://github.com/los-cocos/cocos/tree/master/samples*）。

pyglet

> pyglet 以 BSD 许可证发行。它是 OpenGL 的一组轻量封装模块，辅之其他工具在窗口中呈现和移动游戏界面。因为几乎每台计算机上都安装了 OpenGL，所以仅需要 pip 就能直接安装 。尝试运行几个示例应用（*https://bitbucket.org/pyglet/pyglet/src/default/examples/*），其中包括以不到 800 行代码就实现的一个行星游戏的完整克隆（*https://bitbucket.org/pyglet/pyglet/src/default/examples/astraea/astraea.py?fileviewer=file-view-default*）。

Pygame

> Pygame 以 GNU LGPLv2.1 许可证发行。其社区庞大而活跃，社区中积累了大量的 Pygame 教程（*http://www.pygame.org/wiki/tutorials*），不过请注意它一直使用 SDL 的前一个版本 SDL1。Pygame 还没发布在 PyPI 上（译注：从 Pygame 1.9.2 版本后已发布在 PyPI 上，可以通过 pip install pygame 直接安装），因此先看看包管理器是否能安装它，如果没有，就下载 Pygame（*http://www.pygame.org/download.shtml*）进行安装。

Pygame-SDL2

> Pygame-SDL2（*http://pygame-sdl2.readthedocs.io/en/latest/*）近期已发布，是以

SDL2 作为后端重新实现 Pygame 的一个努力尝试。它以 Zlib 许可证发行，但从 Pygame 中借用了部分代码，这部分代码使用 LGPL2 许可证 。

pySDL2

PySDL2 可运行在 CPython、IronPython 及 PyPy 上，是对 SDL2 库进行封装的一个 Python 轻量接口层。如果需要一个最轻量的 SDL2 Python 接口，那么就选择这个库。推荐阅读 PySDL2 教程（*http://pysdl2.readthedocs.io/en/latest/tutorial/index.html*）进一步了解它。

Web 应用

Python 是一门强大的脚本语言，适用于快速原型和大型项目，被广泛应用于 Web 应用开发（YouTube、Pinterest、Dropbox 及洋葱新闻网的开发都使用了 Python）。

第 5 章剖析了 Werkzeug 和 Flask 两个库，它们与构建 Web 应用相关。在介绍这两个库时，我们简要描述了 Web 服务器网关接口（WSGI），即 PEP 3333（*https://www.python.org/dev/peps/pep-3333/*）中定义的一个 Python 标准。该标准规定了 Web 服务器和 Python Web 应用之间应该如何通信。本节将介绍几个 Python Web 框架，内容包括它们的模板系统、与之交互的服务器及运行平台。

Web 框架 / 微框架

一个 Web 框架包含一组库和一个主处理程序，在主处理程序中可以构建自定义代码来实现一个 Web 应用（例如，一个交互式网站，它的代码运行在服务器上，向客户端提供 API）。多数 Web 框架至少会包含模式和工具来实现以下功能。

1．URL 路由

匹配一个进入的 HTTP 请求，路由到一个特定的 Python 函数（或回调）进行处理。

2．处理请求和响应对象

对接收自或发送给用户浏览器的信息进行封装。

3．模板渲染

将 Python 变量注入 HTML 模板或其他输出，允许程序员把应用逻辑（Python 编写的）与版式（模板中实现）分开。

4．便于调试的开发环境 Web 服务

在开发机器上运行一个微型 HTTP 服务器，方便快速开发；通常可以在文件更新时自动重新加载服务器端代码。

你没有为框架本身编写代码的必要。框架应该为你准备好需要的特性，这些特性必须是已被大量其他开发者使用或测试过的特性。因此如果你还没找到需要的特性，那就继续调研其他可用框架（例如 Bottle、Web2Py、CherryPy）。一个技术评审人还提醒我们应该提及 Falcon（*http://falcon.readthedocs.io/en/stable/*），这个框架专门用于构建 RESTful 风格 API（也就是说不会提供 HTML 格式响应的服务）。

表 7-3 中的所有库都可以使用 pip 进行安装。

```
$ pip install Django
$ pip install Flask
$ pip install tornado
$ pip install pyramid
```

表7-3　Web框架

Python 库	许可证	使用理由
Django	BSD 许可证	• 提供工程整体结构，即一个大部分都预建好的站点，开发者只需设计页面布局及底层的数据和逻辑 • 自动生成一个 Web 管理后台，通过这个后台，非开发人员也可以增删数据（如新闻文章） • 集成了 Django 的对象，关系映射（ORM）工具
Flask	BSD 许可证	• 允许开发者完全控制 Web 开发工具栈 • 提供优雅的装饰器，便于将 URL 路由添加到任何函数 • 可以把你从 Django 或 Pyramid 限制的工程结构中解放出来
Tornado	Apache 2.0 许可证	• 提供卓越的异步事件处理，Tornado 使用自带的 HTTP 服务器 • 提供开箱即用的方式来处理大量 WebSocket 连接（基于 TCP[a] 的全双工、持久通信）或其他类型的长连接
Pyramid	修改后的 BSD 许可证	• 提供某种预建的工程结构，该结构被称为骨架，但 Pyramid 没 Django 做得多，允许开发者使用任意一种数据库接口和模板渲染库 • 基于流行的 Zope 框架和 Pylons，两者都是 Pyramid 的前身

a 传输控制协议（TCP）是一个标准协议，定义了一种方式在两台计算机之间建立网络连接并相互通信。

下面几节详细介绍了表 7-3 中罗列的 Web 框架。

Django

Django 是一个“自带电池”的 Web 应用框架，对于创建内容导向的网站而言，是一个极好的选择。Django 提供了大量开箱即用的工具和模式，以快速构建复杂、数据库支持的 Web 应用为目标，同时鼓励在编码时遵从最佳实践。

Django 的社区非常庞大而活跃，大量可复用模块（*http://djangopackages.com/*）可以直接导入新工程中使用，或者按需裁剪使用。

美国（*http://djangocon.us/*）和欧洲（*http://djangocon.eu/*）都有 Django 年度大会，如今大部分新出的 Python Web 应用都使用 Django 构建。

Flask

Flask 是一个 Python 微框架，对于构建小型的应用、API 或 Web 服务来说是一个绝佳选择。Flask 的目标不是为开发者提供可能需要的所有东西，而是实现一个 Web 应用框架最常用的核心组件，例如 URL 路由、HTTP 请求和响应对象及模板。使用 Flask 构建应用非常类似于编写标准的 Python 模块，某些函数绑定了路由（通过一个装饰器来实现，和下面的示例代码类似）。

```
@app.route('/deep-thought')
def answer_the_question():
  return 'The answer is 42.'
```

使用 Flask，如果应用需要其他组件，则要靠你自己选择。例如，Flask 没有内置数据库访问或表单生成 / 校验特性。因为许多 Web 应用并不需要这些特性。如果你的应用需要，那么有很多扩展可选择（*http://flask.pocoo.org/extensions/*），比如使用 SQLAlchemy 来访问关系型数据库（*http://flask-sqlalchemy.pocoo.org/*），使用 pyMongo 来访问 MongoDB 数据库（*https://docs.mongodb.org/getting-started/python/*），使用 WTForms 来处理表单（*https://flask-wtf.readthedocs.org/*）。

如果一个 Python Web 应用不适用 Django 的预建脚手架，则应该优先考虑使用 Flask。尝试一下 Flask 官网上的示例应用（*https://github.com/pallets/flask/tree/master/examples*）是一种不错的入门方式。如果想要一个工程运行多个应用（Django 默认如此），那么可使用应用调度特性。如果想在一个应用的多个子页面之间复制某些操作行为，则可尝试 Flask 的蓝图特性。

Tornado

Tornado 是一个 Python 异步（事件驱动非阻塞，类似 Node.js）Web 框架，内置事件循

环[6]。这样也能够原生支持 WebSockets 通信协议。与本节介绍的其他框架不同，Tornado 不是一个 WSGI 应用，但是它既可以作为一个 WSGI 应用运行，也可以把 tornado.wsgi 模块（*http://www.tornadoweb.org/en/stable/wsgi.html*）作为一个 WSGI 服务器运行[7]，WSGI 是一个同步接口，而 Tornado 的目的是提供一个异步框架。

相比 Django 和 Flask，Tornado 用起来更难一点，用的人也少些。某些情况下需要使用异步框架来获得更好的性能，即使花费更多的编程时间也值得，此时就用 Tornado。Tornado 的演示应用（*https://github.com/tornadoweb/tornado/tree/master/demos*）适合学习。Tornado 应用写好了性能会非常卓越。

Pyramid

Pyramid 非常类似 Django，不过更强调模块化。它的内置库相对要少些，鼓励用户通过共享模板（被称为脚手架）来扩展基础功能。使用脚手架之前先注册它，创建新项目，使用命令 pcreate 来构建项目骨架时会启用这个脚手架。pcreate 命令类似于 Django 的 django-admin startproject project-name 命令，但提供一些选项来支持不同工程结构、不同数据库后端，还有 URL 路由选项。

Pyramid 用户基数不大，不能与 Django 和 Flask 相比，但使用 Pyramid 的人对其非常热诚。它是一个非常强大的框架，但是到目前为止对于新的 Python Web 应用而言还不是一个流行选择。

https://trypyramid.com/ 网上的 Pyramid 教程适合上手。如果你想向老板推荐 Pyramid，那么可以尝试借助全方位介绍 Pyramid 的门户网站（*https://trypyramid.com/*）。

Web 模板引擎

多数 WSGI 应用都是以 HTML 或其他标记语言响应 HTTP 请求和提供内容服务。模板引擎正是负责渲染这种内容：管理一套模板文件，以一个分层和包含系统来避免不必要的重复，以应用生成的动态内容来填充模板的静态内容。这有助于遵从关注点分离[8]理念，应用逻辑仅在代码中实现，委托模板来展现。

模板文件有时由设计师或前端开发者来编写，页面越复杂，协作的难度也就越大。下面针对应用如何向模板引擎传递动态内容和模板本身如何编写两个方面提供一点建议。

---

6 其受 Twisted 项目的 TwistedWeb 服务器启发，是 Tornado 网络编程套件的一部分。如果你期望的东西在 Tornado 中没找到，那么可以再看看 Twisted，很可能已经实现。不过提醒一下你：众所周知，初学者上手 Twisted 很有难度。

7 实际上，其文档中 WSGI 相关部分也说过："仅当在同一个进程中组合使用 Tornado 和 WSGI 带来的好处远大于降低可伸缩性时，才使用 WSGIContainer。"

8 关注点分离是一种设计原则，意思是良好的代码是模块化的，每个组件都应该仅做一件事。

永不胜于仓促

仅传递模板渲染需要的动态内容给模板文件。抵制传递额外内容以防万一的思维诱惑：需要时添加缺失变量，比之后再删除一个可能不需要的变量更容易。

尽量保持模板无应用逻辑

许多模板引擎允许在模板中使用复杂语句或赋值，并且许多引擎还允许在模板中执行一些 Python 代码。这样看似便利，却会导致复杂性飙升，也经常导致 bug 更难定位。虽然不完全抵制在模板中编写应用逻辑代码，但是相比纯净，更要考虑实用性，前提是要自制。

分离 JavaScript 与 HTML 内容

在很多情况下有必要混合 JavaScript 模板和 HTML 模板，但要保持理智，隔离 HTML 模板传递变量给 JavaScript 代码的部分。

表 7-4 中罗列的所有模板引擎都是新一代的，渲染速度很快[9]，并且基于老一代模板语言的使用经验添加了一些新特性。

表7-4　模板引擎

Python 库	许可证	使用理由
Jinja2	BSD 许可证	• Flask 的默认模板引擎，也可与 Django 配合使用 • 启发于 Django 模板语言，但模板中允许的逻辑结构更多一点 • Jinja2 是 Sphinx、Ansible 及 Salt 的默认模板引擎。如果你用过这几个软件，那么你应该对 Jinja2 有所了解
Chameleon	修改后的 BSD 许可证	• 模板本身即是正常的 XML/HTML • 类似于模板属性语言（TAL）及其衍生物
Mako	MIT 许可证	• Pyramid 的默认模板引擎 • 为速度而生，当模板渲染确实是性能瓶颈时推荐使用 • 允许在模板中编写大量逻辑代码，Mako 就好像是一个 Python 版本的 PHP（*http://php.Net*）

下面详细介绍表 7-4 中的这几个库。

Jinja2

开发新的 Python Web 应用，推荐使用 Jinja2 模板引擎库。Flask 和 Python 的文档生成器

9　页面渲染极少成为 Web 应用的瓶颈，不过数据访问通常是瓶颈。

Sphinx 都采用 Jinja2 作为默认模板引擎，它还可以配合 Django、Pyramid 及 Tornado 使用。由于 Jinja2 使用一种基于文本的模板语言，因此可用于生成任何类型的标记，而不仅仅是 HTML。允许自定义过滤器、标签、测试器及全局函数。Jinja2 受 Django 模板语言启发，又添加了一些特性，如允许少量的模板内逻辑，大大减少了编码量。

下面是一些重要的 Jinja2 标签。

```
{# 这是一句注释 -- 前后有大括号和井号 #}

{# 下面这句是插入一个变量：#}
{{title}}

{# 下面这段定义了一个命名块，可被子模板替换 #}
{% block head %}
<h1>This is the default heading.</h1>
{% endblock %}

{# 下面这段演示如何使用迭代 #}
{% for item in list %}
<li>{{ item }}</li>
{% endfor %}
```

下面是一个使用 Jinja2 的网站示例，结合使用了 Tornado Web 服务器。

```
# 导入 Jinja2
from jinja2 import Environment, FileSystemLoader

# 导入 Tornado
import tornado.ioloop
import tornado.web

# 加载模板文件 templates/site.html
TEMPLATE_FILE = "site.html"
templateLoader = FileSystemLoader(searchpath="templates/")
templateEnv = Environment(loader=templateLoader)
template = templateEnv.get_template(TEMPLATE_FILE)

# 用于渲染的著名电影列表
movie_list = [
  [1, "The Hitchhiker's Guide to the Galaxy"],
  [2, "Back to the Future"],
  [3, "The Matrix"]
]

# template.render() 返回渲染的 HTML 结果字符串
```

```
html_output = template.render(list=movie_list, title="My favorite movies")

# 主页请求处理类
class MainHandler(tornado.web.RequestHandler):
  def get(self):
    # 返回模板渲染结果字符串给浏览器请求
    self.write(html_output)

# 将处理类分配给服务器根路径（127.0.0.1:PORT/）
application = tornado.web.Application([
  (r"/", MainHandler)
])
PORT=8884

if __name__ == "__main__":
  # 设置服务器
  application.listen(PORT)
  tornado.ioloop.IOLoop.instance().start()
```

文件 base.html 用作站点所有页面的基础模板。在这个例子中，所有页面都会实现 content 模板块（当前是空的）。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <link rel="stylesheet" href="style.css" />
  <title>{{title}} - My Web Page</title>
</head>
<body>
<div id="content">
  {# In the next line, the content from the site.html template will be added #}
  {% block content %}{% endblock %}
</div>
<div id="footer">
  {% block footer %}
  &copy; Copyright 2013 by <a href="http://domain.invalid/">you</a>.
  {% endblock %}
</div>
</body>
```

接下来的代码示例是我们要渲染的网页 (site.html)，它基于 base.html 进行了扩展。content 模板块会被自动插入 base.html 对应的模板块中。

```
<!{% extends "base.html" %}
{% block content %}
```

```
  <p class="important">
  <div id="content">
    <h2>{{title}}</h2>
    <p>{{ list_title }}</p>
    <ul>
      {% for item in list %}
      <li>{{ item[0]}} : {{ item[1]}}</li>
      {% endfor %}
    </ul>
  </div>
  </p>
{% endblock %}
```

Chameleon

Chameleon（*https://chameleon.readthedocs.org/*）页面模板引擎（文件扩展后缀为 *.pt）是模板属性语言（TAL）（*http://en.wikipedia.org/wike/Template_Attribute_Language*）、TAL 表达式语法（*http://bit/ly/expressions-tales*）及宏扩展 TAL（Metal）语法（*http://bit/ly/macros-metal*）的一个 HTML/XML 模板引擎实现。Chameleon 解析页面模板，将内容编译为 Python 字节码，从而提升加载速度。Chameleon 兼容 Python 2.5 及以上版本（包括 3.x 和 PyPy），是 Pyramid 使用的两个默认渲染引擎之一（另一个是 Mako）。

页面模板会在 XML 文档中添加一些特殊的元素属性和文本标记，即一组简单的语言结构可以控制文档流、元素重复、文本替换，以及翻译。因为语法基于属性，所以未经渲染的页面模板也是合法的 HTML，可以在浏览器中查看，甚至可以在所见即所得编辑器中编辑。这样更易于和设计师协作，在浏览器中使用静态文件来实现原型也更方便。TAL 语言基础知识非常简单，通过下面这个例子即能掌握。

```
<html>
    <body>
    <h1>Hello, <span tal:replace="context.name">World</span>!</h1>
        <table>
            <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
                <td tal:repeat="col 'juice', 'muffin', 'pie'">
                    <span tal:replace="row.capitalize()" /> <span
tal:replace="col" />
                </td>
            </tr>
        </table>
    </body>
</html>
```

因为 模式常用于文本插入，所以如果不要求未渲染模板

严格有效，则可以使用一种简练且可读性更好的语法，即 ${expression} 模式来替代，如下所示。

```
<html>
    <body>
        <h1>Hello, ${world}!</h1>
        <table>
            <tr tal:repeat="row 'apple', 'banana', 'pineapple'">
                <td tal:repeat="col 'juice', 'muffin', 'pie'">
                    ${row.capitalize()} ${col}
                </td>
            </tr>
        </table>
    </body>
</html>
```

完整的 Default Text 语法还允许在未渲染模板中使用默认内容。

在 Pyramid 世界以外，Chameleon 并未被广泛使用。

Mako

Mako 模板语言为追求性能最大化，会将模板编译成 Python 代码。其语法和 API 借鉴自其他模板语言（例如 Django 和 Jinja2 模板）的优良部分。PyramidWeb 框架自带 Mako 作为默认模板语言。Mako 实现的示例模板如下所示。

```
<%inherit file="base.html"%>
<%
  rows = [[v for v in range(0, 10)] for row in range(0, 10)]
%>
<table>
  % for row in rows:
    ${makerow(row)}
</table>

<%def name="makerow(row)"%>
  <tr>
  % for name in row:
    <td>${name}</td>\
  % endfor
  </tr>
</%def>
```

这是一种文本标记语言，类似于 Jinja2，可以用于生成任意格式的文本，而不仅仅是

XML/HTML 文档，可以像如下代码片段这样来渲染一个非常简单的模板。

