

提供各种IT类书籍pdf下载，如有需要，请 QQ: 2011705918

注：链接至淘宝，不喜者勿入！整理那么多资料不容易，请多多见谅！非诚勿扰！

更多资源请点击



HZ BOOKS

PEARSON

全球资深Python专家Doug Hellmann作序鼎力推荐，Amazon全五星评价，Python领域最有影响力的著作之一

从设计模式、并发技术和程序库角度，围绕Python编程的核心问题，系统而详细地讲解各种实用Python编程技术和技巧，并以3个完整的案例展示“设计-实现-优化”的全过程，带你领略Python语言之美，提升Python编程水平

华章程序员书库

Python in Practice

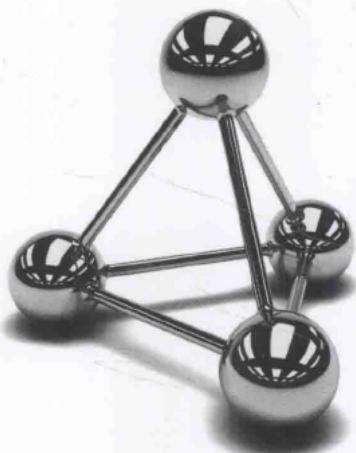
Create Better Programs Using Concurrency, Libraries, and Patterns

# Python编程实战

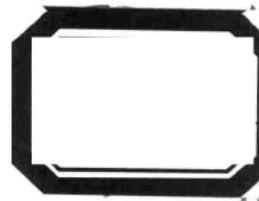
运用设计模式、并发和程序库创建  
高质量程序

(美) Mark Summerfield 著

爱飞翔 译



机械工业出版社  
China Machine Press

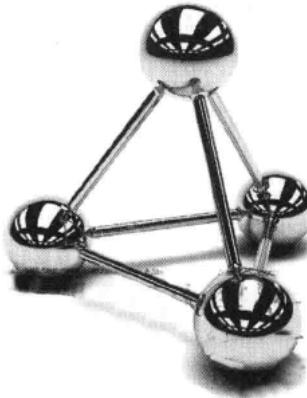


Python in Practice  
Create Better Programs Using Concurrency, Libraries, and Patterns

# Python编程实战

运用设计模式、并发和程序库创建  
高质量程序

(美) Mark Summerfield 著  
爱飞翔 译



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

Python 编程实战：运用设计模式、并发和程序库创建高质量程序 / (美) 萨默菲尔德 (Summerfield, M.) 著；爱飞翔译。—北京：机械工业出版社，2014.7  
(华章程序员书库)

书名原文：Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns

ISBN 978-7-111-47394-7

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

中国版本图书馆 CIP 数据核字 (2014) 第 161735 号

---

本书版权登记号：图字：01-2014-3566

Authorized translation from the English language edition, entitled *Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns*, 9780321905635 by Mark Summerfield, published by Pearson Education, Inc., Copyright © 2014 Qtrac Ltd.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2014.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签，无标签者不得销售。

# Python 编程实战： 运用设计模式、并发和程序库创建高质量程序

[美] Mark Summerfield 著

---

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：关 敏

责任校对：殷 虹

印 刷：北京市荣盛彩色印刷有限公司

版 次：2014 年 8 月第 1 版第 1 次印刷

开 本：186mm × 240mm 1/16

印 张：16.75

书 号：ISBN 978-7-111-47394-7

定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

## *The Translator's Words* 译者序

由于 Python 语言的写法非常简洁，而且应用范围又很广泛，所以近年来吸引了很多开发者积极投身其中。Python 语言的基础教程种类繁多，开发者的入门过程也特别快。在掌握了基础知识之后，许多程序员还想进一步提升自己的 Python 编程水平。这时主要会遇到三大问题。

第一是如何运用设计模式来规划代码结构，使之既易于修改，又易于维护；第二是如何通过并发及 Cython 等技术提升代码执行速度；第三是如何利用各种 Python 程序库来快速开发具体的应用程序和游戏。

针对这三大问题，本书都做了非常精彩的解答。作者把设计模式分为“创建型”、“结构型”与“行为型”三类，并分别比较了每种设计模式的传统用法与它在 Python 中的用法究竟有何异同。读者通过这部分讲解，可以学会如何在 Python 中简化或扩充传统的设计模式，也能明白为何有些模式在 Python 中无须使用。

许多开发者都想通过并发来提升程序性能，但由于要处理“资源竞争”、“加锁”等复杂的问题，所以并发式应用程序很容易出错，而本书作者则会告诉我们怎样利用 `queue` 及 `future` 等高级数据结构来避免这些错误，此外，他还辨析了多进程技术与多线程技术的适用场合。其后，作者又讲解了如何用 Python 来调用 C 或 C++，并通过实例证明了 Python 并不是一门效率低下的语言——只要合理运用并发及 Cython 技术，照样可以写出速度很快的程序。

在具体的应用领域中，作者着重讲解了如何利用 Python 程序库来简化网络及图形编程，并把图形编程又细分为用户界面编程和三维图形编程。由于 Python 的程序库非常丰富，所以读者只要把学到的解题思路灵活地运用到自己的工作领域，并辅以适当的程序库，就能快速开发出符合需求的程序。

本书不仅提供了大量实用的范例代码，而且还有三个完整的案例研究。这三个案例再现

了“设计—实现—优化”的全过程，并使我们领略到 Python 语言之美。

尽管本书作者认为读者应该有一定的 Python 基础，但 Python 其实是一门亲和力很高的语言，即便你对 Python 知之甚少，但只要用其他语言编过程序，也依然可以从 `for x in range(1, 10): print(x)`、`(2, 3, 4) * 2`、`[9, 8, 7, 6][1:3]` 等浅显易懂的写法中轻松得知循环、元组及列表等功能的用法。如果你想快速掌握 Python 语言，那不妨在开发文档的帮助下，试着直接读这本书。

总之，无论你是老手还是新手，本书都是学习 Python 的良好伴侣。相信大家在读完本书之后，应该能用其中的技术和思路把 Python 程序写得更加优秀。

翻译过程中，我得到了机械工业出版社华章公司诸位编辑与工作人员的帮助，在此深表感谢。

本书附有“中英文词汇对照表”，其中列出了相关词汇的各种译法，供大家参考。有兴趣的朋友可至华章公司网站 [www.hzbook.com](http://www.hzbook.com) 下载。由于时间仓促，译者水平有限，错误与疏漏之处在所难免，敬请读者批评指正。大家可发邮件至 [eastarstormlee@gmail.com](mailto:eastarstormlee@gmail.com) 与我联系，也可访问网页 <http://agilemobidev.com/eastarlee/book/python-in-practice> 留言。

爱飞翔

## *Preface* 序

这 15 年来，我用 Python 编程语言开发过各种应用程序，在此过程中，Python 开发者社区也逐渐成熟与壮大起来。从前，我们必须向管理人员“推销”Python 语言，以便其允许开发者在相关工作项目中使用它。而现在则不同了，懂 Python 语言的程序员在就业市场中备受青睐，而参加各种 Python 技术会议的人也络绎不绝，有些是区域性的会议，有些是大型的国内或国际会议。如 OpenStack 这样的项目既能拓宽 Python 语言的应用领域，又能吸引开发好手投入 Python 阵营。社区壮大之后，优秀的 Python 书籍也比原来多了。

Mark Summerfield 是 Python 开发者社区里的知名技术作家，写有 Qt 及 Python 等方面的著作。我是乔治亚州亚特兰大市 Python 开发者聚会的组织人，经常有开发者问我：学 Python 语言看什么书好？在我列出的推荐书目中，Mark 所著的《Programming in Python 3》名列榜首。Mark 的这本新作当然也在书单里，只是目标读者有些不同。

大部分编程书籍都位于两个极端：要么是向某门语言初学者（或刚开始学习编程之人）做一些简单介绍，要么就是专门讲解 Web 开发、图形用户界面应用程序开发、生物信息学等特定的高端话题。在写作《The Python Standard Library by Example》<sup>⊖</sup>一书时，我想填补两极之间的空白，也就是想写给那些有一定基础的程序员或计算机通才，这部分读者既想提升知识水平，又不愿局限在某个狭小的应用范围内。编辑请我审阅本书的出版提案时，我发现这本书的目标读者恰与我自己的书一致，于是颇感欣慰。

我始终在寻找这样一本书：它里面的内容不针对某个特定的框架或程序库，读者在阅读过程中一旦产生某个想法，就立刻能将其运用于目前正在做的项目中。过去几年里我一直在开发一套系统，用于计量 OpenStack 云服务。在这个过程中，开发团队发现：计费系统所收集到的数据也可以有其他用途，比如，报表系统和监视系统也能用到。于是，在设计系统时，我们通过一条管线来传递样本数据，以便将其转换并发布为各种格式，这样一来，就能把数据传给多个数据消费端了。当这条管线的代码快要写好时，我开始参与这本书的技术评审工作了。读完草稿第 3 章前面几节之后，我清楚地意识到，当初把管线实现得太过复杂

---

<sup>⊖</sup> 中译《Python 标准库》，刘炽等译，机械工业出版社 2012 年出版。——译者注

了。Mark 在书中演示的“协程链”技术非常优雅，而且易于理解，所以，我立刻在项目的“路线图”里加了个任务，计划在下个发行周期里用该技术来修改管线设计。

本书有很多实用的建议及范例代码，大家可以像上面所说的那样，用它们来改进自己的项目。此外，书中还介绍了许多有趣的编程技法，即便大家像我一样读过很多代码，也依然有可能发现一些原来不知道的技巧。无论是很有经验的老程序员，还是正在寻求自我提升的新手，本书都会引领大家从不同的角度来观察问题，使你能用学到的技术创建出更为高效的解决方案。

Doug Hellmann  
DreamHost 公司资深开发者  
2013 年 5 月

## Preface 前言

本书面向有志于拓展及深化 Python 知识的读者，它将教你如何改进 Python 程序的质量、可靠性、速度、可维护性以及可用性。书中包含大量实用的范例与思路，可帮助大家提升 Python 编程水平。

本书有四大主题：用设计模式编写出优雅的代码、用并发和“编译过的 Python”（也就是 Cython）提升处理速度、高级网络编程，以及图形。

《Design Patterns: Elements of Reusable Object-Oriented Software》（详情参见附录 B）一书虽然早在 1995 年就出版了，然而时至今日，依然深刻地影响着面向对象编程这一领域。本书从 Python 语言的角度重新审视前书所提到的每种设计模式，给出实用的 Python 范例，并解释为何 Python 程序员用不到某些模式。这些模式在第 1 章、第 2 章及第 3 章中讲解。

Python 的 GIL（Global Interpreter Lock，全局解释器锁）会阻止 Python 代码同时在多个处理器核心上运行<sup>⊖</sup>。于是有人就误以为 Python 不支持多线程，或无法发挥多核硬件的优势。对于“计算密集型”（CPU-bound）程序来说，可以用 `multiprocessing` 模块实现并发，该模块不受 GIL 限制，可以完全利用每个核心。这样一来，处理速度就很容易提高了（大致同 CPU 的核心数成正比）。对于“I/O 密集型”程序来说，我们既可以用 `multiprocessing` 模块来做，也可以用 `threading` 模块或 `concurrent.futures` 模块来做。实际上，使用 `threading` 模块来编写 I/O 密集型程序时，并不用担心由 GIL 所带来的开销，因为网络延迟的影响更大。

遗憾的是，在编写并发程序时，如果采用低级与中级方式<sup>⊖</sup>，那么非常容易出错，任

---

⊖ CPython 有此限制。CPython 是大部分 Python 程序员所使用的“参考实现”（reference implementation）。某些 Python 实现没有这一限制，这其中最有名的就是 Jython（也就是用 Java 语言所实现的 Python）。

⊖ 这里所说的“低级”（low-level）或“中级”（medium-level）方式，是指封装程度不那么高、偏底层的方式。——译者注

何编程语言都有这种问题。要想少出错，就不要使用“显式锁”（explicit lock），而是改用 Python 的 `queue` 及 `multiprocessing` 模块，这些模块提供了封装程度较高的“队列”（queue），此外，也可以改用 `concurrent.futures` 模块来做。第 4 章会告诉大家如何用封装程度较高的并发技术来大幅提高程序性能。

某些程序员之所以使用 C、C++ 或其他“编译型语言”（compiled language）来编程，是因为他们还有另外一个错误的想法，那就是 Python 程序运行得很慢。一般来说，Python 确实要比编译型语言慢，但在目前的硬件上面，用 Python 语言所编写的绝大部分应用程序的运行速度都足够快。即便有时 Python 程序真的不够快，我们也可以一边享受用 Python 编程所带来的好处，一边想办法提升其运行速度。

如果要给某些长期运行的程序提速，那么可以使用 PyPy 这款 Python 解释器（网址是 [pypy.org](http://pypy.org)）。这是一种“即时编译器”（just-in-time compiler），可以极大提升程序执行速度。另外一种优化执行效率的方式是调用运行速度与编译后的 C 程序相仿的代码，对于“计算密集型”程序来说，用这种代码改写后，其执行速度很容易变成原来的 100 倍。要想使 Python 程序运行得和 C 程序一样快，最简单的办法就是调用那种底层以 C 语言来实现的 Python 模块。比方说，标准库里的 `array` 模块或第三方 `numpy` 模块都能飞快地处理数组，并且很省内存（多维数组可以用 `numpy` 来处理）。除此之外，还可以使用标准库的 `cProfile` 模块来探查程序的瓶颈，并用 Cython 来写对速度要求很高的那部分代码。这种写法实际上就是一套“增强版 Python 语言”（enhanced Python）：写好的程序可以编译成纯 C，从而使运行速度提升到极致。

当然，有时候我们所需的功能已经由现成的 C 或 C++ 库实现好了，或者由采用“C 语言调用约定”（C calling convention）的其他语言程序库实现好了。在大多数情况下，都能找到第三方 Python 模块来访问我们所需的那些程序库，这些模块可以在 Python Package Index（简称 PyPI，网址是 [pypi.python.org](http://pypi.python.org)）里找到。不过在个别情况下可能找不到这种模块，此时可以用标准库的 `ctypes` 模块或第三方的 Cython 包来调用 C 程序库里的功能。采用已经实现好的 C 程序库来编写代码能够极大减少开发时间，而且 C 代码的运行速度也相当快。第 5 章讲解 `ctypes` 与 Cython。

Python 标准库提供了许多用于网络编程的模块，比如底层的 `socket` 模块、中层的 `socketserver` 模块，以及高层的 `xmlrpclib` 模块。把用其他语言所写的代码移植到 Python 时，可能会用到底层与中层网络模块，然而直接用 Python 编程时，通常不需要理会那些底层的细节，只需要用高层模块来实现所需的网络功能即可。第 6 章讲解如何使用标准库中的 `xmlrpclib` 模块以及功能强大且易用的第三方 RPyC 模块。

每个程序差不多都要提供某种用户界面，使用户可通过它来向程序下达指令。可以用

`argparse` 模块来编写 Python 程序，使其支持命令行界面；也可以用其他模块来编写，使其支持完整的终端用户界面（例如，在 Unix 系统上，可用第三方 `urwid` 包实现这种界面，此包的网址为 [excess.org/urwid](http://excess.org/urwid)）。此外，有许多 Web 框架能够实现出 Web 界面，比如轻量级的 `bottle` 框架（网址是 [bottlepy.org](http://bottlepy.org)）、重量级的 `Django` 框架（网址是 [www.djangoproject.com](http://www.djangoproject.com)）与 `Pyramid` 框架（网址是 [www.pylonsproject.org](http://www.pylonsproject.org)）。当然，除了上面说的这几种界面外，还可以创建具有“图形用户界面”（Graphical User Interface，GUI）的 Python 应用程序。

经常听到“Web 程序将取代 GUI 程序”这种说法，不过现在还没发展到那一步。实际上，用户可能更喜欢 GUI 程序，而不是 Web 程序。比方说，在 21 世纪初智能手机刚开始流行时，用户总是爱用专门制作好的“app”而不是浏览器中的网页来处理日常事务。有许多第三方 Python 包都可用来编写 GUI 程序。本书第 7 章要介绍的 `Tkinter` 包位于 Python 标准库里，该章会告诉大家如何用它创建样式新潮的 GUI 程序。

目前大多数计算机（包括笔记本电脑及部分智能手机）都配有功能强大的图形渲染硬件，这种硬件通常是独立的 GPU（Graphics Processing Unit，图形处理单元），能够绘制出绚丽的二维及三维图形。而大多数 GPU 都支持 OpenGL API，所以 Python 程序员可以通过第三方包来调用这套 API。第 8 章将会讲解怎样用 OpenGL 绘制三维图形。

本书旨在演示如何编写更好的 Python 程序，如何写出效率高、易维护且易于使用的 Python 代码。阅读之前，需要有 Python 编程基础，因为此书是写给已经学会 Python 语言用法的读者看的，大家应该已经读过 Python 的开发文档或是类似教程了，比如《Programming in Python 3, Second Edition》（详情参见附录 B）等书。而这本书将提供一些有助于提升 Python 编程水平的思路、灵感与实用技巧。

本书全部范例代码都在 Linux 系统的 Python 3.3 版本下测试通过（笔者也尽量在 Python 3.2 及 Python 3.1 版本下测试过了），绝大部分代码还能在 OS X 与 Windows 操作系统中运行。可从本书网站 [www.qtrac.eu/pipbook.html](http://www.qtrac.eu/pipbook.html) 下载范例代码，这些代码也应该能在后续的 Python 3.x 版本下运行。

## 致谢

与写其他技术书籍时一样，笔者写这本书时也得到了大家的诸多建议、帮助及鼓励，在此深表谢意。

Nick Coghlan 从 2005 年起成为 Python 的核心开发者，他提供了大量建设性的批评意见，并展示了许多想法及代码片段，以此表明书中所讲的某些内容还有更好的实现方式。Nick 对

笔者改进本书内容帮助极大，尤其是前面几章。

Doug Hellmann 是资深 Python 开发者与技术作者，他给笔者写了许多条非常有用的评论，从成书之前的出版提案到成书之后正文里的每一章都是如此。Doug 还给笔者提供了许多思路，并为本书撰写了序。

两位友人 Jasmin Blanchette 与 Trenton Schulz 都是有经验的 Python 程序员，他们各自的研究方向迥然不同，但都位于本书所讲的范围之内。Jasmin 与 Trenton 反馈了许多意见，使笔者能够据此改写正文及范例代码中的许多不够明晰之处。

感谢策划编辑 Debra Williams Cauley，在成书过程中，他再次向我提供了支持和帮助。

感谢 Elizabeth Ryan 精心管理了本书的出版流程，感谢 Anna V. Popick 出色的校对工作。

最后，一如往常，感谢妻子 Andrea 的关爱与支持。

## *Contents* 目 录

译者序

序

前言

### **第1章 Python 的创建型设计模式 ..... 1**

|                              |    |
|------------------------------|----|
| 1.1 抽象工厂模式 .....             | 1  |
| 1.1.1 经典的抽象工厂模式 .....        | 2  |
| 1.1.2 Python 风格的抽象工厂模式 ..... | 4  |
| 1.2 建造者模式 .....              | 6  |
| 1.3 工厂方法模式 .....             | 12 |
| 1.4 原型模式 .....               | 18 |
| 1.5 单例模式 .....               | 19 |

### **第2章 Python 的结构型设计模式 ..... 21**

|                                |    |
|--------------------------------|----|
| 2.1 适配器模式 .....                | 21 |
| 2.2 桥接模式 .....                 | 26 |
| 2.3 组合模式 .....                 | 31 |
| 2.3.1 常规的“组合体 / 非组合体”式层级 ..... | 32 |
| 2.3.2 只用一个类来表示组合体与非组合体 .....   | 35 |
| 2.4 修饰器模式 .....                | 37 |
| 2.4.1 函数修饰器与方法修饰器 .....        | 38 |
| 2.4.2 类修饰器 .....               | 42 |

|                                  |           |
|----------------------------------|-----------|
| 2.5 外观模式 .....                   | 47        |
| 2.6 享元模式 .....                   | 52        |
| 2.7 代理模式 .....                   | 54        |
| <b>第3章 Python 的行为型设计模式 .....</b> | <b>58</b> |
| 3.1 责任链模式 .....                  | 58        |
| 3.1.1 用常规方式实现责任链 .....           | 59        |
| 3.1.2 基于协程的责任链 .....             | 60        |
| 3.2 命令模式 .....                   | 63        |
| 3.3 解释器模式 .....                  | 66        |
| 3.3.1 用 eval() 函数求表达式的值 .....    | 67        |
| 3.3.2 用 exec() 函数执行代码 .....      | 70        |
| 3.3.3 用子进程执行代码 .....             | 73        |
| 3.4 迭代器模式 .....                  | 76        |
| 3.4.1 通过序列协议实现迭代器 .....          | 77        |
| 3.4.2 通过双参数 iter() 函数实现迭代器 ..... | 77        |
| 3.4.3 通过迭代器协议实现迭代器 .....         | 79        |
| 3.5 中介者模式 .....                  | 81        |
| 3.5.1 用常规方式实现中介者 .....           | 82        |
| 3.5.2 基于协程的中介者 .....             | 85        |
| 3.6 备忘录模式 .....                  | 87        |
| 3.7 观察者模式 .....                  | 87        |
| 3.8 状态模式 .....                   | 91        |
| 3.8.1 用同一套方法来处理不同的状态 .....       | 93        |
| 3.8.2 用不同的方法来处理不同的状态 .....       | 94        |
| 3.9 策略模式 .....                   | 95        |
| 3.10 模板方法模式 .....                | 98        |
| 3.11 访问者模式 .....                 | 101       |
| 3.12 案例研究：图像处理程序包 .....          | 102       |
| 3.12.1 通用的图像处理模块 .....           | 103       |
| 3.12.2 Xpm 模块概述 .....            | 111       |
| 3.12.3 PNG 包装器模块 .....           | 113       |

|                                      |            |
|--------------------------------------|------------|
| <b>第 4 章 Python 的高级并发技术 .....</b>    | <b>116</b> |
| 4.1 计算密集型并发 .....                    | 119        |
| 4.1.1 用队列及多进程实现并发 .....              | 121        |
| 4.1.2 用 Future 及多进程实现并发 .....        | 126        |
| 4.2 I/O 密集型并发 .....                  | 128        |
| 4.2.1 用队列及线程实现并发 .....               | 129        |
| 4.2.2 用 Future 及线程实现并发 .....         | 134        |
| 4.3 案例研究：并发式 GUI 应用程序 .....          | 136        |
| 4.3.1 创建 GUI .....                   | 138        |
| 4.3.2 编写与工作线程配套的 ImageScale 模块 ..... | 144        |
| 4.3.3 在 GUI 中显示图像处理进度 .....          | 146        |
| 4.3.4 处理 GUI 程序终止时的相关事宜 .....        | 148        |
| <b>第 5 章 扩充 Python .....</b>         | <b>150</b> |
| 5.1 用 ctypes 访问 C 程序库 .....          | 151        |
| 5.2 Cython 的用法 .....                 | 159        |
| 5.2.1 用 Cython 访问 C 程序库 .....        | 159        |
| 5.2.2 编写 Cython 模块以进一步提升程序执行速度 ..... | 164        |
| 5.3 案例研究：用 Cython 优化图像处理程序包 .....    | 169        |
| <b>第 6 章 Python 高级网络编程 .....</b>     | <b>173</b> |
| 6.1 编写 XML-RPC 应用程序 .....            | 174        |
| 6.1.1 数据包装器 .....                    | 174        |
| 6.1.2 编写 XML-RPC 服务器 .....           | 178        |
| 6.1.3 编写 XML-RPC 客户端 .....           | 180        |
| 6.2 编写 RPyC 应用程序 .....               | 188        |
| 6.2.1 线程安全的数据包装器 .....               | 188        |
| 6.2.2 编写 RPyC 服务器 .....              | 193        |
| 6.2.3 编写 RPyC 客户端 .....              | 195        |

|                                       |            |
|---------------------------------------|------------|
| <b>第 7 章 用 Tkinter 开发图形用户界面 .....</b> | <b>199</b> |
| 7.1 Tkinter 简介 .....                  | 201        |
| 7.2 用 Tkinter 创建对话框 .....             | 203        |
| 7.2.1 创建对话框式应用程序 .....                | 205        |
| 7.2.2 创建应用程序中的对话框 .....               | 212        |
| 7.3 用 Tkinter 创建主窗口式应用程序 .....        | 220        |
| 7.3.1 创建主窗口 .....                     | 222        |
| 7.3.2 创建菜单 .....                      | 224        |
| 7.3.3 创建带计分器的状态栏 .....                | 226        |
| <b>第 8 章 用 OpenGL 绘制 3D 图形 .....</b>  | <b>229</b> |
| 8.1 用透视投影法创建场景 .....                  | 230        |
| 8.1.1 用 PyOpenGL 编写 Cylinder 程序 ..... | 231        |
| 8.1.2 用 pyglet 编写 Cylinder 程序 .....   | 235        |
| 8.2 用正交投影法制作游戏 .....                  | 238        |
| 8.2.1 绘制游戏场景 .....                    | 240        |
| 8.2.2 判断用户是否选中了场景里的物体 .....           | 242        |
| 8.2.3 处理用户操作 .....                    | 244        |
| <b>附录 A 结束语 .....</b>                 | <b>248</b> |
| <b>附录 B 参考书目摘录 .....</b>              | <b>250</b> |

# Python 的创建型设计模式

关乎对象创建方式的设计模式就是“创建型设计模式”（creational design pattern）。一般我们都是通过调用构造器（也就是用参数来调用类对象）来创建对象的，但有时候需要以更为灵活的方式来创建对象，而这正是创建型设计模式的用途。

对于 Python 程序员来说，其中某些设计模式彼此之间非常相似，而另外一些则根本用不到（稍后就要讲到）。有些设计模式主要是为 C++ 这种语言设计的，目的是绕开这些编程语言中的某些限制。而 Python 语言没有这些限制，所以就用不到它们了。

## 1.1 抽象工厂模式

“抽象工厂模式”（Abstract Factory Pattern）用来创建复杂的对象，这种对象由许多小对象组成，而这些小对象都属于某个特定的“系列”（family）。

比方说，在 GUI 系统里可以设计“抽象控件工厂”（abstract widget factory），并设计三个“具体子类工厂”（concrete subclass factory）<sup>⊖</sup>：MacWidgetFactory、XfceWidgetFactory、WindowsWidgetFactor，它们都提供创建同一种对象的方法（例如都提供创建按钮的 make\_button() 方法，都提供创建数值调整框的 make\_spinbox() 方法），而具体创建出来的对象的风格则与操作系统平台相符。我们可以编写 create\_dialog() 函数，令其以“工厂实例”（factory instance）为参数来创建 OS X、Xfce 及 Windows 风格的对话框，对话框的具体风格取决于传进来的工厂参数。

<sup>⊖</sup> concrete 一词也译为“具型”、“具象”、“实体”。——译者注

### 1.1.1 经典的抽象工厂模式

为了演示抽象工厂模式，我们来写一段程序，用以生成简单的“示意图”（diagram）。这段程序会用到两个“工厂”（factory）：一个用来生成纯文本格式的示意图，另一个用来生成 SVG（Scalable Vector Graphics，可缩放的矢量图）格式的示意图。图 1.1 列出了这两种格式。此程序有两种写法，diagram1.py 文件按照传统方式来运用抽象工厂模式，而 diagram2.py 则借助了 Python 的某些特性，这样写出来的程序比原来更短小、更清晰。这两个版本所生成的示意图都一样<sup>①</sup>。

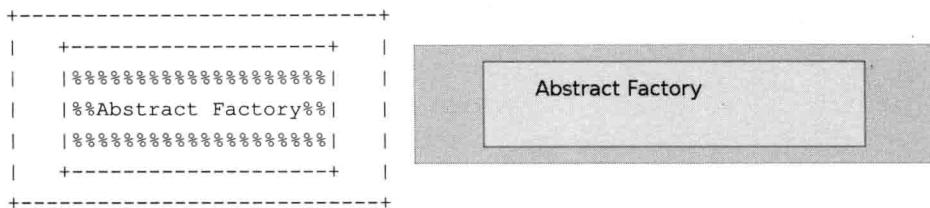


图 1.1 纯文本格式与 SVG 格式的示意图

有一些代码是这两个版本都要用的，首先我们来看 main() 函数。

```
def main():
    ...
    txtDiagram = create_diagram(DiagramFactory()) ❶
    txtDiagram.save(textFilename)

    svgDiagram = create_diagram(SvgDiagramFactory()) ❷
    svgDiagram.save(svgFilename)
```

首先创建两个文件（上述范例代码中没有列出相关语句）。接下来，用默认的纯文本工厂（❶）创建示意图，并将其保存。然后，用 SVG 工厂（❷）来创建同样的示意图，也将其保存。

```
def create_diagram(factory):
    diagram = factory.make_diagram(30, 7)
    rectangle = factory.make_rectangle(4, 1, 22, 5, "yellow")
    text = factory.make_text(7, 3, "Abstract Factory")
    diagram.add(rectangle)
    diagram.add(text)
    return diagram
```

create\_diagram 函数只有一个参数，就是绘图所用的工厂，该函数用这个工厂创建出所需的示意图。此函数并不知道工厂的具体类型，也无须关心这一点，它只需要知道工厂对象具备创建示意图所需的接口即可。以 make 开头的那些方法我们放在后面讲。

说完工厂的用法之后，我们来看工厂本身的写法。下面这个工厂类用来绘制纯文本示意图（该工厂也是其他工厂的基类）：

---

<sup>①</sup> 本书全部范例代码均可从 [www.qtrac.eu/pipbook.html](http://www.qtrac.eu/pipbook.html) 下载。

```

class DiagramFactory:

    def make_diagram(self, width, height):
        return Diagram(width, height)

    def make_rectangle(self, x, y, width, height, fill="white",
                      stroke="black"):
        return Rectangle(x, y, width, height, fill, stroke)

    def make_text(self, x, y, text, fontsize=12):
        return Text(x, y, text, fontsize)

```

虽说“抽象工厂模式”的名字里有“抽象”这个词，但实际上我们可以用一个类来身兼二职：这个类既像基类那样定义抽象接口，又像具体类那样提供实现代码。DiagramFactory 类就是按照这个思路写出来的。

创建 SVG 示意图所用的工厂叫做 SvgDiagramFactory，该类的前几行代码是：

```

class SvgDiagramFactory(DiagramFactory):

    def make_diagram(self, width, height):
        return SvgDiagram(width, height)

    ...

```

这两个 make\_diagram 方法之间的唯一区别在于：DiagramFactory.make\_diagram() 方法返回的是 Diagram 对象，而 SvgDiagramFactory.make\_diagram() 方法返回的是 SvgDiagram 对象。SvgDiagramFactory 里的另外两个方法也是如此（这两个方法没有列在上述范例代码中）。

稍后我们会看到，虽然对应类的接口都一样（比如 Diagram 与 SvgDiagram 类的方法名都相同），但是绘制纯文本示意图所用的 Diagram、Rectangle、Text 等类的实现方式却与 SVG 示意图所用的 SvgDiagram、SvgRectangle、SvgText 等类截然不同。这意味着不同系列的类之间不可混搭（比如 Rectangle 和 SvgText 就不能放在一张示意图里），相关的工厂类会自行保证这一点。

纯文本 Diagram 对象用“二维列表”（list of lists）来保存示意图里的数据，这些数据就是空格、+、|、- 等字符。纯文本的 Rectangle 及 Text 对象也包含由单个字符所构成的二维列表，它们可用来替换大示意图相关位置上的字符（如有必要，还会替换右侧及下方的字符）。

```

class Text:

    def __init__(self, x, y, text, fontsize):
        self.x = x
        self.y = y
        self.rows = [list(text)]

```

上面这几行就是 Text 类的全部代码了。由于是纯文本，所以无须理会 fontSize 参数。

```

class Diagram:
    ...

