



Expert Python Programming

Python 高级编程

[法] Tarek Ziadé 著
姚军 夏海轮 王秀丽 译

旧金山湾区Python社区主持人
Shannon -jj Behrens倾情作序

人民邮电出版社
POSTS & TELECOM PRESS

Python 高级编程

[法] Tarek Ziadé 著
姚军 夏海轮 王秀丽 译

人民邮电出版社
北京

图书在版编目 (C I P) 数据

Python高级编程 / (法) 莱德著 ; 姚军, 夏海轮,
王秀丽译. — 北京 : 人民邮电出版社, 2010.1
ISBN 978-7-115-21703-5

I. ①P… II. ①莱… ②姚… ③夏… ④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(2009)第202900号

版权声明

Copyright ©Packt Publishing 2008. First published in the English language under the title Expert Python Programming.

All Rights Reserved.

本书由英国 Packt Publishing 公司授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有，侵权必究。

Python 高级编程

-
- ◆ 著 [法] Tarek Ziadé
 - 译 姚军 夏海轮 王秀丽
 - 责任编辑 刘映欣
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
 - 邮编 100061 电子函件 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：20
 - 字数：422 千字 2010 年 1 月第 1 版
 - 印数：1~3 000 册 2010 年 1 月北京第 1 次印刷

著作权合同登记号 图字：01-2009-3160 号

ISBN 978-7-115-21703-5

定价：45.00 元

读者服务热线：(010) 67132705 印装质量热线：(010) 67129223
反盗版热线：(010) 67171154

内 容 提 要

本书通过大量的实例，介绍了 Python 语言的最佳实践和敏捷开发方法，并涉及整个软件生命周期的高级主题，诸如持续集成、版本控制系统、包的发行和分发、开发模式、文档编写等。本书首先介绍如何设置最优的开发环境，然后以 Python 敏捷开发方法为线索，阐述如何将已被验证的面向对象原则应用到设计中。这些内容为开发人员和项目管理人员提供了整个软件工程中的许多高级概念以及专家级的建议，其中有些内容的意义甚至超出了 Python 语言本身。

本书针对具备一定 Python 基础并希望通过在项目中应用最佳实践和新的开发技术来提升自己的 Python 开发人员。



序 言

Python 已经出现很长时间了。

曾几何时，我坚持使用 Python，许多公司都认为我疯了。现在，Python 编码人员已经供不应求了。诸如 Google、YouTube、VMware 和 DreamWorks 等重要的公司都在不断地争夺能找到的 Python 人才。

Python 过去一贯落后于 Perl，因为 Perl 拥有 CPAN。而现在，`setuptools` 和 PyPI 已经引发了高可用的、高质量的第三方 Python 程序库的大爆发。Python 也曾经落后于 Java Servlets 和 Ruby on Rails，因为没有标准的用于与 Web 服务器交互的 API。现在，Web 服务器网关接口（WSGI）引领了 Python Web 世界的复兴。有了 Google App Engine，我想我们还将看到更多。

Python 似乎对很固执并对简洁性有真正品味的编程人员具有吸引力。很少有人因为学校里的学习任务或者大公司都在使用 Python 而成为 Python 编程人员。人们只有在发现了 Python 的内在美才会沉迷于它。因此，Python 的书多得令人吃惊。我没有足够的统计数字来证明，但是，似乎 Python 的编程书籍要多于其他语言。然而，一直没有出现足够高级的 Python 书籍，直至本书的出现。

本书介绍了一系列有趣的主题。将介绍 Python 的一组特性，以及以意想不到的方式使用它们的方法。此外，还介绍了一组精选的、有趣的第三方程序库和工具，以及使用 Python 工具和程序库的敏捷编程方法。这包括基于 `nose` 的测试驱动开发，基于 `doctest` 的文档驱动开发，使用 `Mercurial` 进行源代码控制，使用 `Buildbot` 实现持续集成，以及使用 `Trac` 完成项目管理。最后，介绍了一些更传统的主题，如剖析、优化以及诸如 Alex Martelli 的 Borg 方法，还介绍了诸如单例之类的设计模式。

如果你正打算从了解 Python 进步到精通 Python，那么本书正适合你。实际上，这正是 5 年前我所希望拥有的书。我花费了数年，通过踏踏实实地参加 PyCon 和本地的 Python 用户组而得到的一切，现在已经都在这一本简洁的书当中了。

没有什么比成为 Python 编程人员更激动人心的了！

Shannon-jj Behrens
旧金山湾区 Python 兴趣团体主持人

关于作者

Tarek Ziadé 是巴黎 Ingениweb 公司的 CTO，其工作方向为 Python、Zope、Plone 技术和质量保证。他参与 Zope 社区已经有 5 年了，并且曾经为 Zope 自身贡献过代码。

Tarek 创建了 Afpy，这是法国的 Python 用户组，并且编写了两本法语的 Python 书籍。他还在诸如 Solutions Linux、Pycon、OSCON 和 EuroPython 等法国及国际会议上发表了许多演讲，并且主持了许多课程。

我要感谢在编写本书时帮助过我的所有人。

首先感谢整个 Python 社区、AFPY 用户组，感谢 Stefan Schwarzer 关于优化的讲义以及他的引用和了不起的反馈和评论，感谢 Georg Brandl 对第 10 章中 Sphinx 部分的评审，Peter Bulychev 对 CloneDigger 部分的协助，Ian Bicking 对 minimock 部分的协助，Logilab 团队对 PyLint 部分的协助，感谢 Gael Pasgrimaud、Jean-François Roche 和 Kai Lautaportti 在 collective.buildbot 之上的工作，感谢 Cyrille Lebeaupin、Olivier Grisel、Sebastien Douche 和 St é fane Fermigier 对本书的审阅。感谢 OmniGroup 和他们了不起的 OmniGraffle 工具，本书中的所有图都是用它制作出来的（参见 <http://www.omnigroup.com/applications/OmniGraffle>）。

特别感谢 Shannon "jj" Behrens 对本书的深入评审。



关于审校人员

Shannon -jj Behrens 是旧金山湾区 Python 兴趣团体主持人。在对 Python 书籍进行技术评审和不断忙碌工作之余，他享受着和 4 个孩子的游乐时光。

我要感谢 Tarek 耐心地听取我的批评。我还要感谢我可爱的妻子 Gina-Marie Behrens，她使我有足够的时间完成本书的编辑而免受孩子的搅扰。

Paul Kennedy 是 Sydney 科技大学工程和信息技术系的高级讲师。他还是 Quantum 计算和智能系统公司 UTS 中心的知识架构实验室主任。Kennedy 博士从 1989 年开始其跨越工业界和学术界的职业生涯，专注于开发软件。他曾经使用包括 C/C++ 和 Python 在内的多种语言完成了不同领域的软件开发，诸如计算机图形、人工智能、信息生物学及数据挖掘。在最近 10 年中，他主要的工作是教授软件工程和数据挖掘的大学及研究生课程。他于 1998 年完成其计算机科学的博士课程，并且常常作为工业界数据挖掘项目的技术顾问。他是 2006-2008 年澳洲数据挖掘协会的主席，曾经积极向国际编程委员会投稿，参与国际杂志的评审，并且著有 30 种出版物。

Wendy Langer 最早在玩 Hunt the Wumpus 和 Colossal Caves 游戏的间隙学习了 Microbee Basic 编程，这是很久以前的事了。许多年后，她在大学的物理系里学习了 Fortran。最终，在长期徘徊于黑暗中之后，她终于发现了完美的编程语言——Python。尽管现在很多时间还是花在 C++ 编程上，但是她的心始终属于 Python。

作为一名 Web 开发人员，她使用过 Python、Zope、Django、MySQL 和 PostgreSQL 等技术。她也是 Packt 公司出版的由 Ayman Hourieh 编写的 *Learning Website Development with Django* 一书的技术评审。

我要感谢我的母亲及小狗 Jesse，他们保护我在评审本书时免遭许多本地危险物种（如负鼠、猫和邮差）的攻击。

前　　言

Python 很棒！

从 20 世纪 80 年代末出现的最早版本到当前的版本，它一直遵循着相同的理念不断发展：提供一个强调可读性和生产力的多范式语言。

人们曾经将 Python 看作一种新的脚本语言，认为不应该用它来建立大型系统。但是随着岁月流逝，在一些公司的努力下，显然，Python 可以用于构建几乎所有类型的系统。

实际上，许多其他语言的开发人员也醉心于 Python，并将其作为第一选择。

本书展现了作者多年构建各种 Python 应用程序的经验，包括从一两个小时就完成的很小的系统脚本，到许多开发人员历经数年编写的很大的应用程序。

它描述了开发人员使用 Python 的最佳实践。

本书名为《Python 高级编程》，这是因为它包含了一些不关注于语言本身，而更多关注于利用它的工具和技术。

换句话说，本书描述了高级的 Python 开发人员每天的工作方式。

本书内容

第 1 章介绍如何安装 Python，以确保所有读者有最接近的标准化环境。因为本书不是针对初学者的，所以本章差点被删除。但是，因为有些有经验的 Python 开发人员没有意识到这里提到的一些事情，所以最终仍然还是将它保留下来了。如果读者已经很了解这些内容，不要感到失望，因为本书其他的部分应该能够满足你的需要。

第 2 章是关于类级别以下的语法最佳实践。它将以高级的方式介绍迭代程序、生成器和描述符。

第 3 章也是关于语法的最佳实践，但是它将关注于类级别之上。

第 4 章是关于如何选择好名称的。这是用命名最佳实践对 PEP8 的扩展，还给出了一些如何设计良好 API 的提示。

第 5 章说明了编写包和使用模板的方法，然后关注于发行和分发代码的方法。

第 6 章是第 5 章的扩展，描述了编写完整应用程序的方法。它通过一个小的 Atomisator

案例进行示范。

第 7 章的主题是 `zc.buildout`，这是一个用于管理开发环境和发行应用程序的系统，其广泛地用于 Zope 和 Plone 社区，现在也开始在 Zope 世界之外使用了。

第 8 章介绍了对项目代码库管理的一些深入观察，并说明了建立持续集成的方法。

第 9 章介绍通过迭代和增量方法管理软件生命周期的方法。

第 10 章的主题是文档，并且给出了一些关于技术协作和 Python 项目文档的提示。

第 11 章阐述了测试驱动开发及其所用的工具。

第 12 章是关于优化的，给出了剖析技术和优化策略指南。

第 13 章是对第 12 章的扩展，提供了一些使程序运行更快的解决方案。

第 14 章用一组有用的设计模式结束本书。

最后，大家请密切关注 <http://atomisator.ziade.org>，这是为本书英文版构建的网站。它拥有本书中所有的代码，以及勘误表和其他额外软件。

本书所需环境

本书是为在 Linux、Mac OS X 或 Windows 之下工作的开发人员编写的。第 1 章中介绍了所有的先决条件，以确保系统能够启用 Python 并满足基本需求。

这对于 Windows 开发人员很重要，因为他们需要确保拥有与 Mac OS X 和 Linux 用户所拥有的相近的命令行环境。一般来说，所有示例都可以在任何平台上工作。

最后请记住，本书不是用于代替在线资源的，而是用于补充它们。所以，需要通过所提供的互联网链接来完成某些方面的延伸阅读。

本书读者

本书是为希望进一步精通 Python 的开发人员编写的。本书的某些部分（如持续集成）是面对项目领导者的。

本书是对讲解“如何进行 Python 编程”的常规参考书和在线资源的补充，并且更深入地讲解了语法的使用。

本书还说明了敏捷编码的方法。虽然这适用于任何语言，但本书更聚焦于 Python 实例。所以，如果没有实施测试或者使用版本控制系统，将可能通过本书学到许多甚至在其他语言上都有帮助的内容。

从测试驱动开发到分布式版本控制系统和持续集成，读者将学习到大项目上有经验的

Python 开发人员所使用的最新编程技术。

虽然这些主题都在快速发展着，但是本书不会过时，因为它关注于为什么而不是具体怎么做。

所以，即使书中介绍的特定工具已经不再使用，但仍能理解它为什么有用，并能够以专业的眼光选择一个正确的工具。

阅读须知

本书中有许多用于区分不同信息的文本样式。以下是一些样式的示例及其意义的解释。

代码文本如下所示。这个环境可以用 buildout 命令建立。

```
>>> from script_engine import run  
>>> print run('a + b', context={'a': 1, 'b':3})  
4
```

命令行输入和输出如下所示：

```
$ python setup.py --help-commands
```



警告或重要的注释。



技巧和诀窍。

下载本书的示例代码

本书示例代码的下载链接是http://www.packtpub.com/files/code/4947_Code.zip。

所下载的文件中提供了使用说明。

作者的网站<http://atomisator.ziade.org>上也有本书中提到的代码。

第 1 章

准备工作

Python 是一种对开发人员很有价值的语言。

不管你或你的客户使用的是什么操作系统，它都能够正常工作。除非使用平台相关的功能，或者使用了特定平台的程序库，否则就可以跨平台使用。例如，可以将在 Linux 环境下开发的程序部署到安装了其他操作系统的平台上。不过，这并不是其独有的特性（Ruby、Java 等许多语言都能够做到）。综合考虑本书介绍的其他功能，Python 将是公司首选开发语言之一。

本章将介绍开始使用 Python 之前必须掌握的知识，不管使用的是哪种运行环境。主要包括：

- 如何安装 Python；
- 如何使用并增强其命令行；
- 如何通过安装 `setuptools` 扩展 Python；
- 如何配置开发环境，包括老款或新型方法。

如果你对 Python 很熟悉，并且安装了自己喜欢的代码编辑器，那么建议你跳过本章的 1.1 节，直接阅读其他章节，在那里可以找到一些增强编程环境的小技巧。不过，请一定不要跳过关于 `setuptools` 的小节，对于本书其他章节而言它是必须安装的工具。

如果使用的是 Windows 操作系统，那么必须确保安装了本章中介绍的所有软件，运行本书中提供的所有示例都将依赖于这些环境。

1.1 安装 Python

Python 语言能够在各种操作系统中使用，包括 Linux、Macintosh 和 Windows。在 Python

网站的主下载页面 <http://www.python.org/download> 中可以找到由 Python 核心开发团队提供的各种版本。针对其他平台的版本是由社区中的其他开发人员维护的，在“贡献（dedicated）”页面中有相关信息（参见 <http://www.python.org/download/other>）。在此，甚至可以找到早年所用操作系统的版本。



如果有一台电脑，就可以使用 Python，而不管这台电脑上安装的是什么操作系统。
否则，将其丢在一边吧。

在安装 Python 之前，先来看看各种现有的实现版本。

1.1.1 Python 实现版本

Python 的主实现版本是用 C 语言编写的，被称为 CPython。当人们提到 Python 时，通常指的就是它。随着该语言的不断演化，C 语言实现也随着改变。除了 C 语言实现之外，Python 也提供了其他的实现版本，以保持对主流的跟踪。这些实现版本通常比 CPython 要落后几个里程碑，但也为 Python 在特定环境中的使用和宣传提供了巨大的机会。

1. Jython

Jython 是 Python 语言的 Java 实现版本。它将其代码编译成 Java 字节码，因此开发人员可以在 Python 模块（在 Python 中，存储代码的文件被称为模块）中自由地使用 Java 类。Jython 让大家可以在诸如 J2EE 之类的复杂应用程序上，将 Python 作为顶层脚本语言使用。同时，也可以将 Java 应用程序引入到 Python 应用程序中。使 Apache Jackrabbit（一种基于 JCR 实现的文档仓库 API 能够应用于 Python 程序，就是 Jython 存在价值的一个良好示例。Jython 的当前版本是 2.2.1，Jython 开发团队正在研发 2.5 版本。现在许多诸如 Pylons 之类的 Python Web 框架正在通过 Jython 被移植到 Java 世界中。

2. IronPython

IronPython 将 Python 引入了.NET 环境。该项目是由微软提供支持的，IronPython 的主要开发人员供职于该公司。其最新的稳定版本是 1.1（发布于 2007 年 4 月），它是对 Python 2.4.3 版本的实现。它能够在 ASP.NET 环境中使用，开发人员在.NET 应用程序中使用 Python 代码的方法和在 Java 环境中使用 Jython 一样。它对于该语言的推广起着十分重要的作用。.NET 社区和 Java 社区一样，都是最大的开发人员社区之一。TIOBE 社区索引也显示了.NET 语言

是一颗冉冉升起的新星。

3. PyPy

PyPy 或许是各种实现版本中最有趣的一款，其目标是用 Python 语言重写 Python。在 PyPy 中，Python 解释程序本身就是用 Python 编写的。对于 Python 实现版本之一的 CPython 而言，需要一个 C 代码层来承载具体的工作。但在 PyPy 实现版本中，这个 C 代码层将彻底用纯 Python 语言重写。这就意味着可以在运行态时改变 Python 解释程序的行为，以及实现一些在 CPython 实现版本中难以实现的代码模式（参见 <http://codespeak.net/pypy/dist/pypy/doc/objspace-proxies.html>）。PyPy 的运行速度远低于 CPython，不过在近几年中有了很大改善。随着诸如 JIT 编译器之类的技术的引入，其运行速度的提升是很有希望的。它当前的速度因子大概在 1.7~4 之间，针对的目标版本是 Python 2.4。PyPy 可以被看作汇编程序领域开发的领军项目，它在许多领域的创新都是先驱，其他主流的实现版本必将会从中受益。总体来看，PyPy 的开发更多是出于学术研究的动机，关注该项目的都是那些热衷于深入研究语言内部机制的人。因此，通常不会在具体的产品中使用它。

4. 其他实现版本

Python 还有许多其他实现版本和移植版本。例如，Nokia 就在其 S60 系列手机上实现了 Python 2.2.2。Michael Lauer 开发维护了一个应用于 ARM 芯片上的 Linux 环境中的 Python 移植版本，使其能够在诸如 Sharp Zaurus 之类的设备上使用。

这样的例子还有很多，不过本书关注的是在 Linux、Windows 和 Mac OS X 上安装 CPython。

1.1.2 在 Linux 环境下安装

如果使用的是 Linux 操作系统，那么或许已经安装了 Python。因此，可以试着在 shell 环境中调用它，如下所示。

```
tarek@dabox: ~ $ python
Python 2.3.5 (#1, Jul 4 2007, 17:28:59)
[GCC 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如果找到该命令，那么将进入 Python 提供的交互式 shell 环境，其提示符为>>>。同时还显示编译器相关的信息，包括编译 Python 的语言（在此是 GCC）以及目标系统环境（在此是 Linux）等信息。如果使用的是 Windows，那么可以用微软公司的 Visual Studio 作为编

译器。同时，Python 的版本号也将会显示出来。请确保运行的是最新的稳定版本（在本书上市时或许已经是 2.6 版本了）。

如果没有预装，也可以自己动手在系统上安装任意版本的 Python，所需的操作都是很普通的。Python 的版本可以从其文件全名中得知，也可以通过 Python 命令来了解。当然，是否能够执行该命令，取决于环境变量 path 的设置，如下所示。

```
tarek@dabox: ~ $ which python  
/usr/bin/python  
tarek@dabox: ~ $ python<tab>  
python  python2.3  python2.5  
python2.4
```

如果没有找到 python 命令（这在 Linux 系统中并不常见），还可以通过 Linux 系统中的软件包管理工具安装它，例如 Debian 中提供的 apt、Red Hat 中提供的 rpm 等。当然，也可以通过编译源代码的方式来安装。

虽然始终通过软件包管理工具安装是个好习惯，但在此我们仍然将对两种安装方法（通过软件包管理工具安装和通过编译源代码来安装）做进一步介绍。不过，在你的软件包管理工具中，不一定会提供 Python 的最新版本。

1. 通过软件包管理工具安装

安装 Python 最常见的方法是使用 Linux 中的软件包管理工具，这样还能使软件升级更加容易。根据使用的 Linux 发行版本的不同，可能使用以下 3 种不同的命令：

- apt-get install python 在基于 Debian 的发行版本中，诸如 Ubuntu；
- urpmi python 在基于 rpm 的发行版本中，诸如 Fedora 或 Red Hat 系列；
- emerge python 针对 Gentoo 发行版本。

如果在这些软件包安装工具中没有列出最新版本的 Python，就需要手动进行安装了。

当选择完全安装时，可能会同时安装一些其他的软件包。它们都是可选的，不一定会用到它们。不过，如果需要编写 C 扩展，或者对程序进行优化，那么将会用到它们。当选择完全安装时将同时安装以下软件包：

- python-dev 在编译 C 模块时所需的 Python 头文件；
- python-profiler 它包含了一些针对完全遵循 GPL 协议的 Linux 发行版本（诸如 Debian 或 Ubuntu）的非 GPL 协议模块（Hotshot 优化器）；
- gcc 在编译包含 C 代码的扩展时所需的软件包。

2. 通过编译源代码的方式安装

手动安装就是通过一个 cmmi 过程（顺序执行 configure、make、make install 命令）来完

成 Python 的编译并将其部署到系统中。在 <http://python.org/download> 中可以找到 Python 最新版本的源代码包。

使用 wget 下载



wget 程序是一个 Gnu 项目，是用来下载软件的命令行工具，它在所有平台中都能使用。在 <http://gnuwin32.sourceforge.net/packages/wget.htm> 中可以下载在 Windows 平台上使用的二进制版本。

在 Linux 或 Mac OS X 平台中，可以通过诸如 apt 或 MacPorts 之类的软件包管理工具来安装。

编译 Python 时需要使用 make 和 gcc。

- make 读取配置文件（通常名为 Makefile），检查编译该程序所需的各方面条件是否满足，同时将开始执行编译操作。它通过 configure 和 make 命令调用。
- gcc GNU C 编译器，是在构建程序时使用最广的开源编译器。

首先确保系统中已经安装了它们。在某些 Linux 发行版本中（如 Ubuntu），可以直接通过安装 build-essentials 软件包来安装这些构建工具。

在安装 Python 时，应执行以下命令序列。

```
cd /tmp
wget http://python.org/ftp/python/2.5.1/Python-2.5.1.tgz
tar -xzvf tar -xzvf Python-2.5.1.tgz
cd Python-2.5.1
./configure
make
sudo make install
```

采用这样的过程安装，会同时安装针对二进制安装版所需的头文件，它们通常被放在 python-dev 软件包。在源代码包中，也包含 Hotshot 优化器。当完成这些步骤之后，就可以在命令行使用 Python 了。

现在，你的系统已经支持 Python 了，让我们庆祝一下吧！

1.1.3 在 Windows 环境下安装

在 Windows 环境下编译 Python 和在 Linux 环境下类似。但这样做十分麻烦，因为还需要安装复杂的编译环境。python.org 的下载区中提供了标准的安装程序，它提供了向导式的安装过程（如图 1.1 所示），十分简单易用。



图 1.1

1. 安装 Python

如果一切都采用默认选项，Python 将被安装在 c:\Python25 目录下，而不是像其他软件那样被放在 Program Files 文件夹中。这样可以避免在环境变量 path 中出现空格。

最后一步是修改环境变量 PATH，以便能够在 DOS 命令行窗口中调用 Python。

对于绝大多数的 Windows 版本，都可以通过以下步骤完成：

- 在桌面或 start(开始)菜单的 My Computer(我的电脑)图标上单击右键，进入 System Properties(系统属性)对话框；
- 切换到 Advanced(高级)标签页；
- 单击 Environment Variables(环境变量)按钮；
- 编辑系统变量 PATH 的值，添加上两个新的路径，并用“; (分号)”分隔。

要添加的搜索路径如下：

- c:\Python25 以便能够调用到 python.exe；
- c:\Python25\Scripts 以便调用通过扩展为 Python 添加的第三方脚本。

这样，就可以在命令行窗口中运行 Python 了。具体来说，就是单击 Start(开始)菜单中的快捷方式 Run(运行)，输入命令 cmd，然后在弹出的命令行窗口中调用 python，如下所示。

```
C:\> python
Python 2.5.2 (#71, Oct 18 2006, 08:34:43) [MSC v.1310 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

这样就能够运行 Python 了。但这样的环境和 Linux 用户的环境相比还不够完整，要实现

本书中介绍的所有内容，还需要安装 MinGW。

2. 安装 MinGW

MinGW 是针对 Windows 平台开发的编译器。它实现了 gcc 编译器的各种功能，提供了相同的程序库和头文件。MinGW 可以彻底代替 Microsoft Visual C++。可以在系统中保留各种编译器，以便根据自己的需要选择不同的编译器。

首先，要下载 MinGW，找到指向 Sourceforge（参见 <http://sourceforge.net>——最大的、针对开源项目开发人员的网站）的链接。自动安装是最好的选择，它能够自动完成所有的操作。在此，可以下载安装程序并运行它。

和 Python 一样，也需要对环境变量 PATH 进行修改，增加 c:\MinGW\bin，以便能够调用它提供的命令。设置了 PATH 变量之后，就可以在命令行窗口中运行 MinGW 的各种命令，如下所示。

```
C:\>gcc -v
Reading specs from c:/MinGW/bin/..../lib/gcc-lib/mingw32/3.2.3/specs
Configured with: ./gcc/configure --with-gcc --with-gnu-ld --with-gnu-as
--host=
mingw32 --target=mingw32 --prefix=/mingw --enable-threads --disable-nls
--enable
-languages=c++,f77,objc --disable-win32-registry --disable-shared --
enable-sjlj-
exceptions
Thread model: win32
gcc version 3.2.3 (mingw special 20030504-1)
```

这些命令无法手动调用，但当 Python 编译器需要时会自动调用。

3. 安装 MSYS

在 Windows 平台中，还有一个需要安装的工具是 MSYS (Minimal SYStem)。它能在 Windows 平台上提供一个 Bourne Shell 命令行环境，在该环境中可以实现 Linux 或 Mac OS X 操作系统中常见的命令，如 cp、rm 等。

这听起来有点过分，毕竟 Windows 平台也提供了相应的工具，不仅在图形界面中提供了，在 MS-DOS 命令行窗口中也实现了。但该工具对于那些跨多个操作系统工作的开发人员而言，能够提供一个统一的命令格式。

下载 MSYS，然后将其安装到自己的系统。如果选择标准安装，那么 MSYS 将被安装在 c:\msys 目录下。因此，还需要像 MinGW 那样在 PATH 变量中增加 c:\msys\1.0\bin。

在本书中，将在所有示例中使用 Bourne Shell 命令。因此如果采用的是 Windows 平台，那么请安装 MSYS。

现在你已经安装了 MinGW 和 MSYS，再也不必羡慕那些安装了 Linux 操作系统的人，因为它们已经在 Windows 平台上模拟了 Linux 开发环境中所提供的各种主要功能。

1.1.4 在 Mac OS X 环境下安装

Mac OS X 是基于 Darwin 内核的，而 Darwin 则是基于 FreeBSD 的。这使得该平台与 Linux 很相似，也很兼容。Apple 在此之上添加了一个图形引擎（Quartz）及一个特殊的文件树。

从 Shell 角度看，主要的区别在于系统文件树的组织方式。例如，可能找不到一个名为 /home 的根文件夹，但可以找到 /Users 文件夹。应用程序通常会被安装在 /Library 目录下。虽然也有使用 /usr/bin 的，不过那是 Linux 中常用的。

和 Linux 与 Windows 平台一样，在 Mac OS X 上安装 Python 也有两种方法。可以通过软件包安装程序安装，也可以通过编译源程序的方式安装。通过软件包安装程序是最简单的方法，不过你可能会希望自己动手编译 Python。最新版本的 Python 也提供了相应的二进制安装版本。

1. 通过软件包安装程序安装

在 Mac OS X 的最新版本（本书写作时是 Leopard）中打包了 Python。要另外安装 Python，首先要从 <http://www.pythonmac.org/packages> 下载 Python 2.5.x 的二进制包。可下载到一个可以安装到系统中的.dmg 文件，其中有一个.pkg 的文件。安装界面如图 1.2 所示。

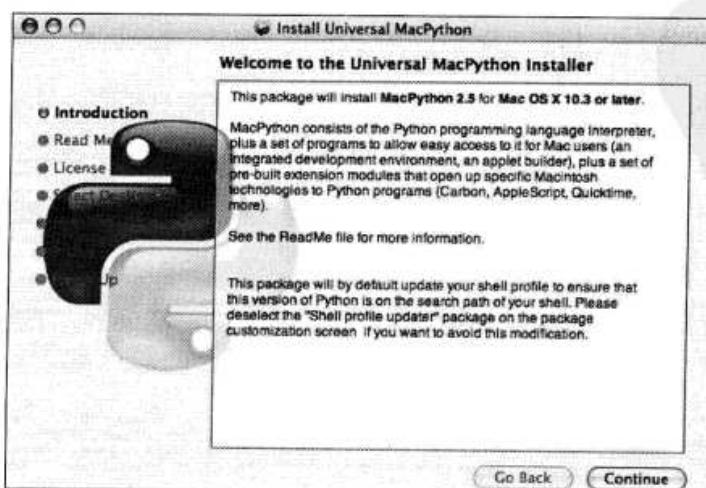


图 1.2

这样将 Python 安装到/Library 文件夹中，然后在系统中创建相应的链接，接下来就可以在 shell 中运行 Python 了。

2. 以编译源程序方式安装

如果想要编译 Python，则需要安装：

- gcc 编译器 Xcode 工具中包含该编译器，可以从安装光盘中找到它，也可以从 <http://developer.apple.com/tools/xcode> 下载；
- MacPorts 一个与 Debian 的软件包管理系统 apt 很类似的软件包管理系统，和在与 Linux 使用 apt 一样，它能够帮你安装相应的依赖包，参见 <http://www.macports.org>。

现在，可以采用与 Linux 中完全相同的步骤来完成编译工作了。

1.2 Python 命令行

执行 `python` 命令后，将出现 Python 命令行，通过它可以与 Python 解释程序进行交互。它很常用，例如，可以通过它来完成一些小型的计算，如下所示。

```
macziade:/home/tziade tziade$ python
Python 2.5 (r25:51918, Sep 19 2006, 08:49:13)
[GCC 4.0.1 (Apple Computer, Inc. build 5341)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>1 + 3
4
>>>5 * 8
40
```

当按下回车键时，Python 将解释这行程序并马上显示处理结果。该特性是从 ABC 语言中继承而来的，Python 的编程方式受它的影响很深。在代码文档中，所有的示例都是以一个命令行会话的形式展示的。

退出命令行

 如果想退出命令行，在 Linux 或 Mac OS X 平台中可以按 `Ctrl+D` 组合键，在 Windows 平台中则应按 `Ctrl+Z` 组合键。

由于命令行交互模式在编程过程中扮演了十分重要的角色，因此需要使其变得更加简单易用。

1.2.1 定制交互式命令行

交互式命令行可通过启动文件来配置。当它启动时会查找环境变量 PYTHONSTARTUP，并且执行该变量中所指定文件里的程序代码。有些 Linux 发行版本提供了一个默认的启动脚本，它通常放在用户主目录下，文件名是 .pythonstartup。按 Tab 键时自动补全内容和命令历史。它们都是对命令行的有效增强，这些工具是基于 readline 模块实现的（需要 readline 程序库）。如果没有这个启动脚本文件，也可以自己创建一个。

下面就是一个最简单的启动脚本文件，它为 Python 命令行添加了按<Tab>键自动补全内容和命令历史功能。

```
# python startup file
import readline
import rlcompleter
import atexit
import os
# tab completion
readline.parse_and_bind('tab: complete')
# history file
histfile = os.path.join(os.environ['HOME'], '.pythonhistory')
try:
    readline.read_history_file(histfile)
except IOError:
    pass
atexit.register(readline.write_history_file, histfile)
del os, histfile, readline, rlcompleter
```

在用户主目录下创建该文件，并将其命名为 .pythonstartup。然后添加环境变量 PYTHONSTARTUP，并根据该文件的路径赋予相应的值。

 在 pbp.script 包中已经提供了 python 启动脚本，名为 pythonstartup.py。从 <http://pypi.python.org/pypi/pbp.scripts> 中下载该文件，然后将文件名改为 .pythonstartup 即可。

设置环境变量 PYTHONSTARTUP

如果使用的是 Linux 或 Mac OS X 操作系统，那么最简单的方法就是在自己的用户主目录下创建启动脚本。然后将其链接到环境变量 PYTHONSTARTUP 上，并将其放到系统的 shell 启动脚本中。例如，Bash 和 Korn Shell 使用的是 .profile 文件，可以在该文件中添加如下一行设置。

```
export PYTHONSTARTUP=~/.pythonstartup
```

如果使用的是 Windows 操作系统，那么就要以 administrator 角色登录，添加一个新的环境变量，然后将脚本保存在一个公共的位置上，而非放在特定的用户区域。

当启动交互式命令行时，将会执行 .pythonstartup 脚本，这些新功能将被启用。例如，按 Tab 键自动补全内容功能能够帮助回忆起模块的名称，如下所示。

```
>>> import md5
>>> md5.<tab>
md5.__class__          md5.__file__           md5.__name__
md5.__repr__           md5.digest_size
md5.__delattr__        md5.__getattribute__   md5.__new__
md5.__setattr__        md5.md5
md5.__dict__           md5.__hash__          md5.__reduce__
md5.__str__            md5.new
md5.__doc__            md5.__init__         md5.__reduce_ex__  md5.
blocksize
```

还能通过修改该脚本来实现更多自动化功能，例如，让 Python 为模块提供一个进入点，以及提供基于类的解释程序模块（参见 <http://docs.python.org/lib/module-code.html> 中的 code module 部分）。但如果希望拥有一个更高级的交互式命令行，还有一个现成的工具——iPython 可供使用。

1.2.2 iPython：增强型命令行

iPython 项目的目标是提供一个扩展的命令行。在它所提供的各种功能中，最有意思的包括：

- 动态对象反射；
- 在命令行中调用系统 shell 功能；
- 程序调优的直接支持；

- 调试工具。

要安装 iPython，首先需要在 <http://ipython.scipy.org/moin/Download> 中下载安装程序，然后根据针对所使用的操作系统的指南来完成安装。

运行 iPython 提供的 shell 后，将出现如下提示。

```
tarek@luvdit: ~$ ipython
Python 2.4.4 (#2, Apr 5 2007, 20:11:18)
Type "copyright", "credits" or "license" for more information.
IPython 0.7.2 -- An enhanced Interactive Python.

?          -> Introduction to IPython's features.
%magic    -> Information about IPython's 'magic' % functions.
help      -> Python's own help system.
object?  -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

iPython 和应用程序调试

当需要对程序进行调试时，iPython 将是一个很友好的命令行工具，特别是针对那些以后台进程形式运行的服务器端代码而言。

1.3 安装 setuptools

Perl 拥有大量的第三方程序库，安装它们也很简单。Perl CPAN 系统使开发人员能够将一组简单的命令集以新程序库的形式发布。近几年中，在 Python 领域也出现了类似的技术，并且逐渐成了安装扩展的标准途径。它是基于：

- 一个存储在 Python 官方网站的集中式仓库，它的名字是 Python Package Index(PyPI)，其前身就是 Cheeseshop（它引用了源于 BBC 的 Monty Python 的原型）；
- 一个名为 setuptools 的包管理系统，它是基于 distutils 开发的，用来发布代码以及和 PyPI 交互。

在安装这些扩展之前，需要对总体框架做些必要的解释。

1.3.1 工作原理

Python 附带提供了一个名为 distutils 的模块，它提供了一系列用于发布 Python 应用程序

的工具。它提供的内容包括：

- 用来提供标准元数据字段（诸如作者名、版权类型等信息）的骨架；
- 一组用来将“包”（在 Python 中，包是一个由一个或多个模块组成的系统文件夹）中的代码构建软件安装包的辅助工具，不仅能够创建出一组预编译的 Python 文件，还能创建在 Windows 中可执行的安装程序。

但 `distutils` 工具仅适用于包，无法定义包之间的依赖关系。`setuptools` 通过添加一个基本的依赖系统以及许多相关功能，有效地弥补了该缺陷。它还提供了一个自动包查询程序，它可以自动获取包的依赖关系，并自动完成这些包的安装。换句话说，Python 中的 `setuptools` 相当于 Debian 中的 `apt`。

 在 Python 中准备一个 `setuptools` 封装器，是部署它的标准方法。第 5 章中对此将会有更详细的说明。

该工具现在十分流行，甚至当编写要发布的 Python 应用程序时，它几乎已经是必需的了。在近几年内，它很有希望被 Python 纳入自己的标准程序库中。在此之前，如果想拥有完整的 Python 系统，充分发挥 `setuptools` 的功能，还需要另外安装 `setuptools`，因为它还不是 Python 标准安装所涵盖的一部分。

1.3.2 使用 EasyInstall 安装 setuptools

要安装 `setuptools`，还需要安装 `EasyInstall`，它是一个软件包下载器和安装程序。该程序是对 `setuptools` 的有效补充，因为它知道如何获取相应的软件包以及如何安装它。安装它的同时也将完成 `setuptools` 的安装。

从 Peak 网站 (<http://peak.telecommunity.com/DevCenter/EasyInstall>) 中下载并运行 `ez_setup.py` 脚本程序，它的位置通常是 http://peak.telecommunity.com/dist/ez_setup.py，如下所示。

```
macziade: ~ tziade$ wget http://peak.telecommunity.com/dist/ez_setup.py
08:31:40 (29.26 KB/s) - « ez_setup.py » saved [8960/8960]
macziade: ~ tziade$ python ez_setup.py setuptools
Searching for setuptools
Reading http://pypi.python.org/simple/setuptools/
Best match: setuptools 0.6c7
...

```

```
Processing dependencies for setuptools
Finished processing dependencies for setuptools
```

如果之前安装了早期版本，将会看到一个出错信息。这时，需要使用升级选项（-U setuptools），如下所示。

```
macziade: ~ tziade$ python ez_setup.py
Setuptools version 0.6c7 or greater has been installed.
(Run "ez_setup.py -U setuptools" to reinstall or upgrade.)
macziade: ~ tziade$ python ez_setup.py -U setuptools
Searching for setuptools
Reading http://pypi.python.org/simple/setuptools/
Best match: setuptools 0.6c7
...
Processing dependencies for setuptools
Finished processing dependencies for setuptools
```

当一切都安装完之后，系统中就有一个名为 easy_install 的命令了。通过该命令可以完成任何扩展插件的安装和升级工作。例如，如果需要安装扩展插件 py.test（它是针对敏捷开发的一组工具，参见 <http://codespeak.net/py/dist>），那么只需执行以下命令。

```
tarek@luvdit:/tmp$ sudo easy_install py
Searching for py
Reading http://cheeseshop.python.org/pypi/py/
Reading http://codespeak.net/py
Reading http://cheeseshop.python.org/pypi/py/0.9.0
Best match: py 0.9.0
Downloading http://codespeak.net/download/py/py-0.9.0.tar.gz
...
Installing pytest.cmd script to /usr/local/bin
Installed /usr/local/lib/python2.3/site-packages/py-0.9.0-py2.3.egg
Processing dependencies for py
Finished processing dependencies for py
```

如果使用 Windows 平台，那么该脚本的名称是 easy_install.exe，在 Python 安装路径的 Scripts 文件夹中可以找到它。由于它和 1.1.3 小节中在环境变量 PATH 中设置的路径相同，因此也可以执行 easy_install 命令（在 Linux 和 Mac OS X 平台上，无需在调用该命令时在前面添加 sudo，因为它本身就拥有 root 特权）。

有了该工具，扩展 Python 将更加容易，并且它们依赖的包都将自动完成安装。如果在

Windows 平台下安装扩展插件时需要编译，就会自动调用 MinGW 完成这一额外的步骤。

1.3.3 将 MinGW 整合到 distutils 中

当需要编译程序时，可以在 Python 的配置文件中指定。这在 Windows 平台中也十分简单。首先在 `python-installation-path\lib\distutils` 文件夹（在 Windows 平台下则应在 `Lib` 文件夹下，第一个字母是大写）下创建一个名为 `distutils.cfg` 文件，并在该文件中添加以下内容。

```
[build]
compiler = mingw32
```

这样就能够把 MinGW 链接到 Python 中，每次 Python 需要编译包含 C 程序代码的包时，就会自动调用 MinGW。



现在已经完成开始编码之前的准备工作了！



1.4 工作环境

花些时间对工作环境进行配置，对于提高生产率而言是十分重要的。当负责一个项目时，强制要求所有开发人员使用相同的工具集总不是个好主意。更好的方法是让每个人从推荐的一组公共工具集中挑选适合自己的部分。

在一个 Python 项目中，除了编写程序之外，还包括与数据文件、诸如源代码库之类的第三方服务交互之类的工作。



开发人员除了编写代码之外，还需要花费大量的时间来完成其他工作。



配置工作环境主要有两种方法：通过一组工具集构建（传统方法），或者使用集成工具完成（新方法）。当然，还有许多折衷的方案，每个开发人员都可以根据自己的爱好和习惯进行选择。

1.4.1 使用文本编辑器与辅助工具的组合

这种环境是历史最悠久的模式，不过或许也是效率最高的一种，因此这样做能够根据自

己的工作需要有机地组合它们。如果始终使用的是同一台电脑，安装和配置自己所需工具是十分简单的。但准备一个可移植的工作环境总是更好的选择。例如，将它们绑定在一起，存储在 U 盘中，这样就可以在不同电脑中使用它们。决定在不同平台使用相同的工具也是一种很好的实践，这样能够在任何地方有效地应用它们。

可移植的 Python 和类似项目

可移植的 Python 是一个针对诸如 Windows 平台提供 Python 系统的项目，它提供了一个开箱即用的、内嵌式的 Python 和代码编辑器。如果目标环境已经安装了 Python，那么可能不需要安装这样的工作环境。不过在该项目中还是能够发现一些有趣的东西的，详情参见 <http://www.portablepython.com>。

 Damn Small Linux (DSL) 也是一个有趣的解决方案，它可以将一组工具存储在一个 U 盘中。大家可能知道，可以通过一个名为 Qemu 的系统模拟器（它可以在任何平台上运行）来运行一个嵌入式的 Linux。因此，可以在 DSL 中安装一个 Python 以实现所需的功能，详情参见 <http://www.damnsmalllinux.org/usb-qemu.html>。

Dragon 公司提供了一个能够构建可移植 Python 环境的 Ubuntu 系统。

现在，该工作环境将由以下几个部分组成：

- 在各种平台上都能够找到的源代码编辑器，可能是开源的、免费的；
- 一些扩展的二进制程序，为 Python 提供一些功能，但不能对其进行修改。

1. 源代码编辑器

和 Python 兼容的编辑器有很多。在一个由多个工具组成的工作环境中，最好是选择一个只关注于代码编写的编辑器。换句话说，在简单的代码编辑器和集成开发环境 (IDE) 之间，边界总是有些模糊的，甚至简单的编辑器也提供了与系统交互和扩展的机制。但一个配置齐全的源代码编辑器可以让你不用为多余的功能费心。

多年以来，最佳的选择是 Vim (<http://www.vim.org>) 或 Emacs (<http://www.gnu.org/software/emacs>)。乍一看，它们的界面并不友好，因为它们采用了不同的键盘快捷键标准，通常需要花费一些时间去熟悉它。不过，当熟悉了这些命令之后，会发现它们还是很有效率的工具。它提供了针对 Python 的模式，还有许多用来编辑其他类型文件的特定模式。

Vim 是一种对 Python 支持不错的编辑器，该社区对其给予了很大的兴趣。可以很方便地通过 Python 对其扩展。例如，可以看看在 PyCon 2007 大会的一场演讲的内容 (<http://www.tummy.com/Community/Presentations/vimpython-20070225/vim.html>)。



Vim 最大的优势是，多年来所有的 Linux 系统中都预装了它，因此在其他电脑甚至是服务器上都能找到它。

下一小节将介绍 Vim 的安装和配置方法。如果你更偏爱 Emacs，那么建议最好先浏览一下 <http://www.python.org/emacs>。

2. Vim 的安装和配置

Vim 的最新版本是 7.1，它提供了诸如与代码编译器绑定等优秀功能。

如果使用 Linux 操作系统，那么肯定已经安装了某个版本的 Vim，但通常其版本是低于 7.1 的。可以通过 `vim -version` 命令来检查它的版本号。如果版本低于 7.0，可以通过 Linux 系统中的软件包管理工具或者用直接编译源代码的方法升级它。

在其他操作系统中，需要另外安装 Vim。Windows 用户能够找到一个可执行的安装程序，它不仅提供了 gvim（一个提供了图形化界面的版本），还提供了相应的控制台版本。如果 Mac OS X 的用户需要使用 7.1 版本，则需要采用编译源代码方式安装，因为还没有提供最新版本的二进制安装包。

在 <http://www.vim.org/download.php> 中获取相应版本的安装包，并视实际的需要对其进行编译。

如果要在多字节字符集（如带重读符号的法文）环境下编译 Vim，那么需要在调用 `configure` 命令时带上`--enable-multibyte` 参数。其编译顺序如下所示。

```
./configure --enable-multibyte
make
sudo make install
```

这样，将在`/usr/local` 目录下安装 Vim，其二进制命令位于：

```
/usr/local/bin/vim.
```

最后还有一点，如果使用的是 Linux 或 Mac OS X 操作系统，那么就在自己的用户主目录下创建一个名为`.vimrc` 的文件；如果使用的是 Windows 操作系统，则应该将其命名为`_vimrc`。将其放保存在 Vim 的安装文件夹下，同时添加一个名为 VIM 的环境变量（其值就是该路径），这样 Vim 就能够知道从哪里获取它。

`vimrc` 文件的内容如下所示。

```
set encoding=utf8
set paste
```

```
set expandtab  
set textwidth=0  
set tabstop=4  
set softtabstop=4  
set shiftwidth=4  
set autoindent  
set backspace=indent,eol,start  
set incsearch  
set ignorecase  
set ruler  
set wildmenu  
set commentstring=\ #\ %s  
set foldlevel=0  
set clipboard+=unnamed  
syntax on
```

例如，`tabstop` 选项就是用来将`<Tab>`键的用途定义为插入 4 个空白符。

在 Vim 中，对于每个选项都可以通过： help 命令来获得它的说明。例如，使用： help rule 命令将会显示出一个与 ruler 选项相关的帮助页面，如图 1.3 所示。

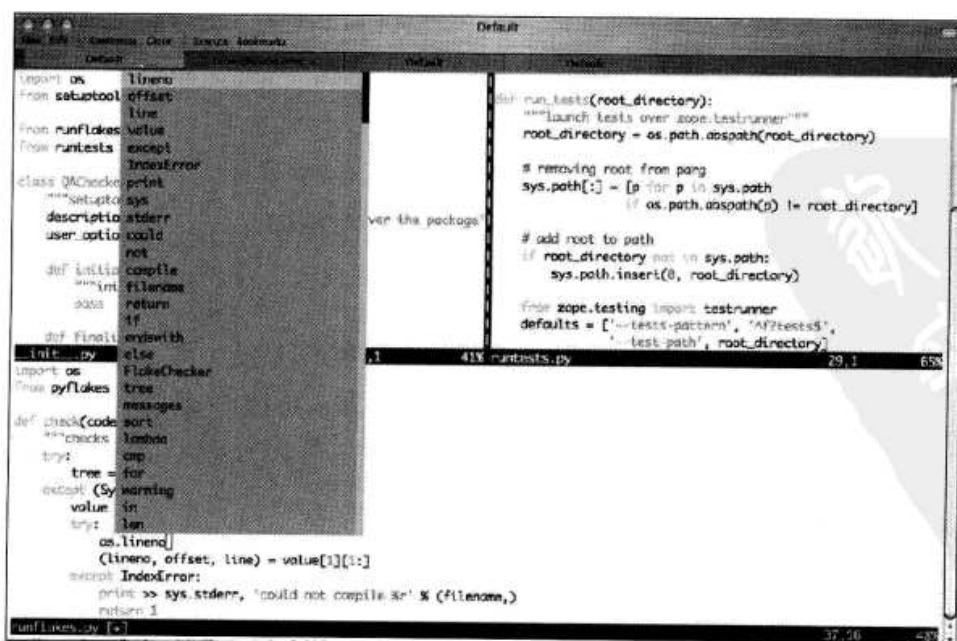


图 1.3

现在，运行 Vim 的准备工作就完成了。

3. 使用其他编辑器

如果不想使用 Vim 或 Emacs，而是想使用对鼠标操作提供更多支持的、具有可视化界面的编辑器，那么也可以选择其他编辑器。不过最好能够提供针对 Python 的模式，并满足以下标准。

- 可以将`<Tab>`键定义为插入 4 个空白符——这是一个十分重要的特性，在绝大多数编辑器都是这样处理的。如果编辑器不是这样定义的，那么建议放弃。否则，最终会对代码中的空白符和 tab 符产生混淆，从而导致编译器出错。
- 在保存文件时会去除多余的空白符。
- 在换行时，能够智能地定位鼠标指针的位置，以加快编码的效率。
- 提供标准的、基于颜色方案的强调显示模式。
- 提供简单的代码自动补全功能。

在比较不同的代码编辑器时，还有许多标准。不过有些是不太必要的，如代码折叠，而有些则是十分有用的，如 API 搜索。如果能够在编辑功能之外，提供 Python 交互命令行，则要比满足前面提及的 5 个标准还更有用。



如果真的不喜欢 Vim 或 Emacs，那么你可能属于新潮的开发人员。

4. 其他二进制程序

要使编辑器更加完善，还应该安装一些二进制程序，以满足以下一些常见的需求。

- `diff`，源于 GNU diffutils，它可以完成比较两个文件夹或文件的内容的功能。在所有 Linux 发行版本和 Mac OS X 中，都默认安装了该程序。在 Windows 平台中则需要另外安装，在 <http://gnuwin32.sourceforge.net/packages/diffutils.htm> 中可以找到安装程序。安装完成后，就可以在 DOS 窗口中使用 `diff` 命令了。
- `grep`，一个用来在文件中查找字符串的命令行工具。与系统工具相比，它的功能更强大，它在各种操作系统平台上的运行机制是一样的，在 Linux 和 Mac OS X 中也是默认提供的。在 Windows 平台需要另外安装，安装程序可以在 <http://gnuwin32.sourceforge.net/packages/grep.htm> 找到。

注意，MSYS 中已经提供了这些工具，包括 Windows 平台下的 `grep` 工具。

1.4.2 使用集成开发环境

IDE 中除了提供源代码编辑器之外，还集成了许多辅助工具。这使得其部署和使用都变

得非常快捷方便。

对于 Python 而言，现在最优秀的开源 IDE 当属 Eclipse (<http://www.eclipse.org>) 与 PyDev (<http://pydev.org>) 插件的组合（该插件不是免费的）。



商业软件中还有一款名为 Wingware 的优秀 IDE，详情可参见
<http://wingware.com>。

Eclipse 也是可移植的，它使得能够在任何电脑中以相同的方式工作。PyDev 是针对 Eclipse 的一个插件，用来添加一些 Python 特性：

- 代码补全；
- 语法格式的强调显示；
- 诸如 PyLint 和 Bicycle Repair Man 之类的质量保证（QA）工具；
- 代码覆盖；
- 集成的调试程序。

安装带 PyDev 插件的 Eclipse

Eclipse 是用 Java 语言编写的，因此首先要安装 Java 运行环境（JRE）。如果使用的是 Mac OS X，那么 JRE 已经安装了。在 Sun 的网站 <http://java.sun.com/javase/downloads/index.jsp> 中可以下载到最新版本的 JRE。下载正常的安装程序，然后根据其说明部署到系统中。

Eclipse 没有提供安装程序，因为它只是由一个文件夹加上一些 Java 脚本构成的。因此要安装 Eclipse，只需要下载一个压缩包，然后在系统中将其解压出来。然后，通过 Eclipse 界面中提供的优雅的软件包管理系统来完成插件的添加。但安装适当的插件集的确有些麻烦，因为最新的 Eclipse 版本可能和这些插件不兼容。

由于这些插件也可以绑定在压缩包中，因此最简单的方法是找一个自定义的 Eclipse 分发版本。现在还没有专门针对 Python 的分发版本，不过可以根据需要在线创建一个。

位于 <http://ondemand.yoxos.com/geteclipse/W4TDelegate> 的 Yoxos 就借助了 AJAX 安装程序实现了该功能。在该网页中，可以选择自己需要的插件，它将根据选择生成一个可下载的压缩包。搜索名为 PyDev 的插件，然后双击它，以便添加到左边的插件列表中。添加该插件时，其依赖的其他插件也会被同时加入。然后，单击右上角的 Download（下载）按钮获得这个自定义的压缩包，如图 1.4 所示。

对这个压缩包进行解压，例如，在 Windows 平台中将其解压到 c:\Program Files\Eclipse 目录，在 Linux 或 Mac OS X 系统中将其解压到用户主目录下。在这个文件夹中，会发现一个能启动该应用程序的快捷链接，如图 1.5 所示。至此，就做好运行 Eclipse 的准备工作。

作了。

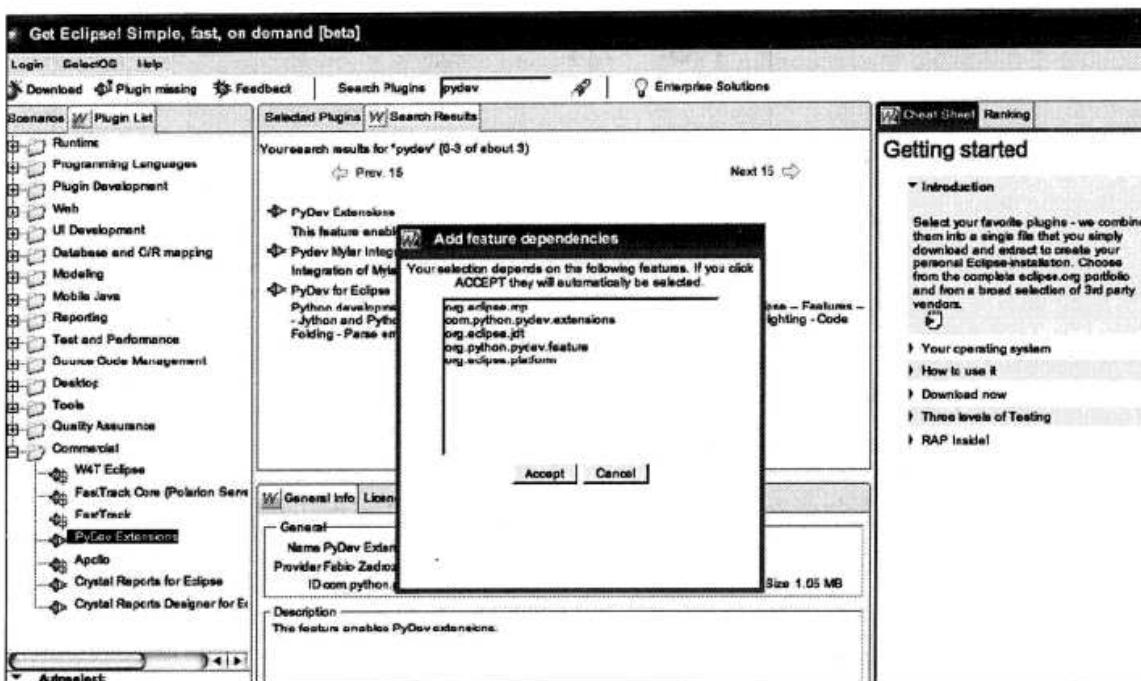


图 1.4

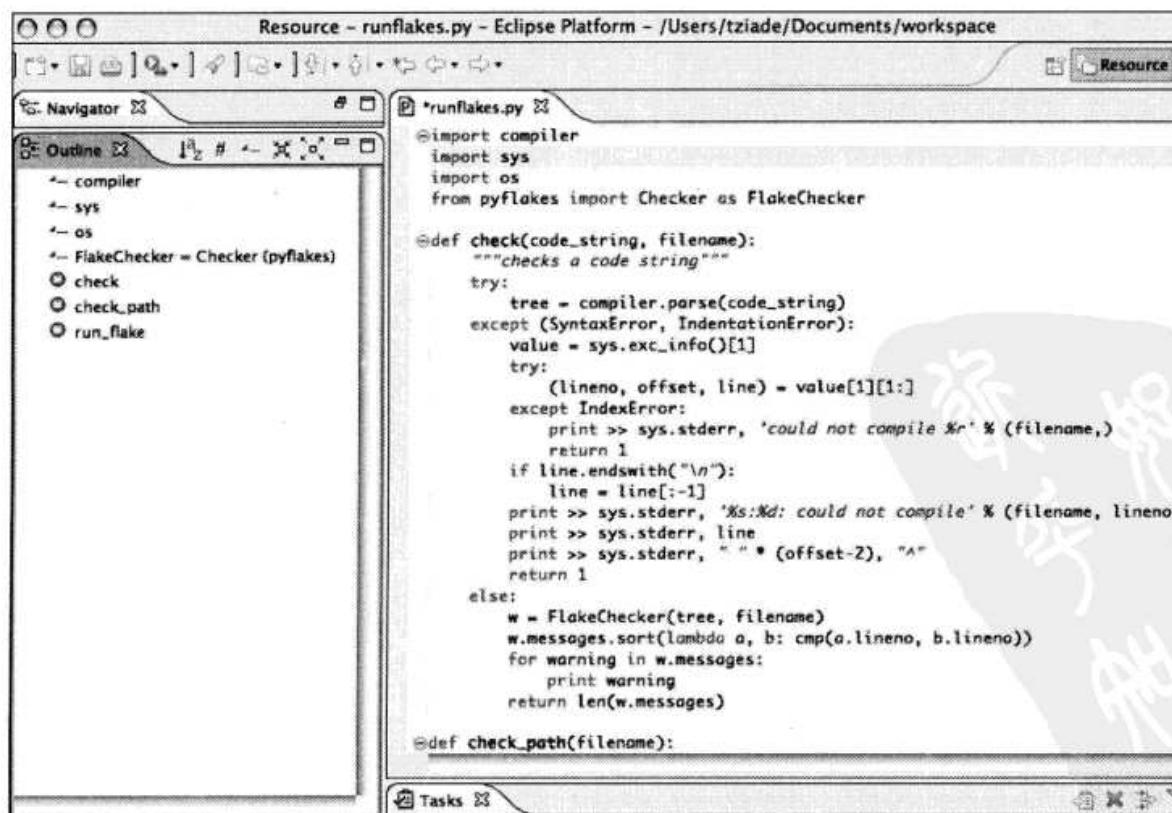


图 1.5

1.5 小结

本章介绍了 4 个方面的内容，如下所示。

- Python 的安装方法 Python 有多个版本，不过本书将主要关注于 CPython。它可以安装在 Linux、Mac OS X 和 Windows 平台上。可以通过编译源代码的方式来安装，但直接使用二进制安装文件会更简单些。
- setuptools 的安装方法 要完整安装 Python，还需要部署 setuptools。
- 定制命令行 Python 提供了一个交互式命令行，它可以通过启动脚本文件来定制。在编写程序时它将扮演十分重要的角色，因为通过它可以对小段代码进行测试。
- 工作环境 当完成命令行的定制之后，开发人员可以使用下列工具：
 - 传统的源代码编辑器，如 Vim、Emacs，也能为 Python 代码提供友好模式的其他编辑器软件。该编辑器需要借助一组工具集来补充。
 - 融合了开发中所需使用的各种功能的集成开发环境。Eclipse 和 PyDev 的组合现在是佳的。

下一章将讲述低于类级的语法最佳实践。

第 2 章

语法最佳实践——低于类级

回头看看所写的第一个程序，你或许就会赞同此观点：正确的语法应该是很容易阅读的代码，而不好的语法则会令人感到烦恼。

除了实现的算法、贯穿于程序的架构之外，应把重心放在如何解决问题上。许多程序员决定重新写一遍代码的原因就是语法丑陋、API 不清晰或者都没有统一的命名规范。

不过 Python 在最近几年有了很大的发展，你或许会对这些新特性感到惊讶。从最早版本到当前版本（现在是 2.6），已经有了很大改进，Python 语言变得更加清晰、整洁、易于编写。Python 基础没有很大改变，但和它相关的工具有了很多人性化的改变。

本章将介绍现代语法中最重要的元素，以及它们的使用技巧，包括：

- 列表推导（List comprehensions）；
- 迭代器（Iterators）和生成器（generators）；
- 描述符（Descriptors）和属性（properties）；
- 装饰（Decorators）；
- contextlib。



速度提升、内存使用等代码性能技巧将在第 12 章中讲述。

如果在阅读本章时需要查阅 Python 相关语法，可以参考官方文档的 3 个重要组成部分：

- 命令行中的帮助功能；
- 在线教程 (<http://docs.python.org/tut/tut.html>)；
- 样式指南 (<http://www.python.org/dev/peps/pep-0008>)。

2.1 列表推导

编写如下所示的代码是令人痛苦的。

```
>>> numbers = range(10)
>>> size = len(numbers)
>>> evens = []
>>> i = 0
>>> while i < size:
...     if i % 2 == 0:
...         evens.append(i)
...     i += 1
...
>>> evens
[0, 2, 4, 6, 8]
```

这对于 C 语言而言或许是可行的，但是在 Python 中它确实会使程序的执行速度变慢了，因为：

- 它使解释程序在每次循环中都要确定序系中的哪一个部分被修改；
- 它使得必须通过一个计数器来跟踪必须处理的元素。

List comprehensions 是这种场景下的正确选择，它使用编排好的特性对前述语法中的一部分进行了自动化处理，如下所示。

```
>>> [i for i in range(10) if i % 2 == 0]
[0, 2, 4, 6, 8]
```

这种编写方法除了高效之外，也更加简短，涉及的元素也更少。在更大的程序中，这意味着引入的缺陷更少，代码更容易阅读和理解。

Python 风格的语法的另一个典型例子是使用 `enumerate`。这个内建函数为在循环中使用序列时提供了更加便利的获得索引的方式，例如下面这个代码块。

```
>>> i = 0
>>> seq = ["one", "two", "three"]
>>> for element in seq:
...     seq[i] = '%d: %s' % (i, seq[i])
...     i += 1
...
```

```
>>> seq
['0: one', '1: two', '2: three']
```

它可以用以下简短的代码块代替。

```
>>> seq = ["one", "two", "three"]
>>> for i, element in enumerate(seq):
...     seq[i] = '%d: %s' % (i, seq[i])
...
>>> seq
['0: one', '1: two', '2: three']
```

然后，还可以用一个 List comprehensions 将其重构如下。

```
>>> def _treatment(pos, element):
...     return '%d: %s' % (pos, element)
...
>>> seq = ["one", "two", "three"]
>>> [_treatment(i, el) for i, el in enumerate(seq)]
['0: one', '1: two', '2: three']
```

最后，这个版本的代码更容易矢量化，因为它共享了基于序列中单个项目的小函数。

Python 风格的语法意味着什么？

 Python 风格的语法是一种对小代码模式最有效的语法。这个词也适用于诸如程序库这样的高级别事物上。在那种情况下，如果程序库能够很好地使用 Python 的风格，它就被认为是 Python 化（Phthonic）的。在开发社群中，这个术语有时被用来对代码块进行分类。

 每当要对序列中的内容进行循环处理时，就应该尝试用 List comprehensions 来代替它。

2.2 迭代器和生成器

迭代器只不过是一个实现迭代器协议的容器对象。它基于两个方法：

- `next` 返回容器的下一个项目；
- `__iter__` 返回迭代器本身。

迭代器可以通过使用一个 `iter` 内建函数和一个序列来创建，示例如下。