```
from mako.template import Template
print(Template("Hello ${data}!").render(data="world"))
```

Python Web 社区非常认可 Mako。其速度快，且允许开发者在页面中嵌入很多 Python 逻辑，当然我们知道对此要警惕，在你认为必要时，它就是必杀技。

Web 部署

对于 Web 部署，本节将介绍两种方案：使用 Web 主机托管（例如，向 Heroku、Gondor 或 PythonAnywhere 等供应商购买服务，让它们来替你管理服务器和数据库），在 AWS 云服务（*https://aws.amazon.com/*）或 Rackspace（*https://www.rackspace.com/*）这类 VPS 服务商提供的机器上搭建自己的基础设施。下面介绍这两种方案。

主机托管

平台即服务（PaaS）是一类云计算基础设施，负责抽象和管理底层基础设施（例如，搭建数据库和 Web 服务器，并及时为系统打安全补丁）、路由及 Web 应用扩容。使用 PaaS，应用开发者只需集中精力编写应用代码，而无须关心部署细节。

市场上存在很多 PaaS 竞品服务供应商，如下供应商特别看重 Python 社区，其中多数都提供某些免费服务而且可以试用。

1. Heroku

   部署 Python Web 应用，推荐使用 PaaS 供应商 Heroku。它支持 Python 2.7 至 Python 3.5 的所有类型的应用：Web 应用、应用服务器及框架。Heroku 提供一组命令行工具（*https://toolbelt.heroku.com/*），可用于与用户 Heroku 账号及支撑应用的实际数据库和 Web 服务器进行交互，这样无须使用 Web 界面也能变更。Heroku 维护了一些详细介绍如何在 Heroku 平台上使用 Python 的文章（*https://devcenter.heroku.com/categories/python*），还有一些教程手把手教用户如何搭建首个应用（*https://devcenter.heroku.com/articles/getting-started-with-python#introduction*）。

2. Gondor

   Gondor 是一家小公司提供的服务，重点关注帮助企业顺利使用 Python 和 Django。其平台专用于 Django 和 Pinax[10] 应用部署和使用。Gondor 平台使用 Ubuntu 12.04 系统，支持 Django 1.4、1.6 和 1.7，支持部分 Python 2 和 Python 3 实现。如果使用

10 Pinax 捆绑了许多流行的 Django 模板、应用和基础设施，从而能够更快地开始一个 Django 项目。

local_settings.py 来配置站点特有信息，平台能够自动为用户配置 Django 站点。详细信息可查看 Gondor 的 Django 项目部署指南（*https://gondor.io/support/django/setup/*）。平台也提供了一个命令行接口工具。

3. PythonAnywhere

PythonAnywhere 支持 Django、Flask、Tornado、Pyramid，以及许多我们没有介绍的其他 Web 应用框架，如 Bottle（微框架，类似 Flask，但社区小得多）和 web2py（特别适合教学）。其收费模型和计算花费的时间相关。当计算超过每日最大值时会被限制，而不是收取更多费用，它适合注重节省成本的开发者。

Web 服务器

除 Tornado（自带 HTTP 服务器）外，我们讨论的所有 Web 应用框架都是 WSGI 应用。这意味着：为了接收 HTTP 请求并发回一个 HTTP 响应，它们必须和一个 WSGI 服务器（由 PEP 3333 定义）交互。

现在多数自托管 Python 应用都是搭配一个 WSGI 服务器（比如 Gunicorn）来提供服务，这个 WSGI 服务器可以作为一个独立的 HTTP 服务器来使用，或者运行在一个轻量 Web 服务器（比如 Nginx）的后面。如果选择后一种方案，WSGI 服务器负责与 Python 应用交互，Web 服务器则负责处理更适合它的任务，静态文件服务、请求路由、分布式拒绝服务（DDoS）防护，以及基础的身份认证。最流行的两个 Web 服务器是 Nginx 和 Apache。

1. Nginx

Nginx 是一个 Web 服务器，支持 HTTP、SMTP 及其他协议的反向代理 [11]，因高性能、相对简单、兼容很多应用服务器（如 WSGI 服务器）而著名。它也包含一些便捷的特性：如负载均衡 [12]、基础身份认证、流媒体等。Nginx 专为服务于高负载网站而设计，逐渐变得非常流行。

2. Apache HTTP 服务器

虽然 Apache 是世界上最流行的 HTTP 服务器（*http://w3techs.com/technologies/overview/web_server/all*），但是我们更推荐 Nginx。不过有些部署服务的新手可能仍然想从 Apache 和 mod_wsgi（被认为是最易用的 WSGI 接口）上手。每个框架的文档中都包含 mod_wsgi 相关部署教程，包括 mod_wsgi 方式部署 Pyramid 应用、Django 应用、Flask 应用（*https://docs.djangoproject.com/en/1.9/howto/deployment/wsgi/modwsgi/*）。

11 反向代理代表客户端从另一个服务器获取信息，并将信息返回客户端，就好像信息来自反向代理一样。
12 负载均衡把工作委托给多个计算资源，以此优化系统性能。

WSGI 服务器

相比传统 Web 服务器，独立的 WSGI 服务器通常使用的资源更少，在性能基准测试（*http://nichol.as/benchmark-of-python-web-servers*）上表现也非常好。它们也可以和 Nginx 或 Apache 配合使用（Nginx 和 Apache 作为反向代理提供服务）。最流行的 WSGI 服务器有 3 个。

1. Gunicorn（绿色独角兽 Green Unicorn）

   对于新开发的 Python Web 应用，推荐选择 Gunicorn，它是纯 Python 实现的 WSGI 服务器，用于服务 Python 应用。与其他 Python Web 服务器不同，Gunicorn 的用户接口经过深思熟虑，使用和配置极其简单。Gunicorn 为配置提供了合理的默认值。然而，其他某些服务器，例如 uWSGI 可定制性更高（因此若想高效使用，会更难一些）。

2. Waitress

   Waitress 是一个纯 Python 实现的 WSGI 服务器，性能合格。它的文档不太详细，但提供了一些 Gunicorn 没有的好用的功能（例如 HTTP 请求缓冲）。在客户端响应缓慢时也不会阻塞，因此得名 Waitress。在 Python Web 开发社区内，Waitress 受到越来越多的关注。

3. uWSGI

   uWSGI 是构建主机托管服务的一个完整技术栈。我们不推荐将其用作一个独立的 Web 请求路由，除非确实清楚为什么要这样做。

   uWSGI 也可以运行在一个完整 Web 服务器（例如 Nginx 或 Apache）的后面，Web 服务器可以通过 uWSGI 协议配置 uWSGI 和应用的行为。uWSGI 的 Web 服务器支持特性允许动态配置 Python、传输环境变量及进一步调优。完整细节请阅读 uWSGI 的魔法变量文档（*https://uwsgi-docs.readthedocs.io/en/latest/Vars.html*）。

第 8 章

代码管理和改进

有些程序库可用于管理或简化开发构建流程、系统集成、服务器管理或性能优化，本章将逐一介绍这些库。

持续集成

关于持续集成一词，没人比马丁 · 福勒[1] 解释得更好。

> 持续集成是一种软件开发实践，团队成员频繁地集成大家的工作，通常每个人每天至少会集成一次，因而每天都会有多次集成。每次集成都会有一次自动化构建（包含测试）来验证，尽可能快地检测集成错误。很多团队发现这种方式可以显著减少集成问题，团队也能更快速地开发出整合良好的软件。

目前最流行的 3 个持续集成工具分别是 Travis-CI、Jenkins 和 Buildbot。它们通常会与 Tox 配合使用，Tox 是一个从命令行 virtuanenv 管理和测试的 Python 工具。Travis 支持单平台上多个 Python 解释器环境的持续集成，Jenkins（最流行的）和 Buildbot（用 Python 编写的）可以在多台机器上管理构建过程。许多团队也使用 Buildbot 和 Docker 实现快速重复地构建复杂的测试环境。

Tox

Tox 是一个自动化工具，可用于打包、测试，以及从终端或集成测试服务器上部署 Python 软件。它是一个 virtualenv 管理和测试的通用命令行工具，具有以下特性。

1 福勒是软件设计和开发方面最佳实践的拥护者，也是持续集成方面最活跃的支持者之一。这段引文摘自他写的一篇关于持续集成的博文（*http://martinfowler.com/articles/continuousIntegration.html*）。他主持过一系列相关的对话（*http://martinfowler.com/articles/is-tdd-dead/*），与 David Heinemeier Hansson（Ruby On Rails 的创建者）和 Kent Beck（极限编程（XP）运动的领导人物，CI 是极限编程的基石之一）一起讨论测试驱动开发（TDD）及其与极限编程的关系。

- 检查针对不同版本 Python 和解释器的软件包是否正确安装。
- 在每个环境中运行测试，配置用户选择的测试工具。
- 充当持续集成服务器的一个前端，减少样板代码，结合持续集成和基于 shell 的测试方式。

使用 pip 进行安装：

```
$ pip install tox
```

系统管理

本节介绍的工具服务器自动化、系统监控或者工作流管理用于管理和监控系统。

Travis-CI

Travis-CI 是一个分布式持续集成服务，免费为开源项目构建测试。它提供多个工作实例来运行 Python 测试，并且与 GitHub 无缝集成。甚至可以对你的 PR（Pull Request）[2] 进行评论，注明这组变更是否破坏了构建。因此如果你的代码托管在 GitHub 上，那么从 Travis-CI 着手引入持续集成是一个简单的好途径。Travis-CI 可以在运行 Linux、Mac OS X 或 iOS 的虚拟机上构建代码。

在代码库中添加一个 .travis.yaml 文件，如下所示。

```
language: python
python:
    - "2.6"
    - "2.7"
    - "3.3"
    - "3.4"
script: python tests/test_all_of_the_units.py
branches:
    only:
        - master
```

这个脚本会在所有罗列出来的 Python 版本上测试你的项目，并且仅构建主分支。还有很多其他选项可以启用，比如在一些步骤之前或之后发送通知。Travis-CI 文档（*http://about.travis-ci.org/docs/*）解释了所有这些选项，并且非常详细。要想在 Travis-CI 中使用 Tox，可以在代码仓库中添加一个 Tox 脚本，并修改 script 那一行。

```
install:
    - pip install tox
```

2 在 GitHub 上，用户可以向某个代码库提交 PR，通知代码库持有者有变更请求。

```
script:
    - tox
```

为项目激活测试，要先用自己的 GitHub 账号登录 Travis-CI 站点（*https://travis-ci.org/*），然后在配置中心激活项目。自此，项目每次推送更新到 GitHub 都会自动运行测试。

Jenkins

Jenkins CI 是目前最流行的可扩展持续集成引擎。可运行在 Windows、Linux 或 Mac OS X 上，也可以作为插件集成进现存的每种源码管理（SCM）工具。Jenkins 是一个 Java servlet（相当于一个 Python WSGI 应用），自带 servlet 容器，可以直接使用 java --jar jenkins.war 运行。详细信息请参考 *https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins* 上的 Jenkins 安装步骤，其中 Ubuntu 页面介绍了如何把 Jenkins 部署在一个 Apache 或 Nginx 反向代理的后面。

通过基于 Web 的控制面板或者基于 HTTP 协议的 RESTful API [3]（例如 *http://myServer:8080/api*）可以访问 Jenkins，这意味着可以从一个远程机器通过 HTTP 协议与 Jenkins 服务器通信。例如，查看 Apache 的 Jenkins 控制面板（*https://builds.apache.org/*）或 Pylons 项目的 Jenkins 控制面板（*https://wiki.jenkins.io/display/JENKINS/Step+by+step+guide+to+set+up+master+and+slave+machines*）。

与 Jenkins API 进行交互最常用的 Python 工具是 python-jenkins，由 OpenStack（*https://www.openstack.org/*）[4] 基础设施团队创建。多数 Python 用户会配置 Jenkins 来执行 Tox 脚本，作为构建流程的一部分。详细信息，请阅读文档搭配 Jenkins 来使用 Tox（*http://tox.readthedocs.io/en/latest/example/jenkins.html*），也可以阅读在多台构建机器上搭建 Jenkins 的指南（*https://wiki.jenkins.io/display/JENKINS/Step+by+step+guide+to+set+up+master+and+slave+machines*）。

Buildbot

Buildbot 是一个 Python 系统，自动化编译 / 测试，以验证代码变更。其工作模式类似 Jenkins：向源码版本控制服务器轮询获取代码变更，根据指令在多台计算机上构建测试代码（内建支持 Tox），然后告诉你发生了什么。它运行在一个 TwistedWeb 服务器的后面。其 Web 用户界面的样子，可参考 Chromium 开放 buildout 控制面板（Chromium 项目是 Chrome 浏览器背后的支撑）。

---

3 REST 是表述性状态传递（REpresentational State Transfer）的缩写。它不是一个标准或协议，而是 HTTP 1.1 标准创建期间提出的一组设计原则。Wikipedia 上提供了一组相关的 REST 架构约束（*https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_constraints*）。

4 OpenStack 为云网络、存储和计算提供免费软件，这样各种组织可以搭建私有云，或者搭建公有云提供给第三方付费使用。

因为 Buildout 是纯 Python 实现，所以可以通过 pip 进行安装。

```
$ pip install buildout
```

Buildout 0.9 版本有一个 REST API，因为它仍然处于 beta 版本状态，所以除非你特别指定版本号（例如，pip install buildbot==0.9.00.9.0rc1），否则没法使用它。大家都认为 Buildout 是最强大也是最复杂的持续集成工具。Buildout 的教程（*https://build.chromium.org/p/chromium/waterfall*）写得非常好，初学者可依照它进行学习。

服务器自动化

Salt、Ansible、Puppet、Chef 及 CFEngine 都是服务器自动化工具，为系统管理员提供一种优雅的方式来管理物理或虚拟机器集群。它们都可以管理 Linux、类 Unix 系统及 Windows 机器。当然我们偏爱 Salt 和 Ansible，因为这两者是用 Python 编写的，但它们算是后起新秀，其他可选方案使用更为广泛。下面介绍一下这些可选方案。

Docker 公司的人说：“他们期望 Docker 是 Salt、Ansible 及其他系统自动化工具的一个补充，而不是一种替代。”关于这个观点可以查看《Docker 如何和谐地进入 DevOps 领域》（*https://stackshare.io/posts/how-docker-fits-into-the-current-devops-landscape*）一文。

Salt

Salt 将其主节点称为 master，将其代理节点称为 minion 或 minion 主机。其主要设计目标是速度，默认使用 ZeroMQ 进行网络数据传输，主节点和 minion 节点之间使用 TCP 连接。Salt 团队成员甚至还实现了自己的（可选的）传输协议 RAET（*https://github.com/saltstack/raet*），比 TCP 更快，丢包率比 UDP 小。

Salt 支持 Python 2.6 和 2.7，可以通过 pip 安装。

```
$ pip install salt # 尚未支持 Python 3...
```

配置一个 master 服务器和任意数量 minion 主机之后，就可以在 minion 上运行任意的 shell 命令或使用复杂命令的预建模块。下面的命令会使用 salt test 模块中的 ping 来罗列出所有可用的 minion 主机。

```
$ salt '*' test.ping
```

可以通过匹配 minion ID，或使用 grains 系统（该系统使用静态的主机信息，如操作系统版本或 CPU 架构，为 Salt 模块提供一种主机分类法）来过滤 minion 主机。例如，下

面的命令使用 grains 系统仅罗列出运行 CentOS 可用的 minion。

```
$ salt -G 'os:CentOS' test.ping
```

Salt 也提供一个状态系统。状态可被用于配置 minion 主机。例如，在命令一个 minion 主机读取下面的状态文件后，它会安装并启动 Apache 服务器。

```
apache:
    pkg:
        - installed
    service:
        - running
        - enable: true
        - require:
            - pkg: apache
```

状态文件可使用 YAML 格式编写，通过 Jinja2 模板系统进行增强，或者可以是纯 Python 模块。详细信息，可参考 Salt 文档（*https://docs.saltstack.com/en/latest/*）。

Ansible

相比其他系统自动化工具，Ansible 最大的优势是不要求在客户端上安装 Python 以外的任何东西。所有其他方案[5] 都会在客户端上持续运行一个后台程序轮询主节点。playbook 是 Ansible 的配置、部署及编排文档，以 YAML 格式编写，并使用 Jinja2 来实现模板化。Ansible 支持 Python 2.6 和 2.7，可通过 pip 安装。

```
$ pip install ansible # 尚未支持 Python 3...
```

Ansible 需要一个清单文件来描述它能访问哪些主机。下面的代码以一个主机和 playbook 为例，用 ping 命令找出清单文件里的所有主机。清单文件（hosts.yml）示例如下。

```
[server_name]
127.0.0.1
```

playbook（ping.yml）示例如下。

```
---
- hosts: all

  tasks:
    - name: ping
      action: ping
```

5 Salt-SSH 是 Salt 架构的一个替代方案，其创建可能是为了响应某些用户的请求。它是从 Salt 中衍生出的一个类 Ansible 的可选方案。

运行 playbook（ping.yml）。

```
$ ansible-playbook ping.yml -i hosts.yml --ask-pass
```

Ansible playbook 用 ping 命令找出 hosts.yaml 文件中指定的所有服务器。使用 Ansible 也可以对服务器进行分组，关于 Ansible 的更多信息，请阅读 Ansible 文档（*http://docs.ansible.com/*）。Servers for Hackers 网站提供的 Ansible 教程也是一份详细的入门材料（*https://serversforhackers.com/c/an-ansible-tutorial*）。

Puppet

Puppet 以 Ruby 编写实现，提供一种自己特有的配置语言 PuppetScript。它有一个特定的服务器——Puppet 主节点（Puppet Master）——负责编排代理（Agent）节点。模块是一些短小的、可分享的代码单元，用于自动化或定义一个系统的状态。Puppet Forge 是一个模块仓库，其中的模块均是由社区为 Puppet 开源版和企业版编写的。

代理节点会把系统的一些基本信息（例如操作系统、内核、架构、IP 地址及主机名）发送给 Puppet 主节点。Puppet 主节点则根据这些信息编制一个目录，先指明应该如何配置每个节点，然后把目录发送给节点。代理强制执行目录中规定的变更，并返回一个报告给 Puppet 主节点。

facter 是 Puppet 提供的一个有趣的工具，可以抽取系统的一些基本事实信息。在编写 Puppet 模块时，这些事实信息可以作为变量被引用。

```
$ facter kernel
Linux
$
$ facter operatingsystem
Ubuntu
```

在 Puppet 中编写模块非常简单：若干 Puppet 清单（扩展后缀为 *.pp 的文件）一起组成 Puppet 模块。下面是一个 Puppet 的 Hello World 示例。

```
notify { 'Hello World, this message is getting logged into the agent node':

    #As nothing is specified in the body, the resource title
    #is the notification message by default.
}
```

下面是另一个例子，包含基于系统的逻辑。引用系统的事实信息需要在变量名前加一个 $ 符号前缀。例如，$hostname 或者下例中的 $operatingsystem。

```
notify{ 'Mac Warning':
```

```
    message => $operatingsystem? {
        'Darwin' => 'This seems to be a Mac.',
        default => 'I am a PC.',
    }
}
```

Puppet 有多种资源类型，不过 package-file-service 范式足够满足多数配置管理场景。下面的 Puppet 代码可以确保系统已安装 OpenSSH-Server 软件包，并且每次 sshd 配置变更时都会通知 sshd 服务（SSH 服务后台程序）重启。

```
package { 'openssh-server':
    ensure => installed,
}

file { '/etc/ssh/sshd_config':
    source => 'puppet:///modules/sshd/sshd_config',
    owner   => 'root',
    group   => 'root',
    mode    => '640',
    notify  => Service['sshd'],  # 任何时候编辑这个文件，sshd 都会重启
    require => Package['openssh-server'],
}

service { 'sshd':
    ensure     => running,
    enable     => true,
    hasstatus  => true,
    hasrestart => true,
}
```

详细信息，请参考 Puppet Labs 网站上的文档（*http://docs.puppetlabs.com/*）。

Chef

如果选择 Chef 进行配置管理，那么你将主要使用 Ruby 来编写基础设施代码。Chef 类似于 Puppet，但与 Puppet 的设计理念相反：Puppet 提供一个框架以灵活性为代价简化工作，而 Chef 几乎没有提供任何框架的能力，其目标是极好的可扩展性，也因此更难于使用。

Chef 客户端运行在基础设施的每个节点上，定期与 Chef 服务器端进行核对，确保系统始终一致并表现期望的状态。每个 Chef 客户端都能独立配置自己。这种分布式方式让 Chef 得以成为一个可伸缩的自动化平台。

Chef 通过使用自定义食谱 recipe（配置元素）进行工作，食谱记录在 cookbook 中。cookbook 基本上是基础设施配置选项包，通常存储在 Chef 服务器端。请阅读

DigitalOcean 网站提供的关于 Chef 的系列教程（*https://www.digitalocean.com/community/tutorials/how-to-install-a-chef-server-workstation-and-client-on-ubuntu-vps-instances*），学习如何创建一个简单的 Chef 服务器。

使用 knife 命令（*https://docs.chef.io/knife.html*）来创建一个简单的 cookbook。

```
$ knife cookbook create cookbook_name
```

Andy Gale 的《Chef 入门》（*http://gettingstartedwithchef.com/first-steps-with-chef.html*）一文对于 Chef 初学者是一个很好的起点。可以在 Chef 超级市场（*https://supermarket.chef.io/cookbooks*）上找到很多社区贡献的 cookbook。它们对于你编写自己的 cookbook 来说是一个很好的起点。更多信息，请阅读 Chef 文档（*https://docs.chef.io/*）。

CFEngine

CFEngine 内存占用很小，用 C 语言编写而成。其主要设计目标是在发生失败时保持健壮性，通过分布式网络（对比于主节点 / 客户端节点架构）中运行的自主代理节点来实现这个目标，自主代理节点之间使用允诺理论（*https://en.wikipedia.org/wiki/Promise_theory*）相互交流。如果想使用无中心领导的架构，则可以尝试这个系统。

系统和任务监控

下面介绍的几个库都是帮助系统管理员监控运行任务的，但应用场景不同：psutil 将一些可以通过 Unix 工具获取的信息以 Python 接口提供出来，Fabric 可以简化命令定义的过程并通过 SSH 在一组远程主机上执行，Luigi 则可以实现像链式 Hadoop 命令一样调度并监控长期运行批量进程。

psutil

psutil 是一个各种系统信息（例如，CPU、内存、磁盘、网络、用户及进程）的跨平台（包括 Windows）接口。我们习惯于通过 Unix 命令（例如，top、ps、df 及 netstat）获取的信息在 Python 中也能获得，使用 pip 安装这个库。

```
$ pip install psutil
```

下面是一个系统过载监控的例子（网络或 CPU，如果有一个测试失败，则会发送一封邮件）。

```
# 获取系统信息的函数
from psutil import cpu_percent, net_io_counters
# 暂停一下的函数
from time import sleep
```

```
# 用于 Email 服务的包
import smtplib
import string

MAX_NET_USAGE = 400000
MAX_ATTACKS = 4
attack = 0
counter = 0
while attack <= MAX_ATTACKS:
    sleep(4)
    counter = counter + 1
    # 检查 CPU 使用率
    if cpu_percent(interval=1) > 70:
        attack = attack + 1
    # 检查网络使用率
    neti1 = net_io_counters()[1]
    neto1 = net_io_counters()[0]
    sleep(1)
    neti2 = net_io_counters()[1]
    neto2 = net_io_counters()[0]
    # 计算每秒字节数
    net = ((neti2+neto2) - (neti1+neto1))/2
    if net > MAX_NET_USAGE:
        attack = attack + 1
    if counter > 25:
        attack = 0
        counter = 0

# 如果 attack 的值高于 4，则发送一封非常重要的邮件
TO = "you@your_email.com"
FROM = "webmaster@your_domain.com"
SUBJECT = "Your domain is out of system resources!"
text = "Go and fix your server!"
BODY = string.join(
    ("From: %s" %FROM,"To: %s" %TO,"Subject: %s" %SUBJECT, "",text), "\r\n")
server = smtplib.SMTP('127.0.0.1' )
server.sendmail(FROM, [TO], BODY)
server.quit()
```

glances 是 psutil 的一个优秀用例，它是一个完整的终端应用，其行为表现像一个高度扩展后的 top（以 CPU 使用量或一种用户指定的顺序罗列出正在运行的进程），也提供“客户端 - 服务器端”模式的监控能力。

Fabric

Fabric 是一个简化系统管理任务的库，允许通过 SSH 连接到多台主机，并在每台主机上执行任务。这对于系统管理或应用部署而言非常便利。使用 pip 安装 Fabric。

```
$ pip install fabric
```

下面是一个完整的 Python 模块，它定义了两个 Fabric 任务：memory_usage 和 deploy。

```
# fabfile.py
from fabric.api import cd, env, prefix, run, task

env.hosts = ['my_server1', 'my_server2'] # SSH连接目标

@task
def memory_usage():
    run('free -m')

@task
def deploy():
    with cd('/var/www/project-env/project'):
        with prefix('. ../bin/activate'):
            run('git pull')
            run('touch app.wsgi')
```

with 语句只是嵌套命令，这样到最后对每个主机来说 deploy() 如下。

```
$ ssh hostname cd /var/www/project-env/project && ../bin/activate && git
pull
$ ssh hostname cd /var/www/project-env/project && ../bin/activate && \
> touch app.wsgi
```

将上面的代码保存在名为 fabfile.py 的文件中（fab 命令默认查找的模块名），这样就可以使用 memory_usage 任务来检查内存使用率了。

```
$ fab memory_usage
[my_server1] Executing task 'memory'
[my_server1] run: free -m
[my_server1] out:             total    used     free   shared  buffers
cached
[my_server1] out: Mem:         6964    1897     5067        0      166
222
[my_server1] out: -/+ buffers/cache:   1509     5455
[my_server1] out: Swap:           0       0        0

[my_server2] Executing task 'memory'
```

```
[my_server2] run: free -m
[my_server2] out:             total    used     free   shared  buffers
cached
[my_server2] out: Mem:         1666     902      764        0      180
572
[my_server2] out: -/+ buffers/cache:    148     1517
[my_server2] out: Swap:         895       1      894
```

并且可以这样来部署：

```
$ fab deploy
```

其他特性包括：并行执行、与远程程序交互及主机分组。Fabric 文档（*http://docs.fabfile.org/*）中的示例适合参考学习。

Luigi

Luigi 是一个任务流水线管理工具，由 Spotify 公司发布。Luigi 可以帮助开发者管理大型、长期运行批量任务的流水线，将 Hive 查询、数据库查询、Hadoop、Java 任务、pySpark 任务及自己编写的任务拼接在一起。这些任务不一定要都是大数据应用，Luigi 的 API 允许调度任何任务。因为 Spotify 使用 Luigi 在 Hadoop 上执行任务，所以它们已在 luigi.contrib 包中提供了所有工具。使用 pip 安装 Luigi：