```

```
def add(self, component):
    for y, row in enumerate(component.rows):
        for x, char in enumerate(row):
            self.diagram[y + component.y][x + component.x] = char
```

上面是 `Diagram.add()` 方法的代码。调用该方法时, `component` 参数可能会是 `Rectangle` 或 `Text` 对象, 该方法会遍历 `component` 里的二维列表(也就是 `components.rows`), 用其中的数据来替换示意图相应位置上的字符。示意图本身的字符是由 `Diagram.__init__()` 方法绘制的(该方法没有列在上面的程序清单中), 调用 `Diagram(width, height)` 时, `__init__()` 方法会按照给定的宽度与高度用空格把 `self.diagram` 填充好。

```
SVG_TEXT = """<text x="{x}" y="{y}" text-anchor="left" \
font-family="sans-serif" font-size="{fontsize}">{text}</text>"""
SVG_SCALE = 20
class SvgText:
    def __init__(self, x, y, text, fontsize):
        x *= SVG_SCALE
        y *= SVG_SCALE
        fontsize *= SVG_SCALE // 10
        self.svg = SVG_TEXT.format(**locals())
```

上面列出了 `SvgText` 类的全部代码以及该类所依赖的两个常量<sup>⊖</sup>。顺便说一句, 使用 `**locals()` 的好处是比较省事, 这样就不用再写成 `SVG_TEXT.format(x=x, y=y, text=text, fontsize=fontsize)` 了。从 Python 3.2 开始, 还可以把 `SVG_TEXT.format(**locals())` 写成 `SVG_TEXT.format_map(locals())`, 因为 `str.format_map()` 方法会自动执行“映射解包”(mapping unpacking) 操作。(参见 1.2 节中的补充知识。)

```
class SvgDiagram:
    ...
    def add(self, component):
        self.diagram.append(component.svg)
```

`SvgDiagram` 类的每个实例都有一份字符串列表, 名叫 `self.diagram`, 列表中的每个字符串都表示一行 SVG 文本。这样一来, 向其中加入新组件(比如 `SvgRectangle` 或 `SvgText` 类型的对象)就变得非常简单了。

### 1.1.2 Python 风格的抽象工厂模式

在上一小节中, 我们分别用 `DiagramFactory` 和其子类 `SvgDiagramFactory` 来创建示意图里的各种组件(比如 `Diagram`、`SvgDiagram` 等), 并以此很好地演示了“抽象工厂”

---

<sup>⊖</sup> 笔者所绘制的 SVG 相当粗糙, 然而足以演示我们这里要讲的设计模式了。大家可以访问 [pypi.python.org](http://pypi.python.org), 在 Python Package Index (PyPI) 里寻找第三方 SVG 模块。

这一设计模式。

不过，刚才那种写法有几个缺点。首先，这两个工厂都没有各自的状态，所以根本不需要创建工作实例。其次，`SvgDiagramFactory`与`DiagramFactory`的代码基本上一模一样，只不过前者的`make_diagram`方法返回`SvgText`实例，而后者返回`Text`实例，其他几个方法也如此，这会产生许多无谓的重复代码。最后，`DiagramFactory`、`Diagram`、`Rectangle`、`Text`类以及SVG系列中与其对应的那些类都放在了“顶级命名空间”(top-level namespace)里。但实际上并没有必要这么做，因为我们只会用到那两个工厂而已。另外，这样做还有个坏处：给SVG示意图的组件类起名时，为了避免名称冲突，必须加上前缀才行(例如表示SVG矩形的那个类要叫做`SvgRectangle`，而不能直接叫成`Rectangle`)，这样代码就显得不够整洁了。(有种避免名称冲突的办法，就是把每个类都放到各自的模块中，然而这并不能消除重复代码。)

本节将用另外一种写法来弥补上述缺陷。(写好的代码放在`diagram2.py`文件中。)

第一处改动是把`Diagram`、`Rectangle`、`Text`等类都嵌套到`DiagramFactory`类里面。修改之后，需要用`DiagramFactory.Diagram`来引用纯文本的`Diagram`类，其余类也是如此。创建SVG示意图所用的那些类也可以嵌套到`SvgDiagramFactory`类里面，这样就不会产生名称冲突了，它们可以和纯文本系列的那些类同名，比方说，表示SVG示意图的那个类也能叫做`Diagram`，我们可通过`SvgDiagramFactory.Diagram`来引用它。类所依赖的常量也可以嵌套到工厂里面，于是顶级命名空间里只剩下`main()`、`create_diagram()`、`DiagramFactory`及`SvgDiagramFactory`了。

```
class DiagramFactory:
    @classmethod
    def make_diagram(Class, width, height):
        return Class.Diagram(width, height)

    @classmethod
    def make_rectangle(Class, x, y, width, height, fill="white",
                       stroke="black"):
        return Class.Rectangle(x, y, width, height, fill, stroke)

    @classmethod
    def make_text(Class, x, y, text, fontsize=12):
        return Class.Text(x, y, text, fontsize)
    ...

```

上面列出了新版`DiagramFactory`类的前几行代码。以`make`开头的方法现在都变成了“类方法”(class method)。也就是说，调用这些方法时，其首个参数是类，而不像普通方法那样，首个参数是`self`。例如，当调用`DiagramFactory.make_text()`方法时，`Class`参数就是`DiagramFactory`，此方法会创建`DiagramFactory.Text`对象并将其返回。

这种修改方式意味着 `SvgDiagramFactory` 子类只需继承 `DiagramFactory`, 而不用再去实现那几个 `make` 方法了。举例来说, 调用 `SvgDiagramFactory.make_rectangle()` 方法时, 由于 `SvgDiagramFactory` 类并没有实现这个方法, 所以会执行基类的 `DiagramFactory.make_rectangle()` 方法, 而执行的时候, `Class` 参数的值是 `SvgDiagramFactory`。这样一来, 基类方法自然就能创建出 `SvgDiagramFactory.Rectangle` 对象并将其返回了。

```
def main():
    ...
    txtDiagram = create_diagram(DiagramFactory)
    txtDiagram.save(textFilename)

    svgDiagram = create_diagram(SvgDiagramFactory)
    svgDiagram.save(svgFilename)
```

经过上述修改之后, `main()` 函数也可以简化, 因为现在不需要再创建工作类的实例了。其余代码和上一小节基本相同, 但有个显著的区别, 就是在访问相关常数及非工厂类时, 必须在名称前面加上工厂类的名字, 因为现在它们都嵌套在工厂类里了。

```
class SvgDiagramFactory(DiagramFactory):
    ...
    class Text:
        def __init__(self, x, y, text, fontsize):
            x *= SvgDiagramFactory.SVG_SCALE
            y *= SvgDiagramFactory.SVG_SCALE
            fontsize *= SvgDiagramFactory.SVG_SCALE // 10
            self.svg = SvgDiagramFactory.SVG_TEXT.format(**locals())
```

上面列出了 `Text` 类的代码, 该类嵌套在 `SvgDiagramFactory` 里面(也就相当于 `diagram1.py` 文件里的 `SvgText` 类), 这段代码还演示了如何访问嵌套在类中的常量。

## 1.2 建造者模式

“建造者模式”<sup>②</sup> (Builder Pattern) 与抽象工厂模式类似, 都可以创建那种需要由其他对象组合而成的复杂对象。而建造者与抽象工厂的区别则在于, 它不仅提供了创建复杂对象所需的方法, 而且还保存了复杂对象里各个部分的细节。

此模式与抽象工厂模式一样, 都能用来拼装对象(也就是用一个或几个简单的对象创建出复杂的对象), 但前者尤其适用于需要把复杂对象各部分的细节与其创建流程相分离的场合。

我们用一段“表单”(form)生成程序来演示建造者模式, 这段程序既可以用 HTML 生成网页表单, 又可以通过 Python 及 Tkinter 生成 GUI 表单。两种表单都具备图形用户界面, 用

---

<sup>②</sup> 也译为“生成器模式”、“构建者模式”。——译者注

户可以向其中输入文本，但是表单上的按钮不起作用<sup>①</sup>。图 1.2 演示了这两种表单，程序源代码位于 formbuilder.py 文件中。

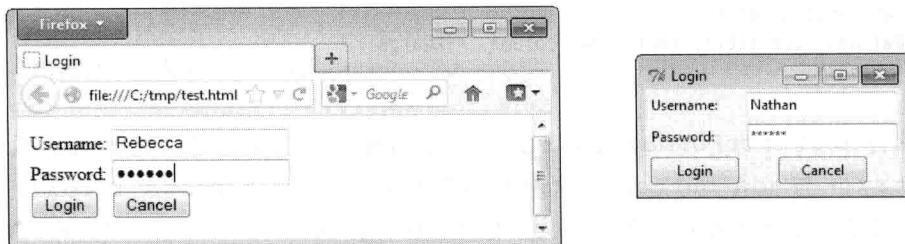


图 1.2 HTML 表单与 Tkinter 表单在 Windows 操作系统中的显示效果

我们从最顶层的调用语句开始，看看构建每个表单所用的代码。

```
htmlForm = create_login_form(HtmlFormBuilder())
with open(htmlFilename, "w", encoding="utf-8") as file:
    file.write(htmlForm)

tkForm = create_login_form(TkFormBuilder())
with open(tkFilename, "w", encoding="utf-8") as file:
    file.write(tkForm)
```

上面这段代码创建了两个表单，并分别将其写入对应的文件中。在这两种情况下，都会调用同一个表单创建函数（也就是 `create_login_form()`），每次调用时，传入与之相应的建造者对象。

```
def create_login_form(builder):
    builder.add_title("Login")
    builder.add_label("Username", 0, 0, target="username")
    builder.add_entry("username", 0, 1)
    builder.add_label("Password", 1, 0, target="password")
    builder.add_entry("password", 1, 1, kind="password")
    builder.add_button("Login", 2, 0)
    builder.add_button("Cancel", 2, 1)
    return builder.form()
```

此函数可以创建 HTML 与 Tkinter 表单，而且只要有适当的建造者对象，它就能创建出任意形式的表单。`builder.add_title()` 方法用于创建表单的标题，而其他方法则会把表单中的“控件”(widget) 添加到给定的行、列位置上。

`HtmlFormBuilder` 与 `TkFormBuilder` 都继承自抽象基类 `AbstractFormBuilder`。

```
class AbstractFormBuilder(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def add_title(self, title):
        self.title = title

    @abc.abstractmethod
```

① 范例代码既要实用，又要便于学习，所以笔者有时只实现了一些最基本的功能，比如本例就是如此。

```

def form(self):
    pass

@abc.abstractmethod
def add_label(self, text, row, column, **kwargs):
    pass
...

```

继承自 `AbstractFormBuilder` 的类必须实现所有抽象方法。上述代码没有列出 `add_entry()` 与 `add_button()` 这两个抽象方法，因为它们和 `add_label()` 方法非常相似，只是名字不同而已。另外，为了使用 `abc` 模块的 `@abstractmethod` “修饰器”（decorator），我们必须把 `AbstractFormBuilder` 类的 `metaclass`（元类）设置成 `abc.ABCMeta`。（2.4 节将详述修饰器。）

## 序列与映射的解包操作



“解包”（unpacking）就是把“序列”（sequence）或“映射”（map）中的每个元素单独提取出来。“序列解包”（sequence unpacking）的一种简单用法是把首个或前几个元素与后面几个元素分别提取出来。比如像下面这样：

```
first, second, *rest = sequence
```

如果 `sequence` 里至少有三个元素，那么执行完上述代码之后，`first == sequence[0]`, `second == sequence[1]`, `rest == sequence[2:]`。

最常用到解包操作的地方可能是函数调用语句。如果函数接受一定数量的“位置参数”（positional argument）<sup>⊖</sup>或某些特定的“关键字参数”（keyword argument），那么可以通过解包操作来提供这些参数。例如：

```

args = (600, 900)
kwargs = dict(copies=2, collate=False)
print_setup(*args, **kwargs)

```

`print_setup()` 函数需要两个位置参数（名为 `width` 及 `height`），并且还有两个可选的关键字参数（名为 `copies` 与 `collate`）。上面这段代码在调用函数时没有直接传递参数值，而是先创建了名为 `args` 的 `tuple` 与名为 `kwargs` 的 `dict`，然后通过序列解包操作 (`*args`) 与映射解包操作 (`**kwargs`) 来传递参数。这么做的效果与 `print_setup(600, 900, copies=2, collate=False)` 相同。

另一种与函数调用有关的用法是以解包符号来声明参数，使函数能够接受任意数量的位置参数或任意数量的关键字参数，也可以同时接受这两类参数。例如：

```

def print_args(*args, **kwargs):
    print(args.__class__.__name__, args,

```

<sup>⊖</sup> 也称“位置相关参数”、“必选参数”。——译者注

```

    kwargs.__class__.__name__, kwargs)
print_args() # prints: tuple () dict {}
print_args(1, 2, 3, a="A") # prints: tuple (1, 2, 3) dict {'a': 'A'}

```

上面这段代码所声明的 `print_args()` 函数可以接受任意数量的位置参数或关键字参数。在函数中，`args` 参数的类型是 `tuple`，而 `kwargs` 参数的类型则是 `dict`。如果还想在 `print_args()` 函数里用这些参数来调用别的函数，那么可以把带有解包符号的参数直接传进去（比如：`function(*args, **kwargs)`）。映射解包操作还有一种常见用法，就是拿它来调用 `str.format()` 方法。比方说，我们可以直接写 `s.format(**locals())`，而不用把每个参数都按照 `key=value` 的形式手工传进去（此用法的范例可参见 1.1.1 节的 `SvgText.__init__()` 方法）。

如果某个类的 `metaclass` 是 `abc.ABCMeta`，那么该类就无法初始化了，只能把它当成抽象基类来用。把 C++ 或 Java 代码移植到 Python 时，这么做确实很有用，不过会稍微增加运行期的开销。许多 Python 程序员都采用一种更为宽松的做法，那就是根本不使用 `metaclass`，而是直接在文档中说明该类只能用作抽象基类。

```

class HtmlFormBuilder(AbstractFormBuilder):
    def __init__(self):
        self.title = "HtmlFormBuilder"
        self.items = {}

    def add_title(self, title):
        super().add_title(escape(title))

    def add_label(self, text, row, column, **kwargs):
        self.items[(row, column)] = ('<td><label for="{}">{}:</label></td>'
            .format(kwargs["target"], escape(text)))

    def add_entry(self, variable, row, column, **kwargs):
        html = """<td><input name="{}" type="{}" /></td>""".format(
            variable, kwargs.get("kind", "text"))
        self.items[(row, column)] = html
    ...

```

上面列出了 `HtmlFormBuilder` 类的前几行代码。如果构建表单时没有指定标题，那么 `__init__` 方法会提供默认的标题。表单里的控件都保存在名为 `items` 的字典中，字典的键是由 `row` 与 `column` 所构成的“二元组”(2-tuple)，而值则是控件的 HTML 代码。

由于基类的 `add_title()` 方法是抽象的，所以本类必须重新实现它，不过在实现的时候，我们可以直接调用基类的实现代码。对于 `HtmlFormBuilder` 来说，在调用基类的 `add_title()` 之前，还必须先用 `html.escape()` 函数把 `title` 参数处理一下（在 Python 3.2 及之前的版本中，应该使用 `xml.sax.saxutil.escape()` 函数处理）。

`add_button()` 方法（此方法没有列在上述范例代码中）的结构与其他以 `add` 开头的

方法相似。

```
def form(self):
    html = ["<!doctype html>\n<html><head><title>{}</title></head>"
            "<body>".format(self.title), '<form><table border="0">']
    thisRow = None
    for key, value in sorted(self.items.items()):
        row, column = key
        if thisRow is None:
            html.append("  <tr>")
        elif thisRow != row:
            html.append("  </tr>\n  <tr>")
            thisRow = row
        html.append("    " + value)
    html.append("  </tr>\n</table></form></body></html>")
    return "\n".join(html)
```

`HtmlFormBuilder.form()` 方法创建了一个 HTML 页面，其中有个 `<form>`，而 `<form>` 里又有个 `<table>`，表单中的各个控件就保存在 `<table>` 的单元格里。把所有 HTML 代码都加到名为 `html` 的列表之后，我们在列表的各元素之间插入换行符，以美化代码排版，然后把所有 HTML 代码当成一个字符串返回给调用者。

```
class TkFormBuilder(AbstractFormBuilder):
    def __init__(self):
        self.title = "TkFormBuilder"
        self.statements = []

    def add_title(self, title):
        super().add_title(title)

    def add_label(self, text, row, column, **kwargs):
        name = self._canonicalize(text)
        create = """self.{name}Label = ttk.Label(self, text="{}")""".format(
            name, text)
        layout = """self.{name}Label.grid(row={}, column={}, sticky=tk.W, \
padx="0.75m", pady="0.75m")""".format(name, row, column)
        self.statements.extend((create, layout))

    ...
    def form(self):
        return TkFormBuilder.TEMPLATE.format(title=self.title,
                                              name=self._canonicalize(self.title, False),
                                              statements="\n          ".join(self.statements))
```

上面节选了 `TkFormBuilder` 类的部分代码。我们把创建表单控件所用的语句（也就是以字符串形式保存的 Python 代码）放到列表中，每个控件用两条语句来创建。

`add_entry()` 与 `add_button()` 方法的代码都没有列在上面，不过其结构与 `add_label()` 方法相同。这些方法都是先取得控件的“规范名称”（canonicalized name），然后再声明两个字符串：`create` 字符串中的代码用于构造控件，而 `layout` 字符串中的代码则用于

调整控件在表单中的位置。最后，这些方法都会把这两个字符串放到 statements 列表中。

form() 方法非常简单，它用 title 及 statements 来填充 TEMPLATE 模板，并把填好的字符串返回。

```
TEMPLATE = """#!/usr/bin/env python3
import tkinter as tk
import tkinter.ttk as ttk

class {name}Form(tk.Toplevel): ❶
    def __init__(self, master):
        super().__init__(master)
        self.withdraw()      # hide until ready to show
        self.title("{title}") ❷
        {statements} ❸
        self.bind("<Escape>", lambda *args: self.destroy())
        self.deiconify()     # show when widgets are created and laid out
        if self.winfo_viewable():
            self.transient(master)
        self.wait_visibility()
        self.grab_set()
        self.wait_window(self)

if __name__ == "__main__":
    application = tk.Tk()
    window = {name}Form(application) ❹
    application.protocol("WM_DELETE_WINDOW", application.quit)
    application.mainloop()
"""

"""

```

在这份模板代码中，我们先根据 title 来创建表单类（在本例中，title 是 Login，所以创建出来的类就叫做 LoginForm，参见❶与❹）。\_\_init\_\_() 方法先设置表单的标题（在本例中，标题是 Login，参见❷），然后用 statements 中的那些语句来创建表单中的控件，并调整其位置（❸）。

由于代码块最后有 if, \_\_name\_\_... 等语句，所以用这份模板所生成的 Python 代码可以独立运行。

```
def _canonicalize(self, text, startLower=True):
    text = re.sub(r"\W+", "", text)
    if text[0].isdigit():
        return "_" + text
    return text if not startLower else text[0].lower() + text[1:]
```

为了使范例代码看起来完整一些，我们把 \_canonicalize() 方法的代码也列出来。从这段代码本身来看，似乎每次执行此函数时都要重新创建正则表达式，但实际上 Python 有非常庞大的内部缓存，用于存放编译过的正则表达式。只要调用过 \_canonicalize() 方法，那么后续调用时就可以直接从缓存中查找正则表达式了，不用再重新编译<sup>②</sup>。

---

<sup>②</sup> 本书假定读者熟悉正则表达式及 Python 语言的 re 模块的基本用法。如果想学习这部分内容，可从网站 [www.py3book.html](http://www.py3book.html) 免费下载《Programming in Python 3, Second Edition》第 13 章的 PDF 文档。

### 1.3 工厂方法模式

如果子类的某个方法要根据情况来决定用什么类去实例化相关对象，那么可以考虑工厂方法模式。此模式可单独使用，也可在无法预知对象类型时使用（比方说，待初始化的对象类型要从文件中读入，或是由用户来输入）。

本节编写一段棋盘生成程序，用以生成“国际跳棋”(checker) 和“国际象棋”(chess) 的棋盘。该程序所输出的两张棋盘如图 1.3 所示。这段程序有四个版本，其源代码分别存放在 gameboard1.py 至 gameboard4.py 中<sup>⊖</sup>。

我们先设计出抽象的棋盘类，然后用其子类创建特定的棋盘。每个子类都会生成相应的棋盘，并把棋子摆放好。每个棋子也有对应的类（比如黑色的跳棋棋子用 BlackDraught 类表示，白色的跳棋棋子用 WhiteDraught 类

表示，黑色的“象”用 BlackChessBishop 表示，白色的“马”用 WhiteChessKnight 表示）。为了和 Unicode 中的字符名称保持一致，我们在表示跳棋棋子时使用了 Draught 一词，而没有使用 Checker，比如白色的跳棋棋子叫做 WhiteDraught，而不叫 WhiteChecker。

我们打算先讲最顶层的代码，这部分代码用于实例化棋盘对象，并把棋盘打印到控制台。然后来看表示棋盘的类和表示棋子的类。一开始，我们采用“硬代码”类的方式来创建这些棋子。然后设法将其改写，去掉那些硬代码类，并缩减代码行数。

```
def main():
    checkers = CheckersBoard()
    print(checkers)

    chess = ChessBoard()
    print(chess)
```

四个版本的程序都要用到上面这个函数。该函数分别创建两种棋盘对象，并将其打印到控制台，打印的时候会调用 AbstractBoard 的 `__str__()` 方法，以便将棋盘对象的内容转换成字符串。

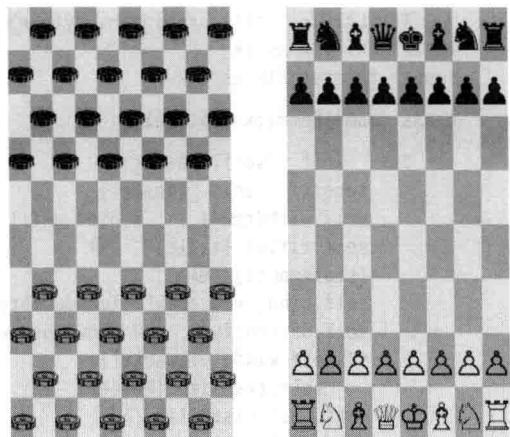


图 1.3 在 Linux 控制台中绘制的国际跳棋与国际象棋棋盘

<sup>⊖</sup> 由于 Windows 系统的控制台对 UTF-8 支持得不好，所以棋盘上很多棋子都显示不出来，即便把代码页设置成 65001，也还是无法显示这些字符。于是，在 Windows 平台上，这段程序会把输出信息写入临时文件，并在控制台中打印出该文件的名字。虽说 Windows 的“变宽字型”(variable-width font) 能够显示出国际跳棋和国际象棋的棋子，但是其标准的“等宽字型”(monospaced font) 似乎不行。“自由并开源”(free and open source) 的 DejaVu 字型 (dejavu-fonts.org) 支持这些字符。

```

BLACK, WHITE = ("BLACK", "WHITE")

class AbstractBoard:

    def __init__(self, rows, columns):
        self.board = [[None for _ in range(columns)] for _ in range(rows)]
        self.populate_board()

    def populate_board(self):
        raise NotImplementedError()

    def __str__(self):
        squares = []
        for y, row in enumerate(self.board):
            for x, piece in enumerate(row):
                square = console(piece, BLACK if (y + x) % 2 else WHITE)
                squares.append(square)
            squares.append("\n")
        return "".join(squares)

```

BLACK 与 WHITE 常量表示棋盘格子的背景色。在后续版本中，还会用来表示棋子的颜色。上面这段代码是从 `gameboard1.py` 中节录的，其他三个版本也与此相同。

`BLACK, WHITE = ("BLACK", "WHITE")` 这行代码本来可以按惯例写成 `BLACK, WHITE = range(2)`。但是用字符串来定义常量在调试时更容易看出错误信息的含义，而且 Python 还会自动把内容相同的字符串规整起来，只保留一份<sup>⊖</sup>。

棋盘对象里包含一份二维列表，其中每个一维列表都表示棋盘中的一行，而一维列表中的元素则表示行中对应单元格上的棋子，如果某个格子上没有棋子，那么对应的元素就是 `None`。`console()` 函数（此函数没有出现在上述程序清单中）所返回的字符串用于表示棋子及其背景色。（在“类 Unix”系统中，`console()` 函数所返回的字符串里会包含转义符，用于修改字符的背景色。）

你可以把 `AbstractBoard` 类的 `metaclass` 设置成 `abc.ABCMeta`（1.2 节中的 `AbstractFormBuilder` 类就是如此），这样的话，它就成了真正的抽象基类。不过此处我们改用另一种做法：凡是需要由子类重新实现的方法都抛出 `NotImplementedError` 异常。

```

class CheckersBoard(AbstractBoard):

    def __init__(self):
        super().__init__(10, 10)

    def populate_board(self):
        for x in range(0, 9, 2):
            for row in range(4):
                column = x + ((row + 1) % 2)
                self.board[row][column] = BlackDraught()
                self.board[row + 6][column] = WhiteDraught()

```

---

⊖ 原文为“interning and identity checks”。string interning（字符串驻留）技术可参考：[https://en.wikipedia.org/wiki/String\\_interning](https://en.wikipedia.org/wiki/String_interning)。——译者注

上述子类用于创建  $10 \times 10$  的国际跳棋棋盘。该类的 `populate_board()` 目前还算不上工厂方法，因为它是用硬编码的类来实例化棋子对象的，稍后我们会以此为基础将之改写成工厂方法。

```
class ChessBoard(AbstractBoard):
    def __init__(self):
        super().__init__(8, 8)

    def populate_board(self):
        self.board[0][0] = BlackChessRook()
        self.board[0][1] = BlackChessKnight()
        ...
        self.board[7][7] = WhiteChessRook()
        for column in range(8):
            self.board[1][column] = BlackChessPawn()
            self.board[6][column] = WhiteChessPawn()
```

在 `gameboard1.py` 这一版程序中，`ChessBoard` 类的 `populate_board()` 方法与 `CheckersBoard` 类的同名方法一样，都不能称为工厂方法，不过，由上面这段代码我们可以看出国际象棋的棋盘是如何生成的。

```
class Piece(str):
    __slots__ = ()
```

上面这个 `Piece` 类是所有棋子的基类，本来也可以直接用 `str` 表示，但如果那样做的话，就没办法判断某个对象是不是棋子了（比如我们想用 `isinstance(x, Piece)` 来判断 `x` 对象是不是棋子）。`__slots__ = ()` 这行语句可以保证实例中不会有任何数据，我们把这个话题放在 2.6 节中讨论。

```
class BlackDraught(Piece):
    __slots__ = ()
    def __new__(Class):
        return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):
    __slots__ = ()
    def __new__(Class):
        return super().__new__(Class, "\N{white chess king}")
```

上面这两个类是所有棋子的范本。每种棋子所对应的类都是 `Piece` 的子类，而 `Piece` 本身又是 `str` 的子类，棋子对象都是“不可变的”（`immutable`），我们用与之对应的 Unicode 字符来初始化它。`__new__()` 中所用的 Unicode 字符其模样与对应的棋子相同。总共有 14 个这样的子类，它们都非常相似，只是类名与所含字符串不同，最好能把这些近乎重复的代码清理掉。

```
def populate_board(self):
```

```

for x in range(0, 9, 2):
    for y in range(4):
        column = x + ((y + 1) % 2)
        for row, color in ((y, "black"), (y + 6, "white")):
            self.board[row][column] = create_piece("draught",
                color)

```

上面列出了从 gameboard2.py 中节选的新版 CheckersBoard.populate\_board() 方法，这次就可以把它叫做工厂方法了，因为此方法会用名为 create\_piece() 的“工厂函数”（factory function）来创建棋子，而不像以前那样直接用硬编码的棋子名称来创建。create\_piece() 函数会根据其参数返回适当类型的对象（比方说，如果 color 是 "black"，那就创建 BlackDraught 对象；如果 color 是 "white"，则创建 WhiteDraught 对象）。新版代码中的 ChessBoard.populate\_board() 方法（并未列在上述程序清单中）与之类似，也会调用其 create\_piece() 函数，根据棋子颜色及名称来创建相应的对象。

```

def create_piece(kind, color):
    if kind == "draught":
        return eval("{}{}{}".format(color.title(), kind.title()))
    return eval("{}{}{}{}".format(color.title(), kind.title()))

```

工厂函数使用 Python 语言内置的 eval() 函数来创建对应类的实例。比方说，如果参数是 "knight" 与 "black"，那么交由 eval() 函数执行的字符串就是 "BlackChessKnight()"。虽说这样做完全可行，但可能会有风险，因为任何字符串都会当成 Python 代码交给 eval() 函数执行。稍后我们将换用另一种办法，以 Python 语言内置的 type() 函数来创建实例。

```

for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    exec("""
class {}(Piece):
    __slots__ = ()
    def __new__(Class):
        return super().__new__(Class, "{}")""".format(name, char))

```

这次不需要再把 14 个相似的类逐个写出来了，而是以一段代码块为模板，将其全都创建好。

调用 itertools.chain() 函数时，可传入一个或多个 iterable（可迭代物，可遍历物），此函数将返回另外一个 iterable，在返回那个的 iterable 上面遍历时，会先遍历刚才调用时传入的首个 iterable，然后再遍历刚才调用时传入的第二个 iterable，依此类推。本例中，我们给函数传了两个 iterable，第一个 iterable 是个二元组（2-tuple），其中的两个值分别是黑

色与白色跳棋棋子的“Unicode 码位”(Unicode code point)，而第二个 iterable 则是个 range 对象(实际上就是个生成器)，用于指定各种黑色与白色的国际象棋棋子。

对于每个码位来说，我们都创建一个仅包含该字符的字符串(比如“♞”)，并根据其“Unicode 名称”(Unicode name)来确定类名(例如，黑色“马”的 Unicode 名称是“black chess knight”，所以创建出来的类就叫做 BlackChessKnight)。确定了字符与类名之后，就可以用 exec() 来创建所需的类了。原来那版程序需要用 100 多行代码来逐个创建这些类，而现在只用十几行就够了。

但是，用 exec() 所带来的风险比用 eval() 还要高，所以必须得找个更好的办法才行。

```
DRAUGHT, PAWN, ROOK, KNIGHT, BISHOP, KING, QUEEN = ("DRAUGHT", "PAWN",
    "ROOK", "KNIGHT", "BISHOP", "KING", "QUEEN")

class CheckersBoard(AbstractBoard):
    ...

    def populate_board(self):
        for x in range(0, 9, 2):
            for y in range(4):
                column = x + ((y + 1) % 2)
                for row, color in ((y, BLACK), (y + 6, WHITE)):
                    self.board[row][column] = self.create_piece(DRAUGHT,
                        color)
```

上面这个 CheckersBoard.populate\_board() 方法是从 gameboard3.py 中节选的。与前一版相比，这个版本的棋子与颜色用的都是常量，而不是“字符串字面值”(string literal)，因为那样很容易打错字，而且这一版采用新的 create\_piece() 方法来创建棋子。

gameboard4.py 程序将“列表推导”(list comprehension)技术与两个 itertools 函数结合起来，用另一种办法实现了 CheckersBoard.populate\_board() 函数(此函数没有列在上述程序清单里)。

```
class AbstractBoard:
    __classForPiece = {(DRAUGHT, BLACK): BlackDraught,
        (PAWN, BLACK): BlackChessPawn,
        ...
        (QUEEN, WHITE): WhiteChessQueen}

    ...

    def create_piece(self, kind, color):
        return AbstractBoard.__classForPiece[kind, color]()
```

在这一版程序(也就是 gameboard3.py)中，create\_piece() 工厂函数是 AbstractBoard 类的方法，CheckersBoard 与 ChessBoard 类都会继承它。该方法接受两个常量做参数，根据棋子种类及颜色在静态的(也就是类级别的)字典中找到对应的类，这个字典的键是 (piece kind, color) 二元组，值是“类对象”(class object)。找到值(也就是所需的类)之后，立即用()操作符将其实例化，并返回创建好的棋子对象。

字典中的类本来可以像 gameboard1.py 那样直接写成硬代码，或是像 gameboard2.py

那样用不太安全的办法动态创建出来，但在 gameboard3.py 中，我们要采用一种较为安全的办法来做：这次仍然是动态创建，只是不再使用 eval() 与 exec() 了。

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!
```

上面这段代码的结构与早前动态创建 14 个子类的那段代码基本相同，只是这次没使用 eval() 与 exec()，而是改用了一种较为安全的办法。

知道了与棋子相对应的字符及类名之后，就可以用自定义的 make\_new\_method() 函数来创建 new() 函数了。创建好 new() 函数后，再用 Python 内置的 type() 函数创建新的类。以这种方式创建类的时候，必须传入类型名称、含有基类名称的元组（在本例中，只有一个基类，就是 Piece）以及含有类属性的字典。在字典中，我们将 \_\_slots\_\_ 属性设为空元组（这样的话，在类的实例中就不会出现私有的 \_\_dict\_\_ 了），并将 \_\_new\_\_ “方法属性”（method attribute）设置成刚才创建好的 new() 函数。

最后，调用内置的 setattr() 函数，把新创建的类（用 Class 变量表示）当作属性（属性名用 name 变量表示，比方说，在创建白色的“兵”时，name 变量的值就是 "WhiteChessPawn"）添加到当前模块（sys.modules[\_\_name\_\_]）中。gameboard4.py 用更为简洁的方式改写了上述程序清单中的最后一行代码：

```
globals()[name] = Class
```

上面这种写法的意思是：在存放全局变量的 dict 里添加元素，新元素的键是 name，值是刚才创建的 Class。这种写法的效果与 gameboard3.py 中的 setattr() 那行语句完全一样。

```
def make_new_method(char): # Needed to create a fresh method each time
    def new(Class): # Can't use super() or super(Piece, Class)
        return Piece.__new__(Class, char)
    return new
```

上面这个函数是用来创建 new() 函数的（而创建好的函数将成为类的 \_\_new\_\_() 方法）。在创建 new() 函数时不能调用 super()，因为此处并没有 super() 函数所需的“类环境”<sup>①</sup>。请注意，尽管 Piece 类没有 \_\_new\_\_() 方法，但其基类 str 有，所以 make\_new\_method() 函数所调用的 Piece.\_\_new\_\_() 实际上指的是 str.\_\_new\_\_()。

前面代码中的 new = make\_new\_method(char) 语句以及 make\_new\_method() 函数其实都可以删掉，把原来调用 make\_new\_method() 函数的代码改成下面这两行语句就好：

---

<sup>①</sup> class context，“context”一词也译为“上下文”。——译者注

