Python 高手之路

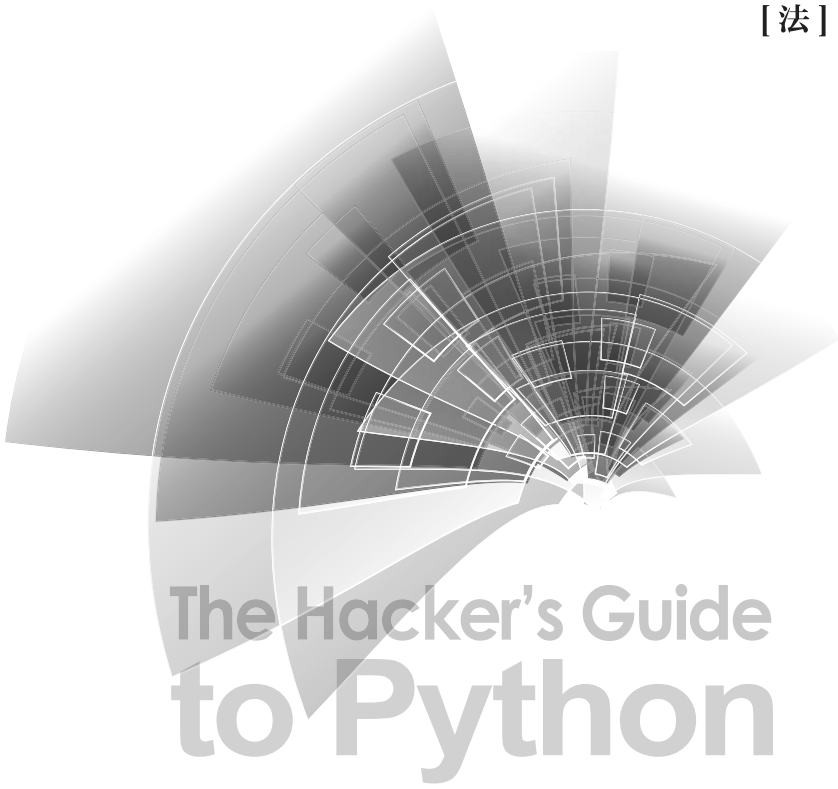
[法] Julien Danjou 著
王飞龙 译

The Hacker's Guide to Python

人民邮电出版社
POSTS & TELECOM PRESS

Python
高手之路

[法] Julien Danjou 著
王飞龙 译



人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Python高手之路 /（法）丹乔（Danjou,J.）著 ；王飞龙译. -- 北京 ：人民邮电出版社，2015.5
书名原文：The hacker’s guide to Python
ISBN 978-7-115-38713-4

Ⅰ. ①P… Ⅱ. ①丹… ②王… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2015)第065975号

版权声明

内 容 提 要

这不是一本常规意义上 Python 的入门书。这本书中没有 Python 关键字和 `for` 循环的使用，也没有细致入微的标准库介绍，而是完全从实战的角度出发，对构建一个完整的 Python 应用所需掌握的知识进行了系统而完整的介绍。更为难得的是，本书的作者是开源项目 OpenStack 的 PTL（项目技术负责人）之一，因此本书结合了 Python 在 OpenStack 中的应用进行讲解，非常具有实战指导意义。

本书从如何开始一个新的项目讲起，首先是整个项目的结构设计，对模块和库的管理，如何编写文档，进而讲到如何分发，以及如何通过虚拟环境对项目进行测试。此外，本书还涉及了很多高级主题，如性能优化、插件化结构的设计与架构、Python 3 的支持策略等。本书适合各个层次的 Python 程序员阅读和参考。

◆ 著　[法] Julien Danjou
译　王飞龙
责任编辑　杨海玲
责任印制　张佳莹　焦志炜
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷
◆ 开本：800×1000　1/16
印张：13
字数：277 千字　2015 年 5 月第 1 版
印数：1- 3 000 册　2015 年 5 月北京第 1 次印刷
著作权合同登记号　图字：01-2014-5625 号

定价：49.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

中文版序

亲爱的中国读者你们好！

祝贺你，你正在读 *The Hacker's Guide to Python* 一书的中文版。我非常高兴看到这本书最终翻译完成，这样你就可以用自己的语言去阅读。这是这本书第三种语言的版本（之前已经有了英语和韩语两个版本）。能够有更多的读者看到这本书真是太棒了！

你将阅读的这本书的大部分内容来自我在 OpenStack 这个大规模项目中开发 Python 代码时的经验。你们是非常幸运的，因为这本书是由王飞龙翻译的，他是一名软件工程师，他和我同在 OpenStack 社区做开发工作。因此，高质量的翻译和对本书内容的精确表述是可期待的，因为飞龙对本书涉及的内容有着很好的理解。

真心希望你们能喜欢这本书。祝你阅读愉快！

Julien Danjou

前言

如果你读到这里，你肯定已经使用 Python 有一阵子了。你可能是通过一些文档学习的，钻研了一些已有的项目或者从头开发，但不管是哪种情况，你都已经在以自己的方式学习它了。直到两年前我加入 OpenStack 项目组之前，这其实也正是我个人熟悉 Python 的方法。

在此之前，我只是开发过一些“车库项目[①]”级别的 Python 库或应用程序，而一旦你参与开发涉及数百名开发人员并有着上万个用户的软件或库时，情况就会有所不同。OpenStack 平台有超过 150 万行 Python 代码，所有代码都需要精确、高效，并根据用户对云计算应用程序的需求进行任意扩展。在有了这一规模的项目之后，类似测试和文档这类问题就一定需要自动化，否则根本无法完成。

我刚开始加入 OpenStack 的时候，我认为自己已经掌握了不少 Python 知识，但这两年，在我起步时无法想象其规模的这样一个项目上，我学到了更多。而且我还有幸结识了很多业界最棒的 Python 黑客，并从他们身上获益良多——大到通用架构和设计准则，小到各种有用的经验和技巧。通过本书，我想分享一些我所学到的最重要的东西，以便你能构建更好的 Python 应用，并且是更加高效地构建。

① 作者这里的意思是规模很小，比较业余的项目。——译者注

目录

第 1 章 项目开始

1.1 Python 版本

你很可能会问的第一个问题就是：“我的软件应该支持 Python 的哪些版本？”这是一个好问题，因为每个 Python 新版本都会在引入新功能的同时弃用一些老的功能。而且，Python 2.x 和 Python 3.x 之间有着**巨大**的不同，这两个分支之间的剧烈变化导致很难使代码同时兼容它们。本书后面章节会进一步讨论，而且当刚刚开始一个新项目时很难说哪个版本更合适。

- 2.5 及更老的版本目前基本已经废弃了，所以不需要再去支持它们。如果实在想支持这些更老的版本，要知道再让程序支持 Python 3.x 会更加困难。如果你确实有可能会遇到一些安装了 Python 2.5 的老系统，那真没什么好办法。
- 2.6 版本在某些比较老的操作系统上仍然在用，如 Red Hat 企业版 Linux（Red Hat Enterprise Linux）。同时支持 Python 2.6 版本和更新的版本并不太难，但是，如果你认为自己的程序不太可能会在 2.6 版本上运行，那就没必要强迫自己支持它。
- 2.7 版本目前是也将仍然是 Python 2.x 的最后一个版本。将其作为主要版本或主要版本之一来支持是正确的选择，因为目前仍然有很多软件、库和开发人员在使用它。Python 2.7 将被继续支持到 2020 年左右，所以它很可能不会很快消失。
- 3.0、3.1 和 3.2 版本在发布后都被快速地更替，并没有被广泛采用。如果你的代码已经支持了 2.7 版本，那么再支持这几个版本的意义并不大。
- 3.3 和 3.4 版本都是 Python 3 最近发行的两个版本，也是应该重点支持的版本。Python 3.3 和 3.4 代表着这门语言的未来，所以除非正专注于兼容老的版本，否则都应该先确保代码能够运行在这两个最新的版本上。

总之，在确实有需要的情况下支持 2.6 版本（或者想自我挑战），必须支持 2.7 版本，如果需要保证软件在可预见的未来也能运行，就需要也支持 3.3 及更高的版本。忽略那些更老的 Python 版本基本没什么问题，尽管同时支持所有这些版本是有可能的：CherryPy 项目（http://cherrypy.org）支持 Python 2.3 及所有后续版本（http://docs.cherrypy.org/stable/intro/install.html）。

编写同时支持 Python 2.7 和 3.3 版本的程序的技术将在第 13 章介绍。某些技术在后续的示例代码中也会涉及，所有本书中的示例代码都同时支持这两个主要版本。

1.2　项目结构

项目结构应该保持简单，审慎地使用包和层次结构，过深的层次结构在目录导航时将如同梦魇，但过平的层次结构则会让项目变得臃肿。

一个常犯的错误是将单元测试放在包目录的外面。这些测试实际上应该被包含在软件的子一级包中，以便：

- 避免被 **setuptools**（或者其他打包的库）作为 tests 顶层模块自动安装；
- 能够被安装，且其他包能够利用它们构建自己的单元测试。

图 1-1 展示了一个项目的标准的文件层次结构。

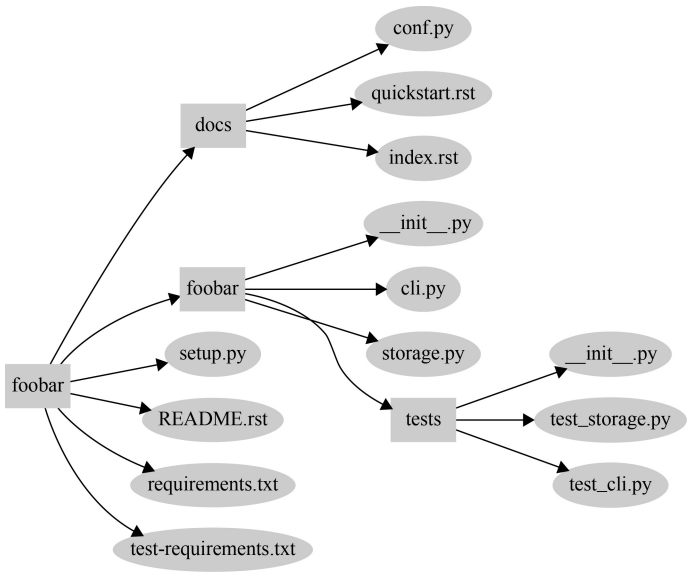


图 1-1　标准的包目录结构

setup.py 是 Python 安装脚本的标准名称。在安装时，它会通过 Python 分发工具（distuils）进行包的安装。也可以通过 README.rst（或者 README.txt，或者其他合适的名字）为用户提供重要信息。requirements.txt 应该包含 Python 包所需要的依赖包，也就是说，所有这些包都会预先通过 pip 这样的工具进行安装以保证你的包能正常工作。还可以包含 test-requirements.txt，它应该列出运行测试集所需要的依赖包。最后，docs 文件夹应该包括 reStructuredText 格式的文档，以便能够被 Sphinx 处理（参见 3.1 节）。

包中还经常需要包含一些额外的数据，如图片、shell 脚本等。不过，关于这类文件如何存放并没有一个统一的标准。因此放到任何觉得合适的地方都可以。

下面这些顶层目录也比较常见。

- etc 用来存放配置文件的样例。
- tools 用来存放与工具有关的 shell 脚本。
- bin 用来存放将被 setup.py 安装的二进制脚本。
- data 用来存放其他类型的文件，如媒体文件。

一个常见的设计问题是根据将要存储的代码的类型来创建文件或模块。使用 functions.py 或者 exceptions.py 这样的文件是很**糟糕的**方式。这种方式对代码的组织毫无帮助，只能让读代码的人在多个文件之间毫无理由地来回切换。

此外，应该避免创建那种只有一个__init__.py 文件的目录，例如，如果 hooks.py 够用的话就不要创建 hooks/__init__.py。如果创建目录，那么其中就应该包含属于这一分类/模块的多个 Python 文件。

1.3 版本编号

可能你已经有所了解，Python 生态系统中正在对包的元数据进行标准化。其中的一项元数据就是*版本号*。

PEP 440（http://www.python.org/dev/peps/pep-0440/）针对所有的 Python 包引入了一种版本格式，并且在理论上所有的应用程序都应该使用这种格式。这样，其他的应用程序或包就能简单而可靠地识别它们需要哪一个版本的包。

PEP440 中定义版本号应该遵从以下正则表达式的格式：

```
N[.N]+[{a|b|c|rc}N][.postN][.devN]
```

它允许类似 1.2 或 1.2.3 这样的格式，但需注意以下几点。

- 1.2 等于 1.2.0，1.3.4 等于 1.3.4.0，以此类推。
- 与 *N[.N]+*相匹配的版本被认为是**最终版本**。
- 基于日期的版本（如 2013.06.22）被认为是无效的。针对 PEP440 格式版本号设计的一些自动化工具，在检测到版本号大于或等于 1980 时就会抛出错误。

最终即将发布的组件也可以使用下面这种格式。

- *N[.N]+aN*（如 1.2a1）表示一个 **alpha 版本**，即此版本不稳定或缺少某些功能。
- *N[.N]+bN*（如 2.3.1b2）表示一个 **beta 版本**，即此版本功能已经完整，但可能仍有 bug。
- *N[.N]+cN* 或 *N[.N]+rcN*（如 0.4rc1）表示**候选版本**（常缩写为 RC），通常指除非有重大的 bug，否则很可能成为产品的最终发行版本。尽管 *rc* 和 *c* 两个后缀含义相同，但如果二者同时使用，*rc* 版本通常表示比 *c* 更新一点。

通常用到的还有以下这些后缀。

- *.postN*（如 1.4.post2）表示一个**后续版本**。通常用来解决发行过程中的细小问题（如发行文档有错）。如果发行的是 bug 修复版本，则不应该使用*.postN* 而应该增加小的版本号。
- *.devN*（如 2.3.4.dev3）表示一个**开发版本**。因为难以解析，所以这个后缀并不建议使用。它表示这是一个质量基本合格的发布前的版本，例如，2.3.4.dev3 表示 2.3.4 版本的第三个开发版本，它早于任何的 alpha 版本、beta 版本、候选版本和最终版本。

这一结构可以满足大部分常见的使用场景。

注意

你可能已经听说过语义版本（http://semver.org/），它对于版本号提出了自己的规则。这一规范和 PEP 440 部分重合，但二者并不完全兼容。例如，语义版本对于预发布版本使用的格式 *1.0.0.-alpha+001* 就与 PEP 440 不兼容。

如果需要处理更高级的版本号，可以考虑一下 PEP 426（http://www.python.org/dev/peps/pep-0426）中定义的**源码标签**，这一字段可以用来处理任何版本字符串，并生成同 PEP 要求一致的版本号。

许多分布式版本控制系统（Distributed Version Control System，DVCS）平台，如 Git 和 Mercurial，都可以使用唯一标识的散列字符串[①]作为版本号。但遗憾的是，它不能与 PEP 440 中定义的模式兼容：问题就在于，唯一标识的散列字符串不能排序。不过，是有可能通过源码标签这个字段维护一个版本号，并利用它构造一个同 PEP 440 兼容的版本号的。

提示

pbr（即 Python Build Reasonableness，https://pypi.python.org/pypi/pbr）将在 4.2 节中讨论，它可以基于项目的 Git 版本自动生成版本号。

1.4 编码风格与自动检查

没错，编码风格是一个不太讨巧的话题，不过这里仍然要聊一下。

Python 具有其他语言少有的绝佳质量[②]：使用缩进来定义代码块。乍一看，似乎它解决了一个由来已久的“往哪里放大括号？”的问题，然而，它又带来了“如何缩进？”这个新问题。

而 Python 社区则利用他们的无穷智慧，提出了编写 Python 代码的 PEP 8[③]（http://www.python.org/dev/peps/pep-0008/）标准。这些规范可以归纳成下面的内容。

- 每个缩进层级使用 4 个空格。
- 每行最多 79 个字符。
- 顶层的函数或类的定义之间空两行。
- 采用 ASCII 或 UTF-8 编码文件。
- 在文件顶端，注释和文档说明之下，每行每条 `import` 语句只导入一个模块，同时要按标准库、第三方库和本地库的导入顺序进行分组。

① 对于 Git，指的是 `git-describe(1)`。
② 你可能有不同意见。
③ *PEP 8 Style Guide for Python Code*, 5th July 2001, Guido van Rossum, Barry Warsaw, Nick Coghlan

- 在小括号、中括号、大括号之间或者逗号之前没有额外的空格。
- 类的命名采用骆驼命名法，如 `CamelCase`；异常的定义使用 `Error` 前缀（如适用的话）；函数的命名使用小写字符，如 `separated_by_underscores`；用下划线开头定义私有的属性或方法，如 `_private`。

这些规范其实很容易遵守，而且实际上很合理。大部分程序员在按照这些规范写代码时并没有什么不便。

然而，**犯错在所难免**，保持代码符合 PEP 8 规范的要求仍是一件麻烦事。工具 **pep8**（https://pypi.python.org/pypi/pep8）就是用来解决这个问题的，它能自动检查 Python 文件是否符合 PEP 8 要求，如示例 1.1 所示。

示例 1.1　运行 pep8

```
$ pep8 hello.py
hello.py:4:1: E302 expected 2 blank lines, found 1
$ echo $?
1
```

pep8 会显示在哪行哪里违反了 PEP 8，并为每个问题给出其错误码。如果违反了那些必须遵守的规范，则会报出**错误**（以 E 开头的错误码），如果是细小的问题则会报**警告**（以 W 开头的错误码）。跟在字母后面的三位数字则指出具体的错误或警告，可以从错误码的百位数看出问题的大概类别。例如，以 E2 开头的错误通常与空格有关，以 E3 开头的错误则与空行有关，而以 W6 开头的警告则表明使用了已废弃的功能。

社区仍然在争论对并非标准库一部分的代码进行 PEP 8 验证是否是一种好的实践。这里建议还是考虑一下，最好能定期用 PEP 8 验证工具对代码进行检测。一种简单的方式就是将其集成到测试集中。尽管这似乎有点儿极端，但这能保证代码一直遵守 PEP 8 规范。6.7 节中将介绍如何将 pep8 与 tox 集成，从而让这些检查自动化。

OpenStack 项目从一开始就通过自动检查强制遵守 PEP 8 规范。尽管有时候这让新手比较抓狂，但这让整个代码库的每一部分都保持一致，要知道现在它有 167 万行代码。对于任何规模的项目这都是非常重要的，因为即使对于空白的顺序，不同的程序员也会有不同的意见。

也可以使用`--ignore` 选项忽略某些特定的错误或警告，如示例 1.2 所示。

示例 1.2 运行 pep8 时指定--ignore 选项

```
$ pep8 --ignore=E3 hello.py
$ echo $?
0
```

这可以有效地忽略那些不想遵循的 PEP 8 标准。如果使用 pep8 对已有的代码库进行检查，这也可以暂时忽略某些问题，从而每次只专注解决一类问题。

注意

如果正在写一些针对 Python 的 C 语言代码（如模块），则 PEP 7（http://www.python.org/dev/peps/pep-0007/）标准描述了应该遵循的相应的编码风格。

还有一些其他的工具能够检查真正的编码错误而非风格问题。下面是一些比较知名的工具。

- pyflakes（https://launchpad.net/pyflakes），它支持插件。
- pylint（https://pypi.python.org/pypi/pylint），它支持 PEP 8，默认可以执行更多检查，并且支持插件。

这些工具都是利用静态分析技术，也就是说，解析代码并分析代码而无需运行。

如果选择使用 pyflakes，要注意它按自己的规则检查而非按 PEP 8，所以仍然需要运行 pep8。为了简化操作，一个名为 flake8（https://pypi.python.org/pypi/flake8）的项目将 pyflakes 和 pep8 合并成了一个命令，而且加入了一些新的功能，如忽略带有`#noqa`的行以及通过入口点（entry point）进行扩展。

为了追求优美而统一的代码，OpenStack 选择使用 flake8 进行代码检查。不过随着时间的推移，社区的开发者们已经开始利用 flake8 的可扩展性对提交的代码进行更多潜在问题的检查。最终 flake8 的这个扩展被命名为 hacking（https://pypi.python.org/pypi/hacking）。它可以检查`except`语句的错误使用、Python 2 与 Python 3 的兼容性问题、导入风格、危险的字符串格式化及可能的本地化问题。

如果你正开始一个新项目，这里强烈建议使用上述工具之一对代码的质量和风格进行自动检查。如果已经有了代码库，那么一种比较好的方式是先关闭大部分警告，然后每次只解决一类问题。

尽管没有一种工具能够完美地满足每个项目或者每个人的喜好，但 flake8 和 hacking 的

结合使用是持续改进代码质量的良好方式。要是没想好其他的，那么这是一个向此目标前进的好的开始。

提示

许多文本编辑器，包括流行的 GNU Emacs（http://www.gnu.org/software/emacs/）和 vim（http://www.vim.org/），都有能够直接对代码运行 pep8 和 flake8 这类工具的插件（如 Flymake），能够交互地突出显示代码中任何不兼容 PEP 8 规范的部分。这种方式能够非常方便地在代码编写过程中修正大部分风格错误。

第 2 章 模块和库

2.1 导入系统

要使用模块和库，需要先进行导入。

Python 之禅

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

导入系统是相当复杂的，不过你可能已经了解了一些基本知识。这里会介绍一些关于这一子系统的内部机理。

sys 模块包含许多关于 Python 导入系统的信息。首先，当前可导入的模块列表都是通过 sys.moudle 变量才可以使用的。它是一个字典，其中键（key）是模块名字，对应的值（value）是模块对象。

```
>>> sys.modules['os']
<module 'os' from '/usr/lib/python2.7/os.pyc'>
```

许多模块是内置的，这些内置的模块在 sys.builtin_module_names 中列出。内置模块可以根据传入 Python 构建系统的编译选项的不同而变化。

导入模块时，Python 会依赖一个路径列表。这个列表存储在 sys.path 变量中，并且告诉 Python 去哪里搜索要加载的模块。可以在代码中修改这个列表，根据需要添加或删除路径，也可以通过编写 Python 代码直接修改环境变量 PYTHONPATH。下面的方法几乎是相等的[①]。

```
>>> import sys
>>> sys.path.append('/foo/bar')

$ PYTHONPATH=/foo/bar python
>>> import sys
>>> '/foo/bar' in sys.path
True
```

在 sys.path 中顺序很重要，因为需要遍历这个列表来寻找请求的模块。

也可以通过自定义的导入器（importer）对导入机制进行扩展。Hy[②]正是利用的这种技术告诉 Python 如何导入其他非标准的.py 或者.pyc 文件的。

顾名思义，导入钩子机制是由 PEP 302（http://www.python.org/dev/peps/pep-0302/）定义的[③]。它允许扩展标准的导入机制，并对其进行预处理，也可以通过追加一个工厂类到 sys.path_hooks 来添加自定义的模块查找器（finder）。

模块查找器对象必须有一个返回加载器对象的 find_module(fullname, path=None) 方法，这个加载器对象必须包含一个负责从源文件中加载模块的 load_module(fullname)

① 说几乎是因为路径并不会被放在列表的同一级上，尽管根据你的使用情况它可能并不重要。
② Hy 是 Python 上的 Lisp 实现，会在 9.1 节介绍。
③ 在 Python 2.3 版本实现的新的带入钩子机制。

方法。

为了进一步说明，下面给出了 Hy 利用自定义的导入器导入.hy 而不是.py 结尾的源文件的方法，见示例 2.1。

示例 2.1 Hy 模块导入器

```
class MetaImporter(object):
    def find_on_path(self, fullname):
        fls = ["%s/__init__.hy", "%s.hy"]
        dirpath = "/".join(fullname.split("."))

        for pth in sys.path:
            pth = os.path.abspath(pth)
            for fp in fls:
                composed_path = fp % ("%s/%s" % (pth, dirpath))
                if os.path.exists(composed_path):
                    return composed_path

    def find_module(self, fullname, path=None):
        path = self.find_on_path(fullname)
        if path:
            return MetaLoader(path)

sys.meta_path.append(MetaImporter())
```

一旦路径被确定是有效的且指向了一个模块，就会返回一个 MetaLoader 对象。

Hy 模块加载器

```
class MetaLoader(object):
    def __init__(self, path):
        self.path = path

    def is_package(self, fullname):
        dirpath = "/".join(fullname.split("."))
        for pth in sys.path:
            pth = os.path.abspath(pth)
            composed_path = "%s/%s/__init__.hy" % (pth, dirpath)
            if os.path.exists(composed_path):
                return True
```

```
        return False

    def load_module(self, fullname):
        if fullname in sys.modules:
            return sys.modules[fullname]

        if not self.path:
            return

        sys.modules[fullname] = None
        mod = import_file_to_module(fullname, self.path) ❶

        ispkg = self.is_package(fullname)

        mod.__file__ = self.path
        mod.__loader__ = self
        mod.__name__ = fullname

        if ispkg:
            mod.__path__ = []
            mod.__package__ = fullname
        else:
            mod.__package__ = fullname.rpartition('.')[0]

        sys.modules[fullname] = mod
        return mod
```

❶ `import_file_to_module` 读取一个 Hy 源文件，将其编译成 Python 代码，并返回一个 Python 模块对象。

`uprefix` 模块（https://pypi.python.org/pypi/uprefix）是这个功能起作用的另一个好的例子。Python 3.0 到 3.2 并没有像 Python 2 中用来表示 Unicode 字符串的 `u` 前缀[①]，这个模块通过在编译前删除字符串的前缀 `u` 来确保在 2.x 和 3.x 之间的兼容性。

2.2　标准库

Python 本身内置的巨大标准库提供了丰富的工具和功能，可以满足你能想到的任何需求。很多 Python 的初学者习惯于自己写代码实现一些基本的功能，然后会惊奇地发现很多

① 它在 Python 3.3 中又被加了回来。

功能已经内置了，直接就可以使用。

任何时候想要自己写函数处理一些简单的工作时，请**停下来**先看看标准库。我的建议是至少大概浏览一遍标准库，这样下次再需要一个函数时就能知道是否可以利用标准库中已有的函数了。

后续章节会讨论其中的一些模块，如 **functools** 和 **itertools**，下面是一些必须了解的标准库模块。

- **atexit** 允许注册在程序退出时调用的函数。
- **argparse** 提供解析命令行参数的函数。
- **bisect** 为可排序列表提供二分查找算法（参见 10.3 节）。
- **calendar** 提供一组与日期相关的函数。
- **codecs** 提供编解码数据的函数。
- **collections** 提供一组有用的数据结构。
- **copy** 提供复制数据的函数。
- **csv** 提供用于读写 CSV 文件的函数。
- **datetime** 提供用于处理日期和时间的类。
- **fnmatch** 提供用于匹配 Unix 风格文件名模式的函数。
- **glob** 提供用于匹配 Unix 风格路径模式的函数。
- **io** 提供用于处理 I/O 流的函数。在 Python3 中，它还包含 **StringIO**（在 Python 2 中有同名的模块），可以像处理文件一样处理字符串。
- **json** 提供用来读写 JSON 格式数据的函数。
- **logging** 提供对 Python 内置的日志功能的访问。
- **multiprocessing** 可以在应用程序中运行多个子进程，而且提供 API 让这些子进程看上去像线程一样。
- **operator** 提供实现基本的 Python 运算符功能的函数，可以使用这些函数而不是自己写 lambda 表达式（参见 8.3 节）。
- **os** 提供对基本的操作系统函数的访问。

- **random** 提供生成伪随机数的函数。
- **re** 提供正则表达式功能。
- **select** 提供对函数 *select()*和 *poll()*的访问，用于创建事件循环。
- **shutil** 提供对高级文件处理函数的访问。
- **signal** 提供用于处理 POSIX 信号的函数。
- **tempfile** 提供用于创建临时文件和目录的函数。
- **threading** 提供对处理高级线程功能的访问。
- **urllib**（以及 Python 2.x 中的 **urllib2** 和 **urlparse**）提供处理和解析 URL 的函数。
- **uuid** 可以生成全局唯一标识符（Universally Unique Identifiers，UUID）。

这个模块清单可以作为一个快速参考，帮助你了解各个库模块的作用。如果能记住一部分就更好了。花在查找标准库上的时间越少，意味着写实际代码的时间就越多。

提示

整个标准库都是用 Python 写的，所以可以直接查看它模块和函数的源代码。有疑问时只需打开代码自己一探究竟。尽管文档中已经包含了你想知道的一切，但总还是有机会让你学一些有用的东西。

2.3　外部库

你是否有过这样的经历，收到一件不错的生日礼物或圣诞礼物，但是打开后却发现送你的人忘了买电池？Python 的“内置电池”哲学让你作为程序员不会遇到这类问题，只要安装了 Python，就拥有了完成任何功能所需的一切条件。

然而，Python 标准库的开发者并不能预测你要实现的“任何”功能到底是什么。即使可以，大多数人也不想去处理一个几个 GB 的文件下载，即使可能只是需要写一个重命名文件的快速脚本。关键在于，即使拥有所有的扩展功能，仍然有许多功能是 Python 标准库没有涵盖的。不过，这并不是说有些事情是根本无法用 Python 实现的，这只是表明有些事情可能需要使用外部库。

Python 标准库是安全且范围明确的：模块文档化程度很高，并且有足够多的人在经常使用它，从而可以保证在你想使用它时肯定不会遇到麻烦。而且，就算万一出了问题，也能确

保在短时间内有人解决。但是，外部库就像是地图上标着“熊出没，请注意”的部分：可能缺少文档，功能有 bug，更新较少或根本不更新。任何正式的项目都可能用到一些只有外部库提供的功能，但是需要谨记使用这些外部库可能带来的风险。

下面是来自一线的案例。OpenStack 使用了 SQLAlchemy（http://www.sqlalchemy.org/），一个 Python 数据库开发工具包。如果了解 SQL 的话会知道，数据库的结构是会发生变化的，所以 OpenStack 还使用了 sqlalchemy-migrate（https://code.google.com/p/sqlalchemy-migrate/）来处理数据库模式的升级。一切运行良好，直到有一天它们不行了，开始出现大量 bug，并且没有好转的迹象。而且，OpenStack 在当时是想要支持 Python 3 的，然而没有任何迹象表明 sqlalchemy-migrate 要支持 Python 3。因此，显然 sqlalchemy-migrate 已经死了，我们需要切换到其他替代方案。截止到作者写作时，OpenStack 正准备升级到 Alembic（https://pypi.python.org/pypi/alembic），虽然也有一些工作要做，但好在不是那么痛苦。

所有这些引出一个重要的问题：“如何保证我不会掉进同样的陷阱里？”很遗憾，没办法保证。程序员也是人，没什么办法可以确保目前维护良好的库在几个月后仍然维护良好。但是，在 OpenStack 中我们使用下列检查表来根据需要给出建议（我建议你也这么做）。

- Python 3 兼容。尽管现在你可能并不准备支持 Python 3，但很可能早晚会涉及，所以确认选择的库是 Python 3 兼容的并且承诺保持兼容是明智的。
- 开发活跃。GitHub（http://github.com）和 Ohloh（http://www.ohloh.net/）通常提供足够的信息来判断一个库是否有维护者仍然在工作。
- 维护活跃。尽管一个库可能是“结束”状态（即功能完备，不会再加入新功能），但应该有维护者仍然在工作，以确保没有 bug。可以通过查看项目的跟踪系统来看维护者对 bug 的反应是否迅速。
- 与各个操作系统发行版打包在一起。如果一个库被打包在主流的 Linux 发行版内，说明有其他项目依赖它，所以，如果真有什么问题，至少你不是唯一一个抱怨的。如果打算公开发布你的软件，那么这项检查也是很有用的。因为如果软件的依赖已经在终端用户的机器上安装了，显然分发你的软件会更容易。
- API 兼容保证。没有比你的软件因为一个它依赖的库发生了变化而使整个 API 崩溃更糟的了。你一定很想知道选择的库在过去是否发生过类似的事件。

尽管可能工作量巨大，但这一检查表对于依赖同样适用。如果知道应用程序会大量依赖

一个特定的库，那么至少应该对这个库的每一个依赖使用这个检查表。

不管最终使用哪个库，都要像其他工具一样对待，因为即使是有用的工具也可能会造成严重的损害。尽管不常发生，但问问你自己：如果你有一把锤子，你会拿着它满屋跑因而可能意外地损坏屋子里的东西，还是会把它放在工具架上或者车库里，远离那些贵重而易碎的东西，仅在需要的时候才拿出来？

对于外部库道理是一样的，不管它们多么有用，都需要注意避免让这些库和实际的源代码耦合过于紧密。否则，如果出了问题，你就需要切换库，这很可能需要重写大量的代码。更好的办法是写自己的 API，用一个包装器对外部库进行封装，将其与自己的源代码隔离。自己的程序无需知道用了什么外部库，只要知道 API 提供了哪些功能即可。想要换一个不同的库？只需要修改包装器就可以了。只要它仍然提供同样的功能，那么完全不需要修改任何核心代码。也许会有例外，但应该不会太多。大部分库都被设计成只专注解决一定范围的问题，因此很容易隔离。

4.7.3 节将会涉及如何使用入口点构建驱动系统（driver system），这个系统让你可以将项目的某些部分设计成可以根据需要切换的模块。

2.4　框架

有许多不同的 Python 框架可用于开发不同的 Python 应用。如果是 Web 应用，可以使用 Django（https://www.djangoproject.com/）、Pylons（http://www.pylonsproject.org/）、TurboGears（http://turbogears.org/）、Tornado（http://www.tornadoweb.org/）、Zope（http://www.zope.org/）或者 Plone（http://plone.org/）。如果你正在找事件驱动的框架，可以使用 Twisted（http://twistedmatrix.com/）或者 Circuits（https://bitbucket.org/prologic/circuits/）等。

框架和外部库的主要不同在于，应用程序是建立在框架之上的，代码对框架进行扩展而不是反过来。而外部库更像是对代码的扩展，赋予你的代码更多额外的能力，而框架会为你的代码搭好架子，只需要通过某种方式完善这个架子就行了，尽管这可能是把双刃剑。使用框架有很多好处，如快速构建原型并开发，但也有一些明显的缺点，如锁定（lock-in）问题。因此，在决定使用某个框架前需要把这些都考虑在内。

这里推荐的为 Python 应用选择框架的方法很大程度上类似于前面介绍过的外部库的选

择方法，适用于框架是通过一组 Python 库来进行分发的情况。有时它们还包含用于创建、运行以及部署应用的工具，但这并不影响你采用的标准。前面已经提到过，在已经写了大量代码之后更换外部库是十分痛苦的，但更换框架比这还要难受一千倍，因为通常需要完全重写你的应用程序。举例说明，前面提及的 Twisted 框架还不能完全支持 Python 3。如果你基于 Twisted 的程序在几年之后想要支持 Python 3，那么你将非常不幸，除非全部重写代码选用另一个框架或者有人最终为 Twisted 提供了 Python 3 的升级支持。

有些框架与其他框架相比更加轻量级。一个简单的比较就是，Django 提供了内置的 ORM 功能，而 Flask 则没有。一个框架提供的功能越少，将来遇到问题的越少。然而，框架缺少的每个功能同时也是另一个需要去解决的问题，要么自己写，要么再千挑万选去找另一个能提供这个功能的库。愿意处理哪种场景取决于个人的选择，但需慎重选择。当问题出现时从一个框架升级至其他框架是极其艰巨的任务，就算 Python 再强大，对于这类问题也没有什么好办法。

2.5 Doug Hellmann 访谈

我曾经有幸和 Doug Hellmann 一起工作过数月。他在 DreamHost 是一位非常资深的软件开发工程师，同时他也是 OpenStack 项目的贡献者。他发起过关于 Python 的网站 Python Module of the Week（http://pymotw.com/），也出版过一本很有名的 Pyhton 书 *The Python Standard Library By Example*（http://doughellmann.com/python-standard-library-by-example），同时他也是 Python 的核心开发人员。我曾经咨询过 Doug 关于标准库以及库的设计与应用等方面的问题。

当你从头开发一个 Python 应用时，如何迈出第一步呢？它和开发一个已有的应用程序有什么不同？

从抽象角度看步骤都差不多，但是细节上有所不同。相对于对比开发新项目和已有项目，我个人在对应用程序和库开发的处理方式上有更多的不同。

当我要修改已有代码时，特别是这些代码是其他人创建的时，起初我需要研究代码是如何工作的，我需要改进哪些代码。我可能会添加日志或是输出语句，或是用 pdb，利用测试数据运行应用程序，以便我理解它是如何工作的。我经常会做一些修改并测试它们，并在每次提交代码前添加可能的自动化测试。

创建一个新应用时，我会采取相同的逐步探索方法。我先创建一些代码，然后手动运行它们，在这个功能可以基本调通后，再编写测试用例确保我已经覆盖了所有的边界情况。创建测试用例也可以让代码重构更容易。

这正是 smiley（https://pypi.python.org/pypi/smiley）的情况。在开发正式应用程序前，我先尝试用 Python 的 trace API 写一些临时脚本。对于 smiley 我最初的设想包括一个仪表盘并从另一个运行的应用程序收集数据，另一部分用来接收通过网络发送过来的数据并将其保存。在添加几个不同的报告功能的过程中，我意识到重放已收集的数据的过程和在一开始收集数据的过程基本是一样的。于是我重构了一些类，并针对数据收集，数据库访问和报告生成器创建了基类。通过让这些类遵循同样的 API 使我可以很容易地创建数据收集应用的一个版本，它可以直接将数据写入数据库而无需通过网络发送数据。

当设计一个应用程序时，我会考虑用户界面是如何工作的，但对于库，我会专注于开发人员如何使用其 API。通过先写测试代码而不是库代码，可以让思考如何通过这个新库开发应用程序变得更容易一点儿。我通常会以测试的方式创建一系列示例程序，然后依照其工作方式去构建这个库。

我还发现，在写任何库的代码之前先写文档让我可以全面考虑功能和流程的使用，而不需要提交任何实现的细节。它还让我可以记录对于设计我所做出的选择，以便读者不仅可以理解如何使用这个库，还可以了解在创建它时我的期望是什么。这就是我用在 stevedore 上的方法。

我知道我想让 stevedore 能够提供一组类用来管理应用程序的插件。在设计阶段，我花了些时间思考我见过的使用插件的通用模式，并且写了几页粗略的文档描述这些类应该如何使用。我意识到，如果我在类的构造函数中放最复杂的参数，方法 `map()` 几乎是可互换的。这些设计笔记直接写进了 stevedore 官方文档的简介里，用来解释在应用程序中使用插件的不同模式和准则。

将一个模块加入 Python 标准库的流程是什么？

完整的流程和规范可以在 Python Developer's Guide（http://docs.python.org/devguide/stdlibchanges.html）中找到。

一个模块在被加入 Python 标准库之前，需要被证明是稳定且广泛使用的。模块需要提供的功能要么是很难正确实现的，要么是非常有用以至于许多开发人员已经创建了他们自己不同的变种。API 应该非常清晰并且它的实现不能依赖任何标准库之外的库。

提议一个新模块的第一步是在社区通过 python-ideas 邮件列表非正式地了解一下大家对此的感兴趣程度。如果回应很积极，下一步就是创建一个 Python 增强提案（PythonEnhancement Proposal，PEP），它包括添加这个模块的动因，以及如何过渡的一些实现细节。

因为包的管理和发现工作已经非常稳定了，尤其是 pip 和 Python Package Index（PyPI），因此在标准库之外维护一个新的库可能更实用。单独的发布使得对于新功能和 bug 修复（bugfix）的更新可以更频繁，对于处理新技术或 API 的库来说这尤其重要。