```
$ pip install luigi
```

它包含一个 Web 界面，用户可以在其中过滤任务、查看流水线工作流的依赖图及进度。Luigi 的 GitHub 代码库中有一些 Luigi 任务示例，也可以阅读 Luigi 文档（*http://luigi.readthedocs.io/en/stable/*）进行学习。

加速

本节罗列了 Python 社区中程序速度优化最常见的几种方式。如果你已经通过剖析代码（*https://docs.python.org/3.5/library/profile.html*）、对比代码片段的不同实现方式的执行速度（*https://docs.python.org/3.5/library/timeit.html*）直接从 Python 语言层面尽可能地提升性能，那么进一步优化可以考虑表 8-1 中的方案。

你可能已经听说过全局解释器锁（GIL），Python 的 C 实现以此允许多个线程同时进行操作。Python 的内存管理并非线程完全安全的，因此需要 GIL 来防止多个线程同时运行同一段代码。

GIL 经常被认为是 Python 的一大限制，实际上并非如此，仅在进程是计算密集型时（NumPy 和加密解密库都是针对这种情况，以 C 语言重写代码，以 Python 绑定提供接口）

GIL 才是一个障碍。对于其他场景（比如网络 I/O 或文件 I/O），GIL 的瓶颈在于单线程等待 I/O 时的代码阻塞，可以使用多线程或事件驱动编程模式来解决阻塞问题。

在 Python 2 中，某些库有快和慢两个实现版本：StringIO 和 cStringIO、ElementTree 和 cElementTree。C 实现版本更快，但需要显式地导入。自 Python 3.3 起，常规版本导入的就是更快的实现版本，不再有 C 前缀的库。

表8-1　加速方案

方案	许可证	使用理由
threading	PSFL	• 允许创建多个执行线程 • threading 库（使用 CPython 时，由于存在 GIL）没法利用多进程；在某个线程阻塞时切换到另一线程，当瓶颈在于等待 I/O 这类阻塞任务上时，这一点非常有用 • Python 的其他实现没有 GIL，比如 Jython 和 IronPython
multiprocessing/ subprocess	PSFL	• multiprocessing 库中的工具绕过了 GIL，可以实际派生出其他 Python 进程 • subprocess 库则允许启动多个命令行进程
PyPy	MIT 许可证	• 一个 Python 解释器（最新版本是 Python 2.7.10 和 3.2.5），提供尽可能即时编译成 C 代码的能力 • 速度优化不费吹灰之力：无须另外编写代码，通常增速明显 • 一般情况下它都可以替代 CPython，不过 PyPy 环境下可以使用的 C 扩展库必须用 CFFI 来提供 Python 语言接口，或者在 PyPy 兼容性列表（*http://pypy.org/compat.html*）中
Cython	Apache 许可证	• 提供两种方式来静态编译 Python 代码：方式一是使用一种注解语言，Cython（*.pyd） • 方式二是静态地编译纯 Python 代码，并使用 Cython 提供的装饰器来指定对象类型
numba	BSD 许可证	• 同时提供一个静态（通过它的 pycc 工具）编译器和一个即时编译器，将代码编译成使用 NumPy 数组的机器码 • 要求 Python 2.7 或 3.4 以上版本、llvmlite 库（*http://llvmlite.pydata.org/en/latest/install/index.html*）及其依赖、LLVM（低层次虚拟机）编译器基础设施
weave	BSD 许可证	• 提供一种方式将若干行 C 代码编织进 Python 代码中，不过应该仅当你已经在使用 Weave 了才使用它 • 否则，还是使用 Cython，因为 Weave 已被废弃

续表

方案	许可证	使用理由
PyCUDA/gnumpy/TensorFlow/Theano/PyOpenCL	MIT/ 经过修改的 BSD/BSD/BSD/MIT 许可证	• 这些库提供了不同的 NVIDIA CPU 使用方式，如果你已经安装过其中某个库，并且也可以安装 NVIDIA 的 CUDA 工具链（*http://docs.nvidia.com/cuda*），那么推荐使用这一方案 • PyOpenCL 可以使用 NVIDIA 处理器之外的处理器 • 它们各自都有不同的应用场景，例如 gnumpy 旨在成为 NumPy 的简易替代品
直接使用 C/C++ 库		• 虽然需要花时间编写 C/C++ 代码，但是提升了速度，一切付出都值得

《编写地道的 Python 代码》一书的作者 Jeff Knupp，就如何绕过 GIL 写过一篇博文（*https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/*），他引用了 David Beazley 对这一话题的深刻见解[6]。

下面进一步讨论表 8-1 中列举的 threading 及其他优化方案。

threading

Python 的 threading 库允许开发者创建多个线程。但由于 GIL 的存在（至少在 CPython 中），每个 Python 解释器中同时仅有一个 Python 进程在运行，这意味着仅当至少有一个线程发生阻塞（例如，正在读写 I/O）时，才可能有性能提升。针对 I/O 读写，另一个优化方案是使用事件处理，请阅读《Python 标准库中的高性能网络编程工具》一文中对 asyncio 的介绍。

使用多线程，当 Python 内核发现某个线程正阻塞在 I/O 读写上时，会切换另一个线程来使用处理器，直到这个线程也被阻塞或结束。当启动多线程后，这一切都是自动进行的。Stack Overflow 上有一些 threading 的优秀用例（*https://bit.ly/threading-in-python*），每周一 Python 模块系列教程中也有一篇关于 threading 模块介绍的文章（*https://pymotw.com/2/threading/*），也可以阅读标准库中的 threading 文档（*https://docs.python.org/3/library/threading.html*）进行学习。

6 David Beazley 写过一篇解释 GIL 的运作原理的指南（*http://www.dabeaz.com/python/UnderstandingGIL.pdf*）。他还论述过 Python 3.2 中的新 GIL（*http://www.dabeaz.com/python/NewGIL.pdf*）。他的结论是：若想 Python 应用的性能最大化，必须深入理解 GIL、GIL 如何影响特定应用、有多少个 CPU 核心可供利用，以及应用的瓶颈在哪。

multiprocessing

Python 标准库中的 multiprocessing 模块提供一种绕过 GIL 的方法——启动额外的 Python 解释器。不同进程之间的通信方式包括：使用 multiprocessing.Pipe 或 multiprocessing.Queue，或者通过 multiprocessing.Array 和 multiprocessing.Value 共享内存。这些方式内部都实现了自动加锁。共享数据要谨慎，这些对象都实现了锁机制来防止不同进程同时访问同一个数据。

下面的代码演示了使用一个工作进程池获得的性能提升并非总是正比于使用的工作进程数的案例，在节省的计算时间和启动新解释器花费的时间之间需要做权衡。这个例子使用蒙特卡洛方法（绘制随机数）来估算 Pi 值[7]。

```
>>> import multiprocessing
>>> import random
>>> import timeit
>>>
>>> def calculate_pi(iterations):
...     x = (random.random() for i in range(iterations))
...     y = (random.random() for i in range(iterations))
...     r_squared = [xi**2 + yi**2 for xi, yi in zip(x, y)]
...     percent_coverage = sum([r <= 1 for r in r_squared]) / len(r_squared)
...     return 4 * percent_coverage
...
>>>
>>> def run_pool(processes, total_iterations):
...     with multiprocessing.Pool(processes) as pool:    ❶
...         # 将总迭代次数分配到多个进程完成
...         iterations = [total_iterations // processes] * processes     ❷
...         result = pool.map(calculate_pi, iterations)  ❸
...     print("%0.4f" % (sum(result) / processes), end=', ')
...
>>>
>>> ten_million = 10000000         ❹
>>> timeit.timeit(lambda: run_pool(1, ten_million), number=10)
3.141, 3.142, 3.142, 3.141, 3.141, 3.142, 3.141, 3.141, 3.142, 3.142,
134.48382110201055                 ❺
>>>                                ❻
>>> timeit.timeit(lambda: run_pool(10, ten_million), number=10)
3.142, 3.142, 3.142, 3.142, 3.142, 3.142, 3.141, 3.142, 3.142, 3.141,
74.38514468498761                  ❼
```

7 该方法的完整推导过程参见 *http://bit.ly/monte-carlo-pi*。具体来说，就是不断地向一个 2×2 的正方形中掷飞镖，正方形内有一个半径为 1 的圆。假设飞镖命中正方形内任意位置的概率相等，那么命中位置在圆内的百分比是 Pi/4，也就是说命中圆内位置的百分比的 4 倍等于 Pi。

❶ 在一个上下文管理器中使用 multiprocessing.Pool，确保进程池仅被创建它的进程使用。

❷ 总的迭代次数应该保持一致，只不过会被分配到不同数量的进程中执行。

❸ pool.map() 会创建多个进程，每个进程一次处理 iterations 列表中的一个元素，不过在进程池初始化时已声明进程数量上限（multiprocessing.Pool(processes)）。

❹ 第一次 timeit 尝试仅使用 1 个进程。

❺ 单个进程运行 1 千万次迭代，重复 10 次，花费 134 秒。

❻ 第二次 timeit 尝试使用 10 个进程。

❼ 10 个进程，每个运行 1 百万次迭代，重复 10 次，花费 74 秒。

上面的例子说明了创建多进程有时间开销，但 Python 中的多进程工具强大且成熟。更多信息请阅读标准库中的 multiprocessing 文档（*https://docs.python.org/3.5/library/multiprocessing.html*），也推荐阅读 Jeff Knupp 写的关于如何绕过 GIL 的博文（*https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/*），文中有若干段落论及 multiprocessing。

subprocess

Python 2.4 在标准库中引入 subprocess 库，其定义见 PEP 324（*https://www.python.org/dev/peps/pep-0324*）。这个库发起一个系统调用（例如 unzip 或 curl），就像从命令行调用命令一样（默认不会调用系统 shell），但开发者可以基于 subprocess 的输入输出管道选择做些事情。我们推荐 Python 2 用户使用 subprocess32 的更新版本，该版本修复了若干 bug，可使用 pip 进行安装。

```
$ pip install subprocess32
```

每周一 Python 模块系列教程上有一篇非常好的关于 subproces 的文章（*https://pymotw.com/2/subprocess/*），推荐阅读。

PyPy

PyPy 是 Python 的一个纯 Python 实现。使用 PyPy 代码不需要做任何变动，就能运行得更快。在选择任何加速其他方案之前应该先尝试一下 PyPy。

无法使用 pip 获取 PyPy，因为它实际上是 Python 的另一个实现。在 PyPy 的下载页面（*http://pypy.org/download.html*）下载对应操作系统和正确 Python 版本的 PyPy。

下面运行的程序对 David Beazley 编写的计算密集型测试代码稍做改动，为多轮测试添

加了一个循环。从结果可以看到 PyPy 与 CPython 之间的性能差异。首先使用 CPython 来运行脚本。

```
$ # CPython
$ ./python -V
Python 2.7.1
$
$ ./python measure2.py
1.06774401665
1.45412397385
1.51485204697
1.54693889618
1.60109114647
```

然后使用 PyPy 来运行同一个脚本。

```
$ # PyPy
$ ./pypy -V
Python 2.7.1 (7773f8fc4223, Nov 18 2011, 18:47:10)
[PyPy 1.7.0 with GCC 4.4.3]
$
$ ./pypy measure2.py
0.0683999061584
0.0483210086823
0.0388588905334
0.0440690517426
0.0695300102234
```

只是下载使用了 PyPy，平均执行时间就从约 1.4 秒降到约 0.05 秒，快了 20 倍多。虽然有时代码的执行速度不会有倍数增长，但是通常都会有一个很大的速度提升，而代价只是使用 PyPy 解释器。如果想要自己编写的 C 扩展库与 PyPy 兼容，请遵从 PyPy 的建议（*http://pypy.org/compat.html*），使用 CFFI 来为其编写 Python 绑定，而不是标准库里的 ctypes。

Cython

不是所有使用 C 扩展的库都适用 PyPy。应对这样的情况，可以选择 Cython，Cython（它不同于 CPython（Python 的标准 C 实现））实现了 Python 语言的一个超集，允许为 Python 编写 C 和 C++ 模块。Cython 也允许从编译好的 C 库中调用函数，并提供一个上下文 nogil，在允许运行某段代码时释放 GIL，前提是这段代码不会以任何方式操作 Python 对象。使用 Cython 可以利用 Python 变量和操作的强类型[8]特点。

8 一门编程语言兼具强类型和动态类型特征是可能的，Stack Overflow 上的讨论对此有解释（*http://stackoverflow.com/questions/11328920/*）。

应用 Cython 配合强类型的一个示例如下。

```
def primes(int kmax):
""" 额外使用 Cython 的关键字配合计算素数 """

    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

素数寻找算法的实现相比下面的纯 Python 实现，使用了一些额外的关键字。

```
def primes(kmax):
""" 以标准 Python 语法计算素数 """

    p = range(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

注意，Cython 版本中声明了整数和整数数组将被编译成 C 类型，同时也创建了一个

Python 列表。

```
# Cython 版本
def primes(int kmax):                                    ❶
""" 额外使用 Cython 的关键字配合计算素数 """
    cdef int n, k, i                                     ❷
    cdef int p[1000]                                     ❸
    result = []
```

❶ 类型声明为一个整数。

❷ 变量 n、k 和 i 声明为整数。

❸ 为 p 预分配一个长度为 1 000 的整数数组。

区别在哪？在 Cython 版本中，可以看到变量类型和整数数组的声明类似于标准 C。例如，相比没有类型提示，“`cdef int n, k, i`”一行中额外的（整数）类型声明，让 Cython 编译器可以生成更高效的 C 代码。Cython 语法不与标准 Python 兼容，代码不能存为 *.py 文件，而是存为 *.pyx 文件。

那么运行速度上又有什么区别呢？我们来试试看！

```
import time
# 激活 pyx 编译器
import pyximport                                ❶
pyximport.install()                             ❷
# Cython 实现的素数寻找算法
import primesCy
# Python 实现的素数寻找算法
import primes

print("Cython:")
t1 = time.time()
print primesCy.primes(500)
t2 = time.time()
print("Cython time: %s" %(t2-t1))
print("")
print("Python")
t1 = time.time()                                ❸
print(primes.primes(500))
t2 = time.time()
print( “Python time: {}".format(t2-t1))
```

❶ pyximport 模块允许导入 *.pyx 文件（例如 primesCy.pyx），文件内容为 Cython 编译版的 primes 函数。

❷ pyximport.install() 命令会让 Python 解释器直接启动一个 Cython 编译器来生成 C 代码，C 代码会被自动编译成一个 *.so 的 C 库文件。这样 Cython 就可以将库文件导入 Python 代码中，简单而高效。

❸ 使用 time.time() 函数可以比较两个不同的函数调用，找到 500 个素数的耗用时间。在一个标准 notebook 上（双核 AMD E-450 1.6GHz），测量到的值如下：

```
Cython time: 0.0054 seconds

Python time: 0.0566 seconds
```

另外，在一台嵌入式 ARM BeagleBone 机器上运行输出如下：

```
Cython time: 0.0196 seconds

Python time: 0.3302 seconds
```

Numba

Numba 是一个 NumPy 相关的 Python 编译器（专用即时（JIT）编译器），通过特殊的装饰器将注解的 Python（及 NumPy）代码编译为 LLVM。简单来讲，就是 Numba 使用 LLVM 将 Python 代码编译成机器码，可以在运行时原生执行。

如果你使用 Anaconda，则推荐使用 conda install numba 安装 Numba。否则，手动安装。安装 Numba 之前必须已安装 NumPy 和 LLVM。确认需要的 LLVM 版本（可在 llvmlite 的 GitHub 页面（*https://github.com/numba/llvmlite*）上查看），并下载与操作系统匹配的版本。LLVM 版本如下。

- LLVM Windows 构建版本（*http://llvm.org/builds/*）。
- LLVM Debian/Ubuntu 构建版本（*http://llvm.org/apt/*）。
- LLVM Fedora 构建版本（*https://apps.fedoraproject.org/packages/llvm*）。
- 对于其他 Unix 系统，如何从源码构建 LLVM，请阅读 *http://ftp.math.utah.edu/pub/llvm/* 网页上关于构建 Clang+LLVM 编译器的讨论。
- 在 Mac OS X 上，使用 brew install homebrew/version/llvm37（或任何其他的最新版本）进行安装。

一旦安装了 LLVM 和 NumPy，即可使用 pip 安装 Numba。或许需要提供一个环境变量 LLVM_CONFIG 设置恰当的路径，帮助安装器找到 llvm-config 文件。

```
$ LLVM_CONFIG=/path/to/llvm-config-3.7 pip install numba
```

在代码中使用命令 numba，只需装饰一下函数定义。

```
from numba import jit, int32

@jit                                    ❶
def f(x):
    return x + 3

@jit(int32(int32, int32))               ❷
def g(x, y):
    return x + y
```

❶ 不带参数，@jit 装饰器进行惰性编译，自己决定是否优化该函数，以及如何优化。

❷ 若想及早编译，请指定类型。该函数会被编译为指定的特化形式，不再允许其他形式，返回值和两个参数的类型均为 numba.int32。

@jit 装饰器提供一个 nogil 标记，允许代码忽略全局解释器锁，numba.pycc 模块可用于提前编译代码。更多详情，请阅读 Numba 的用户手册（*http://numba.pydata.org/numba-doc/latest/user/*）。

GPU 的相关程序库

Numba 可选择构建支持运行在计算机的图形处理单元（GPU）上，GPU 是一种专为现代视频游戏快速并行计算而优化的芯片。使用该能力，需要英伟达的 GPU，并安装好英伟达的 CUDA 工具包，遵照文档配合 GPU 使用 Numba 的 CUDA JIT（*http://numba.pydata.org/numba-doc/0.13/CUDAJit.html*）。

除 Numba 之外，另一个支持 GPU 计算能力的流行库是 TensorFlow。它是 Google 基于 Apache 2.0 许可证发布的。它为快速的矩阵数学运算提供张量（多维矩阵）及一种将张量操作串接在一起的方式。目前它仅能在 Linux 操作系统上使用 GPU，安装说明如下所示。

- 安装支持 GPU 的 TensorFlow，参见 *http://bit.ly/tensorflow-gpu-support*。
- 安装不支持 GPU 的 TensorFlow，参见 *http://bit.ly/tensorflow-no-gpu*。

对于不在 Linux 上运行的场景，在 Google 发布 TensorFlow 之前，来自蒙特利尔大学的 Theano 是基于 GPU 进行矩阵数学运算事实上的 Python 标准库。Theano 目前仍活跃在开发中，*http://deeplearning.net/software/theano/tutorial/using_gpu.html* 上专门说明了如何使用 GPU。Theano 支持 Windows、Mac OS X 和 Linux 操作系统，可通过 pip 进行安装使用：

```
$ pip install Theano
```

对于更低层次的 GPU 应用场景，可以尝试 PyCUDA。

最后说明一下，没有英伟达 GPU 的开发者可以使用 PyOpenCL，它是 Intel OpenCL 库的 Python 包装库，兼容许多不同的硬件集。

与 C/C++/FORTRAN 库进行交互

下面几节介绍的程序库差别很大：CFFI 和 ctypes 是 Python 库，F2PY 用于与 FORTRAN 代码交互的场景，SWIG 可让 C 对象为多种编程语言所用（并不只是 Python），Boost.Python 是一个 C++ 库，可以将 C++ 对象暴露给 Python，反之亦可。表 8-2 将提供更详细的信息。

表8-2　C/C++/FORTRAN的Python接口库

库名称	许可证	使用理由
CFFI	MIT 许可证	• 与 PyPy 的兼容性最好 • 允许在 Python 中编写 C 代码，这些 C 代码会被编译构建为一个带 Python 绑定的共享 C 库
ctypes	Python 软件基金会许可证	• 属于 Python 标准库 • 让开发者能够包装非其编写或其无法掌控的已有 DLL 或共享对象库 • 与 PyPy 的兼容性仅次于 CFFI
F2PY	BSD 许可证	• 让开发者能够使用 FORTRAN 库 • 因为 F2PY 是 NumPy 的一部分，所以得使用 NumPy 才行
SWIG	GPL 许可证（不限制输出）	• 提供一种方式自动生成多种编程语言的程序库，使用一种特殊的文件格式：既非 C 又非 Python
Boost.Python	Boost 软件许可证	• 不是一个命令行工具，而是一个 C++ 库，可以被包含在 C++ 代码中，用于识别哪些对象要暴露给 Python

C 跨语言接口

CFFI 包提供一种简单机制方便 CPython 和 PyPy 与 C 交互。因为 CFFI 与 CPython 和 PyPy 的兼容性最好，而被 PyPy 推荐使用。它支持两种模式：一种是内联应用二进制接口（ABI）兼容模式（参考下面的代码示例），允许动态加载并运行可执行模块中的函数（本质上是通过 LoadLibrary 或 dlopen 暴露相同的功能），另外一种是 API 模式，允许构建 C 扩展模块[9]（译注：最新的 CFFI 支持 4 种模式）。

9　编写 C 扩展时要特别注意（*https://docs.python.org/3/c-api/init.html#threads*），确保向解释器注册了线程。

使用 pip 安装 CFFI：

```
$ pip install cffi
```

下面是一个与 ABI 交互的示例。

```
from cffi import FFI
ffi = FFI()
ffi.cdef("size_t strlen(const char*);")        ❶
clib = ffi.dlopen(None)                                    ❷
length = clib.strlen("String to be evaluated.")     ❸
# 输出：23
print("{}".format(length))
```

❶ 此处字符串可借用自某个 C 头文件中的某个函数声明。

❷ 打开共享库（*.DLL 或 *.so）。

❸ 把 clib 当作一个 Python 模块，使用点号调用定义的函数即可。

ctypes

ctypes 是 CPython 中与 C/C++ 事实上进行交互的标准库，并且归属 Python 官方标准库。它为大多数操作系统原生 C 接口（例如，Windows 上的 kernel32，*nix 上的 libc）提供完整的访问能力，并且支持在运行时加载动态库——共享对象库（*.so）或 DLL 库——并与其交互。ctypes 带来了大量类型用于与系统 API 交互，让开发者可以轻松地定义自己的复杂类型，比如结构体和联合体，也允许按需修改填充和对齐这类东西。它使用起来可能有点烦琐（因为要额外输入那么多字符），但配合使用标准库里的 struct 模块，开发者可以完全控制如何将数据类型翻译成纯 C/C++ 方法可用的东西。

例如，名为 my_struct.h 的文件中有一个 C 结构体定义。

```
struct my_struct {
    int a;
    int b;
};
```

这个结构体在名为 my_struct.py 的文件中实现如下。

```
import ctypes
class my_struct(ctypes.Structure):
    _fields_ = [("a", c_int),
                ("b", c_int)]
```

F2PY

从 Fortran 到 Python 的接口生成器（F2PY）是 NumPy 的一部分，因此为获取它，可通过 pip 安装 NumPy。

```
$ pip install numpy
```

它提供一个多功能的命令行工具 f2py，可以以三种不同方式来使用，F2PY 快速上手指南（*http://docs.scipy.org/doc/numpy/f2py/getting-started.html*）中都有文档说明。如果开发者可以控制源码，则可以为 F2PY 添加一些特殊的指令注释，说明每个参数的意图（哪些是返回值，哪些是输入），然后运行 F2PY 即可。

```
$ f2py -c fortran_code.f -m python_module_name
```

若没法控制源码，F2PY 可以生成一个中间文件，扩展名为 *.pyf，开发者可以修改它来生成相同的结果。这样就分成了 3 步：