```
new = (lambda char: lambda Class: Piece.__new__(Class, char))(char)
new.__name__ = "__new__"
```

上面这段代码先写了 lambda 表达式，然后立刻用 char 来填充外围的 lambda，以此创建出 new() 函数。(gameboard4.py 用的就是这种写法。)

所有的 lambda 函数都叫做 "lambda"，这在调试的时候不易区分，所以创建好 new() 函数之后，我们又给它起了个新名字。

```
def populate_board(self):
    for row, color in ((0, BLACK), (7, WHITE)):
        for columns, kind in (((0, 7), ROOK), ((1, 6), KNIGHT),
                              ((2, 5), BISHOP), ((3,), QUEEN), ((4,), KING)):
            for column in columns:
                self.board[row][column] = self.create_piece(kind,
                                                              color)
    for column in range(8):
        for row, color in ((1, BLACK), (6, WHITE)):
            self.board[row][column] = self.create_piece(PAWN, color)
```

为了使范例代码完整一些，笔者在上面列出了 ChessBoard.populate\_board() 方法的代码，gameboard3.py 及 gameboard4.py 都使用此方法。它用棋子颜色及棋子类型常量来生成棋盘（也可以不写成硬代码，而是从文件中读入，或令用户通过菜单来选择）。gameboard3.py 使用的是早前列出的那个 create\_piece() 工厂函数，而 gameboard4.py 所使用的 create\_piece() 则是最终版。

```
def create_piece(kind, color):
    color = "White" if color == WHITE else "Black"
    name = {DRAUGHT: "Draught", PAWN: "ChessPawn", ROOK: "ChessRook",
            KNIGHT: "ChessKnight", BISHOP: "ChessBishop",
            KING: "ChessKing", QUEEN: "ChessQueen"}[kind]
    return globals()[color + name]()
```

上面是 gameboard4.py 的 create\_piece() 工厂函数，其所用的常量与 gameboard3.py 相同，但它并没有专门把类对象保存到字典中，而是调用内置的 globals() 函数，在返回的全局变量字典里查出所需的类对象，立刻将其实例化，并返回创建好的棋子对象。

## 1.4 原型模式

如果想根据现有对象复制出新的对象并对其进行修改，那么可以考虑“原型模式”(Prototype Pattern)。

在前面，尤其是前一节里，大家已经看到，Python 语言提供了多种创建新对象的方式，只要在运行期能够确定其类型就可以，即便只知道类型的名字，我们也能创建出实例来。

```
class Point:
```

```
__slots__ = ("x", "y")
def __init__(self, x, y):
    self.x = x
    self.y = y
```

上面这个 Point 类经常出现在各种范例代码中，而在 Python 语言里，下面 7 种办法都可以创建出新的 Point 对象：

```
def make_object(Class, *args, **kwargs):
    return Class(*args, **kwargs)

point1 = Point(1, 2)
point2 = eval("{}({}, {})".format("Point", 2, 4)) # Risky
point3 = getattr(sys.modules[__name__], "Point")(3, 6)
point4 = globals()["Point"](4, 8)
point5 = make_object(Point, 5, 10)
point6 = copy.deepcopy(point5)
point6.x = 6
point6.y = 12
point7 = point1.__class__(7, 14) # Could have used any of point1 to point6
```

point1 是按照传统方式（也就是静态方式）创建的，我们把 Point 类对象当成构造器<sup>⊖</sup>使用。其他 point 对象则是动态创建出来的，其中，在创建 point2、point3 和 point4 时，我们把类名当成参数传给相关函数。由于创建 point3 与 point4 时所用的方法都很简洁，所以我们没有必要再像创建 point2 时那样使用有安全隐患的 eval() 函数了。point4 的创建原理与 point3 完全相同，我们调用了 Python 语言内置的 globals() 函数，这样写出来的代码更为优雅。point5 是由通用的 make\_object() 函数创建出来的，我们在调用此函数时，传入了类对象和相关参数。point6 采用经典的“原型”方式创建：首先根据现有对象复制出新对象，然后在新对象上执行初始化或配置操作。point7 是用 point1 的类对象创建出来的，创建时传入了新的参数。

由 point6 的创建过程可知，我们能够通过 Python 语言内置的 copy.deepcopy() 函数以“原型法”（prototyping）来创建新对象。而 point7 则告诉大家，这项任务在 Python 语言中还有更优雅的实现方式：无须先克隆现有对象，然后再修改新对象，而是可以直接用新参数来创建新对象，这样做效率会高很多。

## 1.5 单例模式

在整个程序运行过程中，如果某个类只应该有一个实例，那么可通过单例模式来保证。

有些面向对象编程语言很难创建单例，但在 Python 语言中却非常简单。Python Cookbook ([code.activestate.com/recipes/langs/python/](http://code.activestate.com/recipes/langs/python/)) 提供了非常易用的 Singleton 类，只

---

⊖ 严格来说，\_\_init\_\_() 方法是“初始化器”（initializer），\_\_new\_\_() 方法是“构造器”（constructor）。然而在开发时一般都使用 \_\_init\_\_(), 很少会用到 \_\_new\_\_(), 所以本书把两者都称为“构造器”。

要继承它，就会成为单例。此外还提供了 Borg 类，可以用另一种方式实现单例效果。

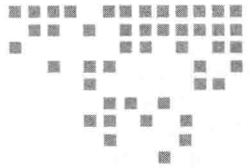
然而在 Python 中实现单例最为简单的办法是：创建模块时，把全局状态放在私有变量中，并提供用于访问此变量的公开函数。比方说，在第 7 章的 currency 范例中，我们要创建一个函数，令其返回含有货币汇率的字典（该字典以货币名称为键，以汇率为值）。这个函数可能会调用很多次，但大部分情况下，汇率数据只获取一次就够了，无须每次调用时都获取一遍。此需求可以通过单例模式来实现。

```
_URL = "http://www.bankofcanada.ca/stats/assets/csv/fx-seven-day.csv"

def get(refresh=False):
    if refresh:
        get.rates = {}
    if get.rates:
        return get.rates
    with urllib.request.urlopen(_URL) as file:
        for line in file:
            line = line.rstrip().decode("utf-8")
            if not line or line.startswith(("#", "Date")):
                continue
            name, currency, *rest = re.split(r"\s*,\s*", line)
            key = "{} ({})".format(name, currency)
            try:
                get.rates[key] = float(rest[-1])
            except ValueError as err:
                print("error {}: {}".format(err, line))
    return get.rates
get.rates = {}
```

这段代码节选自 currency/Rates.py 模块（与往常一样，节选代码时也略去了 import 语句）。我们创建了名为 rates 的字典，用于保存私有数据，并将该字典设置成 Rates.get() 函数的属性。第一次执行公开的 get() 函数时（或者以 refresh=True 为参数调用时），会下载全新的汇率数据；其他时候只需把最近下载的那份数据返回就行了。尽管没有引入类，但我们依然把汇率数据做成了“单例数据值”（singleton data value），若要添加其他单例值，亦可仿照此法。

创建型设计模式在 Python 语言中都很容易实现。单例模式可以直接用模块来实现，而原型模式则显得多余，尽管也可以通过 copy 模块来实现，但 Python 语言能够动态访问类对象，所以没必要那么做。最有用的创建型设计模式是抽象工厂模式、工厂方法模式与建造者模式，它们的实现方式有很多种。创建好基本的对象之后，一般需要通过组合或适配来创建更为复杂的对象。下一章就来谈谈这个问题。



## 第 2 章

Chapter 2

# Python 的结构型设计模式

结构型设计模式的主要用途是将一种对象改装为另一种对象，或将小对象拼合成大对象。结构型设计模式有三个主题：适配接口（adapt interface）、增加功能（add functionality）和处理对象群集（handle collections of object）。

## 2.1 适配器模式

“适配器模式”（Adapter Pattern）是一种接口适配技术，可通过某个类来使用另一个接口与之不兼容的类，运用此模式时，两个类的接口都无须改动。这项技术非常有用，比方说，我们想把某个类从其原先的应用场景中拿出来放在另一个环境下运行，而这个类又不能修改，那就应该考虑适配器模式。

假设有个简单的 Page 类用于渲染页面，它需要知道标题、正文段落以及“渲染器类”（renderer class）的实例。（本节代码均选自 `renderer1.py` 范例程序。）

```
class Page:  
    def __init__(self, title, renderer):  
        if not isinstance(renderer, Renderer):  
            raise TypeError("Expected object of type Renderer, got {}".format(type(renderer).__name__))  
        self.title = title  
        self.renderer = renderer  
        self.paragraphs = []  
  
    def add_paragraph(self, paragraph):
```

```

    self.paragraphs.append(paragraph)

def render(self):
    self.renderer.header(self.title)
    for paragraph in self.paragraphs:
        self.renderer.paragraph(paragraph)
    self.renderer.footer()

```

Page 类并不知道也无须关心传进来的渲染器类实例具体是什么，它只要知道渲染器提供了渲染页面所需的接口就好，也就是说，渲染器类应该有三个方法：header(str)、paragraph(str)、footer()。

在本例中，我们需要保证`__init__()`接收到的`renderer`参数确实是个`Renderer`实例。有一种简单但是很糟糕的办法，就是用`assert isinstance(renderer, Renderer)`语句来判断。这么做有两个缺陷。首先，它抛出的是`AssertionError`，而不是我们所期望的`TypeError`，后者更为具体。其次，假如运行程序时指定了`-o`选项（“optimize”，优化），那么`assert`语句就不会执行，而稍后执行`render()`方法时，将会导致`AttributeError`异常。范例代码中的`if not isinstance(...)`语句则没有这两个问题，它可以抛出`TypeError`异常，而且在加了`-o`选项后依然能正确运行。

但这种写法也有个明显的问题，那就是所有渲染器子类似乎都必须继承自`Renderer`基类。假如用 C++ 语言来编程，那确实如此，而在 Python 语言里也是可以创建这种基类的。不过，Python 的`abc`（abstract base class，抽象基类）模块提供了另一种做法，既能像抽象基类那样检查接口是否匹配，又能像“动态类型”（duck typing）那样非常灵活。这就是说，我们可以在无须继承特定基类的前提下，创建出符合某套接口（也就是具备特定 API）的对象来。

```

class Renderer(metaclass=abc.ABCMeta):

    @classmethod
    def __subclasshook__(Class, Subclass):
        if Class is Renderer:
            attributes = collections.ChainMap(*({Superclass.__dict__}
                                              for Superclass in Subclass.__mro__))
            methods = ("header", "paragraph", "footer")
            if all(method in attributes for method in methods):
                return True
        return NotImplemented

```

`Renderer`类重新实现了`__subclasshook__()`这个“特殊方法”（special method）。Python 语言内置的`isinstance()`函数要通过此方法来决定函数的首个参数是不是第二个参数的子类（如果第二个参数是由类所构成的元组，那就判断首个参数是不是元组中某个类的子类）。

上面这段代码有些棘手，它必须在 Python 3.3 及之后的版本上才能运行，因为其中用到

了 `collections.ChainMap` 类<sup>⊖</sup>。这段代码的原理稍后解释，但就算不明白也无关紧要，因为这些复杂的操作都可以通过范例代码中的 `@Qtrac.has_methods` “类装饰器”（class decorator）来完成，2.2 节将会演示其用法。

`__subclasshook__()` 特殊方法首先通过 `Class` 参数判断该自己是不是在 `Renderer` 类上面调用的，如果不是，就返回 `NotImplemented`。这么做意味着子类无法继承 `__subclasshook__()` 的行为。由于我们假定子类要在抽象基类的基础上添加新的标准而不是新的行为，所以才设计成这样。若想继承 `__subclasshook__()` 的行为也可以，只要在重新实现 `__subclasshook__()` 的时候调用 `Renderer.__subclasshook__()` 就可以了。

此方法如果返回 `True` 或 `False`，那么 `isinstance()` 的判定流程就会在这个抽象基类处终止，并返回 `bool` 值。若返回 `NotImplemented`，则会沿着继承体系按照通常的规则继续判定下去（判断 `Subclass` 是不是本类的子类、是不是“显式注册类”（explicitly registered class）的子类、是不是子类的子类）<sup>⊖</sup>。

如果满足了 `if` 语句的判断条件，那就调用 `Subclass` 的 `__mro__()` 特殊方法，并遍历 `Subclass` 及所有超类（包括 `Subclass` 本身）的私有字典（也就是 `__dict__`）。遍历好的字典会放在元组中，我们通过序列解包操作 (\*) 将其传给 `collections.ChainMap()` 函数。此函数会创建一份 Map 视图，把它从参数中收到的所有映射表（比如字典就可以当作映射表传进去）都当成一张映射表看待。接下来，将待检测的方法放在另一个元组中。最后，遍历元组中的方法，判断它们是不是都在 `attributes` 映射表中，这张映射表的键是 `Subclass` 及其全部超类的所有方法名与属性名。如果 `methods` 中的每个方法都在 `attributes` 映射表里，那就返回 `True`。

请注意，上面这段代码只检测了 `Subclass` 及其全部基类的所有 `attribute` 名称是不是涵盖了我们所需的那些方法，并没有详细判断 `attribute` 到底是属性还是方法。如果某属性恰好与所需方法同名，那它也能通过检测。假如检测时想排除属性名而只考虑方法名，那么可在 `method in attributes` 这行判断语句中加上 `and callable(method)`。由于此问题在实际编程中很少遇到，所以没必要专门改写。

用 `__subclasshook__()` 来创建带有接口检查功能的类是一项非常有用的技术，但如果每个类都要写这十几行代码的话，那就显得重复了，因为这些类之间的差别可能不大，只是基类与所支持的方法不同而已。在下一节中，我们将通过类装饰器来避免重复代码，也就是说，有了类装饰器之后，每次只需编写一两行特殊代码，就能创建出具备接口检查功能的类。（下一节的 `render2.py` 范例程序演示了这种装饰器的用法。）

⊖ `render1.py` 程序与 `render2.py` 所用的 `Qtrac.py` 模块都包含两套代码，一套用于 Python 3.3，另一套用于早前的各种 Python 3 版本。

⊖ 具体判断流程请参阅 <http://hg.python.org/cpython/file/c4a2d0538441/Lib/abc.py> 文件的 `__subclasscheck__` 函数。——译者注

```

class TextRenderer:

    def __init__(self, width=80, file=sys.stdout):
        self.width = width
        self.file = file
        self.previous = False

    def header(self, title):
        self.file.write("{0:^{2}}\n{1:^{2}}\n".format(title,
            "=" * len(title), self.width))

```

上面这个简单的类可以当成页面渲染器来用，因为它具备相关接口。

`header()` 方法会根据指定的宽度把标题输出到正中位置，然后换一行，在标题的每个字母下面输出“=”字符。

```

def paragraph(self, text):
    if self.previous:
        self.file.write("\n")
    self.file.write(textwrap.fill(text, self.width))
    self.file.write("\n")
    self.previous = True

def footer(self):
    pass

```

`paragraph()` 方法使用 Python 标准库的 `textwrap` 模块把文本段按照指定的宽度换行，并打印出来。使用 `self.previous` 这个 Boolean 变量是为了保证除第一段以外，后面每两段之间都有空行隔开。由于页面渲染器的接口定义了 `footer()`，所以即便不打印页脚，也得写个什么事都不做的方法放在这里。

```

class HtmlWriter:

    def __init__(self, file=sys.stdout):
        self.file = file

    def header(self):
        self.file.write("<!doctype html>\n<html>\n")

    def title(self, title):
        self.file.write("<head><title>{}</title></head>\n".format(
            escape(title)))

    def start_body(self):
        self.file.write("<body>\n")

    def body(self, text):
        self.file.write("<p>{}</p>\n".format(escape(text)))

    def end_body(self):
        self.file.write("</body>\n")

    def footer(self):
        self.file.write("</html>\n")

```

`HtmlWriter`类可用来写出简单的HTML页面，它用`html.escape()`函数处理转义字符（在Python 3.2及早前版本中，使用`xml.sax.saxutil.escape()`函数）。

尽管这个类也有`header()`及`footer()`方法，但其行为却和页面渲染器接口所定义的不同。所以，在构建`Page`实例时，我们可以传入`TextRenderer`对象，但却不能直接把`HtmlWriter`对象当成页面渲染器传进去。

一种解决办法是编写`HtmlWriter`的子类，并在子类中提供页面渲染器所需的接口方法。但这种方案很容易出错，因为子类会把`HtmlWriter`的方法同页面渲染器的接口方法混在一起。还有个更好的办法就是创建适配器，令其把我们要使用的`HtmlWriter`类“聚合”（aggregate）进来，并提供`Renderer`所定义的接口，然后将聚合进来的类与接口适配好。图2.1演示了如何引入适配器类。

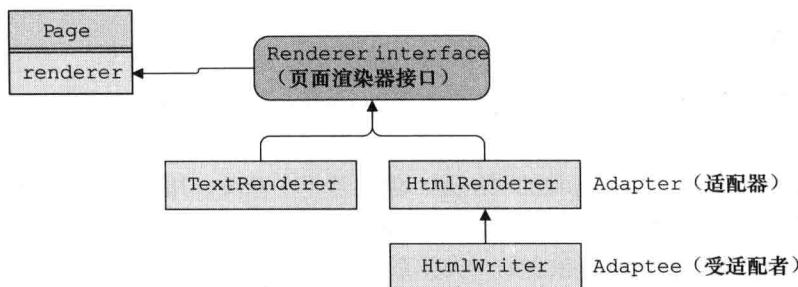


图2.1 创建适配器类，以适应页面渲染器的接口

```

class HtmlRenderer:

    def __init__(self, htmlWriter):
        self.htmlWriter = htmlWriter

    def header(self, title):
        self.htmlWriter.header()
        self.htmlWriter.title(title)
        self.htmlWriter.start_body()

    def paragraph(self, text):
        self.htmlWriter.body(text)

    def footer(self):
        self.htmlWriter.end_body()
        self.htmlWriter.footer()
  
```

上面列出的就是适配器类。在构造时，可把`HtmlWriter`对象当成`htmlWriter`参数传入，该类负责提供页面渲染器接口所需的方法。由于实际渲染任务都会委托给聚合进来的`HtmlWriter`对象，所以`HtmlRenderer`类只相当于在现有的`HtmlWriter`类的外围包了一层新的接口而已。

```

textPage = Page(title, TextRenderer(22))
textPage.add_paragraph(paragraph1)
  
```

```

textPage.add_paragraph(paragraph2)
textPage.render()

htmlPage = Page(title, HtmlRenderer(HtmlWriter(file)))
htmlPage.add_paragraph(paragraph1)
htmlPage.add_paragraph(paragraph2)
htmlPage.render()

```

上面几行代码演示了如何用两种渲染器来创建 Page 类的实例。在构建 TextRenderer 时，我们将默认宽度设为 22 个字符。而在用 HtmlRenderer 来创建 HtmlWriter 适配器时，我们则把一份打开的文件传了进去（创建此文件所用的语句没有列出来），这样的话，HTML 就不会渲染到默认的 sys.stdout 上面了。

## 2.2 桥接模式

“桥接模式”（Bridge Pattern）用于将“抽象”（abstraction，比如接口或算法）与实现方式相分离。

如果不用桥接模式，那么通常的写法是，创建若干个基类，用于表示各种抽象方式，然后从每个基类中继承出两个或多个子类，用于表示对这种抽象方式的不同实现办法。用了桥接模式之后，我们需要创建两套独立的“类体系”（class hierarchy）：“抽象体系”定义了我们所要执行的操作（比如接口或高层算法），而“实现体系”则包含具体实现方式，抽象体系要调用实现体系以完成其操作。抽象体系中的类会把实现体系中的某个类实例聚合进来，而这个实例将充当抽象接口与具体实现之间的桥梁（bridge）。

在前一节所讲的适配器模式中，HtmlRenderer 类就可称为桥接模式，因为它为了完成渲染操作，把 HtmlWriter 实例聚合进来了。

本节要编写一个类，它可以用特定算法来绘制条形图，然而我们想把具体的算法实现代码放在其他类中。barchart1.py 范例程序就是以这种方式来完成此功能的，它使用了桥接模式。

```

class BarCharter:

    def __init__(self, renderer):
        if not isinstance(renderer, BarRenderer):
            raise TypeError("Expected object of type BarRenderer, got {}".
                            format(type(renderer).__name__))
        self.__renderer = renderer

    def render(self, caption, pairs):
        maximum = max(value for _, value in pairs)
        self.__renderer.initialize(len(pairs), maximum)
        self.__renderer.draw_caption(caption)
        for name, value in pairs:
            self.__renderer.draw_bar(name, value)
        self.__renderer.finalize()

```

BarCharter 类在其 render() 方法中描述了条形图绘制算法，而该算法要通过符合条形图渲染接口的渲染器来实现。这个渲染器需要具备接口中定义的四个方法： initialize(int, int)、 draw\_caption(str)、 draw\_bar(str, int)、 finalize()。

与前一节类似，我们也通过 isinstance() 来判断传入的 renderer 对象是否具备所需的接口，这样就无须强迫条形图渲染器必须实现某个特定的基类了。但这次我们不像上一节那样用十行代码编写一个类，而是只用两行代码来创建专门用于检查接口的类。

```
@Qtrac.has_methods("initialize", "draw_caption", "draw_bar", "finalize")
class BarRenderer(metaclass=abc.ABCMeta): pass
```

上面这段代码先创建了名为 BarRenderer 的类，并设置好使用 abc 模块所需的 metaclass 属性。然后，这个类会传给 Qtrac.has\_methods() 函数，该函数将返回一个“类修饰器”。修饰器会向受修饰的类里添加 \_\_subclasshook\_\_() 类方法。当我们调用 isinstance() 来检测某个实例是不是 BarRenderer 类型时，这个新添加的方法会判断实例对应的类是否具备所需的方法。（如果不熟悉类修饰器，那么请先阅读 2.4 节，尤其是 2.4.2 节，然后再回到这里。）

```
def has_methods(*methods):
    def decorator(Base):
        def __subclasshook__(Class, Subclass):
            if Class is Base:
                attributes = collections.ChainMap(*{Superclass.__dict__
                    for Superclass in Subclass.__mro__})
                if all(method in attributes for method in methods):
                    return True
                return NotImplemented
            Base.__subclasshook__ = classmethod(__subclasshook__)
        return Base
    return decorator
```

Qtrac.py 模块的 has\_methods() 函数通过 methods 参数来捕获用户所需的方法，然后创建类修饰器函数，并将其返回。修饰器函数本身又创建了 \_\_subclasshook\_\_() 函数，并用 Python 内置的 classmethod() 函数将其设置成基类的类方法。这个 \_\_subclasshook\_\_() 函数的代码与上一节所列的基本相同，区别在于这次没有把基类写成硬代码，而是通过 Base 参数来表示那个受修饰的类，另外，所需检测的方法名也没写成硬代码，而是用 has\_methods() 函数的 methods 参数来表示的。

要实现接口检测功能，还有个办法，就是继承通用的抽象基类。例如：

```
class BarRenderer(Qtrac.Requirer):
    required_methods = {"initialize", "draw_caption", "draw_bar",
                       "finalize"}
```

上面这段代码节选自 barchart3.py。Qtrac.Requirer 类（没有列出来，但是在 Qtrac.py 文件里）是个抽象基类，它能像 @has\_methods 类修饰器那样检测相关的类是否实现了所需的接口。

```

def main():
    pairs = (("Mon", 16), ("Tue", 17), ("Wed", 19), ("Thu", 22),
              ("Fri", 24), ("Sat", 21), ("Sun", 19))
    textBarCharter = BarCharter(TextBarRenderer())
    textBarCharter.render("Forecast 6/8", pairs)
    imageBarCharter = BarCharter(ImageBarRenderer())
    imageBarCharter.render("Forecast 6/8", pairs)

```

main() 函数设置了一些数据，然后创建了两张条形图，用两种不同的渲染方式分别渲染这些数据。由程序所输出的这两张条形图列在图 2.2 里，接口与类的关系如图 2.3 所示。

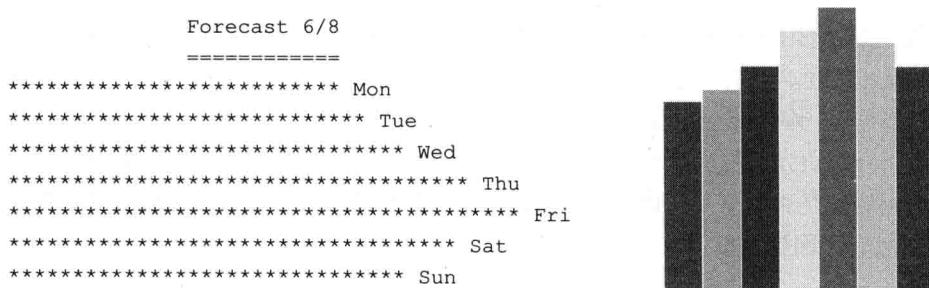


图 2.2 文本形式的条形图与图画形式的条形图

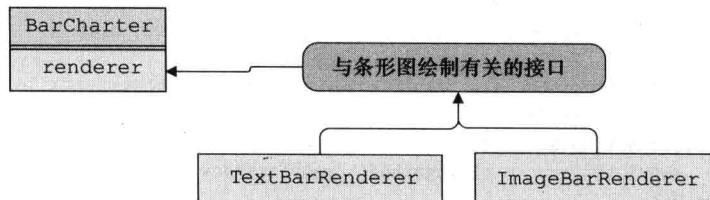


图 2.3 与条形图绘制有关的接口和类

```

class TextBarRenderer:
    def __init__(self, scaleFactor=40):
        self.scaleFactor = scaleFactor

    def initialize(self, bars, maximum):
        assert bars > 0 and maximum > 0
        self.scale = self.scaleFactor / maximum

    def draw_caption(self, caption):
        print("{0:^{2}}\n{1:^{2}}".format(caption, "=" * len(caption),
                                         self.scaleFactor))

    def draw_bar(self, name, value):
        print("{} {}".format("*" * int(value * self.scale), name))

    def finalize(self):
        pass

```

上面这个类实现了条形图绘制接口，并把表示条形图的文本输出到 `sys.stdout`。若想令用户能够配置输出文件，只需对这段代码稍加修改即可，而且在类 Unix 系统中，还可以拿“绘制方框专用的 Unicode 字符”（Unicode box drawing character）来画图，并辅以各种颜色，使输出更加美观。

请注意，尽管 `TextBarRenderer` 的 `finalize()` 什么事都没做，但还是必须写上，因为条形图渲染器的接口里定义了这个方法。

虽说 Python 的标准库涵盖面非常广（这种设计思路叫做“batteries included”<sup>①</sup>），但奇怪的是，这里面似乎缺了一项重要的功能——找不到专门读写标准位图与矢量图的包。一种办法是调用第三方库，比如，可以使用支持多种图片格式的 Pillow 库（网址是：[github.com/python-imaging/Pillow](https://github.com/python-imaging/Pillow)），也可以使用专门处理某种图片格式的库，甚至还可以用“GUI 工具包库”（GUI toolkit library）来做。另一种办法是自己创建图形处理库，我们将在 3.12 节中讲解这种办法。若只需处理 GIF 格式的图像，则可使用 Tkinter<sup>②</sup>（如果将来随同 Python 发布的 Tk/Tcl 是 8.6 版，那么还支持 PNG 格式）。

在 `barchart1.py` 文件中，`ImageBarRenderer` 类采用 `cyImage` 模块来处理图像，若无法使用此模块，则使用 `Image` 模块。由于两模块之间差别不大，所以我们把二者统称为“`Image` 模块”。本书的范例代码中有这两个模块的源代码，而且稍后还会讲解它们（3.12 节讲解 `Image` 模块，5.3 节讲解 `cyImage` 模块）。为了使程序完整，范例代码里还含有 `barchart2.py` 文件，这个版本与 `barchart1.py` 的区别在于，它没使用 `cyImage` 或 `Image`，而是用 Tkinter 来处理图像的，本书不会列出此版本的代码。

由于 `ImageBarRenderer` 比 `TextBarRenderer` 复杂得多，所以我们把该类的静态数据与方法分开，逐个讲解。

```
class ImageBarRenderer:
    COLORS = [Image.color_for_name(name) for name in ("red", "green",
                                                       "blue", "yellow", "magenta", "cyan")]
```

`Image` 模块用 32 位无符号整数来表示像素颜色，颜色值由 `alpha`（透明度）、`red`（红）、`green`（绿）、`blue`（蓝）这四个分量组成。模块里有个 `Image.color_for_name()` 函数，可根据颜色名称返回与之对应的无符号整数，这个名称既可以是 X11 的 `rgb.txt` 文件<sup>③</sup> 中所列的名称（比如 "sienna"），也可以是 HTML 风格的名称（比如 "#A0522D"）。

上面这段代码定义了条形图中各个条块的颜色。

- ① 大意是，标准库中的各个模块其功能都非常强大，而且也很可靠。开发者可通过这些模块，以非常简单的方式实现许多复杂的功能。——译者注
- ② 注意，Tkinter 要求开发者必须在主线程（也就是 GUI 线程）中处理图像。若要实现并发图像处理，则需使用另一种办法，我们将在 4.1 节中讲述那种办法。
- ③ 在 Linux 操作系统中，该文件位于 `/etc/X11/rgb.txt`。——译者注

```
def __init__(self, stepHeight=10, barWidth=30, barGap=2):
    self.stepHeight = stepHeight
    self.barWidth = barWidth
    self.barGap = barGap
```

用户可通过 `__init__()` 方法的参数来调整条形图中条块的样貌。

```
def initialize(self, bars, maximum):
    assert bars > 0 and maximum > 0
    self.index = 0
    color = Image.color_for_name("white")
    self.image = Image.Image(bars * (self.barWidth + self.barGap),
                           maximum * self.stepHeight, background=color)
```

由于条形图渲染器的接口里定义了 `initialize()`，所以 `ImageBarRenderer` 类必须要有上面这个方法才行（同时还必须有 `draw_caption()`、`draw_bar()`、`finalize()` 方法，它们的代码将在后面列出）。该方法新建了一张图像，其宽度同条块数量与条块宽度之积成正比，其高度与条形图有可能出现的最大高度成正比，其初始颜色是白色。

`self.index` 变量用于记录当前应该绘制条形图里的第几个条块（从 0 开始算）。

```
def draw_caption(self, caption):
    self.filename = os.path.join(tempfile.gettempdir(),
                               re.sub(r"\W+", "_", caption) + ".xpm")
```

由于 `Image` 模块没有文本绘制功能，所以无法直接输出 `caption` 参数，我们换一种办法使用此参数：以它为基础来确定图像的文件名。

`Image` 模块本身支持两种图像格式，一种是 `XBM`（文件扩展名为 `.xbm`），用于表示黑白图像，另一种是 `XPM`（文件扩展名为 `.xpm`），用于表示彩色图像。如果安装了 `PyPNG` 模块（该模块详情参见：[pypi.python.org/pypi/pypng](http://pypi.python.org/pypi/pypng)），那么 `Image` 模块还能支持 `PNG` 格式的图像（后缀名为 `.png`）。此处我们选用 `XPM` 格式，因为条形图是彩色的，而且这种格式广受支持。

```
def draw_bar(self, name, value):
    color = ImageBarRenderer.COLORS[self.index %
                                    len(ImageBarRenderer.COLORS)]
    width, height = self.image.size
    x0 = self.index * (self.barWidth + self.barGap)
    x1 = x0 + self.barWidth
    y0 = height - (value * self.stepHeight)
    y1 = height - 1
    self.image.rectangle(x0, y0, x1, y1, fill=color)
    self.index += 1
```

上述方法首先从 `COLORS` 序列中选择与当前条块相匹配的颜色（如果条块数量比颜色数还多，那就用当前条块的序号除以颜色总数，根据余数来确定颜色）。然后计算当前条块（也就是 `self.index`）左上角及右下角的坐标，并调用 `self.image` 实例（此实例的类型是 `Image.Image`）的 `rectangle()` 方法，根据坐标及填充色来绘制矩形。最后递增

`index`, 为绘制下个条块做准备。