```
>>> i = iter('abc')
>>> i.next()
'a'
>>> i.next()
'b'
>>> i.next()
'c'
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

当序列遍历完时，将抛出一个 `StopIteration` 异常。这将使迭代器与循环兼容，因为它们将捕获这个异常以停止循环。要创建定制的迭代器，可以编写一个具有 `next` 方法的类，只要该类能够提供返回迭代器实例的 `__iter__` 特殊方法。

```
>>> class MyIterator(object):
...     def __init__(self, step):
...         self.step = step
...     def next(self):
...         """Returns the next element."""
...         if self.step == 0:
...             raise StopIteration
...         self.step -= 1
...         return self.step
...     def __iter__(self):
...         """Returns the iterator itself."""
...         return self
...
>>> for el in MyIterator(4):
...     print el
...
3
2
1
0
```

迭代器本身是一个底层的特性和概念，在程序中可以没有它们。但是它们为生成器这一更有趣的特性提供了基础。

2.2.1 生成器

从 Python 2.2 起，生成器提供了一种出色的方法，使得需要返回一系列元素的函数所需的代码更加简单、高效。基于 `yield` 指令，可以暂停一个函数并返回中间结果。该函数将保存执行环境并且可以在必要时恢复。

例如（这是 PEP 中关于迭代器的实例），Fibonacci 数列可以用一个迭代器来实现，如下所示。

```
>>> def fibonacci():
...     a, b = 0, 1
...     while True:
...         yield b
...         a, b = b, a + b
...
>>> fib = fibonacci()
>>> fib.next()
1
>>> fib.next()
1
>>> fib.next()
2
>>> [fib.next() for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

该函数将返回一个特殊的迭代器，也就是 `generator` 对象，它知道如何保存执行环境。对它的调用是不确定的，每次都将产生序列中的下一个元素。这种语法很简洁，算法的不确定特性并没有影响代码的可读性。不必提供使函数可停止的方法。实际上，这看上去像是用伪代码设计的序列一样。

PEP 的含义是 Python 增强建议 (Python Enhancement Proposal)。它是在 Python 上进行修改的文件，也是开发社团讨论的一个出发点。

进一步的信息参见<http://www.python.org/dev/peps/pep-0001>。

在开发社团中，生成器并不那么常用，因为开发人员还不习惯如此思考。开发人员多年来习惯于使用意图明确的函数。当需要一个将返回一个序列或在循环中执行的函数时，就应该考虑生成器。当这些元素将被传递到另一个函数中以进行后续处理时，一次返回一个元素能够提高整体性能。

在这种情况下，用于处理一个元素的资源通常不如用于整个过程的资源重要。因此，它们可以仍然保持位于底层，使程序更加高效。例如，Fibonacci 数列是无穷尽的，但是用来生成它的生成器不需要在提供一个值的时候，就预先占用无穷多的内存。常见的应用场景是使用生成器的流数据缓冲区。使用这些数据的第三方程序代码可以暂停、恢复和停止生成器，所有数据在开始这一过程之前不需要导入。

例如，来自标准程序库的 `tokenize` 模块将在文本之外生成令牌，并且针对每个处理过的行返回一个迭代器，这可以被传递到一些处理中，如下所示。

```
>>> import tokenize
>>> reader = open('amina.py').next
>>> tokens = tokenize.generate_tokens(reader)
>>> tokens.next()
(1, 'from', (1, 0), (1, 4), 'from amina.quality import
                                         similarities\n')
>>> tokens.next()
(1, 'amina', (1, 5), (1, 10), 'from amina.quality import
                                         similarities\n')
>>> tokens.next()
```

在此我们看到，`open` 函数遍历了文件中的每个行，而 `generate_tokens` 则在一个管道中对其进行遍历，完成一些额外的工作。

生成器对降低程序复杂性也有帮助，并且能够提升基于多个序列的数据转换算法的性能。把每个序列当作一个迭代器，然后将它们合并到一个高级别的函数中，这是一种避免函数变得庞大、丑陋、不可理解的好办法。而且，这可以给整个处理链提供实时的反馈。

在下面的示例中，每个函数用来在序列上定义一个转换。然后它们被链接起来应用。每次调用将处理一个元素并返回其结果，如下所示。

```
>>> def power(values):
...     for value in values:
...         print 'powering %s' % value
...         yield value
...
>>> def adder(values):
```

```

...     for value in values:
...         print 'adding to %s' % value
...         if value % 2 == 0:
...             yield value + 3
...         else:
...             yield value + 2
...
>>> elements = [1, 4, 7, 9, 12, 19]
>>> res = adder(power(elements))
>>> res.next()
powering 1
adding to 1
3
>>> res.next()
powering 4
adding to 4
7
>>> res.next()
powering 7
adding to 7
9

```



保持代码简单，而不是数据

拥有许多简单的处理序列值的可迭代函数，要比一个复杂的、每次计算一个值的函数更好一些。

Python引入的与生成器相关的最后一个特性是提供了与 `next` 方法调用的代码进行交互的功能。`yield` 将变成一个表达式，而一个值可以通过名为 `send` 的新方法来传递，如下所示。

```

>>> def psychologist():
...     print 'Please tell me your problems'
...     while True:
...         answer = (yield)
...         if answer is not None:
...             if answer.endswith('?'):
...                 print ("Don't ask yourself

```

```

...
    "too much questions")
...
elif 'good' in answer:
    print "A that's good, go on"
...
elif 'bad' in answer:
    print "Don't be so negative"
...
>>> free = psychologist()
>>> free.next()
Please tell me your problems
>>> free.send('I feel bad')
Don't be so negative
>>> free.send("Why I shouldn't ?")
Don't ask yourself too much questions
>>> free.send("ok then i should find what is good for me")
A that's good, go on

```

`send` 的工作机制与 `next` 一样，但是 `yield` 将变成能够返回传入的值。因而，这个函数可以根据客户端代码来改变其行为。同时，还添加了 `throw` 和 `close` 两个函数，以完成该行为。它们将向生成器抛出一个错误：

- `throw` 允许客户端代码传入要抛出的任何类型的异常；
- `close` 的工作方式是相同的，但是将会抛出一个特定的异常——`GeneratorExit`，在这种情况下，生成器函数必须再次抛出 `GeneratorExit` 或 `StopIteration` 异常。

因此，一个典型的生成器模板应该类似于如下所示。

```

>>> def my_generator():
...     try:
...         yield 'something'
...     except ValueError:
...         yield 'dealing with the exception'
...     finally:
...         print "ok let's clean"
...
>>> gen = my_generator()
>>> gen.next()
'something'
>>> gen.throw(ValueError('mean mean mean'))
'dealing with the exception'
>>> gen.close()

```

```

ok let's clean
>>> gen.next()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration

```

finally 部分在之前的版本中是不允许使用的，它将捕获任何未被捕获的 close 和 throw 调用，是完成清理工作的推荐方式。GeneratorExit 异常在生成器中是无法捕获的，因为它被编译器用来确定调用 close 时是否正常退出。如果有代码与这个异常关联，那么解释程序将抛出一个系统错误并退出。

有了这 3 个新的方法，就有可能使用生成器来编写协同程序（coroutine）。

2.2.2 协同程序

协同程序是可以挂起、恢复，并且有多个进入点的函数。有些语言本身就提供了这种特性，如 Io (<http://iolanguage.com>) 和 Lua (<http://www.lua.org>)，它们可以实现协同的多任务和管道机制。例如，每个协同程序将消费或生成数据，然后暂停，直到其他数据被传递。

在 Python 中，协同程序的替代者是线程，它可以实现代码块之间的交互。但是因为它们表现出一种抢先式的风格，所以必须注意资源锁，而协同程序不需要。这样的代码可能变得相当复杂，难以创建和调试。但是生成器几乎就是协同程序，添加 send、throw 和 close，其初始的意图就是为该语言提供一种类似协同程序的特性。

PEP 342 (<http://www.python.org/dev/peps/pep-0342>) 实例化了生成器的新行为，也提供了创建协同程序的调度程序的完整实例。这个模式被称为 Trampoline，可以被看作生成和消费数据的协同程序之间的媒介。它使用一个队列将协同程序连接在一起。

在 PyPI 中的 multitask 模块（用 easy_install multitask 安装）实现了这一模式，使用也十分简单，如下所示。

```

>>> import multitask
>>> import time
>>> def coroutine_1():
...     for i in range(3):
...         print 'c1'
...         yield i
...
>>> def coroutine_2():
...     for i in range(3):
...
```

```

...
        print 'c2'
...
        yield i
...
>>> multitask.add(coroutine_1())
>>> multitask.add(coroutine_2())
>>> multitask.run()
c1
c2
c1
c2
c1
c2

```

在协同程序之间的协作，最经典的例子是接受来自多个客户的查询，并将每个查询委托给对此做出响应的新线程的服务器应用程序。要使用协同程序来实现这一模式，首先要编写一个负责接收查询的协同程序（服务器），以及另一个处理它们的协同程序（句柄）。第一个协同程序在 `trampoline` 中为每个请求放置一个新的句柄。

`multitask` 包为套接字处理（如 `echo` 服务器）提供了很好的 API，通过它实现程序很简单，如下所示。

```

from __future__ import with_statement
from contextlib import closing
import socket
import multitask
def client_handler(sock):
    with closing(sock):
        while True:
            data = (yield multitask.recv(sock, 1024))
            if not data:
                break
            yield multitask.send(sock, data)
def echo_server(hostname, port):
    addrinfo = socket.getaddrinfo(hostname, port,
                                  socket.AF_UNSPEC,
                                  socket.SOCK_STREAM)
    (family, socktype, proto,
     canonname, sockaddr) = addrinfo[0]
    with closing(socket.socket(family,

```

```

                socktype,
                proto)) as sock:
    sock.setsockopt(socket.SOL_SOCKET,
                    socket.SO_REUSEADDR, 1)
    sock.bind(sockaddr)
    sock.listen(5)
    while True:
        multitask.add(client_handler((
            yield multitask.accept(sock))[0]))
if __name__ == '__main__':
    import sys
    hostname = None
    port = 1111
    if len(sys.argv) > 1:
        hostname = sys.argv[1]
    if len(sys.argv) > 2:
        port = int(sys.argv[2])
    multitask.add(echo_server(hostname, port))
    try:
        multitask.run()
    except KeyboardInterrupt:
        pass

```



本章稍后会介绍 contextlib。



另一种协同程序实现

greenlet 是另一种程序库，它的特性之一就是为 Python 协同程序提供了一个良好的实现。

2.2.3 生成器表达式

Python 为编写针对序列的简单生成器提供了一种快捷方式。可以用一种类似列表推导的语法来代替 yield。在此，使用圆括号代替中括号，如下所示。

```
>>> iter = (x**2 for x in range(10) if x % 2 == 0)
>>> for el in iter:
...     print el
...
0
4
16
36
64
```

这种表达式常被称为生成器表达式或 genexp。它们使用类似列表推导的方式减少了序列代码的总量。它们和常规的生成器一样，每次输出一个元素。所以整个序列和列表推导一样，都不会事先进行计算。每当在 `yield` 表达式上创建简单的循环时，都应该使用它，或者用它来代替表现类似迭代器的列表推导。

2.2.4 itertools 模块

当 Python 中添加了迭代器后，就为实现常见模式提供了一个新的模块。因为它是以 C 语言编写，所以提供了最高效的迭代器，`itertools` 覆盖了许多模式，但最有趣的是 `islice`、`tee` 和 `groupby`。

1. `islice`: 窗口迭代器

`islice` 将返回一个运行在序列的子分组之上的迭代器。以下实例将按行从标准输入中读取信息，并且输出从第 5 行开始的每行元素，只要该行的元素超过 4 个，如下所示。

```
>>> import itertools
>>> def starting_at_five():
...     value = raw_input().strip()
...     while value != '':
...         for el in itertools.islice(value.split(),
...                                     4, None):
...             yield el
...         value = raw_input().strip()
...
>>> iter = starting_at_five()
>>> iter.next()
one two three four five six
```

```
'five'
>>> iter.next()
'six'
>>> iter.next()
one two
one two three four five six
'five'
>>> iter.next()
'six'
>>> iter.next()
one
one two three four five six seven eight
'five'
>>> iter.next()
'six'
>>> iter.next()
'seven'
>>> iter.next()
'eight'
```

当需要抽取位于流中特定位置的数据时，都可以使用 `islice`。例如，可能是使用记录的特殊格式文件，或者表现元数据（如 SOAP 封套）封装的数据流。在那种情况下，`islice` 可以看作在每行数据之上的一个滑动窗口。

2. `tee`: 往返式的迭代器

迭代器将消费其处理的序列，但它不会往回处理。`tee` 提供了在一个序列之上运行多个迭代器的模式。如果提供第一次运行的信息，就能帮助我们再次基于这些数据运行。例如，读取文件的表头可以在运行一个处理之前提供特性信息，如下所示。

```
>>> import itertools
>>> def with_head(iterable, headsize=1):
...     a, b = itertools.tee(iterable)
...     return list(itertools.islice(a, headsize)), b
...
>>> with_head([1, 2, 3, 4])
([1], <itertools.tee object at 0x100c698>)
>>> with_head([1, 2, 3, 4], 4)
([1, 2, 3, 4], <itertools.tee object at 0x100c670>)
```

在这个函数中，如果用 `tee` 生成两个迭代器，那么第一个迭代器由 `islice` 获取该迭代的第一个 `headsize` 元素，并将它们作为一个普通列表返回；第二个迭代器返回的元素则是一个新的迭代器，可以工作在整个序列之上。

3. groupby: uniq 迭代器

这个函数有点像 Unix 命令 `uniq`。它可以对来自一个迭代器的重复元素进行分组，只要它们是相邻的，还可以提供另一个函数来执行元素的比较。否则，将采用标识符比较。

`groupby` 的一个应用实例是使用行程长度编码（RLE）来压缩数据。字符串中的每组相邻的重复字符将替换成该字符本身和重复次数。如果字符没有重复，则使用 1。

例如：

```
get uuuuuuuuuuuuuuuup
```

将被替代为：

```
1g1e1t1 8ulp
```

使用 `groupby` 来获取 RLE，只需要几行代码，如下所示。

```
>>> from itertools import groupby
>>> def compress(data):
...     return ((len(list(group)), name)
...             for name, group in groupby(data))
...
>>> def decompress(data):
...     return (car * size for size, car in data)
...
>>> list(compress('get uuuuuuuuuuuuuuuup'))
[(1, 'g'), (1, 'e'), (1, 't'), (1, ''),
 (18, 'u'), (1, 'p')]
>>> compressed = compress('get uuuuuuuuuuuuuuuup')
>>> ''.join(decompress(compressed))
'get uuuuuuuuuuuuuuuup'
```

压缩算法

如果对压缩感兴趣，可以考虑 LZ77 算法。它是 RLE 的增强版本，将查找相邻的相同模式，而不是相同的字符 (<http://en.wikipedia.org/wiki/LZ77>)。

每当需要在数据上完成一个摘要的时候，都可以使用 `groupby`。这时候，内建的 `sorted` 函数就非常有用，可以使传入的数据中相似的元素相邻。

4. 其他函数

在 <http://docs.python.org/lib/itertools-functions.html> 中，可以看到 `itertools` 函数的完整列表，包括本节中未说明的函数。每个函数都附有以纯粹的 Python 编写的，理解其工作方式的对应代码。

- `chain(*iterables)` 创建一个在第一个可迭代的对象上迭代的迭代器，然后继续下一个，以此类推。
- `count([n])` 返回一个给出连续整数的迭代器，例如一个范围。当未给出 `n` 时，将从 0 开始。
- `cycle(iterable)` 在可迭代对象的每个元素之上迭代，然后重新开始，无限次地重复。
- `dropwhile(predicate, iterable)` 只要断言 (`predicate`) 为真，就从可迭代对象中删除每个元素。当断言为假时则输出剩余的元素。
- `ifilter(predicate, iterable)` 近似于内建函数 `filter`。
- `ifilterfalse(predicate, iterable)` 与 `ifilter` 类似，但是将在断言为假时执行迭代。
- `imap(function, *iterables)` 与内建函数 `map` 类似，不过它将在多个可迭代对象上工作，在最短的可迭代对象耗尽时将停止。
- `izip(*iterables)` 和 `zip` 类似，不过它将返回一个迭代器。
- `repeat(object[, times])` 返回一个迭代器，该迭代器在每次调用时返回 `object`。运行 `times` 次，如果未指定 `times` 则运行无限次。
- `starmap(function, iterable)` 和 `imap` 类似，不过它将把可迭代元素作为星号参数向 `function` 传递。这在返回元素是元组 (`tuples`) 时有帮助，它可以作为参数传递给 `function`。
- `takewhile(predicate, iterable)` 从可迭代对象返回元素，当 `predicate` 返回假时停止。

2.3 装 饰 器

装饰器是在 Python 2.4 中新加入的，它使得函数和方法封装（接收一个函数并返回增强版本的一个函数）更容易阅读和理解。原始的使用场景是可以将方法在定义的首部将其定义为类方法或静态方法。在添加装饰器之前，相应的语法如下。

```
>>> class WhatFor(object):
...     def it(cls):
...         print 'work with %s' % cls
```

```
...     it = classmethod(it)
...
...     def uncommon():
...         print 'I could be a global function'
...
...     uncommon = staticmethod(uncommon)
...
...
```

如果方法很大，或者在该方法中有多个转换时，这种语法将变得难以阅读。而使用了装饰器之后，其语法更加浅显易懂，如下所示。

```
>>> class WhatFor(object):
...     @classmethod
...     def it(cls):
...         print 'work with %s' % cls
...     @staticmethod
...     def uncommon():
...         print 'I could be a global function'
>>> this_is = WhatFor()
>>> this_is.it()
work with <class '__main__.WhatFor'>
>>> this_is.uncommon()
I could be a global function
```

在装饰器出现之后，社团中的许多开发人员开始使用它们，因为它是实现某些模式的显而易见的方法。关于这个概念最早的邮件主题是由 IronPython 的开发者 Jim Hugunin 提出的，如图 2.1 所示。

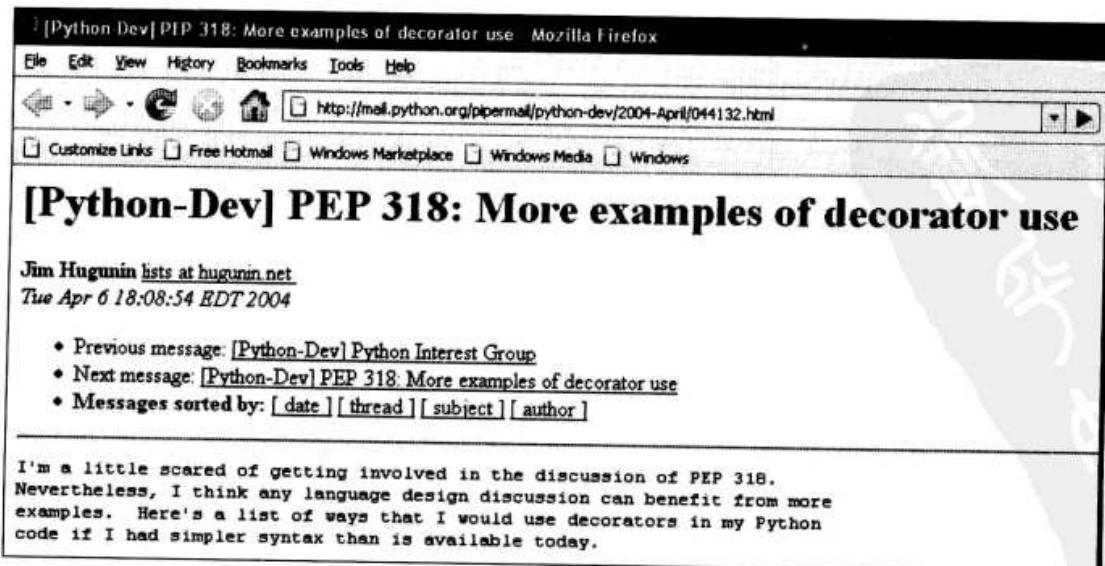


图 2.1

本节剩下的部分将介绍如何编写装饰器，并提供一些实例。

2.3.1 如何编写装饰器

编写自定义装饰器有许多方法，但最简单和最容易理解的方法是编写一个函数，返回封装原始函数调用的一个子函数。

通用的模式如下。

```
>>> def mydecorator(function):
...     def _mydecorator(*args, **kw):
...         # 在调用实际函数之前做些填充工作
...         res = function(*args, **kw)
...         # 做完某些填充工作之后
...         return res
...     # 返回子函数
...     return _mydecorator
...
```

为子函数应用一个诸如`_mydecorator`之类的明确的名称，而不是像`wrapper`这样的通用名称是一个好习惯，因为明确的名称更方便在错误发生的时候回溯——可以知道正在处理指定的装饰器。

当装饰器需要参数时，必须使用第二级封装。

```
def mydecorator(arg1, arg2):
    def _mydecorator(function):
        def __mydecorator(*args, **kw):
            # 在调用实际函数之前做些填充工作
            res = function(*args, **kw)
            # 做完某些填充工作之后
            return res
        # 返回子函数
        return __mydecorator
    return _mydecorator
```

因为装饰器在模块第一次被读取时由解释程序装入，所以它们的使用必须受限于总体上可以应用的封装器。如果装饰器与方法的类或所增强的函数签名绑定，它应该被重构为常规的可调用对象，从而避免复杂性。在任何情况下，当装饰器处理 API 时，一个好的方法是将它们聚集在一个易于维护的模块中。



装饰器应该关注于所封装的函数或方法接收和返回的参数。如果需要，应该尽可能限制其内省（introspection）工作。

常见的装饰器模式包括：

- 参数检查；
- 缓存；
- 代理；
- 上下文提供者。

2.3.2 参数检查

检查函数接收或返回的参数，在特定上下文执行时可能有用。例如，如果一个函数通过 XML-RPC 调用，Python 将不能和静态类型语言中一样直接提供它的完整签名。当 XML-RPC 客户要求函数签名时，就需要这个功能来提供内省能力。



XML-RPC 协议

XML-RPC 协议是一种轻量级的远程过程调用协议，它通过 HTTP 上的 XML 来对调用进行编码。它通常用于在简单客户-服务器交换中代替 SOAP。

和 SOAP 不同（SOAP 能够列出所有可调用函数的页面，即 WSDL），XML-RPC 没有可用函数的目录。

现在该协议已提出了一个扩展，可以用来发现服务器 API，Python 的 xmlrpclib 模块实现这个扩展（参见 <http://docs.python.org/lib/serverproxy-objects.html>）。

装饰器能提供这种签名类型，并确保输入输出与此有关，如下所示。

```
>>> from itertools import izip
>>> rpc_info = {}
>>> def xmlrpc(in_=(), out=(type(None),)):
...     def _xmlrpc(function):
...         # 注册签名
...         func_name = function.func_name
```

```

...     rpc_info[func_name] = (in_, out)
...
...     def _check_types(elements, types):
...         """Subfunction that checks the types."""
...         if len(elements) != len(types):
...             raise TypeError('argument count is wrong')
...         typed = enumerate(izip(elements, types))
...         for index, couple in typed:
...             arg, of_the_right_type = couple
...             if isinstance(arg, of_the_right_type):
...                 continue
...             raise TypeError('arg # %d should be %s' % \
...                             (index, of_the_right_type))
...
...     # 封装函数
...     def __xmlrpc(*args):      # 没有允许的关键字
...         # 检查输入的内容
...         checkable_args = args[1:] # removing self
...         _check_types(checkable_args, in_)
...         # 执行该函数
...         res = function(*args)
...
...         # 检查输出的内容
...         if not type(res) in (tuple, list):
...             checkable_res = (res,)
...         else:
...             checkable_res = res
...         _check_types(checkable_res, out)
...
...         # 函数及其类型检查成功
...         return res
...     return __xmlrpc
... return __xmlrpc
...

```

装饰器将该函数注册到全局字典中，并为其参数和返回值保存一个类型列表。注意，该函数做了很大的简化，以示范参数检查装饰器。

以下就是一个使用实例。

```
>>> class RPCView(object):
...
...
...     @xmlrpc((int, int)) # two int -> None
...     def meth1(self, int1, int2):
...         print 'received %d and %d' % (int1, int2)
...
...
...     @xmlrpc((str,), (int,)) # string -> int
...     def meth2(self, phrase):
...         print 'received %s' % phrase
...         return 12
...
...
```

当读取类定义时，将填写 `rpc_infos` 词典，并且可以将其用于一个特定的环境。此时，将检查参数类型，如下所示。

```
>>> rpc_infos
{'meth2': ((<type 'str'>,), (<type 'int'>,)),
'meth1': ((<type 'int'>, <type 'int'>),
(<type 'NoneType'>,))}
>>> my = RPCView()
>>> my.meth1(1, 2)
received 1 and 2
>>> my.meth2(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 16, in _wrapper
  File "<stdin>", line 11, in _check_types
TypeError: arg #0 should be <type 'str'>
```

参数检查装饰器有许多其他使用场景，如类型强制（参见 <http://wiki.python.org/moin/PythonDecoratorLibrary#head-308f2b3507ca91800def19d813348f78db34303e>），可以根据给定的全局配置值来定义多种级别的类型检查：

- 什么也不检查；
- 检查器仅弹出警告；
- 检查器将抛出 `TypeError` 异常。

2.3.3 缓存

缓存装饰器与参数检查很相似，不过它关注于内部状态而不影响输出的函数。每组参数

可以链接到一个唯一结果上。这种编程风格是函数型编程的特性（参见 http://en.wikipedia.org/wiki/Functional_programming），当输入值有限时可以使用。

因此，缓存装饰器可以将输出与计算它所需的参数放在一起，并且直接在后续的调用中返回它。这种行为被称为 Memoizing（参见<http://en.wikipedia.org/wiki/Memoizing>，常译为自动缓存），很容易被实现为一个装饰器，如下所示。

```
>>> import time
>>> import hashlib
>>> import pickle
>>> from itertools import chain
>>> cache = {}
>>> def is_obsolete(entry, duration):
...     return time.time() - entry['time'] > duration
...
>>> def compute_key(function, args, kw):
...     key = pickle.dumps((function.func_name, args, kw))
...     return hashlib.sha1(key).hexdigest()
...
>>> def memoize(duration=10):
...     def _memoize(function):
...         def __memoize(*args, **kw):
...             key = compute_key(function, args, kw)
...
...             # 是否已经拥有它了？
...             if (key in cache and
...                 not is_obsolete(cache[key], duration)):
...                 print 'we got a winner'
...                 return cache[key]['value']
...
...             # 计算
...             result = function(*args, **kw)
...
...             # 保存结果
...             cache[key] = {'value': result,
...                         'time': time.time()}
...             return result
...     return _memoize
```

```

... return _memoize
...

```

SHA hash 键值使用已排序的参数值建立，该结果将保存在一个全局字典中。hash 使用一个 pickle 来建立，这是冻结所有作为参数传递的对象状态，以确保所有参数均为良好候选者的一个快捷方式。例如，如果一个线程或套接字被用作一个参数，将抛出一个 PicklingError。

duration 参数用于在上次函数调用之后，使存在太久的缓存值失效。

以下是一个使用实例。

```

>>> @memoize()
... def very_very_very_complex_stuff(a, b):
...     # 如果执行该计算会让计算机过热，请考虑停止它
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> @memoize(1) # 1秒之后令缓存失效
... def very_very_very_complex_stuff(a, b):
...     return a + b
...
>>> very_very_very_complex_stuff(2, 2)
4
>>> very_very_very_complex_stuff(2, 2)
we got a winner
4
>>> cache
{'c2727f43c6e39b3694649ee0883234cf': {'value': 4, 'time':
1199734132.7102251}}
>>> time.sleep(2)
>>> very_very_very_complex_stuff(2, 2)
4

```

注意，由于第一个调用是两级封装，所以使用了空的括号。

应用了缓存的函数可以显著地提高程序的总体性能，但是必须小心使用。缓存值可能绑定到函数本身上，以管理其范围和生命周期，代替集中化的字典。但是在任何情况下，更有

效的装饰器应该使用基于高级缓存算法的专门化缓存库，并将其用于具备分布式缓存功能的 Web 应用程序上。Memcached 是 Python 中可用的算法之一。



第 13 章中将更详细地介绍与缓存技术相关信息。



2.3.4 代理

代理装饰器使用一个全局机制来标记和注册函数。例如，一个根据当前用户保护代码访问的安全层可以使用一个集中检查器和相关的可调用对象要求的权限来实现，如下所示。

```
>>> class User(object):
...     def __init__(self, roles):
...         self.roles = roles
...
...
>>> class Unauthorized(Exception):
...     pass
...
...
>>> def protect(role):
...     def __protect(function):
...         def __protect(*args, **kw):
...             user = globals().get('user')
...             if user is None or role not in user.roles:
...                 raise Unauthorized("I won't tell you")
...             return function(*args, **kw)
...         return __protect
...     return __protect
```

这种模式常常应用于 Python Web 框架，用来定义针对可发布类的安全性。例如，Django 提供装饰器来保障函数访问的安全。

以下是一个实例，当前用户被保存在一个全局变量中。装饰器在方法被访问时检查其角色，如下所示。

```
>>> tarek = User(('admin', 'user'))  
>>> bill = User(('user',))  
>>> class MySecrets(object):
```

```

...     @protect('admin')
...     def waffle_recipe(self):
...         print 'use tons of butter!'
...
...
>>> these_are = MySecrets()
>>> user = tarek
>>> these_are.waffle_recipe()
use tons of butter!
>>> user = bill
>>> these_are.waffle_recipe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 7, in wrap
      __main__.Unauthorized: I won't tell you

```

2.3.5 上下文提供者

上下文装饰器用来确保函数可以运行在正确的上下文中，或者在函数前后执行一些代码。换句话说，它用来设置或复位特定的执行环境。例如，当一个数据项必须与其他线程共享时，就需要使用一个锁来确保它在多重访问时得到保护。这个锁可以在一个装饰器中编写，示例如下。

```

>>> from threading import RLock
>>> lock = RLock()
>>> def synchronized(function):
...     def _synchronized(*args, **kw):
...         lock.acquire()
...         try:
...             return function(*args, **kw)
...         finally:
...             lock.release()
...     return _synchronized
>>> @locker
...     def thread_safe(): # make sure it locks the resource
...         pass
...

```

上下文装饰器可以使用 Python 2.5 中新添加的 with 语句来代替。创造这条语句的目的是使 try..finally 模式更加流畅，在某些情况下，它覆盖了上下文装饰器的使用场景。

要想了解更多装饰器的使用场景，可浏览 <http://wiki.python.org/moin/PythonDecoratorLibrary>。

2.4 with 和 contextlib

对于要确保即使发生一个错误时也能运行一些清理代码而言，try…finally 语句是很有用的。对此有许多使用场景，例如：

- 关闭一个文件；
- 释放一个锁；
- 创建一个临时的代码补丁；
- 在特殊环境中运行受保护的代码。

with 语句覆盖了这些使用场景，为在一个代码块前后调用一些代码提供了一种简单的方法。例如，使用一个文件通常可以如下实现。

```
>>> hosts = file('/etc/hosts')
>>> try:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print line
... finally:
...     hosts.close()
...
127.0.0.1      localhost
255.255.255.255 broadcasthost
::1            localhost
```

 本示例仅针对 Linux，因为它将在 etc 文件夹中读取主机文件，不过任何文件都可以以相同方式被使用。

通过使用 with 语句，以上代码可以重写如下。

```
>>> from __future__ import with_statement
>>> with file('/etc/hosts') as hosts:
...     for line in hosts:
...         if line.startswith('#'):
...             continue
...         print host
...
127.0.0.1      localhost
255.255.255.255 broadcasthost
::1            localhost
```

注意，在2.5系列版本中with语句仍然位于`__future__`模块里，而在2.6系列版本中则可以直可用。它的相关描述在<http://www.python.org/dev/peps/pep-0343>中可以找到。

与这条语句兼容的其他项目是来自`thread`和`threading`模块的类：

- `thread.LockType`
- `threading.Lock`
- `threading.RLock`
- `threading.Condition`
- `threading.Semaphore`
- `threading.BoundedSemaphore`

这些类都实现了两个方法——`_enter_`和`_exit_`，这都来自于with协议。换句话说，任何类都可以实现为如下所示。

```
>>> class Context(object):
...     def __enter__(self):
...         print 'entering the zone'
...     def __exit__(self, exception_type, exception_value,
...                exception_traceback):
...         print 'leaving the zone'
...         if exception_type is None:
...             print 'with no error'
...         else:
...             print 'with an error (%s)' % exception_value
...
>>> with Context():
...     print 'i am the zone'
...
```

```

entering the zone
i am the zone
leaving the zone
with no error
>>> with Context():
...     print 'i am the buggy zone'
...     raise TypeError('i am the bug')
...
entering the zone
i am the buggy zone
leaving the zone
with an error (i am the bug)
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: i am the bug

```

`_exit_` 将获取代码块中发生错误时填入的 3 个参数。如果没有发生错误，那么这 3 个参数都将被设置为 `None`。当发生一个错误时，`_exit_` 不应该重新抛出这个错误，因为这是调用者的责任。但是，它可以通过返回 `True` 来避免抛出该异常。这用来实现一些特殊的使用场景，例如下一小节中将会看到的 `contextmanager` 装饰器。但是对于大部分使用场景而言，这个方法的正确行为是执行与 `finally` 类似的清理工作；不管代码块中发生什么，它都不返回任何东西。

2.4.1 contextlib 模块

为了给 `with` 语句提供一些辅助类，标准程序库中添加了一个模块。最有用的辅助类是 `contextmanager`，这是一个装饰器，它增强了包含以 `yield` 语句分开的 `_enter_` 和 `_exit_` 两部分的生成器。使用这个装饰器来编写前面的实例，可以写成如下所示。

```

>>> from contextlib import contextmanager
>>> from __future__ import with_statement
>>> @contextmanager
... def context():
...     print 'entering the zone'
...     try:
...         yield
...     except Exception, e:

```

```

...
    print 'with an error (%s)' % e
...
    # 在此需要重新抛出错误
...
    raise e
...
else:
    print 'with no error'
...

```

如果发生任何异常，该函数需要重新抛出这个异常，以便传递它。注意，`context` 在需要时可以有一些参数，只要它们在调用中提供这些参数即可。这个小的辅助类简化了常规的基于类的上下文 API，正如生成器使用基于类的迭代器 API 所做的一样。

这个模块提供两个其他辅助类：

- `closing(element)` 这是一个由 `contextmanager` 装饰的函数，它将输出一个元素，然后在退出时调用该元素的 `close` 方法。例如，这对于处理流的类而言就很有帮助。
- `nested(context1, context2, ...)` 这是一个合并上下文并使用它们创建嵌套的 `with` 调用的函数。

2.4.2 上下文实例

`with` 语句有一种有趣的用法，就是在进入上下文时记录可以被装饰的代码，然后在它结束时将其改回原来的样子。这避免对代码本身进行修改，例如，允许一个单元测试得到关于代码使用的一些反馈。

下面的例子中创建了一个上下文，以装备指定类的所有公共 API。

```

>>> import logging
>>> from __future__ import with_statement
>>> from contextlib import contextmanager
>>> @contextmanager
...
... def logged(klass, logger):
...     # 记录器
...     def __log(f):
...         def __log(*args, **kw):
...             logger(f, args, kw)
...             return f(*args, **kw)
...         return __log
...
...     # 装备该类
...     for attribute in dir(klass):

```

```

...
    if attribute.startswith('_'):
        continue
...
    element = getattr(klass, attribute)
    setattr(klass, '__logged_%s' % attribute, element)
    setattr(klass, attribute, _log(element))

...
# 正常工作
yield klass

...
# 移除日志
for attribute in dir(klass):
    if not attribute.startswith('__logged_'):
        continue
...
    element = getattr(klass, attribute)
    setattr(klass, attribute[len('__logged_'):],
            element)
    delattr(klass, attribute)
...

```

记录器函数之后可以被用于记录指定上下文中调用的 API。在下面的例子中，调用被添加到一个列表中以跟踪 API 的使用，然后用于执行一些断言。例如，如果以下 API 被调用超过一次，它可能意味着类的公共签名可以重构，以避免重复调用。

```

>>> class One(object):
...     def __private(self):
...         pass
...     def one(self, other):
...         self.two()
...         other.thing(self)
...         self.__private()
...     def two(self):
...         pass
...
>>> class Two(object):
...     def thing(self, other):
...         other.two()
...
>>> calls = []

```

```
>>> def called(meth, args, kw):
...     calls.append(meth.im_func.func_name)
...
>>> with logged(One, called):
...     one = One()
...     two = Two()
...     one.one(two)
...
>>> calls
['one', 'two', 'two']
```

2.5 小结

本章介绍了以下内容。

- 如果要对现有可迭代的对象做一些处理，然后生成新的列表，那么列表推导将是最便利的方法。
- 迭代器和生成器提供了一组高效的生成和处理序列的工具。
- 对于包装具有附加行为的现有函数和方法而言，装饰器提供了一种易于理解的方法。它带来了实现和使用都很简单的新的编码模式。
- with语句改善了try..finally模式。

下一章仍将介绍语法最佳实践，不过将专注于类。

第 3 章

语法最佳实践——类级

本章将聚焦于类的语法最佳实践。在此我们不打算涉及设计模式，因为将在第 14 章中讨论它们。本章将概述 Python 中用于操纵和改进类代码的高级语法。尽管 Python 对象的一些细微之处仍然在发展，但是 2.x 系列版本中的基本方法仍然介绍了一些用于充分理解类的工作方式的语言内部结构。这对于避免一些常见的对象模型缺陷和误用是相当重要的。

本章将讨论以下主题：

- 子类化（Subclassing）内建类型；
- 访问超类中的方法；
- 槽（slots）；
- 元编程。

3.1 子类化内建类型

在 Python 2.2 中，提出了类型（type）和类（class）的统一（相关的草案请参阅 <http://www.python.org/download/releases/2.2.3/descrintro>）——这使内建类型的子类化成为可能，并且添加了一个新的内建类型 object，用来作为所有内建类型的公共祖先。这对于 Python 中的 OOP 机制有着微妙但不重要的影响，使程序员可以对诸如 list、tuple 或 dict 这样的内建类型进行子类化。所以，每当需要实现一个表现与内建类型十分类似的类时，最佳的方法是对其进行子类化。

接下来将展示一个名为 `distinctdict` 的类的代码，它就使用了这个技术。这是 Python 中常规的 dict 类型的子类。这个新类的表现几乎和一个平常的 dict 一样，但是它不允许存在多个相同的键值。当有人试图添加一个与原键值相同的新输入项时，它将抛出一个 `ValueError` 异

常，并带有一个帮助信息，如下所示。

```
>>> class DistinctError(Exception):
...     pass
>>> class distinctdict(dict):
...     def __setitem__(self, key, value):
...         try:
...             value_index = self.values().index(value)
...             # 只要 dict 在两次调用之间没有发生改变
...             # keys() 和 values() 将返回相应的列表
...             # 否则，dict 类型无法保证序列的顺序
...             existing_key = self.keys()[value_index]
...             if existing_key != key:
...                 raise DistinctError(("This value already
...                         "exists for '%s'") % \
...                         str(self[existing_key]))
...         except ValueError:
...             pass
...
...     super(distinctdict, self).__setitem__(key, value)
...
>>> my = distinctdict()
>>> my['key'] = 'value'
>>> my['other_key'] = 'value'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in __setitem__
ValueError: This value already exists for 'value'
>>> my['other_key'] = 'value2'
>>> my
{'other_key': 'value2', 'key': 'value'}
```

如果看看现有的代码，可能会发现许多类部分实现了内建类型，它们就像子类型一样，更快而且更清晰。例如，list 类型就是用一个序列来管理类型内部工作时会使用到的序列，如下所示。

```
>>> class folder(list):
...     def __init__(self, name):
...         self.name = name
```

```

...     def dir(self):
...         print 'I am the %s folder.' % self.name
...         for element in self:
...             print element
...
>>> the = folder('secret')
>>> the
[]
>>> the.append('pics')
>>> the.append('videos')
>>> the.dir()
I am the secret folder:
pics
videos

```

从 Python 2.4 开始，`collections` 模块已经提供了一些可以用来实现有效的容器类的类型：

- 实现双端队列的 `deque` 类型；
- `defaultdict` 类型提供一个类似词典的对象，带有默认的未知键码值，这个类型与 Perl 或 Ruby 中的 hash 工作方式类似。



内建类型覆盖了大部分使用场景

若打算创建一个类似序列或映射的新类，应考虑其特性并观察已有的内建类型。大部分时候最终会使用它们。

3.2 访问超类中的方法

`super` 是一个内建类型，用来访问属于某个对象的超类中的特性（attribute）。



Python 官方文档将 `super` 作为内建函数列出。尽管它的使用与函数类似，但它实际上仍然是一个内建类型。

```

>>> super
<type 'super'>

```

如果已习惯于通过直接调用父类并将 `self` 作为第一个参数来访问类型的特性，它的用法可能会带来一点混乱。参考以下代码。

```
>>> class Mama(object): # 这是老的方法
...     def says(self):
...         print 'do your homework'
...
>>> class Sister(Mama):
...     def says(self):
...         Mama.says(self)
...         print 'and clean your bedroom'
...
>>> anita = Sister()
>>> anita.says()
do your homework
and clean your bedroom
```

特别看一下 `Mama.says(self)` 这一行，使用了刚刚描述的方法调用超类（也就是 `Mama` 类）中的 `says()` 方法，将 `self` 作为第一个参数传入。这意味着，属于 `Mama` 的 `says()` 方法将被调用。但是调用它的实例将返回 `self`，在这个例子中，这是一个 `Sister` 的实例。

而 `super` 的用法将如下所示。

```
>>> class Sister(Mama): # 这是新方法
...     def says(self):
...         super(Sister, self).says()
...         print 'and clean your bedroom'
...
```

这个使用场景很容易理解，但是当面对多继承模式时，`super` 将会变得难以使用。在解释这些问题，包括何时应该避免使用 `super` 之前，理解方法解析顺序（MRO）在 Python 中的工作方式是很重要的。

3.2.1 理解 Python 的方法解析顺序

Python 2.3 中添加了基于为 Dylan 构建的 MRO (<http://www.opendylan.org>)，即 C3 的一个新的 MRO。Michele Simionato 所编写的参考文档在 <http://www.python.org/download/releases/2.3/mro> 上可以找到，它描述了 C3 构建一个类的线性化（也称为优先级，即祖先的一个排序列表）的方法。这个列表被用于特性的查找。



C3 算法将在本小节稍后部分介绍。

MRO 的变化用于解决创建公用基本类型（object）所引入的问题。在变成 C3 线性化方法之前，如果一个类有两个祖先（参见图 3.1），MRO 的计算将很简单，如下所示。

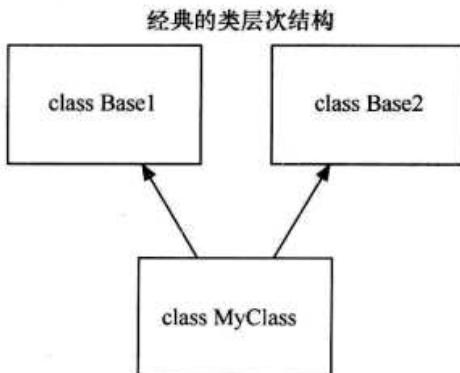


图 3.1

```

>>> class Base1:
...     pass
...
>>> class Base2:
...     def method(self):
...         print 'Base2'
...
>>> class MyClass(Base1, Base2):
...     pass
...
>>> here = MyClass()
>>> here.method()
Base2
  
```

当 here.method 被调用时，解释程序将查找 MyClass 中的方法，然后在 Base1 中查找，最后在 Base2 中查找。

现在，在两个基类之上引入一个 BaseBase 类（Base1 和 Base2 都从其继承，参见图 3.2）。结果是，根据“从左到右深度优先”规则的旧 MRO，在 Base2 中查找之前将通过 Base1 类回到顶部。

以下的代码将产生一种古怪的表现。

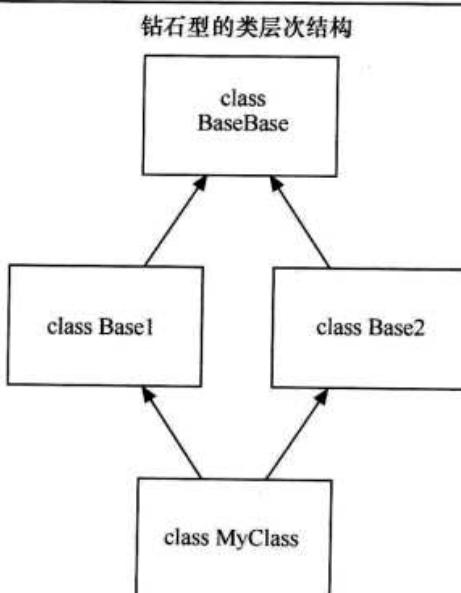


图 3.2

```

>>> class BaseBase:
...     def method(self):
...         print 'BaseBase'
...
...
>>> class Base1(BaseBase):
...     pass
...
...
>>> class Base2(BaseBase):
...     def method(self):
...         print 'Base2'
...
...
>>> class MyClass(Base1, Base2):
...     pass
...
...
>>> here = MyClass()
>>> here.method()
BaseBase
  
```

这种继承方案极其罕见，所以这更多的是一个理论问题而不是实践问题。标准程序库不会构建这样的继承层次结构，许多开发人员都认为这是一个坏方法。但是由于在类型层次顶部引入了 `object`，语言的 C 边（C side）突然出现了多重继承性问题，从而导致了在进行子类型化时的冲突。因为使用已有的 MRO 使其正常工作要花费太多的精力，所以提供一个新的 MRO 是更简单、快捷的解决方案。

所以，相同的实例在当前版本 Python（2.3 以上版本）中应该如下所示。

```
>>> class BaseBase(object):
...     def method(self):
...         print 'BaseBase'
...
...
>>> class Base1(BaseBase):
...     pass
...
...
>>> class Base2(BaseBase):
...     def method(self):
...         print 'Base2'
...
...>>> class MyClass(Base1, Base2):
...     pass
...
...
>>> here = MyClass()
>>> here.method()
Base2
```

新的 MRO 是基于一个基类之上的递归调用。为了概括本节最开始引用的 Michele Simionato 的文章，将 C3 符号应用到示例中，如下所示。

```
L[MyClass(Base1, Base2)] =
    MyClass + merge(L[Base1], L[Base2], Base1, Base2)
```

`L[MyClass]`是 `MyClass` 类的线性化，而 `merge` 是合并多个线性化结果的具体算法。

因此综合的描述应该是（正如 Simionato 所说）：

C 的线性化是 C 加上父类的线性化和父类列表的合并的总和。

`merge` 算法负责删除重复项并保持正确的顺序。其在文章中的描述为（适应于本例）：

取第一个列表的头，也就是 `L[Base1][0]`。如果这个头不在任何表的尾部，那么将它加到 `MyClass` 的线性化中，并且从合并中的列表里删除；否则查找下一个列表的头，如果是个好的表头则取用它。

然后重复该操作，直到所有类都被删除或者不能找到好的表头。在这个例子中，构建合并是不可能的，Python 2.3 将拒绝创建 `MyClass` 类并将抛出一个异常。

`head`（表头）是列表的第一个元素，而 `tail`（表尾）则包含其余元素。例如，在 `(Base1, Base2, ..., BaseN)` 中，`Base1` 是表头，`(Base2, ..., BaseN)` 则是表尾。

换句话说，C3 在每个父类上进行递归深度查找以获得列表的顺序。然后当一个类涉及多个列表时，计算一个从左到右的规则使用层次二义性消除来合并所有列表。

其结果如下。

```
>>> def L(klass):
...     return [k.__name__ for k in klass.__mro__]
...
>>> L(MyClass)
['MyClass', 'Base1', 'Base2', 'BaseBase', 'object']
```

类的 `_mro_` 特性（只读）用来存储线性化计算的结果，计算将在类定义载入时完成。



还可以调用 `MyClass.mro()` 来计算并获得结果。

注意，这将只对新风格的类起作用，所以在代码库中混杂新旧形式的类并不是好方法。MRO 的表现将会有差异。

3.2.2 super 的缺陷

现在回到 `super`。当使用了多重继承的层次结构时，再使用它将是相当危险的，这主要是因为类的初始化。在 Python 中，基类不会在 `__init__` 中被隐式地调用，所以依靠开发人员来调用它们。以下是几个例子。

1. 混用 `super` 和传统调用

在下面的取自 James Knight 网站 (<http://fuhm.net/super-harmful>) 的示例中，类 C 使用 `__init__` 方法调用其基类，这样将使类 B 被调用两次。

```
>>> class A(object):
...     def __init__(self):
...         print "A"
...         super(A, self).__init__()
...
>>> class B(object):
...     def __init__(self):
...         print "B"
...         super(B, self).__init__()
...
>>> class C(A,B):
...     def __init__(self):
```

```

...
    print "C"
...
A.__init__(self)
B.__init__(self)
...
>>> print "MRO:", [x.__name__ for x in C.__mro__]
MRO: ['C', 'A', 'B', 'object']
>>> C()
C A B B
<__main__.C object at 0xc4910>

```

发生这种情况的原因是，C 实例调用了 A.__init__(self)，因而 super(A, self).__init__() 将调用 B 的构造程序。换句话说，super 应该被用到整个类层次中。问题是有时候这种层次结构的一部分将位于第三方的代码中。许多由多重继承引入的层次调用相关的缺陷都可以在 James 的页面上找到。为了避免这些问题，应该总是在子类化之前看看 __mro__ 特性，如果它不存在，将要处理的就是一个旧式的类，避免使用 super 可能更安全一些，如下所示。

```

>>> from SimpleHTTPServer import SimpleHTTPRequestHandler
>>> SimpleHTTPRequestHandler.__mro__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: class SimpleHTTPRequestHandler has no attribute '__mro__'

```

如果 __mro__ 存在，则快速地看看每个 MRO 所涉及的类的构造程序代码。如果到处都使用了 super，那非常好，也可以使用它。否则，就请试着保持一致性。

在下面的例子中可以看到，collections.deque 能够安全地被子类化，可以使用 super，因为它直接子类化了 object，如下所示。

```

>>> from collections import deque
>>> deque.__mro__
(<type 'collections.deque'>, <type 'object'>)

```

在这个例子中，看上去 random.Random 是一个存在于 _random 模块中的另一个类的封装器，如下所示。

```

>>> from random import Random
>>> random.Random.__mro__
(<class 'random.Random'>, <type '_random.Random'>, <type 'object'>)

```

这是一个 C 模块，所以我们应该也是安全的。

最后一个例子是一个 Zope 类，构造程序应该被仔细地检查，如下所示。

```
>>> from zope.app.container.browser.adding import Adding
>>> Adding.__mro__
(<class 'zope.app.container.browser.adding.Adding'>,
<class 'zope.publisher.browser.BrowserView'>,
<class 'zope.location.location.Location'>,
<type 'object'>)
```

2. 不同种类的参数

`super` 用法的另一个问题是初始化中的参数传递。类在没有相同签名的情况下怎么调用其基类的`__init__`代码？这将引发以下问题。

```
>>> class BaseBase(object):
...     def __init__(self):
...         print 'basebase'
...         super(BaseBase, self).__init__()
...
>>> class Basel(BaseBase):
...     def __init__(self):
...         print 'basel'
...         super(Basel, self).__init__()
...
>>> class Base2(BaseBase):
...     def __init__(self, arg):
...         print 'base2'
...         super(Base2, self).__init__()
...
>>> class MyClass(Basel, Base2):
...     def __init__(self, arg):
...         print 'my base'
...         super(MyClass, self).__init__(arg)
...
>>> m = MyClass(10)
my base
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 4, in __init__
TypeError: __init__() takes exactly 1 argument (2 given)
```

一种解决方案是使用`*args`和`**kw`魔法。这样，所有构造程序将传递所有的参数，即使不使用它们，如下所示。

```
>>> class BaseBase(object):
...     def __init__(self, *args, **kw):
...         print 'basebase'
...         super(BaseBase, self).__init__(*args, **kw)
...
>>> class Basel(BaseBase):
...     def __init__(self, *args, **kw):
...         print 'basel'
...         super(Basel, self).__init__(*args, **kw)
...
>>> class Base2(BaseBase):
...     def __init__(self, arg, *args, **kw):
...         print 'base2'
...         super(Base2, self).__init__(*args, **kw)
...
>>> class MyClass(Basel, Base2):
...     def __init__(self, arg):
...         print 'my base'
...         super(MyClass, self).__init__(arg)
...
>>> m = MyClass(10)
my base
basel
base2
basebase
```

但是这是一个糟糕的修复方法，因为它使所有构造程序将接受任何类型的参数。这会导致代码变得很脆弱，因为任何参数都被传递并且通过。另一个解决方法是在`MyClass`中使用经典的`__init__`调用，但是这将会导致产生第一种缺陷。

3.3 最佳实践

为了避免出现前面提到的所有问题，在Python在这个领域上取得进展之前，必须考虑以

下几点。

- 应该避免多重继承 可以采用第14章中提出的一些设计模式来代替它。
- super的使用必须一致 在类层次结构中，应该在所有地方都使用super或者彻底不使用它。混用super和传统调用是一种混乱的方法，人们倾向于避免使用super，这样代码会更明晰。
- 不要混用老式和新式的类 两者都具备的代码库将导致不同的MRO表现。
- 调用父类时必须检查类层次 为了避免出现任何问题，每次调用父类时，必须快速地查看一下所涉及的MRO（使用__mro__）。

3.4 描述符和属性

当许多C++和Java程序员第一次学习Python时，他们会对Python中没有private关键字感到惊讶。最接近的概念是“name mangling”（名称改编）。每当一个特性前面加上了“_”前缀，它就将被解释程序立刻重新命名，如下所示。

```
>>> class MyClass(object):
...     __secret_value = 1
...
>>> instance_of = MyClass()
>>> instance_of.__secret_value
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__secret_value'
>>> dir(MyClass)
['__MyClass__secret_value', '__class__', '__delattr__', '__dict__',
 '__doc__', '__getattribute__', '__hash__', '__init__', '__module__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__str__', '__weakref__']
>>> instance_of._MyClass__secret_value
1
```

Python提供它主要是用来避免继承带来的命名冲突，特性将被重命名为带有类名前缀的名称。这实际上并不是强制性的，因为可以通过其组合名称来访问特性。这个功能可被用来保护一些特性的访问，但是在实践中，从不使用“_”。当一个特性不是公开的时，惯例是使用一个“_”前缀。这不会调用任何改编算法，而只是证明这个特性是该类的私有元素，也是流行的样式。

Python 中还有其他可用的机制来构建类的公共部分和私有代码。描述符和属性这些 OOP 设计的关键特征应该被用于设计清晰的 API。

3.4.1 描述符

描述符用来自定义在引用一个对象上的特性时所应该完成的事情。

描述符是 Python 中复杂特性访问的基础。它们在内部使用，以实现属性、类、静态方法和 super 类等。它们是定义另一个类特性可能的访问方式的类。换句话说，一个类可以委托另一个类来管理其特性。

描述符类基于三个必须实现的特殊方法：

- `__set__` 在任何特性被设置的时候调用，在后面的实例中，将称其为 `setter`；
- `__get__` 在任何特性被读取时调用（被称为 `getter`）；
- `__delete__` 在特性上请求 `del` 时调用。

这些方法将在`__dict__`特性之前被调用。例如，指定一个 `MyClass` 的实例 `instance`，在读取 `instance.attribute` 时使用的算法如下。

```
# 1. 查找定义
if hasattr(MyClass, 'attribute'):
    attribute = MyClass.attribute
    AttributeClass = attribute.__class__
# 2. 属性定义是否有 setter
if hasattr(AttributeClass, '__set__'):
    # let's use it
    AttributeClass.__set__(attribute, instance,
                           value)
return
# 3. 常规方法
instance.__dict__['attribute'] = value
#     or 'attribute' is not found in __dict__
writable = (hasattr(AttributeClass, '__set__') or
            'attribute' not in instance.__dict__)
if readable and writable:
    # 4. 调用描述符
    return AttributeClass.__get__(attribute,
                                  instance, MyClass)
# 5. 用 __dict__ 正常访问
```

```
return instance.__dict__['attribute']
```

换句话说，在类特性被定义并且有一个 `getter` 和一个 `setter` 方法时，平常的包含一个对象实例的所有元素的 `__dict__` 映射都将被劫持。



实现了 `_get_` 和 `_set_` 的描述符被称为数据描述符。
只实现了 `_get_` 的描述符被称为非数据描述符。

下面将创建一个数据描述符，并通过一个实例来使用它。

```
>>> class UpperString(object):
...     def __init__(self):
...         self._value = ''
...     def __get__(self, instance, klass):
...         return self._value
...     def __set__(self, instance, value):
...         self._value = value.upper()
...
>>> class MyClass(object):
...     attribute = UpperString()
...
>>> instance_of = MyClass()
>>> instance_of.attribute
''
>>> instance_of.attribute = 'my value'
>>> instance_of.attribute
'MY VALUE'
>>> instance_of.__dict__ = {}
```

现在，如果在实例中添加一个新的特性，它将被保存在 `__dict__` 映射中，如下所示。

```
>>> instance_of.new_att = 1
>>> instance_of.__dict__
{'new_att': 1}
```

但是，如果将一个新的数据描述符添加到类中，它将优先于实例的 `__dict__`，如下所示。

```
>>> MyClass.new_att = MyDescriptor()
>>> instance_of.__dict__
```

```
'new_att': 1}
>>> instance_of.new_att
''

>>> instance_of.new_att = 'other value'
>>> instance_of.new_att
'OTHER VALUE'
>>> instance_of.__dict__
{'new_att': 1}
```

这对于非数据描述符无效。在那种情况下，该实例将优先于描述符，如下所示。

```
>>> class Whatever(object):
...     def __get__(self, instance, klass):
...         return 'whatever'
...
>>> MyClass.whatever = Whatever()
>>> instance_of.__dict__
{'new_att': 1}
>>> instance_of.whatever
'whatever'
>>> instance_of.whatever = 1
>>> instance_of.__dict__
{'new_att': 1, 'whatever': 1}
```

建立这个额外的规则以避免递归特性查找。

用于将一个特性设置为一个值的算法（与用于删除特性的相似）如下。

```
# 1. 查找定义
if hasattr(MyClass, 'attribute'):
    attribute = MyClass.attribute
    AttributeClass = attribute.__class__
# 2. 属性定义是否有 setter
if hasattr(AttributeClass, '__set__'):
    # let's use it
    AttributeClass.__set__(attribute, instance,
                           value)
return
# 3. 常规方法
instance.__dict__['attribute'] = value
```



Raymond Hettinger 写了一个名为 How-To Guide for Descriptors 的有趣文档，在 <http://users.rcn.com/python/download/Descriptor.htm> 中可以找到。该文章是对本小节内容的补充。

除了隐藏类的内容这个主要角色之外，描述符还可以实现一些有趣的代码模式，例如：

- 内省描述符（Introspection descriptor） 这种描述符将检查宿主类签名，以计算一些信息；
- 元描述符（Meta descriptor） 这种描述符使用类方法本身来完成值计算。

1. 内省描述符

使用类时有一个公共的需求，那就是对其特性进行一次自我测量（内省）。例如，在 Zope (<http://zope.org>) 的可发布类上计算一个安全性映射时，就实现了这种自我测量。Epydoc (<http://epydoc.sourceforge.net>) 也做了相似的工作来计算文档。

计算这种文档的属性类，可以通过检查公共方法来呈现一个易于理解的文档。以下是基于内建函数 dir 的这样一个非数据描述符的实例，它可以工作于任何对象类型之上。

```
>>> class API(object):
...     def __print_values(self, obj):
...         def __print_value(key):
...             if key.startswith('_'):
...                 return ''
...             value = getattr(obj, key)
...             if not hasattr(value, 'im_func'):
...                 doc = type(value).__name__
...             else:
...                 if value.__doc__ is None:
...                     doc = 'no docstring'
...                 else:
...                     doc = value.__doc__
...             return '%s : %s' % (key, doc)
...         res = [__print_value(el) for el in dir(obj)]
...         return '\n'.join([el for el in res
...                         if el != ''])
...     def __get__(self, instance, klass):
```

```

...
        if instance is not None:
            return self._print_values(instance)
        else:
            return self._print_values(klass)

...
>>> class MyClass(object):
...
    __doc__ = API()
...
    def __init__(self):
...
        self.a = 2
...
    def meth(self):
...
        """my method"""
...
        return 1
...
...
>>> MyClass.__doc__
'meth : my method'
>>> instance = MyClass()
>>> print instance.__doc__
a : int
meth : my method

```

这个描述符将过滤以下划线开始的元素，并且显示方法的 docstrings。

2. 元描述符

元描述符使用宿主类的一个或多个方法来执行一个任务。这可能对降低使用提供步骤的类所需的代码量很有用，比如，一个链式的描述符可以调用类的一系列方法以返回一组结果。它可以在失败的时候被停止，并且配备一个回调机制以获得对过程的更多控制，如下所示。

```

>>> class Chainer(object):
...
    def __init__(self, methods, callback=None):
...
        self._methods = methods
...
        self._callback = callback
...
    def __get__(self, instance, klass):
...
        if instance is None:
            # 只针对实例
            return self
...
        results = []
...
        for method in self._methods:
            results.append(method(instance))

```

```

...
    if self._callback is not None:
        if not self._callback(instance,
...
...
...
...
            method,
            results):
...
        break
...
    return results

```

这一实现使各种在类方法之上运行的计算能与记录器这样的外部元素结合起来，如下所示。

```

>>> class TextProcessor(object):
...
    def __init__(self, text):
...
        self.text = text
...
    def normalize(self):
...
        if isinstance(self.text, list):
...
            self.text = [t.lower()
...
                        for t in self.text]
...
        else:
...
            self.text = self.text.lower()
...
    def split(self):
...
        if not isinstance(self.text, list):
...
            self.text = self.text.split()
...
    def threshold(self):
...
        if not isinstance(self.text, list):
...
            if len(self.text) < 2:
...
                self.text = ''
...
            self.text = [w for w in self.text
...
                            if len(w) > 2]
...
...
>>> def logger(instance, method, results):
...
    print 'calling %s' % method.__name__
...
    return True
...
...
>>> def add_sequence(name, sequence):
...
    setattr(TextProcessor, name,
...
            Chainer([getattr(TextProcessor, n)
...
                    for n in sequence], logger))

```

`add_sequence` 用来动态地定义一个新的链式调用方法的描述符。这一组合的结果可以被保存在类定义中，如下所示。

```

>>> add_sequence('simple_clean', ('split', 'threshold'))
>>> my = TextProcessor(' My Taylor is Rich ')
>>> my.simple_clean
calling split
calling threshold
[None, None]
>>> my.text
['Taylor', 'Rich']
>>> # 执行另一个序列
>>> add_sequence('full_work', ('normalize',
...                               'split', 'threshold'))
>>> my.full_work
calling normalize
calling split
calling threshold
[None, None, None]
>>> my.text
['taylor', 'rich']

```

由于 Python 的动态特性，可以在运行时再添加这种描述符以执行元编程。

定义

元编程（Meta-programming）是在运行时通过添加新的计算功能，或者改变已有功能来改变程序行为的一种技巧。它与普通的编程（具备 C++ 背景的人可能对此比较熟悉）不同。在 C++ 中，将需要创建新的代码块，而不是提供可以应对最多情况的简单代码块。

它也和“生成性编程”（generative programming）不同，这种编程方式是通过模板来生成一段静态的源代码。具体信息可参见 http://en.wikipedia.org/wiki/Generative_programming。

3.4.2 属性

属性（Property）提供了一个内建的描述符类型，它知道如何将一个特性链接到一组方法上。属性采用 `fget` 参数和 3 个可选的参数——`fset`、`fdel` 和 `doc`。最后一个参数可以提供用来

定义一个链接到特性的 docstring，就像是个方法一样，如下所示。

```
>>> class MyClass(object):
...     def __init__(self):
...         self._my_secret_thing = 1
...
...     def __get__(self):
...         return self._my_secret_thing
...
...     def __set__(self, value):
...         self._my_secret_thing = value
...
...     def __delete__(self):
...         print 'neh!'
...
...     my_thing = property(__get__, __set__, __delete__,
...                         'the thing')
...
>>> instance_of = MyClass()
>>> instance_of.my_thing
1
>>> instance_of.my_thing = 3
>>> instance_of.my_thing
3
>>> del instance_of.my_thing
neh !
>>> help(instance_of)
Help on MyClass in module __main__ object:
class MyClass(__builtins__.object)
| Methods defined here:
|
|   __init__(self)
|
|   -----
| Data descriptors defined here:
|
|   ...
|   my_thing
|   the thing
```

属性简化了描述符的编写，但是在使用类的继承时必须小心处理。所创建的特性使用当前类的方法创建，而不应使用在派生类中重载的方法。这有些混乱，因为后者是大部分实现属性的语言中相当合乎逻辑的行为。

例如，以下的例子将不能像预想中那样工作。

```
>>> class FirstClass(object):
...     def _get_price(self):
...         return '$ 500'
...     price = property(_get_price)
...
>>> class SecondClass(FirstClass):
...     def _get_price(self):
...         return '$ 20'
...
...
...
>>> plane_ticket = SecondClass()
>>> plane_ticket.price
'$ 500'
```

对这种行为的解决方法是，使用另一种方法手工将属性实例重定向到正确的方法上，如下所示。

```
>>> class FirstClass(object):
...     def _get_price(self):
...         return '$ 500'
...     def _get_the_price(self):
...         return self._get_price()
...     price = property(_get_the_price)
...
>>> class SecondClass(FirstClass):
...     def _get_price(self):
...         return '$ 20'
...
...
>>> plane_ticket = SecondClass()
>>> plane_ticket.price
'$ 20'
```

尽管如此，大部分时候属性都将被添加到类中，以隐藏其复杂性。链接到它们的方法是私有的，所以重载它们是不好的做法。在这种情况下，重载属性本身更好一些，如下所示。

```
>>> class FirstClass(object):
...     def _get_price(self):
...         return '$ 500'
...     price = property(_get_price)
...
>>> class SecondClass(FirstClass):
...     def _cheap_price(self):
...         return '$ 20'
...     price = property(_cheap_price)
...
>>> plane_ticket = SecondClass()
>>> plane_ticket.price
'$ 20'
```

3.5 槽

几乎从未被开发人员使用过的一种有趣的特性是槽。它允许使用`_slots_`特性为指定的类设置一个静态特性列表，并且跳过每个类实例中`_dict_`列表的创建工作。它们用来为特性很少的类节省存储空间，因为将不在每个实例中创建`_dict_`。

除此之外，它们有助于设计签名必须被冻结的类。例如，如果必须限制类之上的语言动态特性，定义槽也是有帮助的，如下所示。

```
>>> class Frozen(object):
...     __slots__ = ['ice', 'cream']
...
>>> '__dict__' in dir(Frozen)
False
>>> 'ice' in dir(Frozen)
True
>>> glagla = Frozen()
>>> glagla.ice = 1
>>> glagla.cream = 1
>>> glagla.icy = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Frozen' object has no attribute 'icy'
```

因为任何新的特性都将在`_dict_`中被添加，所以这无法在派生类上工作。

3.6 元 编 程

`new-style`类带来了一种能力，可以通过两个特殊方法——`__new__`和`__metaclass__`在运行时修改类和对象的定义。

3.6.1 `__new__`方法

特殊方法`__new__`是一个元构造程序，每当一个对象必须被factory类实例化时就将调用它，如下所示。

```
>>> class MyClass(object):
...     def __new__(cls):
...         print '__new__ called'
...         return object.__new__(cls) # default factory
...     def __init__(self):
...         print '__init__ called'
...         self.a = 1
...
>>> instance = MyClass()
__new__ called
__init__ called
```

`__new__`方法必须返回一个类的实例。因此，它可以在对象创建之前或之后修改类。这对于确保对象构造程序不会被设置成一个不希望的状态，或者添加一个不能被构造程序删除的初始化是有帮助的。

例如，因为`__init__`在子类中不会被隐式调用，所以`__new__`可以用来确定已经在整个类层次中完成了初始化工作，如下所示。

```
>>> class MyOtherClassWithoutAConstructor(MyClass):
...     pass
>>> instance = MyOtherClassWithoutAConstructor()
__new__ called
__init__ called
```

```
>>> class MyOtherClass(MyClass):
...     def __init__(self):
...         print 'MyOther class __init__ called'
...         super(MyOtherClass, self).__init__()
...         self.b = 2
...
...
>>> instance = MyOtherClass()
__new__ called
MyOther class __init__ called
__init__ called
```

例如，网络套接字或数据库初始化应该在`__new__`中而不是`__init__`中控制。它在类的工作必须完成这个初始化以及必须被继承的时候通知我们。

例如，`threading`模块中的`Thread`类就使用了这种机制，以避免存在未初始化的实例，如下所示。

```
>>> from threading import Thread
>>> class MyThread(Thread):
...     def __init__(self):
...         pass
...
...
>>> MyThread()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/2.5/lib/
python2.5/threading.py", line 416, in __repr__
    assert self._initialized, "Thread.__init__() was not called"
AssertionError: Thread.__init__() was not called
```

这实际上是通过方法之上的断言来完成的(`assert self._initialized`)，并且可以简化为`__new__`中的单一调用，因为这个实例除此之外就没有什么作用。

避免令人头痛的链式初始化

`__new__`是对于对象状态隐式初始化需求的回应。它使得可以在比`__init__`更低的层次上定义一个初始化，这个初始化总是会被调用。

3.6.2 __metaclass__方法

元类（Metaclass）提供了在类对象通过其工厂方法（factory）在内存中创建时进行交互的能力。它们的效果与`__new__`类似，只不过是在类级别上运行。内建类型`type`是内建的基本工厂，它用来生成指定名称、基类以及包含其特性的映射的任何类，如下所示。

```
>>> def method(self):
...     return 1
...
>>> klass = type('MyClass', (object,), {'method': method})
>>> instance = klass()
>>> instance.method()
1
```

这与类的显式定义类似，如下所示。

```
>>> class MyClass(object):
...     def method(self):
...         return 1
...
>>> instance = MyClass()
>>> instance.method()
1
```

有了这个功能，开发人员可以在调用`type`之前或之后与类创建交互。一个特殊的特性已经被创建以链接到一个定制的工厂上。

`__metaclass__`（在 Python 3000 中，其将被一个显式的构造程序参数替代）可以被添加到一个类定义中，以与创建过程进行交互。`__metaclass__`特性必须被设置为：

- 接受和`type`相同的参数（即，一个类名、一组基类和一个特性映射）；
- 返回一个类对象。

完成以上事情的是类似下面使用的那种无束缚的函数（`equip` 函数），还有另一个类对象上的一个方法，这些并不重要，只要它能够满足规则 1 和 2 就行。在这个实例中，如果类有一个空的`docstring`，那么在 3.4.1 小节中提出的 API 描述符就将被自动地添加到类中，如下所示。

```
>>> def equip(classname, base_types, dict):
...     if '__doc__' not in dict:
```

```

...
    dict['__doc__'] = API()
...
    return type(classname, base_types, dict)
...