```
$ f2py fortran_code.f -m python_module_name -h interface_file.pyf        ❶
$ vim interface_file.pyf     ❷
$ f2py -c interface_file.pyf fortran_code.f     ❸
```

❶ 自动生成一个中间文件，定义 FORTRAN 函数签名和 Python 函数签名之间的接口。

❷ 编辑这个文件就可以正确地为输入输出变量加标签。

❸ 编译代码并构建扩展模块。

SWIG

简化的包装接口生成器（SWIG）支持包括 Python 在内的大量脚本语言。它是一个被广泛使用的命令行工具，从 C/C++ 头文件为解释型语言生成绑定。其使用步骤是，先使用 SWIG 从头文件自动生成一个后缀为 *.i 的中间文件，然后修改该文件来反映真正想要的接口，最后运行构建工具将代码编译成一个共享库。SWIG 教程（*http://www.swig.org/tutorial.html*）对如何完成这些工作有详细介绍。

虽然 SWIG 有一些限制（目前看来对于最新 C++ 特性的一个小子集存在一些问题，并且让大量使用模板的代码也能工作会有点烦琐），但只需一点点额外工作，SWIG 就能提供大量能力，并向 Python 暴露大量特性。此外，开发者可以轻松扩展 SWIG 生成的绑定（在接口文件中）来重载操作符和内建方法，并能高效地将 C++ 异常转换为可被 Python 捕获的类型。

下面的例子演示了如何重载 __repr__，假设代码引用自一个名为 MyClass.h 的文件：

```
#include <string>
class MyClass {
private:
    std::string name;
public:
    std::string getName();
};
```

下面是 myclass.i 文件的内容：

```
%include "string.i"

%module myclass
%{
#include <string>
#include "MyClass.h"
%}

%extend MyClass {
    std::string __repr__()
    {
        return $self->getName();
    }
}

%include "MyClass.h"
```

在 SWIG 的 GitHub 代码库中有更多的 Python 示例。如果系统有包管理器，则使用包管理器安装 SWIG（apt-get install swig、yum install swig.i386 或 brew install swig），否则打开 *http://www.swig.org/survey.html* 下载 SWIG，然后按照对应操作系统的安装说明进行安装。如果你的 Mac OS X 上没有 Perl 兼容的正则表达式库（PCRE），那么使用 Homebrew 安装：

```
$ brew install pcre
```

Boost.Python

Boost.Python 需要更多的手动工作来暴露 C++ 对象，它能提供 SWIG 提供的所有特性，并提供一些包装器实现在 C++ 中以 PyObject 的形式访问 Python 对象，也提供工具将 C++ 对象暴露给 Python。与 SWIG 不同的是，Boost.Python 是一个库，而不是一个命令行工具，无须创建不同格式的中间文件，完全直接以 C++ 编写。如果想使用这个方案，那么推荐打开 *http://bit.ly/boost-python-tutorial* 阅读 Boost.Python 提供的教程。

第 9 章

软件接口

本章先介绍如何使用 Python 从 Web API 获取信息（现在 Web API 常用于组织之间的信息分享），然后重点介绍很多组织在 Python 开发时通常会使用的那些基础设施网络通信工具。

在 Multiprocessing 一节讨论过 Python 对进程间管道和队列的支持。计算机之间的通信要求对话两端的计算机使用一组严格定义的协议，互联网遵从 TCP/IP 协议族（*https://en.wikipedia.org/wiki/Internet_protocol_suite*）[1]。开发者可以在套接字之上实现 UDP 协议（*https://pymotw.com/2/socket/udp.html*）。Python 提供一个名为 ssl 的库、套接字之上的 TLS/SSL 包装库，并提供 asyncio 来实现 TCP、UDP、TLS/SSL 及 subprocess 管道的异步传输（*https://docs.python.org/3/library/asyncio-protocol.html*）。

不过，多数情况下我们都会使用抽象层次更高的库。各种应用层协议实现的客户端程序库：ftplib、poplib、imaplib、nntplib、smtplib、telnetlib 及 xmlrpc。这些库都提供了常规的和 TLS/SSL 包装的客户端类（还有 urllib 可用于 HTTP 请求，不过对于多数应用场景我们都推荐使用 Requests 库）。

本章的第一节内容涵盖 HTTP 请求，以及如何从 Web 上的开放 API 获取数据。第二节是关于 Python 中数据序列化的一点题外话，第三节介绍企业级网络编程中常用的工具。如果某些东西仅在 Python 3 中可用，那么我们会尽量明确地指出来。如果你在使用 Python 2，找不到我们讲述的某个模块或类，则建议对照一下关于 Python 2 与 Python 3 标准库之间

1 TCP/IP（或者因特网协议）族在概念上分为四大部分。其一，链路层协议规定计算机和因特网之间获取信息的方式。在计算机中，这是网卡和操作系统的职责，Python 程序管不了这个。其二，网络层协议（IPv4、IPv6 等）控制如何把比特包从源地址传输到目的地址。Python 的 socket 库为此提供了标准配置选项。其三，传输层协议（TCP、UDP 等）指定两个传输端点之间如何通信。其相关配置选项也在 socket 库中。其四，应用层协议（FTP、HTTP 等）规定目标应用使用的数据应该是什么样的（例如，FTP 协议用于文件传输，HTTP 协议用于超文本传输），Python 标准库分模块各自实现最常用的应用层协议。

的变化清单，参见 *http://python3porting.com/stdlib.html*。

Web 客户端库

超文件传输协议（HTTP）是一种用于分布式协作超媒体信息系统的应用层协议，是万维网数据通信的基础。本节聚焦于如何使用 Requests 库从 Web 上获取数据。

Python 的标准 urllib 模块提供了常规开发需要的大多数 HTTP 能力，不过抽象不够，即使处理看上去很简单的任务（例如，从要求身份认证的 HTTPS 服务器获取数据）也要求相当多的编码工作。urllib.request 模块的文档推荐使用 Requests 库来替代自己。

Requests 可以完成各种 Python HTTP 请求相关的工作，无缝集成 Web 服务。无须手动把查询字符串添加到 URL 或对 POST 数据进行表单编码，keep-alive（持久化 HTTP 连接）和 HTTP 连接池特性也可以通过 request.sessions.Session 类获得，该类借力于 urllib3（Requests 的一个依赖，无须另外安装）。使用 pip 安装 Requests ：

```
$ pip install requests
```

此外，Requests 文档（*http://docs.python-requests.org/en/latest/index.html*）提供了丰富的信息，推荐一读。

Web API

几乎每个 Web 站点都提供了 API，可以使用 API 来获取站点想要分享的数据，并且某些站点，如 Twitter 和 Facebook，还允许你（或你使用的应用）修改数据。你可能听说过 RESTful API 这个词，其中，REST 是表述性状态转移（REpresentational State Transfer）的缩写，它是一种借助于 HTTP 1.1 协议设计语义的范式，但并非一种标准的协议或规约。不过，多数 Web 服务 API 提供者都遵从 RESTful 的设计原则。下面使用一小段代码举例说明常用术语。

```
import requests

                ❶                          ❷          ❸          ❹
result = requests.get('http://pypi.python.org/pypi/requests/json')
```

❶ get 方法是 HTTP 协议的一部分。在 RESTful API 中，API 的设计者选择服务器会使用哪些动作，并在 API 文档中告诉开发者。虽然 *http://bit.ly/http-method-defs* 上列举了所有方法，但是在 RESTful API 中通常可用的是 GET、POST、PUT 和 DELETE。一般而言，这些 HTTP 谓词做的事情正是它们意指的获取数据、改变数据、删除数据。

❷ URI 的基础部分是 API 的根源。

❸ 客户端会指定一个特定元素来获取相关数据。

❹ 可能还存在一个选项，用来指定不同的媒体类型。

这段代码实际上是对 *http://pypi.python.org/pypi/requests/json*（为 PyPI 提供 JSON 数据的后端服务）执行了一次 HTTP 请求。如果在浏览器中查看这个链接，将看到一个巨长的 JSON 字符串。在 Requests 中，HTTP 请求的返回值是一个 Response 对象。

```
>>> import requests
>>> response = requests.get('http://pypi.python.org/pypi/requests/json')
>>> type(response)
<class 'requests.models.Response'>
>>> response.ok
True
>>> response.text   # 这会给出响应的所有文本内容
>>> response.json() # 这会将文本内容转化成一个字典
```

PyPI 响应的是 JSON 格式的文本，以什么格式发送响应数据没有一个规则，不过许多 API 都使用 JSON 或 XML。

JSON 解析

JavaScript 对象标记（JSON）名副其实，用于定义 JavaScript 对象的表示法。Requests 库在其 Response 对象内构造了一个 JSON 解析器。

json 库可以将 JSON 格式的字符串或文件内容解析为一个 Python 字典（或列表，视情况而定）。它也可以将 Python 字典或列表转化为 JSON 字符串。例如，如下字符串包含 JSON 数据：

```
json_string = '{"first_name": "Guido", "last_name":"van Rossum"}'
```

可以这样对其进行解析：

```
import json
parsed_json = json.loads(json_string)
```

解析完成后它就可以作为一个常规字典来使用：

```
print(parsed_json['first_name'])
"Guido"
```

也可以将如下字典转化为 JSON 字符串：

```
d = {
    'first_name': 'Guido',
```

```
    'last_name': 'van Rossum',
    'titles': ['BDFL', 'Developer'],
}

print(json.dumps(d))
'{"first_name": "Guido", "last_name": "van Rossum", "titles": ["BDFL",
"Developer"]}'
```

更早版本的 Python 可使用 simplejson

json 库是在 Python 2.6 版本时才加入标准库的。如果使用更早版本的 Python，那么可以通过 PyPI 安装使用 simplejson 库。

simplejson 提供的 API，与 Python 标准库中 json 模块的相同，但更新更频繁。引入 simplejson，使用更老版本 Python 的开发者仍然可以使用原本 json 库提供的特性。可以将 simplejson 用作 json 库的一个替代品，如下所示：

```
import simplejson as json
```

在将 simplejson 当作 json 引入之后，之前的示例都能正常工作，好像正在使用的是标准 json 库。

XML 解析

标准库中有一个 XML 解析器（xml.etree.ElementTree 的 parse() 和 fromstring() 方法），不过它的底层使用 Expat 库，并且通过创建一个 ElementTree 对象来保存 XML 结构，这意味着我们必须逐级迭代，进入子元素获取内容。如果只是想获取其中的数据，那么可以尝试 untangle 或 xmltodict。使用 pip 获取这两者：

```
$ pip install untangle
$ pip install xmltodict
```

1. untangle

untangle 读取一个 XML 文档，返回一个 Python 对象，其结构逐一映射 XML 文档的节点和属性。例如，有这样一个 XML 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <child name="child1" />
</root>
```

可以这样加载：

```
import untangle
obj = untangle.parse('path/to/file.xml')
```

获取子元素的名称：

```
obj.root.child['name']  # 值为 'child1'
```

2.xmltodict

xmltodict 将 XML 转化成一个字典。例如，有这样一个 XML 文件：

```
<mydocument has="an attribute">
    <and>
        <many>elements</many>
        <many>more elements</many>
    </and>
    <plus a="complex">
        element as well
    </plus>
</mydocument>
```

它可以被加载到一个 OrderedDict 实例中（类 OrderedDict 存在于 Python 标准库的 collections 模块中），如下所示：

```
import xmltodict

with open('path/to/file.xml') as fd:
    doc = xmltodict.parse(fd.read())
```

然后即可访问元素、属性和值：

```
doc['mydocument']['@has']   # 值为 u'an attribute'
doc['mydocument']['and']['many']    # 值为 [u'elements', u'more elements']
doc['mydocument']['plus']['@a']     # 值为 u'complex'
doc['mydocument']['plus']['#text']  # 值为 u'element as well'
```

借助 xmltodict，还可以使用 unparse() 函数反向将字段转化成 XML。xmltodict 的流式模式适用于处理无法全部读入内存的文件，并且它还支持命名空间。

Web 页面数据抽取

很多网站都以 HTML 格式的 Web 页面提供内容，而不是用 CSV 或 JSON 这类友好的格式提供数据。不过 HTML 也是一种结构化数据格式，可以按规则进行 Web 页面数据抽取。

Web 页面数据抽取是使用计算机程序对 Web 页面内容进行筛选，以最恰当的格式收集需要的数据，同时保留数据的原有结构。

现在，越来越多的网站开始提供 API，以此明确要求不要从页面上抽取数据，因为 API 提供的数据才是站点希望分享的数据，所以在进行页面数据抽取之前，查看目标站点的使用条款，争取做一个网络好公民。

lxml

lxml 使用非常广泛，可以快速解析 XML 和 HTML 文档，甚至能处理解析过程中遇到的某些错误格式的标记。使用 pip 获取 ：

```
$ pip install lxml
```

使用 requests.get 获取目标数据的 Web 页面，使用 lxml 的 html 模块进行解析，并将结果存到树状结构变量 tree 中。

```
from lxml import html
import requests

page = requests.get('http://econpy.pythonanywhere.com/ex/001.html') ❶
tree = html.fromstring(page.content)    ❷
```

❶ 这是一个真实存在的 Web 页面，我们展示的数据也是真实的，可以在浏览器中访问这个页面。

❷ 这里使用 page.content，而不是 page.text，因为 html.fromstring() 隐式要求输入为字节序列（bytes）。

变量 tree 以一个良好的树状结构包含了整个 HTML 文件，可以用 XPath 或 CSSSelect 两种不同的方式对其进行查看。两者都是对一个 HTML 树指定一个查找路径的标准方式，由万维网联盟（W3C）组织定义和维护，在 lxml 中分别实现为一个模块。在当前这个例子中，我们使用 XPath。W3Schools XPath 教程（*http://www.w3schools.com/xsl/xpath_intro.asp*）是一个不错的入门材料。

在 Web 浏览器中获取元素的 XPath，也有多种工具可用，比如 Firefox 的 Firebug 或 Chrome 的检查器。如果你正在使用 Chrome，那么可以右击某个元素，选择“检查元素”按钮，对高亮选中的代码右击一次并选择“复制 XPath”按钮即可。

快速分析之后，我们发现页面中目标数据包含在两个元素中 ：一个是 div 元素，属性

title 为 buyer-name ；另一个是 span 元素，类名为 item-price。

```
<div title="buyer-name">Carson Busses</div>
<span class="item-price">$29.95</span>
```

获知了这一点，就可以构建正确的 XPath 查询，像这样来使用 lxml 的 xpath 函数。

```
# 这行代码会创建一个购买者列表
buyers = tree.xpath('//div[@title="buyer=name"]/text()')
# 这行代码会创建一个价格列表
prices = tree.xpath('//span[@class="item-price"]/text()')
```

来看看得到了什么 ：

```
>>> print('Buyers: ', buyers)
Buyers: ['Carson Busses', 'Earl E. Byrd', 'Patty Cakes', 'Derri Anne
Connecticut', 'Moe Dess', 'Leda Doggslife', 'Dan Druff', 'Al Fresco', 'Ido
Hoe', 'Howie Kisses', 'Len Lease', 'Phil Meup', 'Ira Pent', 'Ben D. Rules',
'Ave Sectomy', 'Gary Shattire', 'Bobbi Soks', 'Sheila Takya', 'Rose Tattoo',
'Moe Tell']
>>>
>>> print('Prices: ', prices)
Prices: ['$29.95', '$8.37', '$15.26', '$19.25', '$19.25', '$13.99', '$31.57',
'$8.49', '$14.47', '$15.86', '$11.11', '$15.98', '$16.27', '$7.50', '$50.85',
'$14.26', '$5.68', '$15.00', '$114.07', '$10.09']
```

数据序列化

数据序列化是指将结构化数据转化成一种能够被共享或存储的格式，保留必要的信息让数据传输的接收端（或在从存储中读取数据时）能够在内存中重建对象。在某些情况下，数据序列化的第二个目的是让被序列化数据最小化，这样就可以最小化磁盘空间需求或带宽需求了。

下面将介绍 Pickle 格式（Python 专有格式）、一些跨语言序列化工具、Python 标准库提供的压缩方案，以及 Python 的缓冲协议（减少传输之前对象被复制的次数）。

Pickle

Python 原生数据序列化模块，名为 Pickle。请看如下示例 ：

```
import pickle

# 一个示例字典
grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }
```

```
# 使用 dumps 方法将对象转化为一个序列化字符串
serial_grades = pickle.dumps( grades )

# 使用 loads 方法反序列化为一个对象
received_grades = pickle.loads( serial_grades )
```

函数、方法、类及像管道这样的临时性东西无法被 Pickle 序列化。

根据 Python 的 Pickle 文档可知，Pickle 模块在遇到错误或恶意结构的数据时是不安全的。对于从不可信任或未验证的数据源获取到的数据，不要使用 Pickle 对其反序列化。

跨语言序列化

如果技术选型支持多语言的序列化模块，那么 Google 的 Protobuf（*https://developers.google.com/protocol-buffers/docs/pythontutorial*）和 Apache 的 Avro（*https://avro.apache.org/docs/1.7.6/gettingstartedpython.html*）是两个常见选择。

另外，Python 标准库包含 xdrlib，它用于打包或解包 XDR 格式（*https://docs.python.org/3/library/xdrlib.html*）的数据，这种格式独立于操作系统和传输协议。相比于前述方法，它的抽象程度更低，仅是将打包过的字节拼接在一起，也因此客户端和服务器端都必须知道打包的类型和顺序。下面这个例子演示了服务器端如何接收 XDR 格式的数据。

```
import socketserver
import xdrlib

class XdrHandler(socketservice.BaseRequestHandler):
    def handle(self):
        data = self.request.recv(4)      ❶
        unpacker = xdrlib.Unpacker(data)
        message_size = self.upacker.unpack_uint()    ❷
        data = self.request.recv(message_size)  ❸
        unpacker.reset(data)     ❹
        print(unpacker.unpack_string())     ❺
        print(unpacker.unpack_float())
        self.request.sendall(b'ok')

server = socketserver.TCPServer(('localhost', 12345), XdrHandler)
server.serve_forever()
```

❶ 数据长度可变，因此先打包添加一个无符号整数（4 个字节）来指明消息大小。

❷ 必须事先知晓接收到的是一个无符号整数。

❸ 在这一行先读取消息剩余部分。

❹ 并在接下来这一行使用新数据重置解包器。

❺ 必须事先知道接收到的是一个字符串，接着一个浮点数。

当然，如果传输的两端实际上都是 Python 程序，那么可以使用 Pickle。但如果服务器端的编程语言完全不同于客户端，那么客户端发送数据的相应代码如下所示：

```
import socket
import xdrlib

p = xdrlib.Packer()
p.pack_string('Thanks for all th fish!')    ❶
p.pack_float(42.00)
xdr_data = p.get_buffer()
message_length = len(xdr_data)

p.reset()   ❷
p.pack_uint(message_length)
len_plus_data = p.get_buffer() + xdr_data   ❸

with socket.socket() as s:
    s.connect(('localhost', 12345))
    s.sendall(len_plus_data)
    if s.recv(1024):
        print('success')
```

❶ 打包待发送数据。

❷ 单独打包消息长度。

❸ 将打包好的消息长度插到整个消息的前面。

压缩

Python 标准库也包含模块，支持使用 zlib、gzip、bzip2 或 lzma 算法压缩 / 解压数据，并支持创建 ZIP 格式或 tar 格式的归档文件。将 zip 压缩应用于 Pickle 数据处理：

```
import pickle
import gzip

data = "my very big object"
```

```
# zip 压缩及 pickle 序列化处理
with gzip.open('spam.zip', 'wb') as my_zip:
    pickle.dump(data, my_zip)

# zip 解压及 pickle 反序列化处理
with gzip.open('spam.zip', 'rb') as my_zip:
    unpickled_data = pickle.load(my_zip)
```

缓冲协议

Python 核心开发者之一 Eli Bendersky 写过一篇博文，关于如何使用内存缓冲减少 Python 对相同数据进行内存内复制的次数（*http://tinyurl.com/bendersky-buffer-protocol*）。使用这个技术，甚至可以将数据从文件或套接字读入一个已有的内存缓冲区中。详细信息请阅读 Python 的缓冲协议文档（*https://docs.python.org/3/c-api/buffer.html*），以及 PEP 3118（*http://legacy.python.org/dev/peps/pep-3118/*），该提案建议的改进方案 Python 3 已实现，并向后移植到 Python 2.6 及以上版本。

分布式系统

分布式计算机系统通过相互传递消息共同完成一个任务（如玩游戏、一个互联网聊天室或一次 Hadoop 计算）。本节先针对常见网络任务列举最流行的一些网络编程库，之后讨论网络通信必备的密码技术。

网络编程

在 Python 中，互连网络的通信通常使用异步工具或多线程来处理，以此规避全局解释器锁的单线程限制。表 9-1 中所有的库都解决同一个问题——规避 GIL，并提供数量不一的附加特性。

表9-1　网络编程库

库名称	许可证	使用理由
asyncio	Python 软件基金会许可证	• 提供一个异步事件循环来管理与非阻塞套接字或队列的通信，以及任意用户定义的协程 • 包含异步套接字和队列
gevent	MIT 许可证	• 与 libev、异步 I/O 的 C 库紧耦合 • 提供一个快速的 WSGI 服务器，构建于 libev 的 HTTP 服务器之上

续表

库名称	许可证	使用理由
		• 包含 gevent.monkey 模块，它为标准库提供一些补丁函数，这样使用阻塞套接字编写的第三方模块也可以配合 gevent 使用
Twisted	MIT 许可证	• 为新出的协议（例如 GPS、产品互联网（IoCP）（译注：相当于“物联网”的概念））提供异步实现，还提供一个 Memcached（*https://memcached.org/*）协议的异步实现 • 将自己的事件循环与许多其他事件驱动框架集成，比如 wxPython、GTK • 内建一个 SSH 服务器和一些客户端工具
PyZMQ	LGPL（ZMQ 使用）和 BSD 许可证（Python 部分）	• 允许使用一个套接字风格的 API 设置非阻塞消息队列，并与其交互 • 提供支持分布式计算的套接字行为（请求 / 响应、发布 / 订阅、推送 / 拉取） • 如果希望构建自己的通信基础设施，则使用这个库；虽然名字中有个 Q，但它与 RabbitMQ 不同，可以使用它来构建类似 RabbitMQ 的东西，或者一个与 RabbitMQ 行为完全不同的东西（依赖于套接字模式的选择）
pika	BSD 许可证	• 提供一个轻量的 AMQP 客户端，用来连接 RabbitMQ 或其他消息代理 • 也包含适配器来实现在 Torndao 或 Twisted 事件循环中使用 pika • 使用 RabbitMQ 这类消息代理时，如果想要一个更轻量的客户端（无须 Web 仪表板及其他附属功能），那么可以选择 pika 来将内容推送到 RabbitMQ 这类外部消息代理

Python 标准库中的高性能网络编程工具

Python 3.4 引入 asyncio，包含一些从开发者社区（例如，维护 Twisted 和 gevent 的社区）借鉴来的思想。asyncio 是一个并发工具，并发的一个常见应用是网络服务器。Python 官方的 asyncore（asyncio 的前身）文档叙述如下。