```
def finalize(self):
    self.image.save(self.filename)
    print("wrote", self.filename)
```

上面这个 `finalize()` 方法很简单, 保存图像并将其文件名告诉用户。

尽管 `TextBarRenderer` 与 `ImageBarRenderer` 的实现方式差别很大, 但它们均可作为抽象体系与实现体系之间的桥梁。`BarCharter` 类可以分别用两者所提供的具体实现代码来绘制条形图。

## 2.3 组合模式

“组合模式” (Composite Pattern) 可用来统合类体系中的两种对象: 一种对象能够包含体系中的其他对象, 另一种不能。前者叫做“组合体” (composite), 后者叫做“非组合体” (noncomposite), 两者统称“组件” (component)。按照传统的实现方式, 这两种组件 (一种是单个对象, 一种是对象群集) 所对应的类都继承自同一个基类。组合体与非组合体对象都具备同一套“核心方法” (core method), 此外, 组合体对象还有用于增加、移除、遍历子对象的其他方法。

该模式常用于实现 Inkscape 等绘图程序, 这种程序需要有“群组” (group) 与“解除群组” (ungroup) 功能。用户可选取一批组件, 并对其执行群组或解除群组操作, 而这些组件中, 有的是单个元素 (比如矩形), 有的是组合体 (比如由各种图形所构成的脸谱)。

现在就来看个实际的例子。我们在 `main()` 函数里创建一些对象, 有单个元素, 也有组合体, 然后, 把它们全都打印出来。下面这段代码选自 `stationery1.py`, 代码后面是程序所输出的信息。

```
def main():
    pencil = SimpleItem("Pencil", 0.40)
    ruler = SimpleItem("Ruler", 1.60)
    eraser = SimpleItem("Eraser", 0.20)
    pencilSet = CompositeItem("Pencil Set", pencil, ruler, eraser)
    box = SimpleItem("Box", 1.00)
    boxedPencilSet = CompositeItem("Boxed Pencil Set", box, pencilSet)
    boxedPencilSet.add(pencil)
    for item in (pencil, ruler, eraser, pencilSet, boxedPencilSet):
        item.print()

$0.40 Pencil
$1.60 Ruler
$0.20 Eraser
$2.20 Pencil Set
    $0.40 Pencil
    $1.60 Ruler
    $0.20 Eraser
```

```
$3.60 Boxed Pencil Set
$1.00 Box
$2.20 Pencil Set
$0.40 Pencil
$1.60 Ruler
$0.20 Eraser
$0.40 Pencil
```

每个 SimpleItem 对象都有名称及价格，CompositeItem 对象也有名称，而且可以包含任意数量的 SimpleItem 或 CompositeItem，也就是说，组合体可以无限嵌套。组合体的价格是其全部元素的价格之和。

在本例中，“铅笔套件”（pencil set）包含一只“铅笔”（pencil）、一把“尺子”（ruler）、一块“橡皮”（eraser）。而“盒装铅笔套件”（boxed pencil set）则包含“文具盒”（box）、铅笔套件及另一只铅笔。图 2.4 演示了盒装铅笔套件与其元素之间的关系。

接下来，我们要以两种方法实现组合模式，第一种是传统做法，第二种是用一个类来表示组合体与非组合体。

### 2.3.1 常规的“组合体 / 非组合体”式层级

在常规的实现方式中，所有组件（无论是组合体还是非组合体）都具有相同的抽象基类 AbstractItem，而且组合体要直接继承自另外一个抽象基类 AbstractCompositeItem。整个类体系如图 2.5 所示。我们先看 AbstractItem 这个基类。

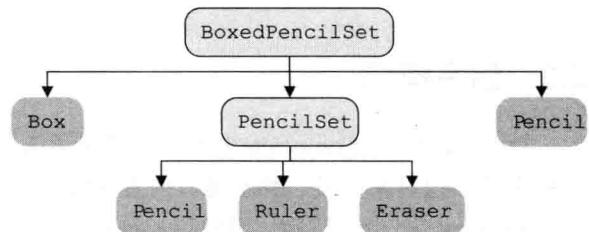


图 2.4 由组合体元素与非组合体元素所构成的层次结构

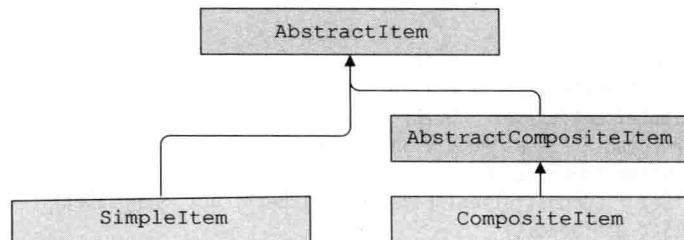


图 2.5 由组合体及非组合体所构成的类体系

```
class AbstractItem(metaclass=abc.ABCMeta):
    @abc.abstractproperty
    def composite(self):
        pass

    def __iter__(self):
        return iter([])
```

我们要求所有子类的对象都能向用户汇报自己是不是组合体，同时还要求子类对象必须可以迭代，`__iter__()`方法的默认行为是返回一个“迭代器”（iterator），该迭代器会在空序列上迭代。

由于`AbstractItem`类至少有一个抽象方法或抽象属性，所以我们无法创建此类的对象。（从Python 3.3版本开始，也可以把`@abstractproperty def method(...): ...`写成`@property @abstractmethod def method(...): ...`。）

```
class SimpleItem(AbstractItem):
    def __init__(self, name, price=0.00):
        self.name = name
        self.price = price

    @property
    def composite(self):
        return False
```

`SimpleItem`类用来表示非组合体。在本例中，每个`SimpleItem`对象都有`name`及`price`属性。

由于`SimpleItem`继承了`AbstractItem`，所以它必须重新实现基类的全部抽象属性及抽象方法，具体到本例，也就是要实现`composite`属性。由于`AbstractItem`类的`__iter__()`方法不是抽象的，所以无须重新实现，基类的代码会返回指向空序列的迭代器，子类沿用这个实现即可。这样做是合理的，因为`SimpleItem`对象不是组合体，所以返回空迭代器可以令我们把`SimpleItem`与`CompositeItem`对象统合起来（至少在迭代时是如此）。比方说，可以把由这两种对象混合而成的对象交给`itertools.chain()`来迭代。

```
def print(self, indent="", file=sys.stdout):
    print("{}${:.2f} {}".format(indent, self.price, self.name),
          file=file)
```

为了便于打印信息，我们给组合体及非组合体都定义了`print()`方法，打印时，缩进宽度会随着嵌套深度而加大。

```
class AbstractCompositeItem(AbstractItem):
    def __init__(self, *items):
        self.children = []
        if items:
            self.add(*items)
```

上面这个类是`CompositeItem`的基类，它实现了组合体所需的添加、移除和迭代等功能。由于该类从`AbstractItem`中继承了抽象的`composite`属性但却没提供实现，所以无法实例化。

```
def add(self, first, *items):
    self.children.append(first)
    if items:
        self.children.extend(items)
```

上述方法接受若干个 item (既可以是 SimpleItem, 也可以是 CompositeItem), 并将其加入本组合体的子对象列表中。编写该方法时, 不能去掉 first 参数而只保留 \*items, 因为假如那样做的话, items 所捕获的元素数量就可能为 0, 虽然无害, 但却会把用户在代码中所犯的逻辑错误掩盖掉。( \*item 的用法请参阅 1.2 节的补充知识中所讲的解包操作。) 另外, 此函数没有禁止“循环引用” (circular reference), 比方说, 用户可通过 add() 方法把某个组合体对象设置成其自身的子对象。

在下一小节中, 我们将看到另一种方法: 只需一行代码即可实现 add() 方法。

```
def remove(self, item):
    self.children.remove(item)
```

我们用了个简单的办法来实现 remove() 方法, 一次只移除一个 item。如果要移除的 item 是组合体, 那么其下各个层级中的子对象也会一并从体系里移除。

```
def __iter__(self):
    return iter(self.children)
```

实现了 \_\_iter\_\_() 这个特殊方法之后, 就可以在 for 循环、“列表推导” (comprehension) 及生成器里遍历组合体对象的子对象了。本来也可以把方法体写成 for item in self.children: yield item, 但由于 self.children 是个序列 (也就是列表), 因此直接用 Python 内置的 iter() 函数来实现更为简单。

```
class CompositeItem(AbstractCompositeItem):
    def __init__(self, name, *items):
        super().__init__(*items)
        self.name = name

    @property
    def composite(self):
        return True
```

上面这个 CompositeItem 类用来表示具体的组合体对象, 它有自己的 name 属性, 但与组合体相关的其他任务 (也就是子对象的增加、移除、迭代操作) 都交由基类处理。由于本类已经实现了抽象的 composite 属性, 而且并未留下其他尚待实现的抽象属性或抽象方法, 所以 CompositeItem 可以实例化。

```
@property
def price(self):
    return sum(item.price for item in self)
```

price 是个“只读属性” (read-only property), 其代码稍微有点难懂。这行代码构建了一条“生成器表达式” (generator expression), 并用内置的 sum() 函数来计算组合体中所有子对象的价格, 如果子对象也是组合体, 那就递归计算下去。

for item in self 表达式使得 Python 调用 iter(self) 来获取针对 self 的迭代器, 而这又会调用 \_\_iter\_\_() 特殊方法, 该方法返回指向 self.children 的迭代器。

```

def print(self, indent="", file=sys.stdout):
    print("{}${:.2f} {}".format(indent, self.price, self.name),
          file=file)
    for child in self:
        child.print(indent + "    ")

```

为了便于打印信息，本类也提供了 `print()` 方法，其首行代码与 `SimpleItem` 类的 `print()` 方法重复了。

本例中的 `SimpleItem` 与 `CompositeItem` 能够应对绝大多数情况。但若要构建更为精细的层次结构，则可以从这两个类或其抽象基类中继承专门的子类。

`AbstractItem`、`SimpleItem`、`AbstractCompositeItem`、`CompositeItem` 这四个类确实搭配得很好，但代码稍显冗长，而且接口也不统一：组合体有 `add()` 及 `remove()` 方法，非组合体却没有。下一小节我们就来解决这些问题。

### 2.3.2 只用一个类来表示组合体与非组合体

上一小节的四个类（两个抽象类，两个具体类）似乎有些多了，而且接口也没有完全统一：只有组合体才支持 `add()` 及 `remove()` 方法。如果能忍受少许额外开销的话，我们可以只用一个类来表示组合体与非组合体：给这两种对象都配备一份列表及一个 `float` 型属性，非组合体对象里的列表是空的，而组合体对象里的 `float` 型属性并不使用。此方案所设计出来的对象其行为更加合理，因为两种对象的接口完全一致：非组合体与组合体一样，也有 `add()` 及 `remove()` 方法。

本节将新建 `Item` 类，组合体与非组合体都可以用这个类表示，无须再借助其他类。本节的范例代码摘录自 `stationery2.py` 文件。

```

class Item:
    def __init__(self, name, *items, price=0.00):
        self.name = name
        self.price = price
        self.children = []
        if items:
            self.add(*items)

```

`__init__()` 方法的参数不太整齐，但是没关系，我们稍后就会看到，用户实际上无须手工调用 `Item()` 来创建对象。

每个对象都必须有名字，而且还必须有价格，构建对象的时候，若未指定价格，则会使用默认值。此外，构建对象时还可以通过 `*items` 参数放入零个或多个子对象，这些子对象将保存在 `self.children` 里面。非组合体对象的 `children` 是个空列表。

```

@classmethod
def create(Class, name, price):
    return Item(name, price=price)

@classmethod

```

```
def compose(Class, name, *items):
    return Item(name, *items)
```

上面这两个工厂方法都是类方法，它们的参数也都比 `Item.__init__()` 整齐，二者均可非常方便地创建 `Item` 对象。有了这两个方法之后，`SimpleItem("Ruler", 1.60)` 与 `CompositeItem("Pencil Set", pencil, ruler, eraser)` 可分别改写为 `Item.create("Ruler", 1.60)` 及 `Item.compose("Pencil Set", pencil, ruler, eraser)`。而且上一小节的四个类现在都合并成 `Item` 类型了。当然，用户如果愿意，也可以直接用 `Item()` 来创建对象，比如：`Item("Ruler", price=1.60)`、`Item("Pencil Set", pencil, ruler, eraser)`。

```
def make_item(name, price):
    return Item(name, price=price)

def make_composite(name, *items):
    return Item(name, *items)
```

我们还提供了上面这两个工厂函数，其作用与刚才提到的那两个工厂方法相同。在使用模块时，这种工厂函数更为便利。例如，如果 `Item` 类在 `Item.py` 模块中，那么有了这两个工厂函数之后，我们就不用再写 `Item.Item.create("Ruler", 1.60)` 了，而是可以写成 `Item.make_item("Ruler", 1.60)`。

```
@property
def composite(self):
    return bool(self.children)
```

`composite` 属性的实现方式与原来不同，因为有些 `Item` 对象是组合体，有些则不是。如果 `Item` 的 `self.children` 列表非空，那么我们就认定此对象是组合体。

```
def add(self, first, *items):
    self.children.extend(itertools.chain((first,), items))
```

`add()` 方法的实现代码与上一小节稍有不同，这次用的办法应该会更加高效一些。`itertools.chain()` 函数接受若干个 `iterable`，并返回一个 `iterable`，在返回的 `iterable` 上面迭代，其效果就等于依次在参数里的各个 `iterable` 上面迭代。

无论对象是不是组合体，都可以在它上面调用 `add()` 方法。若在非组合体上调用 `add()` 方法，则会令其变为组合体。

把非组合体变为组合体时，会产生一个小问题：由于 `price` 属性现在表示所有子对象的总价格，所以该对象本身的价格反而看不到了。若想保留自身价格，当然也有其他办法可循。

```
def remove(self, item):
    self.children.remove(item)
```

如果把组合体最后一个元素移除，那么它就变成了非组合体。这样做的效果是：本对象的价格不再是其所有子对象的价格总和了（这些子对象现在没有了），而会等于其私有的 `self.__price` 属性。为了确保相关逻辑正确，我们在 `__init__()` 方法里为所有对象都

设置了初始价格。

```
def __iter__(self):
    return iter(self.children)
```

在组合体上调用 `__iter__()` 方法会返回其子对象列表，在非组合体上调用，会返回空序列。

```
@property
def price(self):
    return (sum(item.price for item in self) if self.children else
            self.__price)

@property.setter
def price(self, price):
    self.__price = price
```

`price` 属性必须同时适用于组合体及非组合体。对于前者来说，它表示其子对象的价格总和，对于后者来说，它表示本对象的价格。

```
def print(self, indent="", file=sys.stdout):
    print("{}${:.2f} {}".format(indent, self.price, self.name),
          file=file)
    for child in self:
        child.print(indent + "    ")
```

上面这个方法和 `price` 属性一样，也必须对组合体及非组合体都适用才行，此方法的代码与上一小节的 `CompositeItem.print()` 方法相同。如果在非组合体上执行 `print()` 方法，那么执行到 `for` 语句时，该对象就会返回指向空序列的迭代器，这样的话，遍历时就不用担心“无限递归”(infinite recursion) 问题了。

由于 Python 语言很灵活，所以用它来创建组合体与非组合体是件很简单的事：想缩减存储开销时，可以分别建立两个类，而若要提供完全统一的接口，则可以合并成一个类。

3.2 节讲述“命令模式”(Command Pattern) 时，将会谈到组合模式的另一种变化形式。

## 2.4 修饰器模式

一般来说，“修饰器”(decorator) 是个单参数的函数，其参数也是函数，修饰器返回的新函数与经由参数传入的原函数名称相同，但功能更强。框架(例如 web 框架)经常通过修饰器把用户所编写的函数集成进来。

由于修饰器模式非常有用，所以 Python 提供了原生支持。在 Python 语言中，函数与方法都可以用修饰器来修饰。此外，还有“类修饰器”(class decorator)，它也是个单参数的函数，其参数是个类，由这种修饰器所返回的新类的名称与原类相同，但功能更多。有时可以通过类修饰器来实现继承。

由上一节的 `composite` 及 `price` 属性可知，Python 内置的 `property()` 函数能当作

修饰器来用。此外，Python 标准库也内置了一些修饰器。比方说，在实现了 `__eq__()` 及 `__lt__()` 特殊方法（这两个方法分别规定了“`==`”及“`<`”这两个比较操作符的含义）的类上面，可以运用 `@functools.total_ordering` 类修饰器。修饰后的类会包含其他几个与比较操作有关的特殊方法，从而能支持全套的比较操作符（也就是 `<、<=、==、!=、>=、>`）。

由于修饰器只能接受一个参数（这个参数表示待修饰的函数、方法或类），所以从理论上来讲，无法“参数化”（parameterize）<sup>①</sup>。但实际上没有这个限制，稍后我们会创建“参数化的修饰器工厂”（parameterized decorator factory），这种工厂可以返回“修饰器函数”（decorator function），而修饰器函数又可以用来修饰函数、方法或类。

## 2.4.1 函数修饰器与方法修饰器

所有的函数修饰器与方法修饰器的大体结构都相同。首先，创建“包装函数”（wrapper function，本书总是将该函数命名为 `wrapper()`），然后在包装函数里调用原函数。调用前可执行“预处理”（preprocessing），获取到结果后，还可以执行“后加工”（postprocessing）。包装函数的返回值也很灵活：可以把原函数的调用结果直接返回，也可以先修改再返回，还可以返回其他值。最后，修饰器把包装函数作为调用结果返回，返回后的函数会以原函数的名义将其取代。

修饰器以“`@`”符号开头，其缩进级别与受修饰的函数、方法、类的 `def` 或 `class` 语句相同，`@` 符号后面是修饰器的名称。多个修饰器可以叠放，也就是说，修饰过的函数还可以继续修饰，如图 2.6 所示。稍后我们会举例说明。

```
@float_args_and_return
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
```

上面这段代码用 `@float_args_and_return` 修饰器（马上就会列出其代码）来修饰 `mean()` 函数。未修饰的 `mean()` 函数接受两个或多个数值作参数，并返回 `float` 型的平均值。而修饰后的 `mean()` 函数（由于修饰后的函数取代了原函数，所以二者同名）则可以接受两个或多个任意类型的参数，只要这些参数都能转换为 `float` 就行。若是不修饰，那么调用 `mean(5, "6", "7.5")` 时会抛出 `TypeError`，因为 `int` 与 `str` 不能直接相加。但修饰之后就没有这个问题了，因为“`6`”和“`7.5`”这两个字符串可通过 `float("6")` 与 `float("7.5")` 转换成有效的数值。

其实修饰器就是一种“语法糖”（syntactic sugar）。刚才那段代码也可以写成：

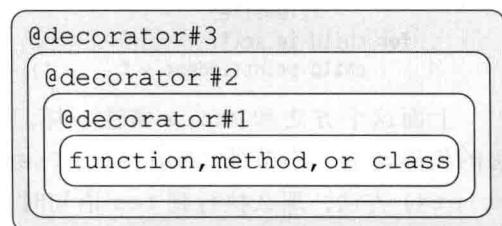


图 2.6 叠起来的多个修饰器

<sup>①</sup> “参数化”一词在此处可理解为“将其制作成模板”。——译者注

```
def mean(first, second, *rest):
    numbers = (first, second) + rest
    return sum(numbers) / len(numbers)
mean = float_args_and_return(mean)
```

上面这段代码先创建了未受修饰的 `mean()` 函数，然后手工调用修饰器，用修饰后的版本替换原来的 `mean()`。虽说修饰器用起来很方便，但有时候还是必须像上面这样自己来调用。本节最后就有个例子，在 `ensure()` 函数里调用 Python 内置的 `@property` 修饰器。2.2 节的 `has_methods()` 函数也曾直接调用 Python 内置的 `@classmethod` 修饰器。

```
def float_args_and_return(function):
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

`float_args_and_return()` 函数是函数修饰器，所以只接受一个函数作其唯一的参数。为便于处理，可以把 `*args` 与 `**kwargs` 当成包装函数的参数，这样写实际上就等于令包装函数可以接受任意参数。（`*args` 与 `**kwargs` 的含义请参阅 1.2 节中的补充知识。）原函数（也就是包裹在 `wrapper()` 里的 `function()` 函数）也许对参数有一些限制，而包装函数应该把收到的参数全都传递给原函数。

本例中，我们在包装函数里把传给原函数的若干个“位置参数”（positional argument）转换成一份 `float` 列表，然后用 `*args` 里的值（这些值可能和原函数接收到的参数不同）来调用原函数，并把原函数的返回值转换成 `float`，用作包装函数的返回值。

创建好包装函数之后，函数修饰器会将其返回。

但不巧的是，如果采用刚才的写法，那么修饰后的函数的 `_name_` 属性就和原函数不同了（变成了“`wrapper`”），而且即便原函数有 `docstring`，修饰后的函数也不会再有了。所以，这种替换方式并不完美。为解决此问题，Python 标准库提供了 `@functools.wraps` 修饰器，我们可以在修饰器里用它来修饰包装函数，以确保修饰后的函数的 `_name_` 与 `_doc_` 属性分别与原函数的名称及 `docstring` 相符。

```
def float_args_and_return(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        args = [float(arg) for arg in args]
        return float(function(*args, **kwargs))
    return wrapper
```

上面是修改之后的修饰器。它用 `@functools.wraps` 修饰器来保证 `wrapper()` 函数的 `_name_` 属性与传入的函数相符（本例中是 "mean"），并确保 `wrapper()` 函数的 `docstring` 与原函数相同（在本例中 `docstring` 为空）。在编写修饰器时，最好总是用 `@functools.wraps` 来修饰 `wrapper()` 函数，这样就能在“回溯信息”（trackback）里打印出受修饰的原函数名了（而不是包装函数的名称 "wrapper"），并且还能使用户访问到原

函数的 docstring。

```
@statically_typed(str, str, return_type=str)
def make_tagged(text, tag):
    return "<{0}>{1}</{0}>".format(tag, escape(text))

@statically_typed(str, int, str) # Will accept any return type
def repeat(what, count, separator):
    return ((what + separator) * count)[:-len(separator)]
```

用来修饰 make\_tagged() 与 repeat() 函数的 statically\_typed() 函数是个修饰器工厂，也就是一种能制作修饰器的函数。由于它不像修饰器那样只接受一个函数、方法或类作为其惟一参数，所以本身并不能算作修饰器。我们想用一套模板来制造修饰器，以指定受修饰的函数能够接受何种类型的位置参数（还可以指定返回值的类型）。为此，我们创建了 statically\_typed() 函数，其参数表示受修饰的函数应该具备何种类型的位置参数，此外还有个可选的关键字参数，用于指定受修饰函数的返回值类型， statically\_typed() 函数会根据这些参数来创建适当的修饰器。

Python 遇到 @statically\_typed(...) 这种代码时，会用给定的参数调用 statically\_typed() 函数，并用其返回的函数做修饰器来修饰 @statically\_typed(...) 后面的那个函数（在本例中，修饰 make\_tagged() 与 repeat() 函数）。

我们可以遵照一套固定的流程来创建修饰器工厂。首先，创建修饰器函数，在该函数内创建包装函数，包装函数的编写方式如前所述。在包装函数尾部，把调用原函数所得的返回值（也可以修改返回值，或用其他值来替换）返回给上一层。在修饰器函数尾部返回包装函数。最后，在修饰器工厂函数尾部返回修饰器。

```
def statically_typed(*types, return_type=None):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            if len(args) > len(types):
                raise ValueError("too many arguments")
            elif len(args) < len(types):
                raise ValueError("too few arguments")
            for i, (arg, type_) in enumerate(zip(args, types)):
                if not isinstance(arg, type_):
                    raise ValueError("argument {} must be of type {}".format(i, type_.__name__))
            result = function(*args, **kwargs)
            if (return_type is not None and
                not isinstance(result, return_type)):
                raise ValueError("return value must be of type {}".format(return_type.__name__))
            return result
        return wrapper
    return decorator
```

上面这段代码首先创建了名为 decorator() 的修饰器函数，然而函数名称在此处无关

紧要。在修饰器函数里，我们用老办法创建了包装函数。不过，本例的包装函数实现起来相当复杂，因为在调用原函数之前，它要检查用户传入的所有位置参数，判断其个数与类型是否符合要求；如果指定了返回值类型，那么在调用完原函数后，还要判断返回值是否符合要求。包装函数在完成上述判断之后，会返回原函数的执行结果。

创建好包装函数之后，修饰器会将其返回，而 `statically_typed()` 函数又会在其末尾将修饰器返回。Python 碰到 `@statically_typed(str, int, str)` 这种源代码之后，会调用 `statically_typed()` 函数。这个函数会返回它所创建的 `decorator()` 函数，而 `decorator()` 函数已经把用户传给 `statically_type()` 函数的参数捕获了。现在回到“@”这里，Python 看到“@”后，会执行 `decorator()` 函数，并把 `@statically_typed(str, int, str)` 后面的那个函数当成参数传给 `decorator()`。用作参数的这个函数既可以是开发者用 `def` 语句定义的，也可以是由其他修饰器所返回的。在本例中，这个函数是 `repeat()`，它是 `decorator()` 函数唯一的参数。`decorator()` 函数用捕捉到的状态（也就是用户传给 `statically_typed()` 函数的参数）来创建新的 `wrapper()` 函数，并将其返回。Python 用这个 `wrapper()` 替换掉原来的 `repeat()` 函数。

请注意，调用 `statically_typed()` 函数时，它所创建的 `decorator()` 函数会返回 `wrapper()`，而这个 `wrapper()` 函数捕获了其外围函数的状态，尤其是 `types` 元组和 `return_type` 关键字参数。像这样能够捕获状态的函数或方法就叫做“闭包”（closure）。正是由于 Python 支持闭包，所以我们才能够创建出“参数化的工厂函数”、修饰器及修饰器工厂。

从静态类型语言（比如 C、C++、Java）转入 Python 的开发者可能比较喜欢用修饰器对函数的参数及返回值执行静态类型检查，但这样做会增加 Python 程序在运行期的开销，而编译型语言则没有这种运行期开销。此外，尽管上面这个例子确实能展示 Python 的灵活性，但在动态类型语言里检测参数及返回值类型本身就是一种不太符合 Python 风格的做法（如果真需要在编译期执行静态类型检查，那么可以使用 5.2 节所说的 Cython）。下一节将要讲到的参数验证可能比参数类型检查更为有用。

修饰器的写法需要逐渐适应，但总体来说还是比较简单的。如果想创建“无参数化的”（unparameterized）函数修饰器或方法修饰器，那么就创建修饰器函数，然后在函数里创建并返回包装函数即可。前面讲到的 `@float_args_and_return` 修饰器与接下来要讲的 `@Web.ensure_logged_in` 修饰器都是如此。若想创建参数化的修饰器，则先要创建修饰器工厂，由工厂来创建修饰器，再由修饰器来创建包装函数，`statically_typed()` 函数就是这样创建的。

```
@application.post("/mailinglists/add")
@Web.ensure_logged_in
def person_add_submit(username):
    name = bottle.request.forms.get("name")
```

```

try:
    id = Data.MailingLists.add(name)
    bottle.redirect("/mailinglists/view")
except Data.Sql.Error as err:
    return bottle.mako_template("error", url="/mailinglists/add",
                                text="Add Mailinglist", message=str(err))

```

上述代码片段节选自一个管理“邮件列表”(mailing list)的 web 应用程序，该程序使用了名为 bottle 的轻量级 web 框架(网址: bottlepy.org)。此框架提供了 @application.post 修饰器，可以把函数与 URL 相关联。对于本例来说，只有当用户登录之后，我们才允许其访问 mailinglists/add 页面，若未登录，则将其重定向至 login 页面。按照传统写法，在每一个产生网页的函数里，都需要使用相同的代码来判断用户是否已经登录，而创建了 @Web.ensure\_logged\_in 这个修饰器之后，就可以把此事交由修饰器处理，这样的话，与登录有关的那部分代码就不会和函数混在一起了。

```

def ensure_logged_in(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        username = bottle.request.get_cookie(COOKIE,
                                              secret=secret(bottle.request))
        if username is not None:
            kwargs["username"] = username
        return function(*args, **kwargs)
        bottle.redirect("/login")
    return wrapper

```

当用户登录网站时，login 页面的后台程序会验证用户名与密码，若二者相符，则在浏览器中设置 cookie，此 cookie 的生命期在本次会话完结时终止。

如果与用户所请求之页面相关联的函数受到 @ensure\_logged\_in 修饰器保护(比如与 mailinglists/add 页面相关联的 person\_add\_submit() 函数)，那么就会执行由修饰器所定义的 wrapper() 函数。这个包装函数首先尝试从 cookie 中获取“用户名”(username)。若无法获取，则说明用户尚未登录，这时我们会将用户重定向到 web 应用程序的 login 页面。若能获取到，则说明用户已经登录，我们将 username 添加到关键字参数里，然后调用原函数并返回其结果。这样做好处是，开发者在编写原函数时，可以假定用户已经登录了，于是原函数就能直接使用 username，而无须再担心安全问题了。

## 2.4.2 类修饰器

我们经常要创建具有很多“可读写”属性的类，这样的类一般会有大量重复或相似的 getter 与 setter。比方说，要创建 Book 类，用以保存书的名称、ISBN、价格及数量。我们需要四个 @property 修饰器，其代码都差不多(例如: @property def title(self): return title)。此外，还需四个 setter 方法，每个方法都要验证用户所传入的参数，然而验证价格与验证数量所用的代码会很相似，只是最小值与最大值不同

而已。假如要建立许多这样的类，那么就会出现大量重复的代码。

幸好 Python 提供了类修饰器，可以消除这种重复代码。比方说，在 2.2 节那个例子中，我们通过类修饰器来给自己所创建的类提供接口检查功能，这样就不用每次都编写十行重复代码了。此处再举一例：我们用类修饰器来实现 Book 类，使其具备四个经过充分验证的属性，外加一个推算而来的只读属性。

```
@ensure("title", is_non_empty_str)
@ensure("isbn", is_valid_isbn)
@ensure("price", is_in_range(1, 10000))
@ensure("quantity", is_in_range(0, 1000000))
class Book:

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

    @property
    def value(self):
        return self.price * self.quantity
```

因为 `self.title`、`self.isbn` 等全都是属性，所以 `__init__()` 方法里的四个赋值操作均由相关的“属性设置器”（property setter）验证过了。然而我们无须手工创建这些属性，也无须手工编写其 `getter` 与 `setter` 代码，只需运用四次类修饰器，即可获得所需的全部功能。

`ensure()` 函数接受两个参数，一个是属性名，另一个是“验证器函数”（validator function），然后返回一个类修饰器。这个类修饰器会运用在 `@ensure` 之后的类上面。

上述代码首先创建未受修饰的 Book 类，然后调用一次 `ensure()` 函数（以便创建 `quantity` 属性），用此函数所返回的类修饰器来修饰 Book。修饰过的 Book 类多了个名叫 `quantity` 的属性。接下来，再次调用 `ensure()` 函数（以便创建 `price` 属性），用此函数所返回的类修饰器继续修饰 Book，该类经过二次修饰之后，便多了 `quantity` 与 `price` 属性。把这个过程再重复两遍，Book 类的四个属性就齐全了。

整个修饰过程看起来是逆向的，似乎和语句书写顺序相反。下面列出与该过程等效的伪代码：

```
ensure("title", is_non_empty_str)( # Pseudo-code
    ensure("isbn", is_valid_isbn)(
        ensure("price", is_in_range(1, 10000))(
            ensure("quantity", is_in_range(0, 1000000))(class Book: ...))))
```

运用修饰器之前，`class Book` 语句必须先执行，因为首次调用 `ensure()` 函数时（首次调用是为了给 Book 新增 `quantity` 属性）要以这个类对象为参数，而调用之后所返回的类对象还要用作上一层 `ensure()` 函数的参数，依此类推。

请注意，`price` 与 `quantity` 属性用的是同一个验证器函数，区别仅在于参数不同。实际上，`is_in_range()` 函数是个工厂函数，它会生成新的 `is_in_range()` 函数，在新函数中，传给原函数的最小值与最大值参数都会以硬编码的形式嵌入其中。

我们稍后就能看到，由 `ensure()` 函数所返回的类修饰器会向 `Book` 类中添加相关的属性，而属性的 `setter` 方法又会调用与该属性相对应的验证器函数，并向验证器传入两个参数，一个是属性名，另一个是用户想要设置的新属性值。若新值有效，则验证器函数照常返回，若无效，则抛出异常（比如 `ValueError`）。在讲解 `ensure()` 的实现代码之前，我们先来看两个验证器。

```
def is_non_empty_str(name, value):
    if not isinstance(value, str):
        raise ValueError("{} must be of type str".format(name))
    if not bool(value):
        raise ValueError("{} may not be empty".format(name))
```

上面这个验证器用来确保 `Book` 的 `title` 属性不是空字符串。由 `ValueError` 的用法可以看出，把属性名放在错误消息里对调试工作是很有帮助的。

```
def is_in_range(minimum=None, maximum=None):
    assert minimum is not None or maximum is not None
    def is_in_range(name, value):
        if not isinstance(value, numbers.Number):
            raise ValueError("{} must be a number".format(name))
        if minimum is not None and value < minimum:
            raise ValueError("{} {} is too small".format(name, value))
        if maximum is not None and value > maximum:
            raise ValueError("{} {} is too big".format(name, value))
    return is_in_range
```

上面这个函数是个工厂函数，每次调用时，都会创建新的验证器函数，用以确保给定的参数是个数字（相关的检测语句用到了名为 `numbers.Number` 的抽象基类），而且其值位于适当范围内。创建好验证器之后，工厂函数就会将其返回。

```
def ensure(name, validate, doc=None):
    def decorator(Class):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            validate(name, value)
            setattr(self, privateName, value)
        setattr(Class, name, property(getter, setter, doc=doc))
    return Class
return decorator
```