>>> class MyClass(object):
...
    __metaclass__ = equip
...
    def alright(self):
...
        """the ok method"""
...
        return 'okay'
...

>>> ma = MyClass()
>>> ma.__class__
<class '__main__.MyClass'>
>>> ma.__class__.__dict__['__doc__'] # __doc__ is replaced !
<__main__.API object at 0x621d0>
>>> ma.y = 6
>>> print ma.__doc__
    alright : the ok method
    y : int

```

这个变化在其他地方可能不可行，因为`__doc__`是内建的基本元类`type`的只读特性。

但是元类使代码变得更加复杂，而且在将其用于工作于所有类型的类时，会使其健壮性变得更差。例如，可能在类中使用槽时遇到不好的交互，或者在一些基类已经实现了一个元类时，这个类与所做的冲突。可能它们只是没有构造好。

对于修改可读写的特性或添加新特性而言，可以避免使用元类，而采用更简单的基于动态修改类实例的解决方案。这些修改更容易管理，因为它们不需要被组合到一个类中（一个类只能有一个元类）。例如，如果两个具体的行为必须被应用到一个类，可以使用一个“增强函数”来添加它们，如下所示。

```

>>> def enhancer_1(klass):
...
    c = [l for l in klass.__name__ if l.isupper()]
    klass.contracted_name = ''.join(c)
...
>>> def enhancer_2(klass):
...
    def logger(function):
...
        def wrap(*args, **kw):
...
            print 'I log everything !'
...
            return function(*args, **kw)

```

```

...
    return wrap
...
for el in dir(klass):
...
    if el.startswith('_'):
...
        continue
...
    value = getattr(klass, el)
...
    if not hasattr(value, 'im_func'):
...
        continue
...
    setattr(klass, el, logger(value))
...
...
>>> def enhance(klass, *enhancers):
...
    for enhancer in enhancers:
...
        enhancer(klass)
...
...
>>> class MySimpleClass(object):
...
    def ok(self):
...
        """I return ok"""
...
        return 'I lied'
...
...
>>> enhance(MySimpleClass, enhancer_1, enhancer_2)
>>> thats = MySimpleClass()
>>> thats.ok()
I log everything !
'I lied'
>>> thats.score
>>> thats.contracted_name
'MSC'

```

这是很强大的表现——可以动态地在已经被实例化的类定义上创建许多不同的变化。在任何情况下,请记住元类或动态增强只是一种“补丁”,它可能会很快地使精心定义的、清晰的类层次结构变得一团糟。它们应该只在以下情况下使用:

- 在框架级别,一个行为在许多类中是强制的时候;
- 当一个特殊的行为被添加的目的不是与诸如记录日志这样的类提供的功能交互时。

更多的例子,可以参看 David Mertz 对元类编程的一个很棒的介绍,其链接为 <http://www.onlamp.com/pub/a/python/2003/04/17/metaclasses.html?page=1>。

3.7 小结

本章重点介绍了以下内容。

- 子类型化内建类型是个很好的特性，但是在这么做之前，应确定不对现有的类型进行子类型化是不合适的。
- 因为 `super` 的使用十分棘手，所以：在代码中避免使用多重继承；用法要保持一致，不要混用新旧样式的类；在子类中调用方法之前应先检查类层次。
- 通过描述符可以自定义在一个对象上引用特性时所应该做的任务。
- 属性对于构建一个公共 API 是很棒的。
- 元编程非常强大，但是记住，它会影响类设计的易读性。

下一章中将重点介绍如何为编码元素选择好的名称，以及 API 设计的最佳实践。

第 4 章

选择好的名称

大部分标准程序库在构建时都不会忽略可用性。例如，内建类型的使用是很自然的，并且设计得十分易于使用。在这方面，Python 可以和建立程序时所考虑的伪代码相比，大部分代码都可以朗读出来。例如，以下片段大家都会感到很容易理解。

```
>>> if 'd' not in my_list:  
...     my_list.append('d')
```

这是编写 Python 程序和使用其他语言相比更加简单的原因之一。当编写一段程序时，你的思路很快就能转化成为代码行。

本章将关注于与编写易于理解和使用的代码相关的最佳实践，包括：

- 使用 PEP8 中描述的命名约定，以及一组命名最佳实践；
- 命名空间重构；
- 使用 API，从初始模型到其重构模型。

4.1 PEP 8 和命名最佳实践

PEP 8 (<http://www.python.org/dev/peps/pep-0008>) 为 Python 编程提供了一个与编码风格相关的指南。除了空格缩进、每行最大长度以及其他与代码布局有关的细节之类的基本规则以外，PEP8 还提供了一个大部分代码库所遵循的命名约定。

本节只是摘要性地介绍了 PEP，并且给出了针对每种元素的最佳实践指南。

4.2 命名风格

Python 中使用不同的命名风格包括：

- CamelCase（每个单词首字母大写）；
- mixedCase（和 CamelCase 类似，但是第一个单词的首字母仍然为小写）；
- UPPERCASE（大写）和 ER_CASE_WITH_UNDERSCORES（大写并且带下划线）；
- lowercase（小写）和 lower_case_with_underscores（小写并且带下划线）；
- 前缀（_leading）和后缀（trailing_）下划线，或者都加下划线（_doubled_）。

小写和大写元素通常是一个单词，有时候是少数几个词的连接。使用下划线的通常是缩写词。使用单词会更好一些。前缀和后缀下划线用来标记私有和特殊的元素。

这些风格将被应用到：

- 变量；
- 函数和方法；
- 属性；
- 类；
- 模块；
- 包。

4.2.1 变量

Python 中有两种变量：

- 常量；
- 公有和私有变量。

1. 常量

对于值不会发生改变的全局变量，使用大写和一个下划线。它告诉开发人员指定的变量代表一个恒定值。

Python 中没有真正的常量——C++中用 `const` 定义的那种常量。任何变量值都可以修改。这就是 Python 使用命名约定来将一个变量标记为常量的原因。

例如，`doctest` 模块提供一个选项标志和指令（都是一些短小的句子，清晰地定义了每个选项的用途）的列表（参见<http://docs.python.org/lib/doctest-options.html>），如下所示。

```
>>> from doctest import IGNORE_EXCEPTION_DETAIL
>>> from doctest import REPORT_ONLY_FIRST_FAILURE
```

这些变量名看上去相当长，但是清晰地描述它们很重要。基本上，都在初始化代码中使用它们，而不会在代码本身的主体中使用，所以冗长的名称并不会令人厌烦。

 大部分时候，缩写的名称会使代码变得模糊。在缩写名称不够清晰时，不要害怕使用完整的词语。

一些常量的名称也是基于低层技术派生的。例如，`os` 模块使用 C 上定义的一些常量，诸如 `EX_XXX` 系列定义了异常编号，如下所示。

```
>>> import os
>>> try:
...     os._exit(0)
... except os.EX_SOFTWARE:
...     print 'internal softwar error'
...     raise
```

使用常量时，将它们集中放在模块的头部是一个好办法。当它用于如下所示的操作时，应将其组合在新的变量之下。

```
>>> import doctest
>>> TEST_OPTIONS = (doctest.ELLIPSIS |
...                   doctest.NORMALIZE_WHITESPACE |
...                   doctest.REPORT_ONLY_FIRST_FAILURE)
```

2. 命名和使用

常量用来定义一组程序所依赖的值，如默认配置文件名称。

将所有常量集中放在包中的独立文件中是一种好方法。例如，Django 采用的就是这种方式。它使用一个名为 `config.py` 的模块来提供所有常量，如下所示。

```
# config.py
SQL_USER = 'tarek'
SQL_PASSWORD = 'secret'
```

```
SQL_URI = 'postgres://%s:%s@localhost/db' % \
    (SQL_USER, SQL_PASSWORD)
MAX_THREADS = 4
```

另一种方法是使用可以被 ConfigParser 模块或者像 ZConfig 之类的高级工具(用于 Zope 中描述其配置文件的解析器)解析的配置文件。但是有些人认为,在 Python 这种程序文件能够像文本文件一样易于编辑和修改的语言中,采用这种方法是对另一种文件格式的过度使用。

对于表现得像标志的选项,将它们和布尔操作组合是一种好方法,就像 doctest 和 re 模块那样。从 doctest 得到的模式很简单,如下所示。

```
>>> OPTIONS = {}
>>> def register_option(name):
...     return OPTIONS.setdefault(name, 1 << len(OPTIONS))
>>> def has_option(options, name):
...     return bool(options & name)
>>> # 现在定义选项
>>> BLUE = register_option('BLUE')
>>> RED = register_option('RED')
>>> WHITE = register_option('WHITE')
>>>
>>> # 然后尝试它们
>>> SET = BLUE | RED
>>> has_option(SET, BLUE)
True
>>> has_option(SET, WHITE)
False
```

当创建这样一组新的常量时,应避免使用常用的前缀,除非模块中有多个组。模块名本身就是一个常见的前缀。

 在 Python 中, 使用二进制逐位计算来组合选项是常见的方法。使用或(|)操作符可以将多个选项组合在一个整数中,而使用与(&)操作符将能够检查整数中表示的选项(参见 has_option 函数)。

如果这个整数可以使用<<操作符移位,以确保和组合得到的整数中的另一个不相同,那么它就能起作用。换句话说,它是 2 的幂(参见 register_options)。

3. 公有和私有变量

对于易变和公有的全局变量，当它们需要被保护时应该使用小写和一个下划线。但是这种变量不常使用，因为在需求保护时，模块通常会提供 `getter` 和 `setter` 来处理它们。在这种情况下，一个前导下划线可以表示该变量为包的私有元素，如下所示。

```
>>> _observers = []
>>> def add_observer(observer):
...     _observers.append(observer)
>>> def get_observers():
...     """Makes sure _observers cannot be modified."""
...     return tuple(_observers)
```

位于函数和方法中的变量遵循相同的规则，并且永远不会被标志为私有，因为它们对上下文来说是局部的。

对于类或实例变量而言，只在将变量作为公共签名的一部分，并且不能带来任何有用的信息或冗余的情况下才必须使用私有标志（即前导下划线）。

换句话说，如果该变量是在方法内部使用，用来提供一个公共特性，并且只扮演这个角色，那么最好是将其声明为私有变量。

例如，只支持一个属性的特性是好的私有成员，如下所示。

```
>>> class Citizen(object):
...     def __init__(self):
...         self._message = 'Go boys'
...     def _get_message(self):
...         return self._message
...     kane = property(_get_message)
>>> Citizen().kane
'Go boys'
```

另一个例子是用来保持内部状态的变量。这个值对于余下的代码没有用处，但是它参与了类的行为，如下所示。

```
>>> class MeanElephant(object):
...     def __init__(self):
...         self._people_to_kill = []
...     def is_slapped_on_the_butt_by(self, name):
...         self._people_to_kill.append(name)
...         print 'Ouch!'
```

```
...     def revenge(self):
...         print '10 years later...'
...         for person in self._people_to_kill:
...             print 'Me kill %s' % person
>>> joe = MeanElephant()
>>> joe.is_slapped_on_the_butt_by('Tarek')
Ouch!
>>> joe.is_slapped_on_the_butt_by('Bill')
Ouch!
>>> joe.revenge()
10 years later...
Me kill Tarek
Me kill Bill
```

[ 不要轻易断言类进行子类化时可能采用的方式。]

4.2.2 函数和方法

函数和方法命名时应该使用小写和下划线。但是，在标准程序库中并不总是遵守这个规则，可以找到一些使用混合大小写(mixedCase)的模式，例如 `threading` 模块中的 `currentThread`(这在 Python 3000 中可能会发生改变)。

这种编写方法的方式在小写范式成为标准之前很常见，在诸如 `Zope` 之类的框架中也使用了混合大小写的方法，使用它的开发人员群体相当大。所以，在混合大小写和小写加下划线之间做什么选择，主要取决于所使用的程序库。

作为一个 `Zope` 开发人员，保持一致性不容易，因为构建一个混合纯 `Python` 模块和引用了 `Zope` 代码的模块的应用程序很困难。在 `Zope` 中，有一些类混用了两种约定，因为代码库正在演变成一个基于 `egg` 的框架，因此每个模块都比之前更接近于纯 `Python`。

在这种类型的程序库环境中，最正统的方法是只对被输出到框架中的元素使用混合大小写，其余代码保持使用 PEP 风格。

1. 关于私有元素的争论

对于私有的方法和函数而言，命名惯例是添加一个前导下划线。考虑到 `Python` 的名称改编特性，这个规则是相当有争议的。当方法有两个前导下划线时，解释程序会立即对其更名，以避免和子类中的方法产生冲突。

所以有些人倾向于对其私有的特性使用双前导下划线，以避免子类中的命名冲突，如下所示。

```
>>> class Base(object):
...     def __secret(self):
...         print "don't tell"
...     def public(self):
...         self.__secret()
>>> Base.__secret
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Base' has no attribute '__secret'
>>> dir(Base)
['__Base__secret', ..., 'public']
>>> class Derived(Base):
...     def __secret(self):
...         print "never ever"
>>> Derived().public()
don't tell
```

Python 中设计名称改编 (name mangling) 的原始动机不是提供类似 C++ 中的私有机关 (gimmick)，而是用来确保一些基类隐式地避免子类中的冲突，尤其是在多继承环境中。但是如果针对每个特性都使用它，就会使代码变得模糊不清，这根本不是 Python 的风格。

因此，有些人认为应该始终使用显式的名称改编，如下所示。

```
>>> class Base(object):
...     def __Base_secret(self):      # 别这样干!!!
...         print "you told it ?"
```

该类名将在整个代码中不断重复，因此 `__` 是首选的。

但是最佳的方法，正如 BDFL (Guido, Benevolent Dictator For Life，参见 [http://en.wikipedia.org/wiki/BDFL¹](http://en.wikipedia.org/wiki/BDFL)) 所说，是通过在子类中编写方法之前查看 `__mro__` 值 (方法解析顺序) 来避免名称改编。改变基类的私有方法必须小心进行。

关于这个主题的更多信息，几年前在 `python-dev` 邮件列表上出现过一个有趣的思路，人们争论名称改编的实用性及其在这种语言中的命运。具体信息可以在 <http://mail.python.org/>

¹ 译者注：Guido van Rossum 是 Python 发明者，人称 Benevolent Dictator For Life (缩写为 BDFL，即仁慈大帝)，这个称号常被用来称呼少数的开放源码软件开发领导者。

pipermail/python-dev/2005-December/058555.html 上找到。

2. 特殊的方法

特殊方法 (<http://docs.python.org/ref/specialnames.html>) 是以两个下划线开始和结束的，常规方法不应该使用这种命名约定。它们被用于操作符重载、容器定义等。为了使程序易读，它们应该被集中放在类定义的最前面，如下所示。

```
>>> class weirdint(int):
...     def __add__(self, other):
...         return int.__add__(self, other) + 1
...     def __repr__(self):
...         return '<weirdo %d>' % self
...
...     #
...     # 公有 API
...     #
...     def do_this(self):
...         print 'this'
...     def do_that(self):
...         print 'that'
```

对于常规的方法，绝不应该使用这种名称，所以不要创建诸如以下这种方法名称。

```
>>> class BadHabits(object):
...     def __my_method__(self):
...         print 'ok'
```

3. 参数

参数使用小写，如果需要的话可以加上下划线。它们遵循与变量相同的命名规则。

4.2.3 属性

属性名称是用小写或者小写加上下划线命名的。大部分时候，它们表示对象的状态，可以是一个名词或一个形容词，在需要的时候也可以是一个小短语，如下所示。

```
>>> class Connection(object):
...     _connected = []
...     def connect(self, user):
...         self._connected.append(user)
```

```

...     def _connected_people(self):
...         return '\n'.join(self._connected)
...     connected_people = property(_connected_people)
>>> my = Connection()
>>> my.connect('Tarek')
>>> my.connect('Shannon')
>>> print my.connected_people
Tarek
Shannon

```

4.2.4 类

类的名称总是使用 CamelCase 格式命名，当定义的是模块的私有类时，还可能有一个前导下划线。

类和实例变量常常是名词短语，其使用逻辑与用动词短语命名方法一致，如下所示。

```

>>> class Database(object):
...     def open(self):
...         pass
>>> class User(object):
...     pass
>>> user = User()
>>> db = Database()
>>> db.open()

```

4.2.5 模块和包

除了特殊模块 `_init_` 之外，模块名称都使用不带下划线的小写字母命名。

以下是一些标准程序库中的例子：

- `os`;
- `sys`;
- `shutil`。

当模块对于包而言是私有的时候，将添加一个前导下划线。编译过的 C 或 C++ 模块名称通常带有一个下划线并且将导入到纯 Python 模块中。

包将遵循相同的规则，因为它们在命名空间中的表现和模块类似。

4.3 命名指南

一组常用的命名规则可以被应用到变量、方法函数和属性上。类和模块的名称在命名空间的构建中也扮演重要的角色，从而也影响了代码易读性。本指南中将提供命名的常见模式和反模式。

4.3.1 使用“has”或“is”前缀命名布尔元素

当一个元素是用来保存布尔值时，“is”和“has”前缀提供一个自然的方式，使其在命名空间中很容易被理解，如下所示。

```
>>> class DB(object):
...     is_connected = False
...     has_cache = False
>>> database = DB()
>>> database.has_cache
False
>>> if database.is_connected:
...     print "That's a powerful class"
... else:
...     print "No wonder..."
No wonder...
```

4.3.2 用复数形式命名序列元素

当一个元素是用来保存一个序列时，使用复数形式命名是个好主意。对一些以类似序列的形式输出的映射而言，也可以从中获益，如下所示。

```
>>> class DB(object):
...     connected_users = ['Tarek']
...     tables = {'Customer': ['id', 'first_name',
...                           'last_name']}
```

4.3.3 用显式的名称命名字典

当一个变量是用来保存一个映射时，应该尽可能使用显式的名称。例如，有一个用来保

存个人地址的 dict，那么可以将其命名为 person_address，如下所示。

```
>>> person_address = {'Bill': '6565 Monty Road',
...                      'Pamela': '45 Python street'}
>>> person_address['Pamela']
'45 Python street'
```

4.3.4 避免通用名称

如果在代码中不构建一个新的抽象数据类型，使用诸如 list、dict、sequence 或 elements 这样的名称是有害的，即使对于局部变量也一样。它将使代码难以阅读理解和使用。还应该避免使用内建的名称，以避免在当前命名空间中遮蔽它。还要避免通用的动词，除非它们在该命名空间中有意义。

作为替代，应该使用与问题域相关的词语，如下所示。

```
>>> def compute(data): # too generic
...     for element in data:
...         yield element * 12
>>> def display_numbers(numbers): # better
...     for number in numbers:
...         yield number * 12
```

4.3.5 避免现有名称

使用已经存在于上下文中的名称也是一个坏习惯，因为它会使得在阅读程序时，特别是调试时产生很多混乱，如下所示。

```
>>> def bad_citizen():
...     os = 1
...     import pdb; pdb.set_trace()
...     return os
>>> bad_citizen()
> <stdin>(4)bad_citizen()
(Pdb) os
1
(Pdb) import os
(Pdb) c
<module 'os' from '/Library/Frameworks/Python.framework/Versions/2.5/
```

```
lib/python2.5/os.  
pyc'>
```

在这个例子中，os名称将被代码遮蔽。内建的来自标准程序库的模块名称都应该避免。

尝试创建独特的名称，即使它们是上下文中的局部名称。对于关键字而言，添加一个后缀下划线是避免冲突的一种方法，如下所示。

```
>>> def xapian_query(terms, or_=True):  
...     """if or_ is true, terms are combined  
...     with the OR clause"""  
...     pass
```

注意，class常常被改成klass或cls，如下所示。

```
>>> def factory(klass, *args, **kw):  
...     return klass(*args, **kw)
```

4.4 参数最佳实践

函数和方法的签名是代码完整性的保证，它们驱动函数和方法的使用并构建其 API。除了我们已经了解的命名规则，对参数也要特别小心。下面是3个简单的规则：

- 根据迭代设计构建参数；
- 信任参数和测试；
- 小心使用魔法参数*args 和 **kw。

4.4.1 根据迭代设计构建参数

如果每个函数都有一个固定的、精心设计的参数列表，会使代码更加健壮。但是这在第一个版本中不能完成，所以参数必须根据迭代设计来构建。它们应该反映创建该元素所针对的使用场景，并且相应地进行演化。

例如，当附加一些参数时，它们应该尽可能有默认值以避免任何退化，如下所示。

```
>>> class BD(object): # version 1  
...     def _query(self, query, type):  
...         print 'done'  
...     def execute(self, query):  
...         self._query(query, 'EXECUTE')
```

```

>>> BD().execute('my query')
done
>>> import logging
>>> class BD(object): # version 2
...     def _query(self, query, type, logger):
...         logger('done')
...     def execute(self, query, logger=logging.info):
...         self._query(query, 'EXECUTE', logger)
>>> BD().execute('my query') # old-style call
>>> BD().execute('my query', logging.warning)
WARNING:root:done

```

当一个公共元素的参数必须变化时，将使用一个 `deprecation` 进程，本节稍后部分将对此进行说明。

4.4.2 信任参数和测试

基于 Python 的动态类型特性的考虑，有些开发人员在函数和方法的头部使用断言来确保参数有正确的内容，如下所示。

```

>>> def division(dividend, divisor):
...     assert type(dividend) in (long, int, float)
...     assert type(divisor) in (long, int, float)
...     return dividend / divisor
>>> division(2, 4)
0
>>> division(2, 'okok')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in division
AssertionError

```

这通常是那些习惯于动态类型并且感觉 Python 中缺少点什么的开发人员的杰作。

这种检查参数的方法是契约式设计编程风格 (Design by Contract, 参见 http://en.wikipedia.org/wiki/Design_By_Contract) 的一部分。在这种设计中，代码在实际运行之前会先检查预言。

这种方法的两个主要问题是：

- (1) DBC 的代码解释它应该如何使用，导致程序易读性降低；
- (2) 这可能使代码执行速度变得更慢，因为每次调用都要进行断言。

后者可以通过在执行解释程序时加上“-O”选项来避免。在这种情况下，所有断言都将在字节码被创建之前从代码中删除，这样检查也就会丢失。

在任何情况下，断言都必须小心进行，并且不应该导致 Python 变成一种静态类型的语言。唯一的使用场景就是保护代码不被无意义地调用。

一个健康的测试驱动开发风格在大部分情况下将提供健壮的基础代码。在这里，功能和单元测试验证了所有创建代码所针对的使用场景。

当程序库中的代码被外部元素使用时，建立断言可能是有用的，因为输入数据可能会导致程序结束甚至造成破坏。这在处理数据库或文件系统的代码中是很可能发生的。

另一种方法是“模糊测试（fuzz testing）”(http://en.wikipedia.org/wiki/Fuzz_testing)，它通过向程序发送随机的数据块以检测其弱点。当发现一个新的缺陷时，代码可以相应地被修复，并加上一次新的测试。

让我们来关注一个遵循 TDD 方法，向正确的方向演变并且每当新的缺陷出现时进行调整，从而使健壮性越来越好的代码库。当它以正确的方式完成时，测试中的断言列表在某种程度上变得和预言列表相似。

不管怎么说，Python 中已经有许多为爱好者所做的 Dbc 程序库，可以在 *Contracts for Python* 中看到 (<http://www.wayforward.net/pycontract/>)。

4.4.3 小心使用*args 和**kw 魔法参数

*args 和**kw 参数可能会破坏函数或方法的健壮性。它们会使签名变得模糊，而且代码常常开始在不应该出现的地方构建小的参数解析器，如下所示。

```
>>> def fuzzy_thing(**kw):
...     if 'do_this' in kw:
...         print 'ok i did'
...     if 'do_that' in kw:
...         print 'that is done'
...     print 'errr... ok'
>>> fuzzy_thing()
errr... ok
>>> fuzzy_thing(do_this=1)
ok i did
errr... ok
>>> fuzzy_thing(do_that=1)
that is done
errr... ok
```

```
>>> fuzzy_thing(hahahahaha=1)
errr... ok
```

如果参数列表很长而且很复杂，那么添加魔法参数是很有诱惑力的。但这通常意味着，它是一个脆弱的函数或方法，它们应该被分解或者重构。

当`*args`被用来处理一系列在函数中以同样方式处理的元素时，要求传入诸如`iterator`之类的唯一容器参数会更好些，如下所示。

```
>>> def sum(*args): #可行
...     total = 0
...     for arg in args:
...         total += arg
...     return total
>>> def sum(sequence): #更好!
...     total = 0
...     for arg in args:
...         total += arg
...     return total
```

对于`**kw`而言，适用同样的规则。修复命名的参数，会使方法签名更有意义，如下所示。

```
>>> def make_sentence(**kw):
...     noun = kw.get('noun', 'Bill')
...     verb = kw.get('verb', 'is')
...     adj = kw.get('adjective', 'happy')
...     return '%s %s %s' % (noun, verb, adj)
>>> def make_sentence(noun='Bill', verb='is', adjective='happy'):
...     return '%s %s %s' % (noun, verb, adjective)
```

另一个有趣的方法是创建一个聚集多个相关参数以提供执行环境的容器类。这种结构与`*args`或`**kw`不同，因为它可以提供工作于数值之上并且能够独立演化的内部构件。将它当作参数来使用的代码将不必处理其内部构件。

例如，传递到一个函数的 Web 请求通常由一个类实例表示。这个类负责保存 Web 服务器传递的数据，如下所示。

```
>>> def log_request(request): #版本1
...     print request.get('HTTP_REFERER', 'No referer')
>>> def log_request(request): #版本2
...     print request.get('HTTP_REFERER', 'No referer')
...     print request.get('HTTP_HOST', 'No host')
```

魔法参数有时候是无法避免的，特别是在元编程的时候，例如，针对带有任何种类签名的函数的装饰器（decorator）。总地来说，当涉及仅仅位于函数中的未知数据时，魔法参数很擅长，如下所示。

```
>>> import logging  
>>> def log(**context):  
...     logging.info('Context is:\n%s\n' % str(context))
```

4.5 类 名

类的名称必须简明、精确，并足以从中理解类所完成的工作。常见的一个方法是使用表示其类型或特性的后缀，例如：

- **SQLEngine**
- **MimeTypes**
- **StringWidget**
- **TestCase**

对于基类而言，可以使用一个 **Base** 或 **Abstract** 前缀，如：

- **BaseCookie**
- **AbstractFormatter**

最重要的是要和类特性保持一致。例如，应尝试避免类及其特性名称之间的冗余，如下所示。

```
>>> SMTP.smtp_send()      # 命名空间中存在冗余信息  
>>> SMTP.send()          # 可读性更强，也更易于记忆
```

4.6 模块和包名称

在模块和包的名称中，应该体现出其内容的用途。这些名称应简短、使用小写字母，不使用下划线，例如：

- **sqlite**
- **postgres**
- **sha1**

如果它们实现一个协议，那么通常会使用 **lib** 前缀，如下所示。

```
>>> import smtplib
```

```
>>> import urllib
>>> import telnetlib
```

它们在命名空间中也必须保持一致，这样使用起来会更简单，如下所示。

```
>>> from widgets.stringwidgets import TextWidget      # 不好的
>>> from widgets.strings import TextWidget           # 更好的
```

同样，应该始终避免使用与来自标准程序库的模块相同的名称。

当一个模块变得比较复杂、包含许多类时，创建一个包并将模块的元素分到其他模块中是一个好习惯。

`_init_` 模块也可以用来将一些 API 放回最高级别中，因为它不会影响这些 API 的使用，而且帮助将代码组织为更小的部件。例如，`foo` 包中的一个模块，如下所示。

```
from module1 import feature1, feature2
from module2 import feature3
```

将允许用户直接地导入功能，如下所示。

```
>>> from foo import feature1
>>> from foo import feature2, feature3
```

但是要意识到，这可能会增加循环依赖的可能性，在`_init_` 模块中添加的代码将被实例化。所以，要小心使用。

4.7 使用 API

在前一节中，已经了解了包和模块都是一等公民，可以简化程序库或应用程序的使用。我们应该小心地组织它们，因为它们将一起创建一个 API。

本节将介绍一些关于解决以下问题的见解：

- 跟踪冗长；
- 构建命名空间树；
- 分解代码；
- 使用 `deprecation` 过程；
- 使用 `egg`。

4.7.1 跟踪冗长

创建程序库时，最常见的错误是“API 冗长（API verbosity）”。当一个功能对包的调用是

一组而不是一个时，将出现这一错误。

下面举一个允许执行一些代码的 `script_engine` 包的例子。

```
>>> from script_engine import make_context
>>> from script_engine import compile
>>> from script_engine import execute
>>> context = make_context({'a': 1, 'b':3})
>>> byte_code = compile('a + b')
>>> print execute(byte_code)
4
```

这个使用场景应该在一个包中的新函数之后提供，如下所示。

```
>>> from script_engine import run
>>> print run('a + b', context={'a': 1, 'b':3})
4
```

之后，低等级和高等级的函数都将可用于高等级调用和其他低等级函数的组合。



这个原则将在第 14 章中通过 Façade 设计模式来描述。



4.7.2 构建命名空间树

要组织一个应用程序的 API，一种简单的技术是通过使用场景构建一个命名空间树，并了解代码的组织方式。

我们来举个例子。一个名为 `acme` 的应用程序要提供一个知道如何创建 PDF 文件的引擎。它将基于一系列模板文件和一个 MySQL 数据库上的查询。

`acme` 应用的 3 个部分是：

- 一个 PDF 生成器；
- 一个 SQL 引擎；
- 一个模板集合。

由此，命名空间树的初稿可能是：

- `acme`
 - `pdfgen.py`
 - `class PDFGen`
 - `sqlengine.py`

- class SQLEngine
- templates.py
 - class Template

现在，在一个代码示例中尝试这个命名空间，并了解 PDF 如何从这个应用程序中创建。我们将猜测类和函数如何命名，并在类似于 acme 的功能的一个粘合程序中被调用，如下所示。

```
>>> from acme.templates import Template
>>> from acme.sqlengine import SQLEngine
>>> from acme.pdfgen import PDFGen
>>> SQL_URI = 'sqlite:///memory:'
>>> def generate_pdf(query, template_name):
...     data = SQLEngine(SQL_URI).execute(query)
...     template = Template(template_name)
...     return PDFGen().create(data, template)
```

第一个版本给了一个关于命名空间可用性的反馈，而且可以被重构，以通过 API 冗长跟踪和常识来简化。

例如，PDFGen 不需要在调用者中创建，因为任何该类的实例都可以生成任何 PDF 实例。因此，它可以保持为私有的。templates 的使用也可以通过以下的风格来简化。

```
>>> from acme import templates
>>> from acme.sqlengine import SQLEngine
>>> from acme.pdf import generate
>>> SQL_URI = 'sqlite:///memory:'
>>> def generate_pdf(query, template_name):
...     data = SQLEngine(SQL_URI).execute(query)
...     template = templates.generate(template_name)
...     return generate(data, template)
```

第二稿的命名空间将变成：

- acme
 - config.py
 - SQL_URI
 - utils.py
 - function generate_pdf
 - pdf.py
 - function generate
 - class _Generator

- `sqlengine.py`
 - `class SQLEngine`
 - `templates.py`
 - `function generate`
 - `class _Template`

所做的修改如下：

- `config.py` 中包含配置元素；
- `utils.py` 提供高等级的 API；
- `pdf.py` 提供了唯一的一个函数；
- `templates.py` 提供一个工厂。

对于每个新的使用场景，这样的结构化改变对设计一个可用的 API 而言是有帮助的。这必须在包被发布和使用之前完成。对于已经发布的包，必须设置一个启用过程，这将在本章稍后做说明。



命名空间树必须通过实际的使用场景小心设计。第 11 章中将介绍如何通过测试来创建它。

4.7.3 分解代码

小就是美，这也适用于所有级别的代码。当一个函数、类或一个代码太大时，就应该对其进行分解。

一个函数或一个方法的内容不应该超过一个屏幕，也就是大约 25~30 行，否则它将很难跟踪和理解。



关于代码复杂性的更多信息，可以参见 Eric Raymond 所著的 *Art of Unix Programming*（中译版《Unix 编程艺术》）中的相关章节。

类的方法的数量应该有一定的限制。当方法超过 10 个时，即使创建者对其也很难做出完整的描绘。一个常见的方法是分离功能并且在该类之外创建多个类。

一个模块的大小也应该有一定的限制。当它超过 500 行时，就应该被分解为多个模块。

这个工作将会影响 API，并且意味着在包的级别上需要付出额外的工作来确保代码分解

和组织的方式不会使 API 难以使用。

换句话说，API 应该总是从用户的角度来测试，以确保它可用、易于记忆和简明。

4.7.4 使用 Egg

当应用程序不断成长时，主文件夹下的包数量也可能会变得相当大。例如，像 Zope 这样的框架在根包 zope 命名空间中就有超过 50 个包。

为了避免使整个代码库都在同一个文件夹里，并能够单独发布每个包，可以使用“Python eggs” (<http://peak.telecommunity.com/DevCenter/PythonEggs>) 来解决。它们提供了一种构建“命名空间的包”的简单方法，就像 Java 中提供的 JAR 一样。

例如，如果希望将 acme.templates 作为单独的包分发，可以使用 setuptools (用于建立 Python Egg 的程序库) 来构建一个基于 egg 的包，在 acme 文件夹中的特殊文件 `__init__.py` 里添加以下内容 (<http://peak.telecommunity.com/DevCenter/setuptools#namespace-packages>)。

```
try:
    __import__('pkg_resources').declare_namespace(__name__)
except ImportError:
    from pkgutil import extend_path
    __path__ = extend_path(__path__, __name__)
```

然后，acme 文件夹中可以保存 templates 文件夹，并使其能够位于 acme.templates 命名空间之下。acme.pdf 甚至可以从独立的 acme 文件夹中被分离出来。

遵循相同的规则，来自相同组织的包可以通过 egg 收集在相同的命名空间中，即使它们互不相关。例如，所有来自 Ingeniweb 的包都是使用 iw 命名空间的，并且可以使用前缀 <http://pypi.python.org/pypi?%3Aaction=search&term=iw.&submit=search>，在 Cheeseshop 上找到。

除了命名空间之外，以 egg 形式分发应用程序也对模块化有帮助，因为可以视每个 egg 为一个独立的组件。



第 6 章将介绍如何创建、发布和部署一个基于 egg 的应用程序。

4.7.5 使用 deprecation 过程

当包已经被发布并且被第三方代码使用时，对 API 的修改就必须小心进行了。处理这种

修改最简单的方法是遵循一个 `deprecation` 过程，在此是包含两个版本的中间发行版本。

例如，如果一个类有个 `run_script` 方法被替换成简化的 `run` 命令，内建的 `DeprecationWarning` 异常可以和 `warnings` 模块一起被用在中间结果中，如下所示。

```
>>> class SomeClass(object): # 版本 1
...     def run_script(self, script, context):
...         print 'doing the work'
>>> import warnings
>>> class SomeClass(object): # 版本 1.5
...     def run_script(self, script, context):
...         warnings.warn(("'run_script' will be replaced "
...                     "by 'run' in version 2"),
...                     DeprecationWarning)
...     return self.run(script, context)
...     def run(self, script, context=None):
...         print 'doing the work'
>>> SomeClass().run_script('a script', {})
_main_:4: DeprecationWarning: 'run_script' will be replaced by 'run'
in version 2
doing the work
>>> SomeClass().run_script('a script', {})
doing the work
>>> class SomeClass(object): # version 2
...     def run(self, script, context=None):
...         print 'doing the work'
```

`warnings` 模块将在第一次调用时警告用户，并且忽略下一个调用。这个模块的另一个很好的功能是可以创建过滤器，用来管理影响应用程序的警告信息。例如，警告可以自动忽略或者变为异常，以进行强制修改（参见 <http://docs.python.org/lib/warning-filter.html>）。

4.8 有用的工具

前面的约定和方法的一部分可以使用以下的工具来控制和解决：

- `Pylint` 一个非常灵活的元代码分析器；
- `CloneDigger` 一个重复代码侦测工具。

4.8.1 Pylint

除了一些质量保证方面的度量之外，Pylint 能够检查指定的源代码是否遵循某种命名约定。它的默认设置对应于 PEP 8，一个 Pylint 脚本提供了一个 shell 报告输出。

要安装 Pylint，可以通过 easy_install 来使用 logilab.installer egg，如下所示。

```
$ easy_install logilab pylintinstaller
```

这步之后，该命令就可用了，并且可以对一个模块或使用通配符表示的多个模块运行，如下所示。

```
$ pylint bootstrap.py
No config file found, using default configuration
*****
Module bootstrap
C: 25: Invalid name "tmpeggs" (should match ([A-Z][A-Z1-9]*|(_.*_))$)
C: 27: Invalid name "ez" (should match ([A-Z][A-Z1-9]*|(_.*_))$)
W: 28: Use of the exec statement
C: 34: Invalid name "cmd" (should match ([A-Z][A-Z1-9]*|(_.*_))$)
C: 36: Invalid name "cmd" (should match ([A-Z][A-Z1-9]*|(_.*_))$)
C: 38: Invalid name "ws" (should match ([A-Z][A-Z1-9]*|(_.*_))$)
...
Global evaluation
-----
Your code has been rated at 6.25/10
```

注意，Pylint 总会在某些时候给出不好的评级或抱怨。例如，一个不被模块本身的代码使用的 import 语句在某些情况下也是很好的（使其在命名空间中可用）。

如果调用采用了混合大小写的方法命名的程序库，那么它也可能降低评级。在任何情况下，全局评估不像 C 中的“lint”那么重要。Pylint 只是一个指出潜在改进点的工具。

调优 Pylint 第一件要做的事情就是，使用 -generate-rcfile 选项在原始目录下创建一个.pylintrc 配置文件，如下所示。

```
$ pylint --generate-rcfile > ~/.pylintrc
```

在 Windows 下，“~”文件夹必须替换成用户文件夹，一般位于 Documents and Settings 文件夹中（参见 HOME 环境变量）。

配置文件中首先需要修改的是，在 REPORTS 小节中将 reports 变量设置为 no，以避免生成冗长的报告。在我们的例子中，只需要用这个工具侦测名称。当完成这个修改之后，这个

工具将只会显示警告，如下所示。

```
$ pylint bootstrap.py
*****
Module bootstrap
C: 25: Invalid name "tmpeggs" (should match ([A-Z][A-Z1-9_]*|(_.*_))$)
C: 27: Invalid name "ez" (should match ([A-Z][A-Z1-9_]*|(_.*_))$)
W: 28: Use of the exec statement
C: 34: Invalid name "cmd" (should match ([A-Z][A-Z1-9_]*|(_.*_))$)
C: 36: Invalid name "cmd" (should match ([A-Z][A-Z1-9_]*|(_.*_))$)
C: 38: Invalid name "ws" (should match ([A-Z][A-Z1-9_]*|(_.*_))$)
```

4.8.2 CloneDigger

CloneDigger (<http://clonedigger.sourceforge.net>) 是一个很好的工具，它尝试访问代码树以侦测代码中的相似之处。它基于网站上所说明的相当复杂的算法，有效地补充了 Pylint。

要安装它，可以使用 easy_install，如下所示。

```
$ easy_install Clonedigger
```

将得到一个可被用于侦测重复代码的 clonedigger 命令，命令选项可以在 <http://clonedigger.sourceforge.net/documentation.html> 上找到。

```
$ clonedigger html_report.py ast_suppliers.py
Parsing html_report.py ... done
Parsing ast_suppliers.py ... done
40 sequences
average sequence length: 3.250000
maximum sequence length: 14
Number of statements: 130
Calculating size for each statement... done
Building statement hash... done
Number of different hash values: 52
Building patterns... 66 patterns were discovered
Choosing pattern for each statement... done
Finding similar sequences of statements... 0 sequences were found
Refining candidates... 0 clones were found
Removing dominated clones... 0 clones were removed
```

output.html 是它生成的一个 HTML 输出，其中包含了 CloneDigger 的一个工作报告。

4.9 小结

本章介绍了以下内容。

- PEP 8 是命名约定的绝对参考。
- 选择名称的时候应该遵循以下几条规则：为布尔元素命名时使用“has”或“is”前缀；为序列元素命名时使用复数；避免使用通用名称；避免遮蔽现有名称，尤其是内建的名称。
- 针对参数的一组好习惯是：根据设计来构建参数；不要试图使用断言实现静态类型检查；不要误用`*args` 和 `**kw`。
- 使用 API 时的一些公共实践是：跟踪冗长；根据设计构建命名空间树；将代码分解为小块；为程序库在一个公共名空间下使用 `egg`；使用 `deprecation` 过程。
- 使用 Pylint 和 CloneDigger 控制代码。

下一章将说明如何编写一个包。

第 5 章

编写一个包

本章将聚焦于编写、发行 Python 包的可重复的过程，其目标是：

- 缩短开始真正的工作之前所需的设置时间，换句话说，就是提供样板化代码；
- 提供编写包的标准化方法；
- 简化测试驱动开发方法的使用；
- 为发行过程提供帮助。

本章内容由以下 4 个部分组成：

- 用于所有包的公用模式，描述所有 Python 包之间的相似之处，以及 distutils 和 setuptools 如何扮演核心角色；
- 产生式编程（generative programming，http://en.wikipedia.org/wiki/Generative_programming）如何通过基于模板的方法对此提供帮助；
- 包模板的创建，设置各种工作所需要的一切；
- 构建一个开发周期。

5.1 用于所有包的公共模式

在前一章中已经看到，组织一个应用程序代码最简单的方法是使用 egg 将其分解到多个包中。这使代码更加简单，而且更容易理解、维护和修改，还最大化了每个包的可复用性。它们就像组件一样工作。

对于指定公司的应用程序，可以有一组和主 egg 相结合的 egg。因此，所有的包都可以使用 egg 结构建立。

本节将介绍命名空间包(namespaced package)如何组织、发行并通过 distutils 和 setuptools

分发。

正如在前一章中所看到的那样，编写 egg 是通过对一个提供公用前缀命名空间的嵌套文件夹中的代码进行分层来完成的。例如，对于 Acme 公司，公共的命名空间可能是 acme。其结果是生成一个命名空间包。

例如，一个代码与 SQL 相关的包可以被称为 acme.sql。使用这样一个包的最佳方式是创建一个 acme.sql 文件夹，它包含 acme 文件夹，而 acme 文件夹下又包含 sql 文件夹（如图 5.1 所示）。

根目录中有一个 setup.py 脚本，它定义 distutils 模块中描述的所有元数据，并将其合并为一个标准的 setup 函数的参数。这个函数由提供大部分 egg 基础架构的第三方程序库 setuptools 扩展。

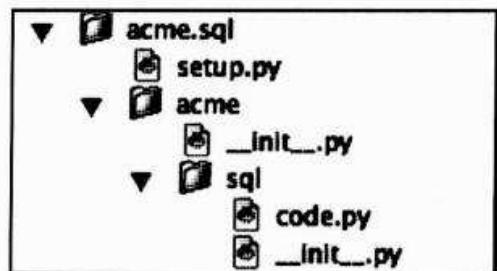


图 5.1

 distutils 和 setuptools 之间的界线正在变得模糊，也许有一天将会合并。

因此，在这个文件中至少有以下内容。

```
from setuptools import setup
setup(name='acme.sql')
```

name 给出了 egg 的全名。由此，该脚本提供多个命令，可以使用--help-commands 选项列出这些命令，如下所示。

```
$ python setup.py --help-commands
Standard commands:
  build      build everything needed to install
  ...
  install    install everything from build directory
  sdist      create a source distribution
  register   register the distribution
  bdist      create a built (binary) distribution
Extra commands:
  develop    install package in 'development mode'
  ...
  test       run unit tests after in-place build
  alias     define a shortcut
```

`bdist_egg` create an "egg" distribution

最重要的命令已经在前面的清单中列出。Standard commands（标准命令）是 distutils 提供的内建命令，而 Extra commands（附加命令）是由诸如 setuptools 这样的第三方包或任何其他定义和注册新命令的包所创建的。

1. `sdist`

`sdist` 命令是最简单的命令。它用来创建一个发行树，运行一个包所需的一切内容都将复制到这里。然后，这棵树被归档到一个或多个档案文件中（它往往只创建一个 tar 档案）。这个档案基本上是一个源树的副本。

这个命令是从目标系统独立地分发一个包的最简单方式。它将创建一个 `dist` 文件夹，其中包含可被分发的档案。使用它之前，必须传递一个附加的参数给 `setup`，以提供版本号。如果不给它提供一个 `version` 值，那它将使用 `version = 0.0.0`，如下所示。

```
from setuptools import setup
setup(name='acme.sql', version='0.1.1')
```

这个版本号在升级时十分有用。每当发行包时都将提升版本号，这样目标系统就知道它已经被修改了。

使用这个附加参数来运行 `sdist` 命令，如下所示。

```
$ python setup.py sdist
running sdist
...
creating dist
tar -cf dist/acme.sql-0.1.1.tar acme.sql-0.1.1
gzip -f9 dist/acme.sql-0.1.1.tar
removing 'acme.sql-0.1.1' (and everything under it)
$ ls dist/
acme.sql-0.1.1.tar.gz
```



在 Windows 下，档案将是一个 ZIP 文件。

版本用来标记档案名称，这个档案可以被分发，并且安装到任何拥有 Python 的系统之上。在 `sdist` 分发中，如果包含有 C 程序库或扩展，目标系统将负责编译它们。这在基于 Linux 和 Mac OS 的系统上是很常见的，因为它们都提供编译器。但是，这在 Windows 下并不常见。这就是当一个包打算在多个平台下运行时，应该总是和一个预编译分发版本一起

分发的原因。

2. MANIFEST.in 文件

使用 `sdist` 构建一个分发版本时，`distutils` 将浏览包的目录，查找包含在档案中的文件。

`distutils` 将包含：

- 所有 `py_modules`、`packages` 和 `scripts` 选项隐含的 Python 源文件；
- 所有在 `ext_modules` 选项中列出的 C 源文件；
- 符合 `test/test*.py` 模式的文件；
- `README`、`README.txt`、`setup.py` 和 `setup.cfg` 文件。

此外，如果包是由 Subversion 或 CVS 管理，那么 `sdist` 将浏览.svn 之类的文件夹以查找其包含的文件。`sdist` 将创建一个列出所有文件的 `MANIFEST` 文件，并将它们包含到档案文件中。

假设不使用这些版本控制系统，并且需要包含更多的文件，可以在与 `setup.py` 相同的目录下定义一个名为 `MANIFEST.in` 的 `MANIFEST` 文件模板，然后在这里指出 `sdist` 包含的文件。

这个模板中的每一行都将定义一个包含或排除规则，示例如下。

```
include HISTORY.txt
include README.txt
include CHANGES.txt
include CONTRIBUTORS.txt
include LICENSE
recursive-include *.txt *.py
```

命令的完整列表可在 <http://docs.python.org/dist/> 中找到。

```
sdist-cmd.html #sdist-cmd.
```

3. build 和 bdist

为了能够分发预编译的分发版本，`distutils` 提供了 `build` 命令，它分 4 个步骤来编译包：

- `build_py` 通过字节编译 (byte-compiling) 构建纯 Python 模块，并将其复制到构建文件夹中；
- `build_clib` 当包含有 C 程序库时，使用 Python 编译器编译并在构建文件夹中创建一个静态程序库；
- `build_ext` 编译 C 扩展，并将结果放到类似 `build_clib` 的构建文件夹中；
- `build_scripts` 编译标记为脚本的模块，当第一行被设置 (!#) 时修改解释程序路径并修复文件模式使其可执行。

每个步骤都是可以被单独调用的命令。编译过程的结果是一个包含安装包所需所有内容的构建文件夹。distutil 包中还没有提供交叉编译选项，这意味着执行该命令的结果只是针对构建时所用操作系统的。



最近有些人在 Python tracker 中提出了一些补丁，使 distutils 能够交叉编译 C 编写的部分。所以，将来也许能够用到这个功能。

当必须创建一些 C 扩展时，构建过程将使用系统编译器和 Python 头文件（Python.h）。这个引用文件在从源文件中编译出 Python 时就是可用的了。对于打包的分发版本，一个名为 python-dev 的附加包通常会包含它，所以这个包也必须安装。

使用的 C 编译器是当前系统中的编译器。对于基于 Linux 的系统或 Mac OS X 而言是 gcc，对于 Windows 而言可以使用 Microsoft Visual C++（有可用的免费命令行版本），也可以使用开源项目 MinGW。第 1 章中就已经讲解过，可以在 distutils 中进行相应的配置。

将使用 bdist 命令来创建一个二进制分发版本。它调用 build 和所有相关的命令，然后和 sdist 一样，创建一个档案文件。

下面在 Mac OS X 下为 acme.sql 创建一个二进制分发版本。

```
$ python setup.py bdist
running bdist
running bdist_dumb
running build
...
running install_scripts
tar -cf dist/acme.sql-0.1.1.macosx-10.3-fat.tar .
gzip -f9 acme.sql-0.1.1.macosx-10.3-fat.tar
removing 'build/bdist.macosx-10.3-fat/dumb' (and everything under it)
$ ls dist/
acme.sql-0.1.1.macosx-10.3-fat.tar.gz acme.sql-0.1.1.tar.gz
```

注意，新创建的档案文件名中包含了系统名及分发版本的名称（Mac OS X 10.3）。

在 Windows 下调用相同命令，也能创建一个特定的分发档案文件，如下所示。

```
C:\acme.sql> python.exe setup.py bdist
...
C:\acme.sql> dir dist
25/02/2008 08:18 <DIR> .
```

```
25/02/2008 08:18 <DIR> ..
25/02/2008 08:24           16 055 acme.sql-0.1.win32.zip
      1 File(s)        16 055 bytes
      2 Dir(s)   22 239 752 192 bytes free
```

如果这个包里包含 C 代码，那么除了提供源代码分发版本之外，应发行尽可能多的不同的二进制分发版本。至少，对于没有安装 C 编译器的人来说，一个 Windows 二进制分发版本就是很重要的。

在一个二进制发行版本中，包含一棵可以直接复制到 Python 树中的树。它主要包含复制到 Python 的 site-packages 文件夹中的一个文件夹。

4. bdist_egg

`bdist_egg` 命令是 `setuptools` 额外提供的一个命令。它基本上和 `bdist` 类似，用来创建一个二进制分发版本，不过它具有一棵与在源代码分发版本中相当的树。换句话说，这个档案文件可以被下载、解压，然后通过将文件夹添加到 Python 的搜索路径（`sys.path`）来使用。

近来，这种分发模式经常用来替代基于 `bdist` 生成的模式。

5. install

使用 `install` 命令可以将包安装到 Python 中。如果以前没有编译，那么它将尝试编译包然后将结果注入到 Python 树中。如果提供源代码分发版本，那么它可以被解压到一个临时文件夹中，然后使用这个命令安装。`install` 命令还将安装在 `install_requires` 元数据中定义的相关模块。

这是通过查看 Python 包索引（PyPI）来完成的。例如，为了和 `acme.sql` 一起安装 `pysqlite` 和 `SQLAlchemy`，`setup` 调用将被修改为：

```
from setuptools import setup
setup(name='acme.sql', version='0.1.1',
      install_requires=['pysqlite', 'SQLAlchemy'])
```

当运行该命令时，两个相关的模块都将被安装。

6. 如何卸载一个包

用来卸载一个之前安装的包的命令，在 `setup.py` 中找不到。这个功能早先已经有人提议过了。因为安装程序可能修改系统其他元素使用的文件，所以这并不是很简单的。

最佳的方法应该是对所有被修改的元素建立一个快照，并且对自己新创建的文件和目录做一个记录。

`install` 中有一个 `record` 选项，用来将所有创建的文件记录到一个文本文件中，如下所示。

```
$ python setup.py install --record installation.txt
running install
...
writing list of installed files to 'installation.txt'
```

它不会对任何现有的文件创建备份，所以删除这些文件将会破坏系统。对这个问题，有许多特定于平台的解决方案。例如，distutils 允许以 RPM 包格式分发。但是现在还没有处理这一任务的通用方法。

目前删除一个包的最简单方法是删除该包所创建的文件，然后删除在 sitepackages 文件夹中 easy-install.pth 文件中列举的所有引用。

7. develop

setuptools 添加了一个有用的处理包的命令。develop 命令编译并且在适当的位置安装包，然后添加一个简单的链接到 Python site-packages 文件夹中。这使用户能够使用该代码的本地副本工作，即使它可在 Python 的 site-packages 文件夹中获得。在下一章中，在创建基于 egg 的应用程序时会发现，这是一个很好的特性。所有被创建的包都可以使用 develop 命令链接到解释程序。

当以这种形式安装一个包时，和常规的安装会有些不同，可以使用-u 选项显式删除，如下所示。

```
$ sudo python setup.py develop
running develop
...
Adding iw.recipe.fss 0.1.3dev-r7606 to easy-install.pth file
Installed /Users/repos/ingeniweb.sourceforge.net/iw.recipe.fss/trunk
Processing dependencies ...
$ sudo python setup.py develop -u
running develop ...
Removing
...
Removing iw.recipe.fss 0.1.3dev-r7606 from easy-install.pth file
```

注意，使用 develop 安装的包，总是要比用其他方式安装的包更好些。

8. test

另一个有用的命令是 test，它提供了一种执行包中所有测试的方法。它将扫描文件夹，并且收集寻找到的测试套件。这个测试运行程序尝试收集包中的测试，但是这相当有局限性。

一种更好的方法是与诸如 `zope.testing` 或 `Nose` 这样的提供更多选项的外部测试运行程序挂钩起来。

为了透明地将 `Nose` 与测试命令挂钩，可以将 `test_suite` 元数据设置为 “`'nose.collector'`”，并将 `Nose` 添加到 `test_requires` 列表中，如下所示。