标准库中的哪三个模块是你最想人们深入了解并开始使用的？

最近我做了许多关于应用程序中动态加载扩展方面的工作。我使用 **abc** 模块为那些作为抽象基类进行的扩展定义 API，以帮助扩展的作者们了解 API 的哪些方法是必需的，哪些是可选的。抽象基类已经在其他一些语言中内置了，但我发现很多 Python 程序员并不知道 Python 也有。

bisect 模块中的二分查找算法是个很好的例子，一个广泛使用但不容易正确实现的功能，因此它非常适合放到标准库中。我特别喜欢它可以搜索稀疏列表，且搜索的值可能并不在其中。

collections 模块中有许多有用的数据结构并没有得到广泛使用。我喜欢用 **namedtuple** 来创建一些小的像类一样的数据结构来保存数据但并不需要任何关联逻辑。如果之后需要添加逻辑的话，可以很容易将 **namedtuple** 转换成一个普通的类，因为 namedtuple 支持通过名字访问属性。另一个有意思的数据结构是 **ChainMap**，它可以生成良好的层级命名空间。ChainMap 能够用来为模板解析创建上下文或者通过清晰的流程定义来管理不同来源的配置。

许多项目（包括 OpenStack）或者外部库，会在标准库之上封装一层自己的抽象。例如，我特别想了解对于日期/时间的处理。对此你有什么建议吗？程序员应该坚持使用标准库，还是应该写他们自己的函数，切换到其他外部库或是开始给 Python 提交补丁？

所有这些都可以。我倾向于避免重复造轮子，所以我强烈主张贡献补丁和改进那些能够

用来作为依赖的项目。但是，有时创建另外的抽象并单独维护代码也是合理的，不管在应用程序内还是作为一个新的库。

你提到的例子中，OpenStack 里的 **timeutils** 模块就是对 Python 的 **datetime** 模块的一层很薄的封装。大部分功能都简短且简单，但通过将这些最常见的操作封装为一个模块，我们可以保证它们在 OpenStack 项目中以一致的方式进行处理。因为许多函数都是应用相关的，某种意义上它们强化了一些问题决策，例如，字符串时间戳格式或者“现在”意味着什么，它们不太适合作为 Python 标准库的补丁或者作为一个通用库发布以及被其他项目采用。

与之相反，我目前正致力于将 OpenStack 的 API 服务项目从早期创建时使用的 WSGI 框架转成采用一个第三方 Web 开发框架。在 Python 中开发 WSGI 应用有很多选择，并且当我们可能需要增强其中一个以便其可以完全适应 OpenStack API 服务器的需要时，将这些可重用的修改贡献对于维护一个“私有的”框架似乎更可取。

当从标准库或其他地方导入并使用大量模块时，关于该做什么你有什么特别的建议吗？

我没有什么硬性限制，但是如果我有过多的导入时，我会重新考虑这个模块的设计并考虑将其拆到一个包中。与上层模块或者应用程序模块相比，对底层模块的这种拆分可能会发生得更快，因为对于上层模块我期望将更多片段组织在一起。

关于 Python 3，有什么模块是值得一提而且能令开发人员有兴趣深入了解的？

支持 Python 3 的第三方库的数量已经到了决定性的时刻。针对 Python 3 开发新库或应用程序从未如此简单过，而且幸亏有 3.3 中加入的兼容性功能使同时维护对 Python 2.7 的支持也很容易。主要的 Linux 发行版正在致力于将 Python 3 默认安装。任何人要用 Python 创建新项目都应该认真考虑对 Python 3 的支持，除非有尚未移植的依赖。目前来说，不能运行在 Python 3 上的库基本会被视为“不再维护”。

许多开发人员将所有的代码都写入到应用程序中，但有些情况下可能有必要将代码封装成一个库。关于设计、规划、迁移等，做这些最好的方式是什么？

应用程序就是“胶水代码”的集合用来将库组织在一起完成特定目的。起初设计时可以将这些功能实现为一个库，然后在构建应用程序时确保库的代码能够很好地组织到逻辑单元中，这会让测试变得更简单。这还意味着应用程序的功能可以通过库进行访问，并且能够被重新组合以构建其他应用程序。未能采用这种方法的话意味着应用程序的功能和用户界面的绑定过于紧密，导致很难修改和重用。

对于计划开始构建自己的 Python 库的人们有什么样的建议呢？

我通常建议自顶向下设计库和 API，对每一层应用单一职责原则（Single Responsibility Principle，SRP）（http://en.wikipedia.org/wiki/Single_responsibility_principle）这样的设计准则。考虑调用者如何使用这个库，并创建一个 API 去支持这些功能。考虑什么值可以存在一个实例中被方法使用，以及每个方法每次都要传入哪些值。最后，考虑实现以及是否底层的代码的组织应该不同于公共 API。

SQLAlchemy 是应用这些原则的绝好例子。声明式 ORM、数据映射和表达式生成层都是单独的。开发人员可以自行决定对于 API 访问的正确的抽象程度，并基于他们的需求而不是被库的设计强加的约束去使用这个库。

当你随机看 Python 程序员的代码时遇到的最常见的编程错误是什么？

Python 的习惯用法和其他语言的一个较大的不同在于循环和迭代。例如，我见过的最常见的反模式是使用 `for` 循环过滤一个列表并将元素加入到一个新的列表中，然后再在第二个循环中处理这个结果（可能将列表作为参数传给一个函数）。我通常建议将过滤循环改成生成器表达式，因为生成器表达式，更有效也更容易理解。列表的组合也很常见，以便它们的内容可以以某种方式一起被处理，但却没有使用 `itertools.chain()`。

还有一些我在代码评审时给出的更细小的建议，例如，使用 `dict()` 而不是长的 `if:then:else` 块作为查找表，确保函数总是返回相同的类型（如一个空列表而不是 None），通过使用元组和新类将相关的值合并到一个对象中从而减少函数的参数，以及在公共 API 中定义要使用的类而不是依赖于字典。

有没有关于选择了一个“错误”的依赖的具体的例子是你亲身经历或目睹过的？

最近，我有个例子，pyparsing（https://pypi.python.org/pypi/pyparsing）的一个新发布取消了对 Python 2 的支持，这给我正在维护的一个库带来了一点儿小麻烦。对 pyparsing 的更新是个重大的修改，而且是明确标识成这样的，但是因为我没有在对 cliff（https://pypi.python.org/pypi/cliff）的设置中限制依赖版本号，所以 pyparsing 的新发布给 cliff 的用户造成了问题。解决方案就是在 cliff 的依赖列表中对 Python 2 和 Python 3 提供不同的版本边界。这种情况突显了理解依赖管理和确保持续集成测试中适当的测试配置的重要性。

你怎么看待框架？

框架像任何工具类型一样。它们确实有帮助，但在选择框架时要特别谨慎，应确保它能

够很好地完成当前的工作。

通过抽取公共部分到一个框架中，你可以将你的开发精力专注于应用中独特的方面。通过提供许多类似运行在开发模式或者写一个测试套件这样的引导代码，它们还可以帮你让一个应用程序迅速达到一个可用的状态而不是从头开发。它们还可以激励你在应用程序开发过程中保持一致，这意味着最终你的代码将更易于理解且更可重用。

虽然使用框架时还有其他一些潜在的缺点需要注意。决定使用某个特定框架通常能够反映应用程序本身的设计。如果设计的限制不能从根本上符合应用程序的需求，那么选择错误的框架会令应用的实现变得更难。如果你试着使用与框架建议不同的模式或惯用方式，你最终将不得不同框架做斗争。

2.6　管理 API 变化

在构造 API 时很难一蹴而就。API 需要不断演化、添加、删除或者修改所提供的功能。

在后面的段落中 women 将讨论如何管理公共 API 的变化。公共 API 是指将应用程序或库暴露给终端用户的 API。内部 API 则有另外的考虑，并且由于它们在内部（也就是说用户不需要直接操作这些 API），因而可以任意处理它们：分解、调整或者根据需要任意使用。

这两种 API 很容易区分。Python 的传统是用下划线作为私有 API 的前缀，如 `foo` 是公共 API，而 `_bar` 是私有的。

在构建 API 时，最糟糕的事情莫过于 API 被突然破坏。Linus Torvalds 就因对 Linux 内核公共 API 破坏的零容忍而闻名。考虑到如此多的人依赖 Linux，可以说他的选择是非常明智的。

Unix 平台的库管理系统很复杂，它依赖于 soname（http://en.wikipedia.org/wiki/Soname）和细粒度的版本标识符。Python 中没有这样的系统，也没有对应的转换。因此完全取决于维护者如何选择正确的版本号和策略。但是，关于如何定义自己的库或应用程序的版本，你依然可以将 Unix 系统作为你的灵感来源。通常，版本号应该反映出 API 对用户的影响，大部分开发人员通过主版本号的增加来表示此类变化，但这取决于你对版本号管理的方法，你也可以采用增加小版本号的方式。

不管如何决定，最重要的一步就是在修改 API 时要通过文档对修改进行详细地记录，包括：

- 记录新的接口；
- 记录废除的旧的接口；
- 记录如何升级到新的接口。

旧接口不要立刻删除。实际上，应该尽量长时间地保留旧接口。因为已经明确标识为作废，所以新用户不会去使用它。在维护实在太麻烦时再移除旧接口。API 变化的记录见示例 2.2.

示例 2.2 API 变化的记录

```
class Car(object):
    def turn_left(self):
        """Turn the car left.

        .. deprecated:: 1.1
          Use :func:`turn` instead with the direction argument set to left
        """
        self.turn(direction='left')

    def turn(self, direction):
        """Turn the car in some direction.

        :param direction: The direction to turn to.
        :type direction: str
        """
        # Write actual code here instead
        pass
```

使用 Sphinx 标记强调修改是个好主意。在构建文档时，用户应该能清楚地知道某个功能不应该再被使用，并且可以直接访问到新功能，并随之解释如何升级旧代码。这个方法的缺点就是，你不能指望开发人员在升级你的 Python 包到新版本时会去读你的修改日志或者文档。

Python 提供了一个很有意思的名为 **warnings** 的模块用来解决这一问题。这一模块允许代码发出不同类型的警告信息，如 **PendingDeprecationWarning** 和 **DeprecationWarning**。这些警告能够用来通知开发人员某个正在调用的函数已经废弃或即将废弃。这样，开发人员

就能够看到他们正在使用旧接口并且应该相应地进行处理[①]。

回到之前的例子，我们可以利用它向用户发出警告，如示例 2.3 所示。

示例 2.3　带警告的 API 变化的记录

```
import warnings

class Car(object):
    def turn_left(self):
        """Turn the car left.

        .. deprecated:: 1.1
           Use :func:`turn` instead with the direction argument set to "left".
        """
        warnings.warn("turn_left is deprecated, use turn instead",
                      DeprecationWarning)
        self.turn(direction='left')

    def turn(self, direction):
        """Turn the car in some direction.

        :param direction: The direction to turn to.
        :type direction: str
        """
        # Write actual code here instead
        pass
```

任何调用了废弃的 `turn_left` 函数的代码，都将引发一个警告：

```
>>> Car().turn_left()
__main__:8: DeprecationWarning: turn_left is deprecated, use turn instead
```

注意

从 Python 2.7 开始，`DeprecationWarning` 默认将不显示。可以通过在调用 `python` 时指定 `-W all` 选项来禁用这一过滤器。关于 `-W` 的可用值的更多信息可以参考 `python` 手册。

① 对要和C打交道的Python开发人员来说，这是一个很方便的与`__attribute__((deprecated))` GCC扩展的对应物。

让你的代码告诉开发人员他们的程序正在使用某些最终将要停止工作的东西是明智的，因为这其实也可以自动化。当运行他们的测试集合时，开发人员可以在执行 `python` 时使用 `-W error` 选项，它会将警告转换为**异常**。这意味着一个废弃的函数每次被调用时都会有一个错误被抛出，这样使用你的库的开发人员就可以很容易知道如何具体修改他们的代码。

示例 2.4 运行 `python -W error`

```
>>> import warnings
>>> warnings.warn("This is deprecated", DeprecationWarning)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
DeprecationWarning: This is deprecated
```

2.7 Christophe de Vienne 访谈

Christophe 是一名 Python 程序员，并且是 WSME（Web Services Made Easy）的作者。开发人员可以使用这个框架定义 Python 风格的 Web 服务，并且支持多种丰富的 API，且允许作为插件被集成到其他 Web 框架中。

在设计 Python 的 API 时开发人员常犯的错误是什么？

有许多我在设计 Python API 时试图避免的错误。

- 过于复杂。常言道：“Keep It Simple”（也有人说“Keep It Simple Stupid”，但我认为“simple”和“stupid”并不兼容）。复杂的 API 会很难理解也难以文档化。不过没必要让实际的库函数也太简单，但让库函数简单也是一个明智的想法。一个好的例子就是 Request 库（http://www.python-requests.org/），与其他标准 **urllib** 库相比，Requests 的 API 是非常简单且自然的，但是在背后它做了很多复杂的工作。相比而言，**urllib** API 几乎与它做的事一样复杂。

- 施展（看得见的）魔力。当你的 API 做一些你的文档没有阐述的事情时，终端用户会想破解你的代码，看看背后到底发生了什么。如果你已经让魔力在后台发生了还好，但终端用户永远都不应该在前台看到任何不正常的事情发生。
- 忘记用例。当写代码深入到库的内部时，很容易忘记实际上你的库应该如何被使用。想出好的用例能够让 API 的设计更加容易。
- 不写单元测试。测试驱动开发（TDD）是开发库时非常有效的方式，尤其是在 Python 中。它强迫开发人员从一开始就假设自己是终端用户并能维护版本间的兼容性。而且这是我知道的唯一能够让你完全重写一个库的方法。尽管这并不总是必需的，但有选择总是好的。

考虑到 WSME 可能运行在多种不同的框架之上，什么样的 API 是它必须支持的？

实际上并没有那么多，因为它能够运行在其上的框架很多方面是类似的。它们使用装饰器（decorator）给外面的世界暴露函数和方法，它们都是基于 WSGI 标准的（所以它们的请求对象看上去非常类似），而且它们多少都以彼此作为灵感来源。也就是说，我们还没有试图将其插入一个异步的 Web 框架中，如 Twisted。

我处理过最大的不同是上下文信息的访问方式。在 Web 框架中，上下文主要是可以从中导出或附加（身份信息、会话数据、数据连接等）信息的请求，以及一些全局的东西，如全局配置、连接池等。大部分 Web 框架会假设它们运行在多线程服务器上并且将所有这些信息看作是线程独立数据（Thread-Specific Data，TSD）的。这使得它们可以通过导入来自一个模块的请求代理对象来访问当前的请求并与其一起工作。尽管它使用起来很容易，但它也暗含了一点小魔法并且使全局对象缺乏特定上下文数据。

例如，Pyramid 框架的工作方式就不太一样。取而代之的是，上下文被显式地注入到使用它的代码段中。这就是为什么视图会接收一个“request”作为参数，它封装了 WSGI 的环境变量并提供对应用程序全局上下文的访问。

它们的优缺点各是什么？

类似 Pyramid 的 API 风格有个很大的优点，它允许一个单独的程序以非常自然的方式运行在几个完全不同的环境中。缺点就是学习曲线有点儿陡。

Python 是怎样让库 API 的设计更简单或更难的呢？

缺乏内置的定义哪部分公共及哪部分私有的方式，这是个（小）问题也是个优点。

当开发人员对哪部分是他们的 API 哪部分不是的问题欠考虑的时候，它就会是问题。但是，通过一点规则、文档和（如果需要的话）类似 `zope.interface` 的工具，它就将不再是问题了。

它的优点在于能够让 API 的重构快速而简单，同时保持对前面版本的兼容。

对于 API 的演化、废弃、移除等你的经验法则是什么？

在做决定的时候我会用下面这几个标准去衡量。

- **对用户来说让库适应他们的代码有多困难？**设想已经有人依赖你的 API，任何你所做出的改动都必须让用户觉得为适应这种变化所付出的努力是值得的。这一规则是为了避免对被广泛使用的 API 部分做出不兼容的修改。也就是说，Python 的优点之一就是让重构代码以适应一个 API 的变化相对容易。
- **这个修改会让维护变得更容易吗？**简化实现，清理代码库，让 API 变得更容易使用，有更完整的单元测试，让 API 一看就很容易理解……所有这些都将让你作为维护者的生活更轻松。
- **修改之后会让我的 API 变得多（不）一致？**如果所有的 API 函数都遵循一个类似的模式（如在第一个位置要求同样的参数），那么确保新函数也遵循同样的模式是很重要的。而且，一次做很多事情通常什么都做不好，让你的 API 专注于它要做的。
- **用户如何从这次修改中获益？**最后但同样重要的是，总是从用户的角度考虑问题。

对为 Python 中的 API 建立你有什么建议吗？

文档可以让新用户更容易采用你的库。忽视文档会赶走很多潜在用户，而且还不止是初学者。但问题在于，写文档是很难的，所以它经常被忽略。

尽早写文档，并在持续集成中包含文档构建。现在我们有 Read the Docs（https://readthedocs.org/），没理由在不包含文档生成和发布（至少对开源软件来说如此）。

使用 docstring 对 API 的类和函数进行文档化。遵循 PEP 257 规范（http://www.python.org/dev/peps/pep-0257/）[①]，以便开发人员不必读你的源代码就能理解你的 API 是做什么的。从 docstring 生成 HTML 文档，并且不要限制对 API 的引用。

自始至终提供实用的例子。至少包括一个“入门指南”，向新手展示如何构建一个可以

① *Docstring Conventions*, David Goodger, Guido van Rossum, 29 May 2001

运行的例子。文档的第一页应该提供一个关于 API 基本情况的快速概览和有代表性的用例。

在文档中一个版本接一个版本地体现 API 演进的细节。（只有 VCS 日志是不够的！）

让你的文档可访问，可能的话，让它读起来更舒服些。你的用户需要能够很容易地找到文档，并从中获取他们需要的信息而没有任何被折磨的感觉。通过 PyPI 发布你的文档就可以实现这一点，通过 Read the Docs 发布也是很不错的方法，因为用户会希望能够在那里找到你的文档。

最后，选择一个高效且吸引人的主题。我为 WSME 选择了“Cloud”Sphinx 主题，但实际上有大量的主题可供选择。没必要为了做出好看的文档而成为 Web 专家。

第 3 章 文档

正如在前面提到过的，文档是软件开发的重要组成部分。但是，仍然有很多项目缺乏很好的文档。文档编写被看作是复杂而艰巨的任务，但其实大可不必如此。利用一些 Python 程序员可用的工具，可以令代码的文档编写工作就像写代码一样容易。

导致文档稀少或干脆没有文档的最大元凶之一就是，很多人想当然地认为文档只能手工编写。即使有多个人做同一个项目，也只是意味着最终会有一个或多个人不得不疲于应付编程并维护文档。而且如果你问任何一个开发人员更喜欢做哪种工作，他们肯定会说他们宁愿开发软件而不愿意为软件写文档。有时文档的流程甚至完全独立于开发流程，这意味着可能写文档的人甚至从来没有实际写过一行代码。而且，任何这种方式生成的文档很可能是过时的。不管文档是由程序员自己来完成还是有专门的文档编写人员来完成，纯手工的文档编写方式几乎都不可能跟上开发节奏的。

归根结底，代码和文档离得越远，对文档的维护就越难。所以说，为什么要让代码和文档完全分开呢？实际上不仅可以直接将文档放到代码里，而且可以很容易地将文档转换成可读的 HTML 或者 PDF 文件。

Python 中文档格式的事实标准是 reStructuredText，或简称 reST。它是一种轻量级的标记语言（类似流行的 Markdown），在易于计算机处理的同时也便于人类读写。Sphinx（http://sphinx-doc.org/）是处理这一格式最常用的工具，它能读取 reST 格式的内容并输出其他格式的文档。

项目的文档应该包括下列内容。

- 用一两句话描述这个项目要解决的问题。
- 项目所基于的分发许可。如果是开源软件的话，应该在每一个代码文件中包含相应信息。因为上传代码到互联网并不意味着人们知道他们可以对代码做什么。

- 一个展示项目如何工作的小例子。
- 安装指南。
- 指向社区支持、邮件列表、IRC、论坛等的链接。
- 指向 bug 跟踪系统的链接。
- 指向源代码的链接，以便开发人员可以下载并立刻投入开发。

还应该包括一个 `README.rst` 文件，解释这个项目是做什么的。这个 README 文件会显示在 GitHub（https://github.com/）或 PyPI（http://pypi.python.org）的项目页面上。两个网站都可以处理 reST 格式。

提示

如果正在使用 GitHub，那么也可以添加一个 `CONTRIBUTING.rst` 文件，这个文件会在有人创建 pull 请求时显示。它应该给出一组检查项以便开发人员在提交代码之前对照检查，如遵守 PEP 8 或者不要忘记运行单元测试。

提示

Read The Docs（http://readthedocs.org）可以自动在线生成和发布文档。在上面注册并配置项目是一个很直接地流程，它会搜索 Sphinx 配置文件，构建文档，然后让用户可以访问文档。它是代码托管网站的非常好的搭配。

3.1　Sphinx 和 reST 入门

首先，需要在项目的顶层目录运行 `sphinx-quickstart`。这会创建 Sphinx 需要的目录结构，同时会在文件夹 `doc/source` 中创建两个文件，一个是 `conf.py`，它包含 Sphinx 的配置信息（当然也是 Sphinx 运行所必需的），另一个文件是 `index.rst`，它将作为文档的首页。

然后就可以通过在调用命令 `sphinx-build` 时给出源目录和输出目录来生成 HTML 格式的文档：

```
$ sphinx-build doc/source doc/build
  import pkg_resources
```

```
Running Sphinx v1.2b1
loading pickled environment... done
No builder selected, using default: html
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
preparing documents... done
writing output... [100%] index
writing additional files... genindex search
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded.
```

现在就可以在心仪的浏览器中打开 doc/build/index.html 并开始阅读文档了。

提示

如果使用了 setuptools 或者 pbr（参见 4.2 节）进行打包，Sphinx 会对它们进行扩展以支持命令 setup.py build_sphinx，这个命令会自动运行 sphinx-build。pbr 对 *Sphinx* 包含比较完善的默认配置，如输出文档到 doc 子目录中。

index.rst 是文档开始的地方，但并不局限于此。reST 支持包含，所以完全可以将文档分成多个文件。刚开始不必太担心语法和语义，尽管 reST 确实提供了不少格式，但后面有很多时间去了解。reST 的完全指南（http://docutils.sourceforge.net/docs/ref/rst/ restructuredtext.html）介绍了如何创建标题、列表、表格等。

3.2 Sphinx 模块

Sphinx 是高度可扩展的：它的基本功能只支持手工文档，但它有许多有用的模块可以支持自动化文档和其他功能。例如，sphinx.ext.autodoc 可以从模块中抽取 rest 格式的文档字符串（docstrings）并生成.rst 文件。sphinx-quickstart 在运行的时候会问你是否想激活某个模块，也可以编辑 conf.py 文件并将其作为一个扩展。

```
extensions = ['sphinx.ext.autodoc']
```

值得注意的是，autodoc **不会**自动识别并包含模块，而是需要显式地指明需要对哪些模块生成文档，类似下面这样：

```
.. automodule:: foobar
   :members: ❶
   :undoc-members: ❷
   :show-inheritance: ❸
```

❶ 要求输出所有已加文档的成员信息（可选）。

❷ 要求输出所有未加文档的成员信息（可选）。

❸ 显示继承关系（可选）。

同时要注意以下几点。

- 如果不包含任何指令，Sphinx 不输出任何内容。
- 如果只指定:members:，那么在模块 / 类 / 方法这一树状序列中未加文档的节点将被忽略，即使其成员是加了文档的。例如，如果给一个类的所有方法都加了文档，但这个类没有加文档，:members:除这个类及其方法。为了避免这种情况，要么必须为该类加上一个文档字符串，要么同时指定 :undoc-members:。
- 模块需要在 Python 可以导入的位置。通过添加.、..和/或../..到 sys.path 中会对此有帮助。

autodoc 可以将实际源代码中的大部分文档都包含进来，甚至还可以单独挑选某个模块或方法生成文档，而不是一个“非此即彼”的解决方案。通过直接关联源代码来维护文档，可以很容易地保证文档始终是最新的。

如果你正在开发一个 Python 库，那么通常需要以表格的形式来格式化你的 API 文档，表格中包含到各个模块的独立的文档页面的链接。sphinx.ext.autogen 模块就是用来专门处理这一常见需求的。首先，需要在 conf.py 中启动它：

```
extensions = ['sphinx.ext.autodoc', 'sphinx.ext.autosummary']
```

现在就可以在一个.rst 中加入类似下面的内容来自动为特定模块生成 TOC：

```
.. autosummary::

   mymodule
   mymodule.submodule
```

这会生成名为 generated/mymodule.rst 和 generated/mymodule.submodule.rst 的文件，其中会包含前面提到的 autodoc 指令。使用同样的格式，还可以指定希望模块 API 的哪部分包含在文档中。

提示

在大规模的项目中，手工添加模块到这个列表中是比较麻烦的。要记得 conf.py 是个普通的 Python 源文件，所以完全可以在里面写自己的代码，包括写代码去自动创建指明哪些模块需要生成文档的.rst 文件。

Sphinx 的另一个有用的功能是能够在生成文档时自动在例子上运行 doctest。doctest 是标准的 Python 模块，它能够针对代码片段搜索文档并运行代码以测试其是否反映代码的实际行为。每个以>>>（即主要的提示符）开始的段落会被看作是一个要测试的代码段。

```
To print something to the standard output, use the :py:func:`print` function.

    >>> print("foobar")
    foobar
```

在你的 API 演进的过程中很容易忘记对例子进行修改，doctest 可以帮助你避免这类问题的发生。如果文档包含一份详细的分步指南，doctest 能够确保其在开发过程中保持最新。也可以使用 doctest 做文档驱动开发（Documentation-Driven Development，DDD）：先写文档和例子，然后写代码去匹配文档。

通过特殊的 doctest 生成器，利用这个功能就像运行 sphinx-build 一样简单：

```
$ sphinx-build -b doctest doc/source doc/build
Running Sphinx v1.2b1
loading pickled environment... done
building [doctest]: targets for 1 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
running tests...

Document: index
---------------
1 items passed all tests:
   1 tests in default
1 tests in 1 items.
1 passed and 0 failed.
Test passed.

Doctest summary
```

```
===============
   1 test
   0 failures in tests
   0 failures in setup code
   0 failures in cleanup code
build succeeded.
```

Sphinx 还提供了很多其他功能，自带或者通过扩展模块，包括

- 项目间使用的链接；
- HTML 主题；
- 图表和公式；
- 输出为 Texinfo 和 EPUB 格式；
- 链接到外部文档。

可能你现在并不需要所有这些功能，但是如果未来需要的话，提前知道有模块能提供这些功能还是不错的。

3.3　扩展 Sphinx

有时现成的方案还不够。如果你写的是一个在 Python 内部调用的 API 那么问题不大，但如果写的是一个 HTTP REST API，Sphinx 就只能为你的 API 生成 Python 端的文档，因此你不得不手工编写 REST API 文档应处理随之而来的其他问题。

WSME（https://pypi.python.org/pypi/WSME）的创建者们有别的主意。他们开发了一个名为 sphinxcontrib-pecanwsme（https://pypi.python.org/pypi/sphinxcontrib-pecanwsme）的 Sphinx 的扩展，它可以分析文档字符串和实际的 Python 代码并自动生成 REST API 文档。你也可以对自己的项目这么做，将代码中的有用信息抽取到文档中，只有让这个过程自动化才有意义。

提示

针对其他 HTTP 框架，如 Flask、Bottle 和 Tornado，可以使用 `sphinxcontrib.httpdomain`（http://pythonhosted.org/sphinxcontrib-httpdomain/）。我个人的观点是，无论任何时候，只要能从代码中抽取信息帮助生成文档，都值得去做并且将其自动化。这比手工维护文档要好得多，尤其是可以利用自动发布工具（如 Read The Docs）的时候。

要开发 Sphinx，首先是要编写一个模块，最好是作为 sphinxcontrib 的一个子模块（如果模块足够通用的话），并且起个名字。Sphinx 需要该模块有个预定义的名为 setup(app)的函数。app 对象会包含用来将你的代码连接到 Sphinx 事件和指令的方法。完整的方法列表可以在 Sphinx 扩展 API（http://sphinx-doc.org/ext/appapi.html）中找到。

例如，sphinxcontrib-pecanwsme 利用 setup(app)函数添加了一个名为 rest-controller 的单条指令。添加的这条指令需要 WSME 控制器的类名是完整的限定名来为其生成文档。

示例 3.1 摘自 sphinxcontrib.pecanwsme.rest.setup 的代码

```
def setup(app):
    app.add_directive('rest-controller', RESTControllerDirective)
```

RESTControllerDirective 是个指令类，它必须包含特定的属性和方法，就像在 Sphinx 扩展 API（http://sphinx-doc.org/ext/appapi.html#sphinx.application.Sphinx.add_directive）中描述的那样。主方法 run()会负责完成从代码中抽取文档的实际工作。

sphinx-contrib 资源库（https://bitbucket.org/birkenfeld/sphinx-contrib/src）包括一组能够帮助你开发自己的扩展模块的小模块。

注意

尽管 Sphinx 是用 Python 开发的，而且默认也主要面向 Python，但是它有很多可用的扩展使它可以支持其他语言。所以，即使项目同时使用了多种语言，也可以用 Sphinx 为整个项目生成文档。

第 4 章 分发

我敢打赌你将来肯定要分发你的软件。你可能只是想将代码打个压缩包然后上传到互联网，即便如此 Python 也提供了相应的工具，确保你的用户在安装你的软件的过程中不会遇到麻烦。你应该已经熟知如何用 `setup.py` 安装 Python 应用程序和库，但是你可能从未深究过它背后的运行机制，也没有探究过如何生成自己的 `setup.py`。

4.1 简史

distutils 自从 1998 年便已经是 Python 标准库的一部分了。它最早由 Greg Ward 开发，目的是要创造一种简单的方式供开发人员为最终用户自动化软件安装过程。

示例 4.1 使用 distutils 的 `setup.py`

```
#!/usr/bin/python
From distutils.core import setup

setup(name="rebuildd",
      description="Debian packages rebuild tool",
      author="Julien Danjou",
      author_email="acid@debian.org",
      url="http://julien.danjou.info/software/rebuildd.html",
      packages=['rebuildd'])
```

就这么简单。用户要生成或安装软件只需通过合适的命令运行 `setup.py` 即可。如果你的发布中包含了除原生 Python 之外的 C 语言模块，它甚至也可以自动处理。

distutils 的开发在 2000 年就停止了。从那时起，一些开发人员开始在其基础上继续开发他们自己的工具。其中最成功的 distutils 的继任者便是打包库 setuptools，它提供了更频繁的

更新和更多的高级功能，如自动依赖处理、Egg 分发格式以及 `easy*install` 命令。由于 distutils 仍然是包含在 Python 标准库中的软件打包的一种标准方式，因此 setuptools 也提供了一定程度上的向后兼容。

示例 4.2　使用 setuptools 的 `setup.py`

```
#!/usr/bin/env python
import setuptools

setuptools.setup(
    name="pymunincli",
    version="0.2",
    author="Julien Danjou",
    author_email="julien@danjou.info",
    description="munin client library",
    license="GPL",
    url="http://julien.danjou.info/software/pymunincli/",
    packages=['munin'],
    classifiers=[
        "Development Status :: 2 - Pre-Alpha",
        "Intended Audience :: Developers",
        "Intended Audience :: Information Technology",
        "License :: OSI Approved :: GNU General Public License (GPL)",
        "Operating System :: OS Independent",
        "Programming Language :: Python"
    ],
)
```

最终，setuptools 的开发也变得缓慢了，因为人们开始认为它很可能像最早的 distutils 一样死去。于是，不久后另一伙开发人员又基于 setuptools 创建了一个新的名为 distribute 的库，它具有一些超越 setuptools 的优点，包括 bug 更少且支持 Python 3。所有的好故事都有个曲折的结局，这个也不例外。2013 年 3 月，setuptools 和 distribute 两个开发组决定基于原始的 setuptools 项目合并他们的代码库（http://mail.python.org/pipermail/distutils-sig/2013- March/020126.html）。所以现在 distribute 已经被废弃，setuptools 又重新成为处理高级 Python 安装的标准方式。

尽管这一切已经在发生，还是有另一个名为 distutils2 的项目在开发中，意欲全面取代 Python 标准库中的 distutils。它与 distutils 和 setuptools 的最明显的区别是，它会将包的元数据存储在纯

文本文件 `setup.cfg` 中，这使得开发人员写起来简单并且外部工具读取也容易。然而，它还是留有 distutisl 的一些缺陷，例如，晦涩的基于命令的设计，缺少对入口点（entry point）以及在 Windows 上执行原生脚本的支持，而这两个功能 setuptools 都支持。因为这些以及其他一些原因，最终在 Python 3.3 标准库中包含 setuptools 的计划再次落空，这一项目在 2012 年被废弃。

然而，packaging 仍有机会通过 distlib（https://readthedocs.org/projects/distlib/)涅槃重生，它正致力于取代 distutils，并将（希望是）成为 Python 3.4 标准库的一部分。它包含来自 packaging 的最好的功能同时实现了与打包有关的 PEP 中描述的基本内容。

简单回顾一下。

- distutils 是标准库的一部分，能处理简单的包的安装。
- setuptools，领先的包安装标准，曾经被废弃但现在又继续开发。
- distribute 从 0.7 版本开始并入了 setuptools。
- distutils2（也称为 packaging）已经被废弃。
- distlib 可能将来会取代 distutils。

尽管这 5 个打包工具是实际工作中最常见的，但还是有许多其他的打包库。在网上搜索相关信息要谨慎，正因为上述复杂的历史变迁，所以有大量的文档都是过期的。不过至少官方文档（http://pythonhosted.org/setuptools/）是最新的。

简而言之，setuptools 是目前分发库的主要选择，但在未来要对 distlib 保持关注。

4.2 使用 pbr 打包

现在我已经用了好几页让你对如此多的分发工具更加迷糊，接下来让我们谈谈另一个工具，也是一个不同的选择，名为 pbr。

你可能已经开发过一些包并试图去写 `setup.py`，或者从其他项目复制一个，或者自己啃文档。这不是一个清晰明确的任务，如同我们在前面讨论过的，选择什么工具通常是第一个障碍。本节将介绍 pbr，一个应该用来开发你的下一个 `setup.py` 的工具，以便你不用再在这部分浪费时间。

pbr 是指 Python Build Reasonableness。这个项目已经在 OpenStack（http://openstack.org）

内部启动，并围绕 setuptools 开发了一系列用来辅助包的安装和部署的工具。它从 distutils2 获得了灵感，利用 `setup.cfg` 文件来描述包的用途。

pbr 使用的 `setup.py` 文件类似下面这样：

```
import setuptools

setuptools.setup(setup_requires=['pbr'], pbr=True)
```

就两行代码，非常简单。实际上安装所需要的元数据存储在 `setup.cfg` 文件中：

```
[metadata]
name = foobar
author = Dave Null
author-email = foobar@example.org
summary = Package doing nifty stuff
license = MIT
description-file =
    README.rst
home-page = http://pypi.python.org/pypi/foobar
requires-python = >=2.6
classifier =
    Development Status :: 4 - Beta
    Environment :: Console
    Intended Audience :: Developers
    Intended Audience :: Information Technology
    License :: OSI Approved :: Apache Software License
    Operating System :: OS Independent
    Programming Language :: Python

[files]
packages =
    foobar
```

看着眼熟？没错，处理的方式都是直接受 distutils2 的启发。

pbr 还提供了其他一些功能，例如：

- 基于 `requirements.txt` 做自动依赖安装；
- 利用 Sphinx 实现文档自动化；

- 基于 *git* history 自动生成 `AUTHORS` 和 `ChangeLog` 文件；
- 针对 *git* 自动创建文件列表；
- 基于 *git* tags 的版本管理。

所有这些对开发人员来说只有一点儿或完全没有任何额外工作要做。pbr 目前维护良好并且开发很活跃，所以如果计划分发软件的话，应该认真考虑一下使用 pbr。

4.3 Wheel 格式

在 Python 出现后的大部分时间里，都没有官方的标准分发格式。尽管不同的分发工具大多使用了一些比较通用的归档格式，但它们的元数据和包的结构彼此并不兼容，例如，由 setuptools 引入的 Egg 格式只是一个有着不同扩展名的压缩文件。这一问题在官方安装标准最终敲定之后变得更加复杂，官方标准同已有标准并不兼容。

为了解决这些问题，PEP 427（http://www.python.org/dev/peps/pep-0427/）针对 Python 的分发包定义了新的标准，名为 Wheel。已经有相应工具作为这一格式的参考实现，也命名为 `wheel`（https://pypi.python.org/pypi/whee）。

pip（https://pypi.python.org/pypi/pip）从 1.4 版本开始支持 Wheel。如果正在使用 **setuptools** 并且安装了 `wheel` 包，那么会自动集成为一个命令：

```
python setup.py bdist_wheel
```

这条命令将在 `dist` 目录中创建 `.whl` 文件。和 Egg 格式类似，一个 Wheel 归档文件就是一个有着不同扩展名的压缩文件，只是 Wheel 归档文件不需要安装。可以通过在包名的后面加一个斜杠加载和运行代码：

```
$ python wheel-0.21.0-py2.py3-none-any.whl/wheel -h
usage: wheel [-h]

            {keygen,sign,unsign,verify,unpack,install,install-scripts,
               convert,help}
            ...

positional arguments:
[...]
```

你可能会惊讶地发现，这并不是由 Wheel 格式引入的功能。实际上 Python 还可以像 Java 运行.jar 文件那样运行普通的压缩文件：

```
python foobar.zip
```

这等同于：

```
PYTHONPATH=foobar.zip python -m __main__
```

换句话说，程序中的__main__模块会自动从__main__.py 中被导入。也可以通过在斜杠后面指定模块名字来导入__main__，就像用 Wheel：

```
python foobar.zip/mymod
```

这等同于：

```
PYTHONPATH=foobar.zip python -m mymod.__main__
```

Wheel 的优点之一在于其命名转换，它允许指定软件的分发是否针对某一特定架构和/或 Python 实现（CPython、PyPy、Jython 等）。这在需要分发用 C 语言写的模块时尤其有用。

4.4　包的安装

setuptools 引入了第一个安装包的有用命令 easy_install。它通过一条命令即可从 Egg 归档文件中安装 Python 模块。遗憾的是，easy_install 从一开始就因为它有争议的行为（如忽视系统管理员的最佳实践以及缺少卸载功能）而口碑不好。

pip 项目提供了更好的安装包的方式。它的开发很活跃，维护良好，并且被包含在 Python 3.4 中[①]。它可以从 PyPI、tarball 或者 Wheel（参见 4.3 节）归档中安装或卸载包。

它的使用很简单：

```
$ pip install --user voluptuous
Downloading/unpacking voluptuous
  Downloading voluptuous-0.8.3.tar.gz
  Storing download in cache at ./.cache/pip/https%3A%2F%2Fpypi.python.org%2
    Fpackages%2Fsource%2Fv%2Fvoluptuous%2Fvoluptuous-0.8.3.tar.gz
  Running setup.py egg_info for package voluptuous
```

① 参见 PEP 453（http://www.python.org/dev/peps/pep-0453/）及 ensureepip 模块。

```
    WARNING: Could not locate pandoc, using Markdown long_description.

Requirement already satisfied (use --upgrade to upgrade): distribute in
  /usr/lib/python2.7/dist-packages (from voluptuous)
Installing collected packages: voluptuous
  Running setup.py install for voluptuous
    WARNING: Could not locate pandoc, using Markdown long_description.

Successfully installed voluptuous
Cleaning up...
```