`ensure()` 函数会以给定的属性名、验证器函数及可选的 `docstring` 为参数来创建类修饰器，而调用 `ensure()` 就相当于用它所创建的类修饰器来修饰特定的类，修饰后的类会有新的属性。

`decorator()` 函数只有一个参数，这个参数是类对象。此函数首先创建 `privateName` 变量，其属性值存放在与该变量同名的 `attribute` 里。（在本例中，`Book` 类的 `self.title` 属性将存放在名为 `self.__title` 的私有 `attribute` 里面。）接下来，创建 `getter` 函数，用以返回保存在私有 `attribute` 中的属性值。Python 内置的 `getattr()` 函数接受两个参数，一个是对象，另一个是 `attribute` 名，它返回对象中的 `attribute` 值，如果没有找到这个 `attribute`，那么就抛出 `AttributeError`。创建完 `getter` 后，又创建了 `setter` 函数，此函数调用捕获到的 `validate()` 函数，在 `validate()` 函数没有抛出异常的情况下，把私有的 `attribute` 修改成新值。Python 内置的 `setattr()` 函数有三个参数，分别是对象、`attribute` 名以及新的 `attribute` 值，此函数会把相应的 `attribute` 设置成新值，若原来没有叫这个名字的 `attribute`，则会新建一个。

编写完 `getter` 及 `setter` 后，`decorator()` 用这两个函数新建了一个属性，并通过内置的 `setattr()` 函数把该属性作为 `attribute` 添加到用户所传入的类里面，这个 `attribute` 的名称与公开的属性名一致。创建属性时所用的 `property()` 函数是由 Python 内置的，它有四个可选参数，分别是 `getter`、`setter`、`deleter` 与 `docstring`，此函数返回创建好的属性。前面我们还曾把这个函数当成“方法修饰器”（method decorator）来用。`decorator()` 函数最后会把修改好的类返回给 `ensure()`，而 `ensure()` 这个“类修饰器工厂函数”（class decorator factory function）又会把 `decorator()` 函数返回给调用者。

## 1. 用类修饰器新增属性

在上面那个例子中，每一个需要验证的 `attribute` 都必须用 `@ensure` 类修饰器来描述。有些 Python 程序员不喜欢把多个类修饰器迭加在一起，他们会把相关的 `attribute` 都放在类里面，然后只用一个类修饰器来修饰，这样写出来的代码更易读懂。

```
@do_ensure
class Book:

    title = Ensure(is_non_empty_str)
    isbn = Ensure(is_valid_isbn)
    price = Ensure(is_in_range(1, 10000))
    quantity = Ensure(is_in_range(0, 1000000))

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

    @property
    def value(self):
        return self.price * self.quantity
```

上面是改写后的 `Book` 类，我们用 `@do_ensure` 类修饰器与 `Ensure` 实例来实现与前例相同的功能。每次构造 `Ensure` 对象时，都要传入验证函数，而 `@do_ensure` 类修饰器

会用带有验证机制的同名属性来替换相应的 `Ensure` 实例。本例所用的验证函数（比如 `is_non_empty_str()` 等）与早前范例所用的相同。

```
class Ensure:
    def __init__(self, validate, doc=None):
        self.validate = validate
        self.doc = doc
```

上面这个类很简单，它用来保存验证器函数，相关属性的 `setter` 函数稍后会用到这个验证器。另外，在构建 `Ensure` 实例时，还可以指定属性的 `docstring`。比方说，`Book` 类的 `title` 属性一开始是个 `Ensure` 实例，但创建好 `Book` 类之后，`@do_ensure` 修饰器就会把每个 `Ensure` 实例都替换成对应的属性。所以，在修饰之后的 `Book` 类中，名为 `title` 的这个 `attribute` 就不再是 `Ensure` 实例了，而变成了 `title` 属性（该属性的 `setter` 函数会用到原来 `Ensure` 实例中的验证函数）。

```
def do_ensure(Class):
    def make_property(name, attribute):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            attribute.validate(name, value)
            setattr(self, privateName, value)
        return property(getter, setter, doc=attribute.doc)
    for name, attribute in Class.__dict__.items():
        if isinstance(attribute, Ensure):
            setattr(Class, name, make_property(name, attribute))
    return Class
```

上面这个类修饰器可分为三部分。在第一部分里，我们定义了名为 `make_property()` 的“嵌套函数”（nested function）。该函数有两个参数，一个是属性名（比如 "title"），另一个是 `Ensure` 类型的 `attribute`，此函数将返回一个属性，该属性会把其值保存在私有的 `attribute` 中（比如 `title` 属性的值就保存在名为 `__title` 的 `attribute` 中）。属性的 `setter` 函数还会调用原来 `Ensure` 实例的验证器函数。在第二部分里，我们遍历类中的每一个 `attribute`，并用新的属性来替换原先的 `Ensure` 实例。第三部分会把修改后的类返回。

执行完修饰器后，受修饰的类里原有的 `Ensure` 型 `attribute` 都会被同名且带有验证机制的属性所取代。

从理论上说，可以不写那个叫做 `make_property()` 的嵌套函数，而是把其中的代码都放在 `if isinstance()` 这行测试语句下面。但实际上，由于“后期绑定”（late binding）机制的某些问题，我们没办法这么做，所以必须把相关代码单独放在一个函数里。在创建修饰器或修饰器工厂时，此问题时有发生，不过这些状况大都可以通过单独编写一个函数来解决（这个函数有可能像本例一样是个嵌套函数）。

## 2. 用类修饰器实现继承

有时我们创建基类只是为了使子类能够继承其中的某些方法或数据。当要创建的子类个数比较多时，这种设计方式就显得很灵活了，因为每个子类无须重复实现基类的方法或数据。然而若是继承下来的那些方法或数据在子类里都无须改动，那么可以改用类修饰器来达到此目的。

比方说，在3.5节中，我们将编写名为Mediated的基类，其中提供了self.mediator数据属性以及on\_change()方法。Button与Text均继承自该类，二者都会用到基类的数据及方法，但却不修改它们。

```
class Mediated:
    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
```

上面这个基类节选自mediator1.py。Button及Text子类都使用通常的写法来继承它（也就是class Button(Mediated): ... 和class Text(Mediated): ...）。但由于子类无须修改继承下来的on\_change()方法，所以我们也可以改用类修饰器来实现继承。

```
def mediated(Class):
    setattr(Class, "mediator", None)
    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)
    setattr(Class, "on_change", on_change)
    return Class
```

上面这段代码节录自mediator1d.py。类修饰器的用法与早前范例相同：即分别用@mediated class Button: ... 及@mediated class Text: ... 来修饰Button及Text类。修饰后的类的行为与通过继承而来的类相同。

函数修饰器与类修饰器都是Python中易用且强大的功能。我们刚才已经看到，类修饰器有时可以实现继承。而创建修饰器就是一种简单的“元编程”(metaprogramming)形式，类修饰器通常可用来取代更为复杂的元编程技术（比如“元类”，metaclass）。

## 2.5 外观模式

如果某套接口因为太过复杂或太专注于底层细节而变得不易使用，那么可考虑用“外观模式”(Facade Pattern)将其简化并统合起来。

由Python标准库所提供的模块可以处理gzip、tarball、zip等格式的压缩文档，不过处

理每种格式所用的接口却不同。现在假定我们想通过一套简单而一致的接口来获知压缩文档里的各个文件名，并将其解压缩。本节将使用外观模式来设计这套接口，把真正的处理工作交给标准库来做。

图 2.7 演示了我们想要提供给用户的接口（其中含有 filename 属性、names() 方法与 unpack() 方法）以及该接口下掩藏的三套底层接口。Archive 实例中存有压缩文档的名称，只有当用户询问其中压缩的文件名或要求对其解压缩时，才需要真正把压缩文档打开。（本节中的范例代码选自 Unpack.py 文件。）

```
class Archive:
    def __init__(self, filename):
        self._names = None
        self._unpack = None
        self._file = None
        self.filename = filename
```

self.\_names 变量用来保存一个 callable，这个 callable 可以返回压缩文档内的文件名列表。self.\_unpack 变量与之相似，其所存放的 callable 对象可把压缩文档中的所有文件都解压到当前目录。self.\_file 存放打开的文件对象，此对象用来表示当前这份压缩文档。self.filename 是个只读属性，用来保存压缩文档本身的文件名。

```
@property
def filename(self):
    return self._filename

@filename.setter
def filename(self, name):
    self.close()
    self._filename = name
```

如果用户打开了压缩文档之后又想修改 filename 属性（比如通过 archive.filename = newname 语句来修改），那么 Archive 会先把当前这份压缩文档关掉。由于 Archive 类采用了“延迟求值”（lazy evaluation）机制，所以修改属性之后，新的压缩文档并不会立即开启，只在有需要时才会打开它。

```
def close(self):
    if self._file is not None:
        self._file.close()
    self._names = self._unpack = self._file = None
```

按理说，用户在用完 Archive 类的实例之后，应该调用其 close() 方法。该方法会

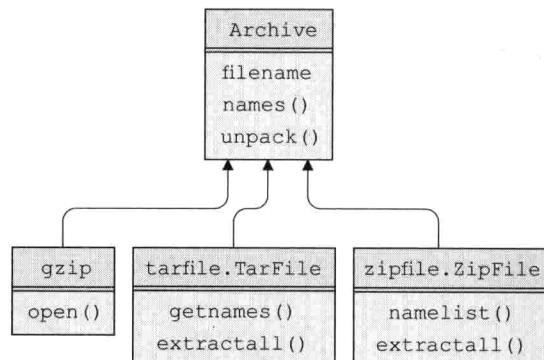


图 2.7 用外观模式来设计压缩文档处理接口

把已经打开的文件对象关掉，并把 `self._name`、`self._unpack` 和 `self._file` 设为 `None`，令这些变量失效。

但实际上，用户只要在 `with` 语句范围内使用，就无须自行调用 `close()`，因为我们已经把 `Archive` 类做成了“情境管理器”(context manager)。比方说，可以这样来使用它：

```
with Archive(zipFilename) as archive:
    print(archive.names())
    archive.unpack()
```

上述代码创建了 `Archive` 实例，用来把 `zip` 格式压缩文档中的文件名打印到控制台上，并把其中所有文件都解压至当前目录。由于 `archive` 是个环境管理器，所以当程序执行到 `with` 语句块的范围之外时，会自动执行 `archive.close()`。

```
def __enter__(self):
    return self

def __exit__(self, exc_type, exc_value, traceback):
    self.close()
```

只要有上面这两个方法，`Archive` 就能成为环境管理器。`__enter__()` 方法返回 `self` (也就是当前这个 `Archive` 实例)，此返回值会赋给 `with ... as` 语句中的变量。`__exit__()` 方法会把当前已经打开的文件对象关掉，由于此方法默认返回 `None`，所以执行过程中发生的异常会照常传播。

```
def names(self):
    if self._file is None:
        self._prepare()
    return self._names()
```

上述方法会返回压缩文档中的文件名列表，若该文档尚未开启，则调用 `self._prepare()` 方法将压缩文件打开，并把适当的 `callable` 赋给 `self._names` 与 `self._unpack`。

```
def unpack(self):
    if self._file is None:
        self._prepare()
    self._unpack()
```

上述方法会把压缩文档中的所有文件都解压，不过我们稍后就会看到，只有在每个文件的名称都“安全”(safe) 时，才会这么做。

```
def _prepare(self):
    if self.filename.endswith((".tar.gz", ".tar.bz2", ".tar.xz",
                            ".zip")):
        self._prepare_tarball_or_zip()
    elif self.filename.endswith(".gz"):
        self._prepare_gzip()
    else:
        raise ValueError("unreadable: {}".format(self.filename))
```

上述方法会把解压前的准备工作指派给合适的方法。由于 tarball 与 zip 格式所需的代码非常相似，所以我们把这两种压缩文档交由同一个方法处理。gzip 格式的压缩文档与二者不同，因此单独放在另一个方法中处理。

这两个“准备方法”(preparation method)都必须把对应的 callable 赋给 `self.names` 及 `self._unpack` 变量，使刚才的 `names()` 与 `unpack()` 方法可以调用这些 callable。

```
def _prepare_tarball_or_zip(self):
    def safe_extractall():
        unsafe = []
        for name in self.names():
            if not self.is_safe(name):
                unsafe.append(name)
        if unsafe:
            raise ValueError("unsafe to unpack: {}".format(unsafe))
        self._file.extractall()
    if self.filename.endswith(".zip"):
        self._file = zipfile.ZipFile(self.filename)
        self._names = self._file.namelist
        self._unpack = safe_extractall
    else: # Ends with .tar.gz, .tar.bz2, or .tar.xz
        suffix = os.path.splitext(self.filename)[1]
        self._file = tarfile.open(self.filename, "r:" + suffix[1:])
        self._names = self._file.getnames
        self._unpack = safe_extractall
```

上述方法首先创建名为 `safe_extractall()` 的嵌套函数，该函数检查压缩文档中的文件名是否均安全，如果 `is_safe()` 方法判定某些文件名不安全，那么 `safe_extractall()` 方法就会抛出 `ValueError`。若是所有文件名都没问题，那么就调用 `tarball.TarFile.extractall()` 或 `zipfile.ZipFile.extractall()` 方法。

创建好 `safe_extractall()` 函数后，我们根据压缩文档的扩展名来创建 `tarfile.Tarfile` 或 `zipfile.ZipFile`，并将其赋给 `self._file`。接下来，把 `self._names` 的值设置成相应的绑定方法（也就是 `namelist()` 或 `getnames()`），并把 `self._unpack` 的值设为刚刚创建好的 `safe_extractall()` 函数。由于该函数是闭包，所以能够捕获 `self`，继而可通过 `self` 来访问 `self._file` 的值，并调用相应的 `extractall()` 方法。（绑定方法与非绑定方法的区别请参阅本节中的补充知识。）

### 绑定方法与非绑定方法



绑定方法 (bound method) 就是已经同类实例相关联的方法。假设现在有 `Form` 类，类中有个 `update_ui()` 方法。如果在 `Form` 的某个方法里写上 `bound = self.update_ui` 这行代码，那就等于把指向 `Form.update_ui()` 方法的对象引用赋给了 `bound`，并把 `Form.update_ui()` 方法同 `Form` 类的特定实例（用 `self` 表示）相绑定。绑定方法可以直接调用，比如：`bound()`。

非绑定方法 (unbound method) 是不与实例相关联的方法。比方说，假如刚才那行代码写的是 `unbound = Form.update_ui`，那么 `unbound` 还是会成为指向 `Form.update_ui()` 方法的对象引用，但这次不会与特定的实例相绑定。也就是说，如果想调用非绑定方法，那么必须把适当的实例用作其首个参数才行，例如：`form = Form(); unbound(form)`。（与传统的 Python 不同，Python 3 严格来说没有“非绑定方法”这一概念，所以 `bound` 就是个底层函数对象，这两种对非绑定方法的处理方式只有在元编程时才会偶尔体现出差别。）

```
def is_safe(self, filename):
    return not (filename.startswith("/", "\\")) or
        (len(filename) > 1 and filename[1] == ":" and
         filename[0] in string.ascii_letters) or
        re.search(r"\[\.\]\[\/\\\]", filename)
```

如果将恶意压缩文档解压缩，那么可能会把重要的系统文件覆写成无用或危险的内容。因此，切勿把包含绝对路径或相对路径的压缩文档打开，而且总应该以“非特权用户”（unprivileged user）的身份开启压缩文档（也就是不要以“根用户”（root）或“管理员”（Administrator）身份开启）。

如果文件名以“斜线”（forward slash）或“反斜线”（backslash）开头（这表示绝对路径），包含“`.. /`”、“`..\`”（由于相对路径的目标不定，所以含有这两种文件名的压缩文档也不安全）或以“`D:`”这样的 Windows 盘符开头，那么 `is_safe()` 方法就返回 `False`。

换句话说，以绝对路径开头或其中包含相对路径的文件名都是不安全的，而其他文件名则会使该方法返回 `True`。

```
def _prepare_gzip(self):
    self._file = gzip.open(self.filename)
    filename = self.filename[:-3]
    self._names = lambda: [filename]
    def extractall():
        with open(filename, "wb") as file:
            file.write(self._file.read())
    self._unpack = extractall
```

上述方法将打开的文件对象赋给 `self._file` 变量，并把适当的 `callable` 赋给 `self._names` 及 `self._unpack` 变量。这次我们需要自己编写 `extractall()` 函数来读写相关数据。

外观模式很适合用来创建简单易用的接口，其优点在于能够隔离底层细节，而缺点则是可能会丧失某些微调能力。然而外观模式并不会把底层功能遮掩或废弃，所以大部分时间里都可以直接使用外观模式，而在需要微调时，则可以深入底层的类。

外观模式看起来很像适配器模式，其区别在于，外观模式是在复杂的接口上提炼出一套简单的接口，而适配器则是把其他接口（未必很复杂）转换成标准接口。这两种模式可以

结合起来。比方说，我们可以定义一套处理压缩文档的标准接口（能够处理 tarball、zip 及 Windows 系统的 .cab 等格式），并用适配器模式把每种格式的压缩文档处理接口都分别转换成标准接口，然后在标准接口上面搭建外观层，这样用户就无须关注当前操作的压缩文档是哪种格式了。

## 2.6 享元模式

如果有许多比较小的对象需要处理，而这些小对象很多又彼此相同，那么就可以使用“享元模式”（Flyweight Pattern）。该模式的实现方式为：只给每种对象创建一个实例，并在有需要时共享此实例。

由于 Python 使用对象引用，所以很自然地体现出了享元模式的思路。比方说，如果有字符串列表很长，而且其中许多字符串都一样，那么，使用对象引用（也就是变量）来存储要比直接使用“字面量字符串”（literal string）存储节省许多内存。

```
red, green, blue = "red", "green", "blue"
x = (red, green, blue, red, green, blue, red, green)
y = ("red", "green", "blue", "red", "green", "blue", "red", "green")
```

在上述代码片段中，`x` 元组用 8 个对象引用来保存 3 个字符串，而 `y` 元组则用 8 个对象引用保存了 8 个字符串，因为这种简化的写法实际上与 `_anonymous_item0 = "red", ...`  
`_anonymous_item7 = "green"; y = (_anonymous_item0, ... _anonymous_item7)` 等效。

要想在 Python 语言中利用享元模式，最简单的办法可能就是使用 `dict` 了，它可以把每个值都同独特的键关联起来。比方说，在创建大量 HTML 页面时，我们想根据 CSS（Cascading Style Sheets，层叠样式表）来指定“字型”（font），这样就不用每次都创建新字型了，而是可以预先（或在首次使用时）把它们保存在 `dict` 里面。等需要使用某个字型时，再将其从 `dict` 里取出来。这样就能保证每种字型无论使用多少次，都只会创建一次。

有时我们需要处理大量对象，而其中绝大部分或所有对象都互不相同，并且这些对象未必很小。在这种情况下，有个简单的办法可以降低内存用量，这就是使用 `__slots__`。

```
class Point:
    __slots__ = ("x", "y", "z", "color")

    def __init__(self, x=0, y=0, z=0, color=None):
        self.x = x
        self.y = y
        self.z = z
        self.color = color
```

上面这个简单的 `Point` 类可以存放点的三维坐标及颜色。由于用了 `__slots__`，所以 `Point` 实例都没有自己的 `dict`（也就是没有 `self.__dict__`）。然而，这样做同时也

意味着不能向单个对象中随意添加 attribute。(该类代码节选自 pointstore1.py。)

在某台电脑中测试时，用上述代码创建含有 100 万个点的元组（测试程序几乎没有做其他事情）需要 2.5 秒，并占用 183MiB 内存。若是不用 `__slots__`，那么执行时间能少零点几秒，但内存占用量却高达 312MiB。

默认情况下，Python 总是会花更多内存来提升执行速度，但如果有必要的话，通常也可以反过来做，那就是通过降低执行速度来节省内存用量。

```
class Point:
    __slots__ = ()
    __dbm = shelve.open(os.path.join(tempfile.gettempdir(), "point.db"))
```

上面列出了第二版 Point 类的前几行代码（该类节选自 pointstore2.py）。它用 DBM（键值对）数据库来保存数据，而数据库本身则存放在磁盘文件中。指向 DBM 的对象引用保存在静态的（也就是类级别的）Point.`__dbm` 变量里。所有 Point 实例都使用同一份底层 DBM 文件。我们首先要打开 DBM 文件，以便后续使用。`shelve` 模块的默认做法是：如果没发现相关的 DBM 文件，那就自动创建一份。（稍后我们将演示如何保证 DBM 文件能正常关闭。）

在存储值时，`shelve` 模块会将其“序列化”（pickle），而在获取值时，则会将其“反序列化”（unpickle）。（由于在反序列化的过程中能够执行任意 Python 代码，所以 Python 的序列化格式是不安全的。因此，切勿使用由不可信的数据源所提供的序列化数据，也不要将无访问限制的数据序列化。如果想在这些情况下使用序列化数据，那么可通过“校验和”（checksum）及“加密”（encryption）等自制的安全措施来保证数据安全。）

```
def __init__(self, x=0, y=0, z=0, color=None):
    self.x = x
    self.y = y
    self.z = z
    self.color = color
```

上述方法的代码与 pointstore1.py 中的完全相同，但这些值都会存储到底层的 DBM 文件里。

```
def __key(self, name):
    return "{:X}:{:}{}".format(id(self), name)
```

上述方法可提供 Point 实例中 x、y、z、color 等 attribute 的“键字符串”（key string）。这个键由实例的十六进制 ID（ID 是由 Python 语言内置的 `id()` 函数所返回的独特数字）及 attribute 名构成。比方说，某个 Point 实例的 ID 是 3 954 827，那么其 x 值所对应的键字符串就是“3C588B:x”，而 y 值则可由“3C588B:y”这个键查到，其余 attribute 亦是如此。

```
def __getattr__(self, name):
    return Point.__dbm[self.__key(name)]
```

上述方法会在访问 Point 对象的某个 attribute 时（比如执行 `x = point.x` 时）调用。

DBM 数据库的键与值必须是 bytes。好在 Python 的 DBM 模块可以接受 str 或 bytes 作键，遇到 str 时，会用默认的 UTF-8 编码将其转换为 bytes。如果像本例一样使用 shelve 模块，那么凡是可序列化的值就都能保存到数据库里，因为 shelve 模块会根据情况在其他类型与 bytes 之间相互转换。

上面两个方法使我们能够获得与 attribute 相关的键，并根据键来查出 attribute 值。另外，由于使用了 shelve 模块，所以获取到的值会从序列化的 bytes 自动转换成原本的类型（比方说，获取到的点颜色值会是 int 或 None 类型）。

```
def __setattr__(self, name, value):
    Point._dbm[self.__key(name)] = value
```

设置 Point 的 attribute 时（例如执行 `point.y = y` 时）会调用上述方法。在该方法中，我们会根据键查出相关的 attribute 值，并通过 shelve 模块把值序列化为 bytes。

```
atexit.register(_dbm.close)
```

在 Point 类最后，我们通过 atexit 模块的 register() 函数来注册 DBM 的 close() 方法，使得程序在终止时能够调用此方法。

在某台测试机上创建含有 100 万个点的数据库大约需要一分钟时间，但程序只占用 29MiB 内存（外加 361MiB 的磁盘文件），而第一版程序则要占用 183MiB 内存。尽管生成 DBM 文件确实需要一些时间，但只要生成好了，查询速度就会很快，因为大部分操作系统都会把频繁使用的磁盘文件缓存起来。

## 2.7 代理模式

若想用一个对象来代表另一个对象，则可使用“代理模式”（Proxy Pattern）。《Design Patterns》一书举了四个用例。第一个用例是“远程代理”（remote proxy）：用本地对象来代表远程对象。RPyC 程序库就是个很好的例子，它可以在服务器端创建对象，并在一或多台客户端中创建针对这些对象的代理（6.2 节将会介绍这个程序库）。第二个用例是“虚代理”（virtual proxy），用来创建能够代表复杂对象的轻量级对象，只在确有必要时才会真正去创建那个复杂对象。本节所举的例子就是这种代理。第三个用例是“保护代理”（protection proxy），可根据客户端的访问权限来确定不同的访问级别。最后一种用例是“智能引用”（smart reference），可用来在“访问对象时执行额外操作”（performs additional actions when an object is accessed）。这些代理模式都可以采用同一套编码方式来实现，其中第四种代理还可以通过描述符来实现（比方说，利用 @property 修饰器，以属性来取代普通对象）<sup>⊖</sup>。

---

<sup>⊖</sup> 《Programming in Python 3, Second Edition》一书（该书详情参见附录 B）讲解了描述符，也可参阅在线文档：<http://docs.python.org/3/reference/datamodel.html#descriptors>。

代理模式也可用于单元测试。例如，受测代码所需访问的资源并非随时可用，或是所需使用的类尚未开发完毕而依然不完整，那就可以考虑为资源或类创建代理，令代理对象提供所有接口，并且用“桩”(stub)来表示那些缺失的功能。这种做法非常有用，Python 3.3 包含了 `unittest.mock` 库，可用来创建“模拟对象”(mock object)，并设置“桩”来表示缺失的方法。(参见：<http://docs.python.org/py3k/library/unittest.mock.html>。)

本节范例所假定的使用场景是：我们可能会创建很多图像，但最后只会用到其中一张。`Image` 模块与功能相仿但速度更快的 `cyImage` 模块都可以创建图像（第 3.12 节及 5.3 节分别讲解二者），但它们一开始就会把图像创建在内存里。而我们只会用到这些图像中的一张，所以更好一些的办法是：创建许多轻量级图像代理，然后只在真正有需要时才去创建实际图像。

除了构造器之外，`Image.Image` 类的接口还有十一个方法：`load()`、`save()`、`pixel()`、`set_pixel()`、`line()`、`rectangle()`、`ellipse()`、`size()`、`subsample()`、`scale()`。（此外，还有一些静态的便捷方法以及等效的模块函数，例如 `Image.Image.color_for_name()` 及 `Image.color_for_name()`。）

代理类只需要实现 `Image.Image` 中我们必须用到的那些方法即可。首先来看代理类的用法。本节范例代码选自 `imageproxy1.py`，绘制出的图像如图 2.8 所示。

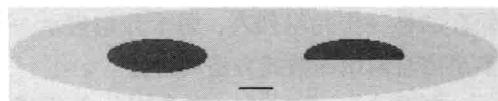


图 2.8 绘制好的图像

```
YELLOW, CYAN, BLUE, RED, BLACK = (Image.color_for_name(color)
    for color in ("yellow", "cyan", "blue", "red", "black"))
```

首先，需要用 `Image` 模块的 `color_for_name()` 函数创建一些颜色常量。

```
image = ImageProxy(Image.Image, 300, 60)
image.rectangle(0, 0, 299, 59, fill=YELLOW)
image.ellipse(0, 0, 299, 59, fill=CYAN)
image.ellipse(60, 20, 120, 40, BLUE, RED)
image.ellipse(180, 20, 240, 40, BLUE, RED)
image.rectangle(180, 32, 240, 41, fill=CYAN)
image.line(181, 32, 239, 32, BLUE)
image.line(140, 50, 160, 50, BLACK)
image.save(filename)
```

上面这段代码先创建了 `ImageProxy` 对象，我们在创建时把需要使用的 `Image` 类传给了构造器。然后又在对象上面绘制了一些内容，最后将绘制好的图像存储起来。假如创建图像时调用的不是 `ImageProxy()` 而是 `Image.Image()`，那么剩下的绘制操作依然能照常执行。但是，采用了图像代理之后，只有在调用 `save()` 方法时才会去创建真正的图像，这样的话，在执行保存操作之前，创建图像的开销（无论是内存开销还是处理开销）就变得非常小，若是最后不保存图像而直接将其丢弃，那么损失也会很低。若用 `Image.Image` 来创建，则一开始就需要很大开销（也就是说，一开始就要创建大小为 `width × height` 的数组

用以保存颜色值), 而且在绘制时还需要执行很多处理工作(例如在填充矩形时, 要计算出需要填充的像素, 并把它们的颜色都设置好), 即便最后决定要丢弃这张图像, 也还是得执行丢弃之前那些操作。

```
class ImageProxy:
    def __init__(self, ImageClass, width=None, height=None, filename=None):
        assert (width is not None and height is not None) or \
               filename is not None
        self.Image = ImageClass
        self.commands = []
        if filename is not None:
            self.load(filename)
        else:
            self.commands = [(self.Image, width, height)]

    def load(self, filename):
        self.commands = [(self.Image, None, None, filename)]
```

只要 ImageProxy 所提供的接口足够用, 它就能代表 Image.Image(如果构造时传入了支持 Image 接口的其他类, 那么也能代表那个类)。ImageProxy 并不保存图像, 它保存的是一份元组列表, 每个元组表示一条命令, 其首个元素是函数或非绑定方法, 其余元素是传给调用函数或方法的参数。

创建 ImageProxy 对象时, 必须指定长和宽(以便按此大小来新建图像)或文件名。如果用文件名创建 ImageProxy, 那么就会保存一条命令, 这条命令旨在调用 Image.Image() 构造器, 构造器所用的 width 及 height 参数都是 None, 而 filename 参数则是创建 ImageProxy 时所传入的文件名, ImageProxy.load() 方法所对应的命令与此相同。创建好 ImageProxy 对象之后, 如果又调用了 ImageProxy.load() 方法, 那么先前的全部命令都将丢弃, self.commands 命令列表中只会留下一条新建图像的命令。若用给定的长度与宽度来创建 ImageProxy 对象, 则对应的命令中保存的是 Image.Image() 构造器, 构造器所用的 width 及 height 参数是创建时所传入的长度与宽度。

如果调用了代理对象所不支持的方法(比如 pixel()), 那么 Python 就会发现这个方法找不到, 从而自动抛出 AttributeError, 而这正是我们想要的效果。还有一种处理办法: 如果代理对象不支持将要调用的方法, 那就把实际的 Image 对象创建出来, 并在此对象上执行后续操作。(imageproxy2.py 程序采用这种办法, 该程序的代码没有列在本节中。)

```
def set_pixel(self, x, y, color):
    self.commands.append((self.Image.set_pixel, x, y, color))

def line(self, x0, y0, x1, y1, color):
    self.commands.append((self.Image.line, x0, y0, x1, y1, color))

def rectangle(self, x0, y0, x1, y1, outline=None, fill=None):
    self.commands.append((self.Image.rectangle, x0, y0, x1, y1,
                           outline, fill))
```

```
def ellipse(self, x0, y0, x1, y1, outline=None, fill=None):
    self.commands.append((self.Image.ellipse, x0, y0, x1, y1,
                          outline, fill))
```

Image.Image类的接口中有四个绘制方法：line()、rectangle()、ellipse()、set\_pixel()。我们的ImageProxy类完全支持这些方法，但并不当场执行操作，而是把操作及其参数做成一条命令，放在self.commands列表里。

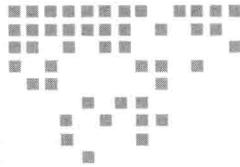
```
def save(self, filename=None):
    command = self.commands.pop(0)
    function, *args = command
    image = function(*args)
    for command in self.commands:
        function, *args = command
        function(image, *args)
    image.save(filename)
    return image
```

只有在保存时才需要创建真正的图像，也只有此时才会有真正的处理开销及内存开销。ImageProxy的设计方式决定了其首个命令一定是新建图像（可能是根据长宽来创建，也可能是从既有文件中加载）。所以我们采用特殊方式来处理第一条命令：将执行该命令所得的返回值保存起来，这个返回值肯定是个Image.Image或cyImage.Image。然后，遍历剩下的命令，并依次执行之，由于执行的都是非绑定方法，所以需要把image变量作为首个参数（也就是self）传进去。最后，调用Image.Image.save()方法，保存图像。

虽说Image.Image.save()方法在发生错误时会抛出异常，但这个方法本身是没有返回值的。然而ImageProxy的save()方法却稍有不同，它会把创建好的Image.Image对象返回给调用者，以备后续处理时所需。这样修改应该不会出问题，因为假如调用者不使用返回值的话（比如调用Image.Image.save()方法时，我们就没打算使用返回值），那么Python就会将其直接丢弃。imageproxy2.py程序无须像这样修改，因为它有个类型为Image.Image的image属性可供访问，如果访问时图像尚未创建，那么会当场创建一份。

像本例这样把命令存储起来，可以为实现“执行–撤销”（do-undo）功能做准备，这一话题请参考3.2节的命令模式以及3.8节的状态模式。

结构型设计模式都可以用Python语言实现出来。适配器模式与外观模式能够把已有的类放在新环境下重新使用，而桥接模式则可以把某个类里的复杂功能嵌入另一个类中。组合模式可以非常方便地创建出对象层次结构，但Python中却很少用到它，因为采用dict就可以实现相同的功能了。修饰器模式特别有用，Python语言对此提供了原生支持，我们还可以用修饰器来修饰类。Python的对象引用机制可以视为享元模式的变种。代理模式在Python中实现起来非常简单。设计模式不仅可用于创建各种简单及复杂的对象，而且还能指导对象的行为，也就是规定单个对象或一组对象应该怎样完成其工作。下一章就要讲解这些“行为型设计模式”。