> 在单核处理器上，一个程序要实现“一次不只做一件事”只有两种方式。其中多线程编程是最简单也是最常用的方式，但还有另一种完全不同的技术具备多线程编程几乎全部的优点，又不是真的使用多个线程。不过只有在程序主要受 I/O 限制时，这种方式才适用。如果程序受处理器计算能力限制，那么真正需要的应该是抢占式调度的多线程。然而，网络服务器程序很少受限于处理器计算能力。

asyncio 仍然只是临时性地存在于 Python 标准库中，其 API 可能会进行向后不兼容的变更，因此不要过于依赖它。

这个模块并非全新的事物。ayncore（Python 3.4 废弃）也有事件循环、异步套接字[2]和异步文件 I/O，另外还有 asynchat（Python 3.4 废弃）提供异步队列[3]。asyncio 新增的重要东西是一个协程的官方正式实现。在 Python 中，协程概念由两者定义：协程函数，函数定义以 async def 开始，而不只是 def（或使用更老的语法，装饰上 @asyncio.coroutine），以及调用协程函数（通常是某种计算或 I/O 操作）获取到的对象。协程可以让出处理器，因此能够参与到一个异步事件循环中，与其他协程轮流运行。

因为对于这门语言来说这是一个新概念，所以官方文档有大量篇幅的详细示例来帮助理解。这个文档非常清晰全面，值得仔细阅读学习。我们只会展示一下事件循环相关的函数和一些可用的类，如下所示。

```
>>> import asyncio
>>>
>>> [l for l in asyncio.__all__ if 'loop' in l]
['get_event_loop_policy', 'set_event_loop_policy', 'get_event_loop', 'set_
event_loop', 'new_event_loop']
>>>
>>> [t for t in asyncio.__all__ if t.endswith('Transport')]
['BaseTransport', 'ReadTransport', 'WriteTransport', 'Transport',
'DatagramTransport', 'SubprocessTransport']
>>>
>>> [p for p in asyncio.__all__ if p.endswith('Protocol')]
['BaseProtocol', 'Protocol', 'DatagramProtocol', 'SubprocessProtocol',
'StreamReaderProtocol']
>>>
>>> [q for q in asyncio.__all__ if 'Queue' in q]
['Queue', 'PriorityQueue', 'LifoQueue', 'JoinableQueue', 'QueueFull',
'QueueEmpty']
```

gevent

gevent 是一个基于协程的 Python 网络开发库，在 C 库 libev 事件循环上借助 greenlet 库提供抽象层次更高的同步 API。greenlet 是一种微型绿色线程（或者说是用户级线程，相对于内核控制的线程而言），开发者可以显式地挂起 greenlet，或者在几个 greenlet 之间

2 一个套接字有三个组成部分：一个 IP 地址（包括端口）、一种传输协议（TCP/UDP 这种），以及一个 I/O 通道（某种类文件的对象）。Python 官方文档包含一篇不错的套接字编程入门教程（*https://docs.python.org/3/howto/sockets.html*）。

3 这个队列不要求 IP 地址或协议，因为它是在同一个计算机上做数据传输，一个进程向它写入数据，另一个进程就可以读取到。类似于 multiprocessing.Queue，不过其 I/O 操作是异步完成的。

跳转。深入学习 gevent，推荐阅读 Kavya Joshi 的演讲报告《Python 创造性实践中灵活运用并发的故事》（*http://bit.ly/kavya-joshi-seminar*）。

gevent 因轻量，与底层 C 库 libev 紧耦合，性能很高，而被广泛使用。如果喜欢异步 I/O 与 greenlet 集成的思路，那么 gevent 就是你期望的库。使用 pip 获取它：

```
$ pip install gevent
```

下面的示例来自 greenlet 文档。

```
>>> import gevent
>>>
>>> from gevent import socket
>>> urls = ['www.google.com', 'www.example.com', 'www.python.org']
>>> jobs = [gevent.spawn(socket.gethostbyname, url) for url in urls]
>>> gevent.joinall(jobs, timeout=2)
>>> [job.value for job in jobs]
['74.125.79.106', '208.77.188.166', '82.94.164.162']
```

更多的示例可以从 *https://github.com/gevent/gevent/tree/master/examples* 上获取。

Twisted

Twisted 是一个事件驱动的网络引擎，可用于构建各种网络协议的应用，包括 HTTP 服务器和客户端，使用 SMTP、POP3、IMAP 或 SSH 协议的应用，即时通信及许多其他应用（*http://twistedmatrix.com/trac/wiki/Documentation*）。使用 pip 获取：

```
$ pip install twisted
```

Twisted 始于 2002 年，社区忠诚度高。与协程库中的 Emacs 类似，它内建了一切需要的网络模块。所有这些模块都必须是异步的，才能一起高效地工作。其中最有用的工具可能是一个数据连接的异步包装模块（在 twisted.enterprise.adbapi 中）、一个 DNS 服务器模块（在 twisted.names 中）、直接访问数据包的模块（在 twisted.pair 中）及附加的协议实现，如 AMP、GPS 及 SOCKSv4（在 twisted.protocols 中）。目前 Twisted 的大部分模块都能运行在 Python 3 环境中，在 Python 3 环境中执行 pip install 安装 Twisted，获取到的是目前已移植到 Python 3 的所有模块。如果发现想用的某样东西在 API 文档（*http://twistedmatrix.com/documents/current/api/moduleIndex.html*）中，但在已经安装的 Twisted 中找不到，那么应该使用 Python 2.7。

进一步学习，推荐阅读 Jessica McKallar 和 Abe Fettig 编写的 *Twisted* 一书。此外，网页 *http://twistedmatrix.com/documents/current/core/examples/* 上提供了 40 多个 Twisted 的使用示例，网页 *http://speed.twistedmatrix.com/* 上展示了最新的速度性能数据。

PyZMQ

PyZMQ 是 ZeroMQ 的 Python 绑定，可使用 pip 获取。

```
$ pip install pyzmq
```

ØMQ（也可拼写为 ZeroMQ、0MQ 或 ZMQ）将自己描述为一个消息传递开发库，其 API 特意设计成大家熟悉的套接字风格，目的是用于可伸缩的分布式或高并发应用。大致来说，其实现了附带队列的异步套接字，提供一些自定义的套接字“类型”，套接字“类型”决定了每个套接字上的 I/O 操作行为。下面是一个示例。

```
import zmq
context = zmq.Context()
server = context.socket(zmq.REP)                ❶
server.bind('tcp://127.0.0.1:5000')             ❷

while True:
    message = server.recv().decode('utf-8')
    print('Client said: {}'.format(message))
    server.send(bytes("I don't know.", 'utf-8'))

# ~~~~~ 在另一个文件中 ~~~~~

import zmq
context = zmq.Context()
client = context.socket(zmq.REQ)                ❸
client.connect('tcp://127.0.0.1:5000')              ❹

client.send(bytes("What's for lunch?", 'utf-8'))
response = client.recv().decode('utf-8')
print('Server replied: {}'.format(response))
```

❶ 套接字类型为 zmq.REP，对应于“请求 – 回复”范式。

❷ 类似于普通套接字，将服务器端绑定到 IP 和端口上。

❸ 客户端套接字类型为 zmq.REQ，ZMQ 把套接字类型定义成一些常量：zmq.REQ、zmq.REP、zmq.PUB、zmq.SUB、zmq.PUSH、zmq.PULL、zmq.PAIR。套接字类型决定了套接字如何发送和接收数据。

❹ 客户端连接到服务器端绑定的 IP 和端口。

因此，这看起来、用起来都类似套接字，并以队列和不同的 I/O 模式进行特性增强。这些模式的目的在于为分布式网络提供构件块。套接字类型的基本模式如下。

请求 – 回复

zmq.REQ 和 zmq.REP 将一组客户端连接到一组服务，可用于实现一个远程过程调用模式或一个任务分发模式。

发布 – 订阅

zmq.PUB 和 zmq.SUB 将一组发布者连接到一组订阅者，这是一种数据分发模式。一个节点分发数据给其他节点，或者多级串接扇形展开成一个分发树。

推送 – 拉取（或流水线）

zmq.PUSH 和 zmq.PULL 以扇入 / 扇出模式连接节点，可以有多级或成环。这是一种并行任务分发和收集模式。

相比面向消息的中间件，ZeroMQ 的重大优势是可用于消息排队，而不需要一个独立的消息代理。PyZMQ 的文档提及 PyZMQ 引入的一些增强特性，比如基于 SSH 的隧道技术。若想读读 ZeroMQ API 的其他文档，推荐 ZeroMQ 指南。

RabbitMQ

RabbitMQ 是一个开源的消息代理软件，实现了高级消息队列协议（AMQP）。消息代理是一个居间程序，根据协议从发送端接收消息后发送给接收端，使得 AMQP 的任意客户端都可以与 RabbitMQ 通信。从 RabbitMQ 的下载页 *https://www.rabbitmq.com/download.html* 获取 RabbitMQ，并遵照对应操作系统的步骤进行安装。

所有主流编程语言都有一些与消息代理交互的客户端库。于 Python 而言，常用的 pika 和 Celery 都可使用 pip 进行安装：

```
$ pip install pika
$ pip insyall celery
```

1. pika

pika 是一个轻量的纯 Python 实现的 AMQP 0-9-1 协议客户端库，它是 RabbitMQ 的首选。RabbitMQ 的 Python 版入门教程（*https://www.rabbitmq.com/getstarted.html*）使用了 pika。如果搭建了 RabbitMQ，那么推荐先使用 pika 进行体验，无论最终选择哪个客户端库，因为它简单易懂，没有额外的特性，使用之后，相关概念会更具体明晰。

2. Celery

Celery 是一个特性更齐全的 AMQP 客户端库，可配合 RabbitMQ 或 Redis（分布式内存数据存储服务）使用，可以追踪任务和结果（并且可以选择将结果存放在用户指定的后端存储服务中），具有一个 Web 后台管理工具 / 任务监控器 Flower（*https://pypi.python.org/pypi/flower*）。在 Web 开发社区中，Celery 非常流行，Django、Pyramid、Pylons、web2py 及 Tornado 都有对应的集成包（Flask 不需要专门的集成包），可以从 Celery 教程（*http://tinyurl.com/celery-first-steps*）开始学习。

密码技术

2013 年，Python 密码学权威组（PyCA）正式成立。该群组的开发者成员对于为 Python 社区提供高质量密码技术开发库都非常感兴趣[4]。这个开发组提供工具根据恰当的密钥加密 / 解密消息，提供密码学哈希函数不可逆但可重复地混淆密码或其他私密数据。

表 9-2 中除 pyCrypto 之外的所有程序库都由 PyCA 维护。几乎所有这些库都是基于 C 库 OpenSSL 的，例外情况会有备注。

表9-2　密码技术工具库可选方案

可选方案	许可证	使用理由
ssl 和 hashlib（以及 Python 3.6 中的 secrets）	Python 软件基金会许可证	• hashlib 提供一个相当好的密码哈希算法，随 Python 版本发布而更新，ssl 提供一个 SSL/TLS 客户端（以及一个服务器端，但更新不一定及时） • secrets 是一个适用于密码学场景的随机数生成器
pyOpenSSL	Apache 2.0 许可证	• 使用最新版本的 OpenSSL，提供标准库 ssl 模块未暴露的一些 OpenSSL 功能函数
PyNaCl	Apache 2.0 许可证	• 包含 libsodium 库的 Python 绑定[a]
libnacl	Apache 许可证	• 为使用 Salt Stack（*http://saltstack.com/*）的开发者提供 libsodium 库的 Python 接口
cryptography	Apache 2.0 许可证或 BSD 许可证	• 允许直接访问构建在 OpenSSL 之上的密码学基元。大多数开发者会使用抽象层次更高的 pyOpenSSL
pyCrypto	公有领域	• 这个库相对比较老，构建在自己的 C 库之上，以前它是 Python 中最流行的密码学开发库

4　Jake Edge 在博文《Python 中密码技术的现状》（*http://bit.ly/raim-kehrer-talk*）中介绍了 Cryptography 库的问世，以及这次新尝试的背后驱动力。文中介绍的 Cryptography 库是一个底层的库，大多数人使用的是抽象层次更高的库（如 pyOpenSSL），可以封装导入 Cryptography。Jake Edge 引用了 Jarret Raim 和 Paul Kehrer 做的题为“Python 中密码技术的现状”（*https://www.youtube.com/watch?v=r_Pj__qjBvA*）的演讲，声称其测试集包含的测试用例超过 6.6 万个，每次构建运行 77 轮测试。

续表

可选方案	许可证	使用理由
bcrypt	Apache 2.0 许可证	• 提供 bcrypt 哈希函数 b，对于想使用该哈希方式或之前用过 py-bcrypt 的开发者来说，非常适用

a libsodium 是网络开发和密码技术库（NaCl，发音为“salt”）的一个分支。其哲学是去芜存菁选取性能好且易用的特定算法。

b 这个库实际上包含了 C 源码，在安装时使用 C 语言快速函数接口（CFFI）进行构建。bcrypt 基于 Blowfish 加密算法。

ssl、hashlib 及 secrets

Python 标准库的 ssl 模块提供了一个套接字 API（ssl.socket），行为类似标准套接字，但是经过 SSL 协议包装，另外还提供了 ssl.SSLContext，它包含 SSL 连接配置。http 模块（或 Python 2 中的 httplib 模块）使用 ssl 来支持 HTTPS。如果使用 Python 3.5，那么 ssl 模块还支持内存 BIO，这样套接字会把输入输出数据写到一个缓冲区，而不是其目的端，从而也可以添加钩子，比如在读写之前进行十六进制编码 / 解码。

Python 3.4 引入的安全改进主要是支持更新的传输协议和哈希算法。这些议题很重要，所以也向后移植到了 Python 2.7 中，PEP 466（*https://www.python.org/dev/peps/pep-0466/*）和 PEP 476（*https://www.python.org/dev/peps/pep-0476/*）中有论述。Benjamin Peterson 做过一个关于 Python ssl 模块进展的演讲（*http://bit.ly/peterson-talk*）可以参考。

如果使用 Python 2.7，那么请确保至少升级到版本 2.7.9，或者至少是包含了 PEP 476 的版本，这样，在使用 https 协议连接时，HTTP 客户端默认会进行证书校验。或者，一直使用 Requests 就行了，因为其始终默认会如此操作。

Python 团队推荐使用 SSL 默认设置，除非对于客户端使用的安全策略有特殊要求。下面这个例子，取自 ssl 库文档的“安全考虑”（*http://bit.ly/ssl-security-consider*）一节，演示了一个安全的邮件客户端，若使用这个库，应该读一读。

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

若要确认一个消息在传输过程中是否遭到破坏，可以使用 hmac 模块。它实现了 RFC 2104（*https://tools.ietf.org/html/rfc2104.html*）中描述的信息身份验证密钥——散列算法

（HMAC）。使用 hashlib.algorithms_available 中任一算法计算哈希值的消息，都可以使用 hmac 来校验。进一步学习，可参考每一周一 Python 系列课程中的 hmac 示例（*https://pymotw.com/2/hmac/*）。hmac.compare_digest() 对比消息摘要的时间是常量的，从而防止遭受时序攻击，攻击者尝试通过消息摘要对比耗用的时长来推测算法。

Python 的 hashlib 模块可以用来为安全存储生成哈希过的密码，或者为确认传输之间数据完整性生成校验和。目前，NIST 特别出版物 800-132（*http://bit.ly/nist-recommendation*）推荐的基于口令的密钥派生函数 2（PBKDF2），被认为是密码哈希的最佳方案之一。如下是该函数的一个应用示例，使用一个盐值[5] 和安全哈希算法 256 位哈希（SHA-256）1 万次迭代来生成一个经过哈希的密码（程序员可以选择不同哈希算法和迭代次数，在可靠性和期望响应速度之间做平衡）。

```
import os
import hashlib

def hash_password(password, salt_len=16, iterations=10000, encoding='utf-8'):
    salt = os.urandom(salt_len)
    hashed_password = hashlib.pbkdf2_hmac(
        hash_name='sha256',
        password=bytes(password, encoding),
        salt=salt,
        iterations=iterations
    )
    return salt, iterations, hashed_password
```

从 Python 3.6 开始可用 PEP 506（*https://www.python.org/dev/peps/pep-0506/*）中提议的 secrets 库。它提供安全令牌生成函数，适用于密码重置和生成难以猜测的 URL 这类应用，其文档包含一些示例，推荐了一些基础层级安全管理的最佳实践。

pyOpenSSL

Cryptography 库出现后，pyOpenSSL 库更新了其绑定实现，使用 Cryptography 库基于 CFFI 的 OpenSSL 库绑定，并加入了 PyCA 组织。pyOpenSSL 故意从 Python 标准库中分离出来，以便可以按照安全社区的速度发布更新[6]，构建于最新版本的 OpenSSL 之上，而不是像 Python 那样构建在操作系统内置的 OpenSSL 之上（除非你自己使用一个更新的版本来构建）。构建服务器程序通常会用 pyOpenSSL，阅读 Twisted 的 SSL 文档

5 盐值是进一步混淆哈希的一个随机字符串。如果大家都使用同一个算法，那么坏人就可以针对常用密码及其对应的哈希值生成一个查找表，使用它来“解码”偷来的密码文件。因此，为了防止这种事情发生，开发者通常会为密码附加一个随机字符串（一个盐值），不过必须存放好这个随机字符串以备将来使用。

6 任何人都可以加入 PyCA 的 cryptography-dev 邮件组以便及时获知开发和其他方面的新闻，也可以加入 OpenSSL 邮件组以获知 OpenSSL 的新闻。

（*http://twistedmatrix.com/documents/12.0.0/core/howto/ssl.html*）通过示例学习如何使用pyOpenSSL。

使用 pip 进行安装：

```
$ pip install pyOpenSSL
```

并以 OpenSSL 导入。如下例子演示了一组可用的函数。

```
>>> import OpenSSL
>>>
>>> OpenSSL.crypto.get_elliptic_curve('Oakley-EC2N-3')
<Curve 'Oakley-EC2N-3'>
>>>
>>> OpenSSL.SSL.Context(OpenSSL.SSL.TLSv1_2_METHOD)
<OpenSSL.SSL.Context object at 0x10d778ef0>
```

pyOpenSSL 团队维护了一些示例代码（*https://github.com/pyca/pyopenssl/tree/master/examples*），包括如何生成证书、如何在一个已经建立连接的套接字之上开始使用 SSL，以及如何实现一个安全的 XMLRPC 服务器程序。

PyNaCl 和 libnacl

C 库 libsodium 是 PyNaCl 和 libnacl 的底层支撑，其理念是有意不为用户提供多种选择，为各种应用场景提供最好的方案即可。它仅支持部分 TLS 协议，如果要支持所有 TLS 协议，那么请使用 pyOpenSSL。如果只是想在可控的计算机之间建立加密连接，使用自己选择的协议，也不想与 OpenSSL 打交道，那就用这个库[7]。

PyNaCl 发音为 py-salt，libnacl 发音为 lib-salt，两者都源自 NaCl（发音为 salt）库。

相比 libnacl，我们更推荐 PyNaCl，因为它由 PyCA 组织开发维护，并且不需要另外单独安装 libsodium。这两个库本质上是一样的。PyNaCl 使用 CFFI 绑定 C 库，libnacl 则使用 ctypes，因此实际上区别不大。使用 pip 安装 PyNaCl：

7 如果你有点偏执，希望能够完全审计你的加密 / 解密代码，不在乎代码运行的速度，对当前的多数算法和默认方案也不感兴趣，那么可以尝试 TweetMaCl。它是一个密码技术库，只包含一个代码文件，刚好 100 条推文的长度。因为 PyNaCl 在其发布版本中捆绑了 libsodium，所以引入 TweetNaCl，可能大多数情况下一切仍然可以运行（不过，笔者没试过这种方式）。

```
$ pip install PyNaCl
```

然后依照其文档中的 PyNaCl 示例学习使用。

Cryptography

Cryptography 提供密码技术套路方法和基元方法，支持 Python 2.6、Python 2.7、Python 3.3 及以上、PyPy。不过对于多数应用场景，PyCA 推荐使用 pyOpenSSL 中抽象层次更高的接口。

Cryptography 分为两层：套路层（recipes）和充满危险的物料层（hazmat）。套路层为常规的对称加密提供一套简单的 API，hazmat 层提供抽象层次较低的密码技术基元。使用 pip 进行安装：

```
$ pip install cryptograhy
```

如下例子使用了一个高抽象层次的对称加密套路方法，这个库中仅有的一个高抽象层次功能：

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"A really secret message.")
plain_text = cipher_suite.decrypt(cipher_text)
```

PyCrypto

PyCrypto 提供一些安全哈希函数和多种加密算法，支持 Python 2.1 及以上版本和 Python 3 及以上版本。它依赖的 C 代码是为项目量身定制的，对于是否采用它，PyCA 保持谨慎的态度，但多年来其仍然是 Python 界事实上的密码技术标准库，因此一些老代码使用了它。使用 pip 进行安装：

```
$ pip install pycrypto
```

然后使用它：

```
from Crypto.Cipher import AES
# 加密
encryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an
IV456')
cipher_text = encryption_suite.encrypt("A really secret message.")
# 解密
decryption_suite = AES.new('This is a key123', AES.MODE_CBC, 'This is an
IV456')
plain_text = decryption_suite.decrypt(cipher_text)
```

bcrypt

如果想对口令密码应用 bcrypt 算法，那就使用这个库。因为接口兼容，py-bcrypt 的用户会发现切换到这个库很容易。使用 pip 进行安装：

```
$ pip install bcrypt
```

它只有 bcrypt.hashpw() 和 bcrypt.gensalt() 两个函数。后者让用户可以选择迭代次数，迭代次数越多，算法越慢（默认值是一个合理的数字），示例如下：

```
>>> import bcrypt
>>>
>>> password = bytes('password', 'utf-8')
>>> hashed_pw = bcrypt.hashpw(password, bcrypt.gensalt(14))
>>> hashed_pw
b'$2b$14$qAmVOCfEmHeC8Wd5BoF1W.7ny9M7CSZpOR5WPvdKFXDbkkX8rGJ.e'
```

将经过哈希的密码存储在某处：

```
>>> import binascii
>>> hexed_hashed_pw = binascii.hexlify(hashed_pw)
>>> store_password(user_id=42, password=hexed_hashed_pw)
```

在做密码对比时，把经过哈希的密码作为 bcrypt.hashpw() 的第二个参数。

```
>>> hexed_hashed_pw = retrieve_password(user_id=42)
>>> hashed_pw = binascii.unhexlify(hexed_hashed_pw)
>>>
>>> bcrypt.hashpw(password, hashed_pw)
b'$2b$14$qAmVOCfEmHeC8Wd5BoF1W.7ny9M7CSZpOR5WPvdKFXDbkkX8rGJ.e'
>>>
>>> bcrypt.hashpw(password, hashed_pw) == hashed_pw
True
```

第 10 章

数据操作

本章概述一些数据操作相关的 Python 流行库，这里的“数据”包括：数值、文本、图像及音频。本章描述的大部分程序库分别服务于不同且唯一的目的，因此本章的目标是描述这些库，而不是对比它们。除非特意备注，所有库都可以使用 pip 直接从 PyPI 上下载安装。