```
setup(
    ...
    test_suite='nose.collector',
    test_requires=['Nose'],
    ...
)
```



第 11 章中将介绍一些测试运行程序，并说明 `Nose` 的使用方法。

9. register 和 upload

要分发一个包，有两个可用的命令：

- `register` 它将把所有的元数据上传到一个服务器；
- `upload` 它将前面建立在 `dist` 文件夹中的所有档案上传到服务器。

主 PyPI 服务器——之前称为 Cheeseshop（奶酪店），位于 <http://pypi.python.org/pypi>，其中包含超过 3000 个来自开发社区的包。这是 `distutils` 包所使用的默认服务器，对 `register` 命令的第一个调用将在主目录中生成一个 `C` 文件。

因为 PyPI 服务器要进行身份验证，所以当修改一个包时，将要求创建一个用户。这可以在提示符下完成，如下所示。

```
$ python setup.py register
running register
...
We need to know who you are, so please choose either:
1. use your existing login,
2. register as a new user,
3. have the server generate a new password for you (and email it to
you), or
4. quit
Your selection [default 1]:
```

现在，主目录中将出现一个名为 `.pypirc` 文件，它包含了必须输入的用户名和密码。这将

在每次 register 和 upload 被调用时使用，如下所示。

```
[server-index]
username: tarek
password: secret
```

Windows 上有一个与 Python 2.4 和 2.5 有关的权限。这个主目录通过 distutils 无法找到，除非添加一个 HOME 环境变量。但是，这在 2.6 版本中已经修复了。如果要添加它，可以使用第 1 章中修改 PATH 命令时所描述的方法。然后为用户添加 HOME 变量，指向 os.path.expanduser('~') 返回的目录。

当指定了 download_url 元数据或 url，并且指定的是一个有效的 URL 时，PyPI 服务器将使其对在项目网页上的用户也可用。

使用 upload 命令将直接使档案文件在 PyPI 上可用，所以 download_url 可以省略，如图 5.2 所示。

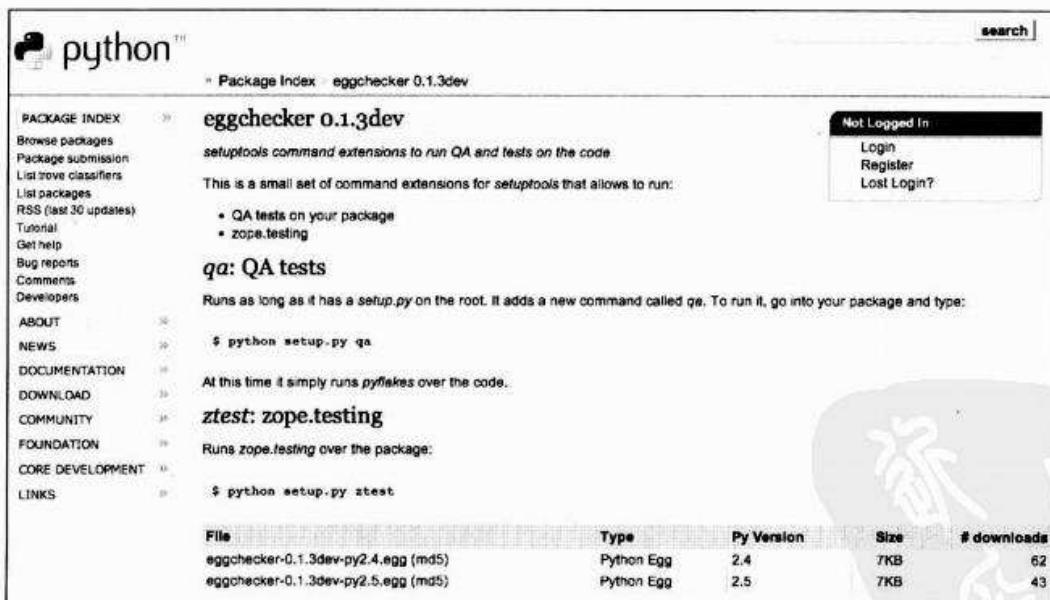


图 5.2

Distutils 定义了一个 Trove 分类(参见 PEP 301, <http://www.python.org/dev/peps/pep-0301/#distutils-trove-classification>)来对包进行分类(如定义在 Sourceforge 上的包)。trove 是可以在 http://pypi.python.org/pypi?%3Aaction=list_classifiers 上找到的静态分类，它也不断地被新来者扩充。

每一行都是由“::”分隔的级别，如下所示。

```

...
Topic :: Terminals
Topic :: Terminals :: Serial
Topic :: Terminals :: Telnet
Topic :: Terminals :: Terminal Emulators/X Terminals
Topic :: Text Editors Topic :: Text Editors :: Documentation
Topic :: Text Editors :: Emacs
...

```

一个包可以被分在多个类别中，这些类别可以在分类符 meta-data 中列出。一个处理（例如）低级 Python 代码的 GPL 包可以使用如下所示的类别。

```

Programming Language :: Python
Topic :: Software Development :: Libraries :: Python Modules
License :: OSI Approved :: GNU General Public License (GPL)

```

10. Python 2.6 中.pypirc 的格式

.pypirc 文件在 Python 2.6 下已经有了一些改进，多个用户及其密码可以同多个类似 PyPI 的服务器一起管理。一个 Python 2.6 配置文件看起来如下所示。

```

[distutils]
index-servers =
    pypi
    alternative-server
    alternative-account-on-pypi

[pypi]
username:tarek
password:secret

[alternative-server]
username:tarek
password:secret
repository:http://example.com/pypi

```

register 和 upload 命令可以借助-r 选项协助选取服务器，可以使用仓库的完整 URL 或小节段，如下所示。

```

# 上传到 http://example.com/pypi
$ python setup.py sdist upload -r alternative-server
# 用默认账户（从 pypi 中获取的）注册

```

```
$ python setup.py register
# 注册到 http://example.com
$ python setup.py register -r http://example.com/pypi
```

通过该特性可以和 PyPI 之外的服务器进行交互。当要处理的包大部分都不是发布在 PyPI 上时，运行自己的类 PyPI 服务器是个良好的习惯。例如，Plone Software Center（参见 <http://plone.org/products/plonesoftwarecenter>）就能够用来部署一个与 distutils upload 和 register 命令交互的 Web 服务器。

11. 创建一个新命令

distutils 允许创建新的命令，如<http://docs.python.org/dist/node84.html>中描述的那样。一个新的命令可以使用一个入口（entry point）来注册。这是由 setuptools 引入的，是一种将包定义为插件的简单方法。

入口点是 setuptools 中的一个命名链接，它指向通过某些 API 可用的类或函数。任何应用程序都能够扫描所有已注册的包，并且将其链接的代码作为一个插件使用。

为了链接新的命令，可以在 setup 调用中使用 entry_points 元数据，如下所示。

```
setup(name="my.command",
      entry_points="""
      [distutils.commands]
      my_command = my.command.module.Class
      """)
```

所有命名链接都被集中在已命名的小节中。当 distutils 载入时，它将扫描登记在 distutils.commands 之下所有的链接。

许多提供扩展性的 Python 应用程序都使用了这一机制。

12 setup.py 使用小结

setup.py 采用的 3 个主要操作：

- 创建一个包；
- 安装这个包，可能在开发模式下；
- 注册并上传到 PyPI 服务器。

所有命令都可以被组合到相同的调用中，以下是一些典型的使用模式。

```
# 使用 PyPI 注册这个包，创建一个源代码发布和一个 egg 发布，然后上传
$ python setup.py register sdist bdist_egg upload
# 就地安装，以便开发
$ python setup.py develop
```

```
# 安装它
$ python setup.py install
```

13. alias 命令

为了简化命令行的工作，`setuptools` 引入了一个新命令 `alias`。在 `setup.cfg` 文件中，它用来为指定的命令组合创建一个别名。例如，可以创建一个 `release` 命令来执行将一个源代码分发版本和一个二进制分发版本上传到 PyPI 服务器上所需的所有操作，如下所示。

```
$ python setup.py alias release register sdist bdist_egg upload
running alias
Writing setup.cfg
$ python setup.py release
...
```

14. 其他重要的元数据

除了要分发的包的名称和版本之外，`setup` 可能接受的最重要的参数包括：

- `description` 包的简单描述；
- `long_description` 包的完整的描述，可以是 reStructuredText 的形式；
- `keywords` 定义包的关键字列表；
- `author` 作者的姓名或组织；
- `author_email` 作者的邮件地址；
- `url` 项目的 URL；
- `license` 许可证（GPL、LGPL 等）；
- `packages` 包中所有名称的列表，`setuptools` 提供了一个名为 `find_packages` 的小函数来计算它；
- `namespace_packages` 命名空间包的一个列表。

针对 `acme.sql` 包而言，完整的 `setup.py` 将如下所示。

```
import os
from setuptools import setup, find_packages
version = '0.1.0'
README = os.path.join(os.path.dirname(__file__), 'README.txt')
long_description = open(README).read() + '\n\n'
setup(name='acme.sql',
      version=version,
      description=("A package that deals with SQL, "
                  "from ACME inc"),
```

```
long_description=long_description,
classifiers=[
    "Programming Language :: Python",
    ("Topic :: Software Development :: Libraries ::",
     "Python Modules"),
    ],
keywords='acme sql',
author='Tarek',
author_email='tarek@ziade.org',
url='http://ziade.org',
license='GPL',
packages=find_packages(),
namespace_packages=['acme'],
install_requires=['pysqlite','SQLAlchemy']
)
```



应常备以下两个综合性指南。

- 在 <http://docs.python.org/dist/dist.html> 中可以找到的 distutils 指南；
- 在 <http://peak.telecommunity.com/DevCenter/setuptools> 中可以找到的 setuptools 指南。

5.2 基于模板的方法

acme.sql 中的样板化代码由一个创建根目录中少数文件命名空间的目录树组成。为了使所有包遵循相同的结构，可以通过一个代码生成工具来抽取和提供通用的代码模板。这个方法被称为产生式编程，它在组织级非常有用。它标准化了代码编写的方式，并使开发人员更多产，因为他们可以关注于真正需要创建的代码。这种方法也是为包准备少数被多个包公用的部分（如复杂的测试配置）的一个好机会。

在 Python 社区中，有许多可用的产生式工具，不过最常用的是 Python Paste (<http://pythonpaste.org>)。

5.2.1 Python Paste

Python Paste 项目是诸如 Pylons (<http://pylonshq.com>) 这样的框架取得成功的一部分原因。

开发人员被那些让他们在几分钟内创建出应用程序框架的大量模板集深深地触动。

从官方的教程中可以了解到，这是创建一个 web 应用程序并运行它的 3 个命令行，如下所示。

```
$ paster create -t pylons helloworld
$ cd helloworld
$ paster serve --reload development.ini
```

Plone 和 Zope 社区也遵循这一哲学，现在也为生成框架提供了 Python Paste 模板，ZopeSkel (<http://pypi.python.org/pypi/ZopeSkel>) 就是其中之一。

Python Paste 中包含了多个工具，我们所感兴趣的模板引擎是 PasteScript，可以使用 easy_install 来安装它。它将从 Paste 项目中得到所有相关的模块，如下所示。

```
$ easy_install PasteScript
Searching for PasteScript
Reading http://pypi.python.org/simple/PasteScript/
Reading http://pythonpaste.org/script/
Best match: PasteScript 1.6.2
Downloading ...
Processing dependencies for PasteScript
Searching for PasteDeploy
...
Searching for Paste>=1.3
...
Finished processing dependencies for PasteScript
```

paster 命令将可用并带有几个默认的模板，可以使用 create 命令的 list-templates 选项将其列出，如下所示。

```
$ paster create --list-templates
Available templates:
basic_package: A basic setuptools-enabled package
paste_deploy: A web application deployed through paste.deploy
```

basic_package 几乎是 acme.sql 用 setup.py 文件创建一个命名空间包所需要的内容。运行时，该命令行将询问几个问题，相应的回答将被用来填充模板，如下所示。

```
$ paster create -t basic_package mypackage
Selected and implied templates:
```

```
PasteScript#basic_package A basic setuptools-enabled package
...
Enter version (Version (like 0.1)) ['']: 0.1
Enter description ['']: My package
Enter long_description ['']: this is the package
Enter keywords ['']: package is mine
Enter author (Author name) ['']: Tarek
Enter author_email (Author email) ['']: tarek@ziade.org
Enter url (URL of homepage) ['']: http://ziade.org
Enter license_name (License name) ['']: GPL
Enter zip_safe [False]:
Creating template basic_package
...
```

其执行结果是一个有效的，与安装工具兼容的单级结构，如下所示。

```
$ find mypackage
mypackage
mypackage/mypackage
mypackage/mypackage/__init__.py
mypackage/setup.cfg
mypackage/setup.py
```

5.2.2 创建模板

Python Paste 可称为剪贴本（paster），举例来说，它可以使用 Cheetah 模板引擎 (<http://cheetahtemplate.org>)，并提供给它用户的输入信息。

为剪贴本创建一个新的模板，需要提供 3 个要素：

- 一个从 `paste.script.templates.Template` 中继承的类；
- 包含文件夹和文件（Cheetah 模板或静态文件）的结构；
- 一个指向 `paste.paster_create_template` 的 `setuptools` 入口点，用来注册类。

5.3 创建包模板

让我们来创建用于 `acme.sql` 包的模板。

本书中创建的所有模板包括收集在 PyPI 中的 `pyp.skels` 里的所有包，可以方便地使用。所以，如果不希望从头创建自己的模板，可以安装它，命令如下。

```
$ easy_install pyp.skels
```

本节将一步步地讲解 `pyp.skels` 的创建方法。

要创建 package 模板，第一件事就是创建新包的结构，命令如下。

```
$ mkdir -p pyp.skels/pyp/skels
$ find pyp.skels
pyp.skels
pyp.skels/pyp
pyp.skels/pyp/skels
```

然后，将创建一个带有以下代码的 `__init__.py` 文件，并将其放在 `pyp` 文件夹中。它将告知 `distutils` 创建一个命名空间包，如下所示。

```
try:
    __import__('pkg_resources').declare_namespace(__name__)
except ImportError:
    from pkgutil import extend_path
    __path__ = extend_path(__path__, __name__)
```

接下来，使用正确的元数据在根目录（`path_to_pyp_package/pyp.skels/__init__.py`）中创建一个 `setup.py` 文件。正确的代码如下所示。

```
from setuptools import setup, find_packages
version = '0.1.0'
classifiers = [
    "Programming Language :: Python",
    ("Topic :: Software Development :: "
     "Libraries :: Python Modules")]
setup(name='pyp.skels',
      version=version,
      description=("PasteScript templates for the Expert "
                  "Python programming Book."),
      classifiers=classifiers,
      keywords='paste templates',
```

```
author='Tarek Ziade',
author_email='tarek@ziade.org',
url='http://atomisator.ziade.org',
license='GPL',
packages=find_packages(exclude=['ez_setup']),
namespace_packages=['pbp'],
include_package_data=True,
install_requires=['setuptools',
                  'PasteScript'],
entry_points="""
# -*- Entry points: -*-
[paste.paster_create_template]
pbp_package = pbp.skels.package:Package
""")
```

该入口点添加了一个在剪贴本中可用的新模板。

下一步是在 pbp/skels 文件夹中的 package 模块里写入 Pacakge 类，如下所示。

```
from paste.script.templates import var
from paste.script.templates import Template
class Package(Template):
    """Package template"""
    _template_dir = 'tmpl/package'
    summary = "A namespaced package with a test environment"
    use_cheetah = True
    vars = [
        var('namespace_package', 'Namespace package',
            default='pbp'),
        var('package', 'The package contained',
            default='example'),
        var('version', 'Version', default='0.1.0'),
        var('description',
            'One-line description of the package'),
        var('author', 'Author name'),
        var('author_email', 'Author email'),
        var('keywords', 'Space-separated keywords/tags'),
        var('url', 'URL of homepage'),
        var('license_name', 'License name', default='GPL')]
```

```

    ]
def check_vars(self, vars, command):
    if not command.options.no_interactive and \
       not hasattr(command, '_deleted_once'):
        del vars['package']
        command._deleted_once = True
    return Template.check_vars(self, vars, command)

```

该类定义了：

- 包含模板结构的文件夹 (_template_dir);
- 将出现在剪贴本中的模板的摘要;
- 表明 Cheetah 是否用于模板结构的标志;
- 一个变量列表，每个变量由名称、标签和默认值（如果需要）组成，这些变量由剪贴本用于询问用户，提示输入用户所需要的数值;
- 在提示输入时所确定的包变量的 check_vars 方法。

最后一步是创建 tmpl/package 目录，并将 acme.sql 目录的内容复制过来。所有包含被修改值（如命名空间）的文件必须带有_tmpl 后缀。这些值将被替换为 \${variable}，其中 variable 是 Package 类中列出的变量名称。例如，setup.py 文件将变成 setup.py_tmpl，并且其内容变为如下所示。

```

from setuptools import setup, find_packages
import os
version = ${repr($version) or "0.0"}
long_description = open("README.txt").read()
classifiers = [
    "Programming Language :: Python",
    ("Topic :: Software Development :: "
     "Libraries :: Python Modules")]
setup(name=${repr($project)},
      version=version,
      description=${repr($description) or $empty},
      long_description=long_description,
      classifiers=classifiers,
      keywords=${repr($keywords) or $empty},
      author=${repr($author) or $empty},
      author_email=${repr($author_email) or $empty},
      url=${repr($url) or $empty},

```

```
license=${repr($license_name) or $empty},  
packages=find_packages(exclude=['ez_setup']),  
namespace_packages=[${repr($namespace_package)}],  
include_package_data=True,  
install_requires=[  
    'setuptools',  
    # -*- Extra requirements: -*-  
],  
test_suite='nose.collector',  
test_requires=['Nose'],  
entry_points="""  
# -*- Entry points: -*-  
""",  
)
```

`repr` 函数将告知 Cheetah 在字符串值的外面添加一个引号。

可以对 `acme.sql` 中的所有文件使用相同的方法创建一个模板。例如，`README.txt` 文件将被复制为 `README.txt_tmpl`，然后所有指向 `acme.sql` 的引用都被替换成 `Package` 类中 `var` 列表里定义的值。

例如，可以使用以下命令获得完整的包名称。

```
 ${namespace_package}.${package}
```

最后，针对文件夹名称而使用的变量值，必须加上“+”前缀和后缀。例如，命名空间包文件夹名称将被命名为`+namespace_package+`，包文件夹则被命名为`+package+`。

在生成 `acme.sql` 之后，`pbp.skeles` 的最终结构将类似于如下所示。

```
$ cd pbp.skeles  
$ find .  
setup.py  
pbp  
pbp/__init__.py  
pbp/skeles  
pbp/skeles/__init__.py  
pbp/skeles/package.py  
pbp/skeles/tmp1  
pbp/skeles/tmp1/package  
pbp/skeles/tmp1/package/README.txt_tmpl  
pbp/skeles/tmp1/package/setup.py_tmpl
```

```

pbp/skels/tmp1/package/+namespace_package+
pbp/skels/tmp1/package/+namespace_package+/_init__.py_tmpl
pbp/skels/tmp1/package/+namespace_package+/_package+
pbp/skels/tmp1/package/+namespace_package+/_package+/_init__.py
...

```

现在，该包可以使用 `develop` 命令符号连接到 Python 的 `site-packages` 目录，并且可用于剪贴本，如下所示。

```

$ python setup.py develop
...
Finished processing dependencies for pbp.skels==0.1.0dev

```

运行了 `develop` 命令之后，应该查找在 `paster` 中列出的模板，命令如下。

```

$ paster create --list-templates
Available templates:
  basic_package: A basic setuptools-enabled package
  pbp_package:   A namespaced package with a test environment
  paste_deploy:  A web application ... paste.deploy
$ paster create -t pbp_package trying.it
Selected and implied templates:
  pbp.skels#package A namespaced package with a test environment
Variables:
  egg:      trying.it
  package:  tryingit
  project:  trying.it
Enter namespace_package (Namespace package) ['pbp']: trying
Enter package (The package contained) ['example']: it
...
Creating template package
...

```

在此生成的树，之后将包含可以立刻投入工作的结构。

5.4 开发周期

包的开发周期由多个迭代组成。每个迭代中，代码将从初始状态转化到新的状态。这个

阶段通常会持续几周，最后得到一个发行版本。对于很简单的小型包不会发生这种情况，但是对于所有包含足够多模块的包都值得这么做。

在迭代结束时，将通过之前看过的命令创建一个发行版本。这时，包从开发状态转移到一个可发行状态，交付的代码可以被看作一个官方发行版本。

然后，一个新的循环将从包的一个增量版本开始。

1. 应该使用什么版本号

对于包版本号的增量没有固定的约定，当开发人员感觉软件已经有了很大的发展时，他们通常会跳到更高的版本号，而不一定是和前一个版本保持连续。

大部分软件通常会从一个很小的数值开始，并且会使用2个或3个数字。当他们尝试完成一个版本时，有时会添加一个字母，例如，用 `rc` 后缀来表示它是一个预览版本（`release candidate`）。以下是一些处于测试阶段的、修改了一些问题的版本：

- 0.1、0.2、0.3
- 0.1.0、0.1.1、0.1.2a、0.1.2b
- 0.1、0.2rc1、0.2rc2

可以自己决定版本号的机制，只要一直沿用即可。在公司中，通常有所有应用程序都要遵循的标准，但是开放源码应用程序有自己的约定。

唯一应该被应用的规则是确保数字的数量始终相同，并且避免使用“-”符号，因为它被许多从包名称中提取版本号的工具用作一个分隔符。

例如，以下版本号是应该避免的：

- 0.1、0.1.1-alpha、0.1.1-b、0.2
- 0.1、0.1-a、0.1-b

2. 每晚构建

如果包在迭代期间也仍然可以发行，那么可以建立开发版本，这也被称为每晚构建版本。这种连续的发行过程使开发人员能得到连续的反馈，并且节约 `beta` 测试的一些工作。例如，他们不需要从版本仓库中获得代码，并且可以和常规版本一样安装开发版本。

为了区分开发版本和常规的版本，用户必须在版本号后附加 `dev` 后缀。例如，如果 0.1.2 版本正在被开发并且还没有发行，将被称为 0.1.2dev 版本。

`distutils` 通过在 `setup.cfg` 中提供一种方法来做标志，那就是添加一个小节来告知 `build` 命令关于开发的状态，如下所示。

```
[egg_info]
tag_build = dev
```

这将在版本前自动添加 `dev` 前缀，如下所示。

```
$ python setup.py bdist_egg
running bdist_egg
running egg_info
...
creating 'dist/iw.selenium-0.1.0dev-py2.4.egg'
```

当包存在于 Subversion 仓库中时，另一个有用的标签可能就是修订号。它可以使用 tag_svn_revision 标志来添加，如下所示。

```
[egg_info]
tag_build = dev
tag_svn_revision = true
```

在这种情况下，修订号将出现在版本号中。

```
$ python setup.py bdist_egg
running bdist_egg
running egg_info
...
creating 'dist/iw.selenium-0.1.0dev_r38360-py2.4.egg'
```

最简单的方法是始终将这个文件保持在主干中，并且在建立常规发行版本之前删除它。换句话说，Subversion 的发行过程可以是：

- 建立主干的一个标签副本；
- 检查标签分支；
- 从这个分支中删除 setup.cfg（或 egg_info 专用的部分）并且提交修改；
- 由此构建发行版本；
- 提升主干中的版本号。

这看起来和下面的过程一样。

```
$ svn cp http://example.com/my.package/trunk http://example.com/
my.package/tags/0.1
$ svn co http://example.com/my.package/tags/0.1 0.1
$ cd 0.1
$ svn rm setup.cfg
$ svn ci -m "removing the dev flag"
$ python setup.py register sdist bdist_egg upload
```



第8章将介绍诸如Subversion之类的版本控制系统是什么，以及它们的工作机制。

5.5 小 结

本章介绍了以下内容。

- 如何创建一个命名空间包；
- setup.py的主要任务，以及如何使用它建立和发行包；
- 基于模板生成包框架的方法；
- 剪贴本的工作原理以及创建包框架的方法；
- 如何发行包及提供每夜构建版本。

下一章将关注于相同主题，不过将从应用程序级别上考虑。



第 6 章

编写一个应用程序

在前一章中，我们已经了解到一个编写包以及在命名空间中集中代码的可重复方法。可以通过集中一系列的包，并且编写一个将所有这一切联系在一起的包来使它们交互，从而编写出一个 Python 应用程序。

本章将通过一个小的案例来示范如何构建、发行以及分发这样的一个应用程序。

6.1 Atomisator 概述

接下来将实现一个名为 Atomisator 的应用程序。

Atomisator 是一个命令行工具，它能够生成由多个新闻 feed 组合而成的 RSS XML 文件，如下所示。

```
$ atomisator
Reading source http://feeds.feedburner.com/dirtsimple Phillip Eby
10 entries read.
Reading source http://blog.ianbicking.org/feed/ Ian Bicking
10 entries read.
20 total.
Writing feed in atomisator.xml
Feed ready.
```

当调用这个工具时，将根据配置文件中列出的所有数据源从 Web 中读取数据，并将其保存到一个数据库中，然后从数据库生成一个具有最新条目的 XML 文件。除了在数据库中保存所有读取的数据而不进行实时合并之外，这个程序与 Planet (<http://www.planetplanet.org>)

很相似。

这个工具还可以对条目应用智能过滤器。例如，每读取一个条目时，它可以与现有的条目进行比较以确认没有重复。本章中将会提供这个应用程序的一个轻型版本，以便了解它的构建方法。

 本章将提供一个简化的实现，它和实际的 Atomisator 项目并不等价。
如果希望获得完整的版本，请查看该项目主页 <http://atomisator.ziade.org>.

6.2 整体描述

我们要做的第一件事，就是列出组成这个应用程序的所有包。Atomisator 也可以写在单一的包中，但是考虑到可维护性，还是将其分为可以独立演化的不同部件更好一些。

应用前一章中解释过的规则，Atomisator 可以被分割到 4 个包中，如下所示。

- atomisator.parser 一个知道如何读取 feed 并返回一个条目列表的 feed 解析器。
- atomisator.db 负责对存储条目的数据库进行读写访问的包。
- atomisator.feed 一个知道如何使用来自数据库的条目构建兼容 RSS 2.0 的 XML 文件的包。
- atomisator.main 主程序包，它将使用一个配置文件，提供 3 个命令行工具：
 - load_feeds 从各种数据源中读取数据；
 - generate_feed 构建 XML 文件；
 - atomisator 在一个调用中集合前面两个命令。

包之间的交互如图 6.1 所示。

- (1) 用户通过命令行调用 atomisator.main，要求生成 feed。
- (2) atomisator.main 读取配置文件，列出所有要读取的数据源及数据库配置。
- (3) atomisator.main 要求 atomisator.parser 从各种数据源读取并返回条目。
- (4) atomisator.parser 读取 feed，并以简单的数据结构返回它们。
- (5) ~ (6) atomisator.main 通过 atomisator.db 更新数据库，它将执行一个智能过滤以避免重复添加条目。
- (7) atomisator.main 要求 atomisator.feed 基于数据库生成一个 feed。
- (8) ~ (9) atomisator.feed 通过 atomisator.db 读取数据库并创建一个文件。

这个过程中，包之间的相关性定义如图 6.2 所示。

atomisator.parser 和 atomisator.db 是独立的包，应该首先编写。接下来应该编写 atomisator.feed，

紧接着是 main 包，它负责建立所有的交互。

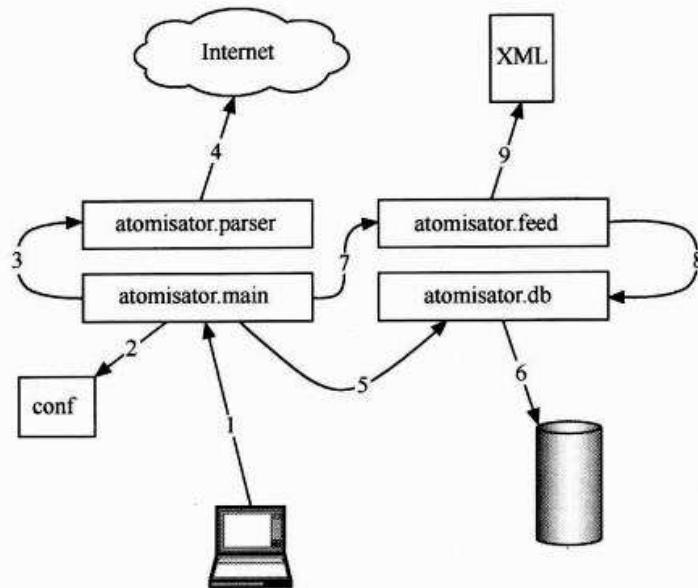


图 6.1

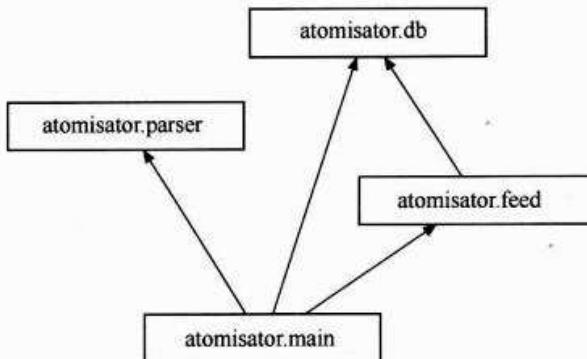


图 6.2

但是第一步，是为 Atomisator 建立一个工作环境。

6.3 工作环境

应用程序中的某些包依赖于其他包，这些相关性可以在 `install_requires` 元数据中定义。在此，调用 `python setup.py develop`，读取并安装代码工作所需的所有依赖模型。但是这要求，这些依赖模型在 PyPI 上的可用的 egg，或者已经安装在同一个 Python 安装中。现在不是这种情况，因为将要创建的用于 Atomisator 应用程序的 `atomisator.*` 包都将被一起构建，目前也尚未发布。

当然，通过 `develop` 命令安装，就能够以正确的依赖顺序完成所有包的安装，但是这将会污染 Python 安装，如果有些依赖性与已经安装的一些包有冲突，还会使得整个系统难以跟踪。

因而，应用程序级别的工作环境应该能够将用于应用程序的所有依赖模型隔离开。

`virtualenv` (<http://pypi.python.org/pypi/virtualenv>) 项目提供了一个很好的解决方案，它能在已有的 Python 安装之上创建一个新的、独立的 Python 解释程序。

其结果是得到一个本地执行环境，能够自由地安装和开发程序库，以构建出专有的环境，如下所示。

```
mkdir my_env
cd my_env/
$ easy_install -U virtualenv
Searching for virtualenv
Reading http://pypi.python.org/simple/virtualenv/
Best match: virtualenv 1.0
Processing virtualenv-1.0-py2.5.egg
Adding virtualenv 1.0 to easy-install.pth file
...
Finished processing dependencies for virtualenv
$ virtualenv --no-site-packages .
New python executable in ./bin/python
Installing setuptools.....done.
$ ls bin/
activate      easy_install      easy_install-2.5
python        python2.5
```

如果使用的是 Windows 平台，那么这些脚本将生成在 Scripts 目录下，本节中的所有实例都是如此。

`virtualenv` 命令将生成一个带有新的独立解释程序的文件夹，以及一个 `easy_install` 脚本和一个 `activate` 脚本。最后一个脚本只是一个提供方便的脚本，使得能够切换环境变量以使独立的 Python 在系统范围内被调用。`--no-site-packages` 选项可以用来切断与主 Python 中安装的包之间的依赖性。这个选项在需要一个 Python 裸环境时很有用。

最近刚被接纳的 PEP 370 (参见 <http://www.python.org/dev/peps/pep-0370>) 在 Python 中为每个用户添加了一个 `site-packages` 文件夹，并能构建出与 `virtualenv` 相同的隔离。Python 2.7 中该特性将可用，它将大大简化定制环境的构建工作。

在新的专用文件夹中为 Atomisator 创建这样一个环境，如下所示。

```
$ mkdir Atomisator
$ cd Atomisator
$ virtualenv --no-site-packages .
New python executable in ./bin/python
Installing setuptools.....done.
```

virtualenv 很了不起的一点是，所有使用本地 Python 解释程序或本地 easy_install 安装的包也将被安装到本地。

6.3.1 添加一个测试运行程序

为了构建我们的应用程序，需要一个测试运行程序。前一章中在包模板中将 Nose 定义为默认的测试运行程序，现在也将它作为全局配置添加到环境中，命令如下。

```
$ bin/easy_install nose
```

这将在使用本地 Python 解释程序的本地 bin 文件夹中添加一个 nosetests 命令。这样，这个测试运行程序能看到所有添加到这个环境中的包。注意，已经对包模板做过了设计，以便使用 test 命令时自动地调用 Nose 测试运行程序。

可以在系统中添加一个符号链接，或者在 PATH 环境变量中添加上 bin 文件夹，以让用户易于使用。

如果有多个环境，最好是指定特定的名称，并且使它们全局可用。在 Linux 或 Mac OS X 下，可以采用以下方法。

```
$ sudo ln -s bin/nosetests /usr/bin/atomisator-nosetests
$ sudo ln -s bin/python /usr/bin/atomisator-python
```



第 11 章中将说明测试运行程序的定义，并且对比一些产品。

6.3.2 添加一个包结构

目前，我们的 Atomisator 文件夹中有一个带有 Python 解释程序和一个测试运行程序的 bin 文件夹。我们打算构建的包应该集中在 packages 子目录下，以便简化版本系统的跟踪，并推进其部署：packages 中的所有文件夹将是用于 Atomisator 应用程序的定制 Python 包。

这个初始结构足以让我们着手开始编写包代码。

6.4 编写各个包

按照前面的图示，各个包将按照依赖顺序进行构建，也就是：

- (1) atomisator.parser;
- (2) atomisator.db;
- (3) atomisator.feed;
- (4) atomisator.main。

6.4.1 atomisator.parser

Python 中读取 RSS 2.0 的标准工具是 Universal Feed Parser (<http://www.feedparser.org>)。许多需要从 feed 中提取条目的程序都在使用它，而不管这些 feed 是以 RSS 1.0、RSS 2.0 或 Atom 格式提供的。对于我们的需求来说，这是个完美的工具。

不需要特别的封装就可以直接使用这个工具，但是控制外部程序在一个程序中的使用方式是一个好习惯。确认外部包中的所有调用都是从相同的定制包或模块中发起的，这样能使系统不断演化时更容易重构。这个工作只需要在代码库的一个地方就能完成，因此与应用程序的其余部分的依赖性更容易控制。

包封装器有如下两种类型。

- 漏封装器 (Leaky wrappers): 它们在外部包之上提供一个只用于发布外部包的包。它可以是一个简单的导入程序，也可以是外部程序库 API 周边的几个辅助类（参见第 14 章中介绍的外观设计模式）。
- 全封装器 (Full wrappers): 它们像程序库之外的黑盒子，提供全功能的 API。

后者可能是确保外部程序库和程序其余部分没有依赖关系的最佳方法。但是，这常常意味着必须编写很多额外的代码来遮蔽它。当项目开始时要正确地使用它也很困难。一个智能的 Leaky 封装器 API 常常更简单，而且能更好地避免重新发明轮子。

对于我们的 feed 解析器而言，选择很简单：创建一个具有单一外观函数的封装器，因为 Universal Feed Parser 将返回基本类型。

编写这样一个包的过程如下：

- (1) 使用合适的模板创建初始包；
- (2) 创建初始的 doctest，描述该示例如何使用；
- (3) 构建测试环境；

(4) 编写代码并调整初始的 doctest。

1. 创建初始包

该包将在 packages 文件夹中使用 pbp_package 模板创建，命令如下。

```
$ cd Atomisator/packages
$ paster create -t pbp_package atomisator.parser
```

这个命令将生成包结构。现在，该包可以调用 develop 命令链接到解释程序中去，如下所示。

```
$ cd atomisator.parser
$ atomisator-python setup.py develop
running develop
...
Finished processing dependencies for atomisator.parser==0.1.0
```

因为我们的模板中有一个位于 atomisator/parser/README.txt 的默认 doctest，在空包上运行带--doctest-extension=.txt 选项的 nosetests 就将运行这个文件，如下所示。

```
$ atomisator-nosetests --doctest-extension=.txt
-----
Ran 1 test in 0.162s
OK
```

这些测试可能已经通过 python setup.py test 启动了，但是使用全局的 nosetests 脚本更容易添加一些选项，如果需要可以在多个包上运行测试。

注意，如果在主目录中添加了一个.noserc 文件，可以省略 doctest-extension 选项（将在第 11 章中解释）。

接下来，与 feedparser (PyPI 上的全局 Feed 解析器名称) 之间的依赖关系将被添加到 atomisator.parser/setup.py 文件中，如下所示。

```
...
setup(name='atomisator.parser',
      version=version,
      description=("A thin layer on the top of "
                  "the Universal Feed Parser"),
      long_description=long_description,
      classifiers=classifiers,
      keywords='python best practices',
```

```

author='Tarek Ziade',
author_email='tarek@ziade.org',
url='http://atomisator.ziade.org',
license='GPL',
packages=find_packages(exclude=['ez_setup']),
namespace_packages=['atomisator'],
include_package_data=True,
zip_safe=False,
install_requires=[
    'setuptools',
    'feedparser'
    # -*- Extra requirements: -*-
],
entry_points="""
# -*- Entry points: -*-
""",
)
...

```

再次运行 atomisator-python setup.py develop 命令，将从 PyPI 获得 feedparser 并将其链接到环境中。然后，就可以开始编写初始包了。

2. 创建初始的 doctest

位于 atomisator.parser/atomisator/parser/ 的 README.txt 文件是人们在使用包时将参考的文档，其内容是一小段说明包的用途和使用范例的文字。

因为它是一个 doctest，所以可以帮助构建使用 reStructuredText 正确的测试驱动开发的实际代码。

 第 11 章中将详细说明如何编写测试，而 reStructuredText 的格式将在第 10 章中介绍。

这个文件的第一稿可以如下所示。

```
=====
atomisator.parser
=====
```

```
The parser knows how to return a feed content, with
the 'parse' function, available as a top-level function::

>>> from atomisator.parser import parse

This function takes the feed url and returns an iterator
over its content. A second parameter can specify a maximum
number of entries to return. If not given, it is fixed to 10::

>>> res = parse('http://example.com/feed.xml')
>>> res
<generator ...>

Each item is a dictionary that contain the entry::

>>> res.next()
```

这段文本指定的元素足够开始构建这个包。确保如果调用 bin/test 脚本，那么将执行它，并且会抛出一个错误，因为现在还没有任何代码，如下所示。

```
$ atomisator-nosetests --doctest-extension=.txt
...
File "atomisator.parser/atomisator/parser/docs/README.txt", line 8, in
README.txt
Failed example:
    from atomisator.parser import parse
Exception raised:
Traceback (most recent call last):
...
    File "<doctest README.txt[0]>", line 1, in ?
        from atomisator.parser import parse
ImportError: cannot import name parse
-----
Ran 1 test in 0.170s
FAILED (failures=1)
```

现在，这种通过修改代码直到测试通过的测试驱动开发是很容易的。

3. 构建测试环境

当创建了一个包时，确保其包含的所有测试可以在没有任何外部依赖模型的情况下启动是金科玉律。在测试装置中需创建许多虚拟函数来模拟所有指向外部元素的调用，有时候这很难做到。例如，一个依赖于 LDAP 服务器的包应该得到真实的数据才能被正确地构建和测试。这种情况下，从一个真实的服务器开始并记录其输出是一个好的习惯。然后，可以用一

个虚拟的服务器来处理这些收集到的数据。

 当虚拟部件的创建很复杂时，也可以使用模拟对象。有关这种方法的详细信息请参见第 11 章。

对于 `atomisator.parser`，避免调用 URL 的最简单方法是使用一个普通的 XML 文件，因为 `feedparser` 也支持它，从而使包依赖于一个 Web 连接。获取一个 feed 并将它保存在测试文件夹中，文件名为 `sample.xml`，如下所示。

```
cd atomisator/parser/tests  
wget http://ziade.org/atomisator/sample.xml
```

 这是为本书所制作的一个简单 feed。所以，稍后的实例在测试中看起来会很类似，但是任何其他的 feed 也能满足需要。

可以对 `README.txt` 做相应修改以便使用它，如下所示。

```
...  
>>> res = parse(os.path.join(test_dir, 'sample.xml'))  
...
```

现在，这个包可以独立于 Web 连接进行测试了。

4. 编写代码

由此，可以添加一个 `Parse` 函数到包中，并且一直构建到测试通过。最后的形式如下。

```
from feedparser import parse as feedparse  
from itertools import islice  
from itertools import imap  
  
def _filter_entry(entry):  
    """Filters entry fields."""  
    entry['links'] = [link['href'] for link in entry['links']]  
    return entry  
def parse(url, size=10):  
    """Returns entries of the feed."""  
    result = feedparse(url)
```

```
return islice(imap(_filter_entry,
                    result['entries']), size)
```

调整后的 doctest 是：

```
...
Each item is a dictionnary that contain the entry::

>>> entry = res.next()
>>> entry['title']
u'CSSSEdit 2.0 Released'

The keys available are:
>>> keys = sorted(entry.keys())
>>> list(keys)
['id', 'link', 'links', 'summary', 'summary_detail',
'tags', 'title', 'title_detail']
```

6.4.2 atomisator.db

按照相同的原则来构建 atomisator.db 包。添加一个新的名为 atomisator.db 的包，并且连接到本地解释程序上。

注意，在使用 Nose 测试运行程序时，如果在相同的命名空间中有多个包，将会运行所有的测试。所以，将在进行 atomisator.db 测试时也运行 atomisator.parser 测试。虽然这往往不是一个问题，但是关注于一个特定的包时可能会想使用过滤选项。

本小节剩余的内容中将着重介绍使用数据库的常见方式，即 SQL 相关的元素。

1. SQLAlchemy

Python 中使用关系数据库最方便的方法是使用 SQLAlchemy (<http://www.sqlalchemy.org>)，这是一个对象—关系映射程序（参见http://en.wikipedia.org/wiki/Object-relational_mapping）。这个工具提供一个能够实现 Python 对象和 SQL 数据库表中的行同步的映射系统，而且不需要编写任何代码。

SQLAlchemy 有多个可用的数据库后端，也就是可以使用多种数据库，如 PostgreSQL、SQLite、MySQL 甚至 Oracle。这种方法的一种优秀特性是，不管切换成哪种后端系统，都只需要使用相同的代码。这意味着在测试环境中可以使用一个简单的 SQLite 文件来构建连接器，而在生产环境中则使用 PostgreSQL，如下所示。

```
>>> if big_daddy_server:
...     sqluri = 'postgres://tarek@localhost/database'
```

```
... else:  
...     sqluri = 'sqlite://relative/path/to/database.db'
```

当然，这种特性只限于对所有数据库系统共用的并且为 SQLAlchemy 所包含的操作。但是，只要所提供的 API 被用于与数据库交互，这种切换就是可能的。这在必须使用诸如为了优化而定义的存储过程之类的特殊调用时是很难做到的。但是对于大部分数据库中的东西，Python 包都可以使用 SQLite 作为测试数据库来构建。

因此 SQLAlchemy 可以被看作 Python 的一个通用数据库使用程序。社区中的许多 Python 项目依赖于这个工具，它现在已经被认为是处理数据库的最佳方法之一。另一个相似的项目也可以考虑，这就是来自 Ubuntu 制造商 Canonical 的 Storm (<https://storm.canonical.com>)。这个工具使用一个隐含的映射系统，而 SQLAlchemy 依赖一个必须在 Python 端定义的显式映射系统，用来描述 Python 对象与 SQL 数据库表的链接方式。

2. 创建映射

`atomisator.db` 数据库模型相当简单，因为它只包含一个条目表、一个链接表和一个标记表。图 6.3 提供了一个简化的数据库视图。主表 `atomisator_entry` 中包含 `atomisator.parser` 所提供的 `feed` 条目以及当前日期，`atomisator_link` 和 `atomisator_tag` 是保存一系列唯一值以及使条目指向它们 (`links` 和 `tags` 字段) 的两个辅助表。SQLAlchemy 自动地提供了这些一对多关系，这样一个结构可以被描述为图 6.3 所示的模型。

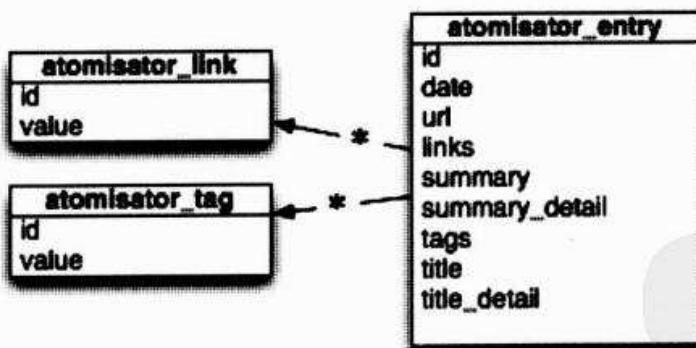


图 6.3

```
from sqlalchemy import *
from sqlalchemy.orm import *
from sqlalchemy.orm import mapper
metadata = MetaData()
link = Table('atomisator_link', metadata,
             Column('id', Integer, primary_key=True),
             Column('url', String(300)),
```

```

        Column('atomisator_entry_id', Integer,
               ForeignKey('atomisator_entry.id')))

class Link(object):
    def __init__(self, url):
        self.url = url
    def __repr__(self):
        return "<Link('%s')>" % self.url

mapper(Link, link)

tag = Table('atomisator_tag', metadata,
            Column('id', Integer, primary_key=True),
            Column('value', String(100)),
            Column('atomisator_entry_id', Integer,
                   ForeignKey('atomisator_entry.id')))

class Tag(object):
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return "<Tag('%s')>" % self.value

mapper(Tag, tag)

entry = Table('atomisator_entry', metadata,
              Column('id', Integer, primary_key=True),
              Column('url', String(300)),
              Column('date', DateTime()),
              Column('summary', Text()),
              Column('summary_detail', Text()),
              Column('title', Text()),
              Column('title_detail', Text()))

class Entry(object):
    def __init__(self, title, url, summary, summary_detail='',
                 title_detail=''):
        self.title = title
        self.url = url
        self.summary = summary
        self.summary_detail = summary_detail
        self.title_detail = title_detail
    def add_links(self, links):
        for link in links:

```

```

        self.links.append(Link(link))
    def add_tags(self, tags):
        for tag in tags:
            self.tags.append(Tag(tag))
    def __repr__(self):
        return "<Entry(%r)>" % self.title
mapper(Entry, entry, properties={
    'links':relation(Link, backref='atomisator_entry'),
    'tags':relation(Tag, backref='atomisator_entry'),
})

```

本书中不打算详细地研究这些代码，可以通过阅读官方教程 (<http://www.sqlalchemy.org/docs/04/ormtutorial.html>) 来了解更多信息。而且，SQLAlchemy 是非常活跃的框架，所以到本书付印时，在这里列出的代码可能就不是最好的做法了。

理解数据库上的每个数据库表将与 Python 端的一个类相连是很重要的一点。每个类的实例将表示数据库表中的一行。



atomisator.db 和本书创建的所有包类似，可以在 PyPI 找到。最近的版本已经完成，但是还有与此不同的演化版本。

3. 提供 API

在这些映射之上，API 必须提供添加和查询条目的方法。最后，和代码一起构建的主 doctest 看上去如下所示。

```

=====
atomisator.db
=====

This package provides a few mappers to store feed entries
in a SQL database.

The SQL uri is provided in the config module::

    >>> from atomisator.db import config
    >>> config.SQLURI = 'sqlite:///:memory:'

Let's create an entry::

    >>> from atomisator.db import create_entry
    >>> entry = {'url': 'http://www.python.org/news',
... 'summary': 'Summary goes here',

```

```

... 'title': 'Python 2.6alpha1 and 3.0alpha3 released',
... 'links': ['http://www.python.org'],
... 'tags': ['cool', 'fun']}
>>> id_ = create_entry(entry)
>>> type(id_)
<type 'int'>
We get the database id back. Now let's look for entries::
>>> from atomisator.db import get_entries
>>> entries = get_entries() # returns a generator object
>>> entries.next()
<Entry('Python 2.6alpha1 and 3.0alpha3 released')>
Some filtering can be done :::
>>> entries = \
...     get_entries(url='http://www.python.org/news')
>>> entries.next()
<Entry('Python 2.6alpha1 and 3.0alpha3 released')>
When no entry is found, the generator is empty:::
>>> entries = get_entries(url='xxxx')
>>> entries.next()
Traceback (most recent call last):
...
StopIteration

```

这个包将提供两个全局函数，用来完成数据库的处理：

- `get_entries` 返回可过滤的条目；
- `create_entry` 添加一个条目。

6.4.3 atomisator.feed

`atomisator.feed` 使用 `atomisator.db` 读取最新的条目，并且生成一个在 RSS 中表示它们的 XML 文件。这可以通过前一章中用于创建代码骨架的 Cheetah 模板引擎来完成。RSS 模板文件实现这个 RSS 2.0 结构，如下所示。

```

<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntaxns#">
<channel>
<title><![CDATA[{$channel.title}]]></title>
<description><![CDATA[{$channel.description}]]></description>

```

第6章 编写一个应用程序

```
<link>${channel.link}</link>
<language>en</language>
<copyright>Copyright 2008, Atomisator</copyright>
<pubDate>${publication_date}</pubDate>
<lastBuildDate>${build_date}</lastBuildDate>
#for $entry in $entries
<item>
    <title><![CDATA[${entry.title}]]></title>
    <description><![CDATA[${entry.summary}]]></description>
    <link><![CDATA[${entry.url}]]></link>
    <pubDate>${entry.date}</pubDate>
</item>
#end for
</channel>
</rss>
```

条目内容由数据库提供，诸如频道标题这样的额外信息由配置给出。这个包的 doctest 看上去如下所示。

```
=====
atomisator.feed
=====

Generates a feed using a template::
>>> from atomisator.feed import generate
>>> print generate('feed', 'the feed', 'http://link')
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:rdf="...">
<channel>
    <title><![CDATA[feed]]></title>
    <description><![CDATA[the feed]]></description>
    <link>http://link</link>
    <language>en</language>
...
<item>
    <title><![CDATA[Python 2.6alpha1 and
            3.0alpha3 released]]></title>
    <description><![CDATA[Summary goes here]]></description>
    <link><![CDATA[http://www.python.org/news]]></link>
```

```

<pubDate>...</pubDate>
</item>
...
</channel>
</rss>
```

这个包可以供 main 包使用，用来刷新 feed 的条目。

6.4.4 atomisator.main

main 包通过读取一个名为 atomisator.cfg 的配置文件来将所有东西组合在一起，这个配置文件提供了一个 feed 列表和几个全局变量，如下所示。

```

[atomisator]
# 要读取的 feed
sites =
    sample1.xml
    sample2.xml
# 数据库位置
database = sqlite:///atomisator.db
# 用于频道的字段
title = My Feed
description = The feed
link = the link
# 生成文件的名字
file = atomisator.xml
```

这个文件被专门的模块 config 所读取。由此，主模块将提供 3 个方法来组合拼图的每个小块，如下所示。

```

from atomisator.main.config import parser
from atomisator.parser import parse
from atomisator.db import config
from atomisator.db import create_entry
from atomisator.feed import generate
config.SQLURI = parser.database
def _log(msg):
    print msg
def load_feeds():
```

```
"""Fetches feeds."""
for count, feed in enumerate(parser.feeds):
    _log('Parsing feed %s' % feed)
    for entry in parse(feed):
        count += 1
        create_entry(entry)
    _log('%d entries read.' % count+1)
def generate_feed():
    """Creates the meta-feed."""
    _log('Writing feed in %s' % parser.file)
    feed = generate(parser.title,
                    parser.description, parser.link)
    f = open(parser.file, 'w')
    try:
        f.write(feed)
    finally:
        f.close()
    _log('Feed ready.')
def atomisator():
    """Calling both."""
    load_feeds()
    generate_feed()
```

然后，它们在 `setup.py` 中连接成控制台脚本，如下所示。

```
...
entry_points = {
    "console_scripts": [
        "load_feeds = atomisator.main:load_feeds",
        "generate_feed = atomisator.main:generate_feed",
        "atomisator = atomisator.main:atomisator"
    ]
}
...
...
```

这将在系统中添加 3 个新的指向这 3 个函数的可执行脚本。

这个包还在 `setup.py` 中将其他 `atomisator` 定义为依赖模块，以便在安装 `atomisator.main` 的同时安装它们，如下所示。

```
...
```

```

install_requires=[  

    'atomisator.db',  

    'atomisator.feed',  

    'atomisator.parser'  

],
...

```

6.5 分发 Atomisator

这个程序现在可以通过将 egg 放入 PyPI 中的方法来分发。每个包可以使用 `sdist`、`bdist` 或 `bdist_egg` 命令来作为 egg 发行。对于我们的应用程序，因为没有需要编译的代码，因此一个源代码分发版本就足够满足所有平台的需求。

对每个 `egg`、`register` 和 `upload` 命令可以和 `sdist` 一起调用，但是之前必需要按照前一章中的方法创建了一个用户账户。`register sdist upload` 命令序列将在 PyPI 注册这个包，构建一个源分发版本，并且上传，如下所示。

```

$ cd atomisator.main  

$ python setup.py register sdist upload  

Using PyPI login from /Users/tarek/.pypirc  

Registering atomisator.parser to http://pypi.python.org/pypi  

Server response (200): OK  

running sdist  

...  

running upload  

Using PyPI login from /Users/tarek/.pypirc  

Submitting dist/atomisator.parser-0.1.0.tar.gz to http://pypi.python.org/pypi  

Server response (200): OK

```

之后，该包就在 PyPI 上可用。

调用 `alias` 命令为每个包创建一个 `release` 别名也是一个好习惯，如下所示。

```

$ python setup.py alias release register sdist upload  

running alias  

Writing setup.cfg  

$ more setup.cfg  

[aliases]

```

```
release = register sdist upload
```

release 命令可以用来推送这个包。在 atomisator.db 中完成这一任务，并且为其他包也完成这项工作，如下所示。

```
$ cd atomisator.db
$ python setup.py alias release register sdist upload
running alias
Writing setup.cfg
$ python setup.py release
running register
Using PyPI login from /Users/tarek/.pypirc
Registering atomisator.db to http://pypi.python.org/pypi
Server response (200): OK
...
running upload
Using PyPI login from /Users/tarek/.pypirc
Submitting dist/atomisator.db-0.1.0.tar.gz to http://pypi.python.org/pypi
Server response (200): OK
$ cd ../atomisator.feed/
$ python setup.py alias release register sdist upload
...
$ python setup.py release
...
$ cd ../atomisator.parser
...
```

这样，这 4 个包就在 PyPI 上可用。如果在 PyPI 中尝试搜索“atomisator” (<http://pypi.python.org/pypi?%3Aaction=search&term=atomisator&submit=search>)，就应该能够找到所有 4 个包。

现在从任何安装了 setuptools 的计算机上，都可以使用以下命令安装 Atomisator。

```
$ easy_install atomisator.main
```

这个命令将安装所有的包及其依赖模块，并且使 atomisator.main 中的 3 个命令立即可用。

6.6 包之间的依赖性

以多个包的形式分发应用程序，在发行的时候会产生一些开销。

例如，如果在 `atomisator.db` 中做了一些影响 `atomisator.main` 的修改，就必须：

- 升级每个包的版本；
- 确保新的 `atomisator.main` 获得正确的 `atomisator.db` 版本；
- 重新发行两个包。

版本依赖性可以在 `setup.py` 中的 `install_requires` 元数据中直接配置。例如，如果 `atomisator.main` 的 1.4.6 版本需要至少 1.4.4 版本的 `atomisator.db` 才能运行，那么它的 `setup.py` 的内容将如下所示。

```
version = '1.4.6'  
...  
    install_requires=[  
        'atomisator.db>=1.4.4',  
        'atomisator.feed',  
        'atomisator.parser'  
    ],  
...
```

要减少这种开销，可以通过编写几个脚本自动化应用程序来实现。在任何情况下，将应用程序分割为几个包要比依赖于单个包更好一些。如果每个包表现应用程序的一个逻辑部分，那么每一个都将按照自己的计划来进行演化。同时，如果两个包总是同时被修改，那么可能意味着它们应该被合并到一个包里。

在开始编码时不要太担心应用程序如何分割。如果分割不正确，那么在开发期间出现问题，将总是能够通过重构代码来修复问题。

6.7 小结

本章介绍了一个示例应用程序的实现，这个应用以多个 PyPI 上相同命名空间下的包来分发。

我们使用 `virtualenv` 创建了一个工作环境，并讨论了与所开发的包一起工作的测试运行程序的设置方法。

要创建更大的程序时，必须完成一些 Python 安装之外的包安装器的工作。

下一章将进一步介绍这一主题，并且提出 `zc.buildout`，这是一个用于构建应用程序环境的常用工具。

第 7 章

使用 `zc.buildout`

前一章中介绍了编写基于多个 egg 的应用程序的方法。当分发这样的应用程序时，用户将包及其依赖模块安装到 Python 的 site-packages 文件夹中，并且获得一些入口点，如命令行实用程序。

但是对于比 Atomisator 更大的应用程序而言，这种方法就存在局限性：如果需要部署一些配置文件或者编写日志文件，将它们放在代码包内部就不现实了。

最好的办法是，创建一个专用的安装程序来将它们无缝地集成到目标系统上。以基于 Linux 的系统为例，日志文件应该被放在 /var/log 中，而配置文件则应该被放在 /etc。但是，创建这样的安装程序需要许多和系统相关的工作。

另一种方法，有点类似 `virtualenv` 所提供的，是建立一个具有运行应用程序所需的所有文件的自包含目录，然后分发它。这个目录也可以包含所需的包，以及在目标系统上启动所有程序的安装程序。

`zc.buildout` (参见 <http://pypi.python.org/pypi/zc.buildout>) 是用来创建这样一个环境的工具，本章将介绍如何：

- 通过一种能够定义运行应用程序所需的所有包的描述性语言来组织一个应用程序；
- 将这样的应用程序作为一个源代码发行版本来部署。

 `zc.buildout` 的替代工具是 Paver 和 AutomateIt (参见 <http://www.blueskyonmars.com/projects/paver> 和 <http://automateit.org>)。

本章由 3 个部分组成：

- `zc.buildout` 原理；
- 基于 `zc.buildout` 的应用程序的分发方法；
- 使用 `paster` 工具创建一个 `zc.buildout` 应用程序环境的应用程序模板。

7.1 zc.buildout 原理

`virtualenv` 对于隔离一个 Python 环境相当方便。正如前一章中所看到的，它是工作在本地的，但是仍然需要手工完成许多设置和维护项目环境的工作。

`zc.buildout` 提供了相同的隔离功能，并且还进一步提供了：

- 一个简单的、在一个配置文件中定义这些依赖性的描述性语言；
- 提供链接代码调用组合的输入点的插件系统；
- 一种部署和发行应用程序源代码及其执行环境的方法。

配置文件将描述环境中所需的 egg，它们的状态（正在本地开发，在 PyPI，或其他地方可用）以及构建应用程序所需要的所有其他元素。

插件系统将注册这些包并按照执行的顺序将它们链接起来。

最后，整个环境是独立和隔离的，因而可以像发行和部署后那样使用。

`zc.buildout` 在其 PyPI 页面 (<http://pypi.python.org/pypi/zc.buildout>) 上提供了很好的文档。本节只是摘要地介绍了构建和在应用程序级别工作时所需要知道的最重要的要素。这些要素包括：

- 配置文件结构；
- `buildout` 命令；
- Recipes。

7.1.1 配置文件结构

`zc.buildout` 依赖于一个结构与 `ConfigParser` 模块兼容的配置文件。这些类 INI 文件中有以 [headers] 分割的小节，小节中的行包含 `name:value` 或 `name=value` 这样的内容。

1. 最小的配置文件

最小的 `buildout` 配置文件中包含一个名为 [buildout] 的小节，其中有一个称作 parts 的变量。这个变量包含一个提供小节列表的多行值，如下所示。

```
[buildout]
parts =
    part1
    part2
[part1]
recipe = my.recipe1
[part2]
recipe = my.recipe2
```

在 parts 中指定的每个小节至少有一个提供包名称的 recipe 值。这个包可以是任何 Python 包，只要它定义一个 zc.buildout 入口点。

用这个文件，buildout 将执行以下工作序列：

- 检查 my.recipe1 包是否安装，如果没有安装，读取并完成本地安装；
- 执行 my.recipe1 入口点所指向的代码；
- 然后，对 parts 做同样的事情。

因而，buildout 是一个基于插件的脚本，将被称为 recipe 的独立包的执行链接起来。用这个工具构建的环境包括 recipe 正确顺序的定义。

2. [buildout]小节选项

除了 parts，[buildout]小节还有多个可用的选项，最重要的是：

- develop 它是一个多行值，列出使用 python setup.py develop 命令将安装的 egg。每个值是指向 setup.py 所在的包文件夹路径；
- find-links 它是一个多行值，提供了一个位置（URL 或文件）列表，easy_install 基于它查找在 eggs 中定义的 egg 或在安装 egg 时的依赖模块。

由此，一个 buildout 可以列出一系列将被安装在该环境中的 egg。

对于 develop 中指出的每个值，该工具运行 setuptools develop 命令并在定义了依赖性时读取 PyPI。

用于查找包的 Web 位置与 easy_install 所用的一样——<http://pypi.python.org/simple>。这是一个不打算供人们使用的，包含可被自动浏览的一个包链接列表的网页。

最后，find-links 选项提供了一个当包在其他位置可用时指向替代来源的方法。

示例如下。

```
[buildout]
parts =
    develop =
        /home/tarek/dev/atomisator.feed
find-links =
    http://acme.com/packages/index
```

使用这个配置，buildout 将和 python setup.py develop 命令一样安装 atomisator.feed 包。可以使用 buildout 命令来构建这个环境。

7.1.2 buildout 命令

buildout 命令是 zc.buildout 安装的，通常会被 easy_install 调用，以对配置文件进行解释，如下所示。

```
$ easy_install zc.buildout
...
$ buildout
While:
    Initializing.
Error: Couldn't open /Users/tarek/buildout.cfg
```

一个空目录中的一个带有 init 选项的初始调用将创建一个默认的 buildout.cfg 文件和几个其他元素，如下所示。

```
$ cd /tmp
$ mkdir tests
$ cd tests
$ buildout init
Creating '/tmp/tests/buildout.cfg'.
Creating directory '/tmp/tests/bin'.
Creating directory '/tmp/tests/parts'.
Creating directory '/tmp/tests/eggs'.
Creating directory '/tmp/tests/develop-eggs'.
Generated script '/tmp/tests/bin/buildout'.
$ find .
.