Chapter 3

## 第3章

# Python 的行为型设计模式

行为型设计模式关注做事的过程，也就是算法及对象间的交互。这些模式提供了构思及规划计算过程的好办法，与前两章所讲的那两种模式一样，某些行为型设计模式也可以直接用 Python 内置的语法来表述。

Perl 语言有句广为人知的口号，叫做“there's more than one way to do it”（做事的办法不止一种），然而 Tim Peters 在“Zen of Python”（Python 之道）中却说：“there should be one—and preferably only one—obvious way to do it”（做事情总该有明确的办法才对，而且最好只有一种）。<sup>⊖</sup> 不过，与其他编程语言一样，想在 Python 中完成某项任务，有时也能找到两种或两种以上的办法，在引入了“推导”（comprehension）和“生成器”（generator）之后更是如此。比如说，我们既可以直接使用列表推导功能，又可以用等效的 `for` 循环来实现；既可以直接使用生成器表达式，又可以用带 `yield` 语句的函数来实现。本章要讲的“协程”（coroutine）也是这样，在完成某些特定工作时，它能提供一种新的实现方式。

### 3.1 责任链模式

“责任链模式”（Chain of Responsibility Pattern）可将请求的发送方与处理请求的接收方解耦。这样的话，某函数就不用直接调用别的函数了，而是可以把请求发送给一个由诸多接收者所组成的链条。链条中的首个接收者可以处理请求并停止责任链（也就是不再继续往下传递），也可以把请求发给下一个接收者。而第二个接收者也有这两种选择，此过程可一直延续至最后一个接收者，该接收者可将请求丢弃，也可抛出异常。

<sup>⊖</sup> 在 Python 的交互提示符界面输入 `import this`，即可显示“Zen of Python”。

假设有个用户界面接收到一些需要处理的事件。有些事件是由用户触发的，比如鼠标事件和键盘事件，有些事件则是由系统触发的，比如计时器事件。下面两小节分别用两种办法处理事件。首先我们看看如何用传统方式创建事件处理责任链，然后再讲解怎样使用协程来实现“基于管道的”(pipeline-based)解决方案。

### 3.1.1 用常规方式实现责任链

本节用常规方法来实现责任链模式以处理事件，每个事件都有对应的事件处理类。

```
handler1 = TimerHandler(KeyHandler(MouseHandler(NullHandler())))
```

图 3.1 所演示的这条责任链由四个独立的事件处理程序类构成。由于未处理的事件会被直接丢弃，所以在构建 MouseHandler 时，也可以用 None 做参数，或者不传参数。

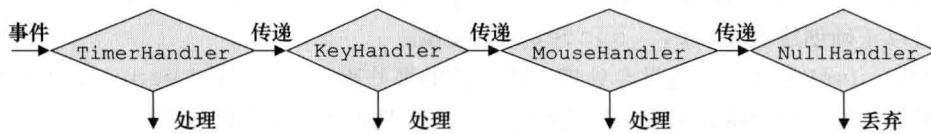


图 3.1 处理事件所用的责任链

由于每个处理程序所能处理的事件类型都是固定的，所以创建责任链时，这些对象的先后顺序无关紧要。

```
while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    handler1.handle(event)
```

我们通常会像上面这样用循环来处理各种事件。只有在碰到 TERMINATE 事件时，才会从循环中跳出并终止应用程序；如果不是 TERMINATE，那就把它交给责任链。

```
handler2 = DebugHandler(handler1)
```

上面这行代码创建了一个新的处理程序（本来也可以把创建好的 DebugHandler 直接赋给 handler1）。由于 DebugHandler 要监听并报告责任链所收到的每个事件，所以它必须位于责任链开头，然而它不必处理这些事件，只需将其向下传递即可。

现在就可以把循环里的 handler1.handle(event) 改成 handler2.handle(event) 了，这样做不仅可以照常处理事件，而且还能把责任链所收到的事件当成调试信息打印出来。

```
class NullHandler:

    def __init__(self, successor=None):
        self.__successor = successor

    def handle(self, event):
        if self.__successor is not None:
            self.__successor.handle(event)
```

上面这个类是所有事件处理器的基类，它提供了处理事件所需的基础逻辑。如果创建实例时指定了后继者，那么实例收到事件之后，就会直接将其传给责任链中的下一个处理程序。若是没有指定后继者，则可将事件丢弃。这是编写 GUI (graphical user interface, 图形用户界面) 时的标准做法，但如果写的是服务器程序，那么也可以考虑将未处理的事件记录下来，或抛出异常。

```
class MouseHandler(NullHandler):
    def handle(self, event):
        if event.kind == Event.MOUSE:
            print("Click:  {}".format(event))
        else:
            super().handle(event)
```

由于子类没有重新实现 `__init__()` 方法，所以会继承由基类所实现的那一份，这使得我们可以正确创建出 `self.__successor` 变量。

这个 `MouseHandler` 类只会处理它所关注的那些事件（也就是 `Event.MOUSE` 事件），若遇到其他事件，则会将其传给责任链中的下一个处理程序（如果说有的话）。

`KeyHandler` 类及 `TimerHandler` 类（这两个类的代码没有列出来）的结构也和 `MouseHandler` 一样。它们与 `MouseHandler` 之间的区别仅在于响应的事件类型（`Event.KEYPRESS` 及 `Event.TIMER`）及处理事件的方式（也就是打印出来的消息）有所不同。

```
class DebugHandler(NullHandler):
    def __init__(self, successor=None, file=sys.stdout):
        super().__init__(successor)
        self.__file = file

    def handle(self, event):
        self.__file.write("*DEBUG*: {}\n".format(event))
        super().handle(event)
```

`DebugHandler` 类与其他 `handler` 类的区别在于，它本身并不处理事件，而且必须位于责任链开头。构建 `DebugHandler` 时，可传入文件对象或类似文件的其他对象，用以接收调试信息，发生事件时，`DebugHandler` 会将其汇报给 `file` 对象，并把事件传给责任链中的下个处理程序。

### 3.1.2 基于协程的责任链

生成器是个函数或方法，它通常不含有 `return` 语句，而是含有一个或多个 `yield` 表达式。只要遇到 `yield`，函数或方法就会产生值，然后在此处“挂起”（`suspend`），并保留其全部状态。这时函数会把计算机处理器的使用权交给值的接收方，所以它虽然挂起了，但并不会阻塞程序的执行过程。等到下次使用这个函数或方法时，它会从 `yield` 后面的那条语句开始执行。这样的话，只要用 `for value in generator:` 等语句来迭代生成器，或

在生成器上反复调用 `next()`，即可从其中提取（pull）值。

“协程”（coroutine）与生成器一样，也使用 `yield` 表达式，但行为却不同。协程执行的是无限循环，而且一开始就会停在首个（或仅有的那个）`yield` 表达式那里，等着有值传给它。协程会把收到的值当成 `yield` 表达式的值，然后继续执行它所需的操作，等处理完之后，又会再度循环，并在下一个 `yield` 表达式那里等着接收下个值。这样的话，我们就能反复调用协程的 `send()` 或 `throw()` 方法向其推送（push）值。

在 Python 中，凡是带有 `yield` 语句的函数或方法都能充当生成器。而利用 `@coroutine` 修饰器及无限循环则可将生成器变为协程。（2.4 节讲解了修饰器模式和 `@functools.wraps` 修饰器的用法。）

```
def coroutine(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        generator = function(*args, **kwargs)
        next(generator)
        return generator
    return wrapper
```

`wrapper` 只调用一次生成器函数，并把所得的生成器对象放在 `generator` 变量中。这个生成器对象其实就相当于原来的生成器函数，只不过它会把生成器函数的参数及函数中的局部变量保存成自己的状态。接下来，`wrapper` 调用一次 Python 语言内置的 `next()` 函数，令生成器对象前进到首个 `yield` 表达式。然后把生成器对象连同对象中所捕获的状态一并返回。而返回的生成器函数就是协程，它停在自己的首个（或仅有的那个）`yield` 表达式那里，等着接收值。

如果调用生成器，那么它会从上次挂起的地方（对于协程来说，也就是上次执行到的或仅有的那个 `yield` 表达式）继续向下执行。然而，如果用 Python 的 `generator.send(value)` 语法给协程发送值，那么协程会把收到的值视为当前这条 `yield` 表达式的结果，并继续从该点向下执行。

由于协程既能接收值又能发送值，所以可用来创建各种管道，比如处理事件所用的责任链就属于这种管道。此外，由于可以使用 Python 的生成器语法，所以无须再编写与后继者相关的那部分机制了。

```
pipeline = key_handler(mouse_handler(timer_handler()))
```

上面这行语句用几个嵌套的函数调用创建了名为 `pipeline` 的责任链。语句里所调用的这些函数都是协程，它们都已执行到自己的首个（或仅有的那个）`yield` 表达式那里，并处于挂起状态，等着再度使用或有值发过来。于是，这条管道立刻就能创建好，程序并不会在创建过程中阻塞。

与上一小节的做法不同，这次在构建责任链最后那个处理程序时没有传入参数。稍后我们将会以 `key_handler()` 为例来分析这些事件处理协程，那时大家就会明白这么做的道理了。

```

while True:
    event = Event.next()
    if event.kind == Event.TERMINATE:
        break
    pipeline.send(event)

```

与传统的写法一样，准备好责任链之后，就可以通过循环来处理各种事件了。由于每个 `handler` 函数都是协程（也就是生成器函数），所以它们都有 `send()` 方法。每次有事件需要处理时，就可以通过该方法把事件发送给管道。在本例中，第一个收到值的协程是 `key_handler()`，它有可能会处理此事件，也有可能会将其向下传递。和上一小节一样，在构建管道时，这些处理程序之间的顺序也是无关紧要的。

```
pipeline = debug_handler(pipeline)
```

上面这行代码用到了 `debug_handler`，它在管道中的顺序就比较重要了。由于 `debug_handler()` 协程要监听事件并将其向下传递，所以，它必须是责任链中的首个处理程序。配置好新的管道之后，就可以在循环中遍历事件了，我们调用 `pipeline.send(event)` 方法把每个事件都发给管道处理。

```

@coroutine
def key_handler(successor=None):
    while True:
        event = (yield)
        if event.kind == Event.KEYPRESS:
            print("Press: {}".format(event))
        elif successor is not None:
            successor.send(event)

```

上述协程的 `successor` 参数既可以是后继者协程，也可以是 `None`，此协程开始后，将会执行无限循环。由于 `@coroutine` 修饰器能够驱使 `key_handler()` 执行到它的首个 `yield` 表达式那里，所以当 `pipeline` 链创建好之后，这个函数就会在首个 `yield` 表达式处“阻塞”，等着有值传给它，以便把此值当作 `yield` 表达式的结果。（这里所谓的“阻塞”当然只是针对协程来说的，并不是说整个程序都阻塞了。）

只要有值传过来（无论是直接传来的，还是由管道中的另一个协程传来的），`event` 变量就能收到。如果这个事件可以由本协程来处理（也就是说，这个事件的类型是 `Event.KEYPRESS`），那么就处理它，在本例中，我们的处理方式很简单：把事件本身打印出来，并且不再向下传递。如果事件类型不在本协程处理范围内，并且责任链中还有后继者的话，那么就把事件传给后继者。若没有后继者，则不处理此事件，直接将其丢弃。

处理、发送或丢弃完事件之后，协程又返回 `while` 循环开头，然后停在 `yield` 那里，等着管道中有值发送过来。

`mouse_handler()` 与 `time_handler()` 协程（二者的代码都没有列出来）的结构和 `key_handler()` 完全相同，其区别仅在于能够处理的事件类型与打印出的消息不同而已。

```

@coroutine
def debug_handler(successor, file=sys.stdout):
    while True:
        event = (yield)
        file.write("*DEBUG*: {}\n".format(event))
        successor.send(event)

```

`debug_handler()` 等着接收事件，收到后，会把详情打印出来，然后将其传给下个协程处理。

尽管协程的机制与生成器相同，但用法却差别很大。使用一般的生成器时，我们是每次提取一个值出来，例如 `for x in range(10):`。而使用协程时，则要每次用 `send()` 推送一个值进去。如此灵活多变的用法说明 Python 语言能以非常清晰而自然的方式描述出许多种不同的算法来。比方说，本节所实现的这个基于协程的责任链就比上一小节的传统做法节省了大量代码。

3.5 节在讲解中介者模式时还会用到协程。

除了本节所举的用例之外，责任链模式当然也能在很多其他场合下使用。比方说，可以用该模式处理服务器所收到的请求。

## 3.2 命令模式

“命令模式”（Command Pattern）可把命令封装成对象。这样的话就可以构建命令序列，以便稍后执行，或是创建“可撤销的”（undoable）命令。2.7 节的 `ImageProxy` 范例已经初步使用了该模式，而本节将更进一步，用它来创建两个类，用以表示可撤销的命令及可撤销的宏（也就是可撤销的命令序列）。

我们先来看一段使用命令模式的代码，然后再讲解其中用到的 `UndoableGrid` 与 `Grid` 等类，以及提供“执行 – 撤销”与宏机制的 `Command` 模块。

```

grid = UndoableGrid(8, 3)      # (1) Empty
redLeft = grid.create_cell_command(2, 1, "red")
redRight = grid.create_cell_command(5, 0, "red")
redLeft()                      # (2) Do Red Cells
redRight.do()                  # OR: redRight()
greenLeft = grid.create_cell_command(2, 1, "lightgreen")
greenLeft()                     # (3) Do Green Cell
rectangleLeft = grid.create_rectangle_macro(1, 1, 2, 2, "lightblue")
rectangleRight = grid.create_rectangle_macro(5, 0, 6, 1, "lightblue")
rectangleLeft()                # (4) Do Blue Squares
rectangleRight.do()            # OR: rectangleRight()
rectangleLeft.undo()           # (5) Undo Left Blue Square
greenLeft.undo()               # (6) Undo Left Green Cell
rectangleRight.undo()          # (7) Undo Right Blue Square
redLeft.undo()                 # (8) Undo Red Cells
redRight.undo()

```

图 3.2 演示了 HTML 页面中的一张表格在八个阶段的样貌。第一幅小图是表格刚创建时

的样子（里面是空的）。后面每幅小图都是表格在每次创建并执行（无论是直接执行还是通过 do() 方法执行）命令及宏之后，或每次调用 undo() 之后的样貌。

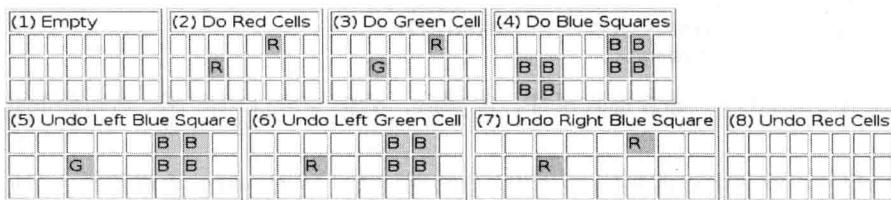


图 3.2 带执行及撤销功能的表格在各个阶段的样貌

```
class Grid:
    def __init__(self, width, height):
        self._cells = [[ "white" for _ in range(height)] 
                      for _ in range(width)]

    def cell(self, x, y, color=None):
        if color is None:
            return self._cells[x][y]
        self._cells[x][y] = color

    @property
    def rows(self):
        return len(self._cells[0])

    @property
    def columns(self):
        return len(self._cells)
```

Grid 类与 Image 类相似，含有保存颜色名称所用的二维列表。

cell() 方法既是“获取器”(getter)，又是“设置器”(setter)：当 color 参数为 None 时，它就是个获取器；而当 color 参数不为 None 时，则成了设置器。rows 与 columns 这两个只读属性分别返回表格的行数和列数。

```
class UndoableGrid(Grid):
    def create_cell_command(self, x, y, color):
        def undo():
            self.cell(x, y, undo.color)
        def do():
            undo.color = self.cell(x, y) # Subtle!
            self.cell(x, y, color)
        return Command.Command(do, undo, "Cell")
```

为了使 Grid 支持撤销功能，我们创建了子类，并向其中添加了两个方法，上面是第一个方法的代码。

绘制表格时所使用的每个命令都必须是 Command.Command 或 Command.Macro 类型的。构造前者时，必须指定两个 callable，用以表示执行及撤销操作，此外还可以传入一

一条描述信息。构造后者时，也可以指定描述信息，而且构造完之后，还可以把任意数量的 Command.Command 对象加入其中。

`create_cell_command()` 方法用 `x` 与 `y` 参数来表示待染色的单元格位置，用 `color` 参数表示颜色。我们在该方法里定义了 `undo()` 及 `do()` 函数，并用它们来创建 Command.Command 对象。这两个命令都很简单，只是为相应的单元格染色而已。

创建 `do()` 与 `undo()` 函数时，我们并不知道单元格在执行 `do()` 命令前的那一刻是什么颜色，所以无法确定 `undo()` 命令所用的颜色。要想解决这个问题，可以在 `do()` 函数执行的时候，先把单元格当前的颜色设置成 `undo()` 函数的 `attribute`，然后再用新的颜色来绘制。这么做是可行的，因为 `do()` 函数是个闭包，它不仅能把 `x`、`y`、`color` 等参数捕获成自己的状态，而且还能捕获到刚刚创建好的 `undo()` 函数。

创建好 `do()` 与 `undo()` 函数之后，我们用这两个函数及一条简单的描述信息来新建 Command.Command 命令，然后把该命令返回给调用者。

```
def create_rectangle_macro(self, x0, y0, x1, y1, color):
    macro = Command.Macro("Rectangle")
    for x in range(x0, x1 + 1):
        for y in range(y0, y1 + 1):
            macro.add(self.create_cell_command(x, y, color))
    return macro
```

上面列出了 `UndoableGrid` 类里另一个用于创建可执行 – 可撤销命令的方法。该方法所创建的宏可以填充由指定坐标所围成的矩形区域。对于每个要染色的单元格来说，我们都用本类的 `create_cell_command()` 方法创建一条与之对应的命令，并把它加到宏里面。全部命令都添加好之后，把宏返回给调用者。

稍后我们将会看到，普通命令与宏均支持 `do()` 及 `undo()` 方法。由于二者都支持同一套方法，而宏又可以包含普通命令，所以彼此间的关系也可说是一种组合模式（参见 2.3 节）。

```
class Command:
    def __init__(self, do, undo, description=""):
        assert callable(do) and callable(undo)
        self.do = do
        self.undo = undo
        self.description = description

    def __call__(self):
        self.do()
```

构建 `Command.Command` 对象时，必须指定两个 `callable`，前者用于“执行”(`do`) 命令，后者用于“撤销”(`undo`) 命令。`(callable())` 函数是从 Python 3.3 版本才开始内置的，在早前版本中，可用 `def callable(function): return isinstance(function, collections.Callable)` 来创等效函数。)

由于我们实现了名为 `__call__()` 的特殊方法，所以能够直接调用 `Command.Command` 对象，这么做与调用其 `do()` 方法等效。若想撤销此命令，则可调用 `undo()` 方法。

```

class Macro:

    def __init__(self, description=""):
        self.description = description
        self.__commands = []
    def add(self, command):
        if not isinstance(command, Command):
            raise TypeError("Expected object of type Command, got {}".
                            format(type(command).__name__))
        self.__commands.append(command)

    def __call__(self):
        for command in self.__commands:
            command()

do = __call__

def undo(self):
    for command in reversed(self.__commands):
        command.undo()

```

`Command.Macro` 类用来封装一系列命令，执行或撤销这个命令序列时，其中的所有命令将视为一个整体来操作<sup>⊖</sup>。`Command.Macro` 的接口与 `Command.Command` 相同，都有 `do()` 与 `undo()` 方法，而且也可以直接在后面加一对括号来执行。此外，宏还提供了 `add()` 方法，可用于向其中添加 `Command.Command` 对象。

撤销宏的时候，必须反向撤销其中的每条命令。比方说，我们把 A、B、C 这三条命令做成一个宏。执行宏的时候（可以直接调用，也可以通过 `do()` 方法来调用），最先执行的是 A 命令，其次是 B 命令，最后是 C 命令。所以宏的 `undo()` 方法应该逆序调用其中每个命令的 `undo()` 方法：先撤销 C 命令，然后撤销 B 命令，最后撤销 A 命令。

Python 中的函数、绑定方法以及其他 callable 都是“一级对象”（first-class object），它们既可以在程序之间传递，也可以保存在 `list` 及 `dict` 等数据结构中。这使我们很容易就能用 Python 语言实现命令模式。大家在这一节中已经看到：该模式非常适合用来实现“执行 - 撤销”功能，同时还能支持宏及“延迟执行”（deferred execution）等特性。

### 3.3 解释器模式

“解释器模式”（Interpreter Pattern）能够以规范的流程解决下面这两个常见的需求：向用户提供某种输入手段，使其可向应用程序中输入“非字符串值”（nonstring value），并允许用户给应用程序“编程”（program）。

以最基本的角度来看，应用程序从用户或其他程序那里收到字符串之后，必须将其合理

---

<sup>⊖</sup> 尽管我们将宏里的全部命令都视为一个整体来操作，但从并发角度来看，该操作并不是“原子的”（atomic）。适当加锁之后，可将其变为原子操作。

地解释出来（还可能会执行字符串中的命令）。假如我们想从用户输入中接收表示整数的字符串，那么最简单的办法就是用 `i = eval(userCount)` 语句来获取整数值，但这么做很不明智，因为用户未必都会按照我们所想的输入像 "1234" 这种正常的字符串，若是输入 "`os.system('rmdir /s /q C:\\\\')`"，那可就危险了。

如果字符串专门用于表示某种特定类型的数据，那么通常可以直接调用 Python 的相关函数来安全地获取其值。

```
try:
    count = int(userCount)
    when = datetime.datetime.strptime(userDate, "%Y/%m/%d").date()
except ValueError as err:
    print(err)
```

上述代码片段用 Python 中的特定函数安全地解析了两个字符串，一个解析为 `int`，另一个解析为 `datetime.date`。

有些时候，要解析的字符串不只一个。比方说，我们想在应用程序中加入计算器功能，或是允许用户自己编写代码来操作应用程序里的数据。要想实现此类需求，一种常见的办法是创建一门“领域特定语言”（Domain Specific Language, DSL）。这种语言直接用 Python 就能创建出来，比方说，我们可以自己编写“递归下降解析器”（recursive descent parser）。然而用 PLY ([www.dabeaz.com/ply](http://www.dabeaz.com/ply))、PyParsing ([pyparsing.wikispaces.com](http://pyparsing.wikispaces.com)) 这样的第三方解析库来做会简单许多，许多程序库都能实现解析功能<sup>①</sup>。

如果我们信得过应用程序的用户，那么可以直接把 Python 解释器交给他们使用。Python 自带的 IDLE “集成开发环境”（Integrated Development Environment, IDE）用的正是这种方法，不过 IDLE 做得比较智能一些：用户所输入的代码会放在另一个线程中执行，即便崩溃了，也不会影响到 IDLE 自身。

### 3.3.1 用 eval() 函数求表达式的值

Python 内置的 `eval()` 函数会把字符串当成表达式来求值（我们可以提供全局环境及局部环境供这个表达式访问），并返回求值结果。本节要讲的 `calculator.py` 程序很简单，完全可以用 `eval()` 函数来做。首先我们看看用户如何与这个计算器程序交流<sup>②</sup>。

```
$ ./calculator.py
Enter an expression (Ctrl+D to quit): 65
A=65
ANS=65
Enter an expression (Ctrl+D to quit): 72
A=65, B=72
```

- 
- ① 笔者所著的《Programming in Python 3, Second Edition》一书讲述了解析问题，也讲了 PLY 及 PyParsing 库的用法，该书详情参见附录 B。
  - ② 将本小节的 4 段代码放在一个文件里，头部加上 `#!/usr/bin/env python3` 及 `import sys; import types; import collections` 这两行代码，尾部加上 `if __name__ == "__main__": main()`，即可构成 `calculator.py` 程序。——译者注

```

ANS=72
Enter an expression (Ctrl+D to quit): hypotenuse(A, B)
name 'hypotenuse' is not defined
Enter an expression (Ctrl+D to quit): hypot(A, B)
A=65, B=72, C=97.0
ANS=97.0
Enter an expression (Ctrl+D to quit): ^D

```

用户输入直角三角形的两个直角边，然后用 `math.hypot()` 函数计算斜边（第一次使用函数时，把函数名输错了）。每输入一条表达式，`calculator.py` 程序就会把当前已经创建好的所有变量打印出来（用户也可以使用这些变量），并把表达式的值告诉用户。（用户输入的文本以粗体印刷，我们假定用户输入每行文本后都会按下 Enter 或 Return 键，`^D` 表示按下 Ctrl+D 组合键。）

为了使计算器用起来方便一些，我们把每条表达式的求值结果都保存在一个变量里，变量名从 A 开始，接下来是 B，依次向后，到达 Z 之后，又折回 A。此外，我们还把 `math` 模块中的所有函数及常量（比如 `hypot()`、`e`、`pi`、`sin()` 等）都引入计算器程序的命名空间，使用户无须修饰其名称即可直接访问（比方说，可以直接使用 `cos()` 函数，而不用写成 `math.cos()`）。）

如果用户输入的字符串无法求值，那么计算器就打印一条错误信息，然后继续提示用户输入字符串，而运行“环境”（context）中已有的内容则保持不变。

```

def main():
    quit = "Ctrl+Z,Enter" if sys.platform.startswith("win") else "Ctrl+D"
    prompt = "Enter an expression {{} to quit): ".format(quit)
    current = types.SimpleNamespace(letter="A")
    globalContext = global_context()
    localContext = collections.OrderedDict()
    while True:
        try:
            expression = input(prompt)
            if expression:
                calculate(expression, globalContext, localContext, current)
        except EOFError:
            print()
            break

```

我们用 EOF（End of File，文件结尾）来表示用户输入已结束。这样做意味着计算器不仅可以接受交互式的用户输入，而且还能在“命令行界面”（shell）中充当管道，把文件里的内容当成由用户所输入的文本。

我们还需要记住当前变量的名称（比如 A、B，等等），以便每次计算完表达式之后更新它。然而在求值的时候，不能简单地把这个名称当成字符串传进去，因为字符串只能拷贝，无法修改。笨办法是使用全局变量。稍好一些（而且比较常用）的办法是创建只含一个元素的列表，比如 `current = ["A"]`，然后把这个列表当成 `current` 参数传给 `calculate()` 函数，这样就可以通过 `current[0]` 来读取或修改其中的字符串了。

本例采用一种更为先进的办法，就是创建一个微型命名空间，其中只有名为 `letter` 的 attribute，它的值是 "A"。`current` 这个简单的命名空间实例可以在程序中自由传递，而且

因为它有名为 `letter` 的 attribute，所以我们能够通过 `current.letter` 这种写法来读取或修改 `letter` 的值。

`types.SimpleNamespace` 类是从 Python 3.3 版本才开始有的。在早前版本中，可以通过 `current = type("_", (), dict(letter="A"))()` 来实现相同的效果。这行语句创建了名为 `_` 的类，该类只有一个名叫 `letter` 的 attribute，其值为 "A"。在调用 Python 内置的 `type()` 函数时，如果只传入一个参数，那么会返回参数对象的类型；若是传入三个参数，则会创建新的类，第一个参数是类名，第二个参数是元组，用来描述基类，第三个参数是包含 attribute 的字典。如果元组为空，那么新类的基类就是 `object`。由于我们并不需要这个类，而只是想使用它的一个实例，所以在调用完 `type()` 函数之后，又加了一对括号，直接创建出新类的实例，并将其赋给 `current`。

Python 内置的 `globals()` 函数可以返回当前的全局环境<sup>Θ</sup>，该函数返回的 `dict` 是可以修改的（比方说，可以像 1.3 节的 `gameboard4.py` 那样，向其中添加 attribute）。Python 还有个内置函数叫做 `locals()`，可用来查询局部环境<sup>⊖</sup>，但是开发者绝不应该修改由这个函数所返回的 `dict`。

我们想把 `math` 模块的常量及函数放在全局环境里，并且想令局部环境在程序刚启动时保持空白。评估表达式的值时，所提供的全局环境必须用 `dict` 来表示，但局部环境未必非得是 `dict`，只要是个“映射对象”(mapping object) 就行。于是，我们采用 `collections.OrderedDict` 作为局部环境，这种字典会保留元素的插入顺序。

由于计算器要接受交互式输入，所以我们创建了事件循环，该循环在遇到 EOF 时会终止。在循环里，我们提示用户输入待求值的表达式（并告诉他们如何退出程序），如果用户输入了文本，那么就调用 `calculate()` 函数，计算表达式的值，并打印结果。

```
import math

def global_context():
    globalContext = globals().copy()
    for name in dir(math):
        if not name.startswith("_"):
            globalContext[name] = getattr(math, name)
    return globalContext
```

上面这个辅助函数首先把程序的全局模块、全局函数及全局变量以“浅拷贝”(shallow-copy) 的方式复制到局部的 `dict` 里，然后遍历 `math` 模块中的所有公开常量及公开函数，把每一个未修饰的名称都当作键添加到名为 `globalContext` 的 `dict` 里，并将其值设为 `math` 模块中与该名称相对应的常量或方法。比方说，如果 `name` 是 "factorial"，那么加入 `globalContext` 里的键就是这个 `name`，而值则指向 `math.factorial()` 函数的引用。经过这番处理之后，用户就可以通过未修饰的名称来使用那些常量及函数了。

还有个更简单的办法，就是用 `from math import *` 来引入常量及模块，并直接把 `globals()` 当成 `global_context()` 的返回值，这样就不需要名为 `globalContext`

<sup>Θ</sup> global context，也称“全局上下文”，指的是“全局符号表”(global symbol table)。——译者注

<sup>⊖</sup> local context，也称“局部上下文”，指的是“局部符号表”(local symbol table)。——译者注

的 dict 了。对于 math 模块来说，这种做法或许没问题，但我们刚才的那种做法更为精细，所以对其他模块来说，可能更合适一些。

```
def calculate(expression, globalContext, localContext, current):
    try:
        result = eval(expression, globalContext, localContext)
        update(localContext, result, current)
        print(", ".join(["{}={}"].format(variable, value)
                      for variable, value in localContext.items())))
        print("ANS={}".format(result))
    except Exception as err:
        print(err)
```

上面这个函数通过 Python 内置的 eval() 函数把用户输入的字符串当成表达式来求值，调用 eval() 的时候，我们会把刚才创建的全局环境及局部环境这两个字典传进去。如果 eval() 顺利执行完，那就根据求值结果来更新局部环境，并打印出局部环境中的所有变量及表达式的求值结果。若发生异常，则将其打印出来——这样做是安全的。由于我们用 collections.OrderedDict 来表示局部环境，所以 items() 方法所返回的元素的先后顺序与插入时的顺序相同，无须手工排列。（假如当时使用普通的 dict，那么就不能直接调用 items() 了，而要写成 sorted(localContext.items())。）

我们一般不应该用 Exception 来捕捉所有类型的异常，这样太过宽泛了，但在本例中，这么做似乎是比较合适的，因为用户所输入的表达式在求值的时候什么类型的异常都可能发生。

```
def update(localContext, result, current):
    localContext[current.letter] = result
    current.letter = chr(ord(current.letter) + 1)
    if current.letter > "Z": # We only support 26 variables
        current.letter = "A"
```

上面这个函数会把表达式的求值结果赋给当前的变量，并准备用循环序列 (A...Z A...Z ...) 中的下一个变量来保存下个表达式的值。这就意味着用户输入完 26 个表达式之后，最末那个表达式的值将赋给 Z，而下一个表达式的求值结果将把变量 A 中原有的值覆盖掉，依此类推。

eval() 函数可以对任意 Python 表达式求值。如果表达式的来源不可靠，那么就会有安全隐患。标准库里还有另外一个功能相似但更为安全的函数，名叫 ast.literal\_eval()，此函数对表达式的限制比 eval() 严格。

### 3.3.2 用 exec() 函数执行代码

Python 内置的 exec() 函数可以随意执行 Python 代码。与 eval() 不同，exec() 并不局限于一条表达式，并且此函数总是返回 None。两者的相同之处在于调用者都能够把全局环境与局部环境放在字典里传进去。exec() 的执行结果可以从传给它的局部环境里获取。

本节以 genome1.py 程序为例讲解 exec() 的用法。该程序创建了名为 genome 的变量（它是个由 A、C、G、T 四个字母随机排列而成的字符串），并将八段用户代码放在给定的环境下执行。

```
context = dict(genome=genome, target="G[AC]{2}TT", replace="TCGA")
execute(code, context)
```

上面这两行语句首先创建了名为 context 的字典，将用户代码所需的某些数据存入其中，然后把用户代码对象（以 code 变量表示，该变量指向 Code 对象）放在刚创建好的 context 下执行。

```
TRANSFORM, SUMMARIZE = ("TRANSFORM", "SUMMARIZE")
Code = collections.namedtuple("Code", "name code kind")
```