```
$ pip install library
```

表 10-1 简要描述了这些 Python 流行库。

表10-1　Python流行库

可选方案	许可证	使用理由
IPython	Apache 2.0 许可证	• 提供增强型 Python 解释器环境，包含特性：输入历史、集成调试器及终端内图像生成和绘图（基于 Qt 的版本支持该特性）
Numpy	BSD 3 条款许可证	• 提供多维数组和线性代数工具，专为速度优化
SciPy	BSD 许可证	• 提供工程学和科学相关的功能和工具，覆盖从线性代数到信号处理、积分、求根、统计分布等主题方向
Matplotlib	BSD 许可证	• 提供科学绘图功能
Pandas	BSD 许可证	• 提供 series 和 DataFrame 对象，这两种对象支持排序、归并、分组、聚合、索引、时间窗口及求子集 • 非常类似 R 语言的数据框或一次 SQL 查询的内容
Scikit-Learn	BSD 3 条款许可证	• 提供机器学习算法，包括降维分类、回归、聚类、模型选择、缺失数据填充和数据预处理
Rpy2	GPLv2 许可证	• 提供一个 R 语言的 Python 接口，支持在 Python 内执行 R 函数，以及在两个环境之间传递数据

可选方案	许可证	使用理由
SymPy	BSD 许可证	• 提供符号数学工具，包括级数展开、极限求值及微积分，目标是构建一个完整的计算机代数系统
nltk	Apache 许可证	• 提供全面的自然语言处理工具集，支持多语言模型和训练数据
pillow/PIL	标准 PIL 许可证（类似 MIT 许可证）	• 支持大量的文件格式，附加一些简单的图像滤波及其他处理功能
cv2	Apache 2.0 许可证	• 提供适用于视频实时分析的计算机视觉处理方法，包括经过训练的人脸和人物侦测算法
Scikit-Image	BSD 许可证	• 提供各种图像处理方法 • 过滤、校正、分色、边缘检测、斑点分析，以及角点检测、分割等

表 10-1 中描述的库及本章剩余部分将详细介绍的所有的库都依赖于 C 库，特别是依赖于 SciPy，或者依赖于 SciPy 项目的 NumPy。这意味着如果要在 Windows 系统上安装这些库可能会有点麻烦。如果你主要使用 Python 来分析科学数据，并且也不熟悉如何在 Windows 上编译 C 和 FORTRAN 代码，那么建议使用 Anaconda 或“商业化 Python 二次发行版”一节中介绍的其他方案。如果不采纳这个建议，则应该始终先尝试 pip install，如果失败，那么请阅读 ScipPy 安装指南（*https://www.scipy.org/install.html*）。

科学应用

Python 常用于高性能科学应用。因为容易编写，表现良好，在学术界和科研项目中应用非常广泛。

科学计算必然要求高性能，所以在 Python 中经常借助一些外部库，这些库通常以执行速度更快的语言（例如，用 C 或 FORTRAN 语言来实现矩阵操作）编写。SciPy 技术栈的各个部分都是常用的 Python 科学计算库：NumPy、SciPy、SymPy、Pandas、Matplotlib 及 IPython，深入介绍这些库超出了本书的内容范围。不过，《Python 科学计算讲义》（*http://scipy-lectures.github.com/*）一文全面介绍了 Python 科学计算生态系统，值得一读。

IPython

IPython 是 Python 解释器环境的一个增强版本，提供彩色界面、更详细的错误信息，并提供一个内嵌模式支持终端内图像展示和绘图（基于 Qt 的版本）。IPython 是 Jupyter notebooks 的默认内核，也是 Spyder IDE 的默认解释器环境，它会随 Anaconda 一起安装。

NumPy

NumPy 虽然是 SciPy 项目的一部分，但是作为一个独立的库进行发布，这样只需要这个基本依赖的人不用安装 SciPy 的其余部分就能使用 NumPy。NumPy 借助多维数组和数组操作函数解决了 Python 上算法执行慢的问题。其底层支撑是自动调谐线性代数软件（ATLAS）库[1]，以及其他 C 和 FORTRAN 语言编写的底层库。NumPy 兼容 Python 2.6 以上和 Python 3.2 以上版本。

一个矩阵乘法的示例如下，它使用了 array.dot() 及“传播”，即逐个元素的乘法（行或列会把值“传播”到缺失的维度）：

```
>>> import numpy as np
>>>
>>> x = np.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>>
>>> x.dot([2,2,1])
array([9, 24])
>>>
>>> x * [[1],[0]]
array([[1, 2, 3],
       [0, 0, 0]])
```

SciPy

SciPy 使用 NumPy 来实现更多的数学函数。SciPy 以 NumPy 数组为基础数据结构，为科学计算编程中的各类常用任务提供功能模块，包括线性代数、微积分、特殊函数常量，以及信号处理。

如下示例演示了 SciPy 的物理常量集。

```
>>> import scipy.constants
>>> fahrenheit = 212
>>> scipy.constants.F2C(fahrenheit)
100.0
>>> scipy.constants.physical_constants['electron mass']
(9.10938356e-31, 'kg', 1.1e-38)
```

1 ATLAS 是一个处于持续开发中的软件项目，提供经过充分测试的高性能线性代数库。该项目对知名的基础线性代数子集（BLAS）和线性代数程序包（LAPACK，Linear Algebra PACKage）进行封装，提供 C 和 FORTRAN 77 编程接口。

Matplotlib

Matplotlib 是一个灵活的绘图库，可创建交互式 2D 和 3D 图形，也可将图形保存为高质量的原图。其 API 非常接近 MATLAB，因此 MATLAB 用户切换到 Python 比较容易。Matplotlib 图库不仅包含了很多绘图示例，还一并提供源码。

从事统计工作的人也可以看看 Seaborn，它是一个专为统计可视化设计的新图形库，流行度与日俱增。《数据科学入门》（*http://bit.ly/data-science-python-guide*）一文介绍了它。

至于 Web 端绘图，可以尝试 Bokeh，它使用自己的可视化库，或者尝试一下 Plotly，它基于 JavaScript 库 D3.js，不过 Plotly 的免费版本可能会要求将绘图结果存储在 Plotly 供应商的服务器上。

Pandas

Pandas（名称源自 Panel Data 一词）是一个基于 NumPy 的数据操作库，提供许多好用的函数，可以轻松地访问、索引、合并及分组数据。其核心数据结构（DataFrame）接近于 R 统计软件环境中对应的概念。这是一种异构数据表格，可以在某些列存放字符串，另一些列中存放数字，可以进行名称索引、时间序列操作及自动对齐数据。也可以像操作 SQL 数据表或 Excel Pivot 表一样，使用 groupby() 这类方法或 pandas.rolling_mean() 这类函数。

Scikit-Learn

Scikit-Learn 是一个机器学习库，具有降维、缺失数据填充、回归和分类模型、树模型、聚类、自动模型参数调优、绘图（借助 matplotlib）等能力。其文档齐备，自带大量示例。Scikit-Learn 不仅操作 Numpy 数组，而且也可以与 Pandas 的数据框配合使用。

Rpy2

Rpy2 是 R 语言统计包的一个 Python 绑定，允许从 Python 中执行 R 函数，并在两个环境中来回传递数据。Rpy2 是 Rpy 绑定的一个面向对象实现。

小数、精度和 numbers 库

Python 定义了一个抽象基类框架来开发数值类型，包含 Number 类、所有数值类型的根类、Integral 类、Rational 类、Real 类及 Complex 类。开发者可以根据 numbers 库[2] 中的说明从这些抽象类继承子类来开发其他数值类型。Python 标准库中还有一个 decimal.Decimal

2 流行工具 SageMath 借助了 Python 库 numbers，这个工具大而全，定义各种类来表示域（field）、环（ring）、代数（algebra）和定义域（domain），并且基于 SymPy 提供符号工具，基于 NumPy、SciPy 和大量其他 Python/ 非 Python 库提供数值工具。

类可以用来处理数值精度，帮助完成计费和精度至关重要的其他工作。这个类型层次结构的工作方式如下所示。

```
>>> import decimal
>>> import fractions
>>> from numbers import Complex, Real, Rational, Integral
>>>
>>> d = decimal.Decimal(1.11, decimal.Context(prec=5))  # 精度
>>>
>>> for x in (3, fractions.Fraction(2,3), 2.7, complex(1,2), d):
...     print('{:>10}'.format(str(x)[:8]),
...           [isinstance(x, y) for y in (Complex, Real, Rational,
Integral)])
...
         3 [True, True, True, True]
       2/3 [True, True, True, False]
       2.7 [True, True, False, False]
     (1+2j) [True, False, False, False]
  1.110000 [False, False, False, False]
```

Python 标准库中，math 库提供指数、三角函数及其他常用函数，cmath 库则为复数提供对应的数学函数。random 库使用梅森旋转算法（*https://en.wikipedia.org/wiki/Mersenne_Twister*）作为核心生成器来提供伪随机数。从 Python 3.4 开始，标准库加入了 statistics 模块来提供均值和中位数函数，以及样本、总体标准差和方差函数。

SymPy

Python 中首选 SymPy 库来处理符号数学，它完全使用 Python 语言编写，提供可选扩展来实现加速、绘图和交互式会话功能。

SymPy 的符号函数可以操作符号、函数及表达式等 SymPy 对象来生成其他符号表达式。

```
>>> import sympy as sym
>>>
>>> x = sym.Symbol('x')
>>> f = sym.exp(-x**2/2) / sym.sqrt(2 * sym.pi)
>>> f
sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))
```

可以对符号或数值进行积分。

```
>>> sym.integrate(f, x)
erf(sqrt(2)*x/2)/2
>>>
>>> sym.N(sym.integrate(f, (x, -1, 1)))
```

```
0.682689492137086
```

这个库还可以求微分，表达式展开成级数，限制符号为实数、满足交换律或其他诸多范畴，给定精度求浮点数最近的有理数等。

文本操作和文本挖掘

很多人因为 Python 的字符串操作工具而选择这门语言开始学习编程。下面首先快速学习一下 Python 标准库中字符串操作相关的部分，然后学习自然语言处理工具集（nltk），社区中几乎人人都会使用这个库进行文本挖掘。

Python 标准库中的字符串工具

在某些自然语言中，有些小写字母比较特殊，str.casefold() 可以帮助其转变为小写字母。

```
>>> 'Grünwalder Straße›.upper()
'GRUNWALDER STRASSE'
>>> 'Grünwalder Straße›.lower()
'grünwalder straße›'
>>> 'Grünwalder Straße›.casefold()
'grünwalder strasse'
```

Python 的正则表达式库 re 功能全面而强大，前面已经介绍，这里不再赘述，不过另外提一点：help(re) 提供的文档非常全面，编程时都不需要打开浏览器另外查找 re 的文档了。

介绍一下标准库里的 difflib 模块，这个模块可以识别字符串之间的差异，并且提供一个函数 get_close_matches()，基于一组已知的正确答案，帮助解决拼写错误问题（例如，旅游网站上的输入错误提示）。

```
>>> import difflib
>>> capitals = ('Montgomery', 'Juneau', 'Phoenix', 'Little Rock')
>>> difflib.get_close_matches('Fenix', capitals)
['Phoenix']
```

nltk

nltk（自然语言处理工具集）是 Python 中首选的文本分析工具，最初由 Steven Bird 和 Edward Loper 发布。2001 年，Steven Bird 在宾夕法尼亚大学开设了一门自然语言处理（NLP）课程，这个库本是为了帮助学生学习这门课而开发的，后来逐步扩展，覆盖多种自然语言，包含该领域最新研究的各种算法。它以 Apache 2.0 许可证发行，每个月 PyPI 上下载次数超过 10 万次。其创建者还写了一本配套书籍《Python 自然语言处理》可用作 Python 和 NLP 的入门读物。

可以在命令行中使用 pip 安装 nltk[3]，它依赖于 NumPy，所以先安装依赖：

```
$ pip install numpy
$ pip install nltk
```

在 Windows 上，使用 pip 没法成功安装 NumPy，可以尝试按照 Stack Overflow（*http://bit.ly/numpy-install-win*）上提供的操作指南进行安装。下面演示一个短小的示例来证明其使用起来是多么简单。首先，要从 *http://www.nltk.org/data.html* 的语料库集（*http://www.nltk.org/nltk_data/*）中获取一份数据集，这个语料库集包含多种自然语言的标记工具和算法测试数据集。因为语料库与 nltk 分开许可使用，所以要确认选择的语料库使用的是什么许可证。如果知道待下载语料库的名称（在当前这个例子中，目标语料库名称是 punkt 分词器[4]，用来将文本文件内容切分为句子或词），则可以在命令行中这样下载：

```
$ python3 -m nltk.downloader punkt --dir=/user/local/share/nltk_data
```

或者可以在一个交互式会话中下载“（停用词）stopwords”包含一个常用词汇列表，这些词汇在字词计数中常常过于突出，比如许多语言中的“the”“in”或“and”。

```
>>> import nltk
>>> nltk.download('stopwords', download_dir='/usr/local/share/nltk_data')
[nltk_data] Downloading package stopwords to /usr/local/share/nltk_data...
[nltk_data] Unzippingcorpora/stopwords.zip.
True
```

如果不知道想要下载的语料库的名称，那么可以在 Python 解释器中调用 nltk.download() 来启动一个交互式下载器，但它不为函数的第一个参数传值。

```
>>> import nltk
>>> nltk.download(download_dir='/usr/local/share/nltk_data')
```

这样就可以加载感兴趣的数据集进行处理分析了。在下面的代码示例中，我们会加载一份保存的“Python 之禅”文本。

```
>>> import nltk
>>> from nltk.corpus import stopwords
>>> import string
>>>
>>> stopwords.ensure_loaded()   ❶
>>> text = open('zen.txt').read()
```

3 目前在 Windows 上，nltk 看似仅支持 Python 2.7，不过建议在 Python 3 环境中尝试一下，在你阅读本书时，仅支持 Python 2.7 的说法可能已经过时。

4 Punkt 分词器算法由 Tibor Kiss 和 Jan Strunk 于 2006 年提出，是一种与语言无关的语句边界识别方式。举例来说：“Mrs. Smith and Johann S. Bach listened to Vivaldi.”这句可以被正确地识别为单个句子。这个算法必须使用大规模数据集来训练，不过默认的英文分词器已经训练过了。

```
>>> tokens = [
...     t.casefold() for t in nltk.tokenize.word_tokenize(text)     ❷
...     if t not in string.punctuation
... ]
>>>
>>> counter = {}
>>> for bigram in nltk.bigrams(tokens):     ❸
...     counter[bigram] = 1 if bigram not in counter else counter[bigram] +
1
...
>>> def print_counts(counter):
...     for ngram, count in sorted(
...             counter.items(), key=lambda kv: kv[1], reverse=True):   ❹
...         if count > 1:
...             print('{:>25}: {}'.format(str(ngram), '*' * count))     ❺
...
>>> print_counts(counter)
       ('better', 'than'): ********     ❻
         ('is', 'better'): *******
        ('explain', 'it'): **
            ('one', '--'): **
        ('to', 'explain'): **
            ('if', 'the'): **
('the', 'implementation'): **
 ('implementation', 'is'): **
>>>
>>> kept_tokens = [t for t in tokens if t not in stopwords.words()]     ❼
>>>
>>> from collections import Counter     ❽
>>> c = Counter(kept_tokens)
>>> c.most_common(5)
[('better', 8), ('one', 3), ('--', 3), ('although', 3), ('never', 3)]
```

❶ 因为语料库会惰性加载，所以需要加这一句确实加载了停用词语料库。

❷ 分词器要求一个经过训练的模型，即默认的 Punkt 分词器自带了一个针对英文（默认支持的语言）训练过的模型。

❸ 双连词（bigram）是指一对相邻的词。对双连词进行遍历，统计它们出现的次数。

❹ sorted() 函数以统计数值为键，逆向排序。

❺ {:>25} 格式表示右对齐输出字符串，总宽度为 25 个字符。

❻ Python 之禅中出现频率最高的双连词是“better than”。

❼ 为了避免“the”和“is”统计数量过大影响到结果，因此移除了这两个停用词。

❽ 在 Python 3.1 及之后的版本中，可以使用 collections.Counter 来计数。

SyntaxNet

Google 的 SyntaxNet 库，构建在 TensorFlow 之上，提供一个经过训练的英文解析器，名为 Parsey McParseface，以及构建其他模型的框架，只要你有标签数据，也支持其他自然语言。SyntaxNet 库目前仅在 Python 2.7 环境中可用。下载和使用方法的详细说明可参考 SyntaxNet 的 GitHub 主页（*https://github.com/tensorflow/models/tree/master/research/syntaxnet*）。

图像操作

Python 中最流行的三个图像处理操作库分别是：Pillow（Python 图像处理库 PIL 的一个分支，适用于格式转换和简单的图像处理）、cv2（开源计算机视觉库 OpenCV 的 Python 绑定，可用于实时人脸检测和其他高级算法）以及后起新秀 Scikit-Image（提供简单的图像处理功能，附加斑点分析、变形及边角检测等基本功能）。下面几个小节会详细介绍这三个库。

Pillow

Python 图像处理库，简称 PIL，是 Python 中图像操作的核心库之一。最后一次发布于 2009 年，还没移植到 Python 3，幸好后来出现了一个开发活跃的分支项目——Pillow（*http://python-pillow.github.io/*）。这个分支库安装更简单，可运行在各种操作系统上面，支持 Python 3。

安装 Pillow 前，必须先满足 Pillow 的依赖条件。在 Pillow 安装说明页（*https://pillow.readthedocs.org/en/3.0.0/installation.html*）可以找到各平台相关的说明。条件满足后，安装简单直接：

```
$ pip install Pillow
```

下面是一个简短的 Pillow 使用示例（导入的模块名是 PIL 而不是 Pillow）：

```
from PIL import Image, ImageFilter
# 读取图像
im = Image.open('image.jpg')
# 显示图像
im.show()

# 对图像应用一个滤波器
```

```
im_sharp = im.filter(ImageFilter.SHARPEN)
# 将滤波后的图像存为一个新文件
im_sharp.save('image_sharpened.jpg', 'JPEG')

# 按波段切分图像（例如，三原色的红、绿、蓝三个波段）
r,g,b = im_sharp.split()

# 查看图像中嵌入的 EXIF（可交换图像文件格式）数据
exif_data = im._getexif()
exif_data
```

http://bit.ly/opencv-python-tutorial 上的 Pillow 教程包含更多的 Pillow 库使用示例。

cv2

开源计算机视觉库，即大家熟知的 OpenCV，是一个比 PIL 更高级的图像操作处理软件。它以 C 和 C++ 语言编写，专注于实时计算机视觉领域，例如，包含首个应用于实时人脸检测的模型（已使用数千张人脸图像进行训练，*https://github.com/Itseez/opencv/blob/master/samples/python/facedetect.py* 上有个例子演示了如何在 Python 代码中使用这个模型）、一个人脸识别模型、一个人体检测模型及其他模型。多种编程语言都有对应的绑定实现，应用非常广泛。

在 Python 中应用 OpenCV 进行图像处理是通过 cv2 和 NumPy 库实现的。OpenCV3 提供 Python 3.4 及以上版本的绑定，不过 cv2 仍旧是链接依赖 OpenCV2。OpenCV 教程页面（*http://tinyurl.com/opencv3-py-tutorial*）详尽地说明了在 Windows 和 Fedora 系统上 Python 2.7 环境中的安装步骤。在 Mac OS X 系统上，就靠你自己了[5]。OpenCV 教程页面还提供了一个在 Ubuntu 系统上 Python 3 环境下的安装方案（*http://tinyurl.com/opencv3-py3-ubuntu*）。如果安装过程遇到困难，那么也可以下载 Anaconda，使用它来安装。Anaconda 为所有平台都提供了 cv2 二进制安装包。可以打开 *http://tinyurl.com/opencv3-py3-anaconda* 网页查阅《OpenCV3、Python 3 及 Anaconda 即学即用》一文，学习在 Anaconda 环境中使用 cv2 和 Python 3。

下面是 cv2 的一个使用示例。

```
from cv2 import *
import numpy as np
# 读取图像
img = cv2.imread('testimg.jpg')
```

5 按照下面的步骤安装，实测有效：首先，使用 brew install opencv 或 brew install opencv3 --with-python3；然后，按照其他库安装说明进行（比如链接 NumPy）；最后，把包含 OpenCV 共享对象文件（例如：*/usr/local/Cellar/opencv3/3.1.0_3/lib/python3.4/site-packages/*）的目录添加到依赖包查找路径中，或者使用随 virtualenvwrapper 库安装的 add2virtualenvironment，将其添加到虚拟环境中。

```
# 显示图像
cv2.imshow('image', img)
cv2.waitKey(0)
cv2.destroyAllWindows()

# 对图像应用灰度滤波器
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# 将滤波后的图像存为一个新文件
cv2.imwrite('graytest.jpg', gray)
```

更多 Python 实现的 OpenCV 使用示例可以去 *http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_tutorials.html* 上查阅。

Scikit-Image

Scikit-Image 库，后来居上，越来越流行，这部分归功于其源码大半使用 Python 编写，文档也是一绝。虽然它不像 cv2 包含那么多成熟的算法（所以仍然需要 cv2 的算法来处理实时视频这类工作），但对于科研人员来说功能也足够了，比如斑点检测和特征检测，并且它还具备一些标准图像处理工具（比如滤波和对比度调整）。举例来说，Scikit-image 曾被用于冥王星的小卫星图像合成（*https://blogs.nasa.gov/pluto/2015/10/05/plutos-small-moons-nix-and-hydra/*）。Scikit-Image 项目主页上（*http://scikit-image.org/docs/dev/auto_examples/*）还有更多的示例。

第 11 章

数据持久化

本章内容主要是关于关系型数据库的 Python 接口库。关系型数据库将结构化数据存储在数据表中，并使用 SQL[1] 来访问。

结构化文件

第 9 章已经提过 JSON、XML 及 ZIP 文件的解析生成工具，并在讨论序列化话题时论及 pickle 和 XDR 这两种序列化方式。在此，我们推荐使用 PyYAML（通过 pip install pyyaml 获取）来解析 YAML 文件。针对 CSV 文件、某些 FTP 客户端使用的 *.Netrc 文件、Mac OS X 系统使用的 *.plist 文件，Python 标准库都包含对应的处理工具，并以 configparser[2] 库来支持 Windows INI 的一种衍生格式。

另外，Python 标准库还包含用 shelve 模块实现一种持久化键值存储，其底层支撑是计算机上可用性最好的一个数据库管理器变种（dbm，一个键值数据库）[3]。