./bin
./bin/buildout
./buildout.cfg
./develop-eggs
./eggs/setuptools-0.6c7-py2.5.egg
./eggs/zc.buildout-1.0.0b30-py2.5.egg
./parts
$ more buildout.cfg
[buildout]
parts =
```

bin 文件夹中包含一个本地 buildout 脚本，还创建了其他 3 个文件夹：

- parts 对应配置文件中定义的小节，它是每个被调用的 recipe 可以写入元素的标准位置；
- develop-eggs 保存将环境链接到 develop 中定义的包的信息；
- eggs 包含该环境使用的 egg，它已经包含了 zc.buildout 和 setuptools 两个 eggs。

接下来，在 cfg 中添加一个 develop 小节，如下所示。

```
[buildout]
parts =
develop =
/home/tarek/dev/atomisator.feed
```

指定的文件夹将被 zc.buildout 再次调用 buildout 命令安装为 develop egg，如下所示。

```
$ bin/buildout
Develop: '/home/tarek/dev/atomisator.feed'
$ ls develop-eggs/
atomisator.feed.egg-link
$ more develop-eggs/atomisator.feed.egg-link
/home/tarek/dev/atomisator.feed
```

develop-eggs 文件夹现在包含了一个指向位于 /home/tarek/dev/atomisator.feed 的 atomisator.feed 包的链接。当然，任何包文件夹都可以使用 develop 选项捆绑到 buildout 脚本上。

7.1.3 recipe

我们已经看到，每个小节将一个包指定为 recipe。例如，zc.recipe.egg 被用来指定一个或多个 buildout 中安装的 egg。这个 recipe 将像 easy_install 一样，通过调用 PyPI 拖动包，并且最终在 PyPI 没有包含它的情况下查找 find-links 中提供的链接。

例如，如果想将 Nose 安装到 buildout，可以在配置文件中配置一个专门的小节，并且在 buildout 小节中的 parts 变量中指向它，如下所示。

```
[buildout]
parts =
    test
develop =
    /home/tarek/dev/atomisator.feed
[test]
recipe = zc.recipe.egg
eggs =
    nose
```

再次运行 buildout 脚本，将执行 test 小节并和 easy_install 一样拖动 Nose egg，如下所示。

```
$ bin/buildout
Develop: '/home/tarek/dev/atomisator.feed'
Installing test.
Getting distribution for nose
Got nose 0.10.3.
```

nosetests 脚本将被安装到 bin 文件夹中，Nose egg 将在 eggs 文件夹中。再次使用 zc.recipe.egg 在 cfg 文件中添加一个名为 other 的新小节，如下所示。

```
[buildout]
parts =
    test
    other
develop =
    /home/tarek/dev/atomisator.feed
```

```
[test]
recipe = zc.recipe.egg
eggs =
    nose
[other]
recipe = zc.recipe.egg
eggs =
    elementtree
    PIL
...
```

这个新的小节定义了两个新的包。再次运行 buildout 脚本，如下所示。

```
$ bin/buildout
Develop: '/home/tarek/dev/atomisator.feed'
Updating test.
Installing other.
Getting distribution for elementtree
Got elementtree 1.2.7-20070827-preview.
Getting distribution for 'PIL'.
Got PIL 1.1.6.
```

在 parts 中指向的小节将按照所定义的顺序运行。再次运行时，zc.buildout 将检查已安装的部件，了解它们是否需要更新，如果需要则安装新的版本。在 other 小节中，eggs 文件夹增加了两个新的 egg。

Recipe 是简单的 Python 包，一般专门用于这个目的。它们也是嵌入的命名空间包，其中第一个部分是组织名，第二个部分是 recipe，第三个部分是 recipe 的名称。

目前已经用过的 recipe 是由 Zope 公司 (zc) 提供的，但是 PyPI 上还有很多 recipe 可以用来处理 buildout 环境中的许多需求。

因为 Zope 或 Plone 这样的框架依赖于这个工具，所以在<http://pypi.python.org>上搜索 buildout 或 recipe 将返回几百个可用于构成各种 buildout 的包。

1. 著名的 recipe

以下是在 PyPi 上找到的有用的 recipe 的简短列表。

- collective.recipe.ant 构建 Ant (Java) 项目
- iw.recipe.cmd 执行一个命令行
- iw.recipe.fetcher 下载 URL 指向的一个文件
- iw.recipe.pound 编译和安装 Pound (一个负载平衡程序)
- iw.recipe.squid 配置和运行 Squid (一个缓存服务器)
- z3c.recipe.ldap 部署 OpenLDAP

2. 创建 recipe

Recipe 是一个简单的类，具有 `install` 和 `update` 两个方法，它们将返回安装文件的列表。因此，编写新的 recipe 代码非常简单且可以用模板完成。

ZopeSkel 项目在 Zope 社区中被用于构建新的 recipe，安装它可以拥有一个新的 recipe 模板，如下所示。

```
$ easy_install ZopeSkel
Searching for ZopeSkel
Best match: ZopeSkel 2.1
...
Finished processing dependencies for ZopeSkel
$ paster create --list-templates
Available templates:
...
recipe:           A recipe project for zc.buildout
...
```

recipe 生成一个嵌套的命名空间包结构，以及必须被完成的一个 Recipe 类的骨架，如下所示。

```
$ paster create -t recipe atomisator.recipe.here
Selected and implied templates:
ZopeSkel#recipe A recipe project for zc.buildout
...
Enter namespace_package ['plone']: atomisator
Enter namespace_package2 ['recipe']:
Enter package ['example']: here
Enter version (Version) ['1.0']:
Enter description ['']: description is here.
Enter long_description ['']:
Enter author (Author name) ['']: Tarek
Enter author_email (Author email) ['']: tarek@ziade.org
...
Creating template recipe
Creating directory ./atomisator.recipe.here
...
$ more atomisator.recipe.here/atomisator/recipe/here/__init__.py
# -*- coding: utf-8 -*-
"""Recipe here"""
class Recipe(object):
    """zc.buildout recipe"""
    def __init__(self, buildout, name, options):
```

```

    self.buildout, self.name, self.options = \
        buildout, name, options
def install(self):
    """Installer"""
    # XXX 实现这里的 recipe 的功能

    # 返回 recipe 创建的文件
    # 当重新安装时, buildout 将删除所有返回的文件
    return tuple()
def update(self):
    """Updater"""
    pass

```

7.1.4 Atomisator buildout 环境

Atomisator 项目可以通过创建一个和包在一起的专用的 buildout 配置来从 zc.buildout 获益，并且在配置中定义一个环境。

buildout 环境的构建可以分为两步：

- (1) 创建一个 buildout 文件夹结构；
- (2) 初始化 buildout。

buildout 文件夹结构

因为 buildout 允许将任何系统文件夹作为一个 develop 包来链接，所以应用程序环境可以与之分离。最清晰的设计是使用一个用于 buildout 和一个用于被开发的包的文件夹。

回顾一下前一章中创建的 Atomisator 文件夹。目前，它包含一个带有本地解释程序的 bin 文件夹和一个 packages 文件夹。在此，将添加一个 buildout 文件夹，如下所示。

```
$ cd Atomisator
$ mkdir buildout
```

然后，在 buildout 文件夹中构建一个新的 buildout 环境，如下所示。

```
$ cd buildout
$ buildout init
Creating 'Atomisator/buildout/buildout.cfg'.
Creating directory 'Atomisator/buildout/bin'.
Creating directory 'Atomisator/buildout/parts'.
Creating directory 'Atomisator/buildout/eggs'.
Creating directory 'Atomisator/buildout/develop-eggs'.
Generated script 'Atomisator/buildout/bin/buildout'.
```

修改 buildout.cfg 以便生成一个本地 nosetests 脚本，并将 Atomisator egg 作为 develop egg 安装，如下所示。

```
[buildout]
develop =
    ../packages/atomisator.main
    ../packages/atomisator.db
    ../packages/atomisator.feed
    ../packages/atomisator.parser
parts =
    test
[test]
recipe = pbp.recipe.noserunner
eggs =
    atomisator.main
    atomisator.db
    atomisator.feed
    atomisator.parser
```

这个配置文件将在 buildout 文件夹中生成一个完整的 Atomisator 环境。

前一章中，在与开发包相同的本地解释程序中安装了 Nose，这多亏了 virtualenv。在 buildout 中工作时，要拥有相同的功能则需要更多的工作：将 Nose 作为一个 egg 安装到 buildout 中，但是不让其他 egg 直接看到这个测试运行程序。为了得到一个相似的环境，一个小的名为 pbp.recipe.noserunner 的 recipe 将生成一个使用特定环境的本地运行程序 nosetests。所有在这个运行程序的 eggs 变量中定义的 egg，都将被添加到测试运行程序执行环境中。

这个 recipe 使用小节名作为生成脚本的名称。所以在我们的案例中将有一个 test 脚本，它可以用来测试所有的 atomisator 包，如下所示。

```
$ bin/test atomisator
.....
-----
Ran 8 tests in 0.015s
OK
```

7.1.5 更进一步

另一个可以执行的步骤是在 buildout 文件夹下的 etc 文件夹中创建和使用 atomisator.cfg 文件。这也需要创建一个新的 recipe，用它来读取 atomisator.cfg 中的值，并且生成 atomisator.cfg。

接下来创建的一个新小节，其内容类似于如下所示。

```
...
[atomisator-configuration]
recipe = atomisator.recipe.installer
sites =
```

```

sample1.xml
sample2.xml
database = sqlite://${buildout:directory}/var/atomisator.db
title = My Feed
description = The feed
link = the link
file = ${buildout:directory}/var/atomisator.xml
...

```

其中，\${buildout:directory}将替换为 buildout 的路径。

7.2 发行与分发

在前一节中已经看到，`buildout` 是一个单独的文件夹，可以包含所有运行应用程序所需要的内容。所有需要的 egg 都被安装在这个文件夹中，控制台脚本则创建在 `bin` 文件夹中。

事实上，Atomisator 文件夹可以被做成一个档案文件，然后在其他拥有 Python 的电脑上解压。通过在新的目标系统上再次运行 `buildout`，可以正确地启动所有的部件，应用程序可以由此运行。

以这样的方法分发源代码，从整体上看可以与每个操作系统提供的包管理系统（如 `apt` 或 `RPM`）相比。所有部件都单独地存在于一个自包含的文件夹中，并且能够在每个系统上运行。因此，它不能无缝地与目标系统集成，并将使用自己的专有标准。这对于许多应用程序来说是不错的，但是纯粹主义者希望可以使用目标系统上的包管理系统来安装，从而简化系统的维护工作。

如果这是必需的，就需要一些额外的平台特定的集成工作。这是一个非常广泛的主题，其内容超出了本书的范围，所以在此不做介绍，但是这里介绍的源代码发行是针对特定目标发行的第一步。

让我们关注于 `buildout` 文件夹的分发。

但是，将 `packages` 文件夹和 `buildout` 文件夹及链接成 `develop` egg 的子文件夹一起交付并不是最佳的选择，因为我们希望发行每个 egg 的标记版本。`buildout` 能够解释任何配置文件。所以最好的办法是创建一个配置文件，这个文件不会将 `develop` 选项与刚刚创建的每个包构建的一组 egg 一起使用。

所以，发行一个 `buildout` 由以下 3 个步骤来完成：

- (1) 发行包；
- (2) 创建发行配置；
- (3) 构建和准备发行版本。

7.2.1 发行包

可以使用 `sdist`、`bdist` 或 `bdist_egg` 命令来将每个包发行为 `egg`。对于我们的应用程序而言，由于没有需要编译的代码，所以一个源代码分发版本就足以应付所有的平台。

对于每个包，按照前一章所看到的相同方式，建立一个源代码分发版本，如下所示。

```
$ python setup.py sdist
running sdist
...
Writing atomisator.db-0.1.0/setup.cfg
tar -cf dist/atomisator.db-0.1.0.tar atomisator.db-0.1.0
gzip -f9 dist/atomisator.db-0.1.0.tar
removing 'atomisator.db-0.1.0' (and everything under it)
$ ls dist/
atomisator.db-0.1.0.tar.gz
```

其结果是一个档案文件，它将被推送到 PyPI 或者保存在一个文件夹中。

7.2.2 添加一个发行配置文件

`zc.buildout` 提供了一个扩展机制，可以按照层次创建配置文件。使用指定另一个配置文件的 `extends` 选项，一个文件就可以继承所有值然后添加新的值，或者覆盖一些原有值。

一个专用于发行的新配置文件可以按以下的风格来创建，以用于特殊的设置：

- 需要指向 `buildout` 已发行的包；
- 需要去掉 `develop` 选项。

得到的结果如下。

```
[buildout]
extends = buildout.cfg
develop =
parts =
    atomisator
    eggs
download-cache = downloads
[atomisator]
recipe = zc.recipe.eggs
eggs =
    atomisator.main
    atomisator.db
    atomisator.feed
    atomisator.parser
```

在这里，`download-cache` 是一个系统文件夹，`buildout` 将从 PyPI 上下载的 egg 保存在这里。`downloads` 文件夹最好是创建在 `buildout` 文件夹内，如下所示。

```
$ mkdir downloads
```

`eggs` 部分是从 `buildout.cfg` 继承的，并且不需要复制到新文件中。`atomisator` 部分将从 PyPI 拖动 egg 并将它们保存在 `downloads` 中。

7.2.3 构建和发行应用程序

接着，可以使用这个特殊的配置来构建 `buildout`。使用-c 选项指定一个特定的配置文件，使用-v 选项可以得到更多细节，如下所示。

```
$ bin/buildout -c release.cfg -v
Installing 'zc.buildout', 'setuptools'.
...
Installing atomisator.
Installing 'atomisator.db', 'atomisator.feed', 'atomisator.parser',
'atomisator.main'.
...
Picked: setuptools = 0.6c8
```

当这个步骤完成时，包将被下载并保存在 `downloads` 文件夹中，如下所示。

```
$ ls downloads/dist/
atomisator.feed-0.1.0.tar.gz
atomisator.main-0.1.0.tar.gz
atomisator.db-0.1.0.tar.gz
atomisator.parser-0.1.0.tar.gz
```

这意味着下一次运行时不需要从 PyPI 拖动包。换句话说，这时可以在脱机模式下构建 `buildout`。

发行版本已经可以通过分发 `buildout` 文件夹（例如，用一个归档的版本）来交付了。

最后要做的一件事情就是在文件夹中添加一个 `bootstrap.py` 文件，它可以自动化 `zc.buildout` 的安装，并和 `buildout init` 同样在目标系统上创建 `bin/buildout` 脚本，如下所示。

```
$ wget http://ziade.org/bootstrap.py
```

[ 社区中有一些工具提供了一些脚本，可以使用附加的选项来准备这些归档版本，如 `collective.releaser` 和 `zc.sourcerelease`。]

7.3 小结

本章中主要介绍了 `zc.buildout`:

- 可以用来构建基于 `egg` 的应用程序;
- 知道如何聚集 `egg` 以构建一个独立的环境;
- 链接 `recipe` (这是一些小的 Python 包) 以构建一个脚本, 这个脚本可以建立用于确定 Python 应用程序源代码分发版本的环境。

总而言之, 使用 `zc.buildout` 的步骤包括:

- 使用一个 `egg` 列表创建一个 `buildout`, 并用它来开发;
- 创建一个专用于发行的配置文件, 并用它来构建一个可分发的 `buildout` 文件夹。

下一章中将进一步介绍这个工具, 说明使用它和其他工具一起管理项目的方法。

第 8 章

代码管理

当项目涉及多个人的时候，工作会更加棘手。所有事情的速度都将减慢，也将更难。这可以归结于多个原因，本章将揭示这些原因，并尝试提供与之对抗的一些方法。

本章由两个部分组成，分别介绍：

- 如何使用版本控制系统；
- 如何配置持续集成机制。

首先，代码基准库有了非常大的发展，因此跟踪所有修改就显得更加重要，尤其是涉及许多开发人员的时候。这就是版本控制系统所担当的角色。

其次，几个没有直接相连的大脑为相同的项目工作。他们各有不同的角色，从事不同领域的工作。因此，缺乏全局可见性将产生许多与事情的进展和其他人所完成工作等方面混乱。这是不可避免的，必须通过一些工具来提供持续的可见性，并减轻这些问题。这可以通过配置一系列用于持续集成（continuous integration）的工具来实现。

现在，我们将详细地讨论这两个领域。

8.1 版本控制系统

版本控制系统（VCS）提供了一种共享、同步和备份任何文件的方法。它们分为两类：

- 集中式系统；
- 分布式系统。

8.1.1 集中式系统

集中式版本控制系统基于单个服务器，这个服务器保存文件并且让客户记录和检查对这

些文件所做的修改。它的原理很简单：每个人都可以获得系统中文件的副本，并且在这份副本上工作。因此，每个用户可以向服务器提交（commit）他们的修改。这些修改将被应用并且提升修订（revision）号。其他用户可以通过一个更新（update）来同步他们的仓库（repository）副本，以获得这些修改。

仓库基于所有的提交来演化，系统将所有的修订归档到一个数据库，用以撤销任何修改或者提供所完成修改的信息，如图 8.1 所示。

在这个集中式配置中，每个用户负责同步自己的本地仓库和主仓库，以便获得其他用户所做的修改。这意味着，当一个本地修改过的文件被其他人修改并检入时，就会发生一些冲突。在这种情况下，将执行一个用户系统上的冲突解决机制，如图 8.2 所示。

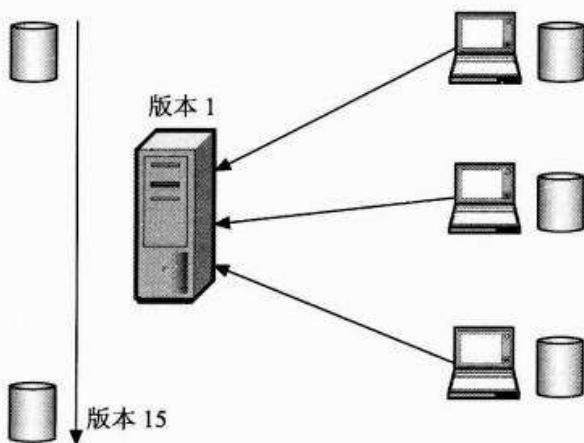


图 8.1

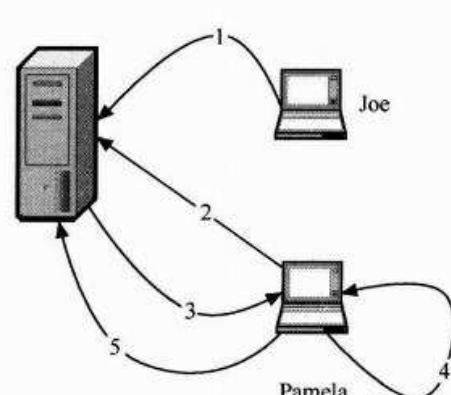


图 8.2

- (1) Joe 检入一个修改。
- (2) Pamela 试图检入对相同文件的一个修改。
- (3) 服务器提示她的文件已经过时。
- (4) Pamela 更新本地副本。版本控制软件可能（也可能不）可以无缝地合并这两个版本（也就是说，没有冲突）。
- (5) Pamela 提交新的版本，它将包含 Joe 和她自己所做的修改。

这个过程在涉及少数开发人员和少量文件的情况下能够很好地运作，但是对于较大的项目而言仍然有问题。例如，一个复杂的修改将涉及许多文件，这种方法很耗费时间，而且在整个工作完成之前一直在本地保持所有文件也是不可行的。

- 这很危险，因为用户可能在他们的计算机上保持没有必要备份的修改。
- 这样，在其检入之前是很难与其他人共享的；如果在完成之前共享则会使仓库处于不稳定状态，这样其他用户将不希望共享。

集中式的 VCS 通过提供“分支”和“合并”来解决这个问题。从主流的修订中找出独立的分支，工作完成之后将其交回到主流中，这样做是有可能的。

第 8 章

代码管理

当项目涉及多人的时候，工作会更加棘手。所有事情的速度都将减慢，也将更难。这可以归结于多个原因，本章将揭示这些原因，并尝试提供与之对抗的一些方法。

本章由两个部分组成，分别介绍：

- 如何使用版本控制系统；
- 如何配置持续集成机制。

首先，代码基准库有了非常大的发展，因此跟踪所有修改就显得更加重要，尤其是涉及许多开发人员的时候。这就是版本控制系统所担当的角色。

其次，几个没有直接相连的大脑为相同的项目工作。他们各有不同的角色，从事不同领域的工作。因此，缺乏全局可见性将产生许多与事情的进展和其他人所完成工作等方面混乱。这是不可避免的，必须通过一些工具来提供持续的可见性，并减轻这些问题。这可以通过配置一系列用于持续集成（continuous integration）的工具来实现。

现在，我们将详细地讨论这两个领域。

8.1 版本控制系统

版本控制系统（VCS）提供了一种共享、同步和备份任何文件的方法。它们分为两类：

- 集中式系统；
- 分布式系统。

8.1.1 集中式系统

集中式版本控制系统基于单个服务器，这个服务器保存文件并且让客户记录和检查对这

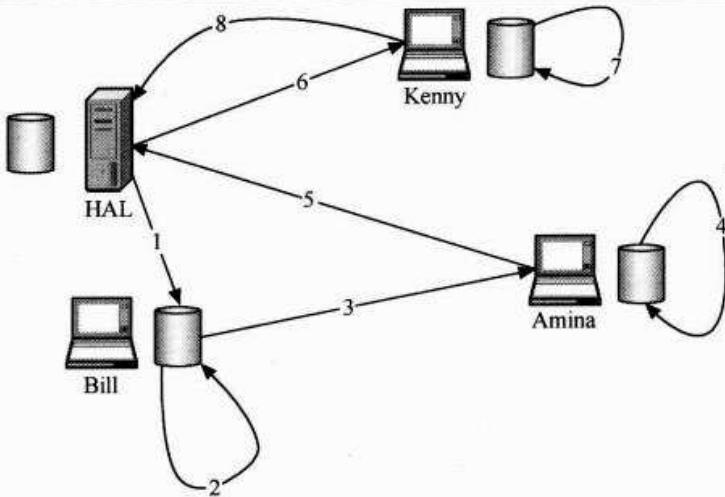


图 8.4

(8) Kenny 照常将修改推送给 HAL。

这里关键的概念是人们推送和拉取其他仓库中的文件，这种行为将根据人们工作和项目管理的方式而变化。因为没有主仓库，所以项目的维护者需要定义人们推和拉取程序修改的策略。

而且，人们在使用多个仓库时必须更加聪明一些。因为修订号对于每个仓库而言都是局部的，没有所有人都可以参考的全局修订号。因此，必须使用标记（tag）来辨别。标记是可以附加到修订中的标签。最后，用户负责备份自己的仓库，这不是集中式架构中管理员负责配置备份策略的方式。

分布式策略

如果供职于每个人都为同一目标工作的公司，那么 DVCS 仍希望拥有集中式服务器。

对此可以应用不同的方法。最简单的方法是设置一个和常规的集中式服务器相似的服务器，每个项目成员都可以向公共流中推送修改。但是这个方法有点过于简单，它无法利用分布式系统的优点，因为人们将会以与集中式系统相同的方法使用推送和拉取命令。

另一种方法是在一个服务器上提供多个不同访问级别的仓库：

- 一个不稳定（unstable）仓库，大家都可以推送修改；
- 一个稳定（stable）仓库，发行管理员之外的所有成员都是只读的，发行管理员有权从不稳定知识库拉取修改并决定所应合并的内容；
- 各种不同的只读的、对应于发行版本的发行仓库，本章稍后将会介绍。

这允许人们提交修改，管理员可以在它们进入稳定仓库之前进行审阅。

因为 DVCS 提供了无限的组合方式，所以还可以建立其他的策略。例如，Linux Kernel 使用了基于星型模式的 Git (<http://git.or.cz>)，Linus Torvalds 维护官方仓库，并从他所信任的一组开发人员那里拉取修改。在这种模式下，希望修改核心的人尝试将这些修改推送给受信

任的开发人员，这样他们有希望通过这些开发人员到达 Linus 那里。

8.1.3 集中还是分布

要在集中式和分布式方法之间做出选择，更多取决于项目的特性和团队的工作方式。

例如，一个由独立团队开发的应用程序就不需要分布式系统所提供的特性。一切都在开发服务器的控制之下，管理员不需处理外部的代码贡献者，也不用担心开发人员工作的备份。开发人员在需要时可以创建分支，然后尽快回到主干。他们在需要合并修改和离开 Internet 连接时都可能会碰到困难，但是对这样的系统所提供的功能仍然很享受。分支和合并在这种环境下不经常发生。

这就是大部分公司不涉及更广泛的代码贡献者社区的原因。它们自己的雇员大量使用集中的版本控制系统，每个人都在一起工作。

对于拥有大量代码贡献者的项目，集中式方法就显得有些呆板，使用 DVCS 则更有意义。现在有许多开源项目都选用了这种模式。例如，目前正在讨论为 Python 采用一个 DVCS，并且可能很快会发生，因为主要的问题在于建立一组良好的做法，并且培训开发人员使用这种新的代码处理方式。

本书中将使用 DVCS 并解释它在项目管理中的使用，以及一组良好的做法，选择的软件是 Mercurial。

8.1.4 Mercurial

Mercurial (<http://www.selenic.com/mercurial/wiki>) 是用 Python 编写的一个 DVCS，提供了简单但强有力的命令行实用程序来处理代码。

最简单的安装方式是调用 `easy_install`，命令如下。

```
$ easy_install mercurial
```

 在某些 Windows 版本下，在 Python Scripts 目录下生成的脚本是错误的，`hg` 在提示符下也不可用。这种情况下，可以把它改名为 `hg.py`，并在提示符下运行 `hg.py`。

如果仍然遇到问题，可以使用一个特殊的二进制安装程序（参见 <http://mercurial.berkwood.com>）。

如果在 Debian 或 Ubuntu 这样的系统下，也可以使用它提供的包管理系统，命令如下。

```
$ apt-get install mercurial
```

这样，在提示符下就能够调用 hg 脚本，它拥有一组完备的选项（在此有删节），如下所示。

```
$ hg -h
Mercurial Distributed SCM
list of commands:
add          add the specified files on the next commit
clone        make a copy of an existing repository
commit       commit the specified files
copy         mark files as copied for the next commit
diff         diff repository (or selected files)
incoming     show new changesets found in source
init         create a new repository in the given directory
pull         pull changes from the specified source
push         push changes to the specified destination
status       show changed files in the working directory
update      update working directory
use "hg -v help" to show aliases and global options
```

在要创建仓库的文件夹中调用 init 命令，就可以创建出所需的仓库，如下所示。

```
$ cd /tmp/
$ mkdir repo
$ hg init repo
```

现在，可以使用 add 命令来将文件添加到仓库中，如下所示。

```
$ cd repo/
$ touch file.txt
$ hg add file.txt
```

这个文件直到调用 commit (ci) 命令时才被检入，如下所示。

```
$ hg status
A file.txt
$ hg commit -m "added file.txt"
No username found, using 'tziade@macziade' instead
$ hg log
changeset: 0:d557683c40bc
tag:           tip
```

```

user:          tziade@macziade
date:         Tue Apr 01 17:56:41 2008 +0200
summary:      added file.txt

```

仓库在其目录中是自包含的，可以使用 clone 命令将其复制到另一个目录，如下所示。

```

$ hg clone . /tmp/repo2
1 files updated, 0 files merged, 0 files removed, 0 files unresolved

```

也可以通过 SSH 在另一个机器上完成这一任务，只要它安装了 SSH 服务器和 Mercurial，如下所示。

```

$ hg clone . ssh://tarek@ziade.org/repo
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changeset with 1 change to 1 files

```

远程仓库可以使用 push 命令来同步，如下所示。

```

$ echo more >> file.txt
$ hg diff
diff -r d557683c40bc file.txt
--- a/file.txt Tue Apr 01 17:56:41 2008 +0200
+++ b/file.txt Tue Apr 01 19:32:59 2008 +0200
@@ -0,0 +1,1 @@
+more
$ hg commit -m 'changing a file'
No username found, using 'tziade@macziade' instead
$ hg push ssh://tarek@ziade.org/repo
pushing to ssh://tarek@ziade.org/repo
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files

```

diff 命令 (di) 在这里用来显示所做的修改。

hg 命令提供的另一个很好的特性是 serve，它为当前仓库提供一个小的 Web 服务器：

```
$ hg serve
```

现在，可以在浏览器中访问`http://localhost:8000`，将会得到一个仓库的视图，如图 8.5 所示。

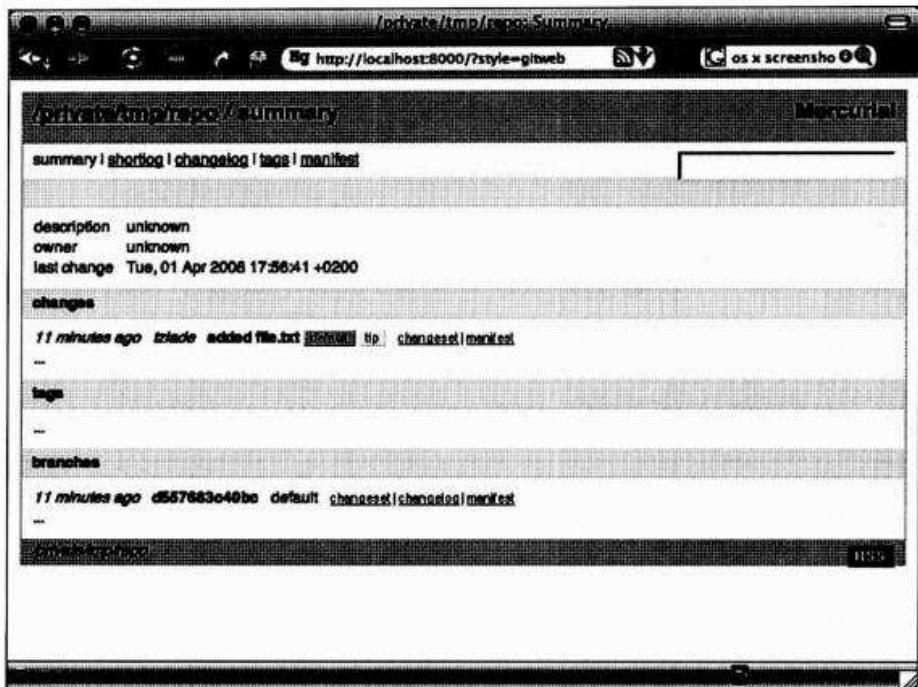


图 8.5

在这个视图之外，hg serve 还能够调用 clone 和 pull 命令，如下所示。

```
$ hg clone http://localhost:8000 copy
requesting all changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ cd copy/
$ ls
file.txt
$ touch file2.txt
$ hg add file2.txt
$ hg ci -m "new file"
No username found, using 'tziade@macziade' instead
```

clone 命令可以复制一个仓库来开始工作。

hg serve 不允许人们推送修改，因为这需要建立一个真正的 Web 服务器来处理验证，正如下一小节中所要看到的那样。但是在某些情况下，如果希望暂时共享一个仓库来让别人拉

取修改时，这可能是有用的。



如果想更深入地研究 Mercurial，可以参考 <http://hgbook.red-bean.com> 上的联机书籍。

8.1.5 使用 Mercurial 进行项目管理

使用 Mercurial 管理仓库的最简单方法是使用其提供的 hgwebdir.cgi 脚本。这是一个 CGI（公共网关接口）脚本，可以用来通过 Web 服务器发布一个仓库，并提供和 hg serve 相同的功能。而且，通过配置一个 mima 文件限制命令的使用，它允许 push 命令以安全的方式运行。



CGI 很健壮而且配置很简单，但是它并非发布一个仓库的最快方式。还有一些基于 fastcgi 或 mod_wsgi 的解决方案。

配置这样一个系统并不困难，但是可能取决于平台相关的部件，所以无法提供一个通用的安装教程。本小节将关注于在 Linux Debian Sarge 和 Apache 2 平台上的配置方法，这是相当常见的配置。

安装这样一个服务器的步骤如下：

- 建立一个专用文件夹；
- 配置 hgwebdir；
- 安装 Apache；
- 设置权限。

1. 建立一个专用文件夹

先前描述的多仓库方法用 Mercurial 来配置很简单，因为一个仓库对应于一个系统文件夹。可以建立一个 repositories 文件夹来保存所有的知识库，这个文件夹位于项目专用的文件夹中。这个项目文件夹位于主文件夹中。这里可以使用 mercurial 用户。

为 Atomisator 创建一个 Mercurial 环境，如下所示。

```
$ sudo adduser mercurial
$ sudo su mercurial
$ cd
```

```
$ mkdir atomisator
$ mkdir atomisator/repositories
$ cd atomisator
```

由此，可以使用 hg 创建 stable 和 unstable 仓库，如下所示。

```
$ hg init repositories/stable
$ hg init repositories/unstable
$ ls repositories/
unstable stable
```



一些团队不使用分离的仓库，但是在一个单一的知识库上，他们会使用一个命名的分支来区分稳定版本和开发版本，并且进行一些必需的合并。具体请参见 <http://www.selenic.com/mercurial/wiki/index.cgi/Branch>。

创建发行版本时，可以通过复制一个 stable 来添加新的仓库。例如，如果发行了 0.1 版本，可以如下这样完成：

```
$ hg clone repositories/stable repositories/release-0.1
```

将前一章中创建的 buildout 和 packages 文件夹复制到 unstable 文件夹中，并且检入它们，以在不稳定仓库中添加 Atomisator 代码。完成这一步之后，unstable 文件夹的内容看上去类似于：

```
$ ls repositories/unstable
buildout packages
```



下一章将介绍与发行相关的知识。

2. 配置 hgwebdir

为了提供这些仓库，必须将 hgwebdir.cgi 文件添加到 atomisator 文件夹中。这个脚本将随同安装一起提供。如果找不到，可以在 Mercurial 网站上下载一个源代码分发版本。但是要确定下载的文件和安装的版本严格对应。

```
$ hg --version
Mercurial Distributed SCM (version 0.9.4)
$ locate hgwebdir.cgi
```

```
/usr/share/doc/mercurial/examples/hgwebdir.cgi
$ cp /usr/share/doc/mercurial/examples/hgwebdir.cgi .
```

这个脚本处理一个名为 hgweb.config 的配置文件，它包含仓库文件夹的路径，如下所示。

```
[collections]
repositories/ = repositories/
[web]
style = gitweb
push_ssl = false
```

collections 小节提供指定包含多个仓库的文件夹的通用方法，该脚本将循环访问这些仓库。Web 小节可用来设置一些选项。在本例中，可以设置其中两个：

- style 用来设置网页的外观和风格，gitweb 可能是最好的默认值。注意，Mercurial 使用模板来显示所有页面，而这些模板都可以配置；
- push_ssl 如果设置为 true（默认值），那么用户将不能通过 HTTP 使用推送命令。

3. 配置 Apache

下一步要配置的是执行 CGI 脚本的 Web 服务器层。最简单的方法是在 atomisator 文件夹中提供一个配置文件，定义一条 Directory、一条 ScriptAliasMatch 和一条 AddHandler 指令。

添加一个包含以上内容的 apache.conf 文件，如下所示。

```
AddHandler cgi-script .cgi
ScriptAliasMatch ^/hg(.*) /home/mercurial/atomisator/
hgwebdir.cgi\$1
<Directory /home/mercurial/atomisator>
    Options ExecCGI FollowSymLinks
    AllowOverride None
    AuthType Basic
    AuthName "Mercurial"
    AuthUserFile /home/mercurial/atomisator/passwords
    <LimitExcept GET>
        Require valid-user
    </LimitExcept>
</Directory>
```

注意：

- AddHandler 指令在某些发行版本上可能是不必要的，但在 Debian Sarge 中必须有；

- ScriptAliasMatch 需要启用 mod_alias;
- 当遇到 POST 指令时，意味着用户将向服务器发送数据，需要使用一个密码文件完成一次验证。



如果对 Apache 不熟悉，请参见<http://httpd.apache.org/docs>。

密码文件由实用程序 htpasswd 生成，并被放在 atomisator 文件夹中，如下所示。

```
$ htpasswd -c passwords tarek
New password:
Re-type new password:
Adding password for user tarek
$ htpasswd passwords rob
New password:
Re-type new password:
Adding password for user rob
```



在 Windows 平台中，如果 htpasswd 在命令提示符下不可用，那么可能需要在 PATH 中手工添加位置。

每当需要添加一位有权限将修改推送到仓库的用户时，都可以用 htpasswd 升级这个文件。最后，为了使该脚本有权限执行以及确保数据可用于 Apache 进程所使用的用户组，还需要以下几个步骤。

```
sudo chmod +x /home/mercurial/atomisator/hgwebdir.cgi
sudo chown -R mercurial:www-data /home/mercurial/atomisator
sudo chmod -R g+w /home/mercurial/atomisator
```

为了与配置挂钩，可以将这个文件添加到 Apache 所访问的 site-enabled 目录中，如下所示。

```
$ sudo ln -s /home/mercurial/atomisator/apache.conf /etc/apache2/sites-enabled/
007-atomisator
$ sudo apache2ctl restart
```

当 Apache 启动之后，可以通过<http://localhost/hg>访问该页面（如图 8.6 所示）。

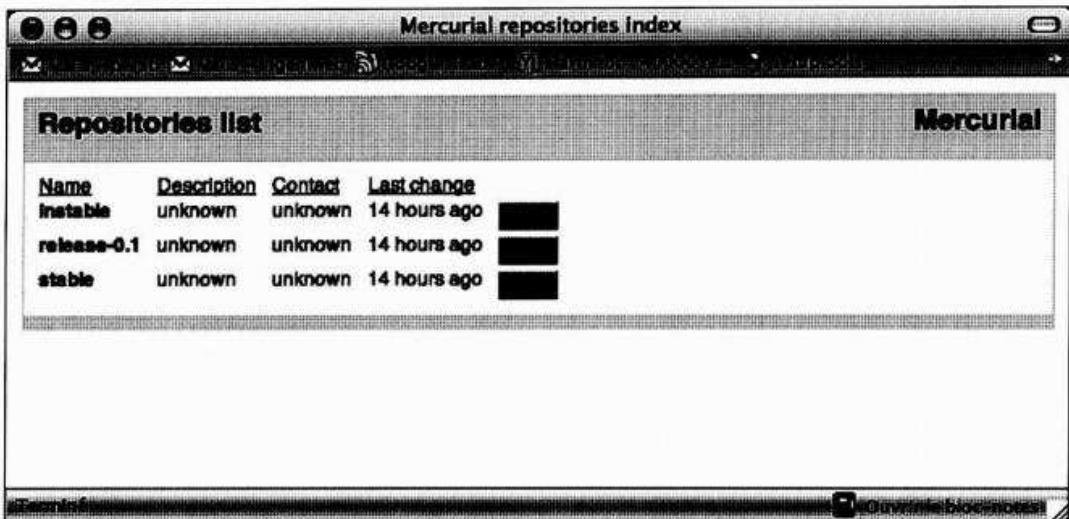


图 8.6

注意，每个仓库都有一个 RSS feed，人们可以通过来它来跟踪修改。每当有人推送一个文件时，就有在 RSS feed 中添加一个新的条目，以及一个指向修改日志的链接。这个修改日志将以一个不同的视图来显示更详细的日志。

如果需要将 Mercurial 仓库作为一个虚拟主机，将需要添加一个特殊的改写规则，这个规则为 hgwebdir 所用的静态文件（诸如样式单）提供服务。

以下是 Web 上用于发行本书的仓库（包含实例的源代码）的 apache.conf 文件，对应于 <http://hg-atomisator.ziade.org/>。

```
<VirtualHost *:80>
    ServerName hg-atomisator.ziade.org
    CustomLog /home/mercurial/atomisator/access_log combined
    ErrorLog /home/mercurial/atomisator/error.org.log
    AddHandler cgi-script .cgi
    RewriteEngine On
    DocumentRoot /home/mercurial/atomisator
    ScriptAliasMatch ^/(.*) /home/mercurial/atomisator/hgwebdir.cgi/$1
<Directory /home/mercurial/atomisator>
    Options ExecCGI FollowSymLinks
    AllowOverride None
    AuthType Basic
    AuthName "Mercurial"
    AuthUserFile /home/mercurial/atomisator/passwords
```

```
<LimitExcept GET>
    Require valid-user
</LimitExcept>
</Directory>
</VirtualHost>
```

每个仓库都可以从这个首页链接到。为了让所有页面都使用相同的样式，每个仓库的.hg 配置目录都将添加一个 hgrc 文件。这个文件能够定义和主 CGI 文件所使用的类似的 Web 小节，如下所示。

```
$ more repositories/stable/.hg/hgvc
[web]
style = gitweb
description = Stable branch
contact = Tarek <tarek@ziade.org>
```

description 和 contact 字段也将用于该网页。

4. 设置权限

我们已经看到一个用来过滤允许推送的用户的全局访问文件。这是第一级的权限，因为需要为每个仓库定义 push 策略。先前所定义的策略是：

- 所有注册的开发人员都允许将修改推送到不稳定（unstable）仓库；
- 除了发行管理员之外，其他用户对稳定（stable）知识库有只读权限。

这可以在每个仓库的 hgrc 文件的 allow_push 参数中进行设置。如果用户 tarek 是发行管理员，那么 stable hgrc 文件的内容将类似于：

```
$ more repositories/stable/.hg/hgvc
[web]
style = gitweb
description = Stable branch
contact = Tarek <tarek@ziade.org>
push_ssl = false
allow_push = tarek
```

注意，这里添加的 push_ssl 是用于通过 HTTP 推送的。不稳定（unstable）仓库的 hgrc 文件类似于：

```
$ more repositories/unstable/.hg/hgvc
[web]
```

```

style = gitweb
description = Unstable branch
contact = Tarek <tarek@ziade.org>
push_ssl = false
allow_push = *
```

这意味着，只要将他们添加到密码文件中，这个仓库允许所有人执行推送操作。

 本书中为了简单起见没有设置 SSL 配置，但是在实际服务器中，为了使服务更安全，应该使用它。例如，在我们的配置中，是允许 HTTP 嗅探的。

5. 设置客户端

为了避免鉴权提示，以及在提交日志时提供一个易于理解的名称，在客户端的 HOME 目录中可以添加一个.hgrc 文件，如下所示。

```
[ui]
username = Tarek Ziade
[paths]
default = http://tarek:secret@atomisator.ziade.org/hg/unstable
unstable = http://tarek:secret@atomisator.ziade.org/hg/unstable
stable = http://tarek:secret@atomisator.ziade.org/hg/stable
```

ui 小节为服务器提供一个提交者的全名，paths 小节则是一个仓库的 URL 列表。注意，在 URL 中放置用户名和密码，以避免在每次推送完成时出现密码提示。这种做法并不安全，有密码提示会更安全些。最安全的方式是通过 SSH 协议而不是 Web 服务器来工作。

使用这个文件，推送可以这样完成，如下所示。

```
$ hg push # will push to the default repository (unstable)
$ hg push stable # will push to stable
$ hg push unstable # will push to unstable
```

 如果需要在其他平台上安装，安装步骤不会有很大的不同。针对特定的平台，<http://www.selenic.com/mercurial/wiki/index.cgi/HgWebDirStepByStep> 上的内容将会有所帮助。

8.2 持续集成

对于持续集成而言，建立一个仓库只是第一步，持续集成是从极限编程（XP）中发展而来的一组软件方法。在 Wikipedia 上对其原理有清晰的描述 (http://en.wikipedia.org/wiki/Continuous_integration#The_Practices)，并且定义了确保软件易于构建、测试和交付的方法。

下面总结一下在使用 `zc.buildout` 和 Mercurial 的、基于 egg 的应用程序环境中的这些方法。

- 维护一个代码仓库 这由 Mercurial 完成。
- 自动化构建 `zc.buildout` 可以满足这个需求，正如在前一章中所看到的那样。
- 使所构建的成为自测试系统 `zc.buildout` 提供了启动整个软件测试活动的一种方法。
- 每个人每天提交新修改 Mercurial 为开发人员提供经常修改的工具，但是这更多需要依赖开发人员的行为，只要不破坏构建版本，应该尽可能地经常提交。
- 每个提交都应该被编译 每当对程序做了修改时，软件应该再次编译并运行所有测试，以确认没有导致退化。如果发生了这样的问题，应该向开发人员发出一封警告邮件（本章中尚未介绍）。
- 保持快速的编译 这对于 Python 应用程序而言不是一个真正的问题，因为大部分时候都不需要编译的步骤。在任何情况下，连续编译两次软件时，第二遍应该会更快一些。
- 在一个复制生产环境的临时环境中测试 能在所有生产环境中测试软件是很重要的，本章中尚未介绍这方面的内容。
- 简化获得最新可分发版本的过程 `zc.buildout` 提供了简单的方法，能够将可分发版本捆绑到档案文件中。
- 每个人都可以了解最新构建的结果 系统应该提供关于构建的反馈信息，本章中尚未介绍这方面的内容。

使用这些方法，能在早期发现问题，甚至是那些与代码或与目标平台相关的问题，从而提升代码质量。

而且，编译和重新执行测试的自动化系统简化了开发人员的工作，因为他们不需要再次启动整个测试。

最后，这些规则将使开发人员对他们提交的修改更负责任。检入有破坏性的代码将产生大家都能看到的反馈。

在我们配置的环境中，还没有涉及的部分只有：

- 在每次提交时编译该系统；
- 在目标系统上编译该系统；
- 提供关于最新构建版本的反馈。

这可以使用自动化构建软件 Buildbot 来实现。

Buildbot

Buildbot (<http://buildbot.net/trac>) 是用 Python 编写的软件，它自动化了任何软件项目的编译和测试周期。它可以配置成每当源代码仓库变化时就生成某些构建版本，并启动某些测试，然后提供一些反馈，如图 8.7 所示。

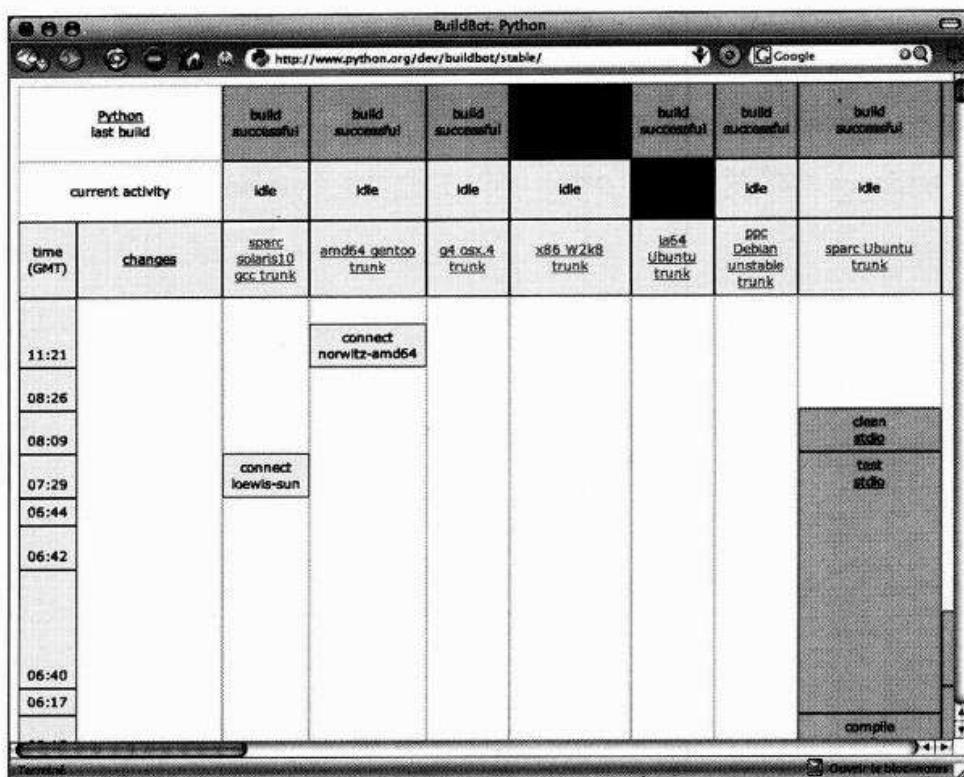


图 8.7

例如，Python 就在核心开发中使用了这个工具，参见 <http://www.python.org/dev/buildbot/stable/>（别忘了在最后要加上“/”）。

每一列对应一个由步骤(step)组成的一个构建(build)，并与一些构建客户端(build slaves)相关联，整个系统由构建服务器端(build master)驱动。

- 构建服务器端集中并且驱动所有工作。
- 一个构建(build)是用于构建应用程序并且运行测试的步骤序列。
- 步骤(step)是一个原子命令，例如：检查项目文件；构建应用程序；运行测试。

构建客户端是负责执行一个构建的机器，它可以位于任何地方，只要能够访问构建服务器。

1. 安装 Buildbot

要安装 Buildbot，需要安装一系列它所需的软件，并创建一个配置 Buildbot 的 Python 脚

本。可在<http://buildbot.net/trac/wiki/UserManual>上的联机用户手册中找到相关描述。

另一个选项是使用 `collective.buildbot` 项目，它提供了一个基于 `zc.buildout` 的配置工具。换句话说，它可以在一个配置文件中定义 `Buildbot`，而不需要考虑所有必需软件的安装和 Python 脚本的编写。

在自己的服务器环境中创建这样一个 `buildout`，此外还有在专用文件夹中的仓库，如下所示。

```
$ cd /home/mercurial/atomisator
$ mkdir buildbot
$ cd buildbot
$ wget http://ziade.org/bootstrap.py
```

然后，在 `buildbot` 文件夹中添加一个包含以下内容的 `buildout.cfg` 文件。

```
[buildout]
parts =
    buildmaster
    linux
    atomisator
[buildmaster]
recipe = collective.buildbot:master
project-name = Atomisator project buildbot
project-url = http://atomisator.ziade.org
port = 8999
wport = 9000
url = http://atomisator.ziade.org/buildbot
slaves =
    linux          ty54ddf32
[linux]
recipe = collective.buildbot:slave
host = localhost
port = ${buildmaster:port}
password = ty54ddf32
[atomisator]
recipe = collective.buildbot:project
slave-names = linux
repository=http://hg-atomisator.ziade.org/unstable
vcs = hg
build-sequence =
```

```

./build
test-sequence =
    buildout/bin/nosetests
email-notification-sender = tarek@ziade.org
email-notification-recipient = tarek@ziade.org
[poller]
recipe = collective.buildbot:poller
repository=http://hg-atomisator.ziade.org/unstable
vcs = hg
user=anonymous

```

这定义了一个构建服务器，以及一个构建客户端和一个 Atomisator 项目。这个项目定义了一个可供调用的 build 脚本和一个运行项目中的测试运行程序的测试序列。

 关于选项的补充信息，可以浏览 <http://pypi.python.org/pypi/collective.buildbot>.

在 build-sequence 中引用的 build 脚本必须被添加到仓库的根目录下，其内容如下。

```

#!/bin/sh
cd buildout
python bootstrap.py
bin/buildout -v

```

别忘了在推送之前设置执行标志，如下所示。

```

$ chmod +x build
$ hg add build
$ hg commit -m "added build script"

```

现在就可以执行 buildout 了，如下所示。

```

$ python bootstrap.py
$ bin/buildout -v

```

 bootstrap.py 是一个小脚本，用以确保系统能够满足构建 buildbot 的要求。

应该在 bin 文件夹中得到两个脚本：一个用来启动构建服务器，另一个用来构建客户端。它们用 buildout 小节来命名，现在来运行它们，如下所示。

```
$ bin/buildmaster.py start
Following twistd.log until startup finished..
2008-04-03 16:06:49+0200 [-] Log opened.
...
2008-04-03 16:06:50+0200 [-] configuration update complete
The buildmaster appears to have (re)started correctly.
$ bin/linux.py start
Following twistd.log until startup finished..
The buildslave appears to have (re)started correctly.
```

现在，在浏览器的地址栏中输入 `http://localhost:9000` 就能访问 Buildbot。然后，单击 atomisator 链接来强制指定一个构建以控制整个系统。

2. 整合 Buildbot 和 Mercurial

完成这个配置还有一步：将仓库提交事件与 Buildbot 挂钩起来。这样，每当有人将文件推送到仓库时会自动重新构建，这可以由 Buildbot 自带的 `hgbuldbot.py` 脚本来完成。

要将它用作一个命令，只需在 `pyp.buildbotenv` 之上运行 `easy_install`。这将安装该脚本并确保同时安装 Buildbot 和 Twisted，如下所示。

```
$ easy_install pyp.buildbotenv
```

然后，在 `.hg` 文件夹中的 `unstable hgrc` 文件 (`/path/to/unstable/.hg/hgrc`) 中添加钩子，如下所示。

```
[web]
style = gitweb
description = Unstable branch
contact = Tarek <tarek@ziade.org>
push_ssl = false
allow_push = *

[hooks]
changegroup.buildbot = python:buildbot.changes.hgbuldbot.hook
[hgbuldbot]
master = atomisator.ziade.org:8999
```

`hooks` 小节将链接 `hgbuldbot` 脚本，`hgbuldbot` 小节则定义主服务器和客户端口。

3. 整合 Apache 和 Buildbot

现在,可以在 Apache 中添加一条改写规则,使得无需指定 9000 端口就能够访问 Buildbot。最简单的方法是创建一个特殊的虚拟主机,并将它添加到 Apache 配置文件集中,如下所示。

```
<VirtualHost *:80>
    ServerName atomisator-buildbot.ziade.org
    CustomLog /var/log/apache2/bot-access_log combined
    ErrorLog /var/log/apache2/bot-error.org.log
    RewriteEngine On
    RewriteRule ^(.*) http://localhost:9000/$1
</VirtualHost>
```

8.3 小 结

本章中介绍了以下内容:

- 集中式和分布式版本控制系统之间的不同;
- Mercurial 的使用方法,这是一个很好的分布式版本控制系统;
- 多仓库策略的设置和使用;
- 持续集成的概念;
- 如何设置 Buildbot 和 Mercurial 以提供持续集成支持。

下一章中将说明如何使用迭代和增量方法来管理软件生命周期。

第 9 章

生命周期管理

软件开发管理是很难的，项目常常被推迟交付，甚至还经常被取消。现代软件管理已经创建了大量降低风险的方法，最常用的、已被证明是有效的方法是迭代开发。有许多使用迭代方法的理论，它们通常被称为敏捷方法（Agile methodologies）。

本章不是完整的软件管理指南，因为这个主题需要整整一本书的篇幅（可以参考由 Addison-Wesley 出版的 *Agile and Iterative Development: A Manager's Guide* 一书。）

本章将给出一些关于如何基于迭代管理软件生命周期，以及如何使用少数工具来完成这一工作的技巧和总结。

9.1 不同的方法

在介绍迭代生命周期之前，先介绍几种软件业现有的开发模型。

9.1.1 瀑布开发模型

瀑布开发模型将软件视为一个整体，每个阶段都将在前一个阶段完成以后才开始。换句话说，所有工作将被组织为一系列阶段，通常包括：

- 需求分析；
- 总体架构及各软件部件组织形式的设计；
- 设计每个部件；
- 使用 TDD（测试驱动开发）方法编码每个部件（有些人不使用 TDD 方法）；
- 整合各部件进行整体系统测试；

- 部署。

因此，整个工作的组织看上去像一个瀑布，如图 9.1 所示。

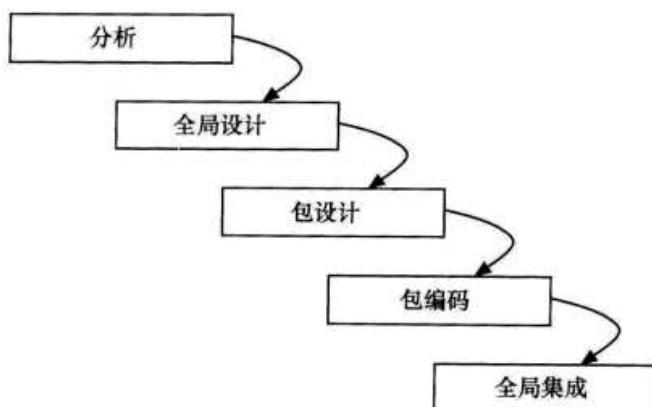


图 9.1

许多大公司都在使用这个模型，认真完成每个步骤并且进行评审，然后才开始下一步。这意味着设计完成之后，开发人员只需要实现这个设计。最后一步是将所有部件整合起来并进行总体测试。

这个模型很死板，因为在整体测试之前几乎不可能考虑到软件的所有方面。实际上，在最后一步常常会发现一些不一致或部件缺失，甚至因为设计缺陷而产生性能问题。

这样一个模型可能更适用于非常明确的目标环境以及一个有经验的团队，但对于大部分软件而言，这是不可能的。

9.1.2 螺旋开发模型

螺旋模型是基于原型之上的反馈。如图 9.2 所示，程序的第一个版本是根据原始需求创建的，不做任何修正。然后，这个原型根据接收到的反馈进行细化，程序的弱点和优点都能够精确地定位，这样它就可以被重构。

在几个周期之后，当所有人都认为原型符合需求时，就对其进行最后的细化并交付。

这个模型大大减少了风险，因为开发人员可以很早就开始软件编码，而不用像瀑布模型那样要先进行复杂的设计。在第一个周期之后，管理人员就能够对项目所需时间有概要性的了解。

9.1.3 迭代开发模型

迭代模型与螺旋模型相似，但它不把应用程序视为一个整体，而是关注于为系统增添一些新的功能，并且通过反馈来对原有的软件进行再造。

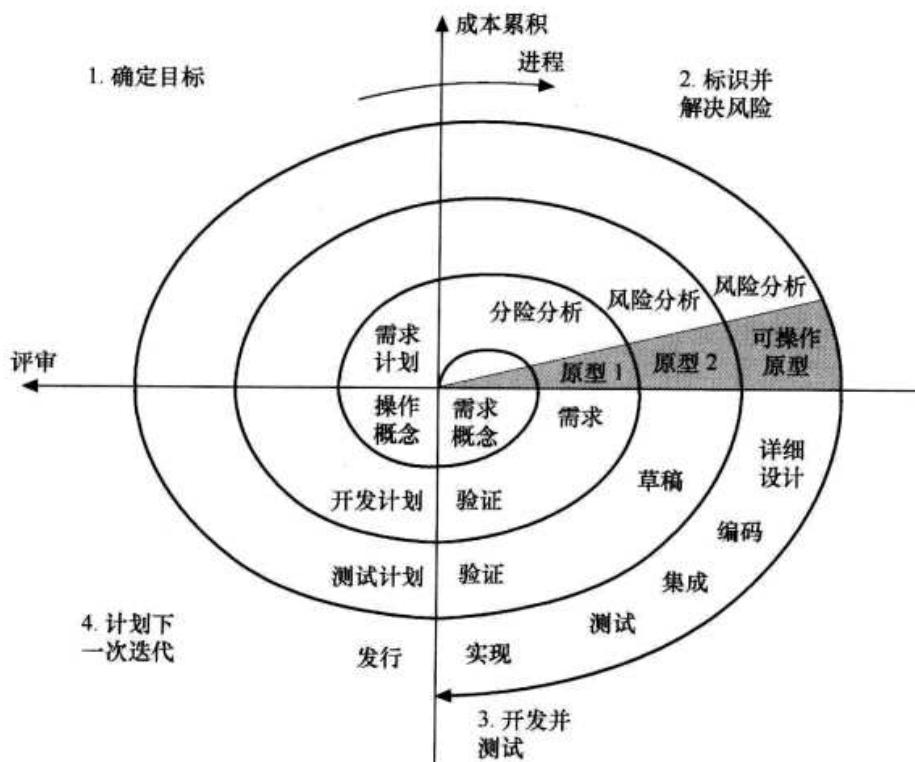


图 9.2

这意味着它和螺旋模型、瀑布模型都不同，在整个项目周期内都将存在分析、设计和编码工作，因为它们在每个周期中只关心功能的一个子集。

因此，一个项目被分成多个可以作为独立项目的迭代，在每个迭代中设计、构建和分发软件。所有涉及的团队，将通过跨学科的工作聚焦于当前迭代的功能。

在每个迭代中提供反馈，对于所有涉及的人员顺利地协同工作是很有帮助的。可以对每个功能在完成之前做很大的改进，往往与开始时的想法不一样。每个人都能从中获得一些思路，并且可以在下一个迭代中修正分析和设计。

因为这种渐进式方法关注于系统中更小的部件，所以提高了反馈的频度。一个迭代往往历时 1~4 周，软件在交付之前需要进行多个迭代。因为整个软件在每个迭代之后都将有所增长，所以它是增量式构建起来的。

基于迭代方法的理论很多，比如 Scrum 或 XP，这是敏捷方法的共同基础。

然而，迭代方法不是懒于设计的借口，它是有代价的。为了保持灵活性，敏捷的编程人员非常强调测试。通过编写测试，他们确保不会在修改时破坏原有的代码。

迭代方法中的经验法则是为每个迭代定义一个固定的长度，并且在结束时建立某些特殊的东西。

本章将提出一个作为许多开源项目共同基础的通用模型，并将通过一组可用于这一目标的工具来描述它。

9.2 定义生命周期

为一个项目定义生命周期，主要包括计划、常规发行和保持进度两部分工作。在周期结束时，推迟实现某些未就绪的元素，以便跟上承诺的计划进度往往更好。这被称为火车方法：当火车离开的时候，要么上车，要么呆在那里。但是，有些项目会推迟发行日期而不删除某些功能，而这种方法往往更难预测。

项目开始之后，通过一个初始化阶段来定义整体计划。通常的工作是大家参加一个启动会议，共同定义软件交付时间，然后将剩余的时间分到各个迭代之中。

根据软件的特性，这些迭代可以在一至四周之内完成。每次迭代应该具有相同的长度，以便在整个项目中保持相同的节奏，如图 9.3 所示。

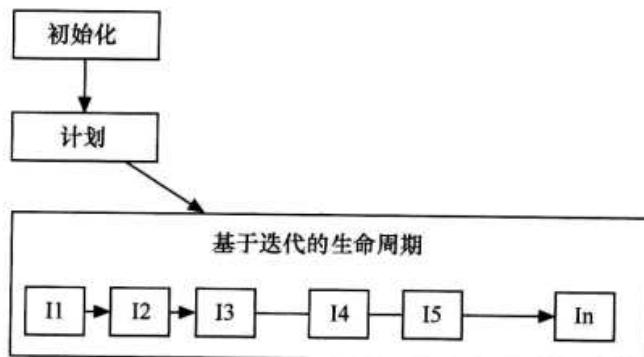


图 9.3

迭代长度根据项目达到稳定状态的时间而变化，它可能持续很长时间。

一次迭代是一个小的独立项目，它可能由以下 4 个阶段组成（如图 9.4 所示）：

- 计划阶段——定义所要做的工作；
- 分析、设计和测试驱动开发阶段——完成所要做的工作；
- 清理阶段——测试完成的工作并调试；
- 发行阶段——交付完成的软件。

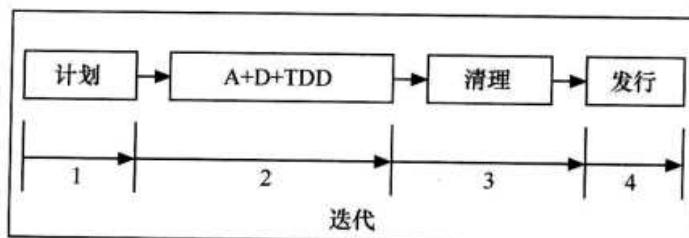


图 9.4

对于持续 2 周的一个迭代，也就是 10 个工作日，每个阶段的时间可以是：

- 1天用于计划；
- 7天用于开发；
- 1~2天用于整体清理；
- 1天用于发行。

9.2.1 计划

迭代的计划阶段将定义一系列必须完成的任务。一个任务是一个开发者所能完成的针对代码的一个原始操作。在给定剩余时间和开发人员数量的前提下，估计每个任务的周期以确定工作量是现实的。

这个计划由管理人员基于开发人员的反馈来完成，所有估计必须由实际工作的人来验证其正确性。在这个步骤的最后，迭代应该通过一个完整的任务列表被清晰地定义。

9.2.2 开发

开发阶段由每个开发人员所执行的任务组成，包括：

- 接受任务，每个人了解所需要处理的工作；
- 评审并改正预计的时间，这样管理人员知道估算是否准确，这个评审对于知道迭代的工作负荷是否现实也很重要；
- 编码；
- 完成时关闭任务；
- 以同样的方式执行另一个任务。

9.2.3 整体调试

整体调试阶段将关闭整个迭代，测试整个软件并完成剩余的任务。

执行最后这个步骤的最有效方法是聚集所有开发人员和管理人员，一起参加一个特殊的会议。

不能完成的任务将被推迟到下一次发行，在这个最后的阶段总能看到一些被掌声打断的展示。

9.2.4 发行

发行阶段由以下工作组成：

- 标记代码并创建发行，就像前一章中介绍的那样；
- 启动一个新的迭代。

9.3 建立一个跟踪系统

计划阶段定义了迭代内所需完成的任务。为了保持对这些任务的跟踪，可以使用一个跟踪系统。

这样的应用程序用来维护每个任务的相关信息，诸如：

- 负责人；
- 任务特性，包括新功能、调试、重构等；
- 到期日；
- 计划迭代；
- 状态，包括未解决、已完成等；
- 预计费用。

每个任务可以被放到一个迭代中，软件需要提供迭代的整体信息。在迭代的开发阶段，每个任务的状态由开发人员更新，这样就可以跟踪整体的状态。

9.3.1 Trac

Trac 是解决此类问题的一个很好的候选者。这个基于 wiki 的 Web 应用程序是个功能完整的问题跟踪系统，还提供了许多有用的功能（如图 9.5 所示）：

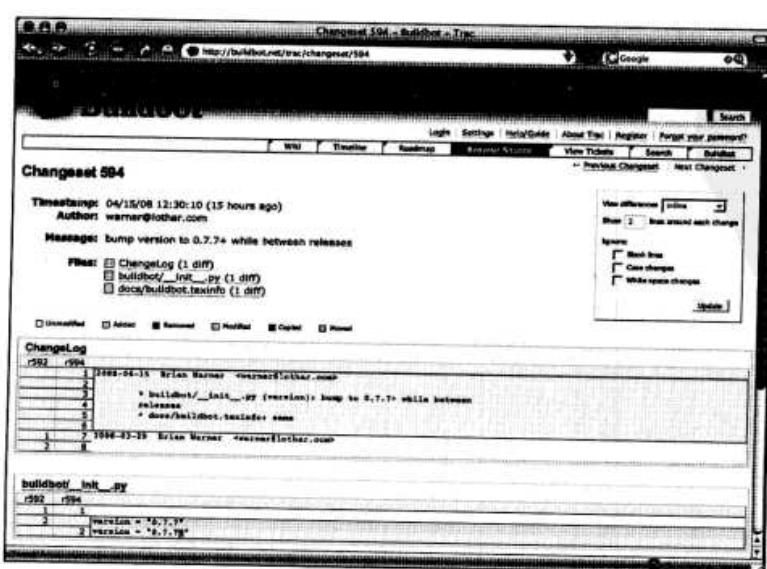


图 9.5

- 用户管理系统；
- 完全可编辑的基于 wiki 的界面；
- 允许为软件添加新功能的插件系统。

Trac 可以通过连接仓库和 Web 界面的插件来与大部分版本控制系统顺利地进行交互。现在可以通过 Web 浏览仓库树，用一条实时的时间线来显示提交，还有易于理解的 diff 报告。

在开放源码社区中，这个工具经常被用做项目网站，它为开发人员提供了处理代码所需的所有功能。以下是使用 Trac 的软件的应用例子。

- Plone <http://dev.plone.org/plone>
- Buildbot <http://buildbot.net/trac>
- Adium <http://trac.adiumx.com/>



Trac 维护一个 Who Uses Trac?(谁使用 Trac?)页面 <http://trac.edgewall.org/wiki/TracUsers>。如果项目使用了它，可以将项目信息添加到该页面中。

它的界面风格非常简洁，也十分易于理解和使用。

1. 安装

PyPI 上名为 `pyp.recipe.trac` 的 recipe 为配置一个与 Mercurial 服务器集成的 Trac 实例提供了一种便捷的方法。

在前一章中为 Buildbot 创建的 `buildbot.cfg` 文件里，可以添加一个新的 trac 小节，如下所示。