也可以通过提供--user 选项让 pip 将包安装在 home 目录中。这可以避免将包在系统层面安装而造成操作系统目录的污染。

提示

如果要通过 pip 重复安装同一个包，可以设置本地缓存从而避免每次都去下载这个包。只需要将环境变量 PIP_DOWNLOAD_CACHE 指向一个目录，pip 就用它来保存下载的 tarball，并且在每次下载包之前先检查这个位置。这在使用 tox 时非常有用，tox 需要下载包来构建虚拟环境。也可以在~/.pip/pip.conf 文件中添加 download-cache 选项。

可以使用 pip freeze 命令列出当前已安装的包：

```
$ pip freeze
Babel==1.3
Jinja2==2.7.1
commando=0.3.4
...
```

所有其他的安装工具都正在被废弃以支持 pip，所以使用它作为包管理的一站式解决方案应该不会有什么问题。

4.5 和世界分享你的成果

一旦有了合适的 setup.py 文件，很容易生成一个用来分发的源代码 tarball。只需要使用 sdist 命令即可，如示例 4.3 所示。

示例 4.3　使用 `setup.py sdist`

```
$ python setup.py sdist
running sdist
[pbr] Writing ChangeLog
[pbr] Generating AUTHORS
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Processing SOURCES.txt
[pbr] In git context, generating filelist from git
warning: no previously-included files matching '*.pyc' found anywhere in
  distribution
writing manifest file 'ceilometer.egg-info/SOURCES.txt'
running check
copying setup.cfg -> ceilometer-2014.1.a6.g772e1a7
Writing ceilometer-2014.1.a6.g772e1a7/setup.cfg

[...]

Creating tar archive
removing 'ceilometer-2014.1.a6.g772e1a7' (and everything under it)
```

这会在你的源代码树的 `dist` 目录下创建一个 tarball，这可以用来安装你的软件。正如在 4.3 节中提到的，可以使用 `bdist_wheel` 命令构建 Wheel 归档文件。

最后一步是要让最终用户在通过 `pip` 命令安装你的包时能够知道包在哪里。这意味你需要将你的项目发布到 PyPI（http://pypi.python.org）。

如果是第一次，你很可能会犯错，最好能在一个安全的沙箱中测试发布流程而不是在生产服务器中。可以使用 PyPI 预付费服务器（PyPI staging server，https://testpypi.python.org/pypi）实现，它复制了主索引的全部功能，但是只用于测试目的。

第一步就是在测试服务器上注册你的项目。打开你的`~/.pypirc` 文件并加入下列行：

```
[distutils]
index-servers =
   testpypi

[testpypi]
username = <your username>
password = <your password>
repository = https://testpypi.python.org/pypi
```

现在就可以在索引中注册你的项目了：

```
$ python setup.py register -r testpypi
running register
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Reusing existing SOURCES.txt
running check
Registering ceilometer to https://testpypi.python.org/pypi
Server response (200): OK
```

最后，可以上传一个源代码分发 tarball：

```
% python setup.py sdist upload -r testpypi
running sdist
[pbr] Writing ChangeLog
[pbr] Generating AUTHORS
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Processing SOURCES.txt
```

```
[pbr] In git context, generating filelist from git
warning: no previously-included files matching '*.pyc' found anywhere in
  distribution
writing manifest file 'ceilometer.egg-info/SOURCES.txt'
running check
creating ceilometer-2014.1.a6.g772e1a7

[...]

copying setup.cfg -> ceilometer-2014.1.a6.g772e1a7
Writing ceilometer-2014.1.a6.g772e1a7/setup.cfg
Creating tar archive
removing 'ceilometer-2014.1.a6.g772e1a7' (and everything under it)
running upload
Submitting dist/ceilometer-2014.1.a6.g772e1a7.tar.gz to https://testpypi.
  python.org/pypi
Server response (200): OK
```

以及一个 Wheel 归档文件：

```
$ python setup.py bdist_wheel upload -r testpypi
running bdist_wheel
running build
running build_py
running egg_info
writing requirements to ceilometer.egg-info/requires.txt
writing ceilometer.egg-info/PKG-INFO
writing top-level names to ceilometer.egg-info/top_level.txt
writing dependency_links to ceilometer.egg-info/dependency_links.txt
writing entry points to ceilometer.egg-info/entry_points.txt
[pbr] Reusing existing SOURCES.txt
installing to build/bdist.linux-x86_64/wheel
running install
running install_lib
creating build/bdist.linux-x86_64/wheel

[...]
```

```
creating build/bdist.linux-x86_64/wheel/ceilometer-2014.1.a6.g772e1a7.
   dist-info/WHEEL
running upload
Submitting /home/jd/Source/ceilometer/dist/ceilometer-2014.1.a6.g772e1a7-
   py27-none-any.whl to https://testpypi.python.org/pypi
Server response (200): OK
```

现在应该可以在 PyPi 预付费服务器中搜索你的包，确认是否上传成功了。也可以试着用 pip 安装它，可以通过-i 选项指定测试服务器：

```
$ pip install -i https://testpypi.python.org/pypi ceilometer
```

如果一切就绪，就可以继续下一步了：上传项目到 PyPI 主服务器。只需要将身份信息和服务器的具体信息添加到你的~/.pypirc 文件中：

```
[distutils]
index-servers =
   pypi
   testpypi

[pypi]
username = <your username>
password = <your password>

[testpypi]
repository = https://testpypi.python.org/pypi
username = <your username>
password = <your password>
```

分别运行 register 和 upload 并配合参数-r pypi 就能正确地将你的包上传到 PyPI 服务器了。

4.6 Nick Coghlan 访谈

Nick 是 Red Hat 公司的 Python 核心开发人员。他已经写过几个 PEP 提案，包括 PEP 436（http://www.python.org/dev/peps/pep-0426/）（Metadata for Python Software Packages 2.0），他

是这个提案的 BDFL[①]代表。

Python 的打包方案（distutils、setuptools、distutils2、distlib、bento、pbr 等）的数量真是令人印象深刻。根据你的观点，是什么（可能的历史性）原因造成这样的分裂和分化？

简单地回答就是软件的发布、分发和集成是很复杂的问题，所以有很大的空间共存多个针对不同使用场景的解决方案。详细的答案请参考 Python 打包用户指南（https://python-packaging-user-guide.readthedocs.org/en/latest/future.html#how-we-got-here)。我在最近关于这个问题的表述中已经指出，问题的主要原因之一是年代，上述这些工具大多产生于软件分发技术的不同时代。

setuptools 如今已经是 Python 分发工具的事实标准。你觉得有什么问题是用户在使用（或者不用）它时需要注意的吗？

setuptools 作为构建系统是相当不错的，尤其是对纯 Python 项目或者只有简单的 C 扩展的项目。它还为插件注册和良好的跨平台脚本生成提供了强大的系统支撑。

尽管如此，`pkg_resources` 中的多版本支持要想用好仍然显得有点儿刁钻古怪。除非有非常充分的理由要在同一个环境中包含互相冲突的版本，否则这样只使用 virtualenv 或 zc.buildout 会更容易。

PEP 426 定义了 Python 包的一种新格式，但它仍然非常新而且尚未批准。它的进展还

① “Benevolent Dictator For Life”是 Python 作者 Guido van Rossum 给的称号。

顺利吗？最开始的动机是什么？你觉得它能解决当前的问题吗？

PEP 426 最早源于 Wheel 格式定义的一部分，但是 Daniel Holth 最终意识到 Wheel 能够同已有的 setuptools 定义的格式一起工作。因此 PEP 426 是已有的 setuptools 元数据和一些来自 distutils2 以及其他打包系统（如 RPM 和 npm）的想法的融合，并且解决了现有工具中遇到的一些问题（如清楚地隔离不同类型的依赖）。

如果 PEP 426 被接受的话，你希望看到出现什么样的工具来充分利用 PEP 426？

主要的好处是 PyPI 将能够通过 REST API 提供完整的元数据访问，并且（希望能）具有根据上传的元数据自动生成策略兼容的分发包的能力。

Wheel 格式非常新，还没有被广泛使用，但看上去很有前途。是什么原因造成它还不是标准库的一部分呢？是否已经有计划包含它？

事实证明，打包标准并不太适合放到标准库里：它的演进太慢，并且对后面版本的扩展并不能用在 Python 的早期版本中。所以，在今年早些时候的 Python 语言峰会上，我们调整了 PEP 流程，以便用 distutils-sig 管理打包与相关 PEP 的完整审批流程。python-dev 将只参与那些直接涉及修改 CPyhon 的提案（如 pip 引导）。

根据你的设想，未来怎样的发展会推动开发人员去构建和分发 Wheel 格式的包？

`pip` 正在接受其成为 Egg 格式的备选方案，允许构建的本地缓存以便快速创建虚拟环境，并且 PyPI 允许上传针对 Windows 和 Mac OS X 平台的 Wheel 归档文件。在它适用于在 Linux 之前，我们仍然有一些问题要解决。

4.7 扩展点

你可能已经在并不了解 setuptools 的情况下使用过它的入口点。如果还没决定用 setuptools 为你的软件提供 `setup.py` 文件，这里有一些功能的介绍也许能帮你做决定。

使用 setuptools 分发软件包括重要的元数据描述，如需要的依赖以及与这个主题更相关的“入口点”的列表。这些入口点能够被其他 Python 程序用来动态发现包所提供的功能。

在下面几节中，我们将讨论如何利用入口点为软件添加扩展能力。

4.7.1　可视化的入口点

要看到一个包中可用的入口点的最简单的方法就是使用一个叫 `entry_point_inspector`（https://pypi.python.org/pypi/entry_point_inspector）的包。

安装后，它提供了名为 `epi` 的命令，可以从终端运行并能交互式地发现某个安装包的入口点，如示例 4.4 所示。

示例 4.4　`epi group list` 的运行结果

```
+--------------------------+
| Name                     |
+--------------------------+
| console_scripts          |
| distutils.commands       |
| distutils.setup_keywords |
| egg_info.writers         |
| epi.commands             |
| flake8.extension         |
| setuptools.file_finders  |
| setuptools.installation  |
+--------------------------+
```

示例 4.4 显示系统有很多不同地包都提供了入口点。你可能注意到，这个列表包含 `console_scripts`（将在 4.7.4 节中讨论）。

示例 4.5　`epi group show console_scripts` 的运行结果

```
+----------+----------+--------+--------------+-------+
| Name     | Module   | Member | Distribution | Error |
+----------+----------+--------+--------------+-------+
| coverage | coverage | main   | coverage 3.4 |       |
+----------+----------+--------+--------------+-------+
```

示例 4.5 显示了一个名为 coverage 的入口点，并引用了 coverage 模块的成员的 `main`。这个入口点是由包 coverage 3.4 提供的。可以使用 `kepi ep show` 获得更多信息。

示例 4.6　`kepi ep show console_scripts coverage` 的运行结果

```
+-------------+---------------------------------+
```

```
  | Field        | Value                            |
  +--------------+----------------------------------+
  | Module       | coverage                         |
  | Member       | main                             |
  | Distribution | coverage 3.4                     |
  | Path         | /usr/lib/python2.7/dist-packages |
  | Error        |                                  |
  +--------------+----------------------------------+
```

这里所用的工具只是很薄的一层，它建立在更复杂的能够发现任何 Python 库或程序的入口点的 Python 库之上。入口点有许多不同的用处，如对控制台脚本和动态代码发现都很有用，这些将在下面几节介绍。

4.7.2 使用控制台脚本

开发 Python 应用程序时，通常要提供一个可启动的程序，也就是最终用户实际可以运行的 Python 脚本。这个程序需要被安装在某个包含在系统路径中的目录里。

大多数项目都会有下面这样几行代码：

```
#!/usr/bin/python
import sys
import mysoftware

mysoftware.SomeClass(sys.argv).run()
```

这实际上是一个理想情况下的场景：许多项目在系统路径中会有一个非常长的脚本安装。但使用这样的脚本有一些主要的问题。

- 没办法知道 Python 解释器的位置和版本。
- 安装的二进制代码不能被其他软件或单元测试导入。
- 很难确定安装在哪里。
- 如何以可移植的方式进行安装并不明确（如是 Unix 还是 Windows）。

setuptools 有一个功能可以帮助我们解决这些问题，即 console_scripts。console_scripts 是一个入口点，能够用来帮助 setuptools 安装一个很小的程序到系统目录中，并通过它调用应用程序中某个模块的特定函数。

设想一个 foobar 程序，它由客户端和服务器端两部分组成。这两部分各自有自己独立的模块——`foobar.client` 和 `foobar.server`。

foobar/client.py

```
def main():
    print("Client started")
```

foobar/server.py

```
def main():
    print("Server started")
```

当然，这个程序做不了什么——客户端和服务器端甚至不能彼此通信。但对于我们这个例子的目的来说，只需要在它们成功启动之后能输出消息即可。

接下来可以在根目录中添加下面的 `setup.py` 文件。

setup.py

```
From setuptools import setup

setup(
    name="foobar",
    version="1",
    description="Foo!",
    author="Julien Danjou",
    author_email="julien@danjou.info",
    packages=["foobar"],
    entry_points={
        "console_scripts": [
            "foobard = foobar.server:main",
            "foobar = foobar.client:main",
        ],
     },
)
```

使用格式 `package.subpackage:function` 可以定义自己的入口点。

当运行 `python setup.py install` 时，setuptools 会创建示例 4.7 所示的脚本。

示例 4.7 setuptools 生成的控制台脚本

```
#!/usr/bin/python
# EASY-INSTALL-ENTRY-SCRIPT: 'foobar==1','console_scripts','foobar'
__requires__ = 'foobar==1'
import sys
From pkg_resources import load_entry_point

if __name__ == '__main__':
    sys.exit(
        load_entry_point('foobar==1', 'console_scripts', 'foobar')()
    )
```

这段代码会扫描 foobar 包的入口点并从 console_scripts 目录中抽取 foobar 键，从而定位并运行相应的函数。

使用这一技术能够保证代码在 Python 包内，并能够被其他应用程序导入（或测试）。

提示

如果在 setuptools 之上使用 pbr，那么生成的脚本会比通过 setuptools 默认创建的要简单（因此也更快），因为它会调用写在入口点中的函数而无需在运行时动态扫描入口点列表。

4.7.3 使用插件和驱动程序

通过入口点可以很容易得发现和动态加载其他包部署的代码。可以使用 **pkg_resources**（http://pythonhosted.org/distribute/pkg_resources.html）从自己的 Python 程序中发现和加载入口点文件。（你可能已经注意到，这与示例 4.7 中 setuptools 创建的控制台脚本所使用的是同一个包。）

在本节中，我们将创建一个 cron 风格的守护进程，它通过注册一个入口点到 pytimed 组中即可允许任何 Python 程序注册一个每隔几秒钟运行一次的命令。该入口点指向的属性应该是一个返回 number_of_seconds 和 callable 的对象。

下面是一个使用 pkg_resources 发现入口点的 pycrond 实现。

pytimed.py

```
import pkg_resources
```

```
import time

def main():
    seconds_passed = 0
    while True:
        for entry_point in pkg_resources.iter_entry_points('pytimed'):
        try:
            seconds, callable = entry_point.load()()
        except:
            # Ignore failure
            pass
        else:
            if seconds_passed % seconds == 0:
                callable()
    time.sleep(1)
    seconds_passed += 1
```

这是一个非常简单而朴素的实现，但对我们的例子来说足够了。现在可以写另一个 Python 程序，需要周期性地调用它的一个函数。

hello.py

```
def print_hello():
    print("Hello, world!")
def say_hello():
    return 2, print_hello
```

使用合适的入口点注册这个函数。

setup.py

```
from setuptools import setup

setup(
    name="hello",
    version="1",
    packages=["hello"],
    entry_points={
        "pytimed": [
            "hello = hello:say_hello",
        ],
```

```
    },)
```

现在如果运行 pytimed 脚本，将会看到在屏幕上每两秒钟打印一次“Hello, world!”，如示例 4.8 示例。

示例 4.8 运行 pytimed

```
% python3
Python 3.3.2+ (default, Aug  4 2013, 15:50:24)
[GCC 4.8.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pytimed
>>> pytimed.main()
Hello, world!
Hello, world!
Hello, world!
```

这一机制提供了巨大的可能性：它可以用来构建驱动系统、钩子系统以及简单而通用的扩展。在每一个程序中手动实现这一机制是非常繁琐的，不过幸运的是，已经有 Python 库可以处理这部分无聊的工作。

stevedore（https://pypi.python.org/pypi/stevedore）基于我们在前面例子中展示的机制提供了对动态插件的支持。在这个例子中我们的用例并不复杂，但是仍然可以用 **stevedore** 稍做简化。

pytimed_stevedore.py

```
from stevedore.extension import ExtensionManager import time

def main():
    seconds_passed = 0
    while True:
        for extension in ExtensionManager('pytimed', invoke_on_load=True):
        try:
            seconds, callable = extension.obj
        except:
            # Ignore failure
            pass
        else:
            if seconds_passed % seconds == 0:
```

```
                callable()
        time.sleep(1)
        seconds_passed += 1
```

我们的例子仍然非常简单，但是如果看了 stevedore 文档就会发现，`ExtensionManager` 有很多用来处理不同场景的子类，例如基于名字或者函数运行结果加载特定的扩展。

第 5 章 虚拟环境

在处理 Python 应用程序时，经常需要部署、使用和测试你的应用程序。但由于外部依赖的问题，做起来实际上很麻烦。有许多原因会导致在你的操作系统上部署或运行应用程序失败，例如：

- 系统中没有需要的库；
- 系统中没有需要的库的正确版本；
- 对两个不同的应用程序可能需要同一个库的两个不同版本。

这会发生在应用部署的时候，或者稍后运行的时候。而通过系统管理器升级已安装的 Python 库则很可能在毫无征兆的情况下导致你的应用程序无法运行。

这一问题的解决方案是针对每个应用程序使用独立的库目录，同时包含自己的依赖。然后使用这个目录——而不是系统安装的那个目录——加载所需的 Python 模块。

工具 `virtualenv` 可以自动为你处理这些目录。安装之后，只需在运行时将目标目录作为它的参数传入即可。

```
$ virtualenv myvenv
Using base prefix '/usr'
New python executable in myvenv/bin/python3
Also creating executable in myvenv/bin/python
Installing Setuptools.....................done.
Installing Pip...............................done.
```

运行后，`virtualenv` 会创建 `lib/pythonX.Y` 目录并利用它安装 setuptools 和 pip，它们是后续安装其他 Python 包所必需的。

可以通过对 `activate` 执行 `source` 命令来激活这个虚拟环境

```
$ source myvenv/bin/activate
```

一旦这么做，shell 的提示符会加上虚拟环境的名字作为前缀。此时调用 python 会执行被复制到虚拟环境中的 Python。可以通过读取 sys.path 环境变量来验证，它将会把虚拟环境目录作为首选组件。

可以通过调用 deactivate 命令随时停止并退出虚拟环境：

```
$ deactivate
```

就这么简单。

如果只想使用在虚拟环境中安装的 Python 一次的话，不用运行 acivate。直接调用虚拟环境中的 python 二进制文件也可以正常工作：

```
$ myvenv/bin/python
```

到目前为止，尽管已经进入激活的虚拟环境中，但还不能访问系统中安装以及在系统中可用的任何模块。这没问题，但我们可能需要安装它们。要做到这一点，只需使用标准的 pip 命令，它可以将包安装在正确的位置而不会对现有系统做任何修改：

```
$ source myvenv/bin/activate
(myvenv) $ pip install six
Downloading/unpacking six
  Downloading six-1.4.1.tar.gz
  Running setup.py egg_info for package six

Installing collected packages: six
  Running setup.py install for six

Successfully installed six
Cleaning up...
```

好了，我们能够在这个虚拟环境中安装所有需要的库，然后从虚拟环境运行我们的应用程序，而不会对当前系统产生任何影响。那么，接下来很自然就会想到要基于依赖的列表写脚本来自动安装虚拟环境，如示例 5.1 所示。

示例 5.1　自动化的虚拟环境创建

```
virtualenv myappvenv
source myappvenv/bin/activate
pip install -r requirements.txt
deactivate
```

在某些特定场景下，仍然需要访问系统中安装的包。那么可以通过在创建虚拟环境时向 virtualen 命令传入--system-site-packages 标志来实现。

你可能已经猜到，虚拟环境对自动运行单元测试集非常有用。这是一个非常通用的模式，它是如此的通用，以至于已经有名为 tox 的工具来专门解决这一问题（详见 6.7 节）。

最近，PEP 405（http://www.python.org/dev/peps/pep-0405/）定义的虚拟环境机制已经被 Python 3.3 接受。也就是说，虚拟环境的使用如此流行以至于如今它已经成为 Python 标准库的一部分。

venv 模块是 Python 3.3 及以上版本的一部分，可以操作虚拟环境而无需使用 virtualenv 包或其他包。可以通过 Python 的-m 标志来加载模块：

```
$ python3.3 -m venv
usage: venv [-h] [--system-site-packages] [--symlinks] [--clear] [--upgrade]
            ENV_DIR [ENV_DIR ...]
venv: error: the following arguments are required: ENV_DIR
```

构建虚拟环境现在变得非常简单：

```
$ python3.3 -m venv myvenv
```

在 myvenv 内部，可以找到当前环境的名为 pyvenv.cfg 的配置文件。默认情况下它并不包含很多配置项。其中的 include-system-site-package 作用和前面介绍过的 virtualenv 的参数--system-site-packages 作用相同。

激活虚拟环境的机制同前面描述的一样，通过执行 source 命令激活脚本：

```
$ source myvenv/bin/activate
(myvenv) $
```

同样，可以调用 deactivate 退出虚拟环境。

venv 模块的缺点就是它不会默认安装 setuptools 和 pip。因此我们只能自己引导环境（如示例 5.2 所示），而不像 vitualenv 那样都帮我们做好。

示例 5.2　引导 venv 环境

```
(myvenv) $ wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/
  ez_setup.py -O - | python
-2013-09-02 22:26:07--  https://bitbucket.org/pypa/setuptools/raw
  /bootstrap/ez_setup.py
```

```
Resolving bitbucket.org (bitbucket.org)... 131.103.20.168, 131.103.20.167
Connecting to bitbucket.org (bitbucket.org)|131.103.20.168|:443...
  connected.
HTTP request sent, awaiting response... 200 OK
Length: 11835 (12K) [text/plain]
Saving to: 'STDOUT'

100%[=======================================>] 11,835     --.-K/s  in 0s

2013-09-02 22:26:08 (184 MB/s) - written to stdout [11835/11835]

Downloading https://pypi.python.org/packages/source/s/setuptools/
  setuptools-1.1.tar.gz
Extracting in /tmp/tmp228fqm
Now working in /tmp/tmp228fqm/setuptools-1.1
Installing Setuptools
running install
running bdist_egg
running egg_info
writing dependency_links to setuptools.egg-i
[...]
Adding setuptools 1.1 to easy-install.pth file
Installing easy_install script to /home/jd/myvenv/bin
Installing easy_install-3.3 script to /home/jd/myvenv/bin

Installed /home/jd/myvenv/lib/python3.3/site-packages/
  setuptools-1.1-py3.3.egg
Processing dependencies for setuptools==1.1
Finished processing dependencies for setuptools==1.1
```

接下来可以通过 `easy_install` 安装 pip：

```
(myvenv) $ easy_install pip
Searching for pip
Reading https://pypi.python.org/simple/pip/
Best match: pip 1.4.1
Downloading https://pypi.python.org/packages/source/p/pip/pip-1.4.1.tar.
  gz#md5=6afbb46aeb48abac658d4df742bff714
Processing pip-1.4.1.tar.gz
Writing /tmp/easy_install-hxo3b0/pip-1.4.1/setup.cfg
```

```
Running pip-1.4.1/setup.py -q bdist_egg --dist-dir /tmp/easy_install-hxo3b0
  /pip-1.4.1/egg-dist-tmp-efgi80
warning: no files found matching '*.html' under directory 'docs'
warning: no previously-included files matching '*.rst' found under directory
  'docs/_build'
no previously-included directories found matching 'docs/_build/_sources'
Adding pip 1.4.1 to easy-install.pth file
Installing pip script to /home/jd/myvenv/bin
Installing pip-3.3 script to /home/jd/myvenv/bin

Installed /home/jd/myvenv/lib/python3.3/site-packages/pip-1.4.1-py3.3.egg
Processing dependencies for pip
Finished processing dependencies for pip
```

接下来就可以通过 pip 安装任何其他所需的包了。

所以尽管 Python 3.3 默认包含了 venv 模块，但是必须承认它有缺点，就是它不会默认做好一些你期望的工作。虽然写个小工具利用 venv 模拟 virtualenv 的默认行为并不难，但是，如果不是只针对 Python 3.3 及以上版本，那么实在没必要这么做。另外，pip 引导代码已经并入 Python 3.4 中，这意味着引导问题在最近的 Python 版本中已经解决了。

不管怎样，大多数程序都要同时支持 Python 2 和 Python 3，所以完全依赖 venv 不是最好的选择。最好的选择仍然是基于 virtualenv。考虑它们的工作方式都一样，所以这应该不是什么问题。

第 6 章 单元测试

重磅消息！现在居然还有人在自己的项目中没有测试策略。本书的目的不是试图说服你开始单元测试。如果你想被说服的话，建议你从了解测试驱动开发的好处开始。编写未经测试的代码是毫无用处的，因为没有办法能最终证明它是可以工作的。

本章将介绍可以用来构建良好测试集的 Python 工具。我们还将讨论如何利用这些工具增强你的软件，让软件更加健壮，避免引入回归问题。

6.1 基础知识

和你了解的也许不同，在 Python 中编写和运行单元测试是非常简单的。它不但不会干扰或者破坏现有程序，还会极大地帮助你和其他开发人员维护软件。

测试应该保存在应用程序或库的 `tests` 子模块中。这可以使测试代码随模块一同分发，以便只要软件被安装了，它们就可以被任何其他人运行或重用而无需使用源代码包。同时，这也可以避免这些测试代码被错误地安装在顶层 `tests` 模块。

通常比较简单的方式是采用模块树的层次结构作为测试树的层级结构。也就是说，覆盖代码 `mylib/foobar.py` 的测试应该存储在 `mylib/tests/test_foobar.py` 中，这样在查找与某个特定文件相关联的测试时会比较方便，如示例 6.1 所示。

示例 6.1 `test_true.py` 中的一个真实的简单测试

```
def test_true():
    assert True
```

这是能够写出来的最简单的单元测试。要运行它，只需加载 `test_true.py` 文件并运行其中定义的 `test_true` 函数。

显然，对于你的所有测试文件都这么做肯定是太痛苦了。这就是 nose（https://nose.readthedocs.org/en/latest/）这个包要解决的——安装之后，它将提供 nosetests 命令，该命令会加载所有以 test_开头的文件，然后执行其中所有以 test_开头的函数。

因此，针对我们的源代码树中的 test_true.py 文件运行 nosetests 将得到以下结果：

```
$ nosetests -v
test_true.test_true ... ok

----------------------------------------------------------
Ran 1 test in 0.003s

OK
```

但是，一旦测试失败，输出就会相应改变，以体现这次失败，包括完整的跟踪回溯。

```
% nosetests -v
test_true.test_true ... ok
test_true.test_false ... FAIL

==========================================================
FAIL: test_true.test_false

Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/nose/case.py", line 197, in
    runTest
    self.test(*self.arg)
  File "/home/jd/test_true.py", line 5, in test_false
    assert False
AssertionError
----------------------------------------------------------
Ran 2 tests in 0.003s

FAILED (failures=1)
```

一旦有 AssertionError 异常抛出，测试就失败了；一旦 assert 的参数被判断为某些假值（False、None、0 等），它就会抛出 AssertionError 异常。如果有其他异常抛出，测试也会出错退出。

很简单，对吗？这种方法尽管简单，但却在很多小的项目中广泛使用且工作良好。除了

nose，它们不需要其他工具或库，而且只依赖 assert 就足够了。

不过，在需要编写更复杂的测试时，只使用 assert 会让人很抓狂。设想一下下面这个测试：

```
def test_key():
    a = ['a', 'b']
    b = ['b']
    assert a == b
```

当运行 nosetests 时，它会给出如下输出：

```
$ nosetests -v
test_complicated.test_key ... FAIL

======================================================
FAIL: test_complicated.test_key
Traceback (most recent call last):
  File "/usr/lib/python2.7/dist-packages/nose/case.py", line 197, in
     runTest
    self.test(*self.arg)
  File "/home/jd/test_complicated.py", line 4, in test_key
    assert a == b
AssertionError

----------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

显然，因为 a 和 b 不同，所以测试不能通过。但是，它们到底有何不同呢？assert 没有给出这一信息，而只是声称此断言是错误的——这是没什么用的。

而且，用这种基本的无框架方式实现一些高级的测试（如忽略某个测试，或者在每个测试之前或之后执行某些操作）也会非常痛苦。

这用 unittest 就比较方便了。它提供了解决上述问题的工具，而且 unittest 是 Python 标准库的一部分。

警告

unittest 在 Python 2.7 中已经做了较大改进，如果正在支持 Python 的早期版本，那么

可能需要使用它的向后移植的名字 unittest2（https://pypi.python.org/pypi/unittest2/0.5.1）。如果需要支持 Python 2.6，可以使用下面的代码段在运行时为任何 Python 版本导入正确的模块：

```
try:
    import unittest2 as unittest
except ImportError:
    import unittest
```

如果用 unittest 重写前面的例子，看起来会是下面的样子：

```
import unittest

class TestKey(unittest.TestCase):
    def test_key(self):
        a = ['a', 'b']
        b = ['b']
        self.assertEqual(a, b)
```

如你所见，实现起来并没有更复杂。需要做的就只是创建一个继承自 unittest.TestCase 的类，并且写一个运行测试的方法。与使用 assert 不同，我们依赖 unittest.TestCase 类提供的一个方法，它提供了一个等价的测试器。运行时，其输出如下：

```
$ nosetests -v
test_key (test_complicated.TestKey) ... FAIL

======================================================
FAIL: test_key (test_complicated.TestKey)
Traceback (most recent call last):
  File "/home/jd/Source/python-book/test_complicated.py", line 7, in
     test_key
    self.assertEqual(a, b)
AssertionError: Lists differ: ['a', 'b'] != ['b']

First differing element 0:
a
b

First list contains 1 additional elements.
First extra element 1:
```

```
b

- ['a', 'b']
+ ['b']

----------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

如你所见，这个输出结果很有用。仍然有断言错误被抛出，而且测试仍然失败了，但至少我们获得了为什么测试会失败的真正信息，它可以帮我们解决这个问题。这就是写测试用例时**永远**不应该使用 assert 的原因。任何人试图 hack 你的代码并最终遇到某个测试失败时都会感谢你没有使用 assert，这同时也为他提供了调试信息。

unittest 提供了一组测试函数，可以用来特化测试，如 assertDictEqual、assertEqual、assertTrue、assertFalse、assertGreater、assertGreaterEqual、assertIn、assertIs、assertIsIntance、assertIsNon、assertualIsNot、assertIsNotNone、assertItemsEqual、assertLess、assertLessEqual、assertListEqual、assertMultiLineEqual、assertNotAlmostEqual、assertNotEqual、assertTupleEqual、assertRaises、assertRaisesRegexp、assertRegexpMatches 等。最好是通读一遍 pydoc unittest，以便全面了解。

也可以使用 fail(msg)方法有意让某个测试立刻失败。例如，已知代码的某个部分如果执行一定会抛出一个错误但没有特定的断言去检查时，这是很方便的，如示例 6.2 所示。

示例 6.2 让测试失败

```
import unittest

class TestFail(unittest.TestCase):
    def test_range(self):
        for x in range(5):
            if x > 4:
                self.fail("Range returned a too big value: %d" % x)
```

有时候，某个测试如果不能运行，忽略它是很有用的。例如，希望根据某个库的存在与

否有条件地运行某个测试。为此，可以抛出 **unitest.SkipTest** 异常。当该测试被执行时，它只是被简单地标注为已忽略。更便利的方法是使用 unittest.TestCase.skipTest() 而不是手工抛出这一异常，另外也可以使用 unittest.skip 装饰器，如示例 6.3 所示。

示例 6.3　忽略测试

```
import unittest

try:
    import mylib
except ImportError:
    mylib = None

class TestSkipped(unittest.TestCase):
    @unittest.skip("Do not run this")
    def test_fail(self):
        self.fail("This should not be run")

    @unittest.skipIf(mylib is None, "mylib is not available")
    def test_mylib(self):
        self.assertEqual(mylib.foobar(), 42)

    def test_skip_at_runtime(self):
        if True:
            self.skipTest("Finally I don't want to run it")
```

执行后，该测试文件会输出下列内容：

```
$ python -m unittest -v test_skip
test_fail (test_skip.TestSkipped) ... skipped 'Do not run this'
test_mylib (test_skip.TestSkipped) ... skipped 'mylib is not available'
test_skip_at_runtime (test_skip.TestSkipped) ... skipped "Finally I don't
  want to run it"

----------------------------------------------------------
Ran 3 tests in 0.000s

OK (skipped=3)
```

提示

在示例 6.3 中你可能已经注意到，unittest 模块提供了一种执行包含测试的 Python 模块的方式。它没有 nosetests 那么方便，因为它不会发现自己的测试文件，但它对于运行特定测试模块仍然是很有用的。

在许多场景中，需要在运行某个测试前后执行一组通用的操作。unittest 提供了两个特殊的方法 **setUp** 和 **tearDown**，它们会在类的每个测试方法调用前后执行一次，如示例 6.4 所示。

示例 6.4 使用 unittest 的 setUp 方法

```
import unittest

class TestMe(unittest.TestCase):
    def setUp(self):
        self.list = [1, 2, 3]

    def test_length(self):
        self.list.append(4)
        self.assertEqual(len(self.list), 4)

    def test_has_one(self):
        self.assertEqual(len(self.list), 3)
        self.assertIn(1, self.list)
```

在这个示例中，setUp 会在运行 test_length 和 test_has_one 之前被调用。它可以非常方便地创建在每个测试中要用到的对象，但你需要保证它们在运行每个测试之前，在干净的状态下被重建。这对于创建测试环境是非常有用的，经常被称为 fixture（参见 6.2 节）。

提示

使用 nosetests 时，经常会只想运行某个特定的测试。你可以选择要运行的测试作为参数，语法是 path.to.your.module:ClassOfYourTest.test_method。确保在模块路径和类名之前有一个冒号。也可以指定 path.to.your.module:ClassOfYourTest 来执行整个类，或者指定 path.to.your.module 来执行整个模块。

提示

通过同时运行多个测试可以加快速度。只需为 nosetests 调用加上--process=N 选项即可创建多个 nosetests 进程。不过，testrepository 是更好的选择（这会在 6.5 节中讨论）。

6.2 fixture

在单元测试中，fixture 表示“测试前创建，测试后销毁”的（辅助性）组件。比较好的方式是为它们构建一个特殊的组件，因为它们会在许多不同的地方被重用。例如，如果你需要一个对象来表示你的应用程序的配置状态，很可能你希望在每个测试前初始化它，并在测试结束后将其重置为默认值。对临时文件创建的依赖也需要该文件在测试开始前被创建，测试结束后被删除。

unittest 只为我们提供了已经提及的 setUp 和 tearDown 函数。不过，是有机制可以 hook 这两个函数的。**fixtures**（https://pypi.python.org/pypi/fixtures）Python 模块（并非标准库的一部分）提供了一种简单的创建 fixture 类和对象的机制，如 useFixture 方法。

fixtures 模块提供了一些内置的 fixture，如 fixtures.EnvironmentVariable，对于在 os.environ 中添加或修改变量很有用，并且变量会在测试退出后重置，如示例 6.5 所示。

示例 6.5　使用 fixtures.EnvironmentVariable

```
import fixtures
import os

class TestEnviron(fixtures.TestWithFixtures):
    def test_environ(self):
        fixture = self.useFixture(
            fixtures.EnvironmentVariable("FOOBAR", "42"))
        self.assertEqual(os.environ.get("FOOBAR"), "42")

    def test_environ_no_fixture(self):
        self.assertEqual(os.environ.get("FOOBAR"), None)
```

当你发现类似的通用模式时，最好创建一个 fixture，以便它可以被你的所有其他测试用

例重用。这极大地简化了逻辑，并且能准确地体现你在测试什么以及以何种方式测试。

注意

本节的示例代码之所以没有用 unittest.TestCase，是因为 fixtures.TestWithFixtures 继承自 unittest.TestCase。

6.3 模拟（mocking）

mock 对象即模拟对象，用来通过某种特殊和可控的方式模拟真实应用程序对象的行为。在创建精确地描述测试代码的状态的环境时，它们非常有用。

如果正在开发一个 HTTP 客户端，要想部署 HTTP 服务器并测试所有场景，令其返回所有可能值，几乎是不可能的（至少会非常复杂）。此外，测试所有失败场景也是极其困难的。

一种更简单的方式是创建一组根据这些特定场景进行建模的 mock 对象，并利用它们作为测试环境对代码进行测试。

Python 标准库中用来创建 mock 对象的库名为 **mock**（https://pypi.python.org/pypi/mock/1.0.1）。从 Python 3.3 开始，它被命名为 unit.mock，合并到 Python 标准库。因此可以使用下面的代码片段：

```
try:
    from unittest import mock
except ImportError:
    import mock
```

要保持 Python 3.3 和之前版本之间的向后兼容。

它使用起来也非常简单，如示例 6.6 所示。

示例 6.6 mock 的基本用法

```
>>> import mock
>>> m = mock.Mock()
>>> m.some_method.return_value = 42
>>> m.some_method()
42
>>> def print_hello():
...
```

```
print("hello world!")
...
>>> m.some_method.side_effect = print_hello
>>> m.some_method()
hello world!
>>> def print_hello():
... print("hello world!")
... return 43
...
>>> m.some_method.side_effect = print_hello
>>> m.some_method()
hello world!
43
>>> m.some_method.call_count
3
```

即使只使用这一组功能，也应该可以模拟许多内部对象以用于不同的数据场景中。

模拟使用动作/断言模式，也就是说一旦测试运行，必须确保模拟的动作被正确地执行，如示例 6.7 所示。

示例 6.7　确认方法调用

```
>>> import mock
>>> m = mock.Mock()
>>> m.some_method('foo', 'bar')
<Mock name='mock.some_method()' id='26144272'>
>>> m.some_method.assert_called_once_with('foo', 'bar')
>>> m.some_method.assert_called_once_with('foo', mock.ANY)
>>> m.some_method.assert_called_once_with('foo', 'baz')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/dist-packages/mock.py", line 846, in
      assert_called_once_with
    return self.assert_called_with(*args, **kwargs)
  File "/usr/lib/python2.7/dist-packages/mock.py", line 835, in
      assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: some_method('foo', 'baz')
Actual call: some_method('foo', 'bar')
```

显然，很容易传一个 mock 对象到代码的任何部分，并在其后检查代码是否按其期望的传入参数被调用。如果不知道该传入何种参数，可以使用 mock.ANY 作为参数值传入，它将会匹配传递给 mock 方法的任何参数。

有时可能需要来自外部模块的函数、方法或对象。mock 库为此提供了一组补丁函数。

示例 6.8 使用 mock.patch

```
>>> import mock
>>> import os
>>> def fake_os_unlink(path):
...     raise IOError("Testing!")
...
>>> with mock.patch('os.unlink', fake_os_unlink):
...    os.unlink('foobar')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in fake_os_unlink
IOError: Testing!
```