```
>>> import shelve
>>>
>>> with shelve.open('my_shelf') as s:
```

1 1970 年，埃德加 • 弗兰克 • 科德（Edgar F. Codd）在 IBM 工作，发表了论文《大型共享数据库的关系型数据模型》（*http://bit.ly/relational-model-data*），首次提出关系型数据库模型。不过这个理论一度被大家忽视，直到 1977 年，拉里 • 埃里森（Larry Ellison）基于这一理论创办了一家公司（最后成了甲骨文（Oracle）公司）。关系型数据库兴起之后，同领域的其他思想，比如键值存储和层次数据库模型，被普遍忽视，直到近几年的 NoSQL 运动，非关系型存储方案才开始在集群计算系统中流行起来。

2 即 Python 2 中的 ConfigParser，阅读 Configparser 官方文档（*https://docs.python.org/3/library/configparser.html#supported-ini-file-structure*）了解该解析器定义的 INI 变种格式。

3 dbm 使用持久化在磁盘上的哈希表来存储键值对。具体怎么存储要看使用的后端是 gdbm、ndbm，还是 dumb。其中 dumb 使用 Python 实现，dbm 的官方文档中有相关介绍。其他两个后端实现的介绍见 gdbm 手册（*http://www.gnu.org.ua/software/gdbm/manual//gdbm.html*）。使用 ndbm 值占用的存储空间有上限。为写入数据，打开文件时会对文件加锁，除非（仅使用 gdbm 时如此）以 ru 或 wu 属性打开数据库文件，并且这样打开写模式下的数据更新操作对于其他数据库连接可能不可见。

```
...     s['d'] = {'key': 'value'}
...
>>> s = shelve.open('my_shelf', 'r')
>>> s['d']
{'key': 'value'}
```

检查一下底层使用的是哪个数据库。

```
>>> import dbm
>>> dbm.whichdb('my_shelf')
'dbm.gnu'
```

还可以从 *http://gnuwin32.sourceforge.net/packages/gdbm.htm* 上获取 dbm 针对 Windows 平台的 GNU 实现，对于其他平台，可以先查看系统的包管理器（brew、apt、yum），不可得再尝试从 *http://www.gnu.org.ua/software/gdbm/download.html* 上获取 dbm 源码。

数据库接口库

Python 数据库 API（DB-API2）定义了 Python 中数据库访问的标准接口。PEP 249（*https://www.python.org/dev/peps/pep-0249/*）是文档说明，*http://halfcooked.com/presentations/osdc2006/python_databases.html* 网上介绍得更详细。几乎所有 Python 数据库驱动都遵照这个接口，因此在 Python 程序中若想查询数据库，根据正在使用的数据库选择对应的驱动库连接数据库即可。例如，sqlite3 对应 SQLite 数据库、psycopg2 对应 Postgres、MySQL-python 对应 MySQL。[4]

如果代码中编写了大量 SQL 字符串、硬编码列名和表名，那么代码很快就会变得混乱、易错并且难以调试。表 11-1 中罗列的库（因为 sqlite3 是 SQLite 的驱动库，所以 sqlite3 除外）都提供了一个数据库抽象层（DAL），将 SQL 的结构、语法及数据类型抽象成 API。

Python 是一门面向对象编程语言，数据库抽象层也可以实现对象关系映射（ORM），提供 Python 对象和底层数据库之间的映射，并为这些类附加属性运算符，以 Python 代码来表现一个抽象版本的 SQL。

表 11-1 中所有的库（除 sqlite3 和 Records 外）都提供一个 ORM，在实现上使用两个模式之一[5]：活动记录（Active Record）模式，记录兼顾表现抽象后的数据并与数据库交互；数据映射（Data Mapper）模式，其中一层与数据库交互，另一层表现数据两层之间是一

4 结构化查询语言（SQL）是一个 ISO 标准，不过数据库供应商可以选择实现部分标准，也可以自己添加特性。这意味着数据库驱动 Python 库必须能够理解目标数据库使用的 SQL 方言。

5 定义见马丁·福勒的《企业应用架构模式》一书。进一步了解 Python ORM 的设计，推荐阅读《开源软件架构》中关于 SQLAlchemy 的一章，全栈 Python 网站上有 Python ORM 相关资源的详细清单（*https://www.fullstackpython.com/object-relational-mappers-orms.html*）。

个映射函数，执行必要的转换逻辑（本质上是在数据库之外执行一个 SQL 视图的逻辑）。

执行数据库查询时，活动记录和数据映射模式的行为一致，不过在数据映射模式中，开发者必须显式声明数据表名，或添加主键，或创建辅助数据表以支持多对多的关系（比如在一张收据上，一个交易 ID 可能会关联多个购买品），如果使用活动记录模式，这一切都是在内部自动完成的。

这些库中最流行的是：sqlite3、SqlAlchemy 及 Django ORM。Records 自成一派，更像是一个 SQL 客户端，为格式化输出提供了很多选项，而其他的库则可以认为是 Django ORM 下层部分一个独立的更轻量的版本（因为它们都使用活动记录模式），但实现不同，API 区别也非常大。

表11-1　数据库相关的Python开发库

库名称	许可证	使用理由
sqlite3（驱动库，不是 ORM）	PSFL	• 包含在标准库中 • 适用于中小流量站点，这类站点仅需要简单的数据类型，数据库查询少，没有网络通信开销，数据库查询延迟低 • 适用于学习 SQL 或 Python 的 DB-API，或者快速构建一个数据库应用原型
SQLAlchemy	MIT 许可证	• 实现数据映射模式，提供一套两层 API。上面 ORM 层类似其他库中的 API，下面数据表层直接与数据库关联 • 开发者可以显式控制（通过下层经典映射 API）数据库的结构和模式；这一点有时大有用处，例如，如果数据库管理员和 Web 开发者不是同一批人 • 支持的 SQL 方言：SQLite、PostgreSQL、MySQL、Oracle、MS-SQL Server、Firebird 及 Sybase（也可以注册自定义的 SQL 分支）
Django ORM	BSD 许可证	• 实现活动记录模式，可以从应用中定义的模型隐式生成数据库基础结构 • 与 Django 紧耦合 • 支持的 SQL 方言：SQLite、PostgreSQL、MySQL 及 Oracle。或者使用第三方库来支持：SAP SQL Anywhere、IBM DB2、MS-SQL Server、Firebird 及 ODBC

续表

库名称	许可证	使用理由
peewee	MIT 许可证	• 实现活动记录模式，在 ORM 中定义的数据表即是在数据库看到的数据表（另加一个索引列） • 支持的 SQL 方言：SQLite、MySQL 及 Postgres（也可以添加自定义的）
PonyORM	AGPLv3	• 实现活动记录模式，基于生成器的查询语法非常直观 • 还提供一个在线图形化的实体，关系图表编辑器（绘制数据模型，定义数据库中的数据表及它们之间的相互关系），可以将绘制的图表翻译成 SQL 代码，用来创建数据表 • 支持的 SQL 方言：SQLite、MySQL、Postgres 及 Oracle（也可以添加自定义的）
SQLObject	LGPL	• Python 中首批使用活动记录模式的开发库之一 • 支持的 SQL 方言：SQLite、MySQL、Postgres、Firebird、Sybase、MAX DB、MS-SQL Server（也可以添加自定义的）
Records（查询接口，不是 ORM）	ISC 许可证	• 提供一种简单的数据库查询方式，并支持生成各种报告文档：输入 SQL、输出 XLS（或 JSON、YAML、CSV、LaTex） • 提供一个命令行接口，可用于交互式查询或一行式报告生成 • 使用强大的 SQLAlchemy 作为后盾支撑

下面会详细介绍表 11-1 中罗列的库。

sqlite3

sqlite 是一个 C 库，在 sqlite3 背后提供数据库能力，以单个文件存储数据库，通常文件扩展类型为 *.db。*https://www.sqlite.org/whentouse.html* 网的“何时使用 sqlite”的页面上声称：对于每天请求量成千上万的站点，使用 sqlite 作为数据库后端被证明是可行的。官网上还提供了 sqlite 支持的 SQL 命令列表，开发者也可以查阅 W3School 的 SQL 快速参考（*http://www.w3schools.com/sql/sql_quickref.asp*）学习使用。如下是一个示例：

```
import sqlite3
db = sqlite3.connect('cheese_emporium.db')

db.execute('CREATE TABLE cheese(id INTEGER, name TEXT)')
db.executemany(
    'INSERT INTO cheese VALUES (?, ?)',
    [(1, 'red leicester'),
```

```
        (2, 'wensleydale'),
        (3, 'cheddar'),
       ]
)
db.commit()
db.close()
```

sqlite 支持的数据类型包括：NULL、INTEGER、REAL、TEXT 和 BLOB（字节序列），开发者也可以按照 sqlite3 文档说明做一些额外的工作来注册新的数据类型（例如，实现一个 datetime.datetime 类型，存储为 TEXT 类型）。

SQLAlchemy

SQLAlchemy 是一个非常流行的数据库工具包。Django 自带选项支持从自己的 ORM 切换到 SQLAchemy，Flask 大教程（*http://bit.ly/pandas-sql-query*）中构建的博客程序使用 SQLAlchemy 来访问数据库，Pandas 使用它作为 SQL 后端。

本章列举的库中仅有 SQLAlchemy 采用 Martin Fowler 的数据映射模式，而非实际实现中更常见的活动记录模式。与其他库不同，SQLAlchemy 不仅提供一个 ORM 层，而且提供一套一般化的 API（称为 Core 层）来编写不涉及 SQL 的与数据库类型无关的代码。ORM 层基于 Core 层，Core 层使用 Table 对象直接映射底层数据库。因为这些对象和 ORM 之间的映射需要开发者显式地指定，所以一开始相对要编写更多的代码，对于刚接触关系型数据库的人来说可能会有些挫败感，但好处是它能全面精细地控制数据库。

SQLAlchemy 可以在 Jython 和 PyPy 环境中运行，也支持从 Python 2.5 到 3.x 全部版本。下面的代码片段将演示创建一个多对多对象映射的必要工作。我们将在 ORM 层创建三类对象：Customer（顾客）、Cheese（奶酪）和 Purchase（购买单）。一个顾客可以进行多次购买（多对一的关系），每次可以购买多种奶酪（多对多的关系）。描述这样的细节是为了说明未被映射的数据表 purchases_cheeses 无须出现在 ORM 层中，因为其目的仅是提供奶酪种类和购买单之间的关联关系。其他 ORM 一般会在背后默默地创建这个数据表，因此这里也体现了 SQLAlchemy 与其他库的一个重大不同之处。

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Date, Integer, String, Table, ForeignKey
from sqlalchemy.orm import relationship

Base = declarative_base()                                    ❶

class Customer(Base):                                        ❷
    __tablename__ = 'Customers'
    id = Column(Integer, primary_key=True)
```

```
    name = Column(String, nullable=False)
    def __repr__(self):
        return "<Customer(name='%s')>" % (self.name)

purchases_cheeses = Table(                    ❸
    'purchases_cheeses', Base.metadata,
    Column('purch_id', Integer, ForeignKey('purchases.id', primary_
key=True)),
    Column('cheese_id', Integer, ForeignKey('cheeses.id', primary_key=True))

class Cheese(Base):                  ❹
    __tablename__ = 'cheeses'
    id = Column(Integer, primary_key=True)
    kind = Column(String, nullable=False)
    purchases = relationship(                     ❺
        'Purchase', secondary='purchases_cheeses', back_populates='cheeses'❻
    )
    def __repr__(self):
        return "<Cheese(kind='%s')>" % (self.kind)

class Purchase(Base):
    __tablename__ = 'purchases'
    id = Column(Integer, primary_key=True)
    customer_id = Column(Integer, ForeignKey('customers.id', primary_
key=True))
    purchase_date = Column(Date, nullable=False)
    customer = relationship('Customer')
    cheeses = relationship(                   ❼
        'Cheese', secondary='purchases_cheeses',
        back_populates='purchases'
    )
    def __repr__(self):
        return ("<Purchase(customer='%s', dt='%s')>" % (self.customer.name,
self.purchase_date))
```

❶ 此处的声明式基类对象是一个元类[6]，对ORM层中每个已映射表的创建过程进行拦截，并在Core层定义一个对应的表。

❷ ORM层中的对象继承自声明式基类。

❸ 这是Core层中的一个未映射表，它并非一个类，也非衍生自声明式基类。其对应数据库中的数据表purchases_cheeses，为提供奶酪和购买单之间多对多映射关系而存在。

❹ 对应ORM层中的一个已映射表Cheese。内部在Core层会创建Cheese.__table__。其

6 对于Python元类，Stack Overflow上有一个非常不错的讲解，参见*http://stackoverflow.com/a/6581949*。

在数据库中对应一个名为 cheeses 的数据表。

❺ relationship 显式地定义了已映射类 Cheese 和 Purchase 之间的关系：它们通过一个辅助数据表 purchases_cheeses 间接关联（相当于通过一个 ForeignKey 直接关联）。

❻ back_populates 添加一个事件监听器，当一个新的 Purchase 对象被加到 Cheese.purchases 中，这个 Cheese 对象也会出现在 Purchase.cheeses 中。

❼ 这是多对多关系的另一半设置。

声明式基类会显式地创建所有数据表。

```
from sqlalchemy import create_engine
engine = create_engine('sqlite://')
Base.metadata.create_all(engine)
```

现在，使用 ORM 层中的对象与数据库进行交互操作，看起来和使用其他 ORM 库是一样的。

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
sess = Session()

leicester = Cheese(kind='Red Leicester')
camembert = Cheese(kind='Camembert')
sess.add_all((camembert, leicester))
cat = Customer(name='Cat')
sess.add(cat)
sess.commit()                                    ❶

import datetime
d = datetime.date(1971, 12, 18)
p = Purchase(purchase_date=d, customer=cat)
p.cheeses.append(camembert)                      ❷
sess.add(p)
sess.commit()
```

❶ 必须显式地调用 commit() 方法将变更推送到数据库。

❷ 对象实例化期间不会将关联对象添加到多对多关系属性中，必须在实例化之后附加关联对象。

下面是几个数据库查询示例（译注：根据上面的代码片段，下面的示例输出有误）。

```
>>> for row in sess.query(Purchase,Cheese).filter(Purchase.cheeses):    ❶
```

```
...     print(row)
...
(<Purchase(customer='Douglas', dt='1971-12-17')>, <Cheese(kind='Camembert')>)
(<Purchase(customer='Douglas', dt='1971-12-17')>, <Cheese(kind='Red
Leicester')>)
(<Purchase(customer='Cat', dt='1971-12-18')>, <Cheese(kind='Camembert')>)
>>>
>>> from sqlalchemy import func
>>> (sess.query(Purchase,Cheese)                    ❷
...     .filter(Purchase.cheeses)
...     .from_self(Cheese.kind, func.count(Purchase.id))
...     .group_by(Cheese.kind)
... ).all()
[('Camembert', 2), ('Red Leicester', 1)]
```

❶ 此处演示了如何通过 purchases_cheeses 数据表做多对多交叉查询，这个数据表并未映射到一个上层 ORM 对象上。

❷ 这个查询是统计每种奶酪的购买次数。

进一步学习，请打开 *http://docs.sqlalchemy.org/en/rel_1_0/* 网页阅读 SQLAlchemy 文档。

Django ORM

Django ORM 是为 Django 提供数据库访问能力的接口。对于活动记录模式的实现，在本章罗列的库中，它可能是和 Ruby on Rails ActiveRecord 库最接近的一个。

因为它与 Django 紧密集成，所以通常在构建 DjangoWeb 应用时才会用到它。如果想要在构建一个 Web 应用的同时学习 Django ORM，那么可以尝试 Django Girls 组织提供的 Django ORM 教程。[7]

如果只是想尝试一下 Django 的 ORM，而不是构建一个完整的 Web 应用，则可以复制 GitHub 上的一个骨架项目（它仅使用了 Django 的 ORM，项目参见 *https://github.com/mick/django_orm_only*），并按照说明进行尝试。不同版本的 Django 之间可能会有些许不同。settings.py 文件内容如下所示：

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'tmp.db',
    }
```

7 Django Girls 是一个现象级的公益组织，聚集了很多杰出的程序员，在欢乐的氛围中，为全世界的女性提供免费的 Django 技术培训。

```
}
INSTALLED_APPS = ('orm_only',)
SECRET_KEY = 'A secret key may also be required.'
```

Django ORM 中每个抽象数据表类都要继承自 Django 的 Model 对象，如下所示。

```
from django.db import models

class Cheese(models.Model):
    type = models.CharField(max_length=30)

class Customer(models.Model):
    name = models.CharField(max_length=50)

class Purchase(models.Model):
    purchase_date = models.DateField()
    customer = models.ForeignKey(Customer)          ❶
    cheeses = models.ManyToManyField(Cheese)        ❷
```

❶ ForeignKey 表示一个多对一的关系，顾客可以多次购买，但一次购买只会和单个顾客相关。对于一对一关系，则使用 OneToOneField。

❷ ManyToManyField 用来表示一个多对多的关系。

我们必须执行一个命令来构建真实的数据表。在命令行中，激活虚拟环境，进入 manage.py 文件所在目录，输入：

```
(venv)$ python manage.py migrate
```

数据表创建好之后，再来看看如何添加数据到数据库。如果不调用 instance.save() 方法，那么新数据行就不会真的被存入数据库。

```
leicester = Cheese.objects.create(type='Red Leicester')
camembert = Cheese.objects.create(type='Camembert')
leicester.save()                                    ❶
camembert.save()

doug = Customer.objects.create(name='Douglas')
doug.save()

# 添加一次购买记录
import datetime
now = datetime.datetime(1971, 12, 18, 20)
day = datetime.timedelta(1)

p = Purchase(purchase_date=now - 1 * day, customer=doug)
```

```
p.save()
p.cheeses.add(camembert, leicester)                    ❷
```

❶ 数据表对象实例必须保存才会被添加到数据库，并且若想在交叉引用其他对象的插入操作中使用某个数据表对象，也必须先保存这个对象才行。

❷ 必须将对象添加到多对多映射关系中。

Django 中通过 ORM 查询数据库的方式如下所示。

```
# 过滤得到发生在过去 7 天内的所有购买记录
queryset = Purchase.objects.filter(purchase_date__gt=now - 7 * day)    ❶

# 展示查询结果集内谁买了哪种奶酪
 for v in queryset.values('customer__name', 'cheeses__type'):    ❷
    print(v)

# 统计各类奶酪的售卖次数
from django.db.models import Count
sales_counts = (                                                ❸
    queryset.values('cheeses__type')
    .annotate(total=Count('cheeses'))                           ❹
    .order_by('cheeses__type')
)
for sc in sales_counts:
    print(sc)
```

❶ 在 Django 中，过滤操作符如 gt（大于）跟在双下画线之后添加到表属性名（如 purchase_date）后面。Django 内部会进行解析。

❷ 外键标识符后的双下画线表示将访问对应数据表的属性。

❸ 可能你没见过这种写法，这里提示你：可以在一个长语句外加圆括号，将长语句拆成多行，可读性更好。

❹ 查询结果集的 annotate 子句会为每个结果添加额外的字段。

peewee

peewee 的主要目标是为使用 SQL 的人提供一个轻量的数据库交互方式。所见即所得（既不需要像 SQLAlchemy 那样，对背后的数据表结构手动构建一个抽象顶层，也不会像 Django ORM 那样在抽象数据表下构建一个底层）。其目标是满足不同于 SQLAlchemy 的技术生态需求，虽然能做的事情不会太多，但做得更快、方式更简单、更符合 Python 风格。

Peewee 不仅可以在开发者未指定主键时隐式地为数据表创建主键，而且可以像这样创建

一个数据表：

```
import peewee
database = peewee.SqliteDatabase('peewee.db')

class BaseModel(peewee.Model):
    class Meta:                                    ❶
        database = database                        ❷

class Customer(BaseModel):
    name = peewee.TextField()                      ❸

class Purchase(BaseModel):
    purchase_date = peewee.DateField()
    customer = peewee.ForeignKeyField(Customer, related_name='purchases')❹

class Cheese(BaseModel):
    kind = peewee.TextField()

class PurchaseCheese(BaseModel):
    """ 设置多对多关系 """
    purchase = peewee.ForeignKeyField(Purchase)
    cheese = peewee.ForeignKeyField(Cheese)

database.create_tables((Customer, Purchase, Cheese, PurchaseCheese))
```

❶ peewee 将模型的配置细节保存在一个名为 Meta 的命名空间中，这个思路借鉴自 Django。

❷ 将每个模型都关联到一个数据库。

❸ 如果未显式地添加一个主键，则会被隐式地自动添加。

❹ 这行代码实现：在不改变数据表的前提下，访问获取 Customer 记录时，自动为记录添加属性 purchases。

调用 create() 方法仅需一步就能完成数据初始化并将数据存入数据库，也可以先初始化数据，再存入数据库。peewee 提供一些配置选项控制是否自动提交，并提供一些工具进行事务操作。如下示例演示了如何一步到位。

```
leicester = Cheese.create(kind='Red Leicester')
camembert = Cheese.create(kind='Camembert')
cat = Customer.create(name='Cat')

import datetime
```

```
d = datetime.date(1971, 12, 18)

p = Purchase.create(purchase_date=d, customer=cat)  ❶
PurchaseCheese.create(purchase=p, cheese=camembert) ❷
PurchaseCheese.create(purchase=p, cheese=leicester)
```

❶ 直接添加保存一个对象（如 cat），并且 peewee 会为其生成主键。

❷ 手动添加新的条目。

查询方式如下所示：

```
>>> for p in Purchase.select().where(Purchase.purchase_date > d - 1 * day):
...     print(p.customer.name, p.purchase_date)
...
Douglas 1971-12-18
Cat 1971-12-19
>>>
>>> from peewee import fn
>>> q = (Cheese
...      .select(Cheese.kind, fn.COUNT(Purchase.id).alias('num_purchased'))
...      .join(PurchaseCheese)
...      .join(Purchase)
...      .group_by(Cheese.kind)
... )
>>> for chz in q:
...     print(chz.kind, chz.num_purchased)
...
Camembert 2
Red Leicester 1
```

peewee 有许多扩展可用，包括支持高级事务[8]、支持为数据添加自定义函数钩子，它们都在存储之前执行，例如压缩或哈希。

PonyORM

与前述的开发库不同，PonyORM 采用的查询语法不是编写类 SQL 语言或布尔表达式，而是使用 Python 的生成器语法。它还提供了一个图形化的模式编辑器，可为开发者自动生成 PonyORM 实体。PonyORM 支持 Python 2.6 和 Python 3.3。

为实现其直观的语法，Pony 要求数据表之间的所有关系都是双向的，所有相关数据表必须显式地相互引用，如下所示：

```
import datetime
```

8 事务上下文允许中间步骤发生错误时进行回滚操作。

```
from pony import orm

db = orm.Database()
db.bind('sqlite', ':memory:')

class Cheese(db.Entity):                          ❶
    type = orm.Required(str)                      ❷
    purchases = orm.Set(lambda: Purchase)         ❸

class Customer(db.Entity):
    name = orm.Required(str)
    purchases = orm.Set(lambda: Purchase)         ❹

class Purchase(db.Entity):
    date = orm.Required(datetime.date)
    customer = orm.Required(Customer)             ❺
    cheeses = orm.Set(Cheese)                     ❻

db.generate_mapping(create_tables=True)
```