```
[buildout]
parts =
    buildmaster
    linux
    trac

[buildmaster]
...
[buildslave]
...
[trac]
recipe = pyp.recipe.trac
project-name = Atomisator
project-url = ${buildmaster:project-url}
repos-type = hg
```

```

repos-path = /home/mercurial/atomisator/repositories/unstable
buildbot-url = http://buildbot-atomisator.ziade.org/
components =
    atomisator.db      tarek
    atomisator.feed    tarek
    atomisator.main    tarek
    atomisator.parser  tarek
    pbp.recipe.trac   tarek
header-logo = atomisator.png

```

这个新小节将定义：

- 用做标题的 project name (项目名称);
- 在 Trac 实例中用做主页的 project url (项目 URL);
- 用以安装正确插件的 repository type (仓库类型), 在本示例中是 hg;
- 指向仓库 (必须在同一个服务器上) 的 repository path (仓库路径);
- 链接到 Trac 导航栏上的导航按钮的 Buildbot url;
- 将用于问题跟踪器的 component list (组件列表), 带有组件名称和所有者;
- 指向一个图像的 header-logo, 这个图像将被用来替代标题中的 Trac 图标。

在本示例中, 图标将被放置在 buildout 文件夹中。

再次运行专用于 Buildbot 和 Trac 的 buildout 实例, 如下所示。

```

$ bin/buildout -v
...
Project environment for 'Atomisator' created.
...
Try running the Trac standalone web server `tracd`:
  tracd --port 8000 /home/mercurial/atomisator/buildbot/parts/trac
...
Creating new milestone 'future'
Creating new component 'atomisator.db'
Creating new component 'atomisator.feed'
Creating new component 'atomisator.main'
Creating new component 'atomisator.parser'

```

这样, 就在 parts/trac 中添加了一个 Trac 环境, 在 bin 目录中添加了两个新的脚本:

- tracd 用于运行 Trac 实例的独立 Web 服务器;
- trac-admin 可以用来管理该实例的命令行 shell 程序。

尝试在 buildout 文件夹运行 tracd 脚本, 如下所示。

```
$ bin/tracd --port 8000 parts/trac  
Server starting in PID 24971.  
Serving on 0.0.0.0:8000 view at http://127.0.0.1:8000/
```

Trac 实例应该可以在浏览器中以 <http://127.0.0.1:8000/trac> 访问到，如图 9.6 所示。

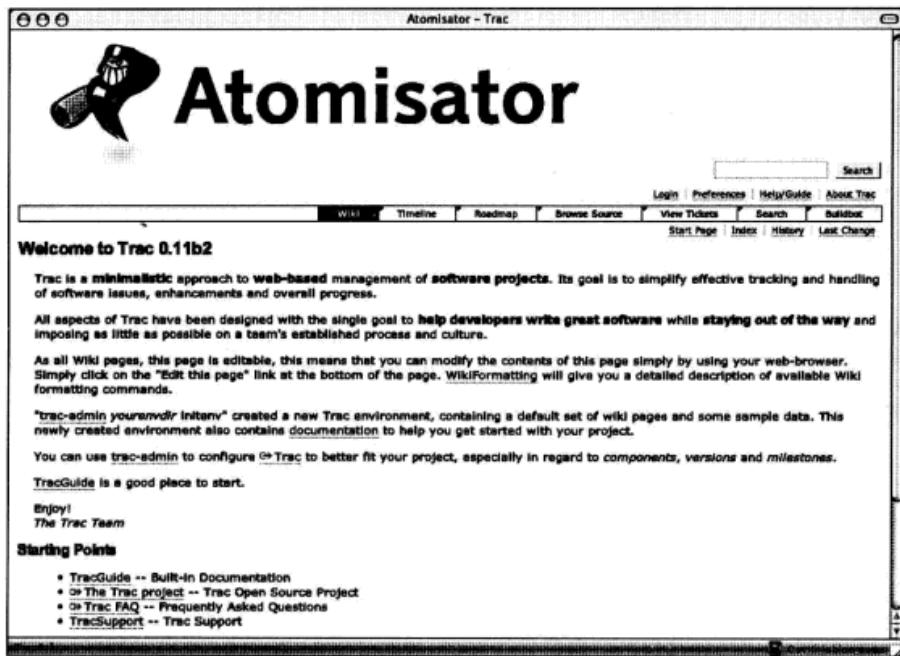


图 9.6

注意，通过 Buildbot 按钮可以访问 Buildbot 网页。

2. Apache 设置

和 Buildbot 类似，Trac 可以通过多个句柄与 Apache 挂钩。最简单的方法是使用 mod_python 句柄，这可以连接到 Trac 提供的前端上。

可以用以下命令将 mod_python 安装到 Debian Linux 上。

```
$ sudo apt-get install libapache2-mod-python
```

对于其他平台，可以参考项目页面<http://www.modpython.org>。

现在，可以在 Apache 配置中添加新的主机，如下所示。

```
<VirtualHost *:80>  
    ServerName atomisator.ziade.org  
    <Location />  
        SetHandler mod_python  
        PythonHandler trac.web.modpython_frontend
```

```

PythonOption TracEnv /home/mercurial/atomisator/buildbot/parts/
trac
    PythonOption TracUriRoot /
    PythonPath "sys.path + ['/home/mercurial/atomisator/buildbot/
parts/trac', '/home/mercurial/atomisator/buildbot/eggs']"
</Location>
<Location "/login">
    AuthType Basic
    AuthName "Trac login"
    AuthUserFile /home/mercurial/atomisator/passwords
    Require valid-user
</Location>
</VirtualHost>

```

在这个配置中，有几个需要注意的地方：

- PythonOption 定义一个 TracEnv 值，所以 Trac 系统知道实例所在的位置；
- PythonPath 选项指向脚本访问 Trac 模块时所需要的本地 buildout 目录；
- /login 小节的设置与之前为 Mercurial 创建的 passwords 文件挂钩，这样用户可以使用相同的用户名登录到系统中。

3. 权限设置

为了使用问题管理系统，需要定义几个组：

- manager（管理员） 能够完全管理 Trac 的人；
- developer（开发人员） 能够修改问题票（ticket）及 wiki 页面的人；
- authenticated（已验证的） 能够创建问题票的人；
- anonymous（匿名的） 能够查看所有内容的人。

这 4 个角色都已经在 Trac 设置好并可以使用了。这是一个默认的权限设置，或者是在 buildout 运行时由 pbp.recipe.trac 自动完成的设置。

剩下的工作是在每个组中添加一些人。Trac 提供了一个命令行应用程序，可以用来在实例中设置一些元素，如下所示。

```

$ bin/trac-admin parts/trac/
Welcome to trac-admin 0.11b2
Interactive Trac administration console.
Copyright (c) 2003-2007 Edgewall Software
Type: '?' or 'help' for help on commands.

Trac [parts/trac]>

```

现在，可以在组中添加一个用户。例如，定义 tarek 为管理员，bill 和 bob 为开发人员，命令如下。

```
Trac [parts/trac] > permission add tarek manager  
Trac [parts/trac] > permission add bob developer  
Trac [parts/trac] > permission add bill developer
```

这将允许 tarek 通过 Web 来管理项目，bill 和 bob 则能够处理问题票和 wiki 页面。

9.3.2 使用 Trac 管理项目生命周期

这些设置使 Trac 在生命周期的管理中很容易使用，可以通过 Web 界面或命令行工具来进行。

1. 计划

在 Trac 中进行计划只需创建一个新的对应于迭代的里程碑（milestone），并确定它何时应该交付。管理员可通过 Web 界面或 trac-admin 命令行来添加它。后者更方便些，但是这意味着必须连接到执行那些任务的服务器上。

使用命令行来添加它，如下所示。

```
Trac [parts/trac] > milestone add atomisator-0.1.0  
Trac [parts/trac] > milestone due atomisator-0.1.0 2008-08-01
```

相同的操作可以通过/admin/ticket/milestones 上的 Web 界面中的 admin 部分来执行。

接着，里程碑将出现在 roadmap 部分。现在可以通过在 Web 界面上单击 New Ticket（新增问题票）按钮，来为该里程碑添加问题票。

创建一个为 atomisator.db 定义任务的问题票，定义一个保存反馈条目的映射表，如图 9.7 所示。

除了摘要和描述之外，要提供的信息包括：

- Assign to（任务分配给...） 开发人员姓名（我们将其分配给 Bob）；
- Type（类型） Task（任务），因为这是一个新功能；
- Component（组件） atomisator.db；
- Milestone（里程碑） atomisator-0.1.0；
- Estimated hours（估计工时） 8。

这个问题票将出现在里程碑中。

缺陷和改进问题票也是以相同的方式输入的。

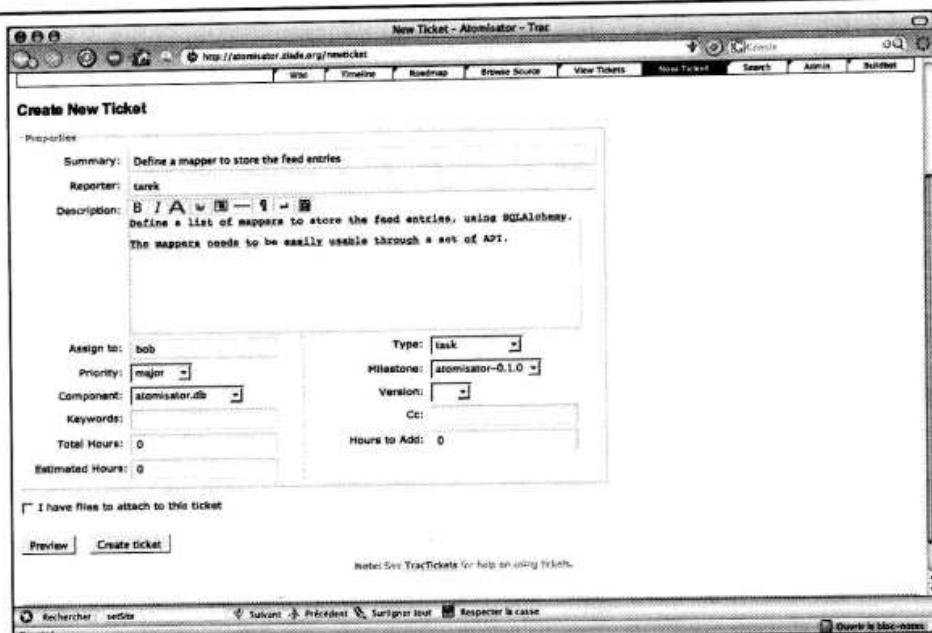


图 9.7

 预计工时属于 Trac 的一个名为 TimeAndEstimation 的插件，`pbp.recipe.trac` 会自动安装它。

相关内容请参见 <http://trac-hacks.org/wiki/TimingAndEstimationPlugin>。

2. 开发

每个开发人员可以通过/report 小节下的报告来查看他的任务，这可以通过 View Tickets（查看问题票）按钮来访问。My Tickets（我的问题票）将显示一个当前用户的问题票列表。

当 Bob 开始着手前面输入的问题票定义的工作时，他将执行 accept 操作。

任务完成时，他将填写 Total Hours（合计工时）字段并且执行 resolve 操作。

 这种轻量级的时间管理不能替代真正的管理计划系统，但是能够提供任务所花费时间的有用指示。

时间跟踪是改进估算的重要内容。

3. 清理

在团队结束一次迭代时，往往有一些任务尚未完成，清理阶段是最大限度地解决小问题的好机会。有些团队会组织缺陷冲刺，并在一两天内完成缺陷的处理。

在这个阶段结束时，剩余的任务将推迟到未来的里程碑，除非它们特别受欢迎。在这种情况下，它们必须被修复，并希望能在该阶段结束之前完成。

这些操作都通过在 Web 界面上编辑每个任务来完成。

4. 发行

发行由以下工作组成：

- 标记代码；
- 将对程序的修改从不稳定仓库中拉取到稳定仓库中；
- 创建发行仓库；
- 准备并交付一个发行版本；
- 将里程碑状态设置为 completed，以便关闭它；
- 创建新的里程碑并开始一个新的周期，有些团队会创建多达 3 个未来的里程碑，能够将问题票推进到具有优先级的未来里程碑中。

标记和拉取的工作发生在 Mercurial 端，这在前一章中已经介绍过，可以使用以下的命令集。

```
$ cd /home/mercurial/atomisator/repositories/unstable  
$ hg tag -f -m "tag for 0.1.0 release" 0.1.0  
$ cd ../stable  
$ hg pull ../unstable  
$ hg clone
```

最后，通过填充到期日来完成里程碑，如下所示。

```
Trac [parts/trac] > milestone completed atomisator-0.1.0 2008-08-01
```

Trac 可以用做信息的中心，创建的发行版本可以添加到 wiki 页面上，并在页面上公告。

9.4 小结

本章中介绍了以下内容：

- 各种管理软件开发生命周期的方法；
- 迭代开发方法的优点；
- 将项目组织为迭代的方法；
- Trac 的安装和使用；

下一章中将关注于软件文档。

第 10 章

编写项目文档

文档是常常被开发人员忽略的工作，有时候管理人员也会忽略它。这是因为项目周期即将结束时十分缺乏时间，而且人们觉得自己不擅长写作。虽然有些人确实不擅长此道，但是大部分人都能够制作出精细的文档。

总之，结果就是在忙乱中编写文档造成了无组织的文档。开发人员大部分时候都不喜欢这个工作，当现有的文档需要更新时情况更糟。许多项目只能提供低劣和过期的文档，因为管理人员不知道如何处理它。

但是，在项目一开始就建立文档过程，并且将文档当作代码的模块来处理，会使工作简单一些。遵循一些规则，甚至可以使编写文档变成有趣的工作。

本章将提供一些开始项目文档工作的技巧：

- 最佳实践总结的技术写作的 7 条规则；
- reStructuredText 入门，这是一种在大部分 Python 项目中使用的普通文本标记语言；
- 关于创建良好项目文档的指南。

10.1 技术性写作的 7 条规则

编写良好文档在很多方面比编写代码更容易。大部分开发人员认为它非常难，但是遵循一组简单的规则就可以使它变得比较容易。

这里谈论的不是如何写一本诗集，而是一些用来帮助读者理解设计、API 或建立代码库的任何相关内容的文本。

每个开发人员都能写出这样的素材，本节将提供 7 条适用任何情况的规则，如下所示。

- 分两步编写：先聚焦于思想，然后审查和修整文档。

- 以读者为目标：谁将读这个文档？
- 使用简单的风格：保持简单明了，使用良好的语法。
- 限制信息的范围：每次只介绍一个概念。
- 使用真实的代码示例：应该抛弃 Foos、bars 这些陈腐的用词。
- 保持简单，只要够用即可：写的不是一本书！
- 使用模板：帮助读者养成习惯。

这些规则大部分是 *Agile Documenting* 一书中创造并采用的，这本由 Andreas Rüping 编著的书的主题是为软件项目生成最好的文档。

10.1.1 分两步编写

Peter Elbow 在 *Writing with Power* 中说过，任何人都几乎不可能一次就写出完美的文档。问题是，许多开发人员在文档编写中会尝试直接写出完美的文档。

成功的唯一方法是每写两个句子就停一下并重读一遍，然后做些改正。这意味着需要同时关注内容和文档的风格。这很难，往往效果也不太好。在完全说清楚意思之前，反而花费了许多时间和精力在修饰文本的风格和外形上。

另一个方法是不管文本的样式和组织，而直接关注于其内容，使所有思想都跃然纸上，而不管它是怎么写出来的。开发人员将连续地写作，即使出现语法错误或其他与内容无关的问题时也不停下来。例如，不管句子是否难以理解，只要思想写下来就行，他只是写下自己想要说的和大概的文字组织。

这样做，开发人员可以关注于他想说的，并且可能从他的大脑中获取比一开始想的还多的内容。

进行自由写作的另一个副作用是，与主题不是直接相关的其他思想很容易混入。好的习惯是把这些思想写在另一张纸或另一个屏幕上，这样就不会丢失它们，然后再回到主题上来。

第二步是重新阅读整个文本并对其进行润色，使之对每个人来说都是易于理解的。润色意味着改进文本的风格，改正错误，重新组织，删除冗余信息，等等。

当用于编写文档的时间有限的时候，将这段时间分成两个阶段是个好习惯，一个阶段编写内容，一个阶段清理和组织文本。



先关注于内容，然后才是风格和清晰性。

10.1.2 以读者为目标

在开始编写文档时，作者应该考虑一个简单的问题：谁是本文档的读者？

这个问题的答案并不总是显而易见，因为技术文档用来说明软件的工作方式，常常是写给每个获得和使用这些代码的人看的。读者可能是一个为某个问题寻找合适的解决方案的管理人员，或者一个需要使用它来实现一个功能的开发人员。如果从架构的观点看该软件包符合其需求，设计人员也可能阅读这个文档。

应该应用一个简单的规则：每个文档应该只有一类读者。

这种思想会使文档编写更加容易一些。作者精确地知道他将要面对哪类客户，就可以为所有类型的读者提供简明而精确的文档。

在文档中提供一些简单的介绍性文字，说明文档的相关内容，指导读者找到合适的部分，这是一种良好的习惯，示例如下。

Atomisator 可以读取 RSS feed 并将其存储在数据库中，并且带有过滤进程的工具产品。

如果你是一个开发人员，请查看 API 描述 (`api.txt`)；

如果你是一个管理人员，请阅读功能列表和 FAQ (`features.txt`)；

如果你是一个设计人员，请阅读架构和基础结构说明 (`arch.txt`)。

以这种方式来引导读者，能够生成更好的文档。



在开始编写之前，要了解面向的读者。



10.1.3 使用简单的风格

Seth Godin 是销售方面最畅销的作家之一，在 Internet 上可以免费阅读其 *Unleashing the Ideavirus* 一书 (http://en.wikipedia.org/wiki/Unleashing_the_Ideavirus)。

最近，他在自己的博客上对他的书籍为什么这么畅销做了一个分析。他制作了一个销售领域所有畅销书的列表，并比较了每本书中每句话的单词数量。

他发现自己的书每句的单词数量最少（13 个单词）。Seth 解释道：这个简单的事实证明，读者喜欢短小精悍的句子，而不是冗长而有格调的。

保持句子短小精悍，那么读者从作品中提取、处理和理解其内容时将消耗较少的脑力。编写技术文档的目标是为读者提供一个软件的指南。它不是一部小说，应该更接近于微波炉说明而不是 Stephen King 的小说。

要记住如下几个技巧。

- 使用简单的句子，句子不应该长于两行。
- 每个段落应该由 3~4 个句子组成，最多表达一个主要的思想，以便文档能够呼吸。
- 不要有太多的重复，避免新闻稿风格——这种风格就是通过不断重复思想以确保读者能够理解。
- 不要使用多种时态，大部分时候用现在时就足够了。
- 如果还是一个真正的好作家，就不要在文档中开玩笑。技术书籍很难做到有趣，能够掌握该方法的作者也很少。如果你确实希望有点幽默，在代码实例中添加幽默会更好一些。



这不是在写小说，所以应尽量保持简单的风格。



10.1.4 限制信息的范围

不好的软件文档有一个简单的特征：当查找某些已知存在的信息时总是找不到。在花费了一些时间来阅读目录之后，即便开始尝试多种单词的组合搜索，也总是找不到想要找的。

当作者没有按照主题进行组织文档时就会发生这种情况。他们可能提供大量的信息，但是以单一或无逻辑的方式来汇集的。例如，如果读者寻找关于应用程序的总体描述，他就没必要阅读 API 文档——这是一个低水平的问题。

为了避免出现这种效果，段落应该被放在具有意义的标题之下，整个文档的标题应该在一个短句中概述其内容。

目录应该由所有小节的标题组成。

编写标题的简单方法是询问自己：在 Google 中输入什么短语来查找这个小节？

10.1.5 使用真实的代码示例

Foo 和 bar 这样无意义的伪变量是不好的。当读者试图通过用法示例来理解代码的工作方式时，不真实的实例将会使理解变得困难。

为什么不使用一个真实的例子？一个常用的方法是确定每个代码示例可以被剪切并粘贴到实际的程序中。

以下就是一个不好的示例。

有一个解析函数，如下所示。

```
>>> from atomisator.parser import parse
```

其用法如下。

```
>>> stuff = parse('some-feed.xml')
>>> stuff.next()
{'title': 'foo', 'content': 'blabla'}
```

更好的例子应该是，在该解析器知道如何使用解析函数返回一个 feed 的内容时，可作为一个顶级函数使用，如下所示。

```
>>> from atomisator.parser import parse
```

其用法如下。

```
>>> my_feed = parse('http://tarekziade.wordpress.com/feed')
>>> my_feed.next()
{'title': 'eight tips to start with python',
 'content': 'The first tip is..., ...'}
```

这个微小的差别可能有点过分夸张，但是实际上它能够使文档更有帮助。读者可以将这些代码行复制到一个 shell 程序中，理解 `parse` 将使用一个 URL 作为参数，并且返回包含博客条目的一个枚举型变量。



代码示例应该在实际的程序中直接可用。



10.1.6 保持简单，够用即可

在大部分敏捷方法中，文档并不是首要的。使软件能工作是最重要的事情，其重要性超过了详尽的文档。所以一个好的做法是只编写真正需要的文档，而不是创建一组完备的文档，正如 Scott Ambler 在 *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*（中译版《敏捷建模：极限编程和统一过程的有效实践》）一书中所说的那样。

例如，对于管理员而言，只需用一个文档说明 Atomisator 是如何工作的就足够了。对他们来说，除了需要知道这个工具的配置和运行方法之外没有其他任何需求。这个文档应将范围限制在一个问题上：如何在自己的服务器上运行 Atomisator？

除了读者和范围之外，将每个小节限制在很少的几页之内也是一个好主意。每个小节最多 4 页，这样作者就必须整合他的思想。如果需要更多页，可能就意味着该软件太过复杂而

难以说明和使用。



可以运行的软件胜过面面俱到的文档——敏捷宣言。



10.1.7 使用模板

Wikipedia 上的每个页面都很相似：左侧的方框用来显示日期或事件摘要；在文档的开始是一个目录，里面有指向本条目中锚点的链接；最后总有一个参考文献小节。

用户习惯于此。例如，他们知道可以快速地看一下目录，如果没有找到所需要的信息，将直接到参考文献小节查看是否可以找到相关主题的另一个网站。这对于 Wikipedia 上的任何页面都是有效的。可以通过学习 Wikipedia way 来获得更高的效率。

所以，使用模板强制文档按照常见的模式，能够使人们更有效地使用它们。他们习惯于这种结构并且知道如何更快地阅读它。

为每类文档提供一个模板也为作者提供了一个快速启动的方案。

本章中，我们将看到各种软件可能拥有的文档，并使用剪贴本（Paster）来为它们提供框架。首先介绍 Python 文档应该使用的标示语法。

10.2 reStructuredText 入门

reStructuredText 也称为 reST（参见 <http://docutils.sourceforge.net/rst.html>）。它是一种普通文本标记语言，在 Python 社区中被广泛地应用于包文档中。reST 很好的一点是文本仍然是可读的，因为其标记语法不会像 LaTeX 那样使文本变得混乱。

以下是这种文档的一个样例。

```
=====
Title
=====
Section 1
=====
This *word* has emphasis.
Section 2
=====
Subsection
```

```
:::::::  
Text.
```

reST 从属于 docutils 包。这个包提供了一批脚本，可以将 reST 文件转换为各种格式的文件，如 HTML、LaTeX、XML 甚至 S5 (Eric Meyer 的幻灯片系统，参见 <http://meyerweb.com/eric/tools/s5>)。

作者可以只关注于内容，然后根据需要来决定如何显示它。例如，Python 本身的文档就是 reST 格式，最后以 HTML 显示在 <http://docs.python.org> 中，此外还有其他各种格式。

开始编写 reST 之前，至少应该知道以下要素：

- 小节结构；
- 列表；
- 内联标签；
- 文字块；
- 链接。

本节是对该语法的概述，更多信息请参考 <http://docutils.sourceforge.net/docs/user/rst/quickref.html>，这是着手使用 reST 的一个好起点。

要安装 reStructuredText，需要安装 docutils，命令如下。

```
$ easy_install docutils
```

将得到一组名称以 rst2 开始的脚本，可以将 reST 转换为各种格式显示。

例如，rst2html 脚本将把指定 reST 文件转成 HTML 格式的输出，如下所示。

```
$ more text.txt
Title
=====
content.

$ rst2html.py text.txt > text.html
$ more text.html
<?xml version="1.0" encoding="utf-8" ?>
...
<html ...>
<head>
...
</head>
<body>
<div class="document" id="title">
<h1 class="title">Title</h1>
```

```
<p>content.</p>
</div>
</body>
</html>
```

10.2.1 小节结构

文档的标题及其小节使用以非字母字符作为下划线符号的方法表示。它们可以带有上划线和下划线，常见的做法是对标题同时使用上下划线，而对于小节则只使用下划线。

小节、标题的下划线字符根据常用情况排序为：=、-、_、:、#、+、^。

当在小节中使用某个字符时，它与小节的级别相关并且必须在整个文档中保持一致。示例如下（见图 10.1）。

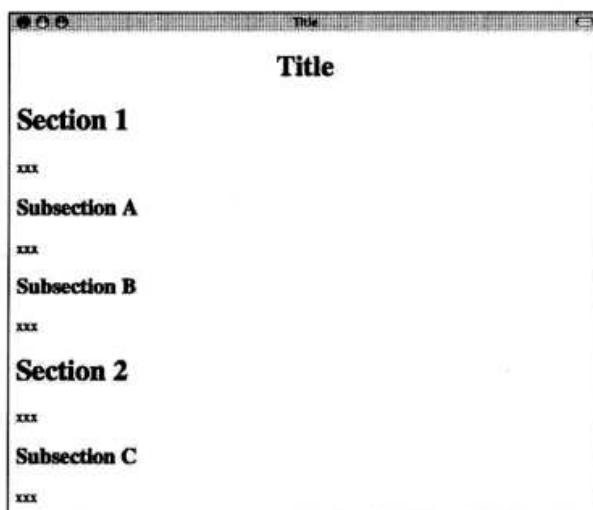


图 10.1

```
=====
Title
=====
Section 1
=====
xxx
Subsection A
-----
xxx
Subsection B
-----
```

```

xxx
Section 2
=====
xxx
Subsection C
-----
xxx

```

这个文件的 HTML 输出如图 10.1 所示。

10.2.2 列表

reST 提供了着重号列表、编号列表和具有自动编号功能的定义列表，如下所示。

```

Bullet list:
- one
- two
- three

Enumerated list:
1. one
2. two
#. 自动编号

Definition list:
one
    one is a number.

two
    two is also a number.

```

10.2.3 内联标签

文本可以使用内联标签来定义样式：

- ***emphasis*** 斜体；
- ****strong emphasis**** 黑体；
- **``inline literal``** 内联预格式化文本；
- **`a text with a link`_** 只要它出现在文档中，将被一个超链接所替代（参见 10.2.5 小节）。

10.2.4 文字块

当需要介绍一些代码实例时，可以使用文字块。文字块的标识是两个冒号，它是一个带缩进的段落，如下所示。

```
This is a code example
::
>>> 1 + 1
2
Let's continue our text
```



别忘了在::和文本块之后添加一个空行，否则它将不会显示。



注意，冒号可以放在文本行中。这时候，在各种显示格式中都将显示为一个冒号，如下所示。

```
This is a code example:::
>>> 1 + 1
2
Let's continue our text
```

如果不想保留这个冒号，可以在 example 和::之间插入一个空格。这样，::将被彻底删除。

10.2.5 链接

如果要将文本改成一个外部链接，可以在前面添加两个逗点，如下所示。

```
Try `Plone CMS`_, it is great ! It is based on Zope_.
.. _`Plone CMS`: http://plone.org
.. _Zope: http://zope.org
```

通常的做法是将所有的外部链接集中放在文档的末尾。当链接的文本中包含空格时，必须将其放在两个'字符之间。

也可以在文本中添加一个标签来定义内部链接，如下所示。

```
This is a code example
... _example:
::
>>> 1 + 1
2
Let's continue our text, or maybe go back to
the example_.
```

小节也可以当作目标来用，如下所示。

```
=====
Title
=====
Section 1
-----
xxx
Subsection A
-----
xxx
Subsection B
-----
-> go back to `Subsection A`_
Section 2
-----
xxx
```

10.3 构建文档

指导读者和作者的最简单方法是为每个人提供帮助和指南，就像前面介绍的那样。

从作者的角度，这可以通过拥有一组可复用的模板和描述如何、何时在项目中使用这些模板的指南来完成，它被称为文档工件集（documentation portfolio）。

从读者的角度，建立一个文档景观（document landscape），能够轻松地浏览文档并高效地查找信息。

构建工件集

一个软件项目可以包含许多种文档，从低层级的文档（将直接引用代码）到提供应用程序高级概述的设计描述。

例如，Scott Ambler 在其 *Agile Modeling*（参见 <http://www.agilemodeling.com/essays/agileArchitecture.htm>）一书中定义了一个详尽的文档类型列表。他构建了一个从早期规格说明书到操作文档的一个工作集，甚至包括了项目管理文档，所以整个文档需求可以使用一个标准模板集来构建。

因为一个完整的工作集和构建软件的方法紧密相关，所以本章将只关注能够完成特定需求的公共子集。构建一个高效的工作集需要很长的时间，因为它将捕获开发人员的工作习惯。

在软件项目中，常见的工作集可以分为 3 类：

- 设计 提供架构信息和详细设计信息，如类图、数据库图等；
- 使用 关于如何使用软件的文档，可以是菜谱和教程的形式或模块级帮助；
- 操作 提供如何部署、升级和操作软件的指南。

1. 设计

设计文档的目的是描述软件的工作机制和代码的组织方式。开发人员通过它来理解系统，它也是人们理解应用程序工作方法的一个很好的切入点。

一个软件可能有不同类型的设计文档：

- 架构概述；
- 数据库模型；
- 带有依赖性和层次关系的类图；
- 用户界面线框图；
- 基础设施描述。

这些文档大部分由一些图和少量文字组成。这些图所用的约定和团队及项目有关，只要保持一致就很好。



UML 提供了覆盖软件设计大部分方面的 13 种图。类图可能是最常用的，它可以描述软件的各个方面（请参见 http://en.wikipedia.org/wiki/Unified_Modeling_Language#Diagrams）。

完全遵循某种特定的建模语言（诸如 UML）往往是做不到的，团队只是按照自己的经验来进行工作。他们从 UML 或其他建模语言中选择好的方法，并且创建自己的方法。

例如，对于架构概述图，有些设计人员只在白板上绘制方框和箭头，而不遵循什么特殊的设计规则及图形标准。其他人可能会使用诸如 Dia (<http://www.gnome.org/projects/dia>) 或 Microsoft Visio (不是开源软件，所以不是免费的) 之类的简单绘图工具，因为这对于理解设

计已经足够了。例如，第 6 章中介绍的所有架构图都是使用 OmniGraffle 制作的。

数据库模型图则取决于所使用的数据库类型。完整的数据建模软件提供了绘制工具，能够自动生成表及表间关系。但是对于 Python 而言，绝大部分时候都显得有些过载。如果使用诸如 SQLAlchemy 这样的 ORM 工具，一些带有字段列表的简单方框，加上第 6 章中所看到的表间关系就足以在开始编写之前描述映射关系了。

类图通常使用 UML 类图的简化版本，例如，在 Python 中不需要指定类的 `protected` 类型成员。所以，用于架构概要图的工具也能够满足这种需求。

用户界面图取决于打算编写 Web 还是桌面应用程序。Web 应用程序常常只描述屏幕的中心区域，因为页首、页脚、左右面板都是公用的。许多 Web 开发人员会手工绘制这些屏幕快照，然后再用照相机或扫描仪电子化，还有些人会在 HTML 中创建原型并进行屏幕截图。对于桌面应用程序而言，原型屏幕的截图或者使用诸如 Gimp 或 Photoshop 之类的工具制作的带有注解的模拟图是常见的做法。

基础结构概要图和架构类似，但是它们关注于软件与第三方元素（如邮件服务器、数据库以及各种数据流）之间的交互。

2. 公用的模板

创建这样的文档时，重要的是确定完全了解目标读者，以及内容的限定范围。这样用于设计文档的通用模板可以为作者提供一个轻巧的结构和一些写作建议。

这样的结构可以包含：

- 标题；
- 作者；
- 标签（关键字）；
- 描述（抽象）；
- 目标（谁应该阅读）；
- 内容（带图形）；
- 对其他文档的引用。

内容应该最多占用 3~4 个页面（1024×768），要确保将其限制在这个范围内。如果范围过大，应该分成几个文档，或者对其进行概括。

模板还提供了作者姓名和一个标签列表，用来管理其发展并且简化分类。这将在本章稍后部分介绍。

剪贴板（Paster）是用来为文档提供模板的正确工具。`pbp.skels` 实现前面描述的设计模板，而且可以像代码生成一样的使用。提供一个目标文件夹并回答几个问题，如下所示。

```
$ paster create -t pbp_design_doc design
Selected and implied templates:
```

```
pbp.skels#pbp_design_doc A Design document
Variables:
egg:    design
package: design
project: design
Enter title ['Title']: Database specifications for atomisator.db
Enter short_name ['recipe']: mappers
Enter author (Author name) ['John Doe']: Tarek
Enter keywords ['tag1 tag2']: database mapping sql
Creating template pbp_design_doc
Creating directory ./design
Copying +short_name+.txt_tmpl to ./design/mappers.txt
```

然后，就将生成以下的结果。

```
=====
Database specifications for atomisator.db
=====
:Author: Tarek
:Tags: database mapping sql
:abstract:
    Write here a small abstract about your design document.
.. contents ::
Who should read this ?
::::::::::
Explain here who is the target readership.
Content
::::::::::
Write your document here. Do not hesitate to split it in several
sections.
References
::::::::::
Put here references, and links to other documents.
```

3. 使用

使用文档用来描述软件特定部分的工作模式。这个文档可以描述低层级的部分，比如一个函数是如何工作的，也可以是较高层级的部分，比如调用程序的命令行参数。这是框架应用程序文档中的最重要部分，因为目标读者主要是那些打算复用这些代码的开发人员。

主要的文档有以下 3 种。

- **Recipe** 解释如何完成某事的一个简短文档。这种文档针对一个读者群体，只关注一个特定的主题。
- **教程** 一个渐进地解释如何使用软件功能的文档。这个文档可以引用 `recipe`，每个实例针对一个读者群体。
- **模块级帮助** 说明模块所包含的内容的低层级文档。这个文档可以在调用模块内建的 `help` 命令时显示。

(1) `recipe`

一个 `recipe` 用来回答一个很具体的问题，并且提供相应的解决方案。

例如，ActiveState 提供了一个联机的 Python 说明书（即一个 `recipe` 集合），开发人员可以描述在 Python 中如何做某事（参见<http://aspn.activestate.com/ASPN/Python/Cookbook>）。

这些 `recipe` 必须简单，其结构类似于：

- 标题；
- 提交者；
- 最后更新时间；
- 版本；
- 类别；
- 描述；
- 源代码；
- 讨论（代码的文字解释）；
- 注释（来自 Web）。

`recipe` 往往只有一个屏幕长，不会特别详细。这个结构完全符合软件的需求，而且可以在一种通用结构中使用，在通用结构中添加目标读者群，用标签替换类别：

- 标题（短语）；
- 作者；
- 标签（关键字）；
- 谁应该阅读这个文档；
- 先决条件（例如其他需要阅读的文档）；
- 问题（简单的描述）；
- 解决方案（主文本，1~2 个页面）；
- 引用（指向其他文档的链接）。

日期和版本在这里都没有用，因为稍后将看到，文档的管理与项目中的源代码管理十分类似。

和 `design` 模板类似，`pbp.skels` 提供了 `pbp_recipe_doc` 模板，它可以用来生成如下这种结构。

```
$ paster create -t pbp_recipe_doc recipes
Selected and implied templates:
  pbp.skels#pbp_recipe_doc A recipe
Variables:
  egg:      recipes
  package:  recipes
  project:  recipes
Enter title (use a short question): How to use atomisator.db
Enter short_name ['recipe'] : atomisator-db
Enter author (Author name) ['John Doe']: Tarek
Enter keywords ['tag1 tag2']: atomisator db
Creating template pbp_recipe_doc
Creating directory ./recipes
Copying +short_name+.txt_tmpl to ./recipes/atomisator-db.txt
```

然后，writer 将完成如下所示的结果。

```
=====
How to use atomisator.db
=====

:Author: Tarek
:Tags: atomisator db
.. contents ::

Who should read this ?

::::::::::

Explain here who is the target readership.

Prerequisites
::::::::::

Put here the prerequisites for people to follow this recipe.

Problem
::::::::::

Explain here the problem resolved in a few sentences.

Solution
::::::::::

Put here the solution.

References
::::::::::

Put here references, and links to other recipes.
```

(2) 教程

教程的用途和 `recipe` 不同。它不是用来解决一个独立问题的，而是一步步地描述应用程序各个功能的使用方法。它可能比 `recipe` 长，并且可以关注应用程序的许多部分。例如，Django 在其网站上提供一系列教程。*Writing your first Django App, part 1*（参见 <http://www.djangoproject.com/documentation/tutorial01>）就用了 10 个页面来说明使用 Django 构建一个应用程序的方法。

这样一个文档的结构可以是：

- 标题（短句）；
- 作者；
- 标签（单词）；
- 描述（抽象）；
- 谁应该阅读本教程；
- 先决条件（例如其他需要阅读的文档）；
- 教程（主文本）；
- 引用（指向其他文档的链接）。

在 `pyp.skel` 中提供的 `pyp_tutorial_doc` 模板也使用了这个结构，这与设计模板相似。

(3) 模块帮助

在我们的集合中，最后一个可以加入的模板就是模块帮助模板。模块帮助将引用单个模块，并提供其内容的一个描述，以及用法示例。

一些工具可以通过提取 `docstrings` 并使用 `pydoc` 计算模块级帮助来构建这样的文档，如 Epydoc（参见 <http://epydoc.sourceforge.net>）。所以，根据 API 内省来生成一个大规模的文档也是有可能的。这种文档往往在 Python 框架中提供，例如，Plone 提供一个 <http://api.plone.org> 服务器，用来保存模块帮助的最新集合。

这种方法的主要问题是：

- 没有在真正对文档感兴趣的模块上进行智能选择；
- 代码可能被文档搞乱。

而且，模块文档中提供的示例有时候可能会引用模块的多个部分，很难分清函数和类的 `docstring`。可在模块的最开始写一段文字，模块的 `docstring` 可以用来完成这个任务。但是这样做会造成一个由一个文本和之后的代码块组成的混合文件。如果代码的总长度只占到全部的 50% 以下，就会显得很混乱。如果是作者，这完全没问题；但是当人们试图阅读代码（而不是文档）时，他们就必须跳过 `docstring` 部分。

另一种方法是将文本单独放在一个文件中，通过一个手工的选择操作来决定哪个 Python 模块将拥有模块级帮助文件。之后这些文档从代码库中分离出来，并且存在于自己的文件中，就像稍后将要看到的那样。这是 Python 的文档制作方法。

许多开发人员承认，文档和代码分离比使用 `docstring` 更好一些。这种方法意味着文档处理完全集成到开发周期中了，否则，它将很快变得十分陈旧。`docstring` 方法通过使代码和用法示例之间保持相近来解决这个问题，但是不要将它带到更高的级别上——一个可以作为普通文档一部分的文档。

用于模块级帮助的模板实际上很简单，因为它在编写的内容之前只包含一点元数据。因为是开发人员想要使用模块，所以目标不需要定义，而只需要：

- 标题（模块名称）；
- 作者；
- 标签（单词）；
- 内容。



下一章将介绍如何使用 `doctest` 和模块块帮助进行测试驱动开发。



4. 操作

操作文档用来描述软件操作的方法。例如：

- 安装和部署文档；
- 管理文档；
- 在出现故障时为用户提供帮助的“常见问题解答”文档；
- 说明如何获得帮助或者提供反馈的文档。

这些文档很具体，但可以使用前面小节中定义的教程模板。

10.4 建立自己的工件集

前面讨论的模板只是用于制作软件文档的一个基础。由此，就像介绍剪贴板的那一章中所说明的那样，可以调整它并添加入其他模板，以建立自己的文档工作集。

记住，项目文档应保持简单，只要够用即可。每个加入的文档应该有一个清晰的目标读者群，应该符合一个实际的需求，而不应该写没有增加真正价值的文档。

构建景观

前一节中构建的文档工件集提供了文档级别的一个结构，但是没有提供一种组织它并为读者创建文档的方法。这就是 Andreas Rüping 称之为文档景观的东西，指的是读者在浏览文档时使用的意境地图。他得出的结论是，组织文档的最好方式就是建立一棵逻

辑树。

换句话说，组成工作集的不同类型文档必须在一棵目录树中找到适合存在的位置。这个位置在作者创建文档和读者查找文档时都应该是显而易见的。

浏览文档时，每个级别上的索引页面是对作者和读者的有效助手。

建立一个文档景观分两步完成：

- 为作者建立一棵树；
- 在作者的树的顶端为读者建立一棵树。

作者和读者之间的区别很重要，因为他们将在不同位置以不同的格式访问文档。

1. 作者的布局

从作者的角度看，每个文档和 Python 的模块一样处理。它应该保存在版本控制系统中，像代码一样运作。

作者不关心文章的最后外观和出现的地方，他们只希望确保编写了一个文档，所以唯一的真理是覆盖所需的主题。

保存在一个文件夹树中的 reStructuredText 文件，在版本控制系统中可以与软件代码一起利用，这是建立作者文档景观的一种方便的解决方案。

如果回头看第 6 章中 Atomisator 的文件夹结构，可以将 docs 文件夹作为这棵树的根。

组织树的最简单方式是根据特性为文档分组，如下所示。

```
$ cd atomisator
$ find docs
docs
docs/source
docs/source/design
docs/source/operations
docs/source/usage
docs/source/usage/cookbook
docs/source/usage/modules
docs/source/usage/tutorial
```

注意，这棵树位于 source 文件夹中，因为 docs 文件夹将在下一小节被用作根文件夹来创建一个特殊的工具。

由此，可以在每个级别添加一个 index.txt 文件（除了根以外），说明文件夹中包含了哪类文档，或者总结每个子文件夹中包含的内容。这些索引文件可以定义其包含文档的列表。例如，operation 文件夹可以包含一个可用操作文档的列表，如下所示。

```
=====
Operations
=====

This section contains operations documents:
- How to install and run Atomisator
- How to install and manage a PostgreSQL database
for Atomisator
```

为了让人们不会忘记更新这些文档，可以自动生成这些列表。

2. 读者的布局

从读者的角度，制作索引文件并将整个文档以易于阅读的漂亮格式提交是很重要的。Web 页面是最好的选择，并且它很容易从 reStructuredText 文件中生成。

Sphinx (<http://sphinx.pocoo.org>) 是一组可以用来从文本树生成一个 HTML 结构的脚本和 docutils 扩展。这个工具用来（例如）创建 Python 文档，现在很多项目都使用它来制作文档。使用其内建的功能，可以生成一个真正精细的浏览系统，以及一个简单但足够用的客户端 JavaScript 搜索引擎。它还使用 pygments 来显示代码示例，生成真正漂亮的语法强调显示。

Sphinx 能够简单地配置，以绑定在前面小节中定义的文档景观。

安装时只需要调用 easy_install，如下所示。

```
$ sudo easy_install-2.5 Sphinx
Searching for Sphinx
Reading http://cheeseshop.python.org/pypi/Sphinx/
...
Finished processing dependencies for Sphinx
```

这将安装几个脚本，如 sphinx-quickstart。这个脚本将生成一个脚本和一个 Makefile，可以用于在每次需要时生成 Web 文档。在 docs 文件夹中运行这个脚本，并回答相应的问题，如下所示。

```
$ sphinx-quickstart
Welcome to the Sphinx quickstart utility.
Enter the root path for documentation.
> Root path for the documentation [.]:
> Separate source and build directories (y/n) [n]: y
> Name prefix for templates and static dir [.]:
> Project name: Atomisator
> Author name(s): Tarek Ziadé
```

```

> Project version: 0.1.0
> Project release [0.1.0]:
> Source file suffix [.rst]: .txt
> Name of your master document (without suffix) [index]:
> Create Makefile? (y/n) [y]: y
Finished: An initial directory structure has been created.
You should now populate your master file ./source/index.txt and create
other documentation
source files. Use the sphinx-build.py script to build the docs, like so:
make <builder>

```

这将在源文件夹添加一个 `conf.py` 文件，其中包含通过回答问题定义的配置信息，还有根目录下的 `index.txt` 和 `docs` 文件夹中的 `Makefile`。

运行 `make html` 将在 `build` 中生成一棵树，如下所示。

```

$ make html
mkdir -p build/html build/doctrees
sphinx-build.py -b html -d build/doctrees -D latex_paper_size= source
build/html
Sphinx v0.1.61611, building html
trying to load pickled env... done
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
creating index...
writing output... index
finishing...
writing additional files...
copying static files...
dumping search index...
build succeeded.

Build finished. The HTML pages are in build/html.

```

现在，生成的文档将被放在 `build/html` 中，起点是 `index.html`（见图 10.2 所示）。

除了文档的 HTML 版本之外，该工具还将创建一些自动生成的页面，如模块类别和索引。

Sphinx 提供了少量的 `docutils` 扩展，它是这些功能的驱动力。这些扩展主要包括：

- 一条创建目录的指令；
- 一个可以将文档注册为模块帮助的标记；
- 一个在索引中添加元素的标记。

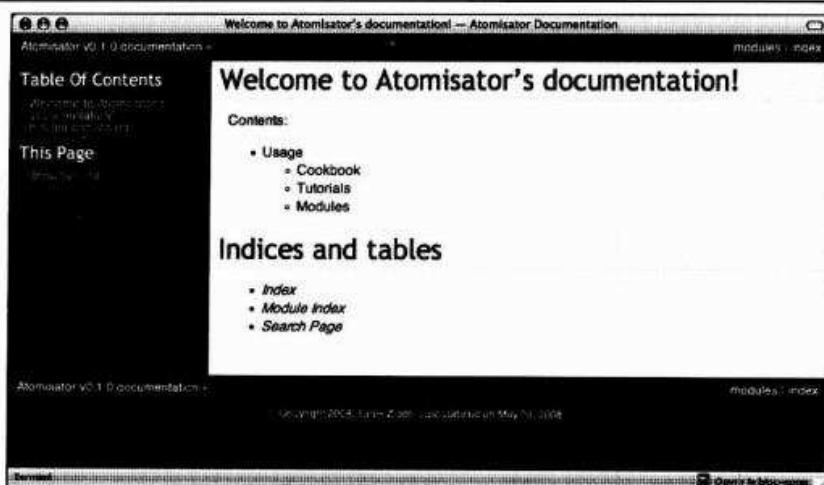


图 10.2

(1) 建立索引页面

Sphinx 提供了一条 `toctree` 指令，可以用来在文档中注入一个目录，以及指向其他文档的链接。每一行必须是一个文件以及针对当前文档的相对路径。也可以使用通配符来添加符合条件的多个文件。

例如，之前已经在作者景观中定义的 `cookbook` 文件夹所对应的索引文件可以类似于：

```
=====
Cookbook
=====

Welcome to the CookBook.

Available recipes:
.. toctree::
:glob:
*
```

使用这种语法，HTML 页面将显示 `cookbook` 文件夹中所有可用的 reStructuredText 文档。这条指令可用来为所有索引文件创建可浏览的文档。

(2) 注册模块帮助

对于模块帮助，可以添加一个标志，这样它就可以自动在模块索引页面中列出并可用，如下所示。

```
=====
session
=====

.. module:: db.session
The module session...
```

注意，db 前缀可以用来避免模块冲突。Sphinx 将它作为模块的类别，并且将所有以 db. 开头的模块分到这个类别中。

对于 Atomisator，条目分组将使用 db、feed、main 和 parser，如图 10.3 所示。

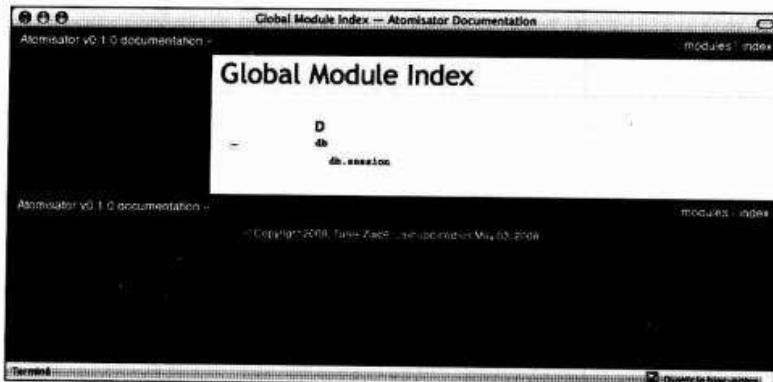


图 10.3

在文档中，可以在有很多模块的时候使用这个功能。

 注意，前面创建的模块帮助模板（`pbp_module_doc`）可以被修改成默认添加 `module` 指令。

(3) 添加 Index 标志

另一个可用的选项是通过将文档链接到一个条目上，填充索引页面，如下所示。

```
=====
session
=====
.. module:: db.session
.. index::
   Database Access
   Session
The module session...
```

这时，将在索引页面中添加两个新的条目：Database Access 和 Session。

(4) 交叉引用

最后，Sphinx 提供了一个内联标签来设置交叉引用。例如，一个指向模块的链接可以如下这样完成。

```
:mod:`db.session`
```

:mod:是模块标签的前缀，`db.session`是所链接到的模块名称(前面所注册的)。记住，:mod:和前面的元素都是 Sphinx 在 reStructuredText 中引入的特殊指令。

 Sphinx 提供了许多功能，可以在它的网站上看到。例如，autodoc 功能是个很了不起的选项，能够自动地从 doctest 中提取并建立文档。
具体信息参见<http://sphinx.pocoo.org>。

10.5 小结

本章详细地说明了如何：

- 使用几条规则来高效地编写文档；
- 使用 Python 风格的 LaTeX，即 reStructuredText；
- 构建一个文档工作集和文档景观；
- 使用 Sphinx 生成精巧的 Web 文档。

制作项目文档中最困难的事情是保持文档准确和及时更新，将文档作为代码库的一部分可以大大简化这件事。由此，每当开发人员修改一个模块时，他（她）应该同时修改对应的文档。

这在大型的项目中可能会相当困难，在模块的标题中添加一个相关文档的列表，对此会有帮助。

确保文档始终准确的辅助方法是通过 doctest 将文档和测试合并。

方面的内容将在下一章中介绍：将介绍测试驱动开发的原则，然后是文档驱动开发。

第 11 章

测试驱动开发

测试驱动开发（TDD）是制造高质量软件的一种简单技术。它在 Python 社区被广泛地应用，在使用静态类型语言的社区中可能用得更多。这可能是由于开发人员认为编译程序可以完成大部分测试，它在生成二进制码的时候会进行许多检查。

因此，他们在开发阶段将停止执行测试。但是这往往会导致产生低质量的代码，以及花费很长时间的调试来使其正常工作。记住，大部分的缺陷与不良的语法无关，而与逻辑错误和可能导致重大破坏的微小误解有关。

本章将分成两个部分：

- 我不测试，提倡 TDD 并快速地描述如何借助标准库进行这种开发；
- 我测试，这是为实际进行测试并希望从中获得更多价值的开发人员准备的。

11.1 我不测试

如果相信 TDD 的价值，请转到下一节。下一节中将关注于更高级的技术，以及在编写测试时如何简化工作。本节主要是为那些还没有使用这种方法的开发人员准备的，并且尝试倡导它的使用。

11.1.1 测试驱动开发原理

TDD 由编写覆盖所需功能的测试用例，然后编写该功能两部分工作组成。换句话说，将在代码存在之前编写测试用例。

例如，如果开发人员要编写一个计算一系列数字的平均值的函数，那么将首先编写几个

使用它的实例，并提供预期的结果，如下所示。

```
assert average(1, 2, 3) == 2
assert average(1, -3) == -1
```

这些例子可以由另一个人提供。现在将着手实现该函数，直到这两个示例能够正常工作，如下所示。

```
>>> def average(*numbers):
...     return sum(numbers) / len(numbers)
...
>>> assert average(1, 2, 3) == 2
>>> assert average(1, -3) == -1
```

缺陷或非预期的结果是该函数应该能够处理的新的测试用例，如下所示。

```
>>> assert average(0, 1) == 0.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

需要对代码做相应的修改，直到新的测试通过为止，如下所示。

```
>>> def average(*numbers):
...     # 确保可以使用浮点型
...     numbers = [float(number) for number in numbers]
...     return sum(numbers) / float(len(numbers))
...
>>> assert average(0, 1) == 0.5
```

更多的测试用例将改进代码，如下所示。

```
>>> try:
...     average()
... except TypeError:
...     # 希望空序列抛出一个类型错误
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 3, in average
ZeroDivisionError: integer division or modulo by zero
```

```
>>>
>>> def average(*numbers):
...     if numbers == ():
...         raise TypeError('You need to provide at '
...                         'least one number')
...     numbers = [float(number) for number in numbers]
...     return sum(numbers) / len(numbers)
...
...
>>> try:
...     average()
... except TypeError:
...     pass
...
```

到此，可以将所有的测试收集在 `test` 函数中，它将在每次修改代码时运行，如下所示。

```
>>> def test_average():
...     assert average(1, 2, 3) == 2
...     assert average(1, -3) == -1
...     assert average(0, 1) == 0.5
...
...     try:
...         average()
...     except TypeError:
...         pass
...
>>> test_average()
```

每次修改时，`test_average` 和 `average` 将一起修改，并且再次运行以确认所有测试用例仍然能够通过。其使用方法是将所有测试集中在当前包的 `tests` 文件夹中，每个模块在那里都可以有与之对应的测试模块。

这个方法提供以下好处：

- 避免软件退化；
- 增进代码质量；
- 提供最好的低层级文档；
- 更快地产生健壮的代码。

1. 避免软件退化

在开发生涯中，都要面对软件退化的问题。软件退化就是修改引入的新缺陷。退化的发

生是因为无法在某个点猜测代码库中的某个修改可能会导致的结果。修改一些代码可能破坏其他的功能，有时候会导致可怕的副作用，比如悄悄地对数据产生破坏。

为了避免退化，在每次修改时都应对软件的所有功能进行测试。

将代码库开放给多个开发人员将会放大这个问题，因为每个人都无法完全了解所有开发活动。虽然版本控制系统能够避免冲突，但是无法避免所有不必要的交互。

TDD 有助于减少软件退化。整个软件可以在每次修改之后自动测试。只要每个功能都有合适的测试集就能做到这一点。当真正实现 TDD 时，测试库和代码库将一起增长。

因为完整的测试可能持续相当长的时间，所以将其委托给 `buidbot` 是个好做法，它可以在后台执行这项工作（这在第 8 章中已经介绍过）。但是在本地重新运行测试应该由开发人员手工完成，至少在所关心的模块上。

2. 增进代码质量

在编写一个新的模块、类或函数时，开发人员关注于如何编写它以及如何生成最佳的代码。但是当他聚焦于算法时，可能会忘记了用户的看法：他的函数何时、如何使用？参数的使用是否简便和有逻辑性？API 的名称是否正确？

应用前面的章节中介绍的技巧，例如选择好的名称，就可以做到这一点。但是要使其具有效率，可用的方法只有编写测试用例。这是开发人员意识到所写的代码是否有逻辑性和易于使用的时机。往往，在模块、类或函数完成之后，马上就会发生第一次重构。

编写测试，也就是代码的一个具体使用场景，对于获取此类用户观点是有帮助的。因而，开发人员使用 TDD 往往能够生成更好的代码。

测试计算很复杂又带有副作用的函数是很困难的。注重测试的代码在编写时应该具有更清晰和模块化的结构。

3. 提供最佳的开发人员文档

测试是开发人员学习软件工作方式的最佳途径。它们的代码主要针对使用场景。阅读它们，能够对代码的工作方式建立快速而且深入的观察。有时候，一个例子值一千句话。

始终与代码库一起更新的测试，将能够生成软件所能拥有的最好的开发人员文档。测试不会像文档一样静止不动，否则就会失效。

4. 更快地产生健壮的代码

不进行测试将会导致需要花费很长的调试时间。在软件的一个部分中的缺陷可能会在“风马牛不相及”的另一个部分中发现。因此，不知道责任在谁，从而花费了过度的调试时间。测试失败是对付小缺陷的更好时机，因为可以获得更好的提示以了解问题的真正所在。而且测试往往比调试更加有趣，因为它是编码工作。

如果度量用于修复代码和编写代码的总时间，一般都会比 TDD 方法所需的时间更长。这在开始编写新代码时并不明显，因为建立测试环境和编写几个测试用例所需的时间看起来要比只编写代码的时间长得多。

但是有些测试环境确实难以建立。例如，当代码与一个 LDAP 或 SQL 服务器交互时，编写测试用例就不直接。这将在 11.2.2 小节中介绍。

11.1.2 哪一类测试

对于任何软件，都可以应用多种测试，其中最主要的是验收测试（或功能测试）和单元测试。

1. 验收测试

验收测试关注于功能，软件被看作一个黑盒子。它只是确定软件确实完成了预期的事情，使用和用户相同的介质，并控制输出。这些测试一般在开发周期之外编写，以验证应用程序符合要求。它们一般像是针对软件的一个检查列表。这些测试往往不是通过 TDD 完成的，而是由管理人员甚至客户来构建。在这种情况下，它们被称作用户验收测试。