通过 mock.pach 方法，可以修改外部代码的任何部分，使其按照需要的方式对软件进行各种条件下的测试，如示例 6.9 所示。

示例 6.9 使用 mock.patch 测试一组行为

```
import requests
import unittest
import mock

class WhereIsPythonError(Exception):
    pass

def is_python_still_a_programming_language():
    try:
        r = requests.get("http://python.org")
    except IOError:
        pass
    else:
        if r.status_code == 200:
```

```
            return 'Python is a programming language' in r.content
    raise WhereIsPythonError("Something bad happened")

def get_fake_get(status_code, content):
    m = mock.Mock()
    m.status_code = status_code
    m.content = content
    def fake_get(url):
        return m
    return fake_get

def raise_get(url):
    raise IOError("Unable to fetch url %s" % url)

class TestPython(unittest.TestCase):
    @mock.patch('requests.get', get_fake_get(
        200, 'Python is a programming language for sure'))
    def test_python_is(self):
        self.assertTrue(is_python_still_a_programming_language())

    @mock.patch('requests.get', get_fake_get(
        200, 'Python is no more a programming language'))
    def test_python_is_not(self):
        self.assertFalse(is_python_still_a_programming_language())

    @mock.patch('requests.get', get_fake_get(
        404, 'Whatever'))
    def test_bad_status_code(self):
        self.assertRaises(WhereIsPythonError,
                          is_python_still_a_programming_language)

    @mock.patch('requests.get', raise_get)
    def test_ioerror(self):
        self.assertRaises(WhereIsPythonError,
                          is_python_still_a_programming_language)
```

示例 6.9 使用了 mock.patch 的装饰器版本，这并不改变它的行为，但当需要在整个测试函数的上下文内使用模拟时这会更方便。

使用模拟可以很方便地模拟任何问题，如 Web 服务器返回 404 错误，或者发生网络问题。我们可以确定代码返回的是正确的值，或在每种情况下抛出正确的异常，总之确保代码总是按照预期行事。

6.4 场景测试

在进行单元测试时，对某个对象的不同版本运行一组测试是较常见的需求。你也可能想对一组不同的对象运行同一个错误处理测试去触发这个错误，又或者想对不同的驱动执行整个测试集。

最后一种情况在 OpenStack Ceilometer[①]（https://launchpad.net/ceilometer）项目中被大量使用。Ceilometer 中提供了一个调用存储 API 的抽象类。任何驱动都可以实现这个抽象类，并将自己注册成为一个驱动。Ceilometer 可以按需要加载被配置的存储驱动，并且利用实现的存储 API 保存和提取数据。这种情况下就需要对每个实现了存储 API 的驱动调用一类单元测试，以确保它们按照调用者的期望执行。

实现这一点的一种自然方式是使用混入类（mixin class）：一方面你将拥有一个包含单元测试的类，另一方面这个类还会包含对特定驱动用法的设置。

```
import unittest

class MongoDBBaseTest(unittest.TestCase):
    def setUp(self):
        self.connection = connect_to_mongodb()

class MySQLBaseTest(unittest.TestCase):
    def setUp(self):
        self.connection = connect_to_mysql()

class TestDatabase(unittest.TestCase):
    def test_connected(self):
        self.assertTrue(self.connection.is_connected())

class TestMongoDB(TestDatabase, MongoDBBaseTest):
```

① 作者是 OpenStack 中监控项目 Ceilometer 的前项目技术主管（Project Technical Lead）。——译者注

```
    pass

class TestMySQL(TestDatabase, MySQLBaseTest):
    pass
```

然而，从长期维护的角度看，这种方法的实用性和可扩展性都不好。

更好的技术是有的，可以使用 **testscenarios** 包（https://pypi.python.org/pypi/testscenarios）。它提供了一种简单的方式针对一组实时生成的不同场景运行类测试。这里使用 testscenarios 重写了示例 6.9 的部分代码来说明 6.3 节中介绍过的模拟，具体见示例 6.10。

示例 6.10　testscenarios 的基本用法

```
import mock
import requests
import testscenarios

class WhereIsPythonError(Exception):
    pass

def is_python_still_a_programming_language():
    r = requests.get("http://python.org")
    if r.status_code == 200:
        return 'Python is a programming language' in r.content
    raise WhereIsPythonError("Something bad happened")

def get_fake_get(status_code, content):
    m = mock.Mock()
    m.status_code = status_code
    m.content = content
    def fake_get(url):
        return m
    return fake_get

class TestPythonErrorCode(testscenarios.TestWithScenarios):
    scenarios = [
        ('Not found', dict(status=404)),
        ('Client error', dict(status=400)),
        ('Server error', dict(status=500)),
    ]
```

```
    def test_python_status_code_handling(self):
        with mock.patch('requests.get',
                        get_fake_get(
                            self.status,
                            'Python is a programming language for sure')):
            self.assertRaises(WhereIsPythonError,
                              is_python_still_a_programming_language)
```

尽管看上去只定义了一个测试，但是 testscenarios 会运行这个测试三次，因为这里定义了三个场景。

```
% python -m unittest -v test_scenario
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ... ok
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ... ok
test_python_status_code_handling (test_scenario.TestPythonErrorCode) ... ok

----------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

如上所示，为构建一个场景列表，我们需要的只是一个元组列表，其将场景名称作为第一个参数，并将针对此场景的属性字典作为第二个参数。

很容易联想到另一种使用方式：可以实例化一个特定的驱动并针对它运行这个类的所有测试，而不是为每个测试存储一个单独的值作为属性。具体如示例 6.11 所示。

示例 6.11 使用 testscenarios 测试驱动

```
import testscenarios
From myapp import storage

class TestPythonErrorCode(testscenarios.TestWithScenarios):
    scenarios = [
        ('MongoDB', dict(driver=storage.MongoDBStorage())),
        ('SQL', dict(driver=storage.SQLStorage())),
        ('File', dict(driver=storage.FileStorage())),
    ]
```

```
    def test_storage(self):
        self.assertTrue(self.driver.store({'foo': 'bar'}))

    def test_fetch(self):
        self.assertEqual(self.driver.fetch('foo'), 'bar')
```

注意

这里之所以不需要使用前面示例中使用的基类 unittest.TestCase，是因为 test-scenarios.TestWithScenarios 继承自 unittest.TestCase。

6.5　测试序列与并行

在执行大量测试时，按它们被运行的情况进行分析是很有用的。类似 nosetests 这样的工具只是将结果输出到 stdout，即标准输出，但这对测试结果的解析或分析并不方便。

subunit（https://pypi.python.org/pypi/python-subunit）是用来为测试结果提供流协议（streaming protocol）的一个 Python 模块。它支持很多有意思的功能，如聚合测试结果[①]或者对测试的运行进行记录或归档等。

使用 subunit 运行测试非常简单：

```
$ python -m subunit.run test_scenario
```

这条命令的输出是二进制数据，所以除非有能力直接阅读 subunit 协议，否则在这里直接再现它的输出结果实在是没什么意义。不过，subunit 还支持一组将其二进制流转换为其他易读格式的工具，如示例 6.12 所示。

示例 6.12　使用 subunit2pyunit

```
$ python -m subunit.run test_scenario | subunit2pyunit
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not
  found)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not
  found) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client
  error)
```

① 甚至可以支持来自不同源程序或语言的测试结果。

```
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client
  error) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server
  error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server
  error) ... ok

----------------------------------------------------------
Ran 3 tests in 0.061s

OK
```

这样的结果就容易理解了。你应该可以认出这个关于场景测试的测试集来自 6.4 节。其他值得一提的工具还有 subunit2csv、subunit2gtk 和 subunit2junitxml。

subunit 还可以通过传入 discover 参数支持自动发现哪个测试要运行。

```
$ python -m subunit.run discover | subunit2pyunit
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not
  found)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Not
  found) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client
  error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Client
  error) ... ok
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server
  error)
test_scenario.TestPythonErrorCode.test_python_status_code_handling(Server
  error) ... ok

----------------------------------------------------------
Ran 3 tests in 0.061s

OK
```

也可以通过传入参数--list 只列出测试但不运行。要查看这一结果，可以使用 subunit-ls。

```
$ python -m subunit.run discover --list | subunit-ls --exists
test_request.TestPython.test_bad_status_code
```

```
test_request.TestPython.test_ioerror
test_request.TestPython.test_python_is
test_request.TestPython.test_python_is_not
test_scenario.TestPythonErrorCode.test_python_status_code_handling
```

提示

可以使用--load-list 选项指定要运行的测试的清单而不是运行所有的测试。

在大型应用程序中，测试用例的数量可能会多到难以应付，因此让程序处理测试结果序列是非常有用的。**testrepository** 包（https://pypi.python.org/pypi/testrepository）目的就是解决这一问题，它提供了 testr 程序，可以用来处理要运行的测试数据库。

```
$testr init
$ touch .testr.conf
% python -m subunit.run test_scenario | testr load
Ran 4 tests in 0.001s
PASSED (id=0)
$ testr failing
PASSED (id=0)
$ testr last
Ran 3 tests in 0.001s
PASSED (id=0)
$ testr slowest
Test id                                        Runtime (s)
---------------------------------------------  -----------
test_python_status_code_handling(Not found)    0.000
test_python_status_code_handling(Server error) 0.000
test_python_status_code_handling(Client error) 0.000
$ testr stats
runs=1
```

一旦 subunit 的测试流被运行并加载到 testrepository，接下来就很容易使用 testr 命令来操作了。

显然，每次手工处理要运行的测试是很烦人的。因此，应该“教会”testr 如何执行要运行的测试，以便它可以自己去加载测试结果。这可以通过编辑项目的根目录中的.testr.conf 文件（见示例 6.13）来实现。

示例 6.13 .testr.conf 文件

```
[DEFAULT]
test_command=python -m subunit.run discover . $LISTOPT $IDOPTION ❶
test_id_option=--load-list $IDFILE ❷
test_list_option=--list ❸
```

❶ 执行 testr run 时要运行的命令。

❷ 加载测试列表要运行的命令。

❸ 列出测试要运行的命令。

第一行的 test_command 是最关键的。现在只需要运行 testr run 就可以将测试加载到 testrepository 中并执行。

注意

如果习惯用 nosetests，testr run 现在是等效的命令。

另外两个选项可以支持测试的并行运行。通过给 testr run 加上--prallel 选项即可轻松实现，如示例 6.14 所示。并行运行测试可以极大地加速测试过程。

示例 6.14 运行 testr run --parallel

```
$ testr run --parallel
running=python -m subunit.run discover .  --list
running=python -m subunit.run discover .  --load-list /tmp/tmpiMq5Q1
running=python -m subunit.run discover .  --load-list /tmp/tmp7hYEkP
running=python -m subunit.run discover .  --load-list /tmp/tmpP_9zBc
running=python -m subunit.run discover .  --load-list /tmp/tmpTejc5J
Ran 26 (+10) tests in 0.029s (-0.001s)
PASSED (id=7, skips=3)
```

在后台，testr 运行测试列出操作，然后将测试列表分成几个子列表，并分别创建 Python 进程运行测试的每个子列表。默认情况下，子列表的数量与当前使用的机器的 CPU 数目相等。可以通过加入--concurrency 标志设置进程的数目。

```
$ testr run --parallel --concurrency=2
```

可以想象，类似 subunit 和 testrepository 这样的工具将为测试效率的提升带来更多可能，而本节只是一个大概介绍。熟悉这些工具是非常值得的，因为测试会极大地影响你将要开发和发布的软件的质量。利用这些有力的工具能够节省很多时间。

testrepository 也可以同 setuptools 集成，并且为其部署 `testr` 命令。这使得与基于 `setup.py` 工作流的集成更加容易，例如，可以围绕 `setup.py` 记录整个项目。`setup.py testr` 命令可以接受一些选项，如`--testr-args`（通过它可以为 `testr` 加入更多选项）或者`--coverage`（这将在下一节介绍）。

6.6　测试覆盖

测试覆盖是完善单元测试的工具。它通过代码分析工具和跟踪钩子来判断代码的哪些部分被执行了。在单元测试期间使用时，它可以用来展示代码的哪些部分被测试所覆盖而哪些没有。

编写测试当然有用，但是知道代码的哪些部分没有被测试到才是关键所在。

显然，要做的第一件事就是在系统中安装 Python 的 **`coverage`** 模块（https://pypi.python.org/pypi/coverage）。安装之后就可以通过 shell 使用 `coverage` 程序[①]。

单独使用 `coverage` 非常简单且有用，它可以指出程序的哪些部分从来没有被运行过，以及哪些可能是“僵尸代码”。此外，在单元测试中使用的好处也显而易见，可以知道代码的哪些部分没有被测试过。前面谈到的测试工具都可以与 `coverage` 集成。

使用 nose 时，只需要加入很少的选项就可以生成一份不错的代码覆盖报告，如示例 6.15 所示。

示例 6.15　使用 `nosetests --with-coverage`

```
$ nosetests --cover-package=ceilometer --with-coverage tests/test_pipeline.py
.........................................
Name                             Stmts   Miss  Cover   Missing
ceilometer                           0      0   100%
ceilometer.pipeline                152     20    87%   49, 59, 113,
  127-128, 188-192, 275-280, 350-362
ceilometer.publisher                12      3    75%   32-34
ceilometer.sample                   31      4    87%   81-84
ceilometer.transformer              15      3    80%   26-32, 35
```

① 如果通过操作系统的软件安装程序进行安装的话，命令名也可能是 `python-coverage`。例如，Debian 系统中就叫 `python-coverage`。

```
ceilometer.transformer.accumulator  17     0   100%
ceilometer.transformer.conversions  59     0   100%

TOTAL                               888    393   56%

----------------------------------------------------------
Ran 46 tests in 0.170s

OK
```

加上--cover-package 选项是很重要的，否则就会看到**每个**被用到的 Python 包，包括标准库和第三方库。这个输出包括没有被运行的代码行，也就是没有被测试的代码行。所有需要做的只是打开你喜欢的文本编辑器然后开始写点儿什么。

但是也可以做得更好一点儿，让 coverage 生成漂亮的 HTML 报表。只需要加上 --cover-html 标志，这个 cover 目录就会在 HTML 页面中打开，然后每一页都会显示源代码的哪些部分运行与否如图 6-1 所示。

如果愿意的话，可以使用--cover-min-percentage=COVER_MIN_PERCENTAGE 选项，如果测试集运行时被执行的代码没有达到指定的最低百分比，这将会让测试集失败。

警告

代码覆盖率是 100%并不意味着代码已经被全部测试可以休息了。它只表明整个代码路径都被运行了，并不意味着每一个可能的条件都被测试到了。也就是说，这是个值得追求的目标，但并不意味着这是终点。

使用 testrepository 时，可以使用 setuptools 集成运行 coverage。

示例 6.16 使用 coverage 和 testrepository

```
$ python setup.py testr --coverage
```

这样可以结合 coverage 自动运行测试集，并在 cover 目录中生成 HTML 报告。

接下来你应该利用这些信息来巩固测试集，并为当前没有被运行过的任何代码添加测试。这是非常重要的，因为它有利于项目的后期维护，并有利于提升代码的整体质量。

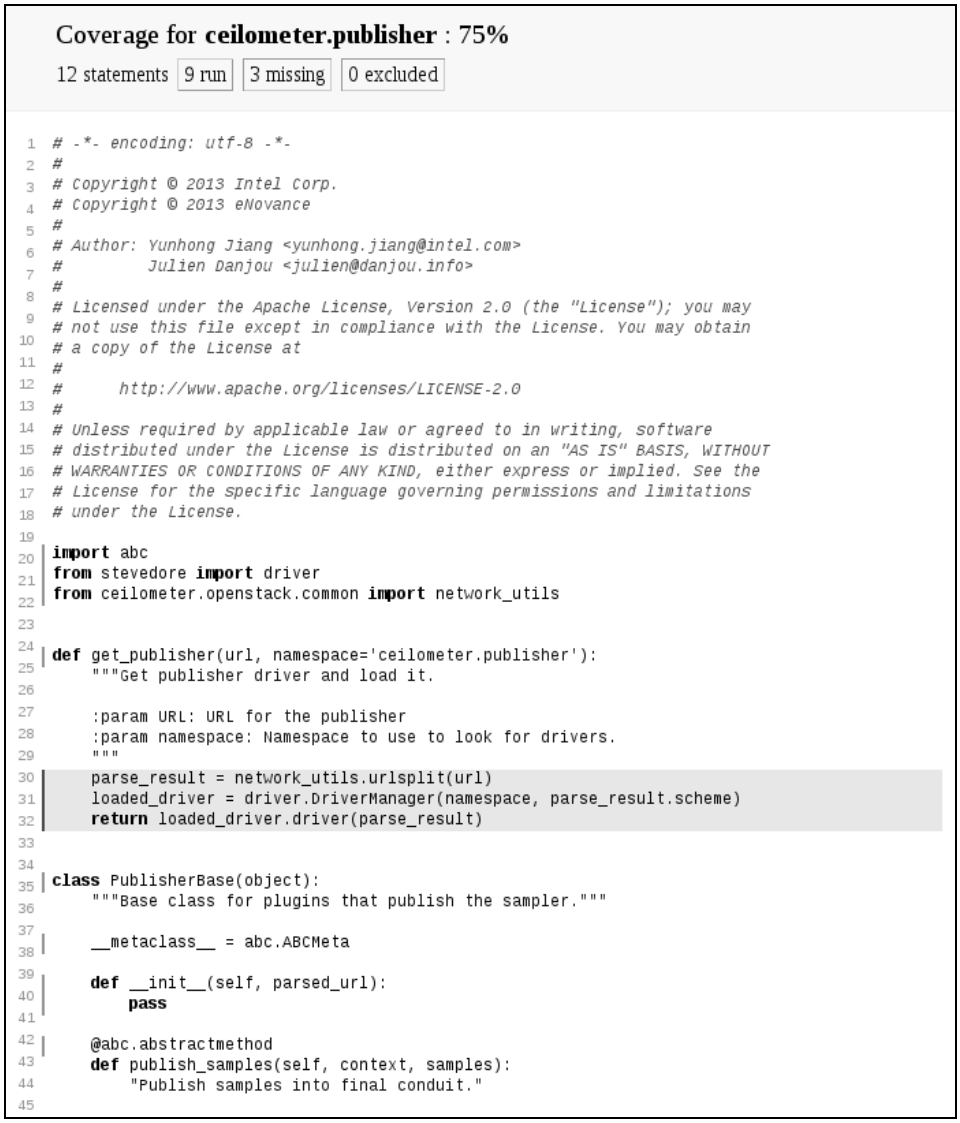


图 6-1　ceilometer.publisher 的覆盖率

6.7　使用虚拟环境和 tox

在第 5 章中，已经介绍并讨论了虚拟环境的使用。它的主要用途之一便是为单元测试提供干净的环境。当你认为你的测试工作正常但是实际上不正常时是相当郁闷的，如涉及依赖列表的情况。

可以写一个脚本去部署虚拟环境，安装 setuptools，然后安装应用程序/库的运行时或者单元测试所需要的所有依赖。但这是非常常见的用例，所以已经有专门针对这一需求的应用程序了，即 **tox**。

tox 的目标是自动化和标准化 Python 中运行测试的方式。基于这一目标，它提供了在一个干净的虚拟环境中运行整个测试集的所有功能，并安装被测试的应用程序以检查其安装是否正常。

使用 tox 之前，需要提供一个配置文件。这个文件名为 `tox.ini`，需要放在被测试项目的根目录，与 `setup.py` 同级。

```
$ touch tox.ini
```

现在可以成功运行 tox：

```
% tox
GLOB sdist-make: /home/jd/project/setup.py
python create: /home/jd/project/.tox/python
python inst: /home/jd/project/.tox/dist/project-1.zip
____________________ summary ____________________
  python: commands succeeded
  congratulations :)
```

显然这本身并不是很有用。在上面的例子中，tox 使用默认的 Python 版本在 `.tox/python` 中创建了一个虚拟环境，使用 `setup.py` 创建了应用程序的一个分发包并在这个虚拟环境中进行了安装。接下来并没有命令运行，因为该配置文件中并没有指定任何命令。

可以通过添加一个要在测试环境中运行的命令来改变其默认行为。编辑 `tox.ini`。让它包含以下内容：

```
[testenv]
commands=nosetests
```

要执行的 `nosetests` 命令很可能会失败，因为在该虚拟环境中我们并没有安装 `nosetests`。因此需要将其作为（将被安装的）依赖的一部分列出来。

```
[testenv]
deps=nose
commands=nosetests
```

再次运行，tox 会重建虚拟环境，安装新的依赖并运行 `nosetests` 命令，它将执行所

有单元测试。显然，我们可能需要添加更多的依赖，这可以通过配置项 deps 列出，也可以使用-rfile 语法从文件中读取。如果正在使用 pbr 管理 setup.py 文件，那么应该知道它是从一个名为 requirements.txt 的文件中读取所有依赖的。因此，让 tox 使用这个文件是一个好主意：

```
[testenv]
deps=nose
    -rrequirements.txt
commands=nosetests
```

文件中[testenv]一节定义的是被 tox 管理的所有虚拟环境参数。但正如前面所提及的，tox 能够真正地管理多个 Python 虚拟环境，通过向 tox 传入-e 标志就可以将测试运行在某个特定 Python 版本之上而不是运行在默认的版本之上。

```
% tox -e py26
GLOB sdist-make: /home/jd/project/setup.py
py26 create: /home/jd/project/.tox/py26
py26 installdeps: nose
py26 inst: /home/jd/project/.tox/dist/rebuildd-1.zip
py26 runtests: commands[0] | nosetests
.......

----------------------------------------------------------
Ran 7 tests in 0.029s

OK
_____________________ summary ________________________
  py26: commands succeeded
  congratulations :)
```

默认情况下，tox 可以模拟多种环境：py24、py25、py26、py27、py30、py31、py32、py33、jython 和 pypy！你甚至可以加入自定义的环境。要添加一个环境或者创建一个新环境，只需添加一个新的配置节[testenv:_envname_]。如果要针对其中的某个环境运行不同的命令，使用下面的 tox.ini 文件是很容易实现的：

```
[testenv]
deps=nose
commands=nosetests
```

```
[testenv:py27]
commands=pytest
```

这只覆盖了针对 py27 环境的命令，所以当运行 `tox -e py27` 时 nose 仍然会被作为依赖的一部分安装，但会执行 `pytest` 命令。

也可以使用 Python 不支持的版本创建新环境：

```
[testenv]
deps=nose
commands=nosetests

[testenv:py21]
basepython=python2.1
```

这里试图使用 Python 2.1 运行测试集，尽管我并不认为它能运行得起来。

如今，通常你可能希望应用程序能支持多个 Python 版本。让 tox 为想要默认支持的 Python 版本运行所有测试是非常有用的。这可以通过指定要使用的环境列表来实现，而在 tox 运行时无须提供参数。

```
[tox]
envlist=py26,py27,py33,pypy

[testenv]
deps=nose
commands=nosetests
```

当不指定任何参数运行 tox 时，列出的所有 4 种环境都将被创建，继而安装依赖和应用程序，然后运行命令 `nosetests`。

也可以使用 tox 来集成其他测试，如 `flake8`，正如 1.4 节中讨论过的。

```
[tox]
envlist=py26,py27,py33,pypy,pep8

[testenv]
deps=nose
commands=nosetests

[testenv:pep8]
deps=flake8
```

```
commands=flake8
```

在这个示例中，使用默认的 Python 版本运行 pep8 环境，不过这应该问题不大[①]。

提示

当运行 tox 时，你会发现所有的环境会按顺序创建并运行。这通常会令整个过程耗时很长。但因为虚拟环境都是隔离的，所以可以并行运行 `tox` 命令。这正是 `detox` 包（https://pypi.python.org/pypi/detox）要做的，即通过 `detox` 命令能够并行运行 envlist 中指定的所有默认环境。你应该运行 `pip install` 安装它。

6.8　测试策略

在项目中包含测试代码当然很好，但是如何运行这些测试也相当重要。实际上，在许多项目中尽管包含了测试代码，但是测试代码却由于各种原因无法运行。

尽管这个主题并不局限于 Python，但是考虑到其重要性，这里还是要强调一下：要对未测试代码零容忍。没有一组合适的单元测试覆盖的代码是不应该被合并的。

最低目标是保证每次代码提交都能通过所有测试，最好是能以自动的方式实现。

例如，OpenStack 会依赖基于 Gerrit（https://code.google.com/p/gerrit/）、Jenkins（http://jenkins-ci.org/）和 Zuul（http://ci.openstack.org/zuul/）的一个特定工作流程。每次代码提交都会经过基于 Gerrit 的代码评审系统，同时 Zuul 负责通过 Jenkins 运行一组测试任务。Jenkins 会针对各个项目运行单元测试以及各种更高级别的功能测试。这可以保证提交的代码能通过所有测试。由众多开发人员完成的代码评审保证所有被提交的代码都是具有相应的单元测试的。

如果正在使用流行的 GitHub 托管服务，Travis CI（https://travis-ci.org/）提供了一种在代码的签入（push）、合并（merge）或签出（pull）请求后运行测试的方式。尽管在提交后执行测试有些差强人意，但这仍然是针对回归问题的一种不错的方式。Travis 支持所有主要的 Python 版本，并可以高度定制。一旦通过它们的 Web 界面在项目中激活了 Travis，就可以通过加入一个简单的 `.travis.yml` 文件（如示例 6.17 所示）来完成后续工作。

① 如果想修改它，还是可以指定 `basepython`。

示例 6.17 `.travis.yml` 文件的例子

```
language: python
python:
  - "2.7"
  - "3.3"
# command to install dependencies
install: "pip install -r requirements.txt --use-mirrors"
# command to run tests
script: nosetests
```

无论你的代码托管在哪里，都应该尽可能实现软件测试的自动化，进而保证项目不断向前推进而不是引入更多 bug 而倒退。

6.9 Robert Collins 访谈

尽管你可能不知道 Robert 是谁，但你很可能已经用过他写的程序，别的暂且不提，他是分布式版本管理系统 Bazaar（http://bazaar.canonical.com/）的最初作者之一。目前，他是惠普云服务的“杰出技术专家”，从事 OpenStack 相关的工作。Robert 还开发过本书中介绍的很多 Python 工具，如 fixtures、testscenarios、testrepository 和 python-subunit。

你会建议使用什么样的测试策略？什么情况下不进行代码测试是可以接受的？

我认为这是一个软件工程上的取舍问题——需要考虑问题被引入未经检测产品的可能性，组件中未发现的问题产生的成本，从事这一工作的团队的规模和凝聚力……以 OpenStack

（http://openstack.org）为例，它拥有超过 1600 名贡献者，因此很难有细致入微的策略，因为很多人会有不同的意见。总体上来讲，应该有一些自动检查的方式作为代码并入主干时的组成部分，保证代码实现的正是其要做的，以及代码要做的也正是需要其完成的。通常来说，功能测试可能会放在不同的代码库中。单元测试的执行速度非常快，而且可以用来定义比较生僻的测试用例。我认为在已经有了测试的情况下，在测试的不同风格之间稍做平衡是完全可以的。

尽管测试的成本很高而回报较低，同时我也觉得在知情的情况下不测试是可以接受的，但这只是相对较少的情况。大部分事情都可以进行低成本的测试，而在早期发现问题的回报都相当高。

在编写 Python 代码时，有哪些令测试更容易且可提高代码质量的切实可行的最佳策略？

分而治之——不要在一个地方做多件事情。这便于重用，也可以让测试的重复运行更容易。尽可能使方法的功能单一，例如，对于单个方法要么用来计算，要么改变状态，但不要两者都做。这样就可以测试所有的计算行为而无须处理状态改变之类的操作，如写入数据库、与 HTTP 服务器交互等。相关的其他方式也同样受益，可以通过替换测试计算逻辑来触发生僻的测试用例的行为，并且通过仿真/测试进一步确认期望的状态传播是否如预期那样发生。测试 IME 最恶心的是深层栈，它们具有复杂的跨层的行为依赖。这里就需要不断改进代码，以使层之间的关系保持简单、可预测且对测试最有用的——可替换。

依你看来，在源代码中组织单元测试的最佳方式是什么？

使用类似`$ROOT/$PACKAGE/tests` 的层次结构，但我对于整个源代码树只创建一个（相对于这种`$ROOT/$PACKAGE/$SUBPACKAGE/tests`）。

在测试文件夹内部，我经常镜像源代码的其余部分的结构，如`$ROOT/$PACKAGE/foo.py` 将会在`$ROOT/$PACKAGE/tests/test_foo.py` 中被测试。

应该避免从源代码树的其余部分导入测试包，除非是在顶层的`__init__`中放一个`test_suite/load_tests` 函数。这样可以在小规模安装时很容易将测试分离。

Python 中有哪些库可以用来做功能测试？

我只是用项目中使用的 `unittest` 的某些部分，它能灵活地满足大部分需求，尤其是同 `testresources` 和并行运行的其他方式结合。

你能展望一下未来 Python 中单元测试库和框架的发展吗？

我能看到的一些大的挑战包括以下几个。

- 在新机器上持续扩展平行处理的能力。要知道现在手机都有 4 个 CPU 了，而已有的单元测试内部 API 并没有针对并行的工作负载进行过优化。我在开发的 StreamResult 就是针对这一问题的。
- 更加复杂的调度支持——针对此问题的一个不太难看的方案就是对类和模块的作用域进行限定。
- 找出方法汇总目前我们拥有的大量测试框架：生成跨多个项目的汇总视图将非常有用，例如，对于集成测试，常常有多个不同的测试运行器在使用。

第 7 章 方法和装饰器

Python 中提供了装饰器（decorator）作为修改函数的一种便捷的方式。它们在 Python 2.2 中伴随 `classmethod()`和 `staticmethod()`被首次引入，但随后又在 PEP 318（http://www.python.org/dev/peps/pep-0318/）中被大规模重构，并提高了灵活性和可读性。Python 提供了一些现成的装饰器（包括上面提到的两个），但似乎大部分开发人员并不了解它们背后的工作机制，本章就是要改变这一现状。

7.1 创建装饰器

装饰器本质上就是一个函数，这个函数接收其他函数作为参数，并将其以一个新的修改后的函数进行替换。很可能你已经使用过装饰器作为自己的包装函数。最简单的装饰器可能就是本体函数（identity function），它除了返回原函数什么都不做。

```
def identity(f):
    return f
```

然后就可以像下面这样使用这个装饰器：

```
@identity
def foo():
    return 'bar'
```

它和下面的过程类似：

```
def foo():
    return 'bar'

foo = identity(foo)
```

这个装饰器没什么用，但确实可以正常运行。只不过它什么都不做。

示例 7.1　注册装饰器

```
_functions = {}
def register(f):
    global _functions
    _functions[f.__name__] = f
    return f

@register
def foo():
    return 'bar'
```

在这个例子中，函数被注册并存储在一个字典里，以便后续可以根据函数名字提取函数。

在后面的几节中我会介绍 Python 中提供的标准装饰器，以及如何（何时）使用它们。

装饰器主要的应用场景是针对多个函数提供在其之前，之后或周围进行调用的通用代码。如果你写过 Emacs Lisp 代码，可能用过 defadvice，它允许你定义围绕某个函数进行调用的代码。同样的东西还有开发人员已经用过的非常棒的方法组合，来源于 CLOS（Common Lisp Object System）。

考虑这样一组函数，它们在被调用时需要对作为参数接收的用户名进行检查：

```
class Store(object):
    def get_food(self, username, food):
        if username != 'admin':
            raise Exception("This user is not allowed to get food")
        return self.storage.get(food)

    def put_food(self, username, food):
        if username != 'admin':
            raise Exception("This user is not allowed to put food")
        self.storage.put(food)
```

显然，第一步就是要先分离出检查部分的代码：

```
def check_is_admin(username):
    if username != 'admin':
        raise Exception("This user is not allowed to get food")

class Store(object):
    def get_food(self, username, food):
```

```
        check_is_admin(username)
        return self.storage.get(food)

    def put_food(self, username, food):
        check_is_admin(username)
        self.storage.put(food)
```

现在代码看上去稍微整洁了一点儿。但是有了装饰器能做得更好：

```
def check_is_admin(f):
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper

class Store(object):
    @check_is_admin
    def get_food(self, username, food):
        return self.storage.get(food)

    @check_is_admin
    def put_food(self, username, food):
        self.storage.put(food)
```

类似这样使用装饰器会让常用函数的管理更容易。如果有过正式的 Python 经验的话，这看起来有点儿老生常谈，但你可能没有意识到这种实现装饰器的原生方法有一些主要的缺点。

正如前面提到的，装饰器会用一个动态创建的新函数替换原来的。然而，新函数缺少很多原函数的属性，如 docstring 和名字。

```
>>> def is_admin(f):
...     def wrapper(*args, **kwargs):
...         if kwargs.get('username') != 'admin':
...             raise Exception("This user is not allowed to get food")
...         return f(*args, **kwargs)
...     return wrapper
...
>>> def foobar(username="someone"):
```

```
...     """Do crazy stuff."""
...     pass
...
>>> foobar.func_doc
'Do crazy stuff.'
>>> foobar.__name__
'foobar'
>>> @is_admin
... def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar.__doc__
>>> foobar.__name__
'wrapper'
```

幸好，Python 内置的 **functools** 模块通过其 update_wrapper 函数解决了这个问题，它会复制这些属性给这个包装器本身。update_wrapper 的源代码是自解释的，如示例 7.2 所示。

示例 7.2　Python 3.3 中 functools.update_wrapper 的源代码

```
WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',
                       '__annotations__')
WRAPPER_UPDATES = ('__dict__',)
def update_wrapper(wrapper,
                   wrapped,
                   assigned = WRAPPER_ASSIGNMENTS,
                   updated = WRAPPER_UPDATES):
    wrapper.__wrapped__ = wrapped
    for attr in assigned:
        try:
            value = getattr(wrapped, attr)
        except AttributeError:
            pass
        else:
            setattr(wrapper, attr, value)
    for attr in updated:
        getattr(wrapper, attr).update(getattr(wrapped, attr, {}))
```

```
    # Return the wrapper so this can be used as a decorator via partial()
    return wrapper
```

如果用这个函数改写前面的示例，代码看起来会更简洁：

```
>>> def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar = functools.update_wrapper(is_admin, foobar)
>>> foobar.__name__
'foobar'
>>> foobar.__doc__
'Do crazy stuff.'
```

手工调用 update_wrapper 创建装饰器很不方便，所以 **functools** 提供了名为 wraps 的装饰器，如示例 7.3 所示。

示例 7.3 使用 functools.wraps

```
import functools

def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper

class Store(object):
    @check_is_admin
    def get_food(self, username, food):
        return self.storage.get(food)
```

目前为止，在我们的示例中总是假设被装饰的函数会有一个名为 username 的关键字参数传入，但情况并非总是如此。考虑到这一点，最好是提供一个更加智能的装饰器，它能查看被装饰函数的参数并从中提取需要的参数。

为此，**inspect** 模块允许提取函数的签名并对其进行操作，如示例 7.4 所示。

示例 7.4　使用 inspect 获取函数参数

```
import functools
import inspect

def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        func_args = inspect.getcallargs(f, *args, **kwargs)
        if func_args.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper

@check_is_admin
def get_food(username, type='chocolate'):
    return type + " nom nom nom!"
```

承担主要工作的函数是 `inspect.getcallargs`，它返回一个将参数名字和值作为键值对的字典。在上面的例子中，这个函数返回`{'username': 'admin', 'type': 'chocolate'}`。这意味着我们的装饰器不必检查参数 `username` 是基于位置的参数还是关键字参数，而只需在字典中查找即可。

7.2　Python 中方法的运行机制

在此之前你可能已经写过很多方法但从未多想，但为了理解装饰器的行为，你就需要知道方法背后的运行机制。

方法是指作为类属性保存的函数。让我们来看一下当直接访问这样一个属性时到底发生了什么。在 Python 2 中情况如示例 7.5 所示，在 Python 3 中情况如示例 7.6 所示。

示例 7.5　Python 2 的方法

```
>>> class Pizza(object):
...     def __init__(self, size):
...         self.size = size
...     def get_size(self):
...         return self.size
```

```
...
>>> Pizza.get_size
<unbound method Pizza.get_size>
```

Python 2 会提示 `get_size` 属性是类 `Pizza` 的一个**未绑定**方法。

示例 7.6 Python 3 的方法

```
>>> class Pizza(object):
...     def __init__(self, size):
...         self.size = size
...     def get_size(self):
...         return self.size
...
>>> Pizza.get_size
<function Pizza.get_size at 0x7fdbfd1a8b90>
```

Python 3 中已经完全删除了未绑定方法这个概念，它会提示 `get_size` 是一个函数。

两种情况的本质是一样的：`get_size` 是一个并未关联到任何特定对象的函数，如果试图调用它的话，Python 会抛出错误（在 Python 2 中情况如示例 7.7 所示，在 Python 3 中情况如示例 7.8 所示）。

示例 7.7 在 Python 2 中调用未绑定的 `get_size`

```
>>> Pizza.get_size()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method get_size() must be called with Pizza instance as
first argument (got nothing instead)
```

示例 7.8 在 Python 3 中调用未绑定的 `get_size`

```
>>> Pizza.get_size()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: get_size() missing 1 required positional argument: 'self'
```

Python 2 中拒绝该方法调用是因为它是未绑定的。Python 3 允许调用，但会提示未提供必需的 `self` 参数。这使得 Python 3 更加灵活，不仅可以向方法传入该类的任意实例，还可以传入任何对象，只要它包含方法期望的属性：

```
>>> Pizza.get_size(Pizza(42))
42
```

尽管不太方便，但它能运行：每次调用类的一个方法都要对该类进行引用。

所以 Python 通过将类的方法绑定给实例为我们完成了后续工作。换句话说，可以通过任何 `Pizza` 访问 `get_size` 方法，进一步说，Python 会自动将对象本身传给方法的 `self` 参数，如示例 7.9 所示。

示例 7.9　调用绑定的 `get_size`

```
>>> Pizza(42).get_size
<bound method Pizza.get_size of <__main__.Pizza object at 0x7f3138827910>>
>>> Pizza(42).get_size()
42
```

不出所料，不需要传入任何参数给 `get_size`，因为它是绑定方法：它的 `self` 参数会自动设置为 `Pizza` 的实例。下面是一个更好的例子：

```
>>> m = Pizza(42).get_size
>>> m()
42
```

一旦有了对绑定方法的引用则无需保持对 `Pizza` 对象的引用。如果有了对方法的引用但是想知道它被绑定到了哪个对象，可以查看方法的`__self__`属性：

```
>>> m = Pizza(42).get_size
>>> m.__self__
<__main__.Pizza object at 0x7f3138827910>
>>> m == m.__self__.get_size
True
```

显然，仍然可以保持对对象的引用，并随时在需要的时候访问它。

7.3　静态方法

静态方法是属于类的方法，但实际上并非运行在类的实例上。具体见示例 7.10.

示例 7.10　`@staticmethod` 的用法

```
class Pizza(object):
```

```
    @staticmethod
    def mix_ingredients(x, y):
        return x + y

    def cook(self):
        return self.mix_ingredients(self.cheese, self.vegetables)
```

如果愿意的话，可以像非静态方法那样写 mix_ingredients，它会接收 self 作为参数但不会真地使用它。装饰器@staticmethod 提供了以下几种功能。

- Python 不必为我们创建的每个 Pizza 对象实例化一个绑定方法。绑定方法也是对象，创建它们是有开销的。使用静态方法可以避免这种开销：