❶ 一个 Pony 数据库实体（Entity）在数据库中存储一个对象的状态，通过它的存在将数据库和对象连接在一起。

❷ Pony 使用标准的 Python 类型（如 str 和 datetime.datetime）来标识数据列的类型。此外，也可以使用用户自定义的实体，如 Purchase、Customer 及 Cheese。

❸ 这里使用了 lambda: Purchase，因为还没定义 Purchase。

❹ orm.Set(lambda: Purchase) 是 Customer 和 Purchase 之间一对多关系的前半部分定义。

❺ orm.Required(Customer) 是 Customer 和 Purchase 之间一对多关系的后半部分定义。

❻ orm.Set(cheese) 和标 ❸ 中的 orm.Set(lambda: Purchase) 共同定义了一个多对多的关系。

数据实体定义好之后，对象实例化过程看起来就和其他库差不多了。随手创建几个实体实例，并调用 orm.commit() 将数据提交到数据库。

```
camembert = Cheese(type='Camembert')
leicester = Cheese(type='Red Leicester')
cat = Customer(name='Cat')
doug = Customer(name='Douglas')

d = datetime.date(1971, 12, 18)
day = datetime.timedelta(1)
Purchase(date=(d - 1 * day), customer=doug, cheeses={camembert, leicester})
```

```
Purchase(date=d, customer=cat, cheeses={camembert})
orm.commit()
```

然后进行查询。Pony 一展身手的时候到了，下面的代码看起来真的是纯粹的 Python 代码。

```
yesterday = d - 1.1 * day
for cheese in (
        orm.select(p.cheeses for p in Purchase if p.date > yesterday)      ❶
    ):
    print(cheese.type)
for cheese, purchase_count in (
        orm.left_join((c, orm.count(p))  ❷
        for c in Cheese
        for p in c.purchases)
    ):
    print(cheese.type, purchase_count)
```

❶ 这就是使用 Python 生成器语法编写一个查询语句的样子。

❷ orm.count() 函数按计数聚合。

SQLObject

SQLObject 首次发布于 2002 年 10 月，是本章库列表中最老的一个 ORM。它实现了活跃记录模式，并最早提出一个新奇的理念——以标准操作符（如 ==、<、<= 等）重载将某些 SQL 逻辑抽象成 Python 操作。现在几乎所有的 ORM 库都实现了这一理念，这两个特点曾经让 SQLObject 十分流行。

虽然 SQLObject 支持多种数据库（常见数据库系统 MySQL、Postgres 和 SQLite，以及其他不那么常见的数据库系统如 SAP DB、SyBase 和 MSSQL），但目前仅支持 Python 2.6 和 Python 2.7。其开发维护仍然活跃，但在 SQLAlchemy 出现之后，就没那么流行了。

Records

Records 是一个精简的 SQL 库，专为使用原生 SQL 查询数据库而设计，支持多种数据库。其主要是把 Tablib 和 SQLAlchemy 捆绑在一起，提供一套友好的 API 和一个命令行应用。这个命令行应用使用起来像一个 SQL 客户端，不过以 YAML、XLS 和其他 Tablib 格式输出。Records 绝非意在替代 ORM 库。其典型用例是查询数据库并创建一个报告（例如，把近期销售数据存为一份电子表格的月报）。数据可以以编程的方式来使用，或者导出成多种好用的数据格式。

```
>>> import records
>>> db = records.Database('sqlite:///mydb.db')
```

```
>>>
>>> rows = db.query('SELECT * FROM cheese')
>>> print(rows.dataset)
name         |price
-------------|-----
red leicester|1.0
wensleydale  |2.2
>>>
>>> print(rows.export('json'))
[{"name": "red leicester", "price": 1.0}, {"name": "wensleydale", "price":
2.2}]
```

Records 也包含一个命令行工具，使用 SQL 来导出数据，像这样：

```
$ records 'SELECT * FROM cheese' yaml --url=sqlite:///mydb.db
- {name: red leicester, price: 1.0}
- {name: wensleydale, price: 2.2}

$ records 'SELECT * FROM cheese' xlsx --url=sqlite:///mydb.db > cheeses.xlsx
```

NoSQL 数据库接口库

NoSQL（Not only SQL）数据库涵盖非传统关系型数据库的所有类型数据库。在 PyPI 上查找 NoSQL 数据库的接口库，会面对许多命名相似的 Python 包，你可能会有些迷惑，不知该选择哪个。推荐针对目标 NoSQL 数据库的项目主站，搜索关键词 Python，获知哪个 Python 接口库最佳（例如，Google 搜索 Python site:vendorname.com）。大多数 NoSQL 数据库主站都提供了 Python API 库，以及如何使用的快速入门教程，列举如下所示。

MongoDB

> MongoDB 是一个分布式文档存储系统，可以将其视为存放在一个集群上的巨大的 Python 字典。它具备自己的过滤查询语言。学习其 Python API，可查阅 *https://docs.mongodb.com/getting-started/python/* 页面上的 MongoDB 入门（Python 版）。

Cassandra

> Cassandra 是一个分布式表存储系统。它提供快速查找能力，可承受列项很多的宽表，但不支持 join 操作。确切地说，其范式是：以不同的列为键，为数据设计多个副本视图。学习其 Python API，可查阅 planet Cassandra 页面（*http://www.planetcassandra.org/apache-cassandra-client-drivers/*）。

HBase

HBase 是一个分布式列存储系统（在此上下文中，列存储表示数据像这样存储 (行 ID、列名、值)，可以存储非常稀疏的数组数据，如 Web 站点“进入”和“出去”链接的数据集）。HBase 构建在 Hadoop 分布式文件系统上。学习它的 Python API，可查阅 HBase 的支持项目页面（*https://hbase.apache.org/supportingprojects.html*）。

Druid

Druid 是一个分布式列存储系统，专门用于收集（也可选在数据存储之前做聚合）事件数据（在此上下文中，列存储表示数据列可以排序，并且数据存储也可以压缩，以求更快的 I/O 和更小的存储空间）。Druid Python API 库的 GitHub 链接是 *https://github.com/druid-io/pydruid*。

Redis

Redis 是一个分布式内存键值存储系统，其要点是以避免不必要的磁盘 I/O 来降低数据读写延迟。例如：存储频繁出现的查询结果，加速 Web 请求。*http://redis.io/clients#python* 网页有 Redis 的 Python 客户端清单，其中强调 redis-py 是官方推荐的接口库，*https://github.com/andymccurdy/redis-py* 上有 redis-py 项目。

Couchbase

Couchbase 也是一个分布式文档存储系统。与 MongoDB 类 JavaScript 的 API 相比，Couchbase 的 API 更类似 SQL。Couchbase 的 Python SDK 参见 *http://developer.couchbase.com/documentation/server/current/sdks/python-2.0/introduction.html*。

Neo4j

Neo4j 是一个图数据库，专门用于存储类图关系的对象。Neo4j 的 Python 指南参见 *http://neo4j.com/developer/python/*。

LMDB

LMDB 是 Symas 公司的内存映射数据库，它速度很快。它是一个以内存映射文件实现键值存储的数据库，这意味着不需要从头开始读文件一直读到目标数据所在位置，因此性能接近内存存储的速度。其 Python 绑定库参见 *https://lmdb.readthedocs.io/*。

附录 A

补充说明

Python 社区

Python 社区丰富多彩、包容、覆盖全球，多元化发展。

BDFL

Guido van Rossum 是 Python 的创造者，经常被戏称为 BDFL，即仁慈的独裁者。

Python 软件基金会

Python 软件基金会（PSF）的使命是促进和保护 Python 编程语言的发展，帮助 Python 程序员社区多元化、国际化发展。进一步了解，请访问 PSF 的主页（*http://www.python.org/psf/*）。

PEP

PEP 是 Python 增强提案（Python Enhancement Proposal）的缩写。提案围绕 Python 语言本身或者相关标准描述变更方案。对 Python 历史或者对语言的大体设计感兴趣的人会发现这些提案非常有意思，那些最终被拒绝的提案也不例外。PEP 1（*https://www.python.org/dev/peps/pep-0001*）中定义了三种不同的 PEP 类型。

1．标准类 PEP

描述一个新特性或实现。

2．信息类 PEP

向社区描述一个设计议案、一般性准则或发布信息。

3．流程类 PEP

描述一个 Python 相关的流程。

值得关注的 PEP

如下 PEP 值得一读。

1.PEP 8——Python 代码风格指南

从头到尾读一读，并遵守该指南。工具 pep8 能助你一臂之力。

2.PEP 20——Python 之禅

PEP 20 是 19 条语录的列表，言简意赅地阐述了 Python 背后的哲学。

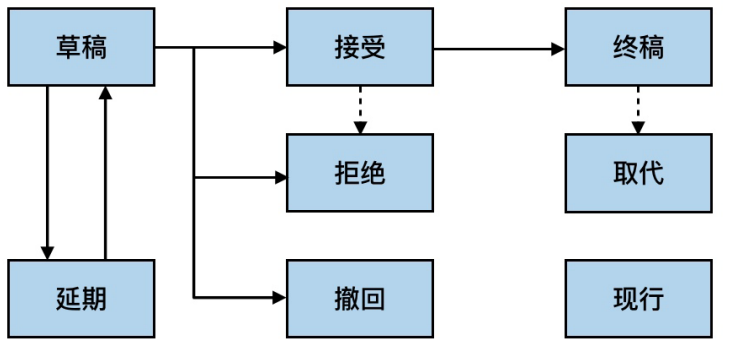
3.PEP 257——文档字符串约定

PEP 257 包含 Python 文档字符串相关语义和约定的指导原则。

通过 PEP 索引（*http://www.python.org/dev/peps/*）可以进一步阅读更多内容。

如何提交一个 PEP

PEP 会经过同行评审，在进行充分讨论之后，做出接受或拒绝的决定。任何人都可以编写一份 PEP 提交评审。图 A-1 中的流程阐明了一份 PEP 草稿提交后的评审流程。



图A-1　PEP评审流程一览

Python 相关会议

Python 社区的一个重大活动是开发者大会。其中最知名的两大会议是，在美国举行的 PyCon，和在欧洲举行的同类会议 EuroPython。Python 相关会议的详细列表参见 *http://www.pycon.org/*。

Python 用户组

用户组的 Python 开发者们会聚在一起就感兴趣的 Python 话题发表演说或进行讨论。Python 软件基金会的 wiki（*http://wiki.python.org/moin/LocalUserGroups*）上维护了本地化用户组的列表。

Python 学习材料

以下是推荐的一些参考资料，按难易级别和应用方向分类。

初阶

1.Python *官方教程*

它（*http://docs.python.org/tutorial/index.html*）是 Python 的官方教程，涵盖所有基础知识点，并提供语言和标准库概览。如需语言新手指南，推荐阅读。

2.Python *初学者教程*

该教程（*http://thepythonguru.com/*）专为新手准备，深入介绍了许多 Python 概念，也会教你一些高级的 Python 概念，如 lambda 表达式和正则表达式。它以如何使用 Python 访问 MySQL 数据库收尾。

3. *学习* Python

该交互式教程（*http://www.learnpython.org/*）让你轻松愉悦地入门 Python。其使用方式与流行网站尝试 Ruby（*http://tryruby.org/*）相同，站点内建一个交互式的 Python 解释器，这样用户无须在本地安装 Python 就能学习课程。

4.《Python 你我他》

这本书是极佳的学习资源，从中可以学习 Python 语言的方方面面，适合更喜欢通过传统书籍学习的人。

5. Python 在线辅导

这个网站（*http://pythontutor.com/*）可视化展现程序如何运行，通过理解计算机执行程序每行源代码后发生的事情，帮助初学者克服学习编程时遇到的根本性障碍。

6.《用 Python 构建自己的计算机游戏》

这本书专为毫无编程经验的人准备。每一章都会提供一个游戏的源代码，这些示例程序用于演示一些编程概念，让读者直观感受程序。

7.《使用 Python 破解密码》

这本书教初学者学习 Python 编程和一些基本的密码技术。书中提供了多种加密方法的源码，以及破解这些加密方法的程序。

8. 笨方法学 Python

它（*http://learnpythonthehardway.org/book/*）是一份绝佳的 Python 编程初学者指南，涵盖从终端控制台到 Web 各方面的入门示例程序。

9.Python 速成

该网站（*http://stephensugden.com/crash_into_python/*）也被称为“三小时学会 Python 编程”，为具备其他语言编程经验的开发者提供了一门 Python 速成课程。

10.《深入学习 Python 3》

这本书适合那些已准备投入 Python 3 怀抱的人。如果你正从 Python 2 迁移到 Python 3，或者已有其他语言的编程经验，那么这本书是一份不错的阅读材料。

11.《像计算机科学家一样思考 Python》

这本书尝试使用 Python 语言来介绍计算机科学的一些基本概念。其目标是编写成：配备大量练习、少量的专业术语，每章都有一节专门讲解代码调试的书。这本书探讨了 Python 语言的各种特性，交叉介绍了多种设计模式和最佳实践。

这本书还包含了若干应用案例，通过将书中讨论的话题应用到实例中，帮助读者更深入地学习这些主题。应用案例包括：设计一个图形用户界面（GUI）和马尔柯夫链分析法。

12.Python Koans 在线教程

Edgecase 公司编写的 Ruby Koans 在线教程颇受欢迎，Python Koans 在线教程（*http://*

bitbucket.org/gregmalcolm/python_koans）是同类教程的 Python 版。它是一个交互式基于命令行的教程，使用一种测试驱动的方式教授基本的 Python 概念：通过逐个解决测试脚本中执行失败的断言语句，循序渐进地学习 Python。

对于习惯于自己解决谜题的人来说，这是一个有趣而吸引人的方式。对于 Python 编程新手而言，多一份学习资源或参考资料也不是一件坏事。

13.《简明 Python 教程》

这是一本免费的入门书，教一些初级的 Python 知识，它适合没有任何编程经验的读者。该教程分为 Python 2.x（*http://www.ibiblio.org/swaroopch/byteofpython/read/*）和 Python 3.x（*http://swaroopch.com/notes/Python_en-Preface/*）两个版本。

14.Codecademy 课程：学习 Python 编程

Codecademy 课程：学习 Python 编程（*http://www.codecademy.com/en/tracks/python*）专为 Python 新手准备。这一免费的交互式课程介绍了 Python 编程的基础知识，在学生逐步学习教程的同时不断测验学生对知识的掌握程度。它还内建一个解释器，方便学习者在做课程作业时得到即时反馈。

中阶

《高效 Python》

这本书包含 59 个编写高质量 Python 代码的有效方法。书中简明扼要地介绍了中级 Python 程序员最常用的一些高效方法。

高阶

1.《Python 专业编程》

这本书适合志在成为高级 Python 程序员的中级程序员。这个阶段的程序员会尝试理解 Python 内部原理，以及如何将自己的代码水平提升到更高层次。

2.《Python 高级编程》

这本书主要论述 Python 编程的最佳实践，适合更高层次的 Python 程序员阅读。它论述的主题包括：装饰器（以缓存、代理及上下文管理器为案例）、方法解析次序、使用 super()、元编程及常规的 PEP 8 最佳实践。

书中针对如何编写和发布一个 Python 包乃至最终发布一个应用，进行了详细的案

例研究，其中介绍了使用 zc.buildout 和多方面的最佳实践，比如，文档编写、测试驱动开发、版本管理、性能优化及性能剖析。

3. Python 魔术方法指南

Python 魔术方法指南（*http://www.rafekettler.com/magicmethods.html*）是 Rafe Kettler 所写博文的一个合集，解释了 Python 中的“魔术方法”。魔术方法的名称两头是双下画线（例如 __init__），可以让类和对象的行为像变魔术一样特别。

工程科学相关

1.《物理学高效计算》

它由 Anthony Scopatz 和 Kathryn D. Huff 编写，专为那些正要开始在某个科学或工程领域使用 Python 的研究生一年级的学生准备。它包含了一些使用 SED 和 AWK 搜索文件的代码片段，还针对学术研究的各个环节——从数据收集和分析到文章发表——提供一些小窍门。

2.《Python 科学编程入门》

Hans Petter Langtangen 写的这本书主要涵盖 Python 在科学领域的用法。书中的示例均选自数学和自然科学领域。

3.《工程领域数值方法的 Python 实现》

Jaan Kiusalaas 写的这本书重点介绍了现代数值方法及如何使用 Python 实现这些方法。

4.《算法评注的 Python 实现：物理、生物及金融领域应用》

Massimo Di Pierro 写的这本书是一本教材，旨在以简单易懂的方式来实现算法、演示算法。

其他主题材料

1.《问题求解：算法与数据结构应用》

这本书涵盖了一系列的数据结构和算法。所有概念的讲解都配有 Python 代码，交互式的代码示例可以直接在浏览器中运行。

2.《集体智慧编程》

这本书介绍了大量基础的机器学习和数据挖掘方法，算法讲解不是非常的数学形式化，而是着重解释算法背后直观的思想，以及演示如何使用 Python 实现算法。

3. 漂亮地道 Python 代码变形记

这个演讲视频（*http://bit.ly/hettinger-presentation*）是 Raymond Hettinger 录制的，向观众演示了如何更好地利用 Python 的最佳特性，如何通过一系列的代码变换方式优化已有代码。

4. 全栈 Python

这个网站（*https://www.fullstackpython.com/*）为 Python Web 开发提供一套全面的学习资源，从搭建 Web 服务器到设计前端、选择数据库、性能优化、扩展等均有涉及。文如其名，它涵盖了从头构建运行一个完整 Web 应用需要做的一切工作。

参考手册

1.《Python 技术手册》

这本书涵盖了大多数跨平台的 Python 用法，从语法到内建库，再到编写 C 扩展这类高级话题。

2.Python 语言参考手册

这（*http://docs.python.org/reference/index.html*）是 Python 官方的在线参考手册，涵盖了 Python 语言的语法和核心语义。

3.《Python 精要参考手册》

David Beazley 编写的这本书是权威的 Python 参考指南，简明扼要地讲解了语言核心和标准库的多数重要部分，涵盖 Python 3 和 Python 2.6。

4.《Python 袖珍参考手册》

Mark Lutz 编写的这本书是一本易用的语言核心参考手册，介绍了常用的语言模块和工具包，涵盖 Python 3 和 Python 2.6。

5.《Python 经典实例》

David Beazley 和 Brian K. Jones 编写的这本书包含大量实用的技巧，涵盖 Python 语言核心的同时也涵盖很多应用领域的常见任务。

6.《编写地道的 Python 代码》

Jeff Knupp 编写的这本书以尽可能易记易懂的形式介绍了最常用最重要的 Python 惯用法。每个惯用法都是编写某个常用代码块的推荐方式，书中解释了这些惯用法为什么如此重要，并且还为每个惯用法都准备了两个代码示例：不良的写法和地道的写法。这本书针对 Python 2.7.3（*https://amzn.com/B00B5VXMRG*）和 Python 3.3（*https://amzn.com/B00B5VXMRG*）分为两个不同版本。

文档

1. 官方文档

Python 语言和标准库官方文档，分为 Python 2.x 版本（*https://docs.python.org/2/*）和 Python 3.x 版本（*https://docs.python.org/3/*）。

2. 官方打包文档

它（*https://packaging.python.org/*）是 Python 官方的打包指南，为 Python 代码打包提供了最新的操作说明。我们也可以通过 testPyPI 来确认打包是否成功。

3．Read the Docs

Read the Docs（*https://readthedocs.org/*）是一个为开源软件提供文档托管服务的社区项目，深受大家喜爱。许多 Python 模块的文档都托管于此，流行的和不常见的模块都有。

4．pydoc

Pydoc 是一个随 Python 安装的工具。用户可以在 shell 中使用它来快速获取和搜索文档。例如，如果需要快速地复习 time 模块，那么在命令行中输入如下命令即可获取其文档。

```
$ pydoc time
```

这个命令实质上相当于打开 Python 交互式命令行（REPL）并运行。

```
>>> import time
>>> help(time)
```

译注：import time 一句为译者所加。必须先写 import 模块，才能对模块使用 help 函数。

新闻

最受喜爱的 Python 新闻来源，如表 A-1 所示。

表A-1　最受喜爱的Python新闻来源

名称	描述
/r/python (*http://reddit.com/r/python*)	Reddit 网站上的 Python 社区，用户可以贡献 Python 相关新闻，也可以进行投票
Import Python Weekly (*http://www.importpython. com/newsletter/*)	一份新闻周刊，包含 Python 文章、项目、视频及推文
Planet Python (*http://planet.python.org/*)	Python 新闻聚集地，参与贡献的开发者越来越多
Podcast.__init__ (*http://podcastinit.com/*)	关注 Python 和优秀开发者的一个播客，一周更新一次
Pycoder's Weekly (*http://www.pycoders.com/*)	一份免费的 Python 新闻周刊，Python 开发者群体自产自销（重点关注有趣的项目，包含文章、新闻和职位公告）
Python News (*http://www.python.org/news/*)	Python 官网（*http://www.python.org*）上的新闻板块，Python 社区新闻精选
Python Weekly (*http://www.pythonweekly.com/*)	一份免费的新闻周刊，其特色是精选的新闻、文章、软件新版本发布信息及 Python 相关职位
Talk Python to Me (*http://talkpython.fm/*)	Python 及相关技术的一个播客

作译者简介

Kenneth Reitz

Python 界的大神、Python 软件基金会会员，因众多开源项目而闻名（其中最著名的是“Requests: HTTP for Humans”），高颜值的摄影爱好者、电子音乐制作师、健身减肥成功的励志男……曾任 Heroku 公司 Python 架构负责人，现任职于 DigitalOcean。

Tanya Schlusser

数据决策方向的独立顾问，为学生和企业团队提供的数据科学培训时长已超过一千小时，并照顾患有阿兹海默症的妈妈。

夏永锋

百度资深研发工程师，曾就职于腾讯，对 Python、GO、Java 开发均有较丰富的实战经验，长期从事后台开发、大数据处理方面的工作 , 爱好编程和技术翻译。

廖邦杰

360 安全开发工程师，核心安全高级威胁应对团队成员，长期从事后台数据处理及 0day 漏洞发现工作，在项目开发和安全分析中均涉及大量的 Python 编程工作。

封面介绍

《Python 编程之美 ：最佳实践指南》的封面动物是印度褐獴（Herpestes fuscus），一种小型哺乳动物，生长于斯里兰卡和印度西南部的森林，与东南亚的短尾獴（Herpestes brachyurus）非常相似，可能是短尾獴的一个亚种。

与其他獴类相比，印度褐獴体型更大一点，有尖尾和毛茸茸的后腿。其皮毛从躯体到腿部逐步由深褐色变成黑色。因为它昼伏夜出（晨昏之时活动），所以人类极少见到。

一直以来，关于印度褐獴的统计数据都很少，大家曾认为其是濒危物种。不过近期完善的科学监控发现该物种其实数量相当庞大，特别是在印度西南部，因此又成了无危物种。最近在斐济的维提岛也发现了印度褐獴的一个种群。

O’Reilly 图书封面的许多动物都已濒临灭绝。对我们这个世界来说，它们都很重要。若想了解如何为它们贡献一份力量，请访问网站 animals.oreilly.com。

封面图片来自林德科尔的《皇家自然史》一书。