同样，验受测试也应该使用 TDD 原则来完成。测试可以在功能编写之前提供。开发人员一般根据功能说明书来制作一些验收测试用例，并确保代码能够通过这些测试。

用于编写这些测试的工具取决于软件提供的用户界面。Python 开发人员最常用的工具如表 11.1 所示。

表 11.1 Python 开发人员常用工具

应用程序类型	工具
应用程序	Selenium（用于带 JavaScript 的 Web UI）
Web 应用程序	zope.testbrowser（不测试 JS）
WSGI 应用程序	paste.test.fixture（不测试 JS）
Gnome 桌面应用程序	dogtail
Win32 桌面应用程序	pywinauto

若想获得包含更多功能测试工具的列表，可以访问 Grig Gheorghiu 维护的一个 wiki 页面 —— <http://www.pycheesecake.org/wiki/ PythonTestingToolsTaxonomy>。

2. 单元测试

单元测试是完全适合采用 TDD 方法的低层级测试。它们关注于单个模块（如一个单元）

并且为之提供测试，它不涉及其他任何模块。测试将模块与应用程序的其余部分隔离开。当需要外部依赖项（诸如数据库访问）时，将由仿真或模拟对象来代替。

3. Python 标准测试工具

Python 在标准程序库中提供了两个用来编写测试的模块：

- `unittest` (<http://docs.python.org/lib/module-unittest.html>) 最初是 Steve Purcell 编写的（以前叫 PyUnit）；
- `doctest` (<http://docs.python.org/lib/module-doctest.html>) 一个文字测试工具。

(1) `unittest`

`unittest` 为 Python 提供的功能基本上和 JUnit 为 Java 所做的一切相同。它提供一个名为 `TestCase` 的基类，这个基类有一个用来验证调用输出的很大的方法集。

创建这个模块是为了编写单元测试，但是也可以用来编写验收测试，只要测试需要使用用户界面。有些测试框架提供了驱动工具的助手类，例如 `unittest` 之上的 Selenium。

要为一个模块编写一个简单的单元测试，首先从 `TestCase` 继承一个子类，然后再编写带有 `test` 前缀的方法。前面的示例，写出来的结果应该如下所示。

```
>>> import unittest
>>> class MyTests(unittest.TestCase):
...     def test_average(self):
...         self.assertEqual(average(1, 2, 3), 2)
...         self.assertEqual(average(1, -3), -1)
...         self.assertEqual(average(0, 1), 0.5)
...         self.assertRaises(TypeError, average)
...
>>> unittest.main()
.
-----
Ran 1 test in 0.000s
OK
```

`main` 函数扫描上下文并查找从 `TestCase` 继承的类。实例化这些类，然后运行所有以 `test` 开始的方法。

如果 `average` 函数在 `utils.py` 模块中，那么 `test` 类将被称为 `UtilsTests`，并且被写入 `test_utils.py` 文件中，如下所示。

```
import unittest
from utils import average
```

```

class UtilsTests(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average(1, 2, 3), 2)
        self.assertEqual(average(1, -3), -1)
        self.assertEqual(average(0, 1), 0.5)
        self.assertRaises(TypeError, average)
    if __name__ == '__main__':
        unittest.main()

```

现在，每当 utils 模块发生改变时，test_utils 模块就将获得更多的测试。

 为了正常工作，test_utils 模块需要有在上下文中可用的 utils 模块。这是因为两个文件在同一个文件夹中，或者测试运行程序将 utils 模块放在 Python 路径下。

Distutils develop 命令在这里很有帮助。

要在整个应用程序之上运行测试，那么将预先假定拥有一个脚本，这个脚本在所有测试模块之外构建一个测试活动（test campaign）。unittest 提供了一个 TestSuite 类，可以聚集多个测试并将它们作为一个测试活动来运行，只要它们都是 TestCase 或 TestSuite 的子类实例就行。

按照惯例，test 模块将提供一个 test_suite 函数，它将返回一个该模块在命令行下调用时用于 __main__ 部分，或者为测试运行程序所用的 TestSuite 实例，如下所示。

```

import unittest
from utils import average
class MyTests(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average(1, 2, 3), 2)
        self.assertEqual(average(1, -3), -1)
        self.assertEqual(average(0, 1), 0.5)
        self.assertRaises(TypeError, average)
class MyTests2(unittest.TestCase):
    def test_another_test(self):
        pass
def test_suite():
    """builds the test suite."""
    def suite(test_class):
        return unittest.makeSuite(test_class)

```

```

        suite = unittest.TestSuite()
        suite.addTests((suite(MyTests), suite(MyTests2)))
        return suite
if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')

```

从 shell 程序上运行这个模块将打印出测试活动的输出，如下所示。

```
$ python test_utils.py
```

```
..
```

```
Ran 2 tests in 0.000s
```

```
OK
```

一般来说，通过一个全局脚本就能运行所有的测试，它将在代码树上查找并运行测试。这被称作测试发现（test discovery），稍后将介绍它。

(2) doctest

doctest 是一个在交互式命令行会话中使用的模块，它用来从文本文件或 docstrings 中提取片段，并且回放它们，以检查实例中写入的输出与实际输出相同。

例如，下面这个文本文件（test.txt）可以作为测试来运行。

```

Check that the computer CPU is not getting too hot::
>>> 1 + 1
2

```

doctest 提供一些提取和运行测试的功能，如下所示。

```

>>> import doctest
>>> doctest.testfile('test.txt', verbose=True)
Trying:
    1 + 1
Expecting:
    2
ok
1 items passed all tests:
   1 tests in test.txt
1 tests in 1 items.
1 passed and 0 failed.
Test passed.
*** DocTestRunner.merge: 'test.txt' in both testers; summing outcomes.
(0, 1)

```

使用 doctest 有许多好处：

- 包可以通过示例创建文档和测试；
- 文档示例总是最新的；
- 使用 doctest 中的示例来编写包，对具备前一小节中描述的用户视角有帮助。

但是，不要用 doctest 来使单元测试过期，它们应该只被用于在文档中提供人类可读的示例。换句话说，当测试关注低层级事务或者需要会使文档混乱的复杂测试装置时，不要使用 doctest。

有些 Python 框架（如 Zope）广泛地应用 doctest，有时候不熟悉代码的人会批评它们。一些 doctest 确实令人难以阅读和理解，因为示例破坏了技术写作的一个规则：它们不能在一个简单的命令提示符下被理解和运行，并且要求读者具有大量的知识。所以，本来是用来帮助新来者的文档，却因为代码示例而使其难以阅读。这些都是基于复杂的测试装置甚至特殊的测试 API，通过 TDD 建立的 doctest。



正如本章中关于文档的说明那样，当使用作为包文档的一部分的 doctest 时，应小心遵循技术写作的 7 条规则。

至此，应该对 TDD 所带来的好处建立了较好的总体认识。如果你仍然没有信心，可以在少数模块上进行尝试。使用 TDD 编写一个模块并度量构建它的时间，然后调试并重构它。你应该很快发现这是一种优秀的方法。

11.2 我 测 试

本节将描述几个开发人员在编写测试时会碰到的问题，以及针对它们的解决方法。此外，还将简略介绍一下社区中可用的测试运行程序和工具。

11.2.1 Unittest 的缺陷

unittest 模块在 Python 2.1 中就被引入了，并且已经被开发人员大量使用。但是，曾经因为 unittest 的弱点和局限性而招致失败的人开发了一些替代的测试框架，并且其在社区中已经兴起。

以下是常见的批评。

- 该框架笨重，因为：
 - 必须在 TestCase 子类中编写所有的测试；

- 必须在方法名称前加上 test;
- 被要求使用 TestCase 提供的断言方法;
- 必须为所要运行的测试活动建立测试套件。
- 框架难以扩展，因为它需要使用许多类的继承或诸如装饰器（decorator）之类的技巧。
- 测试装置有时难以组织，因为 setUp 和 tearDown 机制绑定在 TestCase 级别上，但是它们在每个测试时运行一次。换句话说，如果测试装置关注许多测试模块，那么它的创建和清理工作将十分不容易。
- 在 Python 软件之上不容易运行测试活动，必须编写额外的脚本来收集测试，聚集并运行它。

为了在编写测试时免受框架的僵化特性（这点和 Junit 很相似），需要一种更轻量级的方法。因为 Python 不要求工作在一个 100% 的类环境中，所以提供一种不基于继承的更具 Python 风格的测试框架会更好。普通的方法是：

- 提供简单的将任何函数或类标记为测试的方法；
- 通过插件系统扩展框架；
- 为所有测试级别提供完整的测试装置环境，包括完整的活动、一组模块级和测试级的测试；
- 提供基于测试发现的测试运行程序，并提供大量的选项集。

Python 核心开发人员意识到 unittest 的弱点，并已经在 Python 3k 中完成了一些改进工作。



有人正在开发用于 Python 3k 的 unittest 替代品，其中个项目是基于 test_harness 实现的（参见 <http://oakwinter.com/code/>）。

11.2.2 Unittest 替代品

有些第三方工具尝试通过提供 unittest 扩展的形式解决刚才提到的问题。

最常用的如下。

- nose <http://www.somethingaboutorange.com/mrl/projects/nose>
- py.test <http://codespeak.net/py/dist/test.html>

1. nose

nose 主要是一个具有强大发现功能的测试运行程序。它拥有大量的选项，允许在 Python 应用程序中运行所有类型的测试活动。

可以使用 easy_install 安装它，如下所示。

```
$ easy_install nose
Searching for nose
Reading http://pypi.python.org/simple/nose/
Reading http://somethingaboutorange.com/mrl/projects/nose/
...
Processing dependencies for nose
Finished processing dependencies for nose
```

(1) 测试运行程序

安装完后，命令提示符下就多了一条 nosetests 命令。可以直接使用它来运行 11.1 节中介绍的测试，如下所示。

```
$ nosetests -v
test_average (test_utils.MyTests) ... ok
test_another_test (test_utils.MyTests2) ... ok
builds the test suite. ... ok
-----
Ran 3 tests in 0.010s
OK
```

nose 将递归浏览当前目录以发现测试，并创建一个自己的测试套件。这个简单的功能与 unittest 中启动测试所做的工作比起来已经是个优势。现在不需要为构建和运行测试活动准备样板代码了，唯一需要做的事就是编写测试类。

(2) 编写测试

nose 向前发展了一步，它将运行所有名称符合正则表达式 `(?:^|[b_-.])[Tt]est` 的类和函数，所在的模块也匹配该表达式。一般而言，所有以 test 开头并且位于符合该模式的模块中的可调用部件也将作为测试执行。

例如，`test_ok.py` 将被 nose 识别并运行，如下所示。

```
$ more test_ok.py
def test_ok():
    print 'my test'
$ nosetests -v
test_ok.test_ok ... ok
-----
Ran 1 test in 0.071s
OK
```

常规的 TestCase 类和 doctests 也会被执行。

最后, nose 提供和 TestCase 方法类似的断言函数。但是这些都将作为函数提供, 使用 PEP8 命名约定, 而不是 unittest 使用的 Java 约定 (参见 <http://code.google.com/p/python-nose/wiki/TestingTools>)。

(3) 编写测试装置

nose 支持 3 种级别的测试装置:

- 包级别 例如, setup 和 teardown 函数可以被添加到存储所有包的测试的文件夹的 __init__.py 模块中;
- 模块级 测试模块可以有自己的 setup 和 teardown 函数;
- 测试级别 可调用对象也可以提供使用 with_setup 装饰器的测试装置函数。

例如, 要在模块和测试级别设置一个测试装置, 可以使用以下代码。

```
def setup():
    # setup 代码, 为整个模块启动
    ...
def teardown():
    # tear down 代码, 为整个模块启动
    ...
def set_ok():
    # 只针对 test_ok 的 setup 代码
    ...
@with_setup(set_ok)
def test_ok():
    print 'my test'
```

(4) 与 setuptools 和插件系统的集成

最后, nose 与 setuptools 实现了平滑集成, 这样 test 命令就可以与之一起使用 (python setup.py test)。这可以通过在 setup.py 中添加 test_suite 元数据来完成, 示例如下。

```
setup(
    ...
    test_suite = 'nose.collector'
    ...)
```

nose 还为开发人员编写 nose 插件使用了 setuptools 入口点机制, 这使他们可以覆盖或者修改从测试发现到测试输出的工具的所有特征。



在 <http://nose-plugins.jottit.com> 中可以看到一个插件列表。



(5) 总结

`nose` 是一个完整的测试工具，修复了 `unittest` 存在的许多问题。不过它仍然使用了隐式的前缀名称，这为许多开发人员留下一个约束。但这个前缀是可以定制的，它仍然要求遵循一个约定。

这种在配置语句之上的约定挺不错的，比 `unittest` 所要求的样板代码要好得多。但是使用显式的装饰器，也可能是摆脱 `test` 前缀的一个好方法。

最后，插件方法使 `nose` 变得非常灵活，并允许开发人员定制该工具以符合他的要求。

可以在主目录添加一个`.noserc` 或 `nose.cfg` 文件，以指定 nosetests 启动时的默认选项。



一种好的方法是自动查找 `doctests`。

该文件的例子如下所示。

```
[nosetests]
with-doctest=1
doctest-extension=.txt
```

2. py.test

`py.test` 和 `nose` 非常相似，本小节将只介绍它的特点。这个工具被捆绑到包含其他工具的 `py` 包中。



`nose` 的灵感来自 `py.test`。

它也可以使用 `easy_install` 来安装，如下所示。

```
$ easy_install py
Searching for py
Best match: py 0.9.0
Finished processing dependencies for py
```

现在，命令提示符下将有一个新的命令 `py.test`，它可以和 `nosetests` 一样使用。这个工具和 `nose` 一样，使用一个模式匹配算法来捕捉需要运行的测试。该模式比 `nose` 使用得更严格，只捕捉：

- 在以 `test` 开头的文件中的以 `Test` 开头的类；
- 在以 `test` 开头的文件中的以 `test` 开头的函数。



请注意正确使用大小写：如果函数以大写的“T”开头，那么它将被当作一个类，从而被忽略；如果一个类以小写“t”开头，那么 py.test 将中断，因为它将尝试把其作为函数处理。

测试装置功能与 nose 类似，除了语义上有些不同之外。py.test 将从官方文档中查找每个测试模块中 3 个级别的测试装置。

```
def setup_module(module):
    """ setup up any state specific to the execution
       of the given module.
    """
    def teardown_module(module):
        """ teardown any state that was previously setup
            with a setup_module method.
        """
        def setup_class(cls):
            """ setup up any state specific to the execution
                of the given class (which usually contains tests).
            """
            def teardown_class(cls):
                """ teardown any state that was previously setup
                    with a call to setup_class.
                """
                def setup_method(self, method):
                    """ setup up any state tied to the execution of the given
                        method in a class. setup_method is invoked for every
                        test method of a class.
                    """
                    def teardown_method(self, method):
                        """ teardown any state that was previously setup
                            with a setup_method call.
                        """
    
```

每个函数将以当前模块、类或方法为参数。因此，测试装置将能够在上下文中工作，而不必查找它，这点和 nose 一样。但是 py.test 不像 nose 那样，提供了通过在包级别上添加 `setup` 和 `teardown` 函数来实现全局测试装置的方法。

py.test 独创的功能有：

- 禁用一些测试类的能力；
- 在多台电脑中分发测试的能力；
- 在该工具继续发现任务的同时立即开始测试。

(1) 禁用测试类

该工具提供了一个简单的机制，可以在某些条件下禁用一部分测试。如果在测试类上找到 `disabled` 特性，那么它就将被选中。

例如，正如其文档中所说的那样，当一个测试是依赖于具体平台时，那么这个布尔值将被设置（来自官方文档的例子）如下。

```
class TestEgSomePosixStuff:
    disabled = sys.platform == 'win32'
    def test_xxx(self):
        ...
```

可以使用一个可调用对象来提供复杂的条件，如下所示。

```
def _disabled():
    # 在此为复杂的工作
    return 0
class Test_2:
    disabled = _disabled()
    def test_one(self):
        pass
```

不幸的是，这个特性不能是方法或属性，因为 py.test 只在类上调用 `getattr(cls, 'disabled', 0)`。

(2) 自动分发的测试

py.test 还有一个有趣的功能，它可以在多台电脑上分发测试。只要可以通过 SSH 访问的机器，py.test 都能通过发送将要执行的测试来驱动每台电脑。

但是，这个功能取决于网络。如果网络连接被破坏，那么从机将无法继续工作，因为它完全是由主机驱动的。

当一个项目具有很长的测试活动时，Buildbot 方法是首选。但是，当遇到一个消费许多资源来运行测试的应用程序时，py.test 分布模型可以用来实现测试的 ad hoc 分布。

(3) 立即开始测试

py.test 在发现过程中可以采用一个迭代程序，也就是找到第一个测试时就可以直接启动。这加速了第一个测试输出，这在测试装置很慢的情况下是很好的。而且，第一个故障也将更快地发生。

(4) 总结

`py.test` 和 `nose` 非常相似，因为不需要样板代码来集合测试。测试装置也可以分层配置，但是没有提供在包级别上直接启动一个配置的方法。不幸的是，它也没有提供和 `nose` 一样的插件系统。

最后，`py.test` 关注于更快地建立测试，在这个领域上，它也确实比其他工具更优越。

 这个工具是一个更大框架的一部分，可以被独立分发以避免安装其他元素。

除非希望使用 `py.test` 的特殊功能，否则 `nose` 应该是首选。

11.2.3 仿真和模拟

在编写单元测试时，我们假设隔离了被测试的模块。测试向函数或方法提供一些数据并测试其输出。

这主要确保测试：

- 与应用程序的一个原子部件（可能是一个函数或类）有关；
- 提供确定的、可重复的结果。

有时候，程序中一个部件的隔离并不明显。例如，如果代码要发送邮件，那么它将调用 Python 中的 `smtplib` 模块，通过一个 `telnet` 连接与 SMTP 服务器协同工作。这不应该在测试运行时发生，理想的情况是，单元测试应该在任何没有外部依赖和副作用的电脑上运行。

由于 Python 的动态特性，使用 `monkey patches` 来修改测试装置的运行时代码是可能的，这可以仿真一个第三方代码库的行为。

1. 建立一个仿真品

测试中的仿真行为可以通过发现测试代码与外部协作所需的最小交互集来创建。然后，手工返回输出，或者使用已经记录的真实数据。

这可以从一个空的类或函数开始，并将它用作一个替代品。然后启动测试，不断填充仿真品直到它正确表现。

以发送邮件的 `mailer` 模块中的一个函数 `send` 为例，如下所示。

```
import smtplib
import email.Message
def send(sender, to, subject='None', body='None',
        server='localhost'):
```

```
"""sends a message."""
message = email.Message.Message()
message['To'] = to
message['From'] = sender
message['Subject'] = subject
message.set_payload(body)
server = smtplib.SMTP(server)
try:
    res = server.sendmail(sender, to, message.as_string())
finally:
    server.quit()
return res
```



在本小节中将用 nose 来示范仿真和模拟。



对应的测试可能如下所示。

```
from mailer import send
from nose.tools import *
def test_send():
    res = send('tarek@ziade.org', 'tarek@ziade.org',
               'topic', 'body')
    assert_equals(res, {})
```

只要在本地主机上有一个 SMTP 服务器，这个测试将通过并且有效。否则，它将失败，示例如下。

```
$ nosetests -v
test_mailer.test_send ... ERROR
=====
ERROR: test_mailer.test_send
-----
Traceback (most recent call last):
...
"....Versions/2.5/lib/python2.5/smtplib.py", line 310, in connect
    raise socket.error, msg
error: (61, 'Connection refused')
```

```
Ran 1 test in 0.169s
```

可以添加一个补丁来仿真 SMTP 类，如下所示。

```
from mailer import send
from nose.tools import *
import smtplib
def patch_smtp():
    class FakeSMTP(object):
        pass
    smtplib._SMTP = smtplib.SMTP
    smtplib.SMTP = FakeSMTP
def unpatch_smtp():
    smtplib.SMTP = smtplib._SMTP
    delattr(smtplib, '_SMTP')
@with_setup(patch_smtp, unpatch_smtp)
def test_send():
    res = send('tarek@ziade.org', 'tarek@ziade.org',
               'topic', 'body')
    assert_equals(res, {})
```

然后，再次运行测试，如下所示。

```
$ nosetests -v
test_mailer.test_send ... ERROR
=====
ERROR: test_mailer.test_send
-----
Traceback (most recent call last):
...
TypeError: default __new__ takes no parameters
-----
Ran 1 test in 0.066s
FAILED (errors=1)
```

接着，完成 FakeSMTP 类直到测试通过。它将报告类所应该拥有的几个方法，如下所示。

```
class FakeSMTP(object):
    def __init__(self, *args, **kw):
        # 我们不关心
```

```

    pass
def quit(self):
    pass
def sendmail(self, *args, **kw):
    return {}

```

当然，仿真的类可以和新的测试一起演化，以提供更加复杂的行为，但是它应该尽可能保持简短。相同的原则也适用于更复杂的输出，记录它们并通过仿真的 API 供应。这常常使用诸如 LDAP 或 SQL 这样的第三方服务器来完成。

仿真有实际的局限性。如果决定仿真一个外部相关对象，可能会引入缺陷或者出现真正的服务器不会出现的不必要行为，反之亦然。

2. 使用模拟

模拟对象是可以用于隔离测试过的代码的通用仿真对象（参见 http://en.wikipedia.org/wiki/Mock_object），它们能使输入输出的构建自动完成。在静态类型语言中，模拟对象有更大的用处，在 Python 这种动态语言中 monkey patch 比较困难。但是它们在 Python 中仍然有用，可以缩短模拟外部 API 的代码。

在 Python 中有许多可用的模拟库，最简单和最易用的是 Ian Bicking 的 `minimock`（参见 <http://blog.ianbicking.org/minimock.html>）。

它也很容易安装，如下所示。

```

$ easy_install minimock
Searching for minimock
Reading http://pypi.python.org/simple/minimock
...
Finished processing dependencies for minimock

```

这个库提供 3 个元素：

- `mock` 生成模拟对象并将其放入指定命名空间的函数；
- `Mock` 可以用来手工实例化一个模拟对象的模拟类；
- `restore` 删除 `mock` 所打补丁的函数。

在我们的示例中，使用 `minimock` 来修补 SMTP，要比手工仿真更为简单，如下所示。

```

from mailer import send
from nose.tools import *
import smtplib
from minimock import mock, restore, Mock
def patch_smtp():

```

```

    mock('smtplib.SMTP',
        returns=Mock('smtp_connection',
            sendmail=Mock('sendmail',
                returns={}))
        )
    )

def unpatch_smtp():
    restore()

@with_setup(patch_smtp, unpatch_smtp)
def test_send():
    res = send('tarek@ziade.org',
               'tarek@ziade.org', 'topic', 'body')
    assert_equals(res, {})

```

`returns` 特性允许定义调用返回的元素。当使用模拟对象时，每当一个特性被代码调用，它将立刻为该特性创建一个新的模拟对象，所以不会抛出异常。这是先前编写的 `quit` 方法的情况。

如果一个方法必须返回一个特定值，那么模拟对象可以为那个方法手工实例化，在其 `returns` 参数中返回该值。这个特殊的模拟对象可以作为关键字参数传递。在 `sendmail` 中就做到了这一点。

现在，再次运行测试，如下所示。

```

$ nosetests -v -s
test_mailer.test_send ... Called smtplib.SMTP('localhost')
Called sendmail(
    'tarek@ziade.org',
    'tarek@ziade.org',
    'To: tarek@ziade.org\nFrom: tarek@ziade.org\nSubject: topic\n\nbody')
Called smtp_connection.quit()
ok
-----
Ran 1 test in 0.122s
OK

```

注意，被调用的元素由 `minimock` 打印输出。这使之成为 `doctest` 的一个良好候选者。

11.2.4 文档驱动开发

`doctest` 是 Python 与其他语言相比的真正优势所在。文本使用的代码实例也可以作为测试运行这一事实改变了 TDD 的实施方式。例如，文档的一部分可以在开发周期中通过 `doctests`

来完成。这个方法也确保了所提供的示例是最新的并且确实可以工作。

通过 doctest 而不是常规的单元测试来构建软件，被称为文档驱动开发（DDD）。开发人员以普通的英语来说明代码的作用，同时实现它。

编写一个故事

在 DDD 中，编写 doctest 是由构造一个关于代码块如何工作和如何使用的故事来完成的。原则用普通的英语描述，然后在文本中放置几个代码使用场景。一个好的方法是从编写代码如何工作的文本开始，然后添加一些代码实例。这是第 6 章中编写模块的方式。

回过头看看 atomisator.parser 包的 doctest，第一个版本的文本如下所示。

```
=====
atomisator.parser
=====

The parser knows how to return a feed content with
a function available as a top-level function.
This function takes the feed url and returns an iterator
on its content. A second parameter can specify how
many entries have to be returned before the iterator is
exhausted. If not given, it is fixed to 10
```

接着完成该实例，由此构建的代码如下。

```
=====
atomisator.parser
=====

The parser knows how to return a feed content, with
the 'parse' function, available as a top-level function::
>>> from atomisator.parser import parse
This function takes the feed url and returns an iterator
on its content. A second parameter can specify how
many entries have to be returned before the iterator is
exhausted. If not given, it is fixed to 10::

>>> res = parse('http://example.com/feed.xml')
>>> res
<generator ...>
```

稍后，doctest 将可能继续演化，以考虑新的元素或必需的更改。这个 doctest 对于希望使

用该包的开发人员也是一个良好的文档，应该记得用这个方法来修改。

在文档中编写测试的常见缺点是使其变成一个无法理解的文本块。如果发生这种情况，它就不应该被作为文档的一部分。

也就是说，一些专门通过 doctest 进行工作的开发人员常常将他们的 doctest 分成两类：一种是易于理解和使用的，它们可以作为包文档的一部分；另一种是不可理解的，只能用于构建和测试软件。

许多开发人员认为后一种情况中 doctest 应该被放弃，而赞同使用常规的单元测试。另一些人甚至将 doctest 专用于缺陷修复。

所以，在 doctest 和常规测试之间的平衡是品味问题，取决于团队，只要发布的 doctest 是易读的就行。



在一个项目中使用 DDD 时，应注意可读性并决定哪些 doctest 符合成为发
行文档一部分的判断条件。

11.3 小 结

本章提倡使用 TDD，并提供了更多关于以下方面的信息：

- unittest 的缺点；
- 第三方工具——nose 和 py.test；
- 如何构建仿真和模拟；
- 文档驱动开发。

下一章将关注于优化程序的方法。

第 12 章

优化：通用原则和剖析技术

“过早进行优化是编程中的万恶之源。”

——Donald Knuth

本章是关于优化的，将介绍一组通用原则和剖析技术。它讲解了开发人员应该掌握的 3 条规则，并提供了优化的指南。最后，它还将关注于如何查找瓶颈。

12.1 优化的 3 条规则

不管结果如何，优化工作都是有代价的。当代码工作正常时，不去理会它（有时）可能比不惜一切代价尝试使它运行得更快要好一些。进行优化时，需要记住几条原则：

- 首先要使它能够正常工作；
- 从用户的观点进行；
- 保持代码易读。

12.1.1 首先使它能够正常工作

尝试在编写代码的同时对其进行优化，是最常见的错误。这是不可能的，因为瓶颈常常会出现在意想不到的地方。

应用程序是由非常复杂的交互组成的，在实际使用之前不可能得到一个完整的视图。

当然，这不是不尝试尽可能编写更快的函数或方法的理由。应该认真地使其复杂度尽可能地降低，并避免无用的重复。但是，首要目标是使它能够正常工作，优化不应该阻碍这一目标的实现。

对于行级的代码，Python 的哲学是完成一个目标尽可能有且只有一个方法。所以，只要坚持第 2 章和第 3 章中介绍的 Python 风格的语法，代码就应该会很好。往往，编写的代码越少，代码就越好并且越快。

在使代码能够正常工作并且做好剖析的准备之前，不要做以下的事情：

- 开始编写为函数缓存数据的全局字典；
- 考虑以 C 语言或诸如 Pyrex 之类混杂语言来对代码的一个部分进行扩展；
- 寻找外部程序库来完成一些基本计算。

对于非常专业的程序，如科学计算程序或游戏，专业的程序库和扩展的使用从一开始都无法避免。另一方面，使用像 Numeric 这样的程序库，可以简化专业功能的开发并且能够生成更简单、更快速的代码。而且，不应该在有很好的程序库可利用的时候自己重新编写一个函数。

例如，Soya 3D 是一个基于 OpenGL 的游戏引擎（参见 <http://home.gna.org/oomadness/en/soya3d/index.html>），它是使用 C 和 Pyrex 开发的，用于在显示实时的 3D 图形时执行快速的矩阵操作。

优化应该在已经能够正常工作的程序上进行。

“先让它能够正常工作，然后让它变得更好，最好使它更快。”

—— Kent Beck

12.1.2 从用户的观点进行

我曾经看到过一些团队对应用程序服务器的启动时间进行优化，之后这些应用程序服务器在启动时真地表现得很好。他们提升了速度之后，就向他们的客户推销这一改进。但是，他们落空了，因为客户对此并不关心。这是因为速度提升的优化并不是由客户的反馈推动的，而是出自于开发人员的观点。构建该系统的人每天都要启动服务器，所以启动时间对他们很重要，但是对客户并不重要。

虽然程序能够更快地启动绝对是件好事，但是团队应该认真地安排优化的优先级，并且问自己以下问题：

- 客户是否要求提升它的速度？
- 谁发现程序慢了？
- 它真的很慢，还是可以接受？
- 提升它的速度需要多少成本？值得吗？哪部分需要提升速度？

记住，优化是有成本的，开发人员的观点对客户来说也许并没有意义，除非编写的是框架或程序库，客户也是开发人员。



优化不是一个游戏，它应该只在必要时进行。



12.1.3 保持代码易读（从而易于维护）

尽管 Python 尝试使公共的代码模式能够最快地执行，但优化工作可能使代码变得混乱，并且使它难以阅读。在生成易读从而易于维护的代码和使其运行速度更快之间，要找到一个平衡点。

当达到优化目标的 90% 时，如果剩下的 10% 会使代码完全无法理解，那么，停止优化工作并且寻求其他解决方案可能是个好主意。



优化不应该使代码难以理解。如果发生这种情况，应该寻求替代的方案，如扩展或重新设计。不过，在代码易读性和速度上总会有一个好的折衷方案。



12.2 优化策略

假设程序有一个需要解决的速度问题，不要尝试猜测如何才能够提升它的速度。瓶颈往往难以在代码中找到，需要一组工具来找到真正的问题。

一个好的优化策略可以从以下 3 个步骤开始。

- 寻找其他原因：确定第三方服务器或资源不是问题所在。
- 度量硬件：确定资源足够用。
- 编写速度测试：创建带有速度要求的场景。

12.2.1 寻找其他原因

往往，性能问题会发生在生产环境中，客户会提醒软件的运行和测试环境中不一样。性能问题可能是因为应用程序没有考虑到现实世界中用户数或数据量不断增长的情况。

但是，如果应用程序存在与其他应用程序之间的交互，那么首先要做的是检查瓶颈是否出现在这些交互上。例如，数据库服务器或 LDAP 服务器可能带来额外的开销，并且使得速度变慢。

应用程序之间的物理链接也应该考虑：可能应用程序服务器和其他企业内网中的服务器

之间的网络链接因为错误配置而变慢，或者一个多疑的防病毒软件对所有 TCP 包进行扫描而导致速度降低。

设计文档应该提供描述所有交互和每个链接特性的一个图表，以获得对系统的全面认识，并且在解决速度问题时提供帮助。

 如果应用程序使用了第三方服务器或资源，那么应该评估每个交互，以确定瓶颈不在那里。

12.2.2 度量硬件

当内存不够用时，系统会使用硬盘来存储数据，这也就是页面交换（swapping）。

这会带来许多的开销，性能将急剧下降。从用户的角度看，系统在这种时候会被认为处于死机状态。所以，度量硬件的能力以避免这种情况是很重要的。

虽然系统上有足够的内存很重要，但是确保应用程序不要疯狂表演，并吞噬太多内存也是很重要的。例如，如果程序将使用数百兆大小的视频文件，那么它不应该将该文件全部装入内存中，而应该分块载入或者使用磁盘流。

磁盘的使用也很重要。如果代码中隐藏 I/O 错误，尝试重复写入磁盘，那么一个已满的分区也可能降低应用程序的速度。而且，即使代码只写入一次，硬件和操作系统也可能会尝试多次写入。

 Munin 是一个很好的系统监控工具，可以用它来获得系统健康情况的一个快照，在`http://munin.projects.linpro.no`上可以找到它。应确保系统健康并且符合应用程序需求。不过，仍应确保应用程序不要像怪物一样消费内存和磁盘空间。

12.2.3 编写速度测试

开始优化工作时，应该在测试程序上花工夫，而不是不断进行手工的测试。一个好的做法是在应用程序中专门制作一个测试模块，编写一系列针对优化的调用。这种方案可在优化应用程序的同时跟踪进度。

甚至可以编写一些断言，在这里设置一些速度要求。为了避免速度退化，这些测试可以留到代码被优化之后，如下所示。

```
>>> def test_speed():
...     import time
...     start = time.time()
...     the_code()
...     end = time.time() - start
...     assert end < 10, \
...         "sorry this code should not take 10 seconds !"
...
...
```

 度量执行速度取决于 CPU 的运行能力。在下一节将看到如何编写通用的持续时间度量方法。

12.3 查找瓶颈

查找瓶颈可以通过以下几个步骤来完成：

- 剖析 CPU 的使用情况；
- 剖析内存的使用情况；
- 剖析网络的使用情况。

12.3.1 剖析 CPU 的使用情况

瓶颈的第一个来源是代码本身。标准程序库提供了执行代码剖析所需的所有工具，它们都是基于确定性方法的。

 确定性剖析程序通过在最低层级上添加一个定时器来度量每个函数所花费的时间。这将带来一些开销，但是它是获取时间消耗信息的一个好思路。另一方面，统计剖析程序将采样指令指针的使用情况而不采样代码。后者比较不精确，但是能够全速运行目标程序。

剖析代码有以下两种方法：

- 宏观剖析 在使用的同时剖析整个程序，并生成统计；

- 微观剖析 人工执行以度量程序中某个确定的部分。

1. 宏观剖析

宏观剖析通过在特殊的模式下运行应用程序来完成，在这种模式下解释程序将收集代码使用情况的统计信息。Python 为此提供了多个工具：

- `profile` 一个纯 Python 实现的工具；
- `cProfile` 用 C 实现的，界面和 `profile` 相同，开销更小；
- `hotshot` 另一个用 C 实现的工具，可能会被从标准程序库中删除。

除非程序是在 Python 2.5 以下版本中运行，否则建议使用 `cProfile`。

以下是拥有一个主函数的 `myapp.py` 模块。

```
import time

def medium():
    time.sleep(0.01)

def light():
    time.sleep(0.001)

def heavy():
    for i in range(100):
        light()
        medium()
        medium()

    time.sleep(2)

def main():
    for i in range(2):
        heavy()

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

该模块可以直接从提示符调用，在此摘要显示了它的执行结果。

```
$ python -m cProfile myapp.py
      1212 function calls in 10.120 CPU seconds

Ordered by: standard name

ncalls  tottime cumtime percall file
      1    0.000   10.117  10.117  myapp.py:16(main)
    400    0.004    4.077   0.010  myapp.py:3(lighter)
    200    0.002    2.035   0.010  myapp.py:6(light)
      2    0.005   10.117   5.058  myapp.py:9(heavy)
```

```
3    0.000  0.000  0.000 {range}
602  10.106 10.106  0.017 {time.sleep}
```

所提供的统计是一个由剖析程序生成的统计对象的打印视图。手工调用该工具的结果可能如下所示。

```
>>> from myapp import main
>>> import cProfile
>>> profiler = cProfile.Profile()
>>> profiler.runcall(main)
>>> profiler.print_stats()

      1209 function calls in 10.140 CPU seconds

Ordered by: standard name

  ncalls  tottime cumtime  percall     file
        1    0.000  10.140   10.140  myapp.py:16(main)
      400    0.005   4.093    0.010  myapp.py:3(medium)
      200    0.002   2.042    0.010  myapp.py:6(light)
        2    0.005  10.140   5.070  myapp.py:9(heavy)
        3    0.000    0.000    0.000  {range}
     602   10.128  10.128    0.017  {time.sleep}
```

统计也可以保存到一个文件中，然后由 `pstats` 模块阅读。这个模块提供一个处理剖析文件的类，并给出了一些诸如排序之类的助手方法，如下所示。

```
>>> cProfile.run('main()', 'myapp.stats')
>>> import pstats
>>> p = pstats.Stats('myapp.stats')
>>> p.total_calls
1210
>>> p.sort_stats('time').print_stats(3)
Thu Jun 19 23:56:08 2008      myapp.stats
      1210 function calls in 10.240 CPU seconds

Ordered by: internal time

List reduced from 8 to 3 due to restriction <3>
  ncalls  tottime cumtime percall filename:lineno(function)
     602   10.231  10.231   0.017  {time.sleep}
       2    0.004   10.240   5.120  myapp.py:9(heavy)
     400    0.004   4.159   0.010  myapp.py:3(medium)
```

由此，可以打印输出每个函数的调用者和被调用者，以便浏览代码，如下所示。

```

>>> p.print callees('medium')
Ordered by: internal time
List reduced from 8 to 1 due to restriction <'medium'>
Function           called...
                  ncalls  tottime cumtime
myapp.py:3(lighter) ->      400    4.155   4.155  {time.sleep}

>>> p.print callers('light')
Ordered by: internal time
List reduced from 8 to 2 due to restriction <'light'>
Function      was called by...
                  ncalls  tottime cumtime
myapp.py:3(medium) <-    400    0.004   4.159  myapp.py:9(heavy)
myapp.py:6(light)  <-    200    0.002   2.073  myapp.py:9(heavy)

```

对输出进行排序，可以从不同的视图来查找瓶颈。例如：

- 当调用数量确实很高，并且占据了最多的时间，那么该函数或方法可能进入了一个循环，可以尝试对其进行优化；
- 当一个函数运行时间很长，如果可能，缓存可能是个不错的选择。

另一个从剖析数据中显示瓶颈的好办法是将其转化为图表。用 Gprof2Dot 工具 (<http://code.google.com/p/jrfonseca/wiki/Gprof2Dot>) 就可以将剖析数据以散点图形式展现（如图 12.1 所示）。可以从<http://jrfonseca.googlecode.com/svn/trunk/gprof2dot/gprof2dot.py> 下载这个简单的脚本，然后在统计时使用它，只要安装了 Graphviz 就可以（参见<http://www.graphviz.org/>）。

```

$ wget http://jrfonseca.googlecode.com/svn/trunk/gprof2dot/
gprof2dot.py
$ python2.5 gprof2dot.py -f pstats myapp.stats | dot -Tpng -o output.png

```

 KcacheGrind 也是一个很好的可视化显示剖析数据的工具（参见 <http://kcachegrind.sourceforge.net/cgi-bin/show.cgi>）。

宏观剖析对于发现有问题的函数，或者至少是与问题临近的函数而言是很好的方法。当发现问题时，可以转用微观剖析。

2. 微观剖析

当找到速度很慢的函数时，有时还必须做更多只测试程序某个部分的剖析工作。这需要

通过手动对一部分代码进行速度测试来完成。

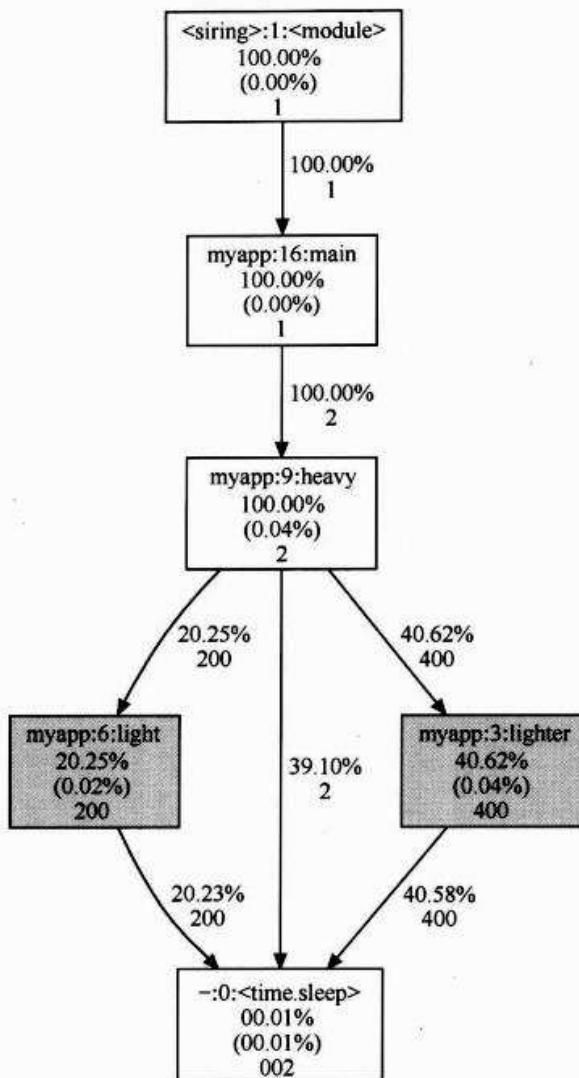


图 12.1

例如，可以在一个装饰器中使用 cProfile，如下所示。

```

>>> import tempfile, os, cProfile, pstats
>>> def profile(column='time', list=5):
...     def _profile(function):
...         def __profile(*args, **kw):
...             s = tempfile.mktemp()
...             profiler = cProfile.Profile()
...             profiler.runcall(function, *args, **kw)
...             profiler.dump_stats(s)
...             p = pstats.Stats(s)
  
```

```
...           p.sort_stats(column).print_stats(list)
...
...       return __profile
...
...       return __profile
...
...
>>> from myapp import main
>>> @profile('time', 6)
... def main_profiled():
...     return main()
...
...
>>> main_profiled()
Fri Jun 20 00:30:36 2008 ...
    1210 function calls in 10.129 CPU seconds

Ordered by: internal time

List reduced from 8 to 6 due to restriction <6>
ncalls  tottime  cumtime  percall  filename:lineno(function)
      602   10.118   10.118   0.017  {time.sleep}
          2    0.005   10.129   5.065  myapp.py:9(heavy)
        400    0.004    4.080   0.010  myapp.py:3(lighter)
        200    0.002    2.044   0.010  myapp.py:6(light)
          1    0.000   10.129  10.129  myapp.py:16(main)
          3    0.000    0.000   0.000  {range}

>>> from myapp import lighter
>>> p = profile()(lighter)
>>> p()
Fri Jun 20 00:32:40 2008      /var/folders/31/
31iTrMYWHny8cxfjH5VuTk+++/TI/-Tmp-/tmpQjstAG
    3 function calls in 0.010 CPU seconds

Ordered by: internal time
ncalls  tottime  cumtime  percall  filename:lineno(function)
      1    0.010    0.010   0.010  {time.sleep}
      1    0.000    0.010   0.010  myapp.py:3(lighter)
```

这种方法能够对应用程序的各部分进行测试，使统计的输出更加精确。

但是在这个阶段，被调用者的列表可能不太令人感兴趣，因为函数已经被作为需要优化的部分。唯一令人感兴趣的是它有多快，然后改进它。

`timeit` 更适合这一需求，它提供一个简单的度量小型代码片断执行时间的方法，它将使用宿主系统提供的最好的底层定时器（`time.time` 或 `time.clock`），如下所示。

```
>>> from myapp import light
>>> import timeit
>>> t = timeit.Timer('main()')
>>> t.timeit(number=5)
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
10000000 loops, best of 3: 0.0268 usec per loop
10000000 loops, best of 3: 0.0269 usec per loop
5.6196951866149902
```

该模块允许重复调用，它适用于对已隔离的代码片断进行测试。这在应用程序上下文之外非常有用，如在命令提示符下，但是在一个现有的应用程序中使用其并不是很方便。

 确定性剖析程序提供的结果依赖于运行它的电脑，因此每次生成的结果都不一样。重复相同的测试并且取其平均值能提供更精确的结果。而且，有些电脑提供了特殊的 CPU 特性，如 SpeedStep。如果电脑在测试启动时处于闲置状态，也将会改变结果。所以对于小的代码片断持续地重复测试是一种好习惯。另外，不忘了有各种缓存，如 DNS 或 CPU 缓存。

和上面相似的装饰器（decorator）是计算应用程序某部分执行时间的更简单方法。这个装饰器将收集程序运行的持续时间，如下所示。

```
>>> import time
>>> import sys
>>> if sys.platform == 'win32':          # same condition in timeit
...     timer = time.clock
... else:
...     timer = time.time
>>> stats = {}
>>> def duration(name='stats', stats=stats):
...     def _duration(function):
...         def __duration(*args, **kw):
...             start_time = timer()
...             try:
```

```

...
        return function(*args, **kw)
...
    finally:
        stats[name] = timer() - start_time
...
    return __duration
...
return __duration

...
>>> from myapp import heavy
>>> heavy = duration('this_func_is')(heavy)
>>> heavy()
>>> print stats['this_func_is']
1.50201916695

```

在执行代码时，装饰器将填充 `stats` 词典的内容，这在函数执行完成之后就可以阅读。

使用这样一个装饰器，可以在应用程序中添加嵌入的测试机制，而且不会破坏应用程序本身。

```

>>> stats = {}
>>> from myapp import light
>>> import myapp
>>> myapp.light = duration('myapp.light')(myapp.light)
>>> myapp.main()
>>> stats
{'myapp.light': 0.05014801025390625}

```

这可以在速度测试的上下文中完成。

3. 测量 Pystones

测量执行时间时，结果取决于电脑硬件。为了能产生一个通用的度量值，最简单的方法是测量固定代码序列的基准速度，从而得出一个系数。由此，函数所耗费的时间就可以被转换为一个通用值，以便与其他电脑进行比较。



可用的基准工具有许多，例如创建于 1972 年的 Whetstone，它是一个用 Algol60 编写的电脑性能分析程序（参见 http://en.wikipedia.org/wiki/Whetstone_%28benchmark%29）。它将测量每秒能够执行几百万个 Whetstone 指令(MWIPS)。它在 <http://freespace.virgin.net/roy.longbottom/whetstone%20results.htm> 上维护了一个结果列表。

Python 在 test 包中提供了一个基准测试工具，用来测量一个精心挑选的操作序列的执行消耗时间。结果是电脑每秒能够执行的 `pystones` 数量，以及用于执行基准测试的时间——在当今的硬件环境中的结果通常在 1 秒左右，如下所示。

```
>>> from test import pystone
>>> pystone.pystones()
(1.0500000000000007, 47619.047619047589)
```

这个系数可以用来将剖析得到的消耗时间转化为 `pystones` 数，如下所示。

```
>>> from test import pystone
>>> benchtime, pystones = pystone.pystones()
>>> def seconds_to_kpystones(seconds):
...     return (pystones*seconds) / 1000
...
...
...
>>> seconds_to_kpystones(0.03)
1.4563106796116512
>>> seconds_to_kpystones(1)
48.543689320388381
>>> seconds_to_kpystones(2)
97.087378640776762
```

`seconds_to_kpystones` 将返回 kilo `pystones` 数（即千个 `pystones`）。这个转换可以包含在 `duration` 装饰器中，以求得以 `stones` 为单位表示的值。

```
>>> def duration(name='stats', stats={}):
...     def _duration(function):
...         def __duration(*args, **kw):
...             start_time = timer()
...             try:
...                 return function(*args, **kw)
...             finally:
...                 total = timer() - start_time
...                 kstones = seconds_to_kpystones(total)
...                 stats[name] = total, kstones
...             return __duration
...     return _duration
>>> @duration()
... def some_code():
```

```

...      time.sleep(0.5)

...
>>> some_code()
>>> stats
{'stats': (0.50012803077697754, 24.278059746455238)}

```

有了 `pystones` 值，就可以在测试中使用这个装饰器，这样可以在执行时间上设置断言。这些测试将可以在任何电脑上运行，从而使开发人员能够有效避免速度退化。当应用程序的一部分被优化之后，他们能在测试中设置其最大执行时间，以确定进一步的修改没有对它产生影响。

12.3.2 剖析内存使用情况

另一个问题是内存消耗。如果一个程序在运行时消耗太多的内存而导致系统出现页面交换，那么可能是应用程序创建了太多的对象。这往往很容易通过经典的剖析来发现，因为如果程序对内存消耗过大，以至于使系统需要进行页面交换的程度，那么也就会检测到 CPU 所要执行的大量工作。但是有时候它也不明显，因此必须对内存的使用情况进行剖析。

1. Python 处理内存的方式

使用 CPython 时，内存的使用可能是最难剖析的。虽然 C 这样的语言允许获得任何元素所占用的内存大小，但 Python 从不会让你知道指定对象消耗了多少内存。这是因为该语言所具有的动态特性，以及对象实例化的自动管理——垃圾回收。例如，两个指向相同字符串的变量在内存中不一定也指向相同的字符串对象。

这种内存管理器所采用的方法大约是基于这样一个简单的命题：如果指定的对象不再被引用，那么将删除它。所有函数的局部引用在解释程序出现下述情况时将被删除：

- 离开该函数；
- 确定该对象不再需要使用。

 在正常情况下，收集器能够工作得很好，但是可以使用 `del` 调用来协助垃圾回收手动删除对象的引用。

所以留在内存中的对象将是：

- 全局对象；
- 仍然以某种方式被引用的对象。

对参数输入输出的边界值要小心。如果一个对象是在参数中创建的，并且函数返回了该对象，那么该参数引用将仍然存在。如果它被作为默认值使用，可能导致不可预测的结果，如下所示。

```
>>> def my_function(argument={}):    #不良实践
...     if '1' in argument:
...         argument['1'] = 2
...     argument['3'] = 4
...     return argument
...
>>> my_function()
{'3': 4}
>>> res = my_function()
>>> res['4'] = 'I am still alive!'
>>> print my_function()
{'3': 4, '4': 'I am still alive!'}
```

这就是始终使用非易变对象的原因，示例如下。

```
>>> def my_function(argument=None): # 较好的实践
...     if argument is None:
...         argument = {}           # 每次都将创建一个刷新的词典
...     if '1' in argument:
...         argument['1'] = 2
...     argument['3'] = 4
...     return argument
...
>>> my_function()
{'3': 4}
>>> res = my_function()
>>> res['4'] = 'I am still alive!'
>>> print my_function()
{'3': 4}
```

垃圾回收给开发人员带来了方便，可以避免跟踪对象并且手工销毁它们。但是这将引入其他问题，因为开发人员从不在内存中清除对象实例，因此如果不注意使用数据结构的方式，可能会变得不可控制。

通常消耗内存的是：

- 增长速度不可控制的缓存；

- 注册全局对象并且不跟踪其使用的对象工厂，如数据库连接创建器，每当查询被调用时将立刻执行；
- 没有正常结束的线程；
- 具有 `__del__` 方法并且涉及循环的对象也是主要的内存消耗者。Python 垃圾回收不会打断该循环，因为它不能确定对象是否应该被先删除。因此，将导致内存泄漏。在任何情况下使用这个方法都是坏主意。

2. 剖析内存

想知道垃圾回收控制了多少对象，以及它实际的大小是需要技巧的。例如，要知道指定的对象有多少字节，这就涉及到统计其所有的特性，处理交叉引用，然后累加所有值。如果再考虑对象间可能存在的互相引用，这将是个相当困难的问题。`gc` 模块不提供高层级的函数，并且要求 Python 在调试模式下编译，以获得完整的信息。

编程人员往往只查询在指定操作执行前后他们应用程序的内存使用情况。但是这一度量只是个近似值，而且很大程度上依赖于系统级别上的内存管理方式。例如，在 Linux 下使用 `top` 命令或者在 Windows 下使用任务管理器都可能发现明显的内存问题，但是要跟踪错误的代码块，就需要在代码方面进行令人痛苦的工作。

幸运的是，有一些工具可用来创建内存快照，并计算载入对象的大小。但是请记住，Python 不会简单地释放内存，而是继续保存在内存中以防再次需要。

(1) Guppy-PE 入门

Guppy-PE (<http://guppy-pe.sourceforge.net>) 是一个框架，提供了被称为 Heap 的内存剖析程序及其他特性。

Guppy 可以通过 `easy_install` 来安装，命令如下。

```
$ sudo easy_install guppy
```

现在，在 `guppy` 命名空间下就可以使用 `hpy` 函数了。它将返回一个对象，该对象能够显示出内存的一个快照，如下所示。

```
>>> from guppy import hpy
>>> profiler = hpy()
>>> profiler.heap()

Partition of a set of 22723 objects. Total size = 1660748 bytes.

      Index Count %     Size % Cumulative % Kind (class / dict of
      class)

0          9948    44   775680   47   775680   47     str
1          5162    23   214260   13   989940   60     tuple
2          1404     6   95472     6   1085412   65 types.CodeType
```

```

3       61      0  91484     6 1176896  71  dict of module
4      152      1  84064     5 1260960  76  dict of type
5     1333      6  79980     5 1340940  81  function
6      168      1  72620     4 1413560  85  type
7      119      1  68884     4 1482444  89  dict of class
8       76      0  51728     3 1534172  92  dict (no owner)
9      959      4  38360     2 1572532  95  __builtin__.wrapper_
descriptor
<43 more rows. Type e.g. '_more' to view.>

```

这个输出展示了当前内存的使用情况，并且按照大小排序，按照对象类型分组。该对象提供了许多特性，可以用来定义列表显示和使用方式，就像显示 CPU 时间信息的 pstats 那样。可以使用 iso 方法来度量具体对象的大小，如下所示。

```

>>> import random
>>> def eat_memory():
...     memory = []
...     def _get_char():
...         return chr(random.randint(97, 122))
...     for i in range(100):
...         size = random.randint(20, 150)
...         data = [_get_char() for i in xrange(size)]
...         memory.append(''.join(data))
...     return '\n'.join(memory)
...
>>> profiler.iso(eat_memory())
Partition of a set of 1 object. Total size = 8840 bytes.
  Index  Count   %  Size   % Cumulative   % Kind
    0      1 100   8840   100     8840   100 str
>>> profiler.iso(eat_memory().__add__(eat_memory()))
Partition of a set of 1 object. Total size = 17564 bytes.
  Index  Count   %  Size   % Cumulative   % Kind
    0      1 100   17564   100     17564   100 str

```

`setrelheap` 方法用来重置这个剖析程序，这样可以使用 `heap` 捕获独立代码块的内存使用情况。但是这个初始化并不完美，因为上下文总是有多个载入的元素。这就是刚刚初始化的 `hpy` 实例不是 0 的原因，如下所示。

```
>>> g = hpy()
```

```
>>> g.setrelheap()
>>> g.heap().size
1120
>>> g.heap().size
1200
>>> g.heap().size
1144
```

`heap` 方法返回一个提供以下信息的 `Usage` 对象。

- `size` 以字节表示的内存总消耗量；
- `get_rp()` 告知对象代码在模块中位置的一个很好的遍历方法；
- 多种结果的排序方法，如 `bytype`。

(2) 使用 Heapy 跟踪内存使用情况

`Heapy` 不容易使用，需要一些练习。本小节将只介绍一些可以用来提供内存使用信息的简单函数。由此，跟踪代码必须通过 `Heapy` API 浏览对象来完成。

 在 `pkgcore` 项目网站上有关于 `Heapy` 如何用来跟踪内存使用情况的一个很好的实例，参见 <http://www.pkgcore.org/trac/pkgcore/doc/dev-notes/heapy.rst>。
这是使用该工具一个很好的出发点。

可以对 `duration` 装饰器（前面已经提出）做些修改，以使其提供函数使用的内存大小（以字节表示）。这个信息可以在词典中和持续时间、`pystone` 值一起读出。在 `profiler.py` 模块中的完整装饰器如下所示。

```
import time
import sys
from test import pystone
from guppy import hpy
benchtime, stones = pystone.pystones()
def secs_to_kstones(seconds):
    return (stones*seconds) / 1000
stats = {}
if sys.platform == 'win32':
    timer = time.clock
else:
    timer = time.time
```

```

def profile(name='stats', stats=stats):
    """Calculates a duration and a memory size."""
    def _profile(function):
        def __profile(*args, **kw):
            start_time = timer()
            profiler = hpy()
            profiler.setref()
            # 12是调用了setref之后的初始内存大小
            start = profiler.heap().size + 12
            try:
                return function(*args, **kw)
            finally:
                total = timer() - start_time
                kstones = secs_to_kstones(total)
                memory = profiler.heap().size - start
                stats[name] = {'time': total,
                               'stones': kstones,
                               'memory': profiler.heap().size}
        return __profile
    return _profile

```

变量 `start` 用来确认所计算的内存不包含调用 `setrelheap` 时 `Heapy` 所消耗的那部分内存。

使用具有 `eat_memory` 的装饰器，除了以秒表示的持续时间和 `pystones` 值以外，还将提供函数消耗的内存数量，如下所示。

```

>>> import profiler
>>> import random
>>> eat_it = profiler.profile('you bad boy!')(eat_memory)
>>> please = eat_it()
>>> profiler.stats
{'you bad boy!': {'stones': 14.306935999128555, 'memory': 8680,
                  'time': 0.30902981758117676}}

```

当然，如果多次运行它，那么在涉及非易变的对象时将导致不同的内存大小。但是，它仍然是个良好的指示器。

`Heapy` 的另一种有趣的用法是检查函数是否释放了它所使用的内存，例如在它产生缓存或注册的元素时。这可以通过重复函数调用并观察使用的内存是否增长来实现。

以下就是一个用于该目的的简单函数。

```

>>> REPETITIONS = 100
>>> def memory_grow(function, *args, **kw):
...     """checks if a function makes the memory grows"""
...     profiler = hpy()
...     profiler.setref()
...     # 12 是调用了 setref 之后的初始内存大小
...     start = profiler.heap().size + 12
...     for i in range(REPEAT):
...         function(*args, **kw)
...     return profiler.heap().size - start
...
>>> def stable():
...     return "some"*10000
...
...
>>> d = []
>>> def greedy():
...     for i in range(100):
...         d.append('garbage data'*i)
...
>>> memory_grow(stable)
24
>>> memory_grow(greedy)
5196468

```

(3) C 代码内存泄漏

如果 Python 代码看上去很好，但是循环执行被隔离的函数时内存占用仍然在增加，那么导致内存泄漏的问题可能位于用 C 写的那部分代码。当诸如 Py_DECREF 之类的调用丢失时，就会发生这样的情况。

Python 核心代码相当健壮，并且进行了内存泄漏的测试。如果使用具有 C 扩展的包，那么应该首先关注这些扩展。

12.3.3 剖析网络使用情况

正如之前说过的，与诸如数据库、LDAP 服务器之类的第三方程序通信的应用程序，可能会受到这些第三方程序执行速度的影响。这可以在应用程序端用常规的代码剖析方法进行跟踪。但是如果第三方软件工作得很好，那么问题可能出在网络上。

问题有可能是错误配置的 hub，网络连接带宽较低，甚至是导致使电脑多次发送相同数据包的通信冲突。

这里有几个要素是切入点。为了找到问题所在，可以调查以下 3 个方面。

- 使用以下工具查看网络流量。
 - ntop <http://www.ntop.org>（仅针对 Linux）
 - wireshark www.wireshark.org（以前叫 Ethereal）
- 使用 net-snmp 跟踪不健康或错误配置的设备。
- 使用 Pathrate 统计工具来估计两台计算机之间的带宽。

如果想要进一步查看网络性能问题，还可阅读 Richard Blum 所著的 *Network Performance Open Source Toolkit* 一书。这本书展示了针对大量使用网络的应用程序的调整策略，并且它是扫描复杂网络问题的一本很好的教程。

在编写使用 MySQL 的应用程序时，Jeremy Zawodny 所著的 *High Performance MySQL* 也是值得一读的好书。

12.4 小 结

本章介绍了以下内容。

- 优化的 3 条规则：首先使它能够正常工作；从用户的观点出来；保持代码易读。
- 基于编写带速度要求的使用场的优化策略。
- 剖析代码内存的方法，以及网络剖析的一些技巧。

知道如何查找问题后，下一章将提供解决问题的具体方案。

第 13 章

优化：解决方案

优化一个程序并不是一个神秘的过程，它是按照一些简单的过程来完成的。Stefan Schwarzer 在 Europython 2006 上用了一个原创的伪代码示例对其做了概括，如下所示。

```
def optimize():
    """Recommended optimization"""
    assert got_architecture_right(), "fix architecture"
    assert made_code_work(bugs=None), "fix bugs"
    while code_is_too_slow():
        wbn = find_worst_bottleneck(just_guess=False,
                                     profile=True)
        is_faster = try_to_optimize(wbn,
                                    run_unit_tests=True,
                                    new_bugs=None)
        if not is_faster:
            undo_last_code_change()
# By Stefan Schwarzer, Europython 2006
```

本章将介绍通过以下方法优化程序的一些解决方案：

- 降低复杂度；
- 多线程；
- 多进程；
- 缓存。



13.1 降低复杂度

对于程序复杂化有很多种定义，也有许多表现它的方法。但是在代码级别，要使独立的语句序列执行得更快，只有有限的几种技术能够快速检查导致瓶颈的代码行。

两种主要的技术是：

- 测量代码的回路复杂度（cyclomatic complexity）；
- 测量 Landau 符号，也称为大 O 记号（Big-O notation）。

现在，优化过程将包括降低复杂度，使代码运行足够快地工作。本节将介绍一些简化循环的简单技巧，但是首先还是先学习如何测量复杂度。

13.1.1 测量回路复杂度

回路复杂度是 McCabe 引入的一种度量，用于测量代码中线性路径的数量。所有 if、for 和 while 循环被计数为一个度量值。

然后，代码可以分类为如表 13.1 所示。

表 13.1

代码分类

回路复杂度	含 义
1~10	不复杂
11~20	适度复杂
21~50	真正复杂
>50	过于复杂

在 Python 中，这可以通过解析 AST（抽象语法树）自动完成，具体信息可以参见 http://en.wikipedia.org/wiki/Abstract_Syntax_Tree。来自 Reg Charney 的 PyMetrics 项目 (<http://sourceforge.net/projects/pymetrics>) 为计算回路复杂度提供了一个很好的脚本。

13.1.2 测量大 O 记号¹

函数复杂度可以用大 O 记号来表示（参见http://en.wikipedia.org/wiki/Big_O_notation）。这个度量定义了输入数据的大小对算法的影响。例如，该算法与输入数据的大小成线性关系还是平方关系？

¹ 译者注：其实就是大家熟悉的函数复杂度的度量方式，由于通常记为 O (*)，因此形象地称其为大 O 记号。

人工计算算法的大 O 记号是优化代码的最佳方法，因为这能够检测和关注真正使代码速度降低的部分。

为了测量大 O 记号，所有常量和低阶条款（low-order terms）将被删除，以便聚焦于输入数据增长时真正受影响的部分。思路是尝试将算法分类（如表 13.2 所示），尽管它们只是近似的。

表 13.2

算法分类

符 号	类 型
O(1)	常数，不依赖于输入数据
O(n)	线性，和“n”一起增长
O(n lg n)	准线性
O(n ²)	2 次方复杂度
O(n ³)	3 次方复杂度
.....
O(n!)	阶乘复杂度

例如，dict 查找是 O(1)（发音为“order 1”），被认为是常数，不管 dict 中有多少个元素，而在一个项目列表中查找特定项目则是 O(n)。

再举一个例子，如下所示。