```
>>> Pizza().cook is Pizza().cook
False
>>> Pizza().mix_ingredients is Pizza.mix_ingredients
True
>>> Pizza().mix_ingredients is Pizza().mix_ingredients
True
```

- 提高代码的可读性。当看到@staticmethod 时，就知道这个方法不依赖于对象的状态。
- 可以在子类中覆盖静态方法。如果使用一个定义在顶层模块中的 mix_ingredients 函数，那么一个继承自 Pizza 的子类在不重写 cook 方法的情况下将无法修改对披萨材料的混合方式。

7.4 类方法

类方法是直接绑定到类而非它的实例的方法：

```
>>> class Pizza(object):
...     radius = 42
...     @classmethod
...     def get_radius(cls):
...         return cls.radius
...
>>> Pizza.get_radius
<bound method type.get_radius of <class '__main__.Pizza'>>
```

```
>>> Pizza().get_radius
<bound method type.get_radius of <class '__main__.Pizza'>>
>>> Pizza.get_radius is Pizza().get_radius
True
>>> Pizza.get_radius()
42
```

然而，如果选择访问这个方法，它总是会被绑定在它所附着的类上，而且它的第一个参数将是类本身。（记住，类也是对象。）

类方法对于创建*工厂方法*最有用，即以特定方式实例化对象。如果用`@staticmethod`代替，则不得不在方法中硬编码类名 `Pizza`，使所有继承自 `Pizza` 的类都无法根据它们的需要使用这个工厂。

```
class Pizza(object):
    def __init__(self, ingredients):
        self.ingredients = ingredients

    @classmethod
    def from_fridge(cls, fridge):
        return cls(fridge.get_cheese() + fridge.get_vegetables())
```

在这个例子中，提供了工厂方法 `from_fridge`，可以传入一个 `Fridge` 对象。如果像 `Pizza.from_fridge(myfridge)` 这样调用这个方法，它会返回一个根据 `myfridge` 中可用的材料做成的全新 `Pizza`。

7.5　抽象方法

抽象方法是定义在基类中可能有或没有任何实现的方法。Python 中一个最简单的抽象方法类似这样：

```
class Pizza(object):
    @staticmethod
    def get_radius():
        raise NotImplementedError
```

任何继承自 `Pizza` 类的子类都需要实现并重写 `get_radius` 方法，否则调用这个方法会引发异常。

实现抽象方法的这种特定方式有一个缺陷：如果写一个继承自 Pizza 的类但忘了实现 get_radius 方法，那么只有在运行时调用这个方法时才会抛出错误，如示例 7.11 所示。

示例 7.11 实现一个抽象方法

```
>>> Pizza()
<__main__.Pizza object at 0x7fb747353d90>
>>> Pizza().get_radius()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in get_radius
NotImplementedError
```

如果使用 Python 内置的 abc 模块（http://docs.python.org/2/library/abc.html）实现抽象方法，在试图实例化一个包含抽象方法的对象时会得到警告提示，如示例 7.12 所示。

示例 7.12 使用 abc 实现抽象方法

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    @abc.abstractmethod
    def get_radius(self):
        """Method that should do something."""
```

当使用 abc 以及它的特殊类时，如果试图实例化 BasePizza 或其未重写 get_radius 方法的子类，会得到 TypeError：

```
>>> BasePizza()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class BasePizza with abstract methods
  get_radius
```

注意

元类（metaclass）的声明在 Python 2 和 Python 3 之间产生了变化，因此前面的例子只适用于 Python 2。

7.6　混合使用静态方法、类方法和抽象方法

这些装饰器各有各的用处，但有时可能会需要同时使用。下面介绍一些相关的小技巧。

抽象方法的原型并非一成不变。在实际实现方法的时候，可以根据需要对方法的参数进行扩展。

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    @abc.abstractmethod
    def get_ingredients(self):
        """Returns the ingredient list."""

class Calzone(BasePizza):
    def get_ingredients(self, with_egg=False):
        egg = Egg() if with_egg else None
        return self.ingredients + [egg]
```

这里可以任意定义 Calzone 的方法，只要仍然支持在基类 BasePizza 中定义的接口。这包括将它们作为类方法或静态方法进行实现：

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    @abc.abstractmethod
    def get_ingredients(self):
        """Returns the ingredient list."""

class DietPizza(BasePizza):
    @staticmethod
    def get_ingredients():
        return None
```

尽管静态方法 get_ingredients 没有基于对象的状态返回结果，但是它仍然满足在

基类 BasePizza 中定义的抽象接口，所以它仍然是有效的。

从 Python 3 开始（在 Python 2 中有问题，详见 issue 5867，http://bugs.python.org/issue5867），可能会支持在@abstractmethod 之上使用@staticmethod 和@classmethod 装饰器，如示例 7.13 所示。

示例 7.13 混合使用@classmethod 和@abstractmethod

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    ingredients = ['cheese']

    @classmethod
    @abc.abstractmethod
    def get_ingredients(cls):
        """Returns the ingredient list."""
        return cls.ingredients
```

注意，像这样在 BasePizza 中定义 get_ingredients 为类方法并不会强迫其子类也将其定义为类方法。将其定义为静态方法也是一样，没有办法强迫子类将抽象方法实现为某种特定类型的方法。

但是等一下，这里我们在抽象方法中居然是有实现代码的。真的可以吗？是的，在 Python 中完全没问题！不同于 Java，Python 中可以在抽象方法中放入代码并使用 super() 调用它，如示例 7.14 所示。

示例 7.14 通过抽象方法使用 super()

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    default_ingredients = ['cheese']

    @classmethod
```

```
    @abc.abstractmethod
    def get_ingredients(cls):
        """Returns the default ingredient list."""
        return cls.default_ingredients

class DietPizza(BasePizza):
    def get_ingredients(self):
        return [Egg()] + super(DietPizza, self).get_ingredients()
```

在这个例子中，每一个新的继承自 BasePizza 基类的 Pizza 子类都必须重写 get_ingredients 方法，但它可以通过基类的默认机制访问原料表。

7.7　关于 super 的真相

从 Python 的最早期开始，开发人员就能够通过单继承和多继承扩展他们的类。不过，很多开发人员似乎并不理解这些机制是如何工作的，以及与其关联的 super() 方法。

单继承和多继承各有利弊，组成或者鸭子类型都超出了本书的讨论范围，如果你不了解这些概念，那么建议读一些相关的资料以便形成自己的观点。

多继承仍然被广泛使用，尤其在使用了混合模式的代码里。这也是了解它仍然很重要的原因，因为它是 Python 核心的一部分。

注意

混入（mixin）类是指继承自两个或两个以上的类，并将它们的特性组合在一起。

到目前为止，你应该知道，在 Python 中类也是对象，而且对于这种构建类的特定声明方式应该非常熟悉了：class classname(expression of inheritance)。

括号内的部分是一个 Python 表达式，返回一个当前类要继承的类对象列表。通常，都会直接指定，但也可以像下面这样写：

```
>>> def parent():
...     return object
...
>>> class A(parent()):
...     pass
...
```

```
>>> A.mro()
[<class '__main__.A'>, <type 'object'>]
```

不出所料，可以正常运行：类 A 继承自父类 object。类方法 mro()返回*方法解析顺序*（method resolution order）用于解析属性。当前的 MRO 系统是在 Python 2.3 中第一次被实现的，关于其内部工作机制详见 Python 2.3 release notes（http://www.python.org/ download /releases/2.3/mro）。

你已经知道了调用父类方法的正规方式是通过 super()函数，但你很可能不知道 super()函数实际上是一个构造器，每次调用它都会实例化一个 super 对象。它接收一个或两个参数，第一个参数是一个类，第二个参数是一个子类或第一个参数的一个实例。

构造器返回的对象就像是第一个参数的父类的一个代理。它有自己的__getattribute__方法去遍历 MRO 列表中的类并返回第一个满足条件的属性：

```
>>> class A(object):
...     bar = 42
...     def foo(self):
...           pass
...
>>> class B(object):
...     bar = 0
...
>>> class C(A, B):
...     xyz = 'abc'
...
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type
'object'>]
>>> super(C, C()).bar
42
>>> super(C, C()).foo
<bound method C.foo of <__main__.C object at 0x7f0299255a90>>
>>> super(B).__self__
>>> super(B, B()).__self__
<__main__.B object at
```

当请求 C 的实例的访问其 super 对象的一个属性时，它会遍历 MRO 列表，并从第一个包含这个属性的类中返回这个属性。

前一个例子中使用了绑定的 super 对象，也就是说，通过两个参数调用 super。如果只通过一个参数调用 super()，则会返回一个未绑定的 super 对象：

```
>>> super(C)
<super: <class 'C'>, NULL>
```

由于对象是未绑定的，所以不能通过它访问类属性：

```
>>> super(C).foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'foo'
>>> super(C).bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'bar'
>>> super(C).xyz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'xyz'
```

粗一看，似乎这种 super 对象没什么用，但是 super 类通过某种方式实现了描述符协议（也就是__get__），这种方式能够让未绑定的 super 对象像类属性一样有用：

```
>>> class D(C):
...     sup = super(C)
...
>>> D().sup
<super: <class 'C'>, <D object>>
>>> D().sup.foo
<bound method D.foo of <__main__.D object at 0x7f0299255bd0>>
>>> D().sup.bar
42
```

通过用实例和属性名字作为参数来调用未绑定的 super 对象的__get__方法（super(C).__get__(D(), 'foo')）能够让它找到并解析 foo。

注意

尽管你可能没有听说过描述符协议，但是你很可能在不知道的情况已经通过@property 装饰器使用过它。它是 Python 的一种机制，允许对象以属性的方式进行存储以返回其他

东西而非其自身。本书不会讨论这个协议的具体细节，想详细了解可参考 Python 数据模型文档（http://docs.python.org/2/reference/datamodel.html#implementing-descriptors）。

在许多场景中使用 super 都是很有技巧的，例如处理继承链中不同的方法签名。遗憾的是，除了类似让方法接收*args, **kwargs 参数这样的技巧，针对这个问题同样“没有银弹”。

在 Python 3 中，super()变得更加神奇：可以在一个方法中不传入任何参数调用它。但没有参数传给 super()时，它会为它们自动搜索栈框架：

```
class B(A):
     def foo(self):
        super().foo()
```

super 是在子类中访问父类属性的标准方式，应该尽量使用它。它能确保父类方法的协作调用不出意外，例如在多继承时父类方法没有被调用或者被调用了两次。

第 8 章 函数式编程

函数式编程并不是考虑使用 Python 时需要考虑的首要问题，但 Python 对函数式编程的支持确实存在而且相当广泛。尽管许多 Python 程序员并没有意识到这一点，这有点儿难堪：除了少数情况，函数式编程可以让你写出更为精确和高效的代码。

在以函数式风格写代码时，函数应该设计成没有其他副作用。也就是说，函数接收参数并生成输出而不保留任何状态或修改任何不反映在返回值中的内容。遵循这种理想方式的函数可以被看成*纯函数式函数*。

一个非纯函数

```
def remove_last_item(mylist):
    """Removes the last item from a list."""
    mylist.pop(-1)  # This modifies mylist
```

一个纯函数

```
def butlast(mylist):
    """Like butlast in Lisp; returns the list without the last element."""
    return mylist[:-1]  # This returns a copy of mylist
```

函数式编程具有以下实用的特点。

- **可形式化证明**。诚然，这只是个纯理论的优点，没有人会用数学方法去证明一个 Python 程序。
- **模块化**。模块化编码能够在一定程度上强制对问题进行分治解决并简化在其他场景下的重用。
- **简洁**。函数式编程通常比其他范型更为简洁。

- **并发**。纯函数式函数是线程安全的并且可以并行运行。尽管在 Python 中还没实现，但期待一些语言能够自动进行处理，这在需要扩展应用程序时非常有用。
- **可测性**。测试一个函数式程序是非常简单的：所有需要做的仅仅是一组输入和一组期望的输出。而且是幂等的。

提示

如果想要更严谨的函数式编程，那么请参考我的建议：暂时从 Python 中跳出来放松一下，去学习 Lisp。我知道在一本 Python 书里谈 Lisp 很奇怪，但是这么多年同 Python 打交道的经验告诉我如何“函数式地思考”。如果你所有的经验都来自于命令式编程和面向对象编程，将很难拓展那种要充分利用函数式编程的思维过程。Lisp 本身也并非纯函数式，但是它比 Python 要更关注函数式编程。

8.1　生成器

生成器（generator）就是对象，在每次调用它的 `next()` 方法时返回一个值，直到它抛出 `StopIteration`。生成器是在 PEP 255（https://www.python.org/dev/peps/pep-0255/）中引入的，并提供了一种比较简单的实现迭代器（iterator）协议（https://docs.python.org/2/library/stdtypes.html#iterator-types）的方式来创建对象。

要创建一个生成器所需要做的只是写一个普通的包含 `yield` 语句的 Python 函数。Python 会检测对 `yield` 的使用并将这个函数标识为一个生成器。当函数执行到 `yield` 语句时，它会像 `return` 语句那样返回一个值，但一个明显不同在于：解释器会保存对栈的引用，它将被用来在下一次调用 `next` 函数时恢复函数的执行。

创建一个生成器

```
>> def mygenerator():
...     yield 1
...     yield 2
...     yield 'a'
...
>>> mygenerator()
<generator object mygenerator at 0x10d77fa50>
>>> g = mygenerator()
```

```
>>> next(g)
1
>>> next(g)
2
>>> next(g)
'a'
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以通过 inspect.isgeneratorfunction 检查一个函数是否是生成器。

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
>>> inspect.isgeneratorfunction(mygenerator)
True
>>> inspect.isgeneratorfunction(sum)
False
```

通过阅读 inspect.isgeneratorfunction 的源代码也可以洞察一些前面提到的生成器函数标识的内部实现。

inspect.isgeneratorfunction 的源代码

```
def isgeneratorfunction(object):
    """Return true if the object is a user-defined generator function.

    Generator function objects provides same attributes as functions.

    See help(isfunction) for attributes listing."""
    return bool((isfunction(object) or ismethod(object)) and
                object.func_code.co_flags & CO_GENERATOR)
```

Python 3 中提供了另一个有用的函数 inspect.getgeneratorstate。

```
>>> import inspect
>>> def mygenerator():
...     yield 1
...
```

```
>>> gen = mygenerator()
>>> gen
<generator object mygenerator at 0x7f94b44fec30>
>>> inspect.getgeneratorstate(gen)
'GEN_CREATED'
>>> next(gen)
1
>>> inspect.getgeneratorstate(gen)
'GEN_SUSPENDED'
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> inspect.getgeneratorstate(gen)
'GEN_CLOSED'
```

这个函数能够给出生成器的当前状态，允许我们判断它是否正在等待第一次被执行（GEN_CREATED），当前正在被解析器执行（GEN_RUNNING），等待被 next()调用唤醒（GEN_SUSPENDED）,或者已经结束运行（GEN_CLOSED）。

在 Python 中，生成器的构建是通过当函数产生某对象时保持一个对栈的引用来实现的，并在需要时恢复这个栈，例如，当调用 next()时会再次执行。

当迭代某种类型的数据时，直观的方式是先构建整个列表，这非常浪费内存。假设我们想找到 1～10 000 000 的第一个等于 50 000 的数字。听上去很简单，不是吗？让我们来挑战一下。这里将 Python 的运行内存限制在 128 MB。

```
$ ulimit -v 131072
$ python
>>> a = list(range(10000000))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

啊！证明不能以 128 MB 内存创建一个拥有 1000 万个元素的列表。

警告

在 Python 3 中，range()会返回生成器。要在 Python 2 中获取生成器，需要使用 xrange()。（这个函数在 Python 3 中不存在，因为已经重复了。）

换成生成器试试。

```
$ python
>>> for value in xrange(10000000):
...     if value == 50000:
...             print("Found it")
...             break
...
Found it
```

这次程序运行没有任何问题。range()函数返回一个可迭代对象，它会动态地生成整数列表。更妙的是，我们只关心第 50 000 个数字，生成器会只生成 50 000 个数字。

生成器运行通过即时生成的值以极少的内存消耗来应对大规模的数据集和循环处理。任何时候想要操作大规模数据，生成器都可以帮助确保有效地对数据进行处理。

yield 还有一个不太常用的功能：它可以像函数调用一样返回值。这允许通过调用它的 send()函数来向生成器传入一个值。

示例 8.1 通过 yield 返回值

```
def shorten(string_list):
    length = len(string_list[0])
    for s in string_list:
        length = yield s[:length]

mystringlist = ['loremipsum', 'dolorsit', 'ametfoobar']
shortstringlist = shorten(mystringlist)
result = []
try:
    s = next(shortstringlist)
    result.append(s)
    while True:
        number_of_vowels = len(filter(lambda letter: letter in 'aeiou', s))
        # Truncate the next string depending
        # on the number of vowels in the previous one
        s = shortstringlist.send(number_of_vowels)
        result.append(s)
except StopIteration:
    pass
```

在这个例子中，写了一个名为 shorten 的函数，它接收一个字符串列表并返回一个同样的字符串组成的列表，只不过是截断的。每个字符串的长度取决于前一个字符串中元音字母的个数。"loremipsum"包含四个元音，所以生成器返回第二个字符串"dolorsit"的前四个字母。"dolo"包含两个元音字母，所以"ametfoobar"将被截断成其前两个字母（"am"）。因此，这个然后生成器便停止并抛出 StopIteration。生成器将返回：

```
['loremipsum', 'dolo', 'am']
```

通过这种方式使用 yield 和 send()使得 Python 生成器的作用类似于 Lua（http://www.lua.org/）和其他语言中的协同程序（coroutine）。

提示

PEP 289（https://www.python.org/dev/peps/pep-0289/）引入了生成器表达式，通过使用类似列表解析的语法可以构建单行生成器。

```
>>> (x.upper() for x in ['hello', 'world'])
<generator object <genexpr> at 0x7ffab3832fa0>
>>> gen = (x.upper() for x in ['hello', 'world'])
>>> list(gen)
['HELLO', 'WORLD']
```

8.2　列表解析

列表解析（list comprehension，简称 listcomp）让你可以通过声明在单行内构造列表的内容。

没有列表解析的情况

```
>>> x = []
>>> for i in (1, 2, 3):
...     x.append(i)
...
>>> x
[1, 2, 3]
```

使用列表解析的实现

```
>>> x = [i for i in (1, 2, 3)]
>>> x
```

```
[1, 2, 3]
```

可以同时使用多条 for 语句并使用 if 语句过滤元素：

```
x = [word.capitalize()
    for line in ("hello world?", "world!", "or not")
    for word in line.split()
    if not word.startswith("or")]
>>> x
['Hello', 'World?', 'World!', 'Not']
```

使用列表解析而不使用循环是快速定义列表的简洁方式。因为我们仍然在讨论函数式编程，值得一提的是通过列表解析构建的列表是不能依赖于程序的状态的[①]。这通常让它们比非列表解析构建的列表更加简洁易读。

注意

也有一些语法用于以同样的方式构建字典和集合：

```
>> {x:x.upper() for x in ['hello', 'world']}
{'world': 'WORLD', 'hello': 'HELLO'}
>> {x.upper() for x in ['hello', 'world']}
set(['WORLD', 'HELLO'])
```

注意，这只在 Python 2.7 及后续版本中有效。

8.3 函数式，函数的，函数化

Python 包括很多针对函数式编程的工具。这些内置的函数涵盖了以下这些基本部分。

- map(function, iterable)对 iterable 中的每一个元素应用 function，并在 Python 2 中返回一个列表，或者在 Python 3 中返回可迭代的 map 对象。

Python 3 中 map 的用法

```
>>> map(lambda x: x + "bzz!", ["I think", "I'm good"])
<map object at 0x7fe7101abdd0>
>>> list(map(lambda x: x + "bzz!", ["I think", "I'm good"]))
['I thinkbzz!', "I'm goodbzz!"]
```

① 技术上可以，但那不是期望的方式。

- filter(function or None, iterable)对iterable中的元素应用function对返回结果进行过滤，并在 Python 2 中返回一个列表，或者在 Python 3 中返回可迭代的 filter 对象，如示例 8.2 所示。

示例 8.2　Python 3 中 filter 的用法

```
>>> filter(lambda x: x.startswith("I "), ["I think", "I'm good"])
<filter object at 0x7f9a0d636dd0>
>>> list(filter(lambda x: x.startswith("I "), ["I think", "I'm good"]))
['I think']
```

提示

可以使用生成器和列表解析实现与 filter 或者 map 等价的函数。

使用列表解析实现 map

```
>>> (x + "bzz!" for x in ["I think", "I'm good"])
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x + "bzz!" for x in ["I think", "I'm good"]]
['I thinkbzz!', "I'm goodbzz!"]
```

使用列表解析实现 filter

```
>>> (x for x in ["I think", "I'm good"] if x.startswith("I "))
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x for x in ["I think", "I'm good"] if x.startswith("I ")]
['I think']
```

在 Python 2 中这样使用生成器会返回一个可迭代的对象而不是列表，就像 Python 3 中的 map 和 filter 函数。

- enumerate(iterable[, start])返回一个可迭代的 enumerate 对象，它生成一个元组序列，每个元组包括一个整型索引（如果提供了的话，则从 start 开始）和 iterable 中对应的元素。当需要参考数组的索引编写代码时这是很有用的。例如，不使用下面的写法：

```
i = 0
while i < len(mylist):
print("Item %d: %s" % (i, mylist[i]))
i += 1
```

可以用这样的写法：

```
for i, item in enumerate(mylist):
print("Item %d: %s" % (i, item))
```

- `sorted(iterable, key=None, reverse=False)`返回 `iterable` 的一个已排序版本。通过参数 `key` 可以提供一个返回要排序的值的函数。
- `any(iterable)`和 `all(iterable)`都返回一个依赖于 `iterable` 返回的值的布尔值。下面这两个函数是等价对：

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

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

这两个函数对于判断这个可迭代对象中的任何或所有值是否满足给定的条件非常有用：

```
mylist = [0, 1, 3, -1]
if all(map(lambda x: x > 0, mylist)):
print("All items are greater than 0")
if any(map(lambda x: x > 0, mylist)):
print("At least one item is greater than 0")
```

- `zip(iter1 [,iter2 [...]])`接收多个序列并将它们组合成元组。它在将一组键和一组值组合成字典时很有用。像上面描述的其他函数一样，它在 Python 2 中返回一个列表，在 Python 3 中返回一个可迭代的对象。

```
>>> keys = ["foobar", "barzz", "ba!"]
>>> map(len, keys)
<map object at 0x7fc1686100d0>
>>> zip(keys, map(len, keys))
<zip object at 0x7fc16860d440>
```

```
>>> list(zip(keys, map(len, keys)))
 [('foobar', 6), ('barzz', 5), ('ba!', 3)]
>>> dict(zip(keys, map(len, keys)))
{'foobar': 6, 'barzz': 5, 'ba!': 3}
```

到这里你可能已经注意到 Python 2 和 Python 3 的返回类型的不同。在 Python 2 中大多数 Python 内置的纯函数会返回列表而不是可迭代的对象，这使得它们的内存利用没有 Python 3.x 中那么高效。如果正计划使用这些函数编写代码，记住在 Python 3 中才能最大的发挥它们的作用。如果你仍然在使用 Python 2，也不用气馁，标准库中的 `itertools` 模块提供了许多这些函数的迭代器版本（`itertools.izip`、`itertoolz.imap`、`itertools.ifilter` 等）。

然而，在上面这个列表中仍然缺少一个重要的工具。处理列表时，一个常见的任务就是在从列表中找出第一个满足条件的元素。这通常可以用函数这么实现：

```
def first_positive_number(numbers):
    for n in numbers:
        if n > 0:
            return n
```

也可以写一个函数式风格的：

```
def first(predicate, items):
    for item in items:
        if predicate(item):
            return item

first(lambda x: x > 0, [-1, 0, 1, 2])
```

或者更精确一点儿：

```
# Less efficient
list(filter(lambda x: x > 0, [-1, 0, 1, 2]))[0] ❶
# Efficient but for Python 3
next(filter(lambda x: x > 0, [-1, 0, 1, 2]))
# Efficient but for Python 2
next(itertools.ifilter(lambda x: x > 0, [-1, 0, 1, 2]))
```

❶ 注意，如果没有元素满足条件，可能会抛出 `IndexError`，促使 `list(filter())`返回空列表。

为了避免在每一个程序中都写同样的函数，可以包含这个小巧且使用的 Python 包 **`first`**（https://pypi.python.org/pypi/first）。

示例 8.3 使用 `first`

```
>>> from first import first
>>> first([0, False, None, [], (), 42])
42
>>> first([-1, 0, 1, 2])
-1
>>> first([-1, 0, 1, 2], key=lambda x: x > 0)
1
```

参数 `key` 可以用来指定一个函数，接收每个元素作为参数并返回布尔值指示该元素是否满足条件。

你可能已经注意到，在本章相当一部分示例中我们使用了 `lambda` 表达式。`lambda` 最早在 Python 中引入实际是为了给函数式编程函数提供便利，如 `map()` 和 `filter()`，否则每次想要检查不同的条件时将需要重写一个完整的新函数：

```
import operator
from first import first

def greater_than_zero(number):
    return number > 0

first([-1, 0, 1, 2], key=greater_than_zero)
```

这段代码和前面的例子功能相同，但更加难以处理：如果想要获得序列中第一个大于 42 的数，则需要定义一个合适的函数而不是以内联（in-line）的方式在 `first` 中调用。

尽管 `lambda` 在帮助我们避免此类问题时是有用的，但它仍然是有问题的。首先也最明显的，如果需要超过一行代码则不能通过 `lambda` 传入 `key` 函数。在这种场景下，则又需要返回繁复的模式，为每个需要的 `key` 编写一个新的函数定义。我们真的需要这样做吗？

`functools.partial` 是以更为灵活的方案替换 `lambda` 的第一步。它允许通过一种反转的方式创建一个包装器函数：它修改收到的参数而不是修改函数的行为。

```
from functools import partial
from first import first

def greater_than(number, min=0):
    return number > min
```

```
first([-1, 0, 1, 2], key=partial(greater_than, min=42))
```

默认情况下，新的 greater_than 函数功能和之前的 greater_than_zero 函数类似，但是可以指定要比较的值。在这个例子中，我们向 functools.partial 传入我们的函数和期望的最小值 min，就可以得到一个 min 设定为 42 的新函数。换句话说，可以写一个函数并利用 functools.partial 根据需要针对给定的某个场景进行自定义。

在这个例子中，尽管需求严格限定但还是需要两行代码。例子中需要做的只是比较两个数，假如 Python 中有内置的这种比较函数呢？事实证明，**operator** 模块就是我们要找的。

```
import operator
from functools import partial
from first import first

first([-1, 0, 1, 2], key=partial(operator.le, 0))
```

这里我们看到，functools.partial 也支持位置参数。在这个例子中，operator.le(a, b) 接收两个数并返回第一个数是否小于等于第二个数。这里向 functools.partial 传入的 0 会被赋值给 a，向由 functools.partial 返回的函数传递的参数会被赋值给 b。所以它运行起来和最初的例子完全一样，不需要使用 lambda 或其他额外的函数。

注意

functools.partial 在替换 lambda 时是很有用的，而且通常被认为是更好的选择。在 Python 语言中 lambda 被看做是一种非常规方式，因为它将函数体限定为一行表达式[①]。另一方面，functools.partial 可以围绕原函数进行很好的封装。

Python 标准库中的 **itertools** 模块也提供了一组非常有用的函数，也很有必要记住。因为尽管 Python 本身提供了这些函数，但我还是看到很多程序员试图实现自己的版本。

- chain(*iterables) 依次迭代多个 iterables 但并不会构造包含所有元素的中间列表。
- combinations(iterable, r) 从给定的 iterable 中生成所有长度为 r 的组合。

① 曾经计划在 Python 3 中移除，但是最终没有。

- compress(data, selectors)对 data 应用来自 selectors 的布尔掩码并从 data 中返回 selectors 中对应为真的元素。
- count(start, step)创建一个无限的值的序列，从 start 开始，步长为 step。
- cycle(iterable)重复的遍历 iterable 中的值。
- dropwhile(predicate, iterable)过滤 iterable 中的元素，丢弃符合 predicate 描述的那些元素。
- groupby(iterable, keyfunc)根据 keyfunc 函数返回的结果对元素进行分组并返回一个迭代器。
- permutations(iterable[, r])返回 iterable 中 r 个元素的所有组合。
- product(*iterables)返回 iterables 的笛卡儿积的可迭代对象，但不使用嵌套的 for 循环。
- takewhile(predicate, iterable)返回满足 predicate 条件的 iterable 中的元素。

这些函数在和 operator 模块组合在一起时特别有用。当一起使用时，itertools 和 operator 能够覆盖通常程序员依赖 lambda 表达式的大部分场景，如示例 8.4 所示。

示例 8.4 结合 itertools.groupby 使用 operator 模块

```
>>> import itertools
>>> a = [{'foo': 'bar'}, {'foo': 'bar', 'x': 42}, {'foo': 'baz', 'y': 43}]
>>> import operator
>>> list(itertools.groupby(a, operator.itemgetter('foo')))
[('bar', <itertools._grouper object at 0xb000d0>), ('baz', <itertools.
  _grouper object at 0xb00110>)]
>>> [(key, list(group)) for key, group in list(itertools.groupby(a, operator.
  itemgetter('foo')))]
[('bar', []), ('baz', [{'y': 43, 'foo': 'baz'}])]
```

在这个例子中，也可以写成 lambda x: x['foo']，但使用 operator 可以完全避免使用 lambda。

第 9 章 抽象语法树

抽象语法树（Abstract Syntax Tree，AST）是任何语言源代码的抽象结构的树状表示，包括 Python 语言。作为 Python 自己的抽象语法树，它是基于对 Python 源文件的解析而构建的。

关于 Python 的这个部分并没有太多的文档，而且刚开始看的时候并不容易理解。尽管如此，Python 作为一门编程语言，要掌握它的用法，了解并理解 Python 的一些深层次的构造仍然是很有意义的。

了解 Python 抽象语法树的最简单的方式就是解析一段 Python 代码并将其转储从而生成抽象语法树。要做到这一点，Python 的 `ast` 模块就可以满足需要。

示例 9.1　将 Python 代码解析成抽象语法树

```
    >>> import ast
    >>> ast.parse
    <function parse at 0x7f062731d950>
    >>> ast.parse("x = 42")
    <_ast.Module object at 0x7f0628a5ad10>
    >>> ast.dump(ast.parse("x = 42"))
    "Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
value=Num(n=42))])"
```

`ast.parse` 函数会返回一个 `_ast.Module` 对象，作为树的根。这个树可以通过 `ast.dump` 模块整个转储，针对这个例子如图 9-1 所示。

抽象语法树的构建通常从根元素开始，根元素通常是一个 `ast.Module` 对象。这个对象在其 `body` 属性中包含一组待求值的语句或者表达式。它通常表示这个文件的内容。

很容易猜到，`ast.Assign` 对象表示赋值，在 Python 语法中它对应=。`Assign` 有一组目

标，以及一个要赋的值。在这个例子中只有一个对象 ast.Name，表示变量 x。值是数值 42。

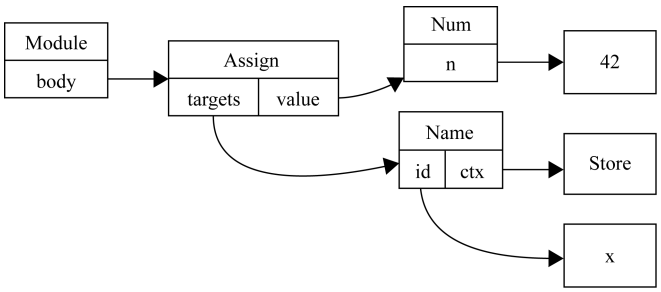


图 9-1

抽象语法树能够被传入 Python 并编译和求值。作为 Python 内置函数提供的 compile 函数是支持抽象语法树的。

```
>>> compile(ast.parse("x = 42"), '<input>', 'exec')
<code object <module> at 0x111b3b0, file "<input>", line 1>
>>> eval(compile(ast.parse("x = 42"), '<input>', 'exec'))
>>> x
42
```

通过 ast 模块中提供的类可以手工构建抽象语法树。显然，这么写 Python 代码太麻烦了，不推荐这种方法。但是用起来还是挺有意思的。

让我们用 Python 的抽象语法树写一个经典的"Hello world!"。

示例 9.2　使用 Python 抽象语法树的 Hello world

```
>>> hello_world = ast.Str(s='hello world!', lineno=1, col_offset=1)
>>> print_call = ast.Print(values=[hello_world], lineno=1, col_offset=1,
nl=True)
>>> module = ast.Module(body=[print_call])
>>> code = compile(module, '', 'exec')
>>> eval(code)
hello world!
```

注意

lineno 和 col_offset 表示用于生成抽象语法树的源代码的行号和列偏移量。在当前环境下设置它们其实没多大意义，因为这里并没有解析任何源代码，但用来回查生成抽象

语法树的代码的位置时比较有用。例如，在 Python 生成栈回溯时就比较有用。不管怎样，Python 拒绝编译任何不提供此信息的抽象语法树对象，这也是我们在这里传入一个假的值 1 的原因。ast.fix_missing_locations()函数可以通过在父节点上设置缺失的值来解决这个问题。

抽象语法树中可用的完整对象列表通过阅读_ast（注意下划线）模块的文档可以很容易获得。

首先需要考虑的两个分类是语句和表达式。语句涵盖的类型包括 assert（断言）、赋值（=）、增量赋值（+=、/=等）、global、def、if、return、for、class、pass、import 等。它们都继承自 ast.stmt。表达式涵盖的类型包括 lambda、number、yield、name（变量）、compare 或者 call。它们都继承自 ast.expr。

还有其他一些分类，例如，ast.operator 用来定义标准的运算符，如加（+）、除（/）、右移（>>）等，ast.cmpop 用来定义比较运算符。

很容易联想到有可能利用抽象语法树构造一个编译器，通过构造一个 Python 抽象语法树来解析字符串并生成代码。这正是 9.1 节中将要讨论的 Hy 项目。

如果需要遍历树，可以用 ast.walk 函数来做这件事。但 ast 模块还提供了 NodeTransformer，一个可以创建其子类来遍历抽象语法树的某些节点的类。因此用它来动态修改代码很容易。为加法修改所有的二元运算如示例 9.3 所示。

示例 9.3　为加法修改所有的二元运算

```
import ast

class ReplaceBinOp(ast.NodeTransformer):
    """Replace operation by addition in binary operation"""
    def visit_BinOp(self, node):
        return ast.BinOp(left=node.left,
                         op=ast.Add(),
                         right=node.right)

tree = ast.parse("x = 1/3")
ast.fix_missing_locations(tree)
eval(compile(tree, '', 'exec'))
```

```
print(ast.dump(tree))
print(x)
tree = ReplaceBinOp().visit(tree)
ast.fix_missing_locations(tree)
print(ast.dump(tree))
eval(compile(tree, '', 'exec'))
print(x)
```

执行结果如下：

```
Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
                    value=BinOp(left=Num(n=1), op=Div(), right=Num(n=3)))])
0.33 33333333333333
Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
                    value=BinOp(left=Num(n=1), op=Add(), right=Num(n=3)))])
4
```

提示

如果需要对 Python 的字符串进行求值并返回一个简单的数据类型，可以使用 `ast.literal_eval`。与 `eval` 不同，`ast.literal_eval` 不允许输入的字符串执行任何代码。比 `eval` 更安全。

9.1　Hy

初步了解抽象语法树之后，可以畅想一下为 Python 创建一种新的语法，并将其解析并编译成标准的 Python 抽象语法树。Hy 编程语言（http://docs.hylang.org/en/latest/）做的正是这件事。它是 Lisp 的一个变种，可以解析类 Lisp 语言并将其转换为标准的 Python 抽象语法树。因此它同 Python 生态系统完全兼容。可以将其与 Clojure（http://clojure.org/）同 Java 的关系类比。Hy 完全可以单独写本书来讲，所以本节只是稍作介绍。

如果你已经写过 Lisp 代码[①]，Hy 语法和它看起来非常类似。安装之后，运行 `hy` 解释器将给出一个标准的 REPL（Read-Eval-Print Loop）提示符，从这里可以同解释器进行交互。

```
% hy
hy 0.9.10
```

① 如果还没有，值得考虑一下。

```
=> (+ 1 1)
2
```

在 Lisp 语法中，圆括号表示一个列表，第一个元素是一个函数，其余元素是该函数的参数。因此，上面的代码相当于 Python 中的 `1+1`。

大多数结构都是从 Python 直接映射过来的，如函数定义。变量的设置则依赖于 `setv` 函数。

```
=> (defn hello [name]
...  (print "Hello world!")
...  (print (% "Nice to meet you %s" name)))
=> (hello "jd")
Hello world!
Nice to meet you jd
```

在内部，Hy 对提供的代码进行解析并编译成 Python 抽象语法树。幸运的是，Lisp 比较容易解析成树，因为每一对圆括号都可以表示成列表树的一个节点。需要做的仅仅是将 Lisp 树转换为 Python 抽象语法树。

通过 `defclass` 结构可以支持类定义，这是从 CLOS（Common Lisp ObjectSystem）获得的启发。

```
 (defclass A [object]
  [[x 42]
   [y (fn [self value]
        (+ self.x value))]])
```

上面的代码定义了一个名为 `A` 的类，继承自 `object`，包括一个值为 `42` 的类属性 `x`，以及用来返回传入参数和 `x` 的和的一个方法。

最棒的是，你可以直接导入**任何 Python 库**到 Hy 中并随意使用。

```
=> (import uuid)
=> (uuid.uuid4)
UUID('f823a749-a65a-4a62-b853-2687c69d0e1e')
=> (str (uuid.uuid4))
'4efa60f2-23a4-4fc1-8134-00f5c271f809'
```

Hy 还包括更多高级结构和宏。如果想在 Python 中拥有 `case` 或 `switch` 语句，可以欣赏一下 `cond` 是怎么做的。