我们想令用户把自编的代码放在 Code 这种“具名元组”（named tuple）里面，构建元组对象时，需要提供一段描述信息以指明此段代码的用意，还要提供待执行的代码片段（以字符串表示）以及代码片段的种类（TRANSFORM 或 SUMMARIZE）。在执行时，用户自编的代码应该会创建用于保存执行结果的 result 对象，或是用于保存错误信息的 error 对象。如果代码片段的种类为 TRANSFORM，那么 result 就应该是个字符串，用来表示新的“基因组”（genome）。若种类为 SUMMARIZE，则 result 应该是个数字。当然了，程序本身也要写得健壮些才行，即使用户提供了不符合上述要求的代码，程序也不应该出错。

```
def execute(code, context):
    try:
        exec(code.code, globals(), context)
        result = context.get("result")
        error = context.get("error")
        handle_result(code, result, error)
    except Exception as err:
        print("{}' raised an exception: {}\\n".format(code.name, err))
```

上面这个函数首先调用 exec() 来执行用户自编的代码，执行代码时，全局环境由 genome1.py 程序提供，而局部环境则由 context 变量提供。然后试着获取 result 及 error 对象（用户自编的代码应该会创建这两个对象中的一个），获取到之后，把它们传给 genome1.py 程序的 handle\_result() 函数。

因为用户自编的代码可能会引发各种异常，所以与上一小节的 eval() 范例一样，我们在捕获异常时也用到了宽泛的 Exception 类型（通常情况下不应该这么做）。

```
def handle_result(code, result, error):
    if error is not None:
        print("{}' error: {}".format(code.name, error))
    elif result is None:
        print("{}' produced no result".format(code.name))
    elif code.kind == TRANSFORM:
        genome = result
        try:
            print("{}' produced a genome of length {}".format(code.name,
                len(genome)))
        except TypeError as err:
            print("{}' error: expected a sequence result: {}".format(
                code.name, err))
    elif code.kind == SUMMARIZE:
```

```
    print("'{}' produced a result of {}".format(code.name, result))
print()
```

如果 `error` 对象不是 `None`, 那么就把它打印出来。如果 `error` 是 `None`, 而 `result` 也是 `None`, 那么就打印“produced no result”(未产生结果)。如果 `result` 不是 `None`, 并且用户自编代码的种类为 `TRANSFORM`, 那么就把 `result` 赋给 `genome`, 并打印出 `genome` 的新长度。为了保护程序本身, 我们把这行打印语句放在 `try...except` 块里执行, 这样的话, 即便用户代码所返回的 `result` 不符合要求(例如在执行 `TRANSFORM` 操作后, 只返回了单一值, 而没有返回字符串或其他序列), `genome1.py` 程序也不会出错。如果 `result` 的类型是 `SUMMARIZE`, 则仅仅打出包含结果的小计行。

`genome1.py` 程序将会执行八段用户自编的代码(每段代码都放在一个 `Code` 对象里)。前两段代码(稍后就会列出来)都能产生适当的结果, 第三段代码写法有误, 第四段代码本身会通过 `error` 变量来汇报错误, 第五段代码什么事也不做, 第六段代码所返回的结果与代码种类不符, 第七段代码调用了 `sys.exit()`, 而第八段代码根本就不会执行, 因为第七段代码将使整个程序终止。下面是程序所输出的信息。

```
$ ./genome1.py
'Count' produced a result of 12
'Replace' produced a genome of length 2394
'Exception Test' raised an exception: invalid syntax (<string>, line 4)
'Error Test' error: 'G[AC]{2}TT' not found
'No Result Test' produced no result
'Wrong Kind Test' error: expected a sequence result: object of type 'int' has
no len()
```

由于用户自编的代码与 `genome1.py` 程序都在同一个解释器里执行, 所以用户代码也可以令程序终止或崩溃, 上面这段输出信息已经清楚地说明了这一点。(印刷时为了符合书页宽度, 最后一条输出信息占用了两行。)

```
Code("Count",
"""
import re
matches = re.findall(target, genome)
if matches:
    result = len(matches)
else:
    error = "{} not found".format(target)
""", SUMMARIZE)
```

上面这个 `Code` 对象就是用户自编的第一段代码, 这段代码的名字叫做“Count”。`eval()` 函数每次只能求一个表达式的值, 而上面这段代码的内容无法写在一条表达式里。`target` 与 `genome` 字符串的值都是从 `context` 对象里获取的, 而该对象就是由 `execute()` 函数传给 `exec()` 函数的局部环境。用户代码所创建的新变量(比如 `result` 及 `error`)也会自动储存到这个 `context` 对象里。

```

Code("Replace",
"""
import re
result, count = re.subn(target, replace, genome)
if not count:
    error = "no '{}' replacements made".format(target)
"""", TRANSFORM)

```

上面这个 Code 对象里存放的代码片段名字叫做“Replace”，它会对 genome 字符串执行简单的“转化”(transformation)操作，用 replace 变量中的字符串来替换每一个符合 target 正则表达式的子字符串，待替换的子字符串之间不会互相重叠。

re.subn 函数(及 regex.subn() 方法)所执行的替换操作与 re.sub() 函数(及 regex.sub() 方法)相同。但是 sub() 函数(及 regex.sub() 方法)只会把替换后的字符串返回，而 subn() 函数(及 regex.subn() 方法)不仅会把字符串返回，还会把替换掉的子字符串个数也告诉调用者。

尽管 execute() 函数与 handle\_result() 函数写得既好用又好懂，但是 genome1.py 程序依然有个弱点：如果用户自编的代码崩溃了，或是调用了 sys.exit() 函数，那么我们的程序也就终止了。下一小节将设法解决此问题。

### 3.3.3 用子进程执行代码

有种办法可以在不损害应用程序的情况下执行用户代码，那就是把它放在单独的进程里执行。本节的 genome2.py 及 genome3.py 程序将演示如何在子进程中执行 Python 解释器，这两个程序会把用户自编的代码通过“标准输入”(standard input)发给解释器来执行，并通过读取“标准输出”(standard output)来获知执行结果。

genome2.py 与 genome3.py 所执行的八段代码与 genome1.py 完全一致。下面是 genome2.py 所输出的信息 (genome3.py 的输出信息与之相同)：

```

$ ./genome2.py
'Count' produced a result of 12
'Replace' produced a genome of length 2394
'Exception Test' has an error on line 3.
    if genome[i] = "A":
        ^
SyntaxError: invalid syntax
'Error Test' error: 'G[AC]{2}TT' not found
'No Result Test' produced no result
'Wrong Kind Test' error: expected a sequence result: object of type 'int' has
no len()
'Termination Test' produced no result
'Length' produced a result of 2406

```

请注意，尽管第七个 Code 对象中的代码调用了 `sys.exit()`，但 `genome2.py` 程序却不受影响，它只会报告这段代码“未产生结果”（produced no result），并继续执行下一段名为“Length”的代码（由于 `sys.exit()` 会使 `genome1.py` 终止，所以那个程序的输出信息到“...error: expected a sequence...”这一行就没有了。）。还要注意的是，`genome2.py` 所报告的错误信息比 `genome1.py` 更有意义（例如，它可以指出“Exception Test”这段代码到底哪里写错了）。

```
context = dict(genome=genome, target="G[AC]{2}TT", replace="TCGA")
execute(code, context)
```

上面这两行代码与 `genome1.py` 完全相同；也是先创建 `context` 变量，然后把用户自编的代码放在这个 `context` 下执行。

```
def execute(code, context):
    module, offset = create_module(code.code, context)
    with subprocess.Popen([sys.executable, "-"], stdin=subprocess.PIPE,
                          stdout=subprocess.PIPE, stderr=subprocess.PIPE) as process:
        communicate(process, code, module, offset)
```

上面这个函数首先把用户自编的代码以及一些辅助代码放在名为 `module` 的字符串里。`offset` 变量表示我们在用户自编的代码前面添加了多少行附加代码，知道这个值之后，就可以把错误信息中的行数写得更准确了。然后，这个函数会启动子进程，并在其中执行新的 Python 解释器实例，这个解释器所对应的可执行文件的路径保存在 `sys.executable` 里面。执行这个解释器时，所传入的参数是“-”（连字符），这表示该解释器会从 `sys.stdin` 中读出 Python 代码并执行，而我们的 `genome2.py` 程序正是要把用户自编的代码发给解释器的 `sys.stdin`<sup>Θ</sup>。最后，我们用 `communicate()` 函数同新进程交流，该函数会把 `module` 中的代码发过去。

```
def create_module(code, context):
    lines = ["import json", "result = error = None"]
    for key, value in context.items():
        lines.append("{} = {!r}".format(key, value))
    offset = len(lines) + 1
    outputLine = "\nprint(json.dumps((result, error)))"
    return "\n".join(lines) + "\n" + code + outputLine, offset
```

上面这个函数先把两行代码放在一份列表中，这两行代码会与用户自编的代码一同构成新的 Python 模块，而子进程里的 Python 解释器将会执行这个模块。列表里的首行代码会引入 `json` 模块，因为子进程要通过该模块向“起始进程”（initiating process，也就是 `genome2.py` 程序所在的进程）汇报执行结果。列表中的第二行代码会初始化 `result` 及 `error` 变量，以保证这两个变量肯定会展现在子进程所要执行的程序里。然后，我们把 `context` 字典里的每个变量都转化成一条赋值语句。执行完用户自编的代码之后，把 `result` 及 `error` 保

---

<sup>Θ</sup> 从 Python 3.2 版本开始，`subprocess.Popen()` 函数已经可以支持环境管理器了（也就是说，我们可以把此函数写在 `with` 语句里面了）。

存在 JSON (JavaScript Object Notation) 格式的字符串里，并将其打印到 `sys.stdout`。

```
UTF8 = "utf-8"

def communicate(process, code, module, offset):
    stdout, stderr = process.communicate(module.encode(UTF8))
    if stderr:
        stderr = stderr.decode(UTF8).lstrip().replace(", in <module>", ":")
        stderr = re.sub(", line (\d+)", lambda match: str(int(match.group(1)) - offset), stderr)
        print(re.sub(r'File."[^"]+?"', "'{}' has an error on line ".format(code.name), stderr))
    return
    if stdout:
        result, error = json.loads(stdout.decode(UTF8))
        handle_result(code, result, error)
    return
print("{} produced no result\n".format(code.name))
```

`communicate()` 函数首先把早前制作好的模块代码发给子进程中的 Python 解释器，令其执行模块里的代码，并等待执行结果。解释器执行完代码后，我们通过局部变量 `stdout` 及 `stderr` 来收集子进程的标准输出及标准错误输出。请注意，由于通信过程要使用“原始字节”(raw byte) 来表示数据，所以我们必须把 `module` 字符串按 UTF-8 标准编码成字节组。

如果子进程输出了错误信息（也就是说，其中的程序抛出了异常，或是 `sys.stderr` 里面写进了某些内容），那么就把信息里原有的行号（这个序号会把我们加在用户代码前面的那些辅助代码也算上）替换为错误语句在用户自编代码中的实际行号，并把“`File "<stdin>"` 替换成 `Code` 对象的名称。最后把替换过的错误消息当成字符串打印出来。

`re.sub()` 函数的第一个参数及参数中的 `(\d+)` 用于匹配并捕获行号，而第二个参数则是个 `Lambda` 函数，`re.sub()` 会逐次调用这个 `lambda`，并以所得的结果来分别替换匹配到的那些行号。（第二个参数通常是字符串，但由于此处需要计算，因此我们使用了 `lambda` 函数。）`lambda` 函数会把表示行号的数字转换成整数，并从中减去偏移量，然后把新的行号当成字符串返回，这样的话，`re.sub()` 函数就可以用它来替换原来的行号了。因为我们在创建模块并将其传给解释器执行之前曾给用户代码上面添加了一些辅助代码，所以错误信息中的行号会发生偏差，而按照上述方式替换之后，无论添加多少行辅助代码，新的行号总能与错误信息行号在用户代码中的真实位置相符。

如果 `stderr` 里面没有内容，但 `stdout` 里面却有内容，那就把 `stdout` 中的字节组解码为字符串（这个字符串应该是 JSON 格式的），并将其解析为 Python 对象，在本例中，字符串会解析为二元组，它的两个元素分别为 `result` 及 `error`。最后，调用 `handle_result()` 函数。（此函数在 `genome1.py`、`genome2.py` 及 `genome3.py` 中均相同，其代码写在上一小节里。）

`genome2.py` 程序所执行的用户代码与 `genome1.py` 相同，但是它会在用户代码前后加上一些辅助代码。用 JSON 格式来返回执行结果是比较安全的，而且也很方便，但是所

能返回的数据类型（比如 `result` 的类型）却仅局限于 `dict`、`list`、`str`、`int`、`float`、`bool` 及 `None` 这几种，`dict` 或 `list` 中的对象类型也只能是这几种。

`genome3.py` 程序与 `genome2.py` 很相似，但它是用“序列化”（`pickle`）形式来表示 `result` 的。这使得用户代码能够返回大部分 Python 类型。

```
def create_module(code, context):
    lines = ["import pickle", "import sys", "result = error = None"]
    for key, value in context.items():
        lines.append("{} = {!r}".format(key, value))
    offset = len(lines) + 1
    outputLine = "\n".join(lines) + "\n" + code + outputLine
    return "\n".join(lines) + "\n" + code + outputLine, offset
```

上面这个函数与 `genome2.py` 中的那个非常接近，但有个小区别，就是它必须引入 `sys` 模块。两者之间的主要差别在于，`genome2.py` 中的函数通过 `json` 模块的 `loads()` 与 `dumps()` 方法来操作 `str`，但 `pickle` 模块中的对应函数所操作的却是 `bytes`。所以我们必须把原始字节数据直接写入 `sys.stdout` 的底层缓冲区里，否则这些字节可能会输出成错误的内容。

```
def communicate(process, code, module, offset):
    stdout, stderr = process.communicate(module.encode(UTF8))
    ...
    if stdout:
        result, error = pickle.loads(stdout)
        handle_result(code, result, error)
    return
```

`genome3.py` 程序的 `communicate()` 方法与 `genome2.py` 的相同，只是调用 `loads()` 方法的那行代码不一样。对于 `genome2.py` 所使用的 JSON 数据来说，我们必须把字节按 UTF-8 标准解码成 `str`，而 `genome3.py` 则不用这样做，它只需直接使用原始字节即可。

`exec()` 可用来执行从用户或其他程序那里获取到的任意 Python 代码，这些代码可以使用 Python 解释器的全部功能，而且也能调用整套标准库。如果把 Python 解释器放在单独的子进程里执行，那么即使用户代码出了问题，程序本身也不会因此而崩溃或终止。但这并不能完全阻止用户代码中的恶意行为。若要执行不受信任的代码，则需使用某种“沙盒”（`sandbox`）机制，比方说，PyPy Python 解释器（[pypy.org](http://pypy.org)）就提供了这样的沙盒。

某些程序可以像本例这样，等用户执行完之后再向下运行，但如果用户代码里面有 bug（比如无限循环），那么程序可能就会一直阻塞下去。有个办法能解决此问题，就是在单独的线程里创建子进程，然后在主线程里使用计时器。若子进程超时，则强行将其终止，并把该问题告诉用户。并发编程将在下一章里介绍。

### 3.4 迭代器模式

“迭代器模式”（Iterator Pattern）可按顺序访问“集合”（collection）或“聚合”（aggregate）

对象中的元素，同时又无须关注集合或聚合内部的实现细节。此模式很有用，Python 语言为其提供了内置支持，而且只要实现某些特殊方法，就可以使我们自编的类也具备迭代功能。

实现此模式有三种办法：可以遵从“序列协议”（sequence protocol），也可以使用 Python 内置的双参数 `iter()` 函数，还可以遵从“迭代器协议”（iterator protocol）。下面三个小节将举例说明这些用法。

### 3.4.1 通过序列协议实现迭代器

实现迭代器模式的一种办法是令自己所编写的类支持序列协议，这意味着必须实现名叫 `__getitem__()` 的特殊方法，此方法可接受整数作下标参数，这个下标从 0 开始，如果不能再继续迭代了，那么应该抛出 `IndexError` 异常。

```
for letter in AtoZ():
    print(letter, end="")
print()

for letter in iter(AtoZ()):
    print(letter, end="")
print()

ABCDEFIGHIJKLMNOPQRSTUVWXYZ
ABCDEFIGHIJKLMNOPQRSTUVWXYZ
```

上面这两段代码都创建了 `AtoZ` 对象，并对其迭代。该对象初次迭代时会返回只含字母 "A" 的字符串，下次迭代时会返回只含字母 "B" 的字符串，依此类推，直到 "Z"。有许多办法都能使 `AtoZ` 对象支持迭代功能，而我们在这里采用的办法是实现 `__getitem__()` 方法，大家马上就能看到该方法的代码。

第二个循环通过内置的 `iter()` 函数来获取针对 `AtoZ` 类实例的迭代器。本例显然无须这么做，但我们稍后就会知道，有些时候 `iter()` 函数其实是有用的（本书中的其他例子也能说明这一点）。

```
class AtoZ:

    def __getitem__(self, index):
        if 0 <= index < 26:
            return chr(index + ord("A"))
        raise IndexError()
```

上面就是 `AtoZ` 类的全部代码。因为我们提供了 `__getitem__()` 方法，所以这个类能够满足序列协议的要求。该类的对象迭代到第 27 轮时，会抛出 `IndexError` 异常。此异常如果是在 `for` 循环中抛出的，那么 Python 会直接将其丢弃，而循环则可以正常终止，于是程序就能够从循环后面的那条语句继续往下执行了。

### 3.4.2 通过双参数 `iter()` 函数实现迭代器

还有个实现迭代器的办法是使用内置的 `iter()` 函数，但要传入两个参数，而不是

像平常那样只传一个。采用这种形式的时候，首个参数必须是 callable（函数、绑定方法或其他 callable 对象），第二个参数必须是个“标记值”（sentinel value）。使用此方式制作而成的迭代器每次迭代都会调用 callable 对象（调用时不传参数），只有当 callable 抛出 StopIteration 异常或返回标记值的时候，才停止迭代。

```
for president in iter(Presidents("George Bush"), None):
    print(president, end=" * ")
print()

for president in iter(Presidents("George Bush"), "George W. Bush"):
    print(president, end=" * ")
print()

George Bush * Bill Clinton * George W. Bush * Barack Obama *
George Bush * Bill Clinton *
```

调用 Presidents() 会创建出 Presidents 类的实例，因为该类实现了名叫 \_\_call\_\_() 的特殊方法，所以这种实例可视为 callable。于是刚创建出来的 Presidents 对象就是个 callable（调用 Python 内置的双参数 iter() 函数时，首参数必须是 callable），第一个 for 循环在迭代时所用的标记值是 None。尽管是 None，但也必须明确指定该值，这样的话，Python 才能知道我们要调用的是双参数 iter() 函数，而非单参数版本。

Presidents 构造器所创建的 callable 对象会依次返回每位美国总统的名字，默认是从 George Washington（乔治·华盛顿）开始，另外，也可以从我们所指定的名字开始。本例从 George Bush（乔治·布什）开始。第一个 for 循环在迭代时以 None 为标记值，于是就会一直打印到“最后一个”名字，也就是时任总统的名字：Barack Obama（巴拉克·奥巴马）。由于第二个 for 循环在迭代时手工指定了标记值，因此 callable 会从 George Bush 开始，一直打印到该标记之前的那个名字。

```
class Presidents:
    __names = ("George Washington", "John Adams", "Thomas Jefferson",
              ...
              "Bill Clinton", "George W. Bush", "Barack Obama")

    def __init__(self, first=None):
        self.index = (-1 if first is None else
                      Presidents.__names.index(first) - 1)

    def __call__(self):
        self.index += 1
        if self.index < len(Presidents.__names):
            return Presidents.__names[self.index]
        raise StopIteration()
```

Presidents 类会把每位美国总统的名字都放在名为 \_\_names 的静态（也就是类级别的）列表里。如果用户没有指定 first 参数，那么 \_\_init\_\_() 方法会把初始下标设为 -1；若指定了该参数，则会找到与总统名字相对应的下标，将其减一，并用作初始下标。

凡是实现了 `__call__()` 特殊方法的类的实例都可以当成 callable 来用。Python 在调用这种 callable 实例时，实际上执行的是其 `__call__()` 方法<sup>①</sup>。

`Presidents` 类的 `__call__()` 特殊方法可能会返回列表中下一位总统的名字，也可能抛出 `StopIteration` 异常。第一个 `for` 循环所指定的标记值是 `None`，而 `__call__()` 方法又不可能返回 `None`，所以永远都到不了这个标记，但是迭代过程仍然能结束，因为 `__call__()` 方法在用完了所有总统名字之后，会抛出 `StopIteration` 异常，从而使循环终止。而在第二个 `for` 循环中，只要 `__call__()` 方法所返回的总统名字同标记值相符，那么 `iter()` 函数就会抛出 `StopIteration` 异常，于是循环也就终止了。

### 3.4.3 通过迭代器协议实现迭代器

要想为自编的类提供迭代器，最简单的办法也许就是令其支持“迭代器协议”(`iterator protocol`)了。该协议要求类必须有名为 `__iter__()` 的特殊方法，而且该方法要返回迭代器对象。迭代器对象的 `__iter__()` 方法必须返回迭代器自身，而 `__next__()` 方法则返回下一个元素，如果没有元素可供迭代，则抛出 `StopIteration` 异常。Python 的 `for` 循环与 `in` 语句在底层都会使用这个协议。而要实现 `__iter__()` 方法，最容易的方式则是令其成为生成器，或令其返回生成器，因为生成器本身就符合迭代器协议。(3.1.2 节详细讲解了生成器。)

本节将创建一个简单的“包”(`bag`)类，这样的类也叫做“多重集”(`multiset`)。包是一种“集合”(`collection`)，它与 `set` 类似，但其中可容纳重复的元素。图 3.3 演示了包和包里的元素。我们要为这个包增加迭代功能，而且要用三种方式来实现。除另有说明者外，本节范例代码均节选自 `Bag1.py`。

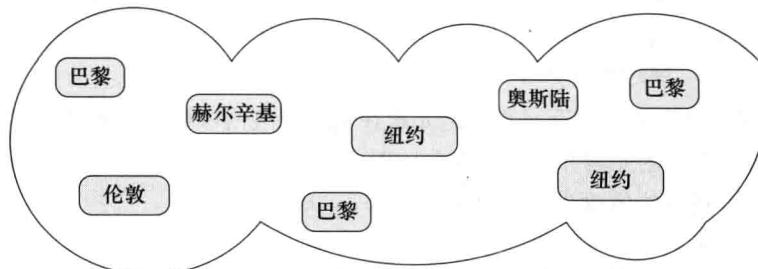


图 3.3 包是能容纳重复元素的“无序集合”(unsorted collection)

```
class Bag:
    def __init__(self, items=None):
        self._bag = {}
        if items is not None:
            for item in items:
                self.add(item)
```

<sup>①</sup> 有些编程语言无法把函数当成“一级对象”来用，这些语言把 callable 实例叫做 functor(函数)。

包中的数据存放在 `self.__bag` 字典里。任何“hashable”(可求出其哈希码的)对象都能充当字典的键(这些键就是包中的元素),而字典的值是个数字,用于表示该元素在包中出现的次数。新建包的时候,也可以指定初始元素。

```
def add(self, item):
    self.__bag[item] = self.__bag.get(item, 0) + 1
```

因为 `self.__bag` 并不是 `collections.defaultdict`, 所以 `add()` 方法只有在确认了包里已经含有 `item` 元素之后, 才应该递增计数器。这一点必须谨慎处理, 以防发生 `KeyError` 异常。我们用 `dict.get()` 方法来查询包里有多少个与 `item` 相同的元素(若是没有, 则将个数视为 0), 然后把元素个数加 1, 并将其设为与 `item` 键相关联的值, 如果原来没有这种元素, 那么该操作会向字典中新增这一元素。

```
def __delitem__(self, item):
    if self.__bag.get(item) is not None:
        self.__bag[item] -= 1
        if self.__bag[item] <= 0:
            del self.__bag[item]
    else:
        raise KeyError(str(item))
```

如果用户试图移除包中没有的元素, 那么就应该抛出 `KeyError` 异常, 并把待删除的元素以字符串形式放在异常信息里。反之, 若待移除的元素确实在包中, 则应递减其数量。若数量小于等于 0, 则将其从包中删去。

该类不用实现 `__getitem__()` 及 `__setitem__()` 这两个特殊方法, 由于包中的元素是无序的, 因此这两个方法对包来说没有意义。用户只需用 `bag.add()` 添加元素、用 `del bag[item]` 删除元素、用 `bag.count(item)` 查询元素在包中的出现次数即可。

```
def count(self, item):
    return self.__bag.get(item, 0)
```

上面这个方法很简单, 它返回给定的元素在包中出现的次数, 若包中不含此元素, 则返回 0。当用户所要查询的元素并未出现在包中时, 也可以抛出 `KeyError` 异常。如果要采用这种做法, 那么可将方法体改为 `return self.__bag[item]`。

```
def __len__(self):
    return sum(count for count in self.__bag.values())
```

上面这个方法不太好实现, 因为它必须分别统计包中每个元素重复出现的次数。为此, 我们遍历 `__bag` 中的每一个值(也就是包中每种元素的出现次数), 并通过 Python 内置的 `sum()` 函数将其汇总。

```
def __contains__(self, item):
    return item in self.__bag
```

如果待查询的元素在包中至少有一个, 那么上述方法就返回 `True`(元素若是真的在包里, 那么与其相关的计数值至少是 1), 否则返回 `False`。

至此，Bag类的方法已经差不多讲完了，只剩下支持迭代器的那个方法了。首先，我们看看 Bag1.py 模块的 Bag.\_\_iter\_\_() 方法。

```
def __iter__(self): # This needlessly creates a list of items!
    items = []
    for item, count in self._bag.items():
        for _ in range(count):
            items.append(item)
    return iter(items)
```

上面是 \_\_iter\_\_() 方法的第一种实现方式。这段代码构建了一份列表，其中每种元素的个数都与该元素在包中重复出现的次数相同。如果包很大，那么创建出来的列表也会非常大，所以这么做是相当低效的，接下来我们要讲两种更好的办法。

```
def __iter__(self):
    for item, count in self._bag.items():
        for _ in range(count):
            yield item
```

上面这段代码节选自 Bag2.py 模块，它是 Bag 类中唯一与 Bag1.py 中不同的方法。

这段代码遍历包中的每种元素，并获取元素本身以及元素的出现次数。每种元素出现几次，我们就调用几次 yield 语句。将方法变为生成器确实会产生少许开销，但这个开销是固定的，与元素个数无关，而且也无须另外创建列表，所以此方法的效率要比 Bag1.py 版本中的高出许多。

```
def __iter__(self):
    return (item for item, count in self._bag.items()
            for _ in range(count))
```

上面是 Bag3.py 模块中的 Bag.\_\_iter\_\_() 方法，它的效率与 Bag2.py 模块中的版本一样高，只不过这次没有把方法变为生成器，而是令其返回生成器表达式。

尽管书里的这个 Bag 写得很好，但大家别忘了，标准库里还有个类已经实现了此功能，那就是 collections.Counter。

## 3.5 中介者模式

“中介者模式”<sup>⊖</sup>（Mediator Pattern）可以创建中介者，并把对象之间的交互逻辑封装到里面。这样的话，对象无须直接知悉对方即可通过中介者与之交流。比方说，点击某个按钮对象之后，我们只要把这件事告诉中介者就行了，然后由中介者来通知那些对按钮点击事件感兴趣的对象。图 3.4 里的 Mediator 在表单上的文本控件与按钮控件之间充当中介者，图中还列出了与这些控件相关的方法。

---

<sup>⊖</sup> 也称“仲裁器模式”。——译者注

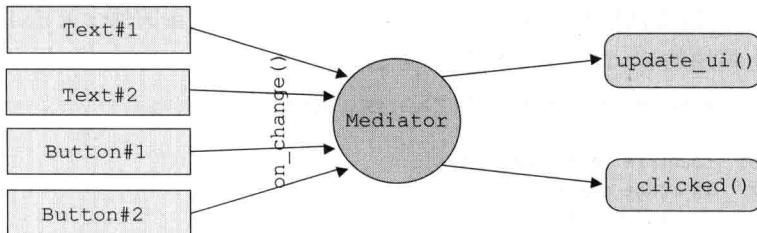


图 3.4 在表单各控件之间充当中介者的 Mediator 类

GUI 编程中无疑会频繁用到此模式。实际上，Python 中的每一种 GUI 工具包（例如 Tkinter、PyQt/PySide、PyGObject、wxPython 等）都提供了与之等效的机制。大家在第 7 章里将会看到用 Tkinter 所写的例子。

本节分别用两种方式实现中介者模式。第一种是常规方式，第二种则用到了协程。我们虚构了一套用户界面工具包，两种方式都需要其中的 Form、Button 及 Text 类（这些类的实现代码稍后就会列出来）。

### 3.5.1 用常规方式实现中介者

本节将以常规方式实现中介者，令其协调事物之间的交互，具体到本例，就是令 Mediator 类协调表单控件与相关方法之间的交互。所有范例代码均出自 mediator1.py 程序。

```

class Form:
    def __init__(self):
        self.create_widgets()
        self.create_mediator()
  
```

与本书中的大部分函数及方法一样，上面这个函数也是经过“彻底重构”（ruthlessly refactored）之后才变成现在这个样子的，在本例中，它会把全部工作都交由其他函数执行。

```

def create_widgets(self):
    self.nameText = Text()
    self.emailText = Text()
    self.okButton = Button("OK")
    self.cancelButton = Button("Cancel")
  
```

上面这个函数会创建两个文本框控件，用于输入用户名及电子邮件地址，并创建两个按钮控件，用来表示“确认”OK 及“取消”Cancel。在真实的用户界面中，当然还要有标签控件，而且所有控件的位置都要摆放得当，但由于本例只是纯粹为了演示中介者模式，所以我们不处理这些问题。大家稍后会看到 Text 和 Button 类的代码。

```

def create_mediator(self):
    self.mediator = Mediator(((self.nameText, self.update_ui),
                             (self.emailText, self.update_ui),
                             (self.okButton, self.clicked),
                             (self.cancelButton, self.clicked)))
    self.update_ui()
  
```

上面这段代码创建了负责整张表单的中介者对象。构造该对象时，可以指定若干对关系，每对关系都是由控件及 callable 所构成的二元组。构建好的中介者对象将会协调这些关系。在本例中，所有 callable 均为绑定方法。（绑定方法与非绑定方法之间的区别请参阅 2.5 节中的补充知识。）这里所指定的四对关系的意思是：只要文本框中的文字有变化，就调用 Form.update\_ui() 方法；只要点击了按钮，就调用 Form.clicked() 方法。创建好中介者之后，用 update\_ui() 方法来初始化表单。

```
def update_ui(self, widget=None):
    self.okButton.enabled = (bool(self.nameText.text) and
                             bool(self.emailText.text))
```

上述方法判断两个文本框中是不是都有文字，如果是，就启用 OK 按钮，否则将其禁用。当文本框中的文本有变化时，显然应该调用此方法。

```
def clicked(self, widget):
    if widget == self.okButton:
        print("OK")
    elif widget == self.cancelButton:
        print("Cancel")
```

上述方法将于用户点击按钮时执行，它只是把按钮文本打印出来而已。但在编写真正的应用程序时，应该令该方法完成一些更为有用的操作。

```
class Mediator:
    def __init__(self, widgetCallablePairs):
        self.callablesForWidget = collections.defaultdict(list)
        for widget, caller in widgetCallablePairs:
            self.callablesForWidget[widget].append(caller)
        widget.mediator = self
```

Mediator 类有两个方法，上面是首个方法的代码。我们要创建这样一个字典：它的键是控件，值是列表，列表中可含有一个或多个 callable。此需求可通过 default dictionary（带默认值的字典）来实现。在访问其中的键时，如果该键尚未出现在字典里，那么就会用构造字典时所传入的 callable 创建一个值，并将该值添加到字典中。因为本例是用 list 对象来构造这个字典的，所以在调用这个 callable 时，就会得到一份新的空列表。这样的话，首次在字典中查询某控件所对应的列表时，字典就会创建新的条目，条目中的键就是那个控件，而值则是个空列表，我们把 caller 变量放在这个列表里。再度查询该控件时，由于已经有了列表，所以 caller 就会直接添加到已有的列表之中。我们还会把这个中介者对象（也就是 self）赋给控件里名为 mediator 的 attribute（若无此 attribute，则创建之）。

`__init__()` 会把元组中的绑定方法按其出现的先后顺序添加到相关列表中，若是不关注顺序，那么构建 default dictionary 时可以不用 list，而改用 set，并且不通过 list.append() 添加绑定方法，而是通过 set.add() 来添加。

```
def on_change(self, widget):
    callables = self.callablesForWidget.get(widget)
```

```

if callables is not None:
    for caller in callables:
        caller(widget)
else:
    raise AttributeError("No on_change() method registered for {}"
                         .format(widget))

```

如果“受协调的”(mediated)那些对象(也就是构造Mediator时所传入的那些控件)的状态发生改变,那么Mediator的on\_change()方法就会执行。此方法会取出与控件相关的每一个绑定方法,并依次调用它们。

```

class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.on_change(self)