```
>>> def function(n):
...     for i in range(n):
...         print i
...
```

在这个例子中，循环速度将依赖于“n”，大 O 记号将为 O(n)。

如果函数中存在条件分支，那么大 O 记号取决于最高的那个分支，如下所示。

```
>>> def function(n):
...     if some_test:
...         print 'something'
...     else:
...         for i in range(n):
...             print i
```

在这个例子中，函数的大 O 记号可能为 O(1) 或 O(n)，取决于测试用例。所以，最坏情况为 O(n)，大 O 符号也就应该选择这个值。

再举一个例子，如下所示。

```
>>> def function(n):
```

```

...     for i in range(n):
...         for j in range(n):
...             print i, j
...

```

一个嵌套的循环达到了 2 次复杂度，即 $O(n^2)$ ，当 n 很大时代价很高。

当然，大 O 记号最终取值还需要分析它调用的函数，如下所示。

```

>>> def function(n):
...     for i in range(n):
...         print i
...
>>> def other_function(n):
...     if some_test:
...         for i in range(n):
...             function(n)
...     else:
...         function(n)
...

```

`other_function` 调用了 `function`（复杂度为 $O(n)$ ），或者在一个有 $O(n)$ 复杂度的循环中调用它，所以最坏的情况是 $O(n*n) = O(n^2)$ 。

前面已经说过，常数和 low-order terms 应该在计算大 O 记号时被删除，因为它们不受数据大小增加的影响。

```

>>> def function(n):
...     for i in range(n*2):
...         print i
...

```

这个函数是 $O(n*2)$ ，但是因为常数被删除，所以只说 $O(n)$ 。在比较多个算法时必须牢记这一点。

还要注意，尽管一般假定复杂度为 $O(n^2)$ （2 次方）的函数要比复杂度为 $O(n^3)$ 的函数更快，但并不总是这样。有时候，对于较小的 n 值，3 次方复杂度的函数可能更快；而对于更大的 n 值而言，二次方函数的速度就能追上并且超过。例如，被简化为 $O(n^2)$ 的 $O(100*n^2)$ 不一定比简化为 $O(n^3)$ 的 $O(5*n^3)$ 快。这就是应该在剖析已经显示出问题所在时进行优化的原因。

如果想要练习大 O 记号的计算，可以在 <http://pages.cs.wisc.edu/~hasti/cs367-common/notes/COMPLEXITY.html#bigO> 上进行。

大 O 记号是改进算法的一个好办法，但是要知道：

- 它是近似值；
- 它只对纯 Python 代码是精确的，也就是不依赖外部资源时。

当不能计算一个算法的复杂度时，例如它包含难以研究的 C 代码时，可以改用诸如 `timeit` 这样的工具或前一章中介绍的剖析装饰器，然后使用足够的输入数据来测试算法的效率。

13.1.3 简化

为了降低算法的复杂度，数据的存储方式是基础。应该小心地选用数据结构。本小节将提供几个实例。

1. 在一个列表中查找

如果需要为一个列表实例提供一个搜索算法，对该列表的已排序版本执行二分查找就可以将复杂度从 $O(n)$ 降到 $O(\log n)$ 。`bisect` 模块可用来实现这一任务，对于指定的一个值，它将使用二分查找返回在一个已排序序列中的下一个插入点的索引值。

```
>>> def find(seq, el):
...     pos = bisect(seq, el)
...     if pos==0 or (pos==len(seq) and seq[-1]!=el):
...         return -1
...     return pos - 1
...
>>> seq = [2, 3, 7, 8, 9]
>>> find(seq, 9)
4
>>> find(seq, 10)
-1
>>> find(seq, 0)
-1
>>> find(seq, 7)
2
```

当然，这也意味着该列表已经排序过或者必须进行排序。换句话说，如果已经有了一个已排序的列表，那么也可以使用 `bisect` 插入一个新的项目，而不需要重新排序该列表（即插

入排序，参见http://en.wikipedia.org/wiki/Insertion_sort。

2. 使用一个集合代替列表

对一个指定的序列，如果需要获得一个只包含该序列中不重复值的序列时，马上会想到的第一个算法如下所示。

```
>>> seq = ['a', 'a', 'b', 'c', 'c', 'd']
>>> res = []
>>> for el in seq:
...     if el not in res:
...         res.append(el)
...
>>> res
['a', 'b', 'c', 'd']
```

在 `res` 列表中使用 `in` 操作符的查找，其最高的复杂度将是 $O(n)$ 。然后在全局循环中调用它，它的复杂度也是 $O(n)$ 。所以，总的复杂度通常会是 2 次方。

对于该操作，使用 `set` 类型将会更快，因为它将使用诸如 `dict` 之类的 `hash` 来查找存储的值。换句话说，对于 `seq` 中的每个值，查询它是否在 `set` 中所需的时间将是一个常数，如下所示。

```
>>> seq = ['a', 'a', 'b', 'c', 'c', 'd']
>>> res = set(seq)
>>> res
set(['a', 'c', 'b', 'd'])
```

这将使程序的复杂度降低为 $O(n)$ 。

当然，这里假定算法的余下部分可以使用一个消除了重复数值的 `set` 对象。

 当尝试降低一个算法的复杂度时，应该认真考虑数据结构。内建的类型有很多，要选择正确的一种。

在调用算法之前，转换数据往往比基于原始数据修改算法使其变得更快要好。

3. 删减外部调用，降低工作负载

程序复杂度的另一部分是调用其他函数、方法和类所引入的。一般来说，应该尽量将代码放在循环之外，对于嵌套的循环这点更加重要。不要在一个循环中反复计算可以在循环开始时之前计算的数值，内循环应该保持简洁。

4. 使用集合

集合模块提供了一种内建容器类型之外的替代品。它们有3种类型：

- `deque` 带有附加功能的一种类似于列表的类型；
- `defaultdict` 具有内建默认工厂特性的类似于`dict`的类型；
- `namedtuple` 为成员分配键值的一种类似于元组的类型。

(1) `deque`

`deque`是列表的一个替代实现。列表是基于数组的，而`deque`则是基于双链表的。所以，当需要在中间或表头中插入新元素时，`deque`要快得多；但是当需要访问随意的索引值时，则会慢得多。当然，现在的硬件能够很快地执行存储副本操作，所以列表的不利之处不像想象中那么严重。因此，确定将列表转换为`deque`时要进行剖析。

例如，如果希望从一个序列中删除指定位置的两个元素，而且不创建第二个列表实例，那么使用`deque`对象将会更快，如下所示。

```
>>> from pbp.scripts.profiler import profile, stats
>>> from collections import deque
>>> my_list = range(100000)
>>> my_deque = deque(range(100000))
>>> @profile('by_list')
... def by_list():
...     my_list[500:502] = []
...
>>> @profile('by_deque')
... def by_deque():
...     my_deque.rotate(500)
...     my_deque.pop()
...     my_deque.pop()
...     my_deque.rotate(-500)
...
...
>>> by_list();by_deque()
>>> print stats['by_list']
{'stones': 47.836141152815379, 'memory': 396,
'time': 1.0523951053619385}
>>> print stats['by_deque']
{'stones': 19.198688593777742, 'memory': 552,
'time': 0.42237114906311035}
```

deque 还提供了更有效的 append 和 pop 方法，在序列的两端都能以相同的速度执行。这使其成为了完美的队列类型。

例如，使用 deque 实现 FIFO（先进先出）队列会更高效，如下所示。

```
>>> from collections import deque
>>> from pbp.scripts.profiler import profile, stats
>>> import sys
>>> queue = deque()
>>> def d_add_data(data):
...     queue.appendleft(data)
...
>>> def d_process_data():
...     queue.pop()
...
>>> BIG_N = 1000000
>>> @profile('deque')
... def sequence():
...     for i in range(BIG_N):
...         d_add_data(i)
...     for i in range(BIG_N/2):
...         d_process_data()
...     for i in range(BIG_N):
...         d_add_data(i)
...
>>> lqueue = []
>>> def l_add_data(data):
...     lqueue.append(data)
...
>>> def l_process_data():
...     lqueue.pop(-1)
...
>>> @profile('list')
... def lsequence():
...     for i in range(BIG_N):
...         l_add_data(i)
```

```

...     for i in range(BIG_N/2):
...         l_process_data()
...     for i in range(BIG_N):
...         l_add_data(i)
...
>>> sequence(); lsequence()
>>> print stats['deque']
{'stones': 86.521963988031672, 'memory': 17998920, 'time':
1.9380919933319092}
>>> print stats['list']
{'stones': 222.34191851956504, 'memory': 17994312, 'time':
4.9804589748382568}

```

 Python 2.6 在 Queue 模块（在 Python 3k 中改名为 queue）中提供了新的有用的队列类，如 LIFO 队列（LifoQueue）和优先队列（PriorityQueue）。

(2) defaultdict

`defaultdict` 类型与 `dict` 类型近似，但是为新的键值添加了一个默认的工厂。这避免了编写一个额外的测试来初始化映射条目，并且比 `dict.setdefault` 方法更有效。

Python 文档为这一功能提供一个使用实例，几乎比 `dict.default` 快 3 倍，如下所示。

```

>>> from collections import defaultdict
>>> from pbp.scripts.profiler import profile, stats
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3),
...       ('blue', 4), ('red', 1)]
>>> @profile('defaultdict')
... def faster():
...     d = defaultdict(list)
...     for k, v in s:
...         d[k].append(v)
...
...
>>> @profile('dict')
... def slower():
...     d = {}

```

```

...     for k, v in s:
...         d.setdefault(k, []).append(v)
...
>>> slower(); faster()
>>> stats['dict']
{'stones': 16.587882671716077, 'memory': 396,
'time': 0.35166311264038086}
>>> stats['defaultdict']
{'stones': 6.5733464259021686, 'memory': 552,
'time': 0.13935494422912598}

```

`defaultdict` 类型取一个工厂作为参数，从而可以用于内建类型，或者构造函数中没有参数的类，如下所示。

```

>>> lg = defaultdict(long)
>>> lg['one']
0L

```

(3) namedtuple

`namedtuple` 是一个类工厂，传入一个类型名称和一个特性列表作为参数，它将创建出一个类。接着，该类可以用来实例化一个类似于元组的对象，并且为其元素提供访问程序，如下所示。

```

>>> from collections import namedtuple
>>> Customer = namedtuple('Customer',
...                         'firstname lastname')
>>> c = Customer(u'Tarek', u'Ziadé')
>>> c.firstname
u'Tarek'

```

它可以用来创建比需要样板代码来初始化的定制类更容易编写的记录。生成的类可以被子类化，以添加更多的操作。

降低复杂度的工作可以通过用算法能很好地处理的高效数据结构来存储数据来实现。

也就是说，当解决方案不明显时，应该考虑放弃并且重写出现问题的部分，而不是为了性能而破坏代码的可读性。

往往，可以使 Python 代码更易读且执行速度更快。所以，要尝试找到一个好的方法来执行工作，而不是尝试避开有漏洞的设计。



PDG

13.2 多线程

线程对于开发人员而言，通常被认为是一个复杂的主题。虽然这句话完全正确，但是 Python 提供了高级的类和函数，简化了本节将要介绍的针对特定使用场景的线程使用。

概括而言，当有些任务需要在后台执行，与此同时主程序还将执行别的工作时，应该考虑线程。

13.2.1 什么是多线程

线程是执行线程的简称。编程人员可以将其工作分割到同时运行并共享相同存储上下文的线程中。除非代码依赖第三方资源，否则在单处理器的机器上使用多线程不会加速代码的执行，甚至会增加一些线程管理的开销。多线程将从多处理器或多核机器中获益，并且将在每个 CPU 上并行执行每个进程，从而加速程序。

线程将共享相同的上下文，这意味着必须在并行访问中保护数据。如果两个线程更新相同的数据而没有进行保护，就会发生竞争条件。这被称为竞争危害（race hazard），因为每个线程运行的代码对数据状态做出错误的假设，会发生不可预见的结果。

锁机制有助于保护数据，线程编程始终需要确保线程以安全的方式访问资源。这可能相当困难，线程编程往往会导致难以调试的缺陷，因为它们很难再现。最糟糕的问题是由于代码设计的错误，两个线程锁住一个资源并且尝试获取另一个线程已经锁住的资源，这时它们将永远互相等待。这被称为死锁（Deadlock）现象，难以进行调试。可再入锁（Reentrant lock）对这种情况略有帮助，它通过尝试两次锁定一个资源的策略来确保线程不会被锁住。

不过，当针对独立的需求应用线程及其配套的工具时，它们可能会提高程序的速度。

多线程往往在核心级上实现。当机器使用的是单核的处理器时，系统将使用时间片机制。这时，CPU 会很快地从一个线程切换到另一个线程，产生一个并行的假象。这也能在进程级别上实现。在多处理器或多核机器上，即使使用时间片，进程和线程都将分布到不同的 CPU 上，使程序真正地变快。

13.2.2 Python 处理线程的方式

和其他语言不同，Python 使用多核级别（multiple kernel-level）线程，可以分别运行任何的解释程序级进程。但是，所有访问 Python 对象的进程都将被一个全局锁序列化。这是因为

大部分解释程序代码和第三方 C 代码都不是线程安全（thread-safe）的，需要保护。

这种机制被称为全局解释程序锁（GIL），有些开发人员开始要求从 Python 3k 中将它删除掉。但是，正如 Guido 所说，这将涉及太多的工作并且会使 Python 实现更复杂，所以 GIL 还是被保留了下来。

 Stackless（无栈）Python 或 Stackless 是 Python 编程语言的一种实验性实现，这么命名是因为它避免它的栈依赖于 C 调用栈。该语言支持生成器、微线程和协程，并提供基于线程的编程优势，而不会产生与常规线程相关的性能和复杂性问题（参见 <http://www.stackless.com>）。

有些开发人员对这种限制感到沮丧，许多开发人员难以理解这一基础设施，从而无法正确地进行多线程编程。许多 Python 编程人员往往使用多进程来代替多线程，因为进程有独立的存储上下文，它们不像线程那么容易受影响而破坏数据。

那么，Python 中多线程的要点是什么？

如果线程中只包含纯 Python 代码，使用线程不能加速程序，因为 GIL 将对线程进行序列化。但是，多线程可以并行地进行 IO 操作，或者在某些第三方扩展中执行 C 代码。

对于使用外部资源或涉及 C 代码的非纯 Python 代码块，多线程对于等待第三方资源返回结果而言是有用的。这是因为，一个显式地解锁 GIL 的休眠线程可以处于等待状态，并在结果返回时被唤醒。最后，在任何需要提供一个反应迅速的界面的时候，多线程都是一个可选方案，即使使用时间片。程序可以在与用户交互的同时在后台执行一些繁重的计算工作。

 更多的细节参见 Dr Dobb 站点上 Shannon Behrens 写的关于并行性的论文 (<http://ddj.com/linux-open-source/206103078>)。

下一小节将尝试覆盖各种常见的使用场景。

13.2.3 什么时候应该使用线程

尽管有 GIL 的局限，但线程在某些情况下可能确实有用。它对以下情况有帮助：

- 建立反应灵敏的界面；
- 委托工作；
- 建立多用户应用程序。

1. 建立反应灵敏的界面

假设要求系统通过一个图形用户界面从一个文件夹向另一个文件夹复制文件，那么这个任务可能被放到后台。窗口将不断地由主线程刷新，这样可以得到操作的实时反馈。这还将能撤销操作而不像原始的 `cp` 或 `copy` 命令那样，在整个工作完成之前不提供任何反馈，必须通过 `Ctrl+C` 组合键停止。

反应灵敏的界面还允许用户同时执行多个任务。例如，Gimp 允许在处理一个图形的同时过滤另一个图形，因为两个任务是独立的。



当创建一个用户界面时，应尝试将长时间运行的任务放到后台，或者至少尝试为用户提供不断的反馈。

2. 委托工作

如果进程依赖于第三方资源，线程可能确实能加速所有的部件。

下面再举一个对文件夹中的文件进行索引，并将索引放入一个数据库的函数示例。根据文件的类型，该函数将调用不同的外部程序，例如，一个专用于 PDF，另一个用于 OpenOffice 文件。

函数不是按顺序依次处理每个文件，而是通过调用正确的程序然后将结果存储在数据库，它可以为每个转换程序建立一个线程，并通过一个队列将任务推送给这些进程。该函数所耗费的总时间将接近于最慢的转换程序的时间，而不是所有工作的总和。

这种消费者—生产者模式常被用于为线程提供一个共享空间，这将在稍后予以介绍。

转换程序线程可以从一开始就初始化，负责将结果放入数据库的代码页，还有一个线程消费队列中可用的结果。

3. 多用户应用程序

线程也常作为多用户应用程序的一种设计模式使用。例如，一个 Web 服务器将把用户请求放入一个新的线程，然后自己回到空闲状态，等待新的请求。对每个请求使用专有的线程简化了许多工作，但是要求开发人员注意资源加锁的问题。当所有共享的数据都被放入一个关系数据库中时，这不是一个问题，因为它能够处理并行性问题。所以多用户应用程序中的线程表现得几乎像一个进程，在同一个进程中只是为了简化应用程序级的管理。

例如，Web 服务器将能够把所有请求放入一个队列，并且等待一个可用的线程来发送工作。而且，内存共享可以支持一些工作并降低内存负载。

使用进程要付出更多资源，因为每个进程都要装入新的解释程序。在进程中共享数据也需要花费更多的工作。



任何多用户应用程序都可以考虑使用线程。

Twisted 框架带有一个基于回调的编程思想，提供了针对服务器编程的开箱即用模式。

另外，eventlet（参见 <http://wiki.secondlife.com/wiki/Eventlet>）也是一个有趣的方法，它可能比 Twisted 更简单。

4. 简单的例子

让我们举一个应用程序的小例子，该程序递归扫描一个目录以处理文本文件。每个文本文件由一个外部转换程序打开并处理。

使用线程可能使其更快，因为多个文件之上的索引工作可以同时完成。

外部转换程序是一个执行某些复杂工作的 Python 小程序，如下所示。

```
#!/usr/bin/python
for i in range(100000):
    i = str(i) + "y"*10000
```

这个脚本将被保存到 converter.py，花费大约 25 kystones，这在使用 Intel Core Duo 2 CPU 的 MacBook 电脑上大约需要半秒钟。

在多线程解决方案中，主程序将处理一个线程池，每个线程都从队列中获取其工作。在这种使用场景下，线程被称为工作者（workers）。队列是共享的资源，主程序将遍历目录，将所找到的文件添加进去。工作者将从队列取出文件并进行处理。

标准程序库中的 Queue 模块（在 Python 3K 中将改名为 queue）对我们的程序来说是完美的类。它提供一个多消费者、多生产者的 FIFO 队列，内部使用 deque 实例并且是线程安全的。

所以，如果想要处理队列中的文件，只要使用 get 和 task_done 方法，这使 Queue 实例知道任务已经完成，以便开始执行 join 方法，如下所示。

```
>>> from Queue import Queue
>>> import logging
>>> import time
>>> import subprocess
>>> q = Queue()
>>> def index_file(filename):
...     logging.info('indexing %s' % filename)
...     f = open(filename)
...     try:
```

```

...
    content = f.read()
...
    # 示例中没有使用 content
...
    # 在此为外部进程
...
    subprocess.call(['converter.py'])
...
    time.sleep(0.5)
...
    finally:
...
        f.close()
...
...
>>> def worker():
...
    while True:
...
        index_file(q.get())
...
        q.task_done()
...

```

`worker` 函数将通过一个线程调用，它将从队列获取文件名，并且调用 `index_file` 函数来处理它们。调用 `sleep` 来模拟外部程序已完成了处理，并使进程等待结果，从而解锁 GIL。

然后，主程序可以启动工作者，扫描被处理的文件，并将它们供给队列。

这将通过使用 `worker` 方法创建 `Thread` 实例来实现。`setDaemon` 是必要的，这样线程才能在程序退出时自动关闭。否则，程序将永远挂起以等待它们退出。这可以手工管理，但是在*这里*没有用。

在 `index_files` 函数的最后，`join` 方法将等待队列全部处理完。

创建一个完整的脚本 `indexer.py`，它将运行一个多线程的版本，以及一个用来对包含文本文件的目录结构进行索引的独立线程，如下所示。

```

from threading import Thread
import os
import subprocess
from Queue import Queue
import logging
import time
import sys
from pbp.scripts.profiler import profile, print_stats
dirname = os.path.realpath(os.path.dirname(__file__))
CONVERTER = os.path.join(dirname, 'converter.py')
q = Queue()
def index_file(filename):
    f = open(filename)
    try:
        content = f.read()

```

```

# process is here
subprocess.call([CONVERTER])
finally:
    f.close()

def worker():
    while True:
        index_file(q.get())
        q.task_done()

def index_files(files, num_workers):
    for i in range(num_workers):
        t = Thread(target=worker)
        t.setDaemon(True)
        t.start()

    for file in files:
        q.put(file)

    q.join()

def get_text_files(dirname):
    for root, dirs, files in os.walk(dirname):
        for file in files:
            if os.path.splitext(file)[-1] != '.txt':
                continue
            yield os.path.join(root, file)

@profile('process')
def process(dirname, numthreads):
    dirname = os.path.realpath(dirname)
    if numthreads > 1:
        index_files(get_text_files(dirname), numthreads)
    else:
        for f in get_text_files(dirname):
            index_file(f)

if __name__ == '__main__':
    process(sys.argv[1], int(sys.argv[2]))
    print_stats()

```

这个脚本可以用于任何目录，只要它包含文本文件。它有两个参数：

- 目录名称；
- 线程数量。

如果只提供目录名称，那么执行时将不会启动线程，目录的处理将在主线程中进行。

在相同的 MacBook 电脑上运行，并选择一个包含 36 个文件，其中 19 个是文本文件的目录。该目录由 6 个子目录组成，如下所示。

```
$ python2 indexer.py zc.buildout-1.0.6-py2.5.egg 1
process : 301.83 kstones, 6.821 secondes, 396 bytes
$ python indexer.py zc.buildout-1.0.6-py2.5.egg 2
process : 155.28 kstones, 3.509 secondes, 2496 bytes
$ python indexer.py zc.buildout-1.0.6-py2.5.egg 4
process : 150.42 kstones, 3.369 secondes, 4584 bytes
$ python indexer.py zc.buildout-1.0.6-py2.5.egg 8
process : 153.96 kstones, 3.418 secondes, 8760 bytes
$ python indexer.py zc.buildout-1.0.6-py2.5.egg 12
process : 154.18 kstones, 3.454 secondes, 12948 bytes
$ python indexer.py zc.buildout-1.0.6-py2.5.egg 24
process : 161.84 kstones, 3.593 secondes, 25524 bytes
```

两个线程看上去速度是单线程的两倍，添加更多的线程也一样。24 个线程只比 12 个线程慢一点，这是因为线程的开销。

这些结果可能会因文件的数量变化而变化，因为磁盘访问也会增加一些开销。但是可以有把握地说，在双核的机器上，使用多线程会使代码速度成倍增加。

当想创建反应灵敏的界面，或者将一些工作委托给第三方应用程序时，应该使用多线程。

因为内存是共享的，数据破坏的危险和竞争条件总是存在的。当使用 queue 模块作为唯一的线程通信和传递数据方法时，这种危险将大大减轻。

合理的策略是不让两个线程接触相同的易失性数据。

13.3 多 进 程

GIL 限制使得加速大量使用纯 Python 的程序成为不可能的事情，它束缚了 CPU。唯一的方法是使用独立的进程，这一般通过在某个地方调用 fork 来实现。fork 是通过 os.fork 的一个系统调用，它用来创建一个新的子进程。两个进程在 fork 之后将接着继续独立执行程序，如下所示。

```
>>> import os
>>> a = []
```

```

>>> def some_work():
...     a.append(2)
...     child_pid = os.fork()
...     if child_pid == 0:
...         a.append(3)
...         print "hey, I am the child process"
...         print "my pid is %d" % os.getpid()
...         print str(a)
...     else:
...         a.append(4)
...         print "hey, I am the parent"
...         print "the child is pid %d" % child_pid
...         print "I am the pid %d" % os.getpid()
...         print str(a)
...
>>> some_work()
hey, I am the parent
the child is pid 25513
I am the pid 25411
[2, 4]
hey, I am the child process
my pid is 25513
[2, 3]

```



在提示符上运行这个实例将会使会话变得很混乱。

`fork` 时将会复制内存上下文，然后每个进程将处理自己的地址空间。为了实现进程间通信，进程需要处理整个系统的资源，或者使用诸如信号量这样的低层级工具。

不幸的是，`os.fork` 在 Windows 下是不可用的，这种情况需要使用新的解释程序，以模拟 `fork` 的功能。所以，代码可能会根据平台的不同而发生变化。

当创建进程时，它们可能需要通信。例如，如果进程被用于进行一些使用关系数据库的独立工作，共享空间一般是最好的选择。

处理信号量是很痛苦的，共享内存、管道或套接字比较好对付。这通常是在进程不是一次性工作者，而是交互时的情况。

`pyprocessing` 程序库能使进程变得十分容易处理。

pyprocessing

pyprocessing 提供了一个可移植的，使进程能够像线程一样处理的方法。如果要安装它，可以用 easy_install 寻找 processing，命令如下。

```
$ easy_install processing
```

这个工具提供了和 Thread 类很像的 Process 类，而且可用于任何平台，如下所示。

```
>>> from processing import Process
>>> import os
>>> def work():
...     print 'hey i am a process, id: %d' % os.getpid()
...
>>> ps = []
>>> for i in range(4):
...     p = Process(target=work)
...     ps.append(p)
...     p.start()
...
hey i am a process, id: 27457
hey i am a process, id: 27458
hey i am a process, id: 27460
hey i am a process, id: 27459
>>> ps
[<Process(Process-1, stopped)>, <Process(Process-2, stopped)>,
<Process(Process-3, stopped)>, <Process(Process-4, stopped)>]
>>> for p in ps:
...     p.join()
...
```

当创建了进程时，将会对内存进行 fork。最有效的进程使用方式是使它们在创建之后各自工作，以避免开销，并从主线程检查它们的状态。除了内存状态被复制外，Process 类还在其构造程序中提供了额外的 args 参数，这可以用来传递所需的数据。

pyprocessing 还提供了一个类似于队列的类，可以用来在一个由该包管理的共享内存空间中让多个进程共享数据。



processing.sharedctypes 还提供了 ctypes 和进程间的对象共享功能（参见 <http://pyprocessing.berlios.de/doc/sharedctypes.html>）。

因此，前面一个工作者实例可以使用进程来代替线程，将 Queue 实例替换为 processing.Queue 即可。

pyprocessing 的另一个巧妙的特性是 Pool 类，它能够自动化地生成和管理一个工作者集合。如果没有提供，工作者的数量将和电脑上的 CPU 数量相同，这由 cpuCount API 给出，如下所示。

```
>>> import processing
>>> import Queue
>>> print 'this machine has %d CPUs' \
...      % processing.cpuCount()
this machine has 2 CPUs
>>> def worker():
...     file = q.get_nowait()
...     return 'worked on ' + file
...
>>> q = processing.Queue()
>>> pool = processing.Pool()
>>> for i in ('f1', 'f2', 'f3', 'f4', 'f5'):
...     q.put(i)
...
>>> while True:
...     try:
...         result = pool.apply_async(worker)
...         print result.get(timeout=1)
...     except Queue.Empty:
...         break
...
worked on f1
worked on f2
worked on f3
worked on f4
worked on f5
```

apply_async 方法将通过进程池来调用 worker 函数，并立即返回结果对象，主进程可以将它作为结果返回。get 方法可以用于等待结果（具有超时设置）。

最后，Array 类和 Value 类提供了共享的内存空间。但是，应该通过设计避免使用它们，因为它们在并行化中将引入瓶颈，从而增加代码复杂性。

`pyprocessing` 网站中有许多值得阅读的代码实例，可以在程序中复用这个包中的代码。到本书编写时，在 `python-dev` 上已经有人要求将这个包纳入标准程序库的一部分，在 2.7 系列中应该会将其纳入。所以 `pyprocessing` 绝对是值得推荐的多进程工具。

13.4 缓存

对运行代价很高的函数和方法的结果，可以进行缓存处理，只要：

- 该函数是确定性的，输入相同值，生成的结果每次都相同；
 - 函数返回值在一定时期内（不确定）持续有用和有效。

确定性的函数是相同的参数将始终返回相同的结果，而不确定性的函数返回结果则可能有变化。

较好的缓存候选者通常是：

- 来自查询数据库的可调用对象的结果；
 - 来自呈现静态值（像文件内容、Web 请求或 PDF 显示）的可调用对象的结果；
 - 来自确定性的、执行复杂计算的可调用对象的结果；
 - 记录具有过期时间的数值的全局映射，如 Web 对话对象；
 - 一些需要经常和快速访问的数据。

13.4.1 确定性缓存

确定性缓存一个简单的例子是计算平方数的函数，记录其执行结果将能加速它，如下所示。

```

>>> @profile('not cached')
... def factory_calls():
...     for i in xrange(100):
...         square(random.randint(1, 10))
...
>>> def cached_factory(n):
...     if n not in cache:
...         cache[n] = square(n)
...     return cache[n]
...
>>> @profile('cached')
... def cached_factory_calls():
...     n = [random.randint(1, 10) for i in range(100)]
...     ns = [cached_factory(i) for i in n]
...
>>> factory_calls(); cached_factory_calls();
>>> print_stats()
not cached : 20.51 kstones, 0.340 secondes, 396 bytes
cached : 6.07 kstones, 0.142 secondes, 480 bytes

```

当然，只要与缓存交互的时间比函数花费的时间少，缓存就是有效的。如果重新计算该值更快一些，那么就一定要这么做。如果使用不正确，缓存也可能是危险的。例如，可能使用了陈旧的数据，或者导致很大的缓存吞噬了内存。

这就是只在值得的时候建立缓存的原因——正确建立它是有成本的。

在前一个例子中，使用一个函数的参数作为缓存的键值，这只有当参数可以 hash 时才有效。例如，这只在 int 和 str 上有效，而在 dict 上则不行。当参数变得复杂并且不一定可以 hash 时，必须手工处理它们并且将其转换为用于缓存的唯一键值，如下所示。

```

>>> def cache_me(a, b, c, d):
...     # 不用关心键值
...     key = 'cache_me::::%s::::%s::::%s' % (a, b, c)
...     if key not in cache:
...         print 'caching'
...         cache[key] = complex_calculation(a, b, c, d)
...     print d # d is just use for display
...     return cache[key]
...

```

当然，有可能通过遍历每个参数以自动创建键值。但是有许多特殊的情况，将需要手工

计算该键值，就像在上面的实例中一样。

这种行为被称为备注（memoizing），可以被转换为一个简单的装饰器，如下所示。

```
>>> cache = {}

>>> def get_key(function, *args, **kw):
...     key = '%s.%s:' % (function.__module__,
...                         function.__name__)
...     hash_args = [str(arg) for arg in args]
...     # 当然，只有当 v 可以 hash 时才能够正常工作
...     hash_kw = ['%s:%s' % (k, hash(v))
...               for k, v in kw.items()]
...     return '%s::%s::%s' % (key, hash_args, hash_kw)
...
...
>>> def memoize(get_key=get_key, cache=cache):
...     def _memoize(function):
...         def __memoize(*args, **kw):
...             key = get_key(function, *args, **kw)
...             try:
...                 return cache[key]
...             except KeyError:
...                 cache[key] = function(*args, **kw)
...                 return value
...             return __memoize
...     return _memoize
...
...
...
>>> @memoize()
... def factory(n):
...     return n * n
...
...
>>> factory(4)
16
>>> factory(4)
16
>>> factory(3)
9
>>> cache
{"__main__.factory::__['3']::__[]": 9,
 "__main__.factory::__['4']::__[]": 16}
```

这个装饰器使用一个可调用的对象来计算键值，默认的 `get_key` 将进行参数内省。如果关键字参数不能 hash，将会抛出一个异常。但是，这个函数只适用于特殊的情况。保存值的映射也是可以配置的。

常见的方法是计算参数的 MD5 hash（或 SHA）。但是要知道，这样一个 hash 是有实际代价的，函数本身必须比键值的计算更慢些，缓存才有意义。对于 `factory` 函数，它刚好是这种情况，如下所示。

```
>>> import md5
>>> def get_key(function_called, n):
...     return md5.md5(str(n)).hexdigest()
...
>>> @memoize(get_key)
... def cached_factory(n):
...     return n * n
...
>>> factory_calls(); cached_factory_calls();
>>> print_stats()
cached : 6.96 kstones, 0.143 secondes, 1068 bytes
not cached : 7.61 kstones, 0.157 secondes, 552 bytes
```

13.4.2 非确定性缓冲

对于非确定性函数而言，即使在指定相同的输入时，也可能产生不同的输出。例如，数据库查询有时只能缓存一段时间。举个例子，如果一个 SQL 表保存了关于用户的信息，为所有显示用户数据的函数做查询缓存是一个好的做法，只要这个表不是经常更新的话。另一个例子是服务器启动之后不会修改的服务器配置文件，将这些值放在缓存中是个好办法。许多服务器会收到一个信号，表明它们应该清除缓存并重新读取配置文件。最后，一个 Web 服务器可能在至少几个小时之内，可以使用缓存服务器（如 SQUID）来对完整的页面或用于所有页面标题的图标进行缓存。以图标为例，客户端浏览器也将维护一个本地缓存，但是将与 SQUID 协商，以了解图标是否已经被修改。



缓存持续时间按照数据的平均更新时间来设置。

]

`memoize` 缓存提供了一个额外的 `age` 参数，用来使太旧的缓存值作废，如下所示。

```
def memoize(get_key=get_key, storage=cache, age=0):
```

```

def _memoize(function):
    def __memoize(*args, **kw):
        key = get_key(function, *args, **kw)
        try:
            value_age, value = storage[key]
            expired = (age != 0 and
                        (value_age+age) < time.time())
        except KeyError:
            expired = True
        if not expired:
            return value
        storage[key] = time.time(), function(*args, **kw)
        return storage[key][1]
    return __memoize
return _memoize

```

假设有一个显示当前时间的函数。如果不显示秒数，那么可以以 30 秒的老化期来缓存它，以获得有合理精确度的缓存值，如下所示。

```

>>> from datetime import datetime
>>> @memoize(age=30)
... def what_time():
...     return datetime.now().strftime('%H:%M')
...
>>> what_time()
'19:36'
>>> cache
{'__main__.what_time::::[]::[]': (1212168961.613435, '19:36')}

```

当然，缓存作废必须由删除过期键值的另一个函数异步实现，以加速 memoize 函数。这对于需要偶尔中止旧的对话的 Web 应用程序而言是很常见的。

13.4.3 主动式缓冲

有许多缓存策略可以用来加速应用程序。例如，如果一个 intranet 在每天早上都会遇到很高的负载，因为用户在阅读前一个下午张贴的新闻，那么对呈现文章的结果进行缓存处理就是有意义的。这样在每次新的 Web 请求时，就不用重新呈现这些文章。一个好的缓存策略应该是在夜里通过 cron 作业模拟这些用户，以使用最大寿命为 12 小时的数据来填充缓存。

Memcached

如果对缓存要求很高，那么 Memcached（见 <http://www.danga.com/memcached>）将是你希望使用的工具。诸如 Facebook 或 Wikipedia 这样的大型应用程序，就使用了这个缓存服务器来提升它们网站的速度。除了简单的缓冲功能，它还具有群集能力，可以立即创建一个非常有效的分布式缓存。

这个工具是基于 Unix 的，但是可以由任何平台和任何语言来驱动。Python 客户端很简单，memoize 函数可以很容易地适应它。

Beaker 是一个使用 Memcached 的缓存中间件的 WSGI 实现（参见 <http://pypi.python.org/pypi/Beaker>）。

缓存可以节约时间，但是不应该用它来掩饰由于设计和实现不好而产生的函数迟钝。

改进代码往往更加安全，这样就不必担心陈旧的缓存、无限的内存增长或不好的设计。缓存只适用于那些无法进一步优化的代码。

缓存的大小应该始终受控于最大寿命和/或最大尺寸。

也可以通过 Memcached 来获得更加有效的缓存机制。

13.5 小 结

本章介绍了以下内容：

- 测量代码的复杂性，以及降低这种复杂性的一些方法；
- Python 中线程的工作方式，以及它们适用的场景；
- 使用进程的简单方法；
- 一些缓存理论及其使用方法。

下一章中将专门介绍设计模式。

第 14 章

有用的设计模式

设计模式是可复用的，某种程度上它是对软件设计中常见问题提供的语言相关的解决方案。关于这个主题，最流行的书籍是 Gamma、Helm、Johnson 和 Vlissides a.k.a“四人组”（GoF）编写的 *Elements of Reusable Object-Oriented Software*（中译版《设计模式：可复用面向对象软件的基础》）。它被认为是这个领域中的重要作品，提供了 23 种设计模式，以及以 SmallTalk 和 C++ 编写的示例。

在设计应用程序时，这些模式是很好的且众所周知的参考资源。它们对于所有开发人员来说都很熟悉，因为其描述了已验证的开发范式。但是应该结合当前使用的语言来学习，因为它们中的一部分在某些语言中是没有意义的，或者已经内建了。

本章将用实例描述 Python 中最有用或有趣的模式。以下 3 节对应于 GoF 定义的 3 种设计模式类别：

- **创建型模式** 用于生成具有特定行为的对象的模式；
- **结构型模式** 有助于针对特定使用场景的代码结构的模式；
- **行为型模式** 有助于对过程进行结构化的模式。

14.1 创建型模式

创建型模式提供了特殊的实例化机制。它可以是一个特殊的对象工厂甚至类工厂。

这在诸如 C 之类的编译型语言中是很重要的模式，因为在运行时按照要求生成类型更加困难。

但是在 Python 中，这个功能是内建的，如内建的 `type`，可以通过代码来定义新的类型，如下所示。

```
>>> MyType = type('MyType', (object,), {'a': 1})
>>> ob = MyType()
>>> type(ob)
<class '__main__.MyType'>
>>> ob.a
1
>>> isinstance(ob, object)
True
```

类和类型是内建的工厂，例如，可以使用元类和类及对象生成交互（参见第3章）。这些功能是实现工厂（Factory）设计模式的基础，本节不做进一步的描述。

除了工厂之外，Python中来自GoF的唯一有趣的创建型模式是单例模式（Singleton）。

单例模式



单例模式用来限制一个类只能实例化为一个对象。



单例模式确保给定的类在应用程序中始终只存在一个实例。例如，当希望限制一个资源在进程中访问一个并且只有一个内存上下文的时候，就可以使用它。举个例子，一个数据库连接器类就可以是一个单例，在内存中处理同步并管理其数据。它假设没有其他实例同时与数据库交互。

这个模式可以大大简化应用程序中并发的处理。为整个应用程序范围提供功能的应用程序往往被声明为单例。例如，在Web应用程序中，负责保存唯一文档ID的类将从单例模式中获益。应该有一个并且只有一个实用程序进行这种工作。

实现单例模式很简单，只需使用`__new__`方法，如下所示。

```
>>> class Singleton(object):
...     def __new__(cls, *args, **kw):
...         if not hasattr(cls, '_instance'):
...             orig = super(Singleton, cls)
...             cls._instance = orig.__new__(cls, *args, **kw)
...         return cls._instance
...
>>> class MyClass(Singleton):
...     a = 1
```

```

...
>>> one = MyClass()
>>> two = MyClass()
>>> two.a = 3
>>> one.a
3

```

不过，这种模式在子类化方面也存在一些问题——所有实例将是 `MyClass` 的实例，而不管方法解析顺序（`__mro__`）的结果，如下所示。

```

>>> class MyOtherClass(MyClass):
...     b = 2
...
>>> three = MyOtherClass()
>>> three.b
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: 'MyClass' object has no attribute 'b'

```

为了避免这个限制，Alex Martelli 提出了一个基于共享状态的替代实现，即 `Borg`。

这个思路相当简单。单例模式中真正重要的不是类存在的实例数量，而是它们在任何时候都将共享相同状态的事实。所以 Alex Martelli 引入了一个类，使该类的所有实例共享相同的 `__dict__`，如下所示。

```

>>> class Borg(object):
...     _state = {}
...     def __new__(cls, *args, **kw):
...         ob = super(Borg, cls).__new__(cls, *args, **kw)
...         ob.__dict__ = cls._state
...         return ob
...
>>> class MyClass(Borg):
...     a = 1
...
>>> one = MyClass()
>>> two = MyClass()
>>> two.a = 3
>>> one.a
3

```

```
>>> class MyOtherClass(MyClass):
...     b = 2
...
>>> three = MyOtherClass()
>>> three.b
2
>>> three.a
3
>>> three.a = 2
>>> one.a
2
```

这解决了子类的问题，但是仍然依赖于子类代码的工作方式。例如，如果 `_getattr_` 被重载，那么这个模式可能会被破坏。

虽然如此，单例不应该有多级别的继承，标记为单例的类已经是特殊类了。

也就是说，该模式被许多开发人员视为处理应用程序中唯一性问题的重要方法。如果需要一个单例，既然 Python 模块也是一个单例，那么为什么不使用带有函数的模块来代替呢？



单例工厂是处理应用程序中唯一性的隐含方法。也可以不使用它，除非在使用一个 Java 风格的框架，否则应该使用模块来代替类。

14.2 结构型模式

结构型模式在大的应用程序中确实很重要，它们决定了代码的组织方式并且提供了开发人员与应用程序各部分交互的方法。

在 Python 世界中，结构型模式最著名的实现是 Zope Component Architecture (Zope 组件架构，简写为 ZCA，参见 <http://wiki.zope.org/zope3/ComponentArchitectureOverview>)。它实现了本节中描述的大部分模式，并提供了处理这些模式的丰富工具集。ZCA 并不只是用于 ZOPE 框架，还可用于诸如 Twisted 之类的其他框架。它还提供了界面和适配器的一个实现。

所以即使是不大的作品，也应该考虑用它，而不是从头开始重写这样的模式。

有许多从 GoF 11 原作中继承而来的结构化模式。

例如，Python 提供了类似装饰器的模式，允许使用 `@decorator` 语法装饰一个函数，但是

不是在运行时。这在未来版本中将被扩展为类(参见<http://www.python.org/dev/peps/pep-3129>)。

其他流行的模式是：

- 适配器 (Adapter);
- 代理 (Proxy);
- 外观 (Facade)。

适配器

 适配器用来封装一个类或一个对象 A，这样它可以工作在用于一个类或对象 B 的上下文中。

当一些代码被用来处理一个指定的类时，从另一个类给予它对象是很好的，这些对象提供代码所使用的方法和特性即可。这形成了 Python 中的 *duck-typing* 思想的基础。

 如果它走起路来像鸭子，说起话来也像鸭子，那么它就是鸭子！

当然，这假定代码没有调用 `instanceof` 以验证该实例是一个特定类。

 我说了它是一只鸭子，没有必要检查它的 DNA！

适配器模式基于这种思想并定义了一个封装机制。在这个机制中，封装了一个类或对象，以便使其在不是为它提供的上下文中工作。`StringIO` 是个典型的例子，它改编了 `str` 类型使其可以像 `file` 类型一样使用（如图 14.1 所示）。

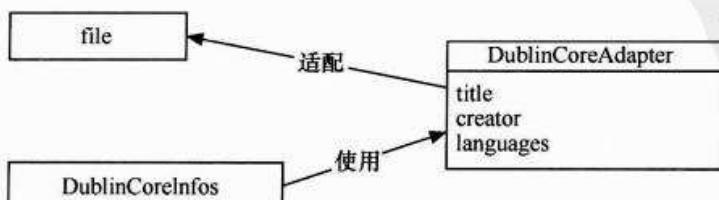


图 14.1

```
>>> from StringIO import StringIO  
>>> my_file = StringIO(u'some content')
```

```
>>> my_file.read()
u'some content'
>>> my_file.seek(0)
>>> my_file.read(1)
u's'
```

再举一个例子。DublinCoreInfos 类知道如何为指定的文档显示 Dublin Core 信息（参见 <http://dublincore.org/>），它读取几个字段（如作者名称或标题）并且打印。为了能够为一个文件显示 Dublin Core，它必须和 StringIO 一样进行改变。图 14.1 给出了该模式的类-UML 图。

DublinCoreAdapter 封装了一个文件实例，并且提供了针对它的元数据访问，如下所示。

```
>>> from os.path import split, splitext
>>> class DublinCoreAdapter(object):
...     def __init__(self, filename):
...         self._filename = filename
...     def title(self):
...         return splitext(split(self._filename)[-1])[0]
...     def creator(self):
...         return 'Unknown' # 真地能够获取它
...     def languages(self):
...         return ('en',)
...
>>> class DublinCoreInfo(object):
...     def summary(self, dc_ob):
...         print 'Title: %s' % dc_ob.title()
...         print 'Creator: %s' % dc_ob.creator()
...         print 'Languages: %s' %
...               ', '.join(dc_ob.languages())
...
>>> adapted = DublinCoreAdapter('example.txt')
>>> infos = DublinCoreInfo()
>>> infos.summary(adapted)
Title: example
Creator: Unknown
Languages: en
```

除了允许替换之外，适配器模式还能改变开发人员的工作方式。改编一个对象使其工作

于特定的上下文，设想对象是什么类无关紧要。重要的是这个类实现了 DublinCoreInfo 所期待的特性，这种行为由一个适配器来修复或完成。所以，代码只要以某种方式告知它与实现特定行为的对象兼容即可，而这可以由接口（Interfaces）来表示。

1. 接口

接口是 API 的一个定义。它描述了一个类为所需要的行为必须实现的方法和特性的列表。这个描述不实现任何代码，只定义希望实现该接口的类所用的明确契约（contract）。之后，任何类都可以以其希望的方式实现一个或多个对象。

虽然在 Python 中，相比显式接口定义更喜欢 duck typing，但是有时候使用接口可能更好。例如，显式接口定义使框架在接口上定义功能变得更容易了。

这样做的好处是，类是低耦合的，这被认为是好的做法。例如，为了执行一个给定的处理，类 A 不依赖于 B，而是依赖于 I，类 B 实现 I，但它可以是任何其他类。

例如，这种技术内建于 Java 中，代码可以处理实现指定接口的对象，而不管它是来自于哪种类。这是一个显式的 duck-typing 行为：Java 使用接口在编译时验证类型安全，而不是使用 duck-typing 在运行时绑定。

许多开发人员要求将接口添加为 Python 的核心功能。当前，这些希望使用显式接口的人现在都被迫使用 Zope 接口或 PyProtocols。

以前，Guido 拒绝将接口添加到 python 中，因为它们不适合 Python 的动态的 duck-typing 特性。但是，接口系统已经在某些情况下证明了它们的价值，所以 Python 3000 将引入一个被称为 optional type annotations（可选类型符号）的特性。这个特性可以作为针对第三方接口库的语法。

 最近 Python 3000 还添加了抽象基类（ABC）支持。以下是从 PEP 中摘录的。

“ABC 只是被添加到对象继承树中的 Python 类，它用来向外部检查程序传达该对象的某些功能。”

 适配器对于使类和执行上下文保持松耦合而言是完美的。但是，如果将适配器当作编程思想，而不是作为在特定的过程中对必须使用的对象做一个快速修复，那么应该考虑换用接口。

2. 代理



代理对一个代价昂贵或远程的资源提供了一个非直接访问的机制。

代理（Proxy）在客户（client）和主题（subject）之间，如图 14.2 所示。它用来优化对高代价主题的访问，例如，在前一章中描述的 memoize 装饰器就可以被认为是一个代理。

代理还可以用来提供到一个主题的智能访问。例如，大的视频文件可以封装在代理中，以避免在用户仅仅请求其标题时就将文件载入到内存中。

urllib2 给出了一个实例。urlopen 是一个针对位于远程 URL 上的内容的代理，当它被创建时，报头可以独立于内容读取，如下所示。

```
>>> class Url(object):
...     def __init__(self, location):
...         self._url = urlopen(location)
...     def headers(self):
...         return dict(self._url.headers.items())
...     def get(self):
...         return self._url.read()
...
>>> python_org = Url('http://python.org')
>>> python_org.headers()
{'content-length': '16399', 'accept-ranges': 'bytes', 'server':
'Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 mod_ssl/2.2.3 OpenSSL/0.9.8c',
'last-modified': 'Mon, 09 Jun 2008 15:36:07 GMT', 'connection':
'close', 'etag': '"6008a-400f-91f207c0"', 'date': 'Tue, 10 Jun 2008
22:17:19 GMT', 'content-type': 'text/html'}
```

这可以用于在获取页面主体以更新本地拷贝之前，通过查看报头 last-modified 来确定页面是否已经改变。以一个大文件为例，如下所示。

```
>>> ubuntu_iso=Url('http://ubuntu.mirrors.proxad.net/hardy/ubuntu-8.04-
desktop-i386.iso')
```

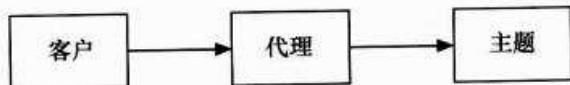


图 14.2



PDG

```
>>> ubuntu_iso.headers['last-modified']
'Wed, 23 Apr 2008 01:03:34 GMT'
```

代理的另一个使用场景是数据唯一性。

例如，一个在多个位置显示相同文档的网站。文档附加了专用于每个位置的额外字段，如访问计数和几个许可设置。代理可以用于这种情况，由它处理与位置相关的事务，并且指向原始的文档而不是复制它。所以，一个指定的文档可以拥有许多代理，如果它的内容变化，所有位置都将从中获益而不必处理版本同步。



使用代理作为可能存在于其他地方的某些东西的本地句柄，可以：

- 使进程更快；
- 避免外部资源访问；
- 降低内存负载；
- 确保数据唯一性。

3. 外观



外观提供对子系统的高级别的、更简单的访问。

外观只是一个使用应用程序的某个功能的快捷方式，不需要应对子系统的底层复杂性。例如，这可以通过在包级别提供更高级别的函数来完成。



例子请参见 4.7.1 小节。

外观模型通常是在现有系统基础上使用的，在这里，包的常见用法将综合到高级别的函数中。一般不需要类来提供这样一个模式，`__init__.py` 模块中的简单函数就足够了。



外观简化了包的用法，一般在几次有使用反馈的迭代之后加入。

14.3 行为型模式

行为型模式用于简化类之间的交互，简化的手段是结构化它们的交互进程。

本节将提供 3 个例子：

- 观察者 (Observer);
- 访问者 (Visitor);
- 模板 (Template)。

14.3.1 观察者



观察者模式用来通知一系列对象状态的变化。

]

使用观察者模式可以在一个应用程序中以可插入的方式来添加特性，并且解除现有代码库和新功能之间的耦合。事件框架是观察者模式的典型实现，在图 14.3 中有介绍。每当一个事件发生时，事件的所有观察者都会得到触发此事件的对象的通知。

事件是发生某些事情的时刻。在图形用户界面应用程序中，事件驱动编程（参见 http://en.wikipedia.org/wiki/Eventdriven_programming）常常被用来实现代码到用户操作的链接。例如，一个函数可以链接到 `MouseMove` 事件上，这样每当鼠标移到窗口上面时，这个函数就将被调用。在这种情况下，将这些代码从窗口管理事务中解耦，大大简化了工作：函数将单独编写，并且注册为事件观察者（如图 14.3 所示）。这种方法从微软公司的 MFC framework 的最早版本（参见 http://en.wikipedia.org/wiki/Microsoft_Foundation_Class_Library）起就存在于所有 GUI 开发工具（如 Delphi）中。

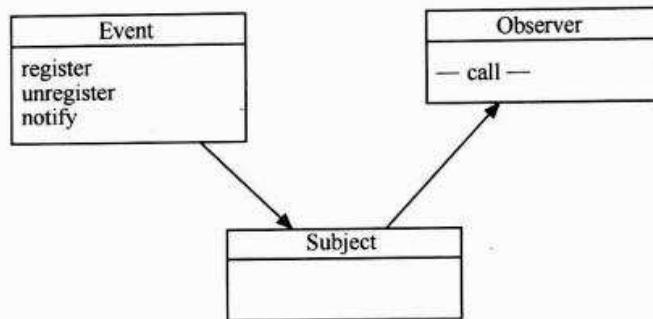


图 14.3

但是这些代码也可能生成事件。例如，在一个将文档存储到数据库的应用程序中，代码可能提供 DocumentCreated、DocumentModified 和 DocumentDeleted 3 个事件。

一个针对文档的新特性可以将其自身注册为一个观察者，每当文件被创建、修改或删除时就能得到通知，并进行相应的工作。这样，可以在应用程序中添加一个文档索引程序。当然，这要求负责创建、修改或删除文档的代码触发事件。但是这比在应用程序代码库中到处添加索引的钩子程序要容易多了。

在 Python 中，可以用 Event 类来实现类级别观察者的注册，如下所示。

```
>>> class Event(object):
...     _observers = []
...     def __init__(self, subject):
...         self.subject = subject
...
...
...     @classmethod
...     def register(cls, observer):
...         if observer not in cls._observers:
...             cls._observers.append(observer)
...
...
...     @classmethod
...     def unregister(cls, observer):
...         if observer in cls._observers:
...             self._observers.remove(observer)
...
...
...     @classmethod
...     def notify(cls, subject):
...         event = cls(subject)
...         for observer in cls._observers:
...             observer(event)
...
...
```

思路是观察者使用 Event 类方法注册自己，并用携带触发这些事件的 Event 实例来获得通知，如下所示。

```
>>> class WriteEvent(Event):
...     def __repr__(self):
...         return 'WriteEvent'
...
...
```

```

>>> def log(event):
...     print '%s was written' % event.subject
...
>>> WriteEvent.register(log)
>>> class AnotherObserver(object):
...     def __call__(self, event):
...         print 'Yeah %s told me !' % event
...
>>> WriteEvent.register(AnotherObserver())
>>> WriteEvent.notify('a given file')
a given file was written
Yeah WriteEvent told me !

```

对这个实现，可以做以下改进：

- 允许开发人员修改顺序；
- 使事件对象保存比主题更多的信息。

 如果希望使用现有的工具，可以尝试 Pydispatch，它提供了一个很好的多消费者和多生产者的调度机制（参见 <http://www.sqlobject.org/module-sqlobject.include.pydispatch.html>）。

14.3.2 访问者

 访问者模式有助于将算法从数据结构中分离出来。

访问者和观察者有着相似的目标，都能在不修改代码的情况下扩展指定类的功能。但是访问者更进一步，它将定义一个负责保存数据的类，并将算法推进被称为访问者的其他类中。每个访问者专门用于一个算法并且可以将其应用到数据上，这种行为和 MVC 范式（参见 <http://en.wikipedia.org/wiki/Model-view-controller>）相当相似。在这种范式中，文档是被动的容器，通过控制器推送到视图，模式将包含控制器修改的数据。

访问者通过在数据类中提供一个可被所有类型的访问者访问的入口点来完成。一般的描述是，一个接受 Visitor 实例并调用它们的 Visitable 类，如图 14.4 所示。

Visitable 类决定如何调用 Visitor 类，例如，决定调用哪个方法。举个例子，一个负责打

对于内建类型内容的访问者可以实现 `visit_TYPENAME` 方法，每个类型可在其 `accept` 方法中调用指定的方法，如下所示。



图 14.4

```
>>> class vlist(list):
...     def accept(self, visitor):
...         visitor.visit_list(self)
...
...
...
...
...
>>> class vdict(dict):
...     def accept(self, visitor):
...         visitor.visit_dict(self)
...
...
...
...
>>> class Printer(object):
...     def visit_list(self, ob):
...         print 'list content :'
...         print str(ob)
...     def visit_dict(self, ob):
...         print 'dict keys: %s' % ','.join(ob.keys())
...
...
...
...
>>> a_list = vlist([1, 2, 5])
>>> a_list.accept(Printer())
list content :
[1, 2, 5]
>>> a_dict = vdict({'one': 1, 'two': 2, 'three': 3})
>>> a_dict.accept(Printer())
dict keys: one,three,two
```

但是这个模式意味着每个被访问的类都必须有一个 `accept` 方法，这很令人痛苦。

因为 Python 允许代码内省，因此自动链接访问者和被访问类是更好的主意，如下所示。

```
>>> def visit(visited, visitor):
...     cls = visited. class . name
```

```

...
    meth = 'visit_%s' % cls
...
    method = getattr(visitor, meth, None)
...
    if meth is None:
...
        meth(visited)

...
>>> visit([1, 2, 5], Printer())
list content :
[1, 2, 5]
>>> visit({'one': 1, 'two': 2, 'three': 3}, Printer())
dict keys: three,two,one

```

例如，这个模式在 `compiler.visitor` 模块中以这种方式被使用，`ASTVisitor` 类调用编译后的代码树的每个节点来调用访问者。这是因为，Python 没有匹配的操作符，如 Haskell。

另一个实例是一个根据文件扩展名调用访问者方法的目录遍历程序，如下所示。

```

>>> def visit(directory, visitor):
...     for root, dirs, files in os.walk(directory):
...         for file in files:
...             # foo.txt → .txt
...             ext = os.path.splitext(file)[-1][1:]
...             if hasattr(visitor, ext):
...                 getattr(visitor, ext)(file)
...
...
>>> class FileReader(object):
...     def pdf(self, file):
...         print 'processing %s' % file
...
...
>>> walker = visit('/Users/tarek/Desktop', FileReader())
processing slides.pdf
processing sholl23.pdf

```

如果应用程序中有被多个算法访问的数据结构，那么访问者模式将有助于分散关注点：对于一个数据容器，只关注于提供数据访问和存储功能，而不考虑其他的，这样做更好。

好的做法是创建没有任何方法的数据结构，就像 C 中的 `struct` 那样。

14.3.3 模板



模板模式通过定义抽象的步骤（在子类中实现）来帮助开发人员设计出通用的算法。

模板模式使用了 Liskov 替换原则：

“如果 S 是 T 的子类型，那么程序中类型 T 的对象可以使用类型 S 的对象替代，而不需要修改程序中任何所需的属性。”（Wikipedia）

换句话说，一个抽象类可以定义一个算法在具体类中实现的步骤。这个抽象类也可以给出算法的一个基本或部分的实现，让开发人员重载它的部件。例如，Queue 模块中 Queue 类的一些方法可以重载以改变它的行为。

接下来将实现如图 14.5 所示的一个例子。Indexer 是以 5 个步骤处理一个文本的索引程序，这些步骤是任何索引技术所共有的：

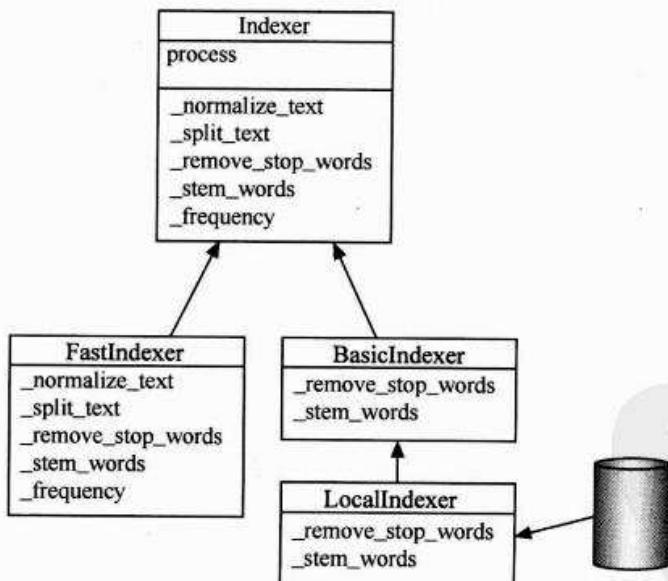


图 14.5

- 文本规范化；
- 文本拆分；
- 删 除无用词；
- 抽取词干；
- 频率计数。

`Indexer` 提供了处理算法的部分实现，但是要求在子类中实现`_remove_stop_words` 和`_stem_words` 方法。`BasicIndexer` 实现最小的部分，同时 `LocalIndex` 使用了一个无用词文件和一个词干数据库。`FastIndexer` 实现了所有步骤，可能将基于像 Xapian 或 Lucene 这样的快速索引程序。

下面是一个仅供娱乐的实现。

```
>>> class Indexer(object):
...     def process(self, text):
...         text = self._normalize_text(text)
...         words = self._split_text(text)
...         words = self._remove_stop_words(words)
...         stemmed_words = self._stem_words(words)
...         return self._frequency(stemmed_words)
...     def _normalize_text(self, text):
...         return text.lower().strip()
...     def _split_text(self, text):
...         return text.split()
...     def _remove_stop_words(self, words):
...         raise NotImplementedError
...     def _stem_words(self, words):
...         raise NotImplementedError
...     def _frequency(self, words):
...         counts = {}
...         for word in words:
...             counts[word] = counts.get(word, 0) + 1
```

由此，`BasicIndexer` 的实现可以如下所示。

```
>>> from itertools import groupby
>>> class BasicIndexer(Indexer):
...     _stop_words = ('he', 'she', 'is', 'and', 'or')
...     def _remove_stop_words(self, words):
...         return (word for word in words
...                 if word not in self._stop_words)
...     def _stem_words(self, words):
...         return ((len(word) > 2 and word.rstrip('aeiouy')
...                 or word)
...                 for word in words)
```

```
...     def _frequency(self, words):
...         freq = {}
...         for word in words:
...             freq[word] = freq.get(word, 0) + 1
...
...
>>> indexer = BasicIndexer()
>>> indexer.process('My Tailor is rich and he is also '
...                   'my friend')
{'tailor': 1, 'rich': 1, 'my': 2, 'als': 1, 'friend': 1}
```



对于可能变化并且可以表达为独立子步骤的算法，应该考虑模板模式——这可能是 Python 中最常用的模式。

14.4 小结

设计模式可复用，某种程度上是对软件设计中的常见问题提供语言相关的解决方案。它们是所有开发人员文化的一部分，不管他们用的是什么语言。

所以，为指定语言的最常用模式准备实现实例，是对该语言最好的评述。在各种网站上都有对每种 GoF 设计模式的 Python 实现，Python Cookbook (<http://aspn.activestate.com/ASPN/Python/Cookbook/>) 是一个特别值得一看的地方。