```
(cond
 ((> somevar 50)
  (print "That variable is too big!"))
 ((< somevar 10)
  (print "That variable is too small!"))
 (true
  (print "That variable is jusssst right!")))
```

Hy 是 一个非常不错的项目，因为它允许你进入 Lisp 的世界又不用离开你熟悉的领域，因为你仍然在写 Python。hy2py 工具甚至能够显示 Hy 代码转换成 Python 之后的样子。[①]

9.2　Paul Tagliamonte 访谈

Paul 是 Debian 开发人员，在 Sunlight 基金会工作。2013 年他创建了 Hy，作为 Lisp 的爱好者我随后加入了这个美妙的冒险。

最初你为什么会创立 Hy 项目？

最初，我创立 Hy 这个项目是源自一次关于如何将 Lisp 代码编译成 Python 而不是 Java 的 JVM（Clojure）的谈话。不久之后，我开发了 Hy 的第一个版本，有些像 Lisp，甚至执行起来就像 Lisp 一样，但它却运行缓慢。我是说，非常慢。比原生的 Python 要慢一个数量级，因为这个 Lisp 运行时本身是用 Python 实现的。

① 尽管它有一些限制性。

非常受挫，我几乎要放弃了，继续推进只是因为向一个同事承诺了要用抽象语法树实现这个运行时，而不是用 Python 实现。这个疯狂的想法实际上激发了整个项目。这发生在 2012 年假期前不久，所以我整个假期都在忙着开发 Hy。大概一周之后，做出来的东西和现在的 Hy 代码库已经非常接近了，大部分 Hy 的开发人员甚至已经知道如何围绕这个编译器进行开发了。

就在实现了一个简单的 Flask 应用之后，我在波士顿 Python 大会上做了一场关于这个项目的演讲，并收到了热烈的反馈，非常热烈。实际上，我已经开始将 Hy 作为讲解 Python 内部机制的一个很好的途径，例如，REPL 是如何工作的[①]，PEP 302 导入钩子，以及 Python 抽象语法树，都是关于用代码写代码这一概念的很好的诠释。

在那次演讲之后，我对有些小节不太满意，所以我重写了编译器的很多代码，以解决一些流程上的原则问题，从而得到当前这个代码库的迭代，它已经运行了相当长一段时间了。

此外，Hy（这门语言）是帮助人们理解如何阅读 Lisp 代码的一种很好的方式，因为这样他们就可以在自己熟悉的环境中舒服地使用 s 表达式（甚至使用他们已经依赖的一些库），平滑地过度到其他（真正的）Lisp，如 Common Lisp、Scheme 或者 Clojure，以及试验一些新的想法（如宏系统、单调性和没有语句的概念）。

你是怎么知道该如何正确地使用抽象语法树的呢？对于查看抽象语法树的人，有什么提示、技巧和建议吗？

Python 的抽象语法树是很有意思的。它并不是非常私密（事实上，它明显不是私密的），但是也并非公开的接口。版本之间并不保证稳定，事实上，Python 2 和 Python 3 之间有些非常烦人的不同，甚至 Python 3 的不同版本间也有不同。此外，不同的实现对抽象语法树的解释也不同，甚至有独特的抽象语法树。更不用说 Jython、PyPy 或者 CPython 要以一致的方式处理 Python 抽象语法树。

例如，CPython 能够处理抽象语法树实体的轻微的乱序（通过 `lineno` 和 `col_offset`）。然而 PyPy 会抛出一个断言错误。尽管有时令人抓狂，但抽象语法树一般都很正常。构造一个可在不同 Python 实例上运行的抽象语法树不是完全不可能的。通过一两个条件，就可以使这个工具变得非常顺手，不过创建一个在 CPython 2.6 到 3.3 以及 PyPy 都能运行的抽象语法树还是挺让人抓狂的。

① `code.InteractiveConsole`

抽象语法树的文档极度缺乏，所以大部分知识来自于对生成抽象语法树的逆向工程。通过编写简单的 Python 脚本，可以使用类似 `import ast; ast.dump(ast.parse("print foo"))` 这样的代码生成等价的抽象语法树，来辅助理解。伴随着一些猜测和坚持，最终形成了对这种方式的基本理解。

将来我也许会将我对抽象语法树模块的理解记录下来，但是我发现写代码是学习抽象语法树的最好方式。

Hy 现在处于什么状态？未来的目标是什么？

Hy 仍然在开发中。仍有一些需要解决的细小问题以及需要修复的一些 bug，以便使 Hy 同其他的 LISP-1 的变种没有显著区别。这是一项艰巨的任务，但其时机已经成熟。

我还希望能保持 Hy 的效率，目前看还行。

长期来说，我希望 Hy 能成为一个教学工具，用来解释一些即便对有经验的 Python 支持者来说也比较陌生的概念。我也希望它能证明，对这些我们尽力提供的工具产生兴趣可以令这些 Python 的支持者得到更多乐趣。

我希望人们可以将 Hy 看做是一个绝好的教学工具，以便能让人们对 Common Lisp、Clojure 或者 Scheme 产生兴趣。我希望人们回家之后能读一下为什么 Lisp 的变量这样工作，以及在他们的日常编码工作中如何借用这种哲学思想。

Hy 和 Python 的互操作性如何？代码分发和打包呢？

绝对是极好的、令人震惊的互操作性。事实上，甚至可以通过 `pdb` 调试 Hy 而无需任何修改。为了彻底测试它，我写乐 Flask 应用、Django 应用和各种模块。Python 可以导入 Python，Hy 可以导入 Hy，Hy 可以导入 Python，Python 可以导入 Hy。这是 Hy 非常独特的地方，即使是 Clojure 也无法做到这一点，因为 Clojure 的互操作是绝对单向的（Clojure 可以导入 Java，但是 Java 导入 lojure 却有问题）。这也充分证明了我们的工具有多强大。

Hy 几乎是直接将 Hy 代码（以 s 表达式）转换为 Python 抽象语法树。这个编译步骤也证明了生成字节代码相当明智（以至于通过看由 Python 抽象语法树生成的 Python 源代码就可以调试 Hy 的一些讨厌的问题），这也意味着 Python 在处理非 Python 语言编写的模块时非常困难。

Common Lisp 主义，如完全支持将`*earmuffs*`和 `using-dashes` 翻译成 Python 的对等物（在这种情况下，`*earmuffs*` 会变成 `EARMUFFS`，`using-dashes` 会变成

`using_dashes`），这意味着 Python 处理它们并不难。

确保良好的互操作性是我们最高优先级的工作之一，所以如果你发现任何 bug，请给我们看 bug。

选择 Hy 的优点与缺点各是什么呢？

这是个很有意思的问题，我立场并不中立，所以我持保留态度。

青出蓝但胜于蓝，Hy 通过一些特殊的方式表现得比 Python 要好。因为我们做了大量的努力去使 Python 不同版本间的行为更顺畅以使新的 Python 3 的 future 实现得更快。主要通过使用如 Python 2 中 future 的除法，以及确保两个版本间语法的标准化这类方式实现。

此外，Hy 有一些 Python 很难处理的东西（即使有抽象语法树这样优秀的模块），就是一个完整的宏系统。宏是非常特殊的函数，它能在编译阶段修改代码，不同于有 `ast.NodeVisitor` 作为一级函数的语言。这使得创建你的特定领域语言（DSL）非常简单，特定领域语言由基本语言（在这里就是 Hy/Python），以及额外的许多表现力独特、代码简洁的宏组成的。

许多时候，聪明的特定领域语言可以替代语言来扮演这个角色，如 Lua。

至于说缺点，给予 Hy 力量的东西也可能会伤害它。从社会层面而非技术层面说，凭借 s 表达式编写的 Lisp，背负着难于学习，、阅读和维护的恶名。出于对 Hy 复杂性的恐惧，人们可能不愿意工作于使用 Hy 的项目上。

Hy 就是人们由爱到恨的 Lisp——Python 开发者不喜欢它的语法，Lisp 开发者不愿意使用 Hy，因为它是 Python。Hy 直接使用 Python 的对象，所以对于经验丰富的 Lisp 开发者可能会对基础对象的行为感到惊讶。

希望人们可以透过 Hy 的语法，并考虑将其利用到项目中，从而拓展其视野，并探索 Python 中一些之前未曾接触的部分。

第 10 章 性能与优化

“过早地优化是万恶之源。”

——Donald Knuth，摘自 *Structured Programming with go to Statements*

10.1 数据结构

如果使用正确的数据结构，大多数计算机问题都能以一种优雅而简单的方式解决，而 Python 就恰恰提供了很多可供选择的数据结构。

通常，有一种诱惑是实现自定义的数据结构，但这必然是徒劳无功、注定失败的想法。因为 Python 总是能够提供更好的数据结构和代码，要学会使用它们。

例如，每个人都会用字典，但你看到过多少次这样的代码：

```
def get_fruits(basket, fruit):
    # A variation is to use "if fruit in basket:"
    try:
        return basket[fruit]
    except KeyError:
        return set()
```

最好是使用 `dict` 结构已经提供的 get 方法。

```
def get_fruits(basket, fruit):
    return basket.get(fruit, set())
```

使用基本的 Python 数据结构但不熟悉它提供的所有方法并不罕见。这也同样适用于集合的使用。例如：

```
def has_invalid_fields(fields):
```

```
    for field in fields:
        if field not in ['foo', 'bar']:
            return True
    return False
```

这可以不用循环实现：

```
def has_invalid_fields(fields):
    return bool(set(fields) - set(['foo', 'bar']))
```

set 数据结构包含许多能解决不同问题的方法，否则这些问题需要通过嵌套的 for/if 块才能实现。

还有许多高级的数据结构可以极大地减少代码维护负担。例如，可以看看下面的代码：

```
def add_animal_in_family(species, animal, family):
    if family not in species:
        species[family] = set()
    species[family].add(animal)

species = {}
add_animal_in_family(species, 'cat', 'felidea')
```

当然，这段代码是完全有效的，但想想看你会在你的程序中需要多少次上面代码的变种？10 次？100 次？

Python 提供的 collections.defaultdict 结构可以更优雅地解决这个问题。

```
import collections

def add_animal_in_family(species, animal, family):
    species[family].add(animal)

species = collections.defaultdict(set)
add_animal_in_family(species, 'cat', 'felidea')
```

每次试图从字典中访问一个不存在的元素，defaultdict 都会使用作为参数传入的这个函数去构造一个新值而不是抛出 KeyError。在这个例子，set 函数被用来在每次需要时构造一个新的集合。

此外，collections 模块提供了一些新的数据结构用来解决一些特定问题，如 OrderedDict 或者 Counter。

在 Python 中找到正确的数据结构是非常重要的，因为正确的选择会节省你的时间并减少代码维护量。

10.2 性能分析

Python 提供了一些工具对程序进行性能分析。标准的工具之一就是 `cProfile`，而且它很容易使用，如示例 10.1 所示。

示例 10.1 使用 `cProfile` 模块

```
$ python -m cProfile myscript.py
        343 function calls (342 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
        1    0.000    0.000    0.000    0.000  :0(_getframe)
        1    0.000    0.000    0.000    0.000  :0(len)
      104    0.000    0.000    0.000    0.000  :0(setattr)
        1    0.000    0.000    0.000    0.000  :0(setprofile)
        1    0.000    0.000    0.000    0.000  :0(startswith)
      2/1    0.000    0.000    0.000    0.000  <string>:1(<module>)
        1    0.000    0.000    0.000    0.000  StringIO.py:30(<module>)
        1    0.000    0.000    0.000    0.000  StringIO.py:42(StringIO)
```

运行结果的列表显示了每个函数的调用次数，以及执行所花费的时间。可以使用`-s` 选项按其他字段进行排序，例如，`-s time` 可以按内部时间进行排序。

如果你像我一样使用 C 语言很多年，那你很可能已经知道 Valgrind（http://valgrind.org/）这个优秀的工具，除了其他功能之外，它能够提供对 C 程序的性能分析数据。生成的数据能够被另一个不错的工具 KCacheGrind（http://kcachegrind.sourceforge.net/html/Home.html）可视化地展示。

`cProfile` 生成的性能分析数据很容易转换成一个可以被 KCacheGrind 读取的调用树。`cProfile` 模块有一个`-o` 选项允许保存性能分析数据，并且 `pyprof2calltree`（https://pypi.python.org/pypi/pyprof2calltree）可以进行格式转换，如示例 10.2 所示。

示例 10.2　用 KCacheGrind 可视化 Python 性能分析数据

```
$ python -m cProfile -o myscript.cprof myscript.py
$ pyprof2calltree -k -i myscript.cprof
```

这可以提供很多有用的信息，让你可以判断程序的哪个部分耗费了太多的资源，如图 10-1 所示。

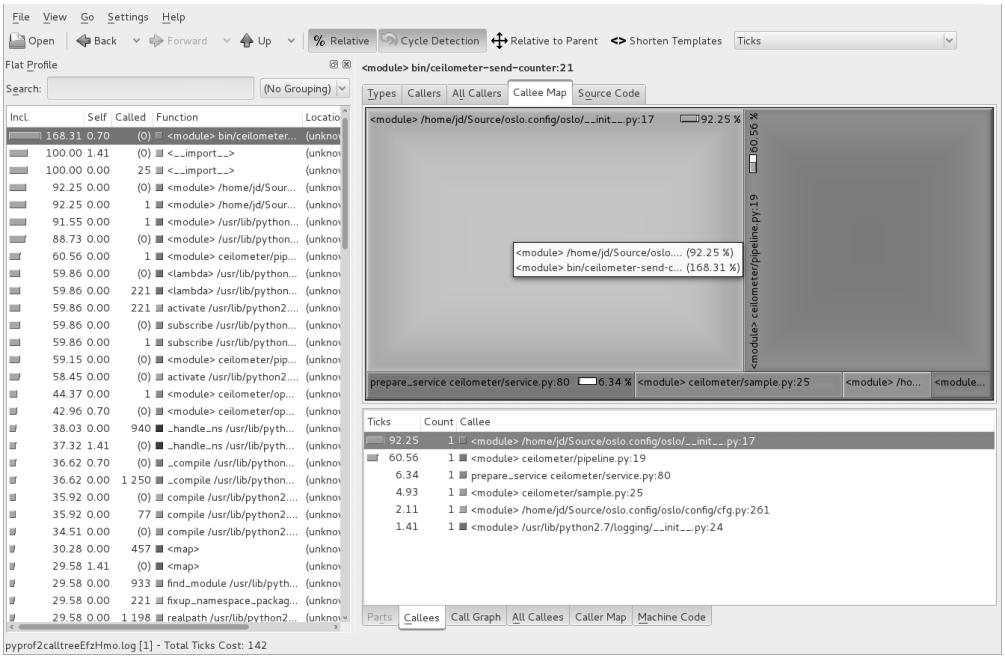


图 10-1　KCacheGrind 示例

虽然从宏观角度看这么用没问题，它有时也可以对代码的某些部分提供一些微观角度的分析。但这样的上下文中，我发现用 `dis` 模块可以看到一些隐藏的东西。`dis` 模块是 Python 字节码的反编译器，用起来也很简单。

```
>>> def x():
...     return 42
...
>>> import dis
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
```

dis.dis 函数会反编译作为参数传入的函数，并打印出这个函数运行的字节码指令的清单。为了能适当地优化代码，这对于理解程序的每行代码非常有用。

下面的代码定义了两个函数，功能相同，都是拼接三个字母。

```
abc = ('a', 'b', 'c')

def concat_a_1():
    for letter in abc:
           abc[0] + letter

def concat_a_2():
    a = abc[0]
    for letter in abc:
           a + letter
```

两者看上去作用一样，但如果反汇编它们的话，可以看到生成的字节码有点儿不同。

```
>>> dis.dis(concat_a_1)
  2          0 SETUP_LOOP              26 (to 29)
             3 LOAD_GLOBAL              0 (abc)
             6 GET_ITER
        >>   7 FOR_ITER                18 (to 28)
            10 STORE_FAST               0 (letter)

  3         13 LOAD_GLOBAL              0 (abc)
            16 LOAD_CONST               1 (0)
            19 BINARY_SUBSCR
            20 LOAD_FAST                0 (letter)
            23 BINARY_ADD
            24 POP_TOP
            25 JUMP_ABSOLUTE            7
        >>  28 POP_BLOCK
        >>  29 LOAD_CONST               0 (None)
            32 RETURN_VALUE
```

```
>>> dis.dis(concat_a_2)
  2          0 LOAD_GLOBAL              0 (abc)
             3 LOAD_CONST               1 (0)
             6 BINARY_SUBSCR
             7 STORE_FAST               0 (a)
```

```
  3         10 SETUP_LOOP              22 (to 35)
            13 LOAD_GLOBAL              0 (abc)
            16 GET_ITER
       >>   17 FOR_ITER                14 (to 34)
            20 STORE_FAST               1 (letter)

  4         23 LOAD_FAST                0 (a)
            26 LOAD_FAST                1 (letter)
            29 BINARY_ADD
            30 POP_TOP
            31 JUMP_ABSOLUTE           17
       >>   34 POP_BLOCK
       >>   35 LOAD_CONST               0 (None)
            38 RETURN_VALUE
```

如你所见，在函数的第二个版本中运行循环之前我们将 `abc[0]`保存在了一个临时变量中。这使得循环内部执行的字节码稍微短一点，因为不需要每次迭代都去查找 `abc[0]`。通过 `timeit` 测量，第二个版本的函数比第一个要快 10%，少花了不到一微秒。显然，除非调用这个函数 100 万次，否则不值得优化，但这就是 `dis` 模块所能提供的洞察力。

是否应该依赖将值存储在循环外这样的“技巧”是有争议的，这类优化工作应该最终由编译器完成。但是，由于 Python 语言是高度动态的，因此编译器很难确保优化不会产生什么副作用。所以，编写代码一定要小心。

另一个我在评审代码时遇到的错误习惯是无理由地定义嵌套函数（分解嵌套函数见示例 10.3）。这实际是有开销的，因为函数会无理由地被重复定义。

示例 10.3　分解嵌套函数

```
>> import dis
>>> def x():
...     return 42
...
>>> dis.dis(x)
  2          0 LOAD_CONST               1 (42)
             3 RETURN_VALUE
>>> def x():
...     def y():
```

```
...           return 42
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (<code object y at 0x100ce7e30, file
    "<stdin>", line 2>)
              3 MAKE_FUNCTION            0
              6 STORE_FAST               0 (y)

  4           9 LOAD_FAST                0 (y)
             12 CALL_FUNCTION            0
             15 RETURN_VALUE
```

可以看到函数被不必要地复杂化了，调用 MAKE_FUNCTION、STORE_FAST、LOAD_FAST 和 CALL_FUNCTION，而不是直接调用 LOAD_CONST，这无端造成了更多的操作码，而函数调用在 Python 中本身就是低效的。

唯一需要在函数内定义函数的场景是在构建函数闭包的时候，它可以完美地匹配 Python 的操作码中的一个用例。反汇编一个闭包如示例 10.4 所示。

示例 10.4 反汇编一个闭包

```
>>> def x():
...     a = 42
...     def y():
...           return a
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 STORE_DEREF              0 (a)

  3           6 LOAD_CLOSURE             0 (a)
              9 BUILD_TUPLE              1
             12 LOAD_CONST               2 (<code object y at 0x100d139b0, file
    "<stdin>", line 3>)
             15 MAKE_CLOSURE             0
             18 STORE_FAST               0 (y)
```

```
  5          21 LOAD_FAST                0 (y)
             24 CALL_FUNCTION            0
             27 RETURN_VALUE
```

10.3　有序列表和二分查找

当处理大的列表时，有序列表比非有序列表有一定优势。例如，有序列表的元素获取时间为 O(log n)。

但是有很多次，我看到有人试图实现自己的数据结构或算法去处理这样的场景。这是个糟糕的想法，没必要花时间在已经解决的问题上。

首先，Python 提供了一个 `bisect` 模块，其包含了二分查找算法。非常容易使用，如示例 10.5 所示。

示例 10.5　`bisect` 的用法

```
>>> farm = sorted(['haystack', 'needle', 'cow', 'pig'])
>>> bisect.bisect(farm, 'needle')
3
>>> bisect.bisect_left(farm, 'needle')
2
>>> bisect.bisect(farm, 'chicken')
0
>>> bisect.bisect_left(farm, 'chicken')
0
>>> bisect.bisect(farm, 'eggs')
1
>>> bisect.bisect_left(farm, 'eggs')
1
```

`bisect` 函数能够在保证列表有序的情况下给出要插入的新元素的索引位置。

如果想要立即插入，可以使用 `bisect` 模块提供的 `insort_left` 和 `insort_right` 函数，如示例 10.6 所示。

示例 10.6　`bisect.insort` 的用法

```
>>> farm
['cow', 'haystack', 'needle', 'pig']
```

```
>>> bisect.insort(farm, 'eggs')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig']
>>> bisect.insort(farm, 'turkey')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig', 'turkey']
```

可以使用这些函数创建一个一直有序的列表，如示例 10.7 所示。

示例 10.7 SortedList 的实现

```
import bisect

class SortedList(list):
    def __init__(self, iterable):
        super(SortedList, self).__init__(sorted(iterable))

    def insort(self, item):
        bisect.insort(self, item)

    def index(self, value, start=None, stop=None):
        place = bisect.bisect_left(self[start:stop], value)
        if start:
            place += start
        end = stop or len(self)
        if place < end and self[place] == value:
            return place
        raise ValueError("%s is not in list" % value)
```

显然，不应该用直接的 append 或 extend 函数来追加或扩展这个列表，否则列表将不再有序。

此外还有许多 Python 库实现了上面代码的各种不同版本，以及更多的数据类型，如二叉树和红黑树。Python 包 blist（https://pypi.python.org/pypi/blist）和 bintree（https://pypi.python.org/pypi/bintrees/）就包含了用于这些目的的代码，不要开发和调试自己的版本。

10.4 namedtuple 和 slots

有时能创建只拥有一些固定属性的简单对象是非常有用的。一个简单的实现可能需要下

面这几行代码：

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

这肯定可以满足需求。但是，这种方法的缺点就是它创建了一个继承自 object 的类。在使用这个 Point 类时，需要实例化对象。

Python 中这类对象的特性之一就是会存储所有的属性在一个字典内，这个字典本身被存在__dict__属性中：

```
>>> p = Point(1, 2)
>>> p.__dict__
{'y': 2, 'x': 1}
>>> p.z = 42
>>> p.z
42
>>> p.__dict__
{'y': 2, 'x': 1, 'z': 42}
```

好处是可以给一个对象添加任意多的属性。缺点是通过字典来存储这些属性内存方面的开销很大，要存储对象、键、值索引等。创建慢，操作也慢，并且伴随着高内存开销。看看下面这个简单的类。

```
[source, python]
class Foobar(object):
    def __init__(self, x):
        self.x = x
```

我们通过 Python 包 memory_profiler 来检测一下内存使用情况：

```
$ python -m memory_profiler object.py
Filename: object.py

Line #    Mem usage    Increment    Line Contents
================================================
     5                             @profile
     6    9.879 MB     0.000 MB    def main():
     7   50.289 MB     0.410 MB    f = [ Foobar(42) for i in range(100000) ]
```

因此，使用对象但不使用这个默认行为的方式是存在的。Python 中的类可以定义一个

__slots__属性，用来指定该类的实例可用的属性。其作用在于可以将对象属性存储在一个 list 对象中，从而避免分配整个字典对象。如果浏览一下 CPython 的源代码并且看看 Objects/typeobject.c 文件，就很容易理解这里 Python 做了什么。下面给出了相关处理函数的部分代码：

```
static PyObject *
type_new(PyTypeObject *metatype, PyObject *args, PyObject *kwds)
{
    [...]
    /* Check for a __slots__ sequence variable in dict, and count it */
    slots = _PyDict_GetItemId(dict, &PyId___slots__);
    nslots = 0;
    if (slots == NULL) {
        if (may_add_dict)
            add_dict++;
        if (may_add_weak)
            add_weak++;
    }
    else {
        /* Have slots */
        /* Make it into a tuple */
        if (PyUnicode_Check(slots))
            slots = PyTuple_Pack(1, slots);
        else
            slots = PySequence_Tuple(slots);
        /* Are slots allowed? */
        nslots = PyTuple_GET_SIZE(slots);
        if (nslots > 0 && base->tp_itemsize != 0) {
            PyErr_Format(PyExc_TypeError,
                         "nonempty __slots__ "
                         "not supported for subtype of '%s'",
                         base->tp_name);
            goto error;
        }
        /* Copy slots into a list, mangle names and sort them.
           Sorted names are needed for __class__ assignment.
           Convert them back to tuple at the end.a
        */
```

```
        newslots = PyList_New(nslots - add_dict - add_weak);
        if (newslots == NULL)
            goto error;
        if (PyList_Sort(newslots) == -1) {
            Py_DECREF(newslots);
            goto error;
        }
        slots = PyList_AsTuple(newslots);
        Py_DECREF(newslots);
        if (slots == NULL)
            goto error;
    }
    /* Allocate the type object */
    type = (PyTypeObject *)metatype->tp_alloc(metatype, nslots);
    [...]
    /* Keep name and slots alive in the extended type object */
    et = (PyHeapTypeObject *)type;
    Py_INCREF(name);
    et->ht_name = name;
    et->ht_slots = slots;
    slots = NULL;
    [...]
    return (PyObject *)type;
```

正如你所看到的，Python 将__slots__的内容转化为一个元组，构造一个 list 并排序，然后再转换回元组并存储在类中。这样 Python 就可以快速地抽取值，而无需分配和使用整个字典。

声明这样一个类并不难，如示例 10.8 所示。

示例 10.8　使用__slots__的类声明

```
class Foobar(object):
    __slots__ = 'x'

    def __init__(self, x):
        self.x = x
```

可以很容易地通过 memory_profiler 比较两种方法的内存占用情况，如示例 10.9 所示。

示例 10.9 使用了__slots__的对象的内存占用

```
% python -m memory_profiler slots.py
Filename: slots.py

Line #    Mem usage    Increment    Line Contents
================================================
     7                             @profile
     8    9.879 MB     0.000 MB    def main():
     9   21.609 MB    11.730 MB    f = [ Foobar(42) for i in range(100000) ]
```

看似通过使用 Python 类的__slots__属性可以将内存使用率提升一倍，这意味着在创建大量的简单对象时使用__slots__属性是有效且高效的选择。但这项技术不应该被滥用于静态类型或其他类似场合，那不是 Python 程序的精神所在。

由于属性列表的固定性，因此不难想象类中列出的属性总是有一个值，且类中的字段总是按某种方式排过序的。

这也正是 **collection** 模块中 **namedtuple** 类的本质。它允许动态创建一个继承自 **tuple** 的类，因而有着共有的特征，如不可改变，条目数固定。namedtuple 所提供的能力在于可以通过命名属性获取元组的元素而不是通过索引，如示例 10.10 所示。

示例 10.10 用 namedtuple 声明类

```
>>> import collections
>>> Foobar = collections.namedtuple('Foobar', ['x'])
>>> Foobar = collections.namedtuple('Foobar', ['x', 'y'])
>>> Foobar(42, 43)
Foobar(x=42, y=43)
>>> Foobar(42, 43).x
42
>>> Foobar(42, 43).x = 44
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> Foobar(42, 43).z = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foobar' object has no attribute 'z'
>>> list(Foobar(42, 43))
```

```
[42, 43]
```

因为这样的类是继承自 tuple 的，因此可以很容易将其转换为 list。但不能添加或修改这个类的对象的任何属性，因为它继承自 tuple 同时也因为__slots__的值被设置成了一个空元组以避免创建__dict__。基于 collections.namedtuple 构建的类的内存占用如示例 10.11 所示。

示例 10.11　基于 collections.namedtuple 构建的类的内存占用

```
% python -m memory_profiler namedtuple.py
Filename: namedtuple.py

Line #    Mem usage    Increment   Line Contents
================================================
     4                             @profile
     5     9.895 MB     0.000 MB   def main():
     6    23.184 MB    13.289 MB       f = [ Foobar(42) for i in range(100000) ]
```

因此，namedtuple 类工厂的使用同使用带有__slots__的对象一样有效，唯一的不同在于它同 tuple 类兼容。因此，它可以作为参数传入任何期望 iterable 类型参数的原生 Python 函数。同时它也享有已有的针对元组的优化。[①]

namedtuple 还提供了一些额外的方法，尽管以下划线作为前缀，但实际上是可以公开访问的。_asdict 可以将 namedtuple 转换为字典实例，_make 可以转换已有的 iterable 对象为 namedtuple，_replace 替换某些字段后返回一个该对象的新实例。

10.5　memoization

memoization 是指通过缓存函数返回结果来加速函数调用的一种技术。仅当函数是纯函数时结果才可以被缓存，也就是说函数不能有任何副作用或输出，也不能依赖任何全局状态。

正弦函数 sin 就是一个可以用来 memoize 化的函数，如示例 10.12 所示。

示例 10.12　基本的 memoization 技术

```
>>> import math
>>> _SIN_MEMOIZED_VALUES = {}
```

① 例如，小于 PyTuple_MAXSAVESIZE（默认是 20）的元组在 CPython 中会使用更快的内存分配器。

```
>>> def memoized_sin(x):
...    if x not in _SIN_MEMOIZED_VALUES:
...        _SIN_MEMOIZED_VALUES[x] = math.sin(x)
...    return _SIN_MEMOIZED_VALUES[x]
>>> memoized_sin(1)
0.84 14709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965}
>>> memoized_sin(2)
0.90 92974268256817
>>> memoized_sin(2)
0.90 92974268256817
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
>>> memoized_sin(1)
0.84 14709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
```

在第一次用参数调用 `memoized_sin` 时还没有值存在`_SIN_MEMOIZED_VALUES` 中，因此需要计算这个值并将其存储在字典中。之后，如果再次以相同的参数值调用这个函数，结果将从这个字典中获取而不需要重新计算。尽管 `sin` 函数本身计算非常快，但是这对涉及更复杂计算的高级函数并不成立。

如果已经了解了装饰器（参见 7.1 节），肯定可以想到装饰器用在这里正合适，这么想完全正确。PyPI 包含了一些通过装饰器实现的 memoization，从简单场景到最复杂且最完备的情况都有覆盖。

从 Python 3.3 开始，**`functools`** 模块提供了一个 LRU（Least-Recently-Used）缓存装饰器。它提供了同此处描述的 memoization 完全一样的功能，其优势在于限定了缓存的条目数，当缓存的条目数达到最大时会移除最近最少使用的条目。

该模块还提供了对缓存命中、缺失等的统计。在我看来，对于缓存来说它们都是必备的实现。如果不能对缓存的使用和效用进行衡量，那么使用 memoization 是毫无意义的。

示例 10.13 是将上面的 `memoized_sin` 函数的示例用 `functools.lru_cache` 改写后的。

示例 10.13　使用 **functools.lru_cache**

```
>>> import functools
>>> import math
>>> @functools.lru_cache(maxsize=2)
... def memoized_sin(x):
...     return math.sin(x)
...
>>> memoized_sin(2)
0.90  92974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(2)
0.90  92974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(3)
0.14  11200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=2, maxsize=2, currsize=2)
>>> memoized_sin(4)
-0.7568024953079282
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=3, maxsize=2, currsize=2)
>>> memoized_sin(3)
0.14 11200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=2, misses=3, maxsize=2, currsize=2)
>>> memoized_sin.cache_clear()
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=0, maxsize=2, currsize=0)
```

10.6　PyPy

PyPy（http://pypy.org/）是符合标准的 Python 语言的一个高效实现。实际上，现在最权威的 Python 实现 CPython（这么叫是因为它是用 C 语言写的）有可能非常慢。PyPy 的目的是要用 Python 写一个 Python 解释器。随着时间的推移，现在在用 RPython 编写，RPython

是 Python 语言的一个限制性子集。

RPython 对 Python 语言的限制的主要方式是，要求变量类型能够在编译时推断。RPython 的代码会被翻译成 C 代码从而构建解释器，当然 RPython 也可以用来实现其他语言而不只是 Python。

除了技术上的挑战，PyPy 吸引人的地方在于目前它是 CPython 的更快的替代品。PyPy 包含内置的 JIT（Just-In-Time）编译器。简单来说，就是通过利用解释的灵活性对编译后的代码的速度进行整合从而运行得更快。

到底多快呢？看情况，但对于纯算法代码会更快一点。对于普通的代码，大多数情况下 PyPy 声称可以达到 3 倍的速度。尽管如此，也不要期望太高，PyPy 同样有一些 CPython 的局限性，如可恶的 GIL（Global Interpreter Lock，全局解释器锁）。

尽管 PyPy 并非一种严格意义上的优化技术，但是将其作为一种支持的 Python 实现还是不错的。达到这一目标只需要与支持的其他 Python 版本保持同样的编码策略，基本上，只需要保证像在 CPython 上一样在 PyPy 上测试你的软件。`tox`（参见 6.7 节）支持使用 PyPy 构造虚拟环境，就像 CPython 2 和 CPython 3 一样，所以实现起来还是相当简单的。

如果想让你的软件运行在 PyPy 上，最好是在项目的初期就开始，以避免在后期支持时可能带来的大量工作。

注意

Hy 项目从项目初期就成功地采用了这一策略。Hy 一直支持 PyPy 和其他所有 Python 版本，且没有任何问题。但是，我们却没能在所有 OpenStack 项目中这样做，我们正在被一些由于各种原因不能在 PyPy 上运行的代码路径和依赖所阻碍，因为它们没有在项目的早期进行充分地测试。

PyPy 与 Python 2.7 兼容，并且它的 JIT 编译器可以运行在 32 位和 64 位 x86 和 ARM 体系结构上，并且可以运行在不同的操作系统上（Linux、Windows、Mac OS X 等）。其对 Python 3 的支持正在开发中。

10.7 通过缓冲区协议实现零复制

通常程序都需要处理大量的大型字节格式的数组格式的数据。一旦进行复制、分片和修

改等操作，以字符串的方式处理如此大量的数据是非常低效的。

设想一个读取二进制数据的大文件的小程序，并将其部分复制到另一个文件中。这里将使用 memory_profiler（https://pypi.python.org/pypi/memory_profiler）衡量内存的使用情况，memory_profiler 是一个不错的 Python 包，可以用来逐行查看程序的内存使用情况。

```
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
        content = source.read(1024 * 10000)
        content_to_write = content[1024:]
    print("Content length: %d, content to write length %d" %
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
        target.write(content_to_write)

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

接下来使用 memory_profiler 运行上面的程序：

```
$ python -m memory_profiler memoryview/copy.py
Content length: 10240000, content to write length 10238976
Filename: memoryview/copy.py

Mem usage    Increment   Line Contents
================================================
                     @profile
 9.883 MB    0.000 MB   def read_random():
 9.887 MB    0.004 MB       with open("/dev/urandom", "rb") as source:
19.65 6 MB   9.770 MB         content = source.read(1024 * 10000)          ❶
29.42 2 MB   9.766 MB         content_to_write = content[1024:]            ❷
29.42 2 MB    0.000 MB       print("Content length: %d, content to write
length %d" %
29.43 4 MB    0.012 MB             (len(content), len(content_to_write)))
29.43 4 MB    0.000 MB       with open("/dev/null", "wb") as target:
29.43 4 MB    0.000 MB           target.write(content_to_write)
```

❶ 从/dev/urandom 读取 10 MB 的数据且没有太多其他操作。Python 为此要分配约 10 MB 的内存以将该数据存储为字符串。

❷ 复制整块的数据但是减去开始的 1 KB，因为我们不想将最开始的 1 KB 数据写入目标文件中。

这个例子中有意思的地方在于，正如你看到的，内存的使用在构造变量 content_to_write 时增长到了约 10 MB。事实上，分片操作符会复制全部的内容，减去开始的 1 KB 后写入一个新的字符串对象中。

当处理大量数据时，针对大的字节数组执行此类操作可能会变成一场灾难。如果写过 C 语言代码，应该知道使用 memcpy()的开销是巨大的，无论是内存的占用还是对通常意义上的性能来说，复制内存都是缓慢的。

但是作为 C 程序员，你应该也知道字符串是字符的数组，完全可以通过基本的指针算法的使用查看数组的某一部分但不复制数组[①]。

在 Python 中可以使用实现了**缓冲区协议**的对象。PEP 3118（https://www.python.org/dev/peps/pep-3118/）定义了缓冲区协议，其中解释了用于为不同数据类型（如字符串类型）提供该协议的 C API。

对于实现了该协议的对象，可以使用其 memoryview 类的构建函数去构造一个新的 **memoryview** 对象，它会引用原始的对象内存。

示例如下：

```
>>> s = b"abcdefgh"
>>> view = memoryview(s)
>>> view[1]
98 ❶
>>> limited = view[1:3]
<memory at 0x7fca18b8d460>
>>> bytes(view[1:3])
b'bc'
```

❶ 字母 b 的 ASCII 码。

在这个例子中，会利用 memoryview 对象的切片运算符本身返回一个 memoryview 对象的事实。这意味着它不会复制任何数据，而只是引用了原始数据的一个特定分片，如

① 假设整个字符串在一个连续的内存区域。

图 10-2 所示。

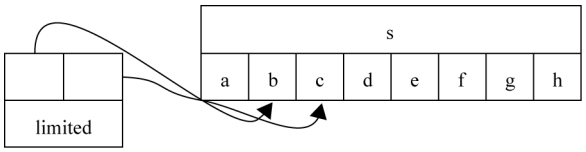


图 10-2　对 memoryview 对象使用切片

出于这一点考虑，我们可以重写这个程序，这次对数据的引用将使用 memoryview 对象。

```
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
        content = source.read(1024 * 10000)
        content_to_write = memoryview(content)[1024:]
    print("Content length: %d, content to write length %d" %
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
        target.write(content_to_write)

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

这个程序只使用了第一个版本约一半的内存：

```
$ python -m memory_profiler memoryview/copy-memoryview.py
Content length: 10240000, content to write length 10238976
Filename: memoryview/copy-memoryview.py

 Mem usage   Increment   Line Contents
=======================================
                        @profile
   9.887 MB    0.000 MB   def read_random():
   9.891 MB    0.004 MB   with open("/dev/urandom", "rb") as source:
19.66 0 MB    9.770 MB   content = source.read(1024 * 10000) ❶
 19.66 0 MB    0.000 MB   content_to_write = memoryview(content)[1024:] ❷
```

```
    19.66 0 MB    0.000 MB   print("Content length: %d, content to write
length %d" %
    19.67 2 MB    0.012 MB    (len(content), len(content_to_write)))
    19.67 2 MB    0.000 MB   with open("/dev/null", "wb") as target:
    19.67 2 MB    0.000 MB   target.write(content_to_write)
```

❶ 从/dev/urandom 读取 10 MB 的数据且没有太多其他操作。Python 为此要分配约 10 MB 的内存以将该数据存储为字符串。

❷ 直接引用整块数据减去开始的 1 KB 的数据，因为我们不想将最开始的 1 KB 数据写入目标文件中。没有复制意味着没有额外的内存开销。

当处理 socket（套接字）时这类技巧尤其有用。如你所知，当数据通过 socket 发送时，它不会在一次调用中发送所有数据。下面是一个简单的实现：

```
import socket
s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000)     ❶
while data:
    sent = s.send(data)
    data = data[sent:]            ❷
```

❶ 构建一个字节对象，包含 1 亿多个字母 a。

❷ 移除前面已经发送的字节。

显然，通过这种机制，需要不断地复制数据，直到 socket 将所有数据发送完毕。而使用 memoryview 可以实现同样的功能而无需复制数据，也就是零复制。

```
import socket
s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000)        ❶
mv = memoryview(data)
while mv:
    sent = s.send(mv)
    mv = mv[sent:]                   ❷
```

❶ 构建一个字节对象，包含 1 亿多个字母 a。

❷ 构建一个新的 memoryview 对象，指向剩余将要发送的数据。

这段程序不会复制任何内容，不会使用额外的内存，也就是说只是像开始时那样要给变

量分配 100 MB 内存。

前面已经看到了将 memoryview 对象用于高效地写数据的场景，同样的方法也可以用在读数据时。Python 中的大部分 I/O 操作知道如何处理实现了缓冲区协议的对象。它们不仅可以从这类对象中读，还可以向其中写。在这个例子中不需要 memoryview 对象，只需要让 I/O 函数写入预分配的对象。

```
>>> ba = bytearray(8)
>>> ba
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba)
...
8
>>> ba
bytearray(b'`m.z\x8d\x0fp\xa1')
```

利用这一技术，很容易预先分配一个缓冲区（就像在 C 语言中一样，以减少对 malloc() 的调用次数）并在需要时进行填充。使用 memoryview 甚至可以在内存区域的任何点放入数据。