```

上面这个辅助类是设计给受协调的类来继承的。它会保存指向中介者对象的引用,当控件状态改变时,系统会执行本对象的on\_change()方法,而该方法将以对象自身(用self表示)为参数,调用中介者对象的on\_change()方法。

由于Mediated的子类不会修改其方法,所以我们也像早前那样把这个基类做成类修饰器(参见2.4.2节)。

```

class Button(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self.enabled = True
        self.text = text

    def click(self):
        if self.enabled:
            self.on_change()

```

Button类继承自Mediated。于是其中就有名为self.mediator的attribute以及名为on\_change()的方法,该方法会在控件状态改变(比如用户点击了此按钮)时执行。

在本例中,系统调用Button.click()方法时,继承自Mediated的Button.on\_change()方法将会执行,而该方法又会调用中介者对象的on\_change()方法,继而调用与该按钮相关联的其他方法。此处指的是Form.clicked()方法,调用时,widget参数代表按钮控件。

```

class Text(Mediated):

    def __init__(self, text=""):
        super().__init__()
        self._text = text

```

```

@property
def text(self):
    return self.__text

@text.setter
def text(self, text):
    if self.text != text:
        self.__text = text
        self.on_change()

```

Text 类在结构上与 Button 相同，也继承自 Mediated。

对于任何控件（按钮、文本框等），只要控件是 Mediated 的子类，并且其状态改变时会执行 on\_change() 方法，那么我们就可以把它与其他控件的交互交给 Mediator 处理。创建 Mediator 时，必须注册控件及其状态改变时所需调用的方法。这意味着表单中所有控件都是“松散地耦合起来的”（loosely coupled），它们之间不会直接联系，因为那样容易出问题。

### 3.5.2 基于协程的中介者

中介者好比一条管道，它可以接收由 on\_change() 方法所传来的信息，并将其发送给对此感兴趣的对象，而 3.1.2 节所讲的“协程”（coroutine）就具备这种能力。本节所有代码均选自 mediator2.py 程序，未列出的那部分代码与上一小节中的 mediator1.py 相同。

此节所用的办法与上一小节不同。原来我们是把控件和相关方法绑定成一对，等控件状态改变时，令中介者对象去调用与之关联的方法。

而现在则不同，每个控件都有中介者对象，而该对象实际上是一条由协程所拼合起来的管道。控件状态改变时，它会把自己发给管道，由管道中的组件（也就是协程）来决定是否需要执行操作，以应对控件所发生的变化。

```

def create_mediator(self):
    self.mediator = self._update_ui_mediator(self._clicked_mediator())
    for widget in (self.nameText, self.emailText, self.okButton,
                   self.cancelButton):
        widget.mediator = self.mediator
    self.mediator.send(None)

```

用协程实现中介者模式时，不需要单独的 Mediator 类。我们会用协程创建一条管道，本例的管道有两个组件，分别是 self.\_update\_ui\_mediator() 与 self.\_clicked\_mediator()。（它们都是 Form 类的方法。）

等管道就位之后，我们把每个控件的 mediator 设成这条管道。最后，向管道中发送 None。由于没有哪个控件是 None，所以这样做不会引发与特定控件有关的操作，但却可以执行“表单级别”（form-level）的操作，比如在 \_update\_ui\_mediator() 中启用或禁用 OK 按钮。

```

@coroutine
def _update_ui_mediator(self, successor=None):
    while True:
        widget = (yield)
        self.okButton.enabled = (bool(self.nameText.text) and
                                  bool(self.emailText.text))
        if successor is not None:
            successor.send(widget)

```

上面这个协程是管道的一部分。(3.1.2节已经讲解了`@coroutine`修饰器，并且列出了它的代码。)

控件有变化时，会把自己发给管道，而协程则会把该控件当成`yield`表达式的值赋给`widget`变量。无论有没有控件改变，我们都得判断是否需要启用或禁用OK按钮。`(widget`变量收到了值，并不意味着一定有控件发生了变化，比如初始化表单时，`widget`变量所收到的值是`None`，但此时并没有控件发生变化。) 处理完OK按钮之后，如果管道中还有后继的协程，那么就把发生变化的控件传过去。

```

@coroutine
def _clicked_mediator(self, successor=None):
    while True:
        widget = (yield)
        if widget == self.okButton:
            print("OK")
        elif widget == self.cancelButton:
            print("Cancel")
        elif successor is not None:
            successor.send(widget)

```

上述管道协程只关注OK按钮与Cancel按钮的点击事件。如果发生改变的控件是这两个按钮之一，那么本协程就处理它；如果不是，并且管道中还有后继者，那么就将该控件传给下个协程。

```

class Mediated:

    def __init__(self):
        self.mediator = None

    def on_change(self):
        if self.mediator is not None:
            self.mediator.send(self)

```

`Button`类与`Text`类都和`mediator1.py`相同，但是`Mediated`类却稍有变化：执行`on_change()`方法时，它会把状态有变化的控件(用`self`表示)发给由`mediator`变量所表示的管道。

与上一小节类似，`Mediated`类也可以改用类修饰器来实现。本书范例代码中的`mediator2d.py`程序就是这么做的。(2.4.2节讲解了如何用类修饰器实现继承。)

我们可以改变中介者模式，用它来实现“多路复用”(multiplexing)，使多个对象与多个

对象之间通过同一条管道交流。此问题亦可参见3.7节的观察者模式与3.8节的状态模式。

## 3.6 备忘录模式

“备忘录模式”(Memento Pattern)可在不破坏封装的前提下保存并恢复对象状态。

Python语言对该模式提供了原生支持。`pickle`模块可将绝大部分Python对象“序列化”(`pickle`)及“反序列化”(`unpickle`)(有几个限制,比如文件对象就不能序列化)。在Python语言中,可以序列化的内容包括:`None`、`bool`、`bytearray`、`byte`、`complex`、`float`、`int`、`str`。如果`dict`、`list`、`tuple`里面只含有“可序列化的对象”(pickleable object)(可序列化的集合也算),那么这三种对象就可以序列化,另外,顶级函数与顶级类也能序列化。如果自编的顶级类其`__dict__`可以序列化,那么该类实例就可以序列化,也就是说,绝大多数自编类的对象都可以序列化。用`json`模块也能实现同样效果,但它只支持Python基本类型、字典与列表。(3.3.3节有`json`与`pickle`的使用范例。)

在极个别情况下,无法用`pickle`模块直接将对象序列化,但即便如此,我们也可以自行实现序列化功能。比方说,可以实现`__getstate__()`与`__setstate__()`特殊方法,可能还需要实现`__getnewargs__()`方法。与之类似,如果想把自编的类用JSON格式保存起来,那么可以自己扩充`json`模块的编码器与解码器。

我们也可以自创格式与协议,但由于Python对该模式支持得相当好,所以这么做意义不大。

反序列化的过程实际上可以执行任意Python代码,如果数据是从不可信的来源中获取的(比如从物理介质中获取,或通过网络连接传来),那么就不应该将其反序列化。在这种情况下,使用JSON格式更为安全。还有种办法,就是在序列化时加入“校验和”(`checksum`)及“加密”(`encryption`)机制,以防数据遭人篡改。

## 3.7 观察者模式

“观察者模式”(Observer Pattern)可以处理对象间“多对多”(many-to-many)的依赖关系,当某对象改变状态时,所有相关对象都会得到通知。目前,该模式及其变种似乎通常以MVC(model/view/controller,模型/视图/控制器)的形式来表述。MVC范式用模型来表示数据,并有若干个展示数据的视图,还有一些控制器在“输入”(比如用户交互)与“模型”之间协调。模型所发生的变化会自动反映到相关的视图上面。

MVC范式有种常见的简化版,就是把控制器合并到视图里面,这样就成了MV范式(model/view,模型/视图):视图既展示数据,又充当用户输入与模型之间的中介。具体到观察者模式,就意味着视图是模型的“观察者”(observer),而模型则是“观测物”(the subject being observed)。

本节要创建的模型是个数值，模型里还包含了该数的最小值与最大值（例如滚动条、数值滑块控件、温度监测器等都需要用到这两个值）。我们将创建两个不同的观察者（也就是视图）来观察这个模型，其中一个会在模型的值有变化时以 HTML 进度条的形式将其输出，而另一个则会保留一份变更记录，把发生变化的时间（timestamp，时间戳）与值本身记录下来。现在我们试着运行一下 `observer.py` 程序。

```
$ ./observer.py > /tmp/observer.html
0 2013-04-09 14:12:01.043437
7 2013-04-09 14:12:01.043527
23 2013-04-09 14:12:01.043587
37 2013-04-09 14:12:01.043647
```

该程序会把历史记录发送至 `sys.stderr`，并将 HTML 输出至 `sys.stdout`，而我们则把 `sys.stdout` 重定向为 HTML 文件。图 3.5 就是程序所输出的 HTML 页面，它用“单行的 HTML 表格”（one-row HTML table）来表示进度条，最上方的那个进度条代表初次观测时的模型值（此时模型值是 0，所以进度条没有进度），模型值每变更一次，就多画一个进度条。图 3.6 演示了范例程序所用的“模型 / 视图”架构。

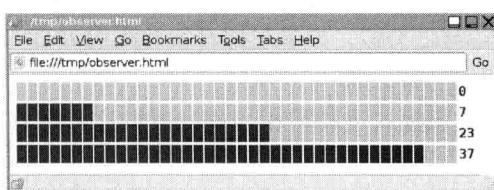


图 3.5 由 `observer.py` 程序所输出的网页，  
其中反映了模型值的变化过程

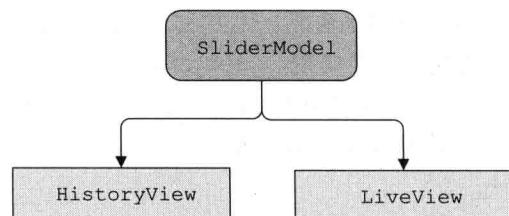


图 3.6 模型及两个视图

本节中的范例程序 `observer.py` 把添加、移除及通知观察者的功能都放在 `Observed` 这个基类里面了。`SliderModel` 类用于表示有最小值及最大值限制的模型，由于它继承了 `Observed` 类，所以其值可受观测。程序里有两个视图要观察该值，一个是 `HistoryView`，另一个是 `LiveView`。这些类稍后再讲，我们先来看看程序的 `main()` 函数是怎样使用它们的，并看看图 3.5 中的结果是如何得来的。

```

def main():
    historyView = HistoryView()
    liveView = LiveView()
    model = SliderModel(0, 0, 40) # minimum, value, maximum
    model.observers_add(historyView, liveView) # liveView produces output
    for value in (7, 23, 37):
        model.value = value # liveView produces output
    for value, timestamp in historyView.data:
        print("{:3} {}".format(value, datetime.datetime.fromtimestamp(timestamp)), file=sys.stderr)
  
```

这段代码首先创建两个视图。接下来创建模型，该模型的最小值是 0，当前值是 0，最

大值是 40。然后把两个视图添加为模型的观察者。添加成观察者之后，LiveView 立刻就会输出网页中的第一个进度条，而 HistoryView 则会把首个模型值及当前时间记录下来。我们接着又更新了三次模型值，每次更新时，LiveView 都会输出一个单行的 HTML 表格，HistoryView 则会记录模型值及其变更时间。

程序最后会把全部历史记录都输出到 `sys.stderr` 中（也就是输出到控制台里）。`datetime.datetime.fromtimestamp()` 函数以时间戳为参数，返回与之等效的 `datetime.datetime` 对象。这个时间戳表示当前时刻距离 epoch<sup>①</sup> 的秒数，该值是调用 `time.time()` 函数所获取到的。`str.format()` 方法会把 `datetime.datetime` 对象转换成 ISO-8601 格式的字符串。

```
class Observed:
    def __init__(self):
        self.__observers = set()

    def observers_add(self, observer, *observers):
        for observer in itertools.chain((observer,), observers):
            self.__observers.add(observer)
            observer.update(self)

    def observer_discard(self, observer):
        self.__observers.discard(observer)

    def observers_notify(self):
        for observer in self.__observers:
            observer.update(self)
```

凡是打算支持观察者模式的模型或类都应该继承 `Observed` 类。该类用 `set` 来保存观察者对象。当用户向 `Observed` 注册新的观察者对象时，观察者的 `update()` 方法会执行，这使得它能够用模型当前的状态来初始化自己。模型状态发生变化时，应该调用继承而来的 `observers_notify()` 方法，这样的话，就会执行每个观察者对象（也就是每个视图）的 `update()` 方法，以确保它们都能反映出模型的最新状态。

`observers_add()` 方法的写法值得注意。我们想接受一个或多个观察者对象，但假若只用 `*observers` 做参数，则不符合要求，因为它还可以接受 0 个或多个对象。于是，我们先用 `observer` 参数接受一个观察者，再用 `*observers` 参数接受剩下的 0 个或多个观察者。遍历观察者的时候，本来可以通过“拼接元组”（tuple concatenation）的办法来做（例如 `for observer in (observer,) + observers:)`，但我们采用了效率更高的 `itertools.chain()` 方法。2.3.2 节曾经讲过这个方法，它可以接受任意数量的 `iterable`，并返回单个 `iterable`，遍历这个 `iterable`，也就相当于依次遍历参数里的那些 `iterable`。

---

<sup>①</sup> 在编程语言中，epoch 一般指世界标准时间 1970 年 1 月 1 日 0 时 0 分 0 秒。——译者注

```

class SliderModel(Observed):
    def __init__(self, minimum, value, maximum):
        super().__init__()
        # These must exist before using their property setters
        self.__minimum = self.__value = self.__maximum = None
        self.minimum = minimum
        self.value = value
        self.maximum = maximum

    @property
    def value(self):
        return self.__value

    @value.setter
    def value(self, value):
        if self.__value != value:
            self.__value = value
            self.observers_notify()
...

```

上面这个类就是本例所用的模型，凡是模型类都可以按此套路来写。继承 Observed 就意味着得到了私有的 set 观察者（刚开始是空的，可用来保存观察者对象）以及 observers\_add()、observer\_discard()、observers\_notify() 方法。模型状态改变时（比如模型值变了），必须调用 observers\_notify() 方法通知观察者，这样的话，它们才有机会执行相应的操作。

该类还有 minimum 与 maximum 属性，但它们的代码没有列出来。这两个属性的结构与 value 相同，而且其 setter 也要调用 observers\_notify() 方法。

```

class HistoryView:
    def __init__(self):
        self.data = []

    def update(self, model):
        self.data.append((model.value, time.time()))

```

上面这个视图是模型的观察者，其 update() 方法除了 self 参数之外，只有一个参数，就是受观测的模型。update() 方法在执行时，会把值和时间戳放在二元组里，然后将其加入 self.data 列表中，于是，该列表就相当于一份模型值的变更日志。

```

class LiveView:
    def __init__(self, length=40):
        self.length = length

```

上面这个类也是观察模型的视图。它用单行 HTML 表格来表示模型值，而 \_\_init\_\_() 方法的 length 参数则表示每行所要绘制的单元格总个数。

```

def update(self, model):
    tippingPoint = round(model.value * self.length /

```

```

        (model.maximum - model.minimum))
td = '<td style="background-color: {}>&ampnbsp</td>'
html = ['<table style="font-family: monospace" border="0"><tr>']
html.extend(td.format("darkblue") * tippingPoint)
html.extend(td.format("cyan") * (self.length - tippingPoint))
html.append("<td>{}</td></tr></table>".format(model.value))
print("".join(html))

```

这个视图首次观察模型时，会执行上述方法，其后模型有变化时，也会执行它。该方法会输出一张 HTML 表格来表示模型，这张表格只有一行，行中的单元格总数与 self.length 相同，空白单元格是“蓝绿色”（cyan），而填充后的单元格则是“深蓝色”（dark blue）。此函数会算出填充后的单元格（如果有的话）与空白单元格的分界点，以此确定每种单元格的数量。

观察者模式广泛运用于 GUI 编程，而且在“仿真”（simulation）及服务器等其他事件处理架构中也能用到，比如“数据库触发器”（database trigger）、Django 的“信号系统”（signaling system）、Qt GUI 应用程序框架的“信号”（signal）与“槽”（slot）机制，以及 WebSocket 的许多用例。

## 3.8 状态模式

“状态模式”（State Pattern）可用来设计其行为随状态而变的对象，也就是有“mode”的对象。

我们创建 Multiplexer 类来演示此模式，该类有两种状态，实例的行为会随其状态而变。multiplexer（数据选择器、多路复用器）处于活动状态时，可以接受“连接”（connection），也就是由“事件名称”（event name）和“回调函数”（callback）所组成的值对，其中的回调函数可以是 Python 中的任意 callable（比如 lambda、函数、绑定方法等）。设置好连接之后，当 multiplexer 收到事件并处于活动状态时，就会执行相关的回调函数。如果 multiplexer 已经“休眠”（dormant），那么在其上调用任何方法都不会出问题，因为它们并不会执行实际操作。

为了演示 multiplexer 的用法，我们创建一些回调函数，用来统计收到的各种事件，然后将其连接到活动的 multiplexer 上面。接下来，给 multiplexer 发送随机事件，并打印出这些回调函数所统计到的事件个数。下面这些代码均选自 multiplexer1.py 程序。我们来看看程序运行后的输出信息：

```

$ ./multiplexer1.py
After 100 active events: cars=150 vans=42 trucks=14 total=206
After 100 dormant events: cars=150 vans=42 trucks=14 total=206
After 100 active events: cars=303 vans=83 trucks=30 total=416

```

给活动的 multiplexer 发送 100 个随机事件之后，我们将其状态切换成休眠，然后，再向其发送 100 个事件，此时相关的回调函数应该不会执行。最后，我们将 multiplexer 切换回活动状态，又向其发送 100 个事件，这次，相关的回调函数就会运行了。

我们首先讲解如何构造 multiplexer 对象、如何将事件与回调函数相连接以及如何向 multiplexer 发送事件，然后看看回调函数及 Event 类，最后再来谈 Multiplexer 类本身。

```
totalCounter = Counter()
carCounter = Counter("cars")
commercialCounter = Counter("vans", "trucks")

multiplexer = Multiplexer()
for eventName, callback in (("cars", carCounter),
    ("vans", commercialCounter), ("trucks", commercialCounter)):
    multiplexer.connect(eventName, callback)
multiplexer.connect(eventName, totalCounter)
```

上面这段代码创建了几个计数器。因为这些类的实例都是 callable 对象，所以可当作函数（比如回调函数）来用。如果在创建计数器时指定了名字，那么会按名字单独计数；若像 totalCounter 那样不指定名字，则会统一计数。

创建好计数器之后，新建 multiplexer 对象（该对象默认处于活动状态）。接下来把回调函数同事件相连接。我们关注“cars”、“vans”及“trucks”这三个事件名称。carCounter() 函数与“cars”相连；commercialCounter() 函数与“vans”及“trucks”相连；而 totalCounter() 函数则与这三个事件都相连。

```
for event in generate_random_events(100):
    multiplexer.send(event)
print("After 100 active events: cars={} vans={} trucks={} total={}"
      .format(carCounter.cars, commercialCounter.vans,
              commercialCounter.trucks, totalCounter.count))
```

上面这段代码将生成 100 个随机事件，并将其发送给 multiplexer。如果某个事件是“car”事件，那么 multiplexer 就会调用 carCounter() 及 totalCounter() 函数，并将该事件作为唯一的参数传进去。同理，若是“vans”事件或“trucks”事件，则会调用 commercial Counter() 与 totalCounter() 函数。

```
class Counter:

    def __init__(self, *names):
        self.anonymous = not bool(names)
        if self.anonymous:
            self.count = 0
        else:
            for name in names:
                if not name.isidentifier():
                    raise ValueError("names must be valid identifiers")
                setattr(self, name, 0)
```

如果构造 Counter 的时候没有指定名字，那么实例中就会有个匿名计数器，其值放在 self.count 变量中。反之，若指定了名字，则会根据这些名字，用 Python 内置的 setattr() 函数来创建相应的 attribute。例如，carCounter 实例中会有名为 self.cars 的 attribute，而 commercialCounter 里则会有名叫 self.vans 及 self.trucks 的 attribute。

```
def __call__(self, event):
    if self.anonymous:
        self.count += event.count
    else:
        count = getattr(self, event.name)
        setattr(self, event.name, count + event.count)
```

调用 Counter 实例，也就相当于调用上面这个名叫 `__call__()` 的特殊方法。如果 Counter 是匿名的（例如 `totalCounter`），那么就递增 `self.count` 的值。否则，就把与事件名称相对应的属性值找出来。比方说，如果事件名称是“`trucks`”，那么就找到 `self.trucks` 的值，并将其赋给 `count` 变量。然后把原有的 `count` 与事件里面的 `count` 相加，将二者之和设置成 attribute 的新值。

在调用内置的 `getattr()` 函数时，由于没有提供默认值，所以假如实例中没有待获取的 attribute（比如我们把“`trucks`”误写为“`truck`”），那么该方法就会抛出 `AttributeError` 异常，这正是我们想要的效果。这样做还能防止创建出名称错误的 attribute，因为当写错名称时，`getattr()` 会抛出异常，从而使程序不会执行到 `setattr()` 那一行。

```
class Event:

    def __init__(self, name, count=1):
        if not name.isidentifier():
            raise ValueError("names must be valid identifiers")
        self.name = name
        self.count = count
```

上面这几行就是 `Event` 类的全部代码了。本节主要是想通过 `Multiplexer` 类来演示状态模式，而 `Event` 类只不过是为此所编写的一部分基础代码，所以它非常简单。另外，`Multiplexer` 类也可以算作观察者模式（参见 3.7 节）。

### 3.8.1 用同一套方法来处理不同的状态

处理类中的状态主要有两种办法。第一种是像本节这样，用同一套方法来处理不同的状态。第二种则是像下一小节（即 3.8.2 节）那样，用不同的方法来处理不同的状态<sup>⊖</sup>。

```
class Multiplexer:

    ACTIVE, DORMANT = ("ACTIVE", "DORMANT")

    def __init__(self):
        self.callbacksForEvent = collections.defaultdict(list)
        self.state = Multiplexer.ACTIVE
```

`Multiplexer` 类有两个状态（或者说两种“mode”）：`ACTIVE` 及 `DORMANT`。如果 `Multiplexer` 实例处于 `ACTIVE` 状态，那么与状态相关的那些方法就会执行相应的操作；

---

⊖ 原书将第一种办法称为“use state-sensitive methods”，直译为“使用能区分各种状态的方法”，将第二种办法称为“use state-specific methods”，直译为“使用各状态所特有的方法”。——译者注

但若处于 DORMANT 状态，则任何事都不做。我们通过代码来确保 Multiplexer 对象刚创建好的时候会处于 ACTIVE 状态。

`self.callbacksForEvent` 字典的键是事件名称，而值则是由 `callable` 所构成的列表。

```
def connect(self, eventName, callback):
    if self.state == Multiplexer.ACTIVE:
        self.callbacksForEvent[eventName].append(callback)
```

上述方法可将事件名称与回调函数关联起来。由于 `self.callbackForEvent` 是个 default dictionary(3.5.1 节讲过 default dictionary 的用法)，所以假如给定的事件名不在字典里，那么该字典会创建一份空列表，并将事件名当作键，将该列表当作值，把此条目放入字典。若是字典里已经有这个事件名，则直接返回与之对应的列表。无论是哪一种情况，我们都可以把新的回调函数追加到列表里。

```
def disconnect(self, eventName, callback=None):
    if self.state == Multiplexer.ACTIVE:
        if callback is None:
            del self.callbacksForEvent[eventName]
        else:
            self.callbacksForEvent[eventName].remove(callback)
```

如果调用 `disconnect()` 方法时没有指明 `callback` 参数，那么我们就认定用户想解除事件名称同所有回调函数之间的关联。若指定了 `callback` 参数，则只会把这个回调函数从事件名称的回调函数列表里删去。

```
def send(self, event):
    if self.state == Multiplexer.ACTIVE:
        for callback in self.callbacksForEvent.get(event.name, ()):
            callback(event)
```

如果有人给 multiplexer 发送事件，并且 multiplexer 正处在活动状态，那么 `send()` 方法会遍历与该事件相关的全部回调函数（也可能没有回调函数与该事件相关联），并用事件本身作参数来调用每个回调函数。

### 3.8.2 用不同的方法来处理不同的状态

`multiplexer2.py` 程序几乎与 `multiplexer1.py` 相同，但是其 `Multiplexer` 类却采用两套方法来分别处理两种状态，而不是像上一小节那样只用一套方法来做。`Multiplexer` 类的两个状态以及 `__init__()` 方法都和原来一样，而 `self.state` 则由 attribute 变成了属性。

```
@property
def state(self):
    return (Multiplexer.ACTIVE if self.send == self.__active_send
            else Multiplexer.DORMANT)
```

multiplexer 并不像原来那样保存状态，而是通过 if 语句来判断状态。如果 self.send 这个公共方法指向与 ACTIVE 状态相配套的那个方法，那么就认定该对象当前处于活动状态；若是指向与 DORMANT 状态相配套的那个方法，则认定对象当前处于休眠状态。我们接下来就会讲解这两套方法。

```
@state.setter
def state(self, state):
    if state == Multiplexer.ACTIVE:
        self.connect = self._active_connect
        self.disconnect = self._active_disconnect
        self.send = self._active_send
    else:
        self.connect = lambda *args: None
        self.disconnect = lambda *args: None
        self.send = lambda *args: None
```

当 multiplexer 的状态发生变化时，state 属性的 setter 会把 self.connect、self.disconnect 及 self.send 设置成与当前状态相符的那套方法。比方说，如果状态变成了 DORMANT，那么这三个公共方法都会各自指向 lambda 版的匿名方法。

```
def __active_connect(self, eventName, callback):
    self.callbacksForEvent[eventName].append(callback)
```

上面这个 \_\_active\_connect() 是专门在 ACTIVE 状态下使用的私有方法。self.connect 要么指向该方法，要么就指向什么事都不做的 lambda 匿名方法。和 disconnect 及 send 有关的那两个方法没有列出来，但其写法与 \_\_active\_connect() 相似。重点在于，这些方法都只会在特定的状态下调用，所以不需要检测实例的状态，因而能运行得稍微快一点。

用协程来实现 Multiplexer 当然也很简单，但由于前面已经举了与协程有关的例子，所以这里就不再重复了。（multiplexer3.py 程序演示了如何用协程来实现 multiplexer。）

虽说本节只以 multiplexer 为例来讲解状态模式，但其实在许多场合中都经常能见到这种“有状态的对象”（stateful object 或 modal object）。

### 3.9 策略模式

“策略模式”（Strategy Pattern）能把一系列“可互换的”算法封装起来，并根据用户需求来选择其中一种。

本节将用两种算法把列表中的元素输出到表格里面。列表中的元素个数不限，而用户可以指定表格的行数。第一种算法将会产生如图 3.7 所示的 HTML 页面，该图演示了表格行数为 2、3、4 时的效果。第二种算法则会以纯文本形式输出，下面列出了表格行数为 4 和 5 时的效果。

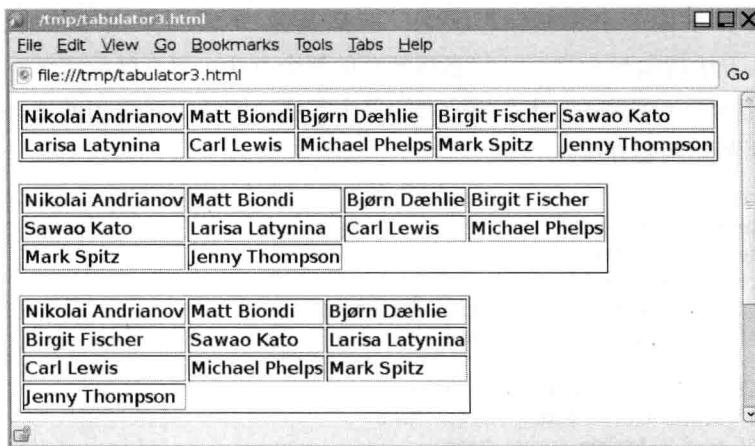


图 3.7 由 tabulator 程序所输出的 HTML 表格

```
$ ./tabulator3.py
```

```
...
+-----+-----+-----+
| Nikolai Andrianov | Matt Biondi      | Bjørn Dæhlie   |
| Birgit Fischer   | Sawao Kato       | Larisa Latynina|
| Carl Lewis        | Michael Phelps  | Mark Spitz    |
| Jenny Thompson    |                   |               |
+-----+-----+-----+
+-----+-----+
| Nikolai Andrianov | Matt Biondi      |
| Bjørn Dæhlie     | Birgit Fischer   |
| Sawao Kato        | Larisa Latynina |
| Carl Lewis         | Michael Phelps  |
| Mark Spitz         | Jenny Thompson   |
+-----+-----+
```

想实现算法切换功能有很多办法。最容易想到的一种是：创建 Layout 类，令其实例接受 Tabulator 实例，再由该实例负责将列表中的元素排布在表格里。tabulator1.py 程序就是这么做的（该程序没有列出来）。稍微好一点的做法是：不把制表算法表示成 Tabulator 实例的方法，而是把它们做成 Tabulator 子类的静态方法，这样就不用把 Tabulator 实例传给 Layout 了，只需传入含有制表算法的 Tabulator 子类即可。tabulator2.py 程序采用的便是这种做法（该程序也没有列出来）。

本节将演示一种更简单、更精良的做法，就是把实现了相关算法的制表函数直接传给 Layout 类。

```
WINNERS = ("Nikolai Andrianov", "Matt Biondi", "Bjørn Dæhlie",
           "Birgit Fischer", "Sawao Kato", "Larisa Latynina", "Carl Lewis",
           "Michael Phelps", "Mark Spitz", "Jenny Thompson")
```

```

def main():
    htmlLayout = Layout(html_tabulator)
    for rows in range(2, 6):
        print(htmlLayout.tabulate(rows, WINNERS))
    textLayout = Layout(text_tabulator)
    for rows in range(2, 6):
        print(textLayout.tabulate(rows, WINNERS))

```

上面这个函数分别用两个不同的制表函数来创建 Layout 对象，然后用每个 layout 输出四张表，这些表格的行数分别是 2、3、4、5。

```

class Layout:

    def __init__(self, tabulator):
        self.tabulator = tabulator

    def tabulate(self, rows, items):
        return self.tabulator(rows, items)

```

上面这个类只封装了“制表”(tabulate) 算法。欲实现此算法的函数需根据指定的行数把一系列元素制成表格，并将其返回。

其实该类还可以像 tabulator4.py 这样继续精简：

```

class Layout:

    def __init__(self, tabulator):
        self.tabulate = tabulator

```

精简后的类把 tabulator 参数中的 callable (也就是传给 \_\_init\_\_() 方法的制表函数) 存放在名叫 self.tabulate 的 attribute 中。尽管 tabulator3.py 与 tabulator4.py 中的 Layout 类稍有不同，但 main() 函数调用 layout 对象的方式却没有变化。

虽说实际的制表算法与设计模式本身没有关联，但为了使范例程序完整，我们简要地讲解其中一种制表算法。

```

def html_tabulator(rows, items):
    columns, remainder = divmod(len(items), rows)
    if remainder:
        columns += 1
    column = 0
    table = ['<table border="1">\n']
    for item in items:
        if column == 0:
            table.append("<tr>")
        table.append("<td>{}</td>".format(escape(str(item))))
        column += 1
        if column == columns:
            table.append("</tr>\n")
            column %= columns
    if table[-1][-1] != "\n":

```

```

    table.append("</tr>\n")
table.append("</table>\n")
return "".join(table)

```

`html_tabulator()` 与 `text_tabulator()` 函数都必须根据给定的行数计算出表格的列数。计算好列数之后 (`columns` 变量的值就是表格的列数), 遍历所有元素, 并用 `column` 变量来记录当前遍历到行中的哪一列了。

`text_tabulator()` 函数 (该函数的代码没有列在书中) 要比 `html_tabulator()` 函数稍微长一些, 但步骤是相同的。

现实环境中的各种算法其代码与性能也许有很大差异, 而策略模式则使用户能够根据需求来选择最为合适的方案。把各种算法当成 `callable` (`lambda`、函数、绑定方法都是 `callable`) 安插在程序里是件非常容易的事, 因为 `callable` 在 Python 语言里是“一等对象”, 也就是说, 我们可以像使用其他对象那样, 在函数间传递 `callable`, 或将其存放到集合里。

## 3.10 模板方法模式

“模板方法模式” (Template Method Pattern) 可用来定义算法的各个步骤, 并将某些步骤交由子类实现。

本节将创建 `AbstractWordCounter` 类, 并在其中提供两个方法。第一个方法是 `can_count(filename)`, 它会返回 `Boolean` 值, 用以表示本类能不能在给定的文件里统计单词数量 (能否统计取决于文件的扩展名)。第二个方法是 `count(filename)`, 它会返回统计出来的单词数。有两个子类要继承 `AbstractWordCounter`, 其中一个用于统计纯文本文件, 另一个则用于统计 HTML 文件。我们首先看看如何用这两个类来统计单词数量 (代码选自 `wordcount1.py`):

```

def count_words(filename):
    for wordCounter in (PlainTextWordCounter, HtmlWordCounter):
        if wordCounter.