```
>>> ba = bytearray(8)
>>> ba_at_4 = memoryview(ba)[4:]                          ❶
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba_at_4)                          ❷
...
4
>>> ba
bytearray(b'\x00\x00\x00\x00\x0b\x19\xae\xb2')
```

❶ 引用 bytearray，从其偏移索引 4 到其结尾。

❷ 将/dev/urandom 的内容写入 bytearray 中从偏移索引 4 到结尾的位置，精确且高效地只读了 4 字节。

提示

array 模块中的对象以及 struct 模块中的函数都能正确地处理缓冲区协议，因此在需要零复制时都能高效完成。

10.8 Victor Stinner 访谈

Victor 是资深的 Python 黑客，许多 Python 模块的核心贡献者和作者。他最近撰写了 PEP 454（https://www.python.org/dev/peps/pep-0454/），其中提出了一个新的 `tracemalloc` 模块，用于在 Python 中跟踪内存块的分配，并写了一个简单的 AST 优化器。

优化 Python 代码的一个初步策略是什么？

针对 Python 的策略其实和在其他语言中一样。首先需要定义良好的用例，以得到一个稳定可重现的基准。没有可靠基准的情况下尝试不同的优化方法很可能导致时间的浪费和不成熟的优化。无用的优化可能使代码更糟，更不易懂，甚至更慢。有用的优化必须至少让程序加速 5%。

如果发现代码的某个部分比较“慢”，那么需要针对这段代码设计一个基准测试。对于较短的函数的基准测试通常称为“微基准测试”。通过微基准测试衡量优化效果时，速度提升应该至少达到 20%或者 25%。

在不同的计算机、不同的操作系统甚至不同的编译器上运行同一个基准测试会很有意思。例如，函数 `realloc()` 的性能在 Linux 和 Windows 上是不同的。有时候针对不同的平台的代码实现也可能会不同，尽管这是应该尽量避免的。

关于 Python 代码的性能分析和优化有许多不同的工具，你最喜欢的是哪个？

Python 3.3 提供了一个新的 `time.perf_counter()` 函数，用来为基准测试衡量已耗

用时间。它是最好的解决方案。

测试应该运行不止一次，最少 3 次，5 次基本够了。重复测试可以填充磁盘缓存和 CPU 缓存。我倾向于保证最小时间，其他一些开发人员则倾向于使用几何平均值。

对于微基准测试，`timeit` 模块简单易用且能很快得到结果，但使用默认的参数结果并不稳定。应该手工重复测试，以得到稳定的结果。

优化是非常花时间的，所以最好能专注那些耗费最多 CPU 的函数。为了找到这些函数，Python 提供了 `cProfile` 和用来记录每个函数时间消耗的 `profile` 模块。

能够改进性能的最有意思的 Python 技巧是什么？

应该尽可能重用标准库。它们经过良好的测试并且通常都很高效。Python 内置的类型都是用 C 实现的，所以性能都很好。应使用正确的容器以得到最佳的性能，Python 提供许多不同的容器，如 `dict`、`list`、`deque`、`set` 等。

也有一些用来优化 Python 的非常手段，但是应该避免使用它们，因为这一点点的速度提升会丧失代码的可读性。

Python 之禅（PEP 20）说："应该有一种——最好只有一种——显而易见的方式去实现。"实际上，写 Python 代码有很多不同的方式，且性能各异，所以只能信赖针对特定用例的基准测试。

在哪些领域中 Python 的性能很差？哪些领域中应该小心使用？

通常，在开发新的应用程序时我不太担心性能问题。不成熟的优化是所有问题之源。当找到了缓慢的函数时，应该修改算法。如果算法和容器都是经过仔细挑选的，那么可以考虑用 C 语言重写短函数以获得更好的性能。

CPython 的一个众所周知的性能瓶颈是全局解释器锁（Global Interpreter Lock，GIL）。两个线程不能在同时执行 Python 字节码。然而，这个限制只在两个线程执行纯 Python 代码时有影响。如果大多数处理时间花在函数调用上，并且这些函数释放了 GIL，那么 GIL 并非性能瓶颈。例如，大多数 I/O 函数都会释放 GIL。

`multiprocessing` 模块可以很容易地用来绕过 GIL。另一个稍微复杂的方式是编写异步代码。Twisted、Tornado 和 Tulip 都是利用了这一技术的面向网络的库。

你见过最多的导致性能差的“错误”是什么？

没有很好地理解 Python 就可能写出效率低的代码。例如，我见过在不需要复制时错误地使用了 `copy.deepcopy()`。

另一个性能杀手是低效的数据结构。少于 100 个元素的情况下，容器类型对性能没有影响。对于更多元素的场景，应该了解每个操作（`add`、`get`、`delete`）的复杂性和影响。

第 11 章 扩展与架构

如今所有的炒作都是有关灵活性和可扩展性的，因此我假设这是你的开发流程中早晚都要考虑的东西。这个问题的很多方面并非 Python 本身特有的，许多知识和其主要实现（即 CPython）有关。

一个应用程序的可扩展性、并发性和并行性在很大程度上取决于它的初始架构和设计的选择。如你所见，有一些范例（如多线程）在 Python 中被误用，而其他一些技术（如面向服务架构）可以产生更好的效果。

11.1 多线程笔记

什么是多线程？它是在一个 Python 进程中将代码运行在不同的处理器上[①]的能力。这意味着代码的不同部分可以并行运行。

为什么需要多线程呢？最常见的场景如下。

（1）需要运行后台任务但不希望停止主线程的执行。例如，在图形用户界面场景下，主循环需要等待对事件的响应。

（2）需要将工作负载分布在几个 CPU 上。

所以，刚开始要解决这些问题的话，多线程似乎是扩展和并行化应用程序的好办法。当需要分散工作负载时，只需要为新的请求启动一个新线程而不用一次只能处理一个。

太棒了，搞定！让我们继续。

不，很抱歉！首先，如果你已经在 Python 领域中混了很久，那么你肯定遇到过 GIL 这

① 或者如果多个 CPU 不存在的话，依次在某一个处理器上。

个词，而且知道它多么讨厌。GIL 是指 Python 全局解释锁（Global Interpreter Lock），当 CPython[①]每次要执行字节码时都要先申请这个锁。但是，这意味着，如果试图通过多线程扩展应用程序，将总是被这个全局锁所限制。

所以尽管多线程看上去是一个理想的解决方案，但实际上我看到的大多数应用程序都很难获取到 150%的 CPU 利用率，也就是使用 1.5 个核（core）。考虑到现如今计算节点通常至少有 2 个或 4 个核，这是很没面子的。这都归咎于 GIL。

目前没有任何工作试图从 CPython 中移除 GIL，因为考虑到实现和维护的难度大家都觉得不值得这么做。

然而，CPython 只是 Python 的可用实现之一[②]。例如，Jython（http://www.jython.org/）就没有全局解释锁（http://www.jython.org/jythonbook/en/1.0/Concurrency.html），这意味着它可以有效地并行运行多个线程。遗憾的是，这些项目相对于 CPython 都非常滞后，所以实际上并不能作为目标平台来使用。

注意

PyPy 是另一个 Python 实现，但是是使用 Python 开发的（参见 10.6 节）。PyPy 也有 GIL，但目前有一个非常有意思的工作正在试图用基于 STM（Software Transactional Memory，http://www.jython.org/jythonbook/en/1.0/Concurrency.html）的实现替换它。这对于未来构建和运行多线程软件是非常值得期待的变化。某些处理器正在试图提供硬件支持，而 Linux 内核的开发者也在寻求废弃内核锁的方法。这些都是积极的信号。

没有好的方案是不是我们又回到了最初的场景呢？并非如此，至少还有以下两种方案可用。

（1）如果需要运行后台任务，最容易的方式是基于事件循环构建应用程序。许多不同的 Python 模块都提供这一机制，甚至有一个标准库中的模块——`asyncore`，它是 PEP 3156（https://www.python.org/dev/peps/pep-3156/）中标准化这一功能的成果。有些框架就是基于这一概念构建的，如 Twisted（http://twistedmatrix.com/trac/）。最高级的框架应该提供基于信号量、计时器和文件描述符活动来访问事件，我们将在 11.3 节中进行讨论。

（2）如果需要分散工作负载，使用多进程会更简单有效，参见 11.2 节。

① 用 C 语言开发的 Python 参考实现，也就是通常在 shell 中输入 `python` 之后运行的那个。
② 尽管是使用最普遍的。

对于我们这些开发人员、普通人来说，这意味着我们在使用多线程时要三思。我在rebuildd（http://julien.danjou.info/projects/rebuildd）中使用多线程来分发作业，rebuildd 是我多年前写的一个做 Debian 构建（build）的守护进程。尽管用线程去控制每个构建作业很方便，但我很快便掉进了并发陷阱中。如果有机会再做一次的话，我会使用基于异步事件处理或者多进程的方式来做，也就不用再担心这个问题了。

处理好多线程是很难的。其复杂程度意味着与其他方式相比它是 bug 的更大来源，而且考虑到通常能够获得的好处很少，所以最好不要在多线程上浪费太多精力。

11.2 多进程与多线程

正如前面解释的，因为 GIL 的问题，多线程并非好的可扩展性方案。更好的方案是 Python 中提供的 **multiprocessing** 包。它提供了类似 **multithreading** 模块中的接口，区别在于它会启动一个新的进程（通过 `fork(2)`）而不是一个新的系统线程。

下面是一个简单的例子，计算 100 万个随机整数的和 8 次，同时将其分散到 8 个线程中。

使用多线程的 worker

```
import random
import threading

results = []

def compute():
    results.append(sum(
        [random.randint(1, 100) for i in range(1000000)]))

workers = [threading.Thread(target=compute) for x in range(8)]
for worker in workers:
    worker.start()
for worker in workers:
    worker.join()
print("Results: %s" % results)
```

程序的运行结果如示例 11.1 所示。

示例 11.1　`time python worker.py` 的运行结果

```
    $ time python worker.py
    Results: [50517927, 50496846, 50494093, 50503078, 50512047, 50482863,
50543387, 50511493]
    python worker.py  13.04s user 2.11s system 129% cpu 11.662 total
```

这个程序运行在四核的 CPU 上，这意味着 Python 最多可以利用 400%的 CPU 能力。但显然它做不到，即使并行运行 8 个进程，它仍然卡在了 129%，这只是硬件能力的 32%。

现在，我们使用 **`multiprocessing`** 重写一下如示例 11.2 所示。对于这种简单的例子来说，实现是相当直接的。

示例 11.2　使用 `multiprocessing` 的 worker

```
    import multiprocessing
    import random

    def compute(n):
        return sum(
            [random.randint(1, 100) for i in range(1000000)])

    # Start 8 workers
    pool = multiprocessing.Pool(8)
    print("Results: %s" % pool.map(compute, range(8)))
```

在同样的条件下运行这个程序，结果如示例 11.3 所示。

示例 11.3　`time python worker.py` 的运行结果

```
    $ time python workermp.py
    Results: [50495989, 50566997, 50474532, 50531418, 50522470, 50488087,
50498016, 50537899]
    python workermp.py  16.53s user 0.12s system 363% cpu 4.581 total
```

执行时间减少到 60%，这次程序可以消耗 363%的 CPU 能力，超过 CPU 能力的 90%。

此外，`multiprocessing` 模块不仅可以有效地将负载分散到多个本地处理器上，而且可以通过它的 `multiprocessing.managers` 对象在网络中分散负载。它还提供了双向传输，以使进程间可以彼此交换信息。

每次考虑在一定的时间内**并行**处理一些工作时，最好是依靠多进程创建（fork）多个作

业，以便能够在多个 CPU 核之间分散负载。

11.3 异步和事件驱动架构

事件驱动编程会一次监听不同的事件，对于组织程序流程是很好的解决方案，并不需要使用多线程的方法。

考虑这样一个程序，它想要监听一个套接字的连接，并处理收到的连接。有以下三种方式可以解决这个问题。

（1）每次有新连接建立时就创建（fork）一个新进程，需要用到 `multiprocessing` 这样的模块。

（2）每次有新连接建立时创建一个新线程，需要用到 `threading` 这样的模块。

（3）将这个新连接加入事件循环（event loop）中，并在事件发生时对其作出响应。

（现在）众所周知的是，使用事件驱动方法对于监听数百个事件源的场景的效果要好于为每个事件创建一个线程的方式[①]。这并不意味着二者互不兼容，这只是表明可以通过事件驱动机制摆脱多线程。

我们已经在前面讨论了前面两种选择的优劣，本节只讨论事件驱动机制。

事件驱动架构背后的技术是事件循环的建立。程序调用一个函数，它会一直阻塞直到收到事件。其核心思想是令程序在等待输入输出完成前保持忙碌状态，最基本的事件通常类似于“我有数据就绪可被读取”或者“我可以无阻塞地写入数据”。

在 Unix 中，用于构建这种事件循环的标准函数是系统调用 `select(2)` 或者 `poll(2)`。它们会对几个文件描述符进行监听，并在其中之一准备好读或写时做出响应。

在 Python 中，这些系统调用通过 `select` 模块开放了出来。很容易用它们构造一个事件驱动系统，尽管这显得有些乏味。使用 `select` 的基本示例如示例 11.4 所示。

示例 11.4 使用 `select` 的基本示例

```
import select
import socket
```

① 关于这个的进一步阅读，可以看看 C10K 问题（http://www.kegel.com/c10k.html#nb.kqueue）。

```
server = socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM)
# Never block on read/write operations
server.setblocking(0)

# Bind the socket to the port
server.bind(('localhost', 10000))
server.listen(8)

while True:
    # select() returns 3 arrays containing the object (sockets, files…) that
    # are ready to be read, written to or raised an error
    inputs, outputs, excepts = select.select(
        [server], [], [server])
    if server in inputs:
        connection, client_address = server.accept()
        connection.send("hello!\n")
```

不久前一个针对这些底层接口的包装器被加入到了 Python 中，名为 `asyncore`。它还没有被广泛使用，而且演进也不太多。

或者，还有很多其他框架通过更为集成化的方式提供了这类功能，如 Twisted（https://twistedmatrix.com/trac/）或者 Tornado（http://www.tornadoweb.org/en/stable/）。Twisted 多年来在这方面已经成为了事实上的标准。也有一些提供了 Python 接口的 C 语言库（如 `libevent`、`libev` 或者 `libuv`）也提供了高效的事件循环。

尽管它们都能解决同样的问题，但不利的一面在于现在选择太多了，而且它们之间大多数不能互操作。而且，它们大多基于回调机制，这意味着在阅读代码时，程序的流程不是很清晰。

gevent（http://www.gevent.org/）或者 Greenlet（http://greenlet.readthedocs.org/en/latest/）怎么样呢？它们没有使用回调，但实现的细节很吓人，而且包括一些 CPython 在 x86 上的特有的代码以及对标准函数的 monkey 补丁。如果要长期使用和维护的话实际并非好的选择。

最近，Guido Van Rossum 开始致力于一个代号为 tulip 的解决方案，其记录在 PEP 3156 中。[①]这个包的目标就是提供一个标准的事件循环接口。将来，所有的框架和库都将与这个接口兼容，而且将实现互操作。

① *Asynchronous IO Support Rebooted: "asyncio" Module*, Guido van Rossum, 2012

tulip 已经被重命名并被并入了 Python 3.4 的 asyncio 包中。如果不打算依赖 Python 3.4 的话，也可以通过 PyPI（https://pypi.python.org/pypi/asyncio）上提供的版本装在 Python 3.3 上，只需通过运行 pip install asyncio 即可安装。Victor Stinner 已经开始进行移植并将 tulip 命名为 trollius（https://pypi.python.org/pypi/trollius），目标是令其可以兼容 Python 2.6 及其后续版本。

现在你已经拿到了所有的牌，你肯定会想：**那我到底该用什么在事件驱动的应用中构建一个事件循环呢**？

在当前的 Python 开发中，这个问题很难回答。这门语言仍然在转换阶段。截止到本书写作时，还没有什么应用使用了 asyncio 模块。这意味着用了它很可能面临巨大的挑战。

下面是目前我能给出的一些建议。

- 如果只针对 Python 2，asyncio 基本不用考虑。对我来说，接下来最好的选择是基于 libev 的库，如 pyev（https://pypi.python.org/pypi/pyev）。
- 如果目标是同时支持 Python 的主要版本（Python 2 和 Python 3），最好使用能同时支持两个版本的库，如 pyev。不过，我必须强烈建议你记住，未来很可能需要迁移到 asyncio 上。所以有一个最小化的抽象层会很有帮助，并且不要在整个程序中到处产生内部结构对事件库的依赖。如果愿意冒险的话，尝试混合使用 asyncio 和 trollius 也是个不错的方案。
- 如果只针对 Python 3，那就用 asyncio。刚开始会比较痛苦，因为没有太多的例子和文档可以参考，但这是个安全的选择。你将会是先驱。

如示例 11.5 所示，pyev 的接口是很容易掌握的。通过对 libev 的使用，它通不但支持用于得 Io 对象，而且支持对子进程的跟踪，计时器、信号量和空闲时的事件回调。libev 还可以自动利用 polling 的最好的接口，如 Linux 上的 epoll(2)或者 BSD 上的 kqueue(2)。

示例 11.5 pyev 示例

```
import pyev
import socket

server = socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM)
# Never block on read/write operations
```

```
server.setblocking(0)

# Bind the socket to the port
server.bind(('localhost', 10000))
server.listen(8)

def server_activity(watcher, revents):
    connection, client_address = server.accept()
    connection.send("hello!\n")
    connection.close()

loop = pyev.default_loop()
watcher = pyev.Io(server, pyev.EV_READ, loop, server_activity)
watcher.start()
loop.start()
```

11.4　面向服务架构

考虑到前面阐述的问题和解决方案，Python 在解决大型复杂应用的可扩展性方面的问题似乎难以规避。然而，Python 在实现面向服务架构（Service-Oriented Architecture，SOA）方面的表现是非常优秀的。如果不熟悉这方面的话，线上有大量相关的文档和评论。

SOA 是 OpenStack 所有组件都在使用的架构。组件通过 HTTP REST 和外部客户端（终端用户）进行通信，并提供一个可支持多个连接协议的抽象 RPC 机制，最常用的就是 AMQP。

在你自己的场景中，模块之间沟通渠道的选择关键是要明确将要和谁进行通信。

当需要暴露 API 给外界时，目前最好的选择是 HTTP，并且最好是无状态设计，例如 REST（Representational state transfer）风格的架构。这类架构非常容易实现、扩展、部署和理解。

然而，当在内部暴露和使用 API 时，使用 HTTP 可能并非最好的协议。有大量针对应用程序的通信协议存在，对任何一个协议的详尽描述都需要一整本书的篇幅。

在 Python 中，有许多库可以用来构建 RPC（Remote Procedure Call）系统。Kombu（http://kombu.readthedocs.org/en/latest/）与其他相比是最有意思的一个，因为它提供了一种基于很多后端的 RPC 机制。AMQ 协议（http://www.amqp.org/）是主要的一个。但同样支持 Redis（http://redis.io/）、MongoDB（https://www.mongodb.org/）、BeanStalk（http://kr.github.io/beanstalkd/）、

Amazon SQS（http://aws.amazon.com/cn/sqs/）、CouchDB（http://couchdb.apache.org/）或者 ZooKeeper（http://zookeeper.apache.org/）。

最后，使用这样松耦合架构的间接收益是巨大的。如果考虑让每个模块都提供并暴露 API，那么可以运行多个守护进程暴露这些 API。例如，Apache httpd 将使用一个新的系统进程为每一个连接创建一个新 worker，因而可以将连接分发到同一个计算节点的不同 worker 上。要做的只是需要有一个系统在 worker 之间负责分发工作，这个系统提供了相应的 API。每一块都将是一个不同的 Python 进程，正如我们在上面看到的，在分发工作负载时这样做要比用多线程好。可以在每个计算节点上启动多个 worker。尽管不必如此，但是在任何时候，能选择的话还是最好使用无状态的组件。

ZeroMQ（http://zeromq.org/）是个套接字库，可以作为并发框架使用。下面的例子实现了和前面例子中同样的 worker，但是利用了 ZeroMQ 作为分发和通信的手段。

使用 ZeroMQ 的 Worker

```
import multiprocessing
import random
import zmq

def compute():
    return sum(
        [random.randint(1, 100) for i in range(1000000)])

def worker():
    context = zmq.Context()
    work_receiver = context.socket(zmq.PULL)
    work_receiver.connect("tcp://0.0.0.0:5555")
    result_sender = context.socket(zmq.PUSH)
    result_sender.connect("tcp://0.0.0.0:5556")
    poller = zmq.Poller()
    poller.register(work_receiver, zmq.POLLIN)

    while True:
        socks = dict(poller.poll())
        if socks.get(work_receiver) == zmq.POLLIN:
            obj = work_receiver.recv_pyobj()
            result_sender.send_pyobj(obj())

context = zmq.Context()
```

```
# Build a channel to send work to be done
work_sender = context.socket(zmq.PUSH)
work_sender.bind("tcp://0.0.0.0:5555")
# Build a channel to receive computed results
result_receiver = context.socket(zmq.PULL)
result_receiver.bind("tcp://0.0.0.0:5556")
# Start 8 workers
processes = []
for x in range(8):
    p = multiprocessing.Process(target=worker)
    p.start()
    processes.append(p)
# Start 8 jobs
for x in range(8):
    work_sender.send_pyobj(compute)
# Read 8 results
results = []
for x in range(8):
    results.append(result_receiver.recv_pyobj())
# Terminate all processes
for p in processes:
    p.terminate()
print("Results: %s" % results)
```

如你所见，ZeroMQ 提供了非常简单的方式来建立通信信道。我这里选用了 TCP 传输层，表明我们可以在网络中运行这个程序。应该注意的是，ZeroMQ 也提供了利用 Unix 套接字的 inproc 信道。显然在这个例子中，基于 ZeroMQ 构造的通信协议是非常简单的，这是为了保持本书中的例子尽量清晰和简洁，不难想象基于其上建立一个更为复杂的通信层。

通过这种协议，不难想象通过网络消息总线（如 ZeroMQ、AMQP 等）构建一个完全分布式的应用程序间通信。

注意，类似 HTTP、ZeroMQ 或者 AMQP 这样的协议是同语言无关的。可以使用不同的语言和平台构建系统的各个部分。尽管我们都认同 Python 是一门优秀的语言，但其他团队也许有他们的偏好，或者对于问题的某个部分，其他语言可能是更好的选择。

最后，使用传输总线（transport bus）解耦应用是一个好的选择。它允许你建立同步和异步 API，从而轻松地从一台计算机扩展到几千台。它不会将你限制在一种特定技术或语言上，现如今，没理由不将软件设计为分布式的，或者受任何一种语言的限制。

第 12 章 RDBMS 和 ORM

RDBMS（Relational DataBase Management System，关系型数据库管理系统）和 ORM（Object-Relational Mapping，对象关系映射）是一个不太讨好的题目，但是早晚都要处理。许多应用程序都需要存储某种形式的数据，而开发人员通常会选择使用关系型数据库。并且当开发人员选择使用关系型数据库时，它们通常会选择某种 ORM 库。

注意

本章将不再过多以 Python 中心，请多多包含。这里只讨论关系型数据库，但是这里涉及的很多内容同样适用于其他类型的数据库。

RDBMS 是关于将数据以普通表单的形式存储的，而 SQL 是关于如何处理关系代数的。二者结合就可以对数据进行存储，同时回答关于数据的问题。然而，在面向对象程序中使用 ORM 有许多常见的困难，统称为对象关系阻抗失配（object-relational impedance mismatch，http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch）。根本在于，关系型数据库和面向对象程序对数据有不同的表示方式，彼此之间不能很好地映射：不管怎么做，将 SQL 表映射到 Python 的类都无法得到最优的结果。

ORM 应该使数据的访问更加容易，这些工具会抽象创建查询、生成 SQL 的过程，无需自己处理。但是，你迟早会发现有些想做的数据库操作是这个抽象层不允许的。为了更有效地利用数据库，必须对 SQL 和 RDBMS 有深入了解以便能直接写自己的查询而无需每件事都依赖抽象层。

但这不是说要完全避免用 ORM。ORM 库可以帮助快速建立应用模型的原型，有些甚至能提供非常有用的工具，如模式（schema）的升降级。重要的是要了解它并不能完全替代 RDBMS。许多开发人员试图在它们选择的语言中解决问题而不使用他们的模型 API，通常他们给出的方案却并不优雅。

设想一个用来记录消息的 SQL 表。它有一个名为 id 的列作为主键和一个用来存放消息的字符串列。

```
CREATE TABLE message (
  id serial PRIMARY KEY,
  content text
);
```

我们希望收到消息时避免重复记录，所以一个典型的开发人员会这么写：

```
if message_table.select_by_id(message.id):
    # We already have the message, it's a duplicate, ignore and raise
    raise DuplicateMessage(message)
else:
    # Insert the message
    message_table.insert(message)
```

这在大多数情况下肯定可行，但它有些主要的弊端。

- 它实现了一个已经在 SQL 模式中定义了的约束，所以有点儿代码重复。
- 执行了两次 SQL 查询，SQL 查询的执行可能会时间很长而且需要与 SQL 服务器往返的通信，造成额外的延迟。
- 没有考虑到在调用 select_by_id 之后程序代码 insert 之前，可能有其他人插入一个重复消息的可能性，这会引发程序抛出异常。

下面是一种更好的方式，但需要同 RDBMS 服务器合作而不是将其看作是单纯的存储。

```
try:
    # Insert the message
    message_table.insert(message)
except UniqueViolationError:
    # Duplicate
    raise DuplicateMessage(message)
```

这段代码以更有效的方式获得了同样的效果而且没有任何竞态条件（race condition）问题。这是一种非常简单的模式，而且和 ORM 完全没有冲突。这个问题在于开发人员将 SQL 数据库看作是单纯的存储并且在他们的控制器代码而不是他们的模型中重复他们已经（或者可能）在 SQL 中实现的约束。

将 SQL 后端看作是模型 API 是有效利用它的好方法。通过它本身的过程性语言编写简

单的函数调用即可操作存储在 RDBMS 中的数据。

另外需要强调的一点是，ORM 支持多种数据库后端。许多 ORM 库都将其作为一项功能来吹捧，但它实际上却是个陷阱，等待诱捕那些毫无防备的开发人员。没有任何 ORM 库能提供对所有 RDBMS 功能的抽象，所以你将不得不削减你的代码，只支持那些 RDBMS 最基本的功能（或者你容忍），而且将不能在不破坏抽象层的情况下使用任何 RDBMS 的高级功能。

有些在 SQL 中尚未标准化的简单得事情在使用 ORM 时处理起来会很痛苦，如处理时间戳操作。如果代码写成了与 RDBMS 无关的就更是如此。基于这一点，在选择适合你的应用程序的 RDBMS 时要更加仔细[①]。

减轻 ORM 库的这个问题的可行办法就是像 2.3 节描述的那样对它们进行隔离。这种方法不仅可以在需要时轻松将 ORM 库切换到另一个，而且可以在发现查询的使用效率不高的地方对其进行优化，越过大多数 ORM 引用。

建立这种隔离的一种简单办法是只在应用的某一个模块中使用 ORM，如 `myapp.storage`。这种方法应该只在高度抽象的层面输出数据操作的函数和方法。ORM 应该只在这个模块中使用。在此之后，就可以加入任何提供了相同 API 的模块以替换 `myapp.storag`。

最后，本节的目标不是要在是否使用 ORM 的辩论中做出选择，互联网上已经有大量的关于其优缺点的讨论。本节的重点在于帮你理解对 SQL 和 RDBMS 充分了解在应用程序中充分利用它们的潜力有多么重要。

Python 中最常使用的（和有争议的事实标准）ORM 库是 SQLAlchemy（http://www.sqlalchemy.org/）。它支持大量的不同后端并且对大多数通用操作都提供了抽象。模式升级可以通过第三方库完成，如 alembic（https://pypi.python.org/pypi/alembic）。

有些框架，如 Django（https://www.djangoproject.com/），提供了它们自己的 ORM 库。如果选择使用一个框架，那么使用内置的库是明智的选择，通常（显然）与外部 ORM 库相比，内置的库与框架集成得更好。

① 如果犹豫不决的话，选 PostgreSQL（http://www.postgresql.org/）。

警告

大多数框架依赖的 MVC（Model View Controller）架构很容易被滥用。它们在它们的模型中直接实现 ORM，但却没有足够的抽象，任何在视图（view）和控制器（controller）中使用的模型（model）都将被 ORM 直接使用。这是应该避免的。应该写包含 ORM 库的数据模型而不是组成它的数据模型，这能提供更好的可测试性和更好的隔离，也可以在需要时更容易地切换到另外一种存储技术上。

12.1　用 Flask 和 PostgreSQL 流化数据

前面一节讨论了掌握数据存储系统有多么重要。这里将展示如何用 PostgreSQL 的一个高级特性构造一个 HTTP 事件流系统。

这个小应用的目的是将消息存储在一个 SQL 表中并通过 HTTP REST API 提供对这些消息的访问。每个消息由一个整数类型的 `channel`、一个字符串类型的 `source`、一个字符串类型的 `content` 组成。创建这个表的代码非常简单，如示例 12.1 所示。

示例 12.1　创建 `message` 表

```
CREATE TABLE message (
  id SERIAL PRIMARY KEY,
  channel INTEGER NOT NULL,
  source TEXT NOT NULL,
  content TEXT NOT NULL
);
```

另外还需要做的是序列化这些消息，以便客户端能够实时对它们进行处理。这需要用到 PostgreSQL 的 `LISTEN`（http://www.postgresql.org/docs/9.2/static/sql-listen.html）和 `NOTIFY`（http://www.postgresql.org/docs/9.2/static/sql-notify.html）功能。这些功能可以监听来自函数的消息，这个函数由用户提供，由 PostgreSQL 执行，如示例 12.2 所示。

示例 12.2　`notify_on_insert` 函数

```
CREATE OR REPLACE FUNCTION notify_on_insert() RETURNS trigger AS $$
BEGIN
  PERFORM pg_notify('channel_' || NEW.channel,
                CAST(row_to_json(NEW) AS TEXT));
```

```
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

这会创建一个用 pl/pgsql 编写的触发器函数，pl/pgsql 语言只有 PostgreSQL 可以理解。需要注意的是，这个函数也可以用其他语言编写，如 Python 本身，因为 PostgreSQL 是通过嵌入 Python 解释器支持 pl/python 语言的。

函数会执行一个对 `pg_notify` 的调用。这是实际发送通知的函数。第一个参数是一个代表一个信道的字符串，第二个参数是携带实际净荷（palyload）的字符串。这里根据 `channel` 列在行内的的值来动态定义信道。在这个例子中，净荷是以 JSON 格式表示的整个行。没错，PostgreSQL 原生地就知道如何将行转换为 JSON。

我们希望对 `message` 表的每一次 `INSERT` 操作都发送通知消息，所以需要在这样的事件上出发这个函数，如示例 12.3 所示。

示例 12.3 `notify_on_insert` 的触发器

```
CREATE TRIGGER notify_on_message_insert AFTER INSERT ON message
FOR EACH ROW EXECUTE PROCEDURE notify_on_insert();
```

搞定。这个函数已经插入并且在 `message` 表每一次 `INSERT` 操作成功后都会被执行。

可以通过 `psql` 中的 `LISTEN` 操作检查它是否工作正常：

```
$ psql
psql (9.3rc1)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

mydatabase=> LISTEN channel_1;
LISTEN
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'hello world');
INSERT 0 1
Asynchronous notification "channel_1" with payload
"{"id":1,"channel":1,"source":"jd","content":"hello world"}"
received from server process with PID 26393.
```

一旦行被插入，通知就被发送，并且可以通过 PostgreSQL 客户端进行接收。现在需要做的就是构建 Python 应用对这个事件进行流化（stream），如示例 12.4 所示。

示例 12.4　在 Python 中接收通知

```
import psycopg2
import psycopg2.extensions
import select

conn = psycopg2.connect(database='mydatabase', user='myuser',
                    password='idkfa', host='localhost')

conn.set_isolation_level(
   psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

curs = conn.cursor()
curs.execute("LISTEN channel_1;")

while True:
   select.select([conn], [], [])
   conn.poll()
   while conn.notifies:
      notify = conn.notifies.pop()
      print("Got NOTIFY:", notify.pid, notify.channel, notify.payload)
```

上面的代码利用库 `psycopg2` 连接 PostgreSQL。也可以使用一个提供了抽象层的库，如 `SQLAlchemy`，但是它们都无法提供对 PostgreSQL`LISTEN/NOTIFY` 功能的访问。通过访问底层数据库连接去执行代码也是可能的，但是在这个例子中没必要那么做，因为这里并不需要任何 ORM 库提供的其他功能。

这个程序会在 `channel_1` 上进行监听。一旦收到通知则将其打印到屏幕上。如果运行这个程序并向 `message` 表中插入一行，则会得到如下输出：

```
$ python3 listen.py
Got NOTIFY: 28797 channel_1
{"id":10,"channel":1,"source":"jd","content":"hello world"}
```

现在我们将使用 Flask（http://flask.pocoo.org/），一个简单的 HTTP 微型框架，去构造应

用程序。这里将使用由 HTML5[①]中定义的 **Server-Sent Events**（http://www.w3.org/TR/2009/WD-eventsource-20090423/）消息协议，如示例 12.5 所示。

示例 12.5 Flask 流化应用程序

```
import flask
import psycopg2
import psycopg2.extensions
import select

app = flask.Flask(__name__)

def stream_messages(channel):
    conn = psycopg2.connect(database='mydatabase', user='mydatabase',
                            password='mydatabase', host='localhost')
    conn.set_isolation_level(
        psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

    curs = conn.cursor()
    curs.execute("LISTEN channel_%d;" % int(channel))

    while True:
        select.select([conn], [], [])
        conn.poll()
        while conn.notifies:
            notify = conn.notifies.pop()
            yield "data: " + notify.payload + "\n\n"

@app.route("/message/<channel>", methods=['GET'])
def get_messages(channel):
    return flask.Response(stream_messages(channel),
                          mimetype='text/event-stream')

if __name__ == "__main__":
    app.run()
```

这个应用程序非常简单并且只是为这个例子支持了流化。我们使用 Flask 将请求路由到

① 另一种选择是使用 HTTP/1.1 中定义的 Transfer-Encoding: chunked。

GET/message/<channel>，一旦代码被调用，它将以 mimetype 为 text/event-stream 的格式进行响应，发回一个生成器函数而非一个字符串。Flask 接下来将调用这个函数并在每次生成器生成东西时发送结果。

生成器 stream_messages 重用了之前写的用来监听 PostgreSQL 通知的代码。它接收信道 Id 作为参数，监听这个信道，并生成其有效载荷。记住，我们在触发器函数中用的是 PostgreSQL 的 JSON 编码函数，所以从 PostgreSQL 收到的就是 JSON 格式的数据，因为发送 JSON 数据给 HTTP 客户端没有任何问题，所以无需转换编码。

注意

为简单起见，这个示例应用程序被写在了一个单独的文件中。在一本书中描述一个横跨多个模块的例子有点儿困难。如果这是一个真正的应用程序，那么最好是将存储处理的实现放到一个自己的 Python 模块中。

现在可以运行这个服务器了：

```
$ python listen+http.py
 * Running on http://127.0.0.1:5000/
```

在另一个终端中，可以进行连接并在事件进入时对数据进行抽取。在连接时，不会接收收据并且连接保持开放状态。

```
$ curl -v http://127.0.0.1:5000/message/1
* About to connect() to 127.0.0.1 port 5000 (#0)
*   Trying 127.0.0.1...
* Adding handle: conn: 0x1d46e90
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x1d46e90) send_pipe: 1, recv_pipe: 0
* Connected to 127.0.0.1 (127.0.0.1) port 5000 (#0)
> GET /message/1 HTTP/1.1
> User-Agent: curl/7.32.0
> Host: 127.0.0.1:5000
> Accept: */*
>
```

但一旦插入一些行到 message 表中时：

```
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'hello world');
INSERT 0 1
mydatabase=> INSERT INTO message(channel, source, content)
mydatabase-> VALUES(1, 'jd', 'it works');
INSERT 0 1
```

终端上 `curl` 运行的位置就会有数据输出：

```
data: {"id":71,"channel":1,"source":"jd","content":"hello world"}

data: {"id":72,"channel":1,"source":"jd","content":"it works"}
```

关于这个应用程序的一个朴素的且可以说更轻便的实现[①]不是通过一个 `SELECT` 语句一次次查询是否有新数据插入表内。不过，没有必要在这里展示这样一个推送系统（push system），尽管它比持续地轮询数据库要效率高。

12.2 Dimitri Fontaine 访谈

我认识 Dimitri 已经 20 年了。他是一位经验丰富的 PostgreSQL 主要贡献者，在 2ndQuadrant 公司（http://2ndquadrant.com/en/）工作，并在 pgsql-hackers 的邮件列表上与其他数据库大牛辩论。我们彼此分享了很多开源的体验，并且他非常热心的回答了很多在处理数据库时你应该了解的问题。

对用 RDBMS 作为存储后端的开发人员你有什么建议吗？有什么是他们需要了解的吗？

① 能够同其他 RDBMS 服务器兼容，如 MySQL。

这是一个很好的问题，因为它让我有更多机会专门澄清一些非常错误的假设。如果你觉得这个问题有意义的话，那么现在你真的有必要看一下我的回答。

让我们从一些无聊的部分开始：RDBMS 代表 Relational DataBase Management System（关系型数据库系统）。它们是在 20 世纪 70 年代发明的，用来解决一些那个时代的每个开发人员都会遇到的常见问题，并且 RDBMS 实现的主要服务并不是数据存储，因为人们已经知道如何实现数据的存储了。

RDBMS 提供的主要服务如下。

- 并发：可以**执行多个并发的线程**对数据进行读写，RDBMS 能够正确地进行处理。这是你需要 RDBMS 提供的主要功能。
- 并发语义：讨论 RDBMS 中有关并发行为的细节必然涉及高级规范中的**原子性**（Atomicity）和**隔离性**（Isolation），它们可能是 **ACID**（Atomicity，Consistency，Isolation，Durability）中最关键的部分。**原子性**是指在开始（`BEGIN`）一个事务和其结束之前（不管是执行 `COMMIT` 还是 `ROLLBACK`）的这段时间内，任何系统中的其他并发行为都无法获知你在干什么，不管是什么的一种性质。当使用一个合适的包含**数据定义语言**（Data Definition Language，即 DDL，如 `CREATE TABLE` 或 `ALTER TABLE`）的 RDBMS 时，**隔离性**是指在系统中你自己的事务内允许你看到的其他并发行为。**SQL 标准**中定义了四个级别的隔离，在事务隔离文档（http://www.postgresql.org/docs/9.2/static/transaction-iso.html）中有所描述。

RDBMS 承担了对数据的全部责任。所以它允许开发人员**描述**自己的一致性规则，并且会在关键的时候对这些规则进行检查，例如，当事务**提交**时或者语句的边界，取决于约束声明的**可延迟性**（deferability）。

对于数据的第一个约束是关于其期望的输入输出格式的，即使用合适的**数据类型**。合适的 RDBMS 知道更多关于如何处理**文本**、**数值**和**日期格式**，并能恰当地处理实际出现在当今日历中的日期（罗马儒略历[①]到目前还不算特别大，除非处理历史日期，否则你可能需要格里高里历[②]）。

但是数据类型不仅仅与输入输出格式有关，它们还允许实现行为和某种程度的**多态性**，

① Julian calendar，参见 http://en.wikipedia.org/wiki/Julian_calendar
② Gregorian calendar，参见 http://en.wikipedia.org/wiki/Gregorian_calendar。——译者注

因为我们期望的基本相等测试是针对特定数据类型的。我们不会以同样的方式比较文本和数字，比较日期和 IP 地址，比较点框和线条，比较布尔和圆形，比较 UUID 和 XML，比较数组和范围，等等。

保护数据还意味着合适的 RDBMS 的唯一选择是要能主动拒绝不满足一致性的规则的数据，首当其冲的就是你已经选择的数据类型。如果你觉得处理类似 `0000-00-00` 这样在日历中根本不存在的日期可以的话，那么你需要重新考虑一下。

保证**一致性**的其他有关**约束**的表现方式还包括 `CHECK` 约束，`NOT NULL` 约束和**约束触发器**，后者通常被称为**外键**。所有这些都可以作为数据类型定义和行为在用户层面的扩展，主要区别是可以选择 DEFER 检查这些约束从每条语句结束到当前事务结束。

RDBMS 的**关系**特性主要体现在对数据的建模，以及保证所有在同一个**关系**中的**元组**共享通用的规则集，即结构和约束。当执行时，即表示我们正在强制使用适当的显式模式来处理数据。

令数据工作在适当模式上的过程被称为**规范化**（normalization），并且可以在设计中实现许多有些细微不同的**范式**（Normal Form）。但是有时也会需要规范化过程无法提供的灵活性。常见的办法是先规范化数据模式，然后反过来看如何进行**反规范化**（denormalization）以获得需要的灵活性。你可能会碰巧发现你并不需要额外的灵活性。

当你发现确实需要更多的灵活性时，PostgreSQL 为初学者提供了一些**反规范化**的选择：组合类型（composite type）、记录（record）、数组（array）、hstore、json 或 XML。

但**反规范化**有一个很重要的缺点，就是我们接下来要讨论的**查询语言**（Query Language），它被设计为处理而非**规范化**数据。当然，PostgreSQL 已经对查询语言进行了扩展，当在使用组合类型、数组或 hstore，甚至最近发布的 json 时，支持尽可能多地**非规范化**数据。

RDBMS 对数据非常了解并能在需要的情况下帮助实现非常细粒度的安全模型。访问模式被控制在关系和列层面，并且 PostgreSQL 还实现了 `SECURITY DEFINER` 存储过程，可以对敏感数据提供非常受限的访问，很像在使用 `suid` 程序。

RDBMS 可以使用**结构化查询语言**对数据进行访问，结构化语言在 20 世纪 80 年代已经成为了**事实上的标准**并且目前由一个专门的委员会负责管理。对于 PostgreSQL，每年的每个主要发布版本都有大量的扩展被加入，以支持极为丰富的 **DSL** 语言。所有查询规划和优化

的工作都由 RDBMS 来完成，以便你可以专注于**声明式**的查询，即只需要描述对于你所拥有的数据想要什么样的结果。

这也是在这里要对 NoSQL 多加注意的原因，因为大部分这些新兴的产品实际上移除的不光是结构化查询语言，还包括你已经掌握并期望包含的很多其他基础的东西。

我的建议是开发人员要记住**存储后端**和 RDBMS 间的区别。它们是非常不同的服务，如果需要的只是存储后端的话，也许应该考虑避免用 RDBMS。

但是大多数情况下，你真正需要的是一个完全成熟的 RDBMS。这种情况下，最好的选择是 PostgreSQL。去读一下它的文档，看看它提供的数据类型、操作符、函数、特性和扩展的清单，以及在博客上看一些使用示例。

然后考虑一下在你的开发中将 PostgreSQL 作为一个工具来利用，并将其包含在应用程序的架构中。你需要实现的部分服务在 RDBMS 层面已经给予了最好的支持，而且 PostgreSQL 擅长成为整个实现中最值得信赖的部分。

用或不用 ORM 的最好方式各是什么？

SQL 代表 Structured Query Language（结构化查询语言）并且其对于 PostgreSQL 已经被证明是**图灵完备**的。它的实现和优化都相当有分量。

由于 ORM 代表 Object Relational Mapper（对象关系映射器），其思想是你能够处理一对一的映射，即数据库关系和类，以及数据库元组和对象（或者说类实例）。

即使对于 PostgreSQL 这样种已经实现了强静态类型的 RDBMS，关系定义也是动态建立的，每一次查询结果都是一个新关系。每一次子查询的结果也是一个新关系，而且可能只存在于这个子查询期间。每一个 `JOIN`，无论是 `INNER` 或者 `OUTER`，都将动态生成一个新关系以处理 `JOIN`。

作为一个直接结果，很容易明白 ORM 能完成的最好的工作就是所谓的 **CRUD** 应用，即**创建**（Create）、**读取**（Read）、**更新**（Update）、**删除**（Delete）。**读取**部分会受限制，只能对单个表做非常简单的 `SELECT` 查询。如果比较较大的输出列表，可以测量出提取额外的列和必要的列之间的查询性能的区别。现在，如果 ORM 在它的**投影**（或输出列表）中包含所有的已知列，那么它将强制你的 RDBMS 在发送前提取额外的数据（并解压缩），如果在 RDBMS 和应用程序之间使用 **SSL** 的话可能还需要再次压缩。而且，还要考虑到网络带宽的使用并记得我们正在测量的是基于**主键**的**毫秒级**的查询。

所以，任何从 RDBMS 中提取但最终没有使用的列，都是对宝贵资源的严重浪费，是可扩展性的第一杀手。

即使在 ORM 能够只获取你请求的数据，接下来你也必须以某种方式管理在每种情况下要显示的具体列，并避免使用会自动计算字段列表的简单抽象的魔术方法。

CRUD 查询的其余部分是简单的 `INSERT`、`UPDATE` 和 `DELETE`。首先，当使用高级的 RDBMS（如 PostgreSQL）时所有这些命令都接受连接（join）和子查询。而且仍需提及的是，比如 PostgreSQL 实现了 `RETURNING` 子句，允许返回给客户端任何刚编辑过的数据，如**默认值**（对于代理键通常是序列号）和其他在 RDBMS（一般通过 `BEFORE <action>`触发器）上自动**计算**的值。但你的 ORM 能够意识到这些吗？什么语法可以从中受益呢？通常情况下，一个关系或者是一个表（ 调用一个**返回集合的函数**的结果），或者是任意查询的结果。常见的做法是使用 ORM 构造已定义的表和其他**模型类**（或其他辅助模块）之间的**关系映射**。

如果从总体上考虑 SQL 的整个语义的话，那么**关系映射**实际上应该能够将任意查询映射到一个类。因此可能需要为每个运行的查询建立一个新类。

充分智能编译器（Sufficiently Smart Compiler）的传说同样适用于 ORM。关于这个传说的更多细节，可以读一下 James Hague 的 *On Being Sufficiently Smart*（http://prog21.dadgum.com/40.html）。

其思想应用到我们这个场景下就是你相信 ORM 能比你写出更高效的 SQL 查询，认为即使你没有给出足够的信息也能精准地给出你想要的数据集。

有时候 SQL 确实会变得相当复杂。但是你不太可能通过一个自己无法控制的 SQL 生成器的 API 让其变得更简单。

在讨论了所有典型的 ORM 之后，也需要说一下其他的选择。

将 SQL 查询构造为字符串会难以扩展。你会想要组合几个**限制条件**（`WHERE` 子句）并且动态地添加一些连接（join）到查询中，以便可以有选择地获取更细节的数据等。

我现在的想法是，你真正想要的工具可能并不是 ORM，而是一种通过编程接口更好地组合 SQL 查询的方式。

名为 Postmodern（http://marijnhaverbeke.nl/postmodern/）的 PostgreSQL 驱动针对这个问题提出了几乎类似的抽象，它是一个结合了 S-SQL（http://marijnhaverbeke.nl/postmodern/s-sql.html）方案的 Common Lisp 库。当然，Lisp 借助其本身能够很容易地开发**可组合**的组件。

实际上有两种场景可以放心地使用 ORM，只要你愿意接受下面的条件：你需要尽快将 ORM 使用的代码移出代码库。

- **上市时机**。当真的需要尽可能快地占领市场份额的时候，唯一的办法就是尽快地发布应用和想法的第一个版本。如果你的团队比手写 SQL 更擅长 ORM 的使用的话，那就全力去做。但是你必须意识到，一旦你的应用取得成功，要解决的可扩展性问题之一就是你的 ORM 生成的糟糕的查询，并且 ORM 的使用已经把你逼到了死角并做了一些糟糕的代码设计决定。但如果你到了这个地步的话，那么你应该已经足够成功到花一些重构的钱去删除那些对 ORM 的依赖，对吧？
- **CRUD 应用**。真正要处理的只是一次编辑一个元组，并且不关心性能问题。例如，基本的管理应用界面。

对于 Python 开发，选用 PostgreSQL 与选用其他数据库相比有什么优缺点吗？

下面是我作为开发人员选择 PostgreSQL 的主要原因。

- 社区支持：PostgreSQL 社区非常欢迎新用户，而且通常都会花时间充分理解你的问题以给出最好的可能答案。邮件列表仍然是与社区沟通的最好方式。参考 PostgreSQL 邮件列表（http://www.postgresql.org/list/）以便了解更多细节。
- 数据完整性和持久性：任何发送给 PostgreSQL 的数据都在其定义中都是**安全**（存储）的并能在其后再次获取。
- 数据类型、函数、操作符、数组和范围：PostgreSQL 有着非常丰富的数据类型集合，它们非常有用，并且拥有大量用于处理这些数据类型的操作符和函数。甚至可以使用**数组**或 **JSON** 数据类型进行反规范化，而且仍然可以通过包含连接（join）的高级查询操作它们。例如，你知道正则表达式操作符~吗？还有函数 `regexp_split_to_array` 和 `regexp_split_to_table`？
- 计划器和优化器：你得尝试推进你所知道的关于它们的限制，以便了解它们到底有多复杂和强大。我已经多次看到只为提升几毫秒而完善成的两三页长的查询。
- 事务性的 DDL：几乎可以回滚（`ROLLBACK`）任何命令。现在就试试看，只需要对你的数据库打开 `psql shell` 并输入 `BEGIN; DROP TABLE foo; ROLLBACK;`（将其中的 `foo` 替换成你本地实例中存在的表的名字）。太妙了，对吗？

- `INSERT INTO ... RETURNING`：可以从 `INSERT` 语句中直接返回任何东西，如一个递增序列的 `id` 值。与执行一次 `SELECT` 语句相比，你省去了一次网络通信并且用同一个协议和工具就得到了结果。
- `WITH (DELETE FROM ... RETURNING *) INSERT INTO ... SELECT`：PostgreSQL 在查询中支持常用表表达式（Common Table Expression），被称为 `WITH` 查询，而且幸好它支持 `RETURNING` 子句，因此它还支持 DML 命令。那简直太牛了，是吧？
- 窗口函数 `CREATE AGGREGATE`：如果你不知道窗口函数是什么，那么去读一下 PostgreSQL 手册或者我博客中的 *Understanding Window Functions*（http://tapoueh.org/ blog/2013/08/20-Window-Functions）一文。然后你将意识到 PostgreSQL 允许你使用任何已经存在的**聚合**（aggregate）作为窗口函数，而且允许你在 SQL 中动态地定义新的聚合。
- PL/Python（以及其他语言，如 C、SQL、Javascript 或 Lua）：你可以在服务器上运行你自己的代码，就在数据所在的位置，以便你无需通过网络获取数据进行处理再在查询中将其发回去执行下一级的 JOIN。不管怎样，你可以完全在服务上执行。
- 特定索引（GiST、GIN、SP-GiST、partial&functional）：你知不知道可以从 PostgreSQL 中创建 Python 函数去处理你的数据，并索引函数调用的结果？以便当你发出一个包含了调用该函数的 `WHERE` 子句的查询时，它只从查询中以该数据被调用一次，然后就会直接匹配索引的内容。PostgreSQL 对非排序数据类型实现了索引框架，如二维类型（`ranges`、`geometry` 等）和容器数据类型。许多场景已经默认支持了，这要多谢 PostgreSQL 的**扩展系统**。可以看看扩展支持模块（Additional Supplied Modules，http://www.postgresql.org/docs/9.3/static/contrib.html）和 PostgreSQL 扩展网络（http://pgxn.org/）。
- 扩展：这些扩展包括 **hstore**，一个包含灵活索引的完全成熟的键值存储；**ltree**，用于索引嵌套的标签；**pg_trgm**，作为一个穷人的全文搜索方案支持正则表达式搜索的索引和非锚定 `LIKE` 查询；**ip4r**，用于在一定范围内快速搜索一个 IP 地址，以及更多其他扩展。
- 外部数据封装器：**外部数据封装器**是实现了 SQL/MED 标准（Management of External Data）的一组完整扩展。其思想是将一个连接驱动嵌入 PostgreSQL 服务器中，并将其通过 `CREATE SERVER` 命令暴露出来。PostgreSQL 给外部数据封装器的作者提供

了一个 API，允许他们实现对远程数据的读写，以及 `where` 子句的叠加，以获得更高效的连接（joining）能力。你甚至可以使用 PostgreSQL 的高级 SQL 功能操作那些通过其他技术维护的数据。

- `LISTEN/NOTIFY`：PostgreSQL 实现了一个名为 `LISTEN/NOTIFY` 的异步服务器到客户端的协议。当有些有意思的事情发生时，应用程序可能会收到来自服务器的主动推送的消息，例如更新某些数据时。`NOTIFY` 命令接收一个有效载荷以便可以在对象被删除或更新时通知你的缓存应用相应的对象 ID。当然，只有在事务成功提交后通知才会发生。
- `COPY` 流协议：PostgreSQL 实现了一个**流协议**，并用它实现了全集成的复制方案。现在，可以很容易地在应用程序中使用它，并能带来极大的性能提升。在需要一次处理十几行的时候，有时会在这之前针对一个临时表使用 `COPY`，然后执行一个单独的语句连接这张**临时表**。PostgreSQL 知道如何在所有数据的修改语句（`insert`、`update`、`delete`）中对这些表进行连接，并且批处理操作通常会更快。

第 13 章 Python 3 支持策略

据我所知，目前 Python 3 仍然不是任何操作系统的默认 Python 解释器，尽管它在 2008 年 12 月就已经发布了。

如你所知，问题在于 Python 3 和 Python 2 不兼容。在 Python 3 发布的时候，Python 2.6 和 Python 3 之间的差异如此之大，以至于人们甚至没有去考虑如何过渡，而只是害怕地耸耸肩。

但后来事情发生了变化，Python 2.7 从 Python 3.1 引入了很多新功能，缩小了二者的差距。Python 的后续版本变化也趋于理智，而且我可以很高兴地宣布，现在是可以同时支持 Python 2.7 和 Python 3.3 的了，几乎没有什么困难。

关于移植应用的官方文档（http://zeromq.org/）是有的，但我不建议不折不扣地参考它。文档中讨论了很多关于 2to3 工具（将 Python 2 的代码转换为 Python 3），并且包含了一些建议，如为你的项目建立一个特殊的 Python 3 分支。

在我看来，这是一个糟糕的建议。在几年前这也许是最合适的建议，但考虑到现在 Python 2.7 和 Python 3.3 的“兼容”情况，最好还是不要使用这种方法。

注意

也有 3to2 工具，但基于上述原因，我并不建议使用。

首先，2to3 并不总是对的，它并不是万能的。它只是处理语法的转换，而不维护与 Python 2 的向后兼容，而且在任何情况下，都需要自己手工处理语义的转换。此外，运行 2to3 相当慢，因此也很难成为一个长期的解决方案。有些文档甚至建议在 `setup.py` 阶段运行它，这多少有点儿碰运气的成分。

有些文档推荐使用不同的项目分支去支持 Python 2 和 Python 3。经验表明这样维护起来

相当麻烦，而且用户会困惑该使用哪个版本。更糟糕的是，当用户提交 bug 但却没有指明用的哪个分支时你会很困惑。

更好的方法就是使用一个代码库并保持对 Python 2 和 Python 3 兼容。这也是目前我们在 OpenStack 上投入精力做的。

最后，确保代码能够对两个 Python 版本都可用的唯一方式就是单元测试。没有单元测试**不可能**知道代码在两个上下文环境和不同版本间是否能工作正常。如果应用中还没有任何测试[①]，那么首先要做的就是大幅增加代码覆盖率，可以回顾一下第 6 章中的内容。

tox 是对多个 Python 版本进行自动测试的很好的工具，在 6.7 节中已经介绍过了。

一旦有了单元测试并配置了 tox，就很容易使用下面的方法对两个 Python 版本运行测试：

```
tox -e py27,py33
```

根据提示的错误进行修改，重新运行 tox，直到所有测试都通过为止。如果做得对的话，错误的数量应该缓慢并稳定下降，从而最终实现代码对 Python 2 和 Python 3 的全兼容。

如果有针对 Python 写的 C 模块需要移植，那么很抱歉，关于这个没什么好说的，你只能读文档然后移植你的代码。如果可能的话使用 `cffi`（http://cffi.readthedocs.org/en/release-0.8/）选项重写可能会有用。

在后面几节中会讨论一些在不同 Python 版本间移植可能遇到的问题。这里假设你已经有了一个 Python 2 的代码库。尽管下面讨论的一些内容可能在理论上也可以用于 Python 3 到 Python 2 的移植，但我个人是肯定不会这么干的。

13.1　语言和标准库

这门语言并没有彻底地修改。我敢肯定你已经简单看过了。本书不会包含全部的修改清单，那太无趣了，而且可以在网上找到。*Porting to Python 3*（http://python3porting.com/）这本书给出了要支持 Python 3 所需做修改的良好概述。

如果你还没来得及看一下 Pyton 3 所做的语言修改，建议看一下。这是一门非常好的语言，少了很多生僻的场景，针对不同对象基类有了更加清晰的接口。你会喜欢 Python 3 的。

① 我已经听说过有这样的项目存在。

但它也带来了巨大的兼容性问题。某些语句的语法变化（如异常捕获）已经完全去除了对旧的 Python 版本的兼容性，而且如果使用的话，处理起来会非常痛苦。在 1.4 节中讨论的 `hacking` 工具能帮你解决这些不兼容的用法，并避免引入更多。

支持 Python 的多版本时，应该尽量避免同时支持 Python 3.3 和早于 Python 2.6 的版本。Python 2.6 是第一个为向 Python 3 移植提供足够兼容性的版本。

影响你最多的可能是字符串处理方面。在 Python 3 中过去称为 **`unicode`**，现在叫 **`str`**（如图 13-1 和图 13-2 所示）。这意味着任何字符串都是 Unicode 的，也就是说 `u'foobar'` 和`'foobar'`是同一样东西[①]。

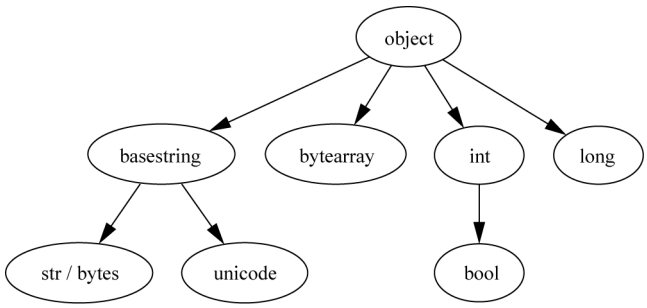


图 13-1 Python 2 基类

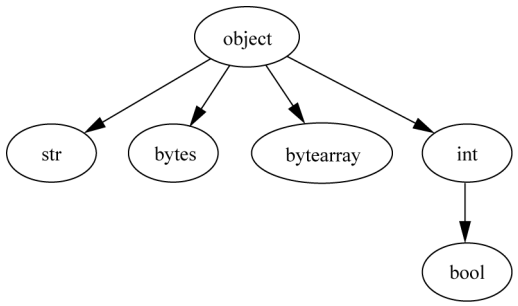


图 13-2 Python 3 基类

实现 `unicode` 方法的类应该将其重命名为 `str`，因为 `unicode` 方法将不再使用。可以通过一个类装饰器自动完成这个过程。

① `u` 前缀在 Python 3.0 中被移除了，但在 Python 3.3 中又被加了回来，参考 PEP 414（https://www.python.org/ dev/peps/pep-0414/）。

```
# -*- encoding: utf-8 -*-
import six

# This backports your Python 3__str__ for Python 2
def unicode_compat(klass):
    if not six.PY3:
        klass.__unicode__ = klass.__str__
        klass.__str__ = lambda self: self.__unicode__().encode('utf-8')
    return klass


@unicode_compat
class Square(object):
    def __str__(self):
       return u"■ " + str(id(self))
```

这种方式可以针对所有返回 Unicode 的 Python 版本实现一个方法，装饰器会处理兼容性问题。

另一个处理 Python 和 Unicode 的技巧是使用 `unicode_literals`，它从 Python2.6 开始提供[①]。

```
>>> 'foobar'
'foobar'
>>> from __future__ import unicode_literals
>>> 'foobar'
u'foobar'
```

许多函数不再返回列表而是返回可迭代对象（如 `range`）。此外，字典方法（如 `keys` 或者 `items`）现在也返回可迭代对象，而函数 `iterkeys` 和 `iteritems` 则已经被删除。这是一个巨大的变化，但 `six`（将在 13.3 节中讨论）将帮你处理这个问题。

显然，标准库也经历了从 Python 2 到 Python 3 的演化，但无需过分担心。一些模块已经被重命名或者删除，但最终呈现的是更为清晰的布局。我不知道是否有官方的清单，但是 http://docs.pythonsprints.com/python3_porting/py-porting.html 就有一份很好的清单，或者也可以用搜索引擎找到。

`six` 模块会在 13.3 节中讨论，它对于维护 Python 2 和 3 的兼容性很有用。

① 另一个不支持老版本的原因。

13.2 外部库

你的头号敌人就是所依赖的外部库。如果你读了我在 2.3 节（外部库）中的建议并且参考了我的检查表，在这里你就不会遇到麻烦了。因为那个检查表已经包含了对 Python 3 支持的需求。不过，可能你很早就开始了自己的项目并且已经犯了这个错误。

遗憾的是，对于这个问题没有什么特别的好办法。但幸运的是，如果你参考了我的其他建议，将外部库进行足够的隔离以避免其扩散到整个代码库，则是可以考虑替换它的。事实上，如果某个库不太可能支持 Python 3 的话，那么这可能是最好的办法。不过，中小型的库比大的框架更容易移植到 Python 3，所以你可能需要对它们做一点儿尝试。

在寻找 PyPI 上的包时，可以查看它的收藏分类符`"Programming Language :: Python :: 2"`和`"Programming Language :: Python :: 3"`，它指明了包所支持的 Python 版本。不过，要注意的是它们可能不是最新的。

在 OpenStack 项目早期所做出的关于外部库的一个选择就是 `eventlet`（https://github.com/eventlet/eventlet），一个并发网络库。它不支持 Python 3，而且仍然试图支持 Python 2.5（可以想象，这不利于移植）。这个决定是 OpenStack 在很早之前做出的，当时还没有进行任何的 Python 3 兼容性检查。我们已经意识到这个模块在未来会是个大问题，但截止到目前，如何解决还没有具体的计划。

千万别犯同样的错误！

13.3 使用 `six`

正如我们所看到的，Python 3 破坏了与早期版本间的兼容性并且周边很多东西发生了变化。但是，这门语言的基础并没有发生变化，所以是可以实现一种转换层的，也就是一个能实现向前和向后兼容的模块——Python 2 和 Python 3 之间的桥梁。

这样的模块是有的，名字就叫做 **`six`**（http://pythonhosted.org/six/），因为 2 乘以 3 等于 6。

`six` 首先要做的就是提供一个名为 **`six.PY3`** 的变量。它是一个布尔值，用来表明是否正在运行 Python 3。对于任何有两个版本（Python 2 和 Python 3）的代码库而言这都是一个关键变量。不过在用的时候要谨慎，如果代码中到处都是 `if six.PY3`，那么后续会很难

维护。

正如在 8.1 节中所讨论的，Python 3 有一个非常好的功能能够返回可迭代对象而不是列表。这意味着类似 dict.iteritems 这样的方法将会消失，并且 dict.items 将返回一个迭代器而不是列表。显然，这会破坏你的代码。six 对此提供了 six.iteritems，使得所有要做的只是将

```
for k, v in mydict.iteritems():
    print(k, v)
```

替换为

```
import six

for k, v in six.iteritems(mydict):
    print(k, v)
```

看，Python 3 的兼容性立刻就解决了！six 提供了大量类似的辅助函数以提升不同版本间的兼容性。

raise 语法在 Python 3 中也发生了变化[①]，因此再次抛出异常应该使用 six.reraise。

如果正在使用元类，Python 3 对其进行了彻底修改。six 针对这个转换有一个不错的技巧。例如，如果正在使用 **abc** 抽象基类元类，则可以像下面这样使用 six：

```
import abc
from six import with_metaclass

class MyClass(with_metaclass(abc.ABCMeta, object)):
    pass
```

谈到 Python 3 必然会涉及其引入的字符串和 Unicode 混乱问题。在 Python 2 中，字符串的基本类型是 str，其只能用来处理 ASCII 码字符串。而后来加入的 unicode 类型，则用来处理文本的真正字符串。在 Python 3 中，基本的类型仍然是 str，但它共享了 Python 2 中 unicode 类的属性，并能处理更为高级的编码。bytes 类型代替 str 类型，用来处理基本的字符流。

six 提供了一组不错的函数和常量用来处理这种转换，如 six.u 和 six.string_types。同样对整数也提供了相应的兼容性，通过 six.integer_types 能够处理在 Python 3 中移除的

① 现在只接受一个参数，一个异常。

long 类型。

如同在 13.1 节中讨论的，有些模块已经变动了，因此 six 提供了一个不错的名为 six.moves 的模块，用来透明地处理这些变动。

例如，在 Python 3 中 ConfigParser 模块被重命名为 configparser。因此，在 Python 2 中使用 ConfigParser 的代码：

```
from ConfigParser import ConfigParser

conf = ConfigParser()
```

就可以修改成下面的方式以兼容主要的 Python 版本：

```
from six.moves.configparser import ConfigParser

conf = ConfigParser()
```

提示

也可以通过 six.add_move 添加自己的变动来处理其他转换。

six 库可能不足以覆盖你的所有用例。在这种情况下，构建一个封装了 six 的兼容模块是值得的。通过在一个特殊的模块中隔离这个，可以确保未来有能力针对 Python 的后续版本做一些增强，或者在你不再需要继续支持某个特定 Python 版本的时候销毁（部分的）它。six 是开源的，因此你可以直接贡献它而不用维护自己的兼容模块。

最后需要提及的是 modernize 模块（https://pypi.python.org/pypi/modernize）。它是在 2to3 之上的一层很薄的包装器，用来通过迁移代码到 Python 3 使其“现代化”。但是不同于单纯转换语法为 Python 3 代码，它使用 six 模块。与标准的 2to3 工具相比，它是更好的选择，通过执行大多数繁重的工作使你的移植工作有个良好的开端。还是值得试试的。

第 14 章 少即是多

本章中汇总了我发现的一些有意思的更为高级的功能，它们有助于写出更好的代码。

14.1 单分发器

我经常说 Python 是 Lisp 的一个很好的子集，并且随着时间的推移，我越来越觉得这话是对的。最近我偶然发现了 PEP 443（https://www.python.org/dev/peps/pep-0443/），它描述了一种与 CLOS（Common Lisp Object System）提供的方式类似的泛型函数分发方式。

如果你熟悉 Lisp 的话，对这些应该并不陌生。Lisp 对象系统是 Common Lisp 的一个基本组件，提供了一种很好的定义和处理方法分发的方式。这里会先展示一下 Lisp 中的泛型方法——尽管在一本 Python 书中包含 Lisp 代码更多是为了好玩儿！

一开始让我们先定义几个非常简单的类，没有任何父类和属性：

```
(defclass snare-drum ()
  ())

(defclass cymbal ()
  ())

(defclass stick ()
  ())

(defclass brushes ()
  ())
```

上面的代码定义了几个类：`snare-drum`、`symbal`、`stick` 和 `brushes`。它们不包括任何父类和属性。这些类组成了一套架子鼓，我们可以将它们组合起来并发出声音。于是，

我们定义一个 play 方法接收两个参数，并返回声音（以字符串形式）。

```
(defgeneric play (instrument accessory)
  (:documentation "Play sound with instrument and accessory."))
```

这只定义了一个泛型方法：它并不依附于任何类，所以还不能被调用。在这个阶段，只是通知对象系统，这个方法是个泛型方法，可以通过各种参数调用。现在我们来实现这个方法的不同版本从而模拟演奏军鼓。

```
(defmethod play ((instrument snare-drum) (accessory stick))
  "POC!")

(defmethod play ((instrument snare-drum) (accessory brushes))
  "SHHHH!")
```

现在代码中已经定义了具体方法。他们接收两个参数：instrument（乐器），它是军鼓的一个实例；accessory（附件），它是 stick（鼓槌）或者 brushes（刷子）的一个实例。

在这个阶段，应该可以看出这一系统和 Python（或类似）的对象系统的第一个主要区别：方法并没有绑定到任何特定的类上。这个方法是**通用的**，并且任何类都可以实现它们。

让我们来试试。

```
* (play (make-instance 'snare-drum) (make-instance 'stick))
"POC!"

* (play (make-instance 'snare-drum) (make-instance 'brushes))
"SHHHH!"

* (play (make-instance 'cymbal) (make-instance 'stick))
debugger invoked on a SIMPLE-ERROR in thread
#<THREAD "main thread" RUNNING {1002ADAF23}>:
  There is no applicable method for the generic function
    #<STANDARD-GENERIC-FUNCTION PLAY (2)>
  when called with arguments
    (#<CYMBAL {1002B801D3}> #<STICK {1002B82763}>).

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.
```

```
   restarts (invokable by number or by possibly-abbreviated name):
     0: [RETRY] Retry calling the generic function.
     1: [ABORT] Exit debugger, returning to top level.

   ((:METHOD NO-APPLICABLE-METHOD (T)) #<STANDARD-GENERIC-FUNCTION PLAY (2)>
#<CYMBAL {1002B801D3}> #<STICK {1002B82763}>) [fast-method]
```

如你所见，调用哪个函数取决于参数的类——对象系统根据传递哪个类作为参数，为我们将函数调用**分发**给正确的函数。如果以对象系统不知道的实例调用 play，会抛出错误。

继承同样也被支持，与 Python 中的 super() 类似的（更为强大且不那么容易出错的）实现是通过(call-next-method)。

```
    (defclass snare-drum () ())
   (defclass cymbal () ())

   (defclass accessory () ())
   (defclass stick (accessory) ())
   (defclass brushes (accessory) ())

   (defmethod play ((c cymbal) (a accessory))
     "BIIING!")

   (defmethod play ((c cymbal) (b brushes))
     (concatenate 'string "SSHHHH!" (call-next-method)))
```

在这个例子中，定义了 stick 和 brushes 两个类作为 accessory 的子类。play 方法会返回声音 BIIING!，不管用哪个附件实例去敲 cymbal（铙钹），除非是用 brushes 实例，即最精确的方法总能确保被调用。(call-next-method) 函数用来调用最接近的父类的方法，在本例中就是那个会返回"BIIING!"的方法。

```
   * (play (make-instance 'cymbal) (make-instance 'stick))
   "BIIING!"

   * (play (make-instance 'cymbal) (make-instance 'brushes))
   "SSHHHH!BIIING!"
```

注意，在 CLOS 中可以通过 eql specializer 为类的某一个特定实例定义专门的方法。

但如果你真的非常好奇 CLOS 提供的众多功能，建议你读一下 Jeff Dalton 作为发起人撰

写的 CLOS 简明指南（http://www.aiai.ed.ac.uk/~jeff/clos-guide.html）。

Python 通过 singledispatch 实现了这个工作流的一个简单版本，它将在 Python 3.4 中作为 functools 模块的一部分。下面是前面的 Lisp 程序的一个粗略的对应实现：

```
import functools

class SnareDrum(object): pass
class Cymbal(object): pass
class Stick(object): pass
class Brushes(object): pass

@functools.singledispatch
def play(instrument, accessory):
    raise NotImplementedError("Cannot play these")

@play.register(SnareDrum)
def _(instrument, accessory):
    if isinstance(accessory, Stick):
        return "POC!"
    if isinstance(accessory, Brushes):
        return "SHHHH!"
    raise NotImplementedError("Cannot play these")
```

这里定义了 4 个类，以及一个基本的 play 函数，它会抛出 NotImplementedError，表明默认情况下不知道该做什么。接下来可以为特定乐器——SnareDrum（军鼓）——开发此函数的特定版本。这个函数会检查传入了哪个附件类型，并返回适当的声音。如果它无法识别这个附件，则再次抛出 NotImplementedError。

如果运行这个程序，它应该像下面这样工作：

```
>>> play(SnareDrum(), Stick())
'POC!'
>>> play(SnareDrum(), Brushes())
'SHHHH!'
>>> play(Cymbal(), Brushes())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jd/Source/cpython/Lib/functools.py", line 562, in wrapper
    return dispatch(args[0].__class__)(*args, **kw)
```

```
  File "/home/jd/sd.py", line 10, in play
    raise NotImplementedError("Cannot play these")
NotImplementedError: Cannot play these
>>> play(SnareDrum(), Cymbal())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jd/Source/cpython/Lib/functools.py", line 562, in wrapper
    return dispatch(args[0].__class__)(*args, **kw)
  File "/home/jd/sd.py", line 18, in _
    raise NotImplementedError("Cannot play these")
NotImplementedError: Cannot play these
```

singledispatch 模块检查传入的第一个参数的类，并调用 play 函数的适当版本。对于 object 类，总是会运行函数的最先定义的版本。所以，如果传入的是未注册的乐器实例，则基函数会被调用。

如果急切地想试试它的话，singledispatch 函数通过 Python Package Index 已经在 Python 2.6 到 Python 3.3 中提供了（https://pypi.python.org/pypi/singledispatch/）。

正如在 Lisp 版本的代码中所看到的，CLOS 提供了可根据方法原型中定义的**任意参数**的类型分发的多分发器，不只是第一个参数。遗憾的是，Python 中的分发器被命名为 singledispatch 是有原因的：因为它知道如何根据第一个参数进行分发。Guido van Rossum 在几年前写了一篇名为 multimethod（http://www.artima.com/weblogs/viewpost.jsp?thread=101605）的短文对此进行了解释。

此外，没办法直接调用父类的函数——既没有 Lisp 中的(call-next-method)，也没有 Python 中的 super()函数。只能用一些技巧绕过这个限制。

总结：泛型函数是增强对象系统的有力方式，尽管我很高兴地看到 Python 在朝着这个方向努力，但它仍然缺少一些 CLOS 所能提供的开箱即用的高级功能。

14.2 上下文管理器

Python 2.6 中引入的 with 语句，可能会让过去的 Lisp 程序员想起以前经常用到的宏 with-*。Python 通过使用实现了上下文管理协议的对象，提供了类似的机制。

open 函数返回的对象就支持这个协议，这就是经常能看到下面这样的代码的原因：

```
with open("myfile", "r") as f:
  line = f.readline()
```

open 返回的对象有两个方法，一个称为__enter__，另一个称为__exit__。它们分别在 with 块开始和结束时被调用。

一个上下文对象的简单实现如示例 14.1 所示。

示例 14.1　上下文对象的简单实现

```
class MyContext(object):
   def __enter__(self):
      pass
   def __exit__(self, exc_type, exc_value, traceback):
      pass
```

这段代码什么都不做，但却是合法的。

你想什么时候使用上下文管理器呢？如果对象符合下面的模式，则使用上下文管理协议就比较合适：

（1）调用方法 A；

（2）执行一段代码；

（3）调用方法 B。

这里希望**调用方法 B** 必须**总是**在**调用方法 A** 之后。open 函数很好地阐明了这一模式，打开文件并在内部分配一个文件描述符的构造函数便是方法 A。释放对应文件描述符的 close 方法就是方法 B。显然，close 方法总是应该在实例化文件对象**之后**进行调用。

contextlib 标准库中提供了 **contextmanager**，通过生成器构造__enter__和__exit__方法，从而简化了这一机制的实现。可以使用它实现自己的简单上下文管理器，如示例 14.2 所示。

示例 14.2　contextlib.contextmanager 的简单用法

```
import contextlib

@contextlib.contextmanager
def MyContext():
   yield
```

例如，我曾经在 Ceilometer（https://launchpad.net/ceilometer）中对我们所建立的流水线（pipeline）架构使用过这种设计模式。简单来说，一个流水线就是一个管道，一方面传入对象，另一方面将对象分发到不同的地方。发送数据的步骤如下。

（1）调用流水线的 `publish(objects)`方法，并传入你的对象作为参数（可以调用任意多次）。

（2）一旦完成，则调用 `flush()`方法以表明当前的发布已经完成。

要注意的是，如果不调用 `flush()`方法，对象将不会被发送到管道中，或者至少不完全发送到管道中。程序员很容易忘记 `flush()`的调用，这将引起程序毫无征兆地中断。

最好能让 API 提供一个上下文管理器对象，去阻止 API 的用户犯这种错误。通过示例 14.3 所示的代码很容易实现。

示例 14.3 在流水线对象上使用上下文管理器

```
import contextlib

class Pipeline(object):
    def _publish(self, objects):
        # Imagine publication code here
        pass

    def _flush(self):
        # Imagine flushing code here
        pass

    @contextlib.contextmanager
    def publisher(self):
        try:
            yield self._publish
        finally:
            self._flush()
```

现在，当用户在使用流水线发布某些数据时，他们无需使用`_publish`或者`_flush`。用户只需请求一个使用了名祖（eponym）函数的`publisher`并使用它。

```
pipeline = Pipeline()
with pipeline.publisher() as publisher:
```

```
    publisher([1, 2, 3, 4])
```

当提供一个这样的 API 时，就不会遇到用户错误。当看到符合的设计模式时，应该尽量用上下文管理器。

在某些情况下，同时使用多个上下文管理器是很有用的。例如，同时打开两个文件以复制它们的内容，如示例 14.4 所示。

示例 14.4　同时打开两个文件

```
with open("file1", "r") as source:
    with open("file2", "w") as destination:
        destination.write(source.read())
```

记住 `with` 语句可以支持多个参数，所以应该像示例 14.5 这样写。

示例 14.5　通过一条 `with` 语句同时打开两个文件

```
with open("file1", "r") as source, open("file2", "w") as destination:
    destination.write(source.read())
```

Python 高手之路

Python 是一门优美的语言，它快速、灵活且内置了丰富的标准库，已经用于越来越多的不同领域。通常大多数关于 Python 的书都会教读者这门语言的基础知识，但是掌握了这些基础知识后，读者在设计自己的应用程序和探索最佳实践时仍需要完全靠自己。本书则不同，介绍了如何利用 Python 有效地解决问题，以及如何构建良好的 Python 应用程序。

从本书中读者将学到什么

- 最佳实践：书中给出了构建应用程序时可参考的方法和建议，帮助读者充分利用 Python 的特性，构建不会过时的应用程序。如果读者正在做一些东西，可以立刻应用本书中提及的技术去改进自己当前的工作。
- 解决问题：书中介绍了测试、移植、扩展 Python 应用程序和库等方面的实际问题并提供了相应的解决方案，还介绍了一些非常好的小技巧，讨论了一些长期维护软件的策略。
- 语言的内部机制：书中阐述了 Python 语言的一些内部机制，帮助读者更好地理解如何开发更高效的代码，并获得对这门语言内部工作原理更深刻的洞察力。
- 专家访谈录：书中包含多篇对不同领域专家的访谈，让读者可以从开源社区和 Python 社区的知名黑客那里获得意见、建议和技巧。

本书英文原版配套网址是 https://julien.danjou.info/books/the-hacker-guide-to-python。

ISBN 978-7-115-38713-4

人民邮电出版社－信息技术分社
http://weibo.com/ptpitbooks

封面设计：董志桢

分类建议：计算机／程序设计／Python
人民邮电出版社网址：www.ptpress.com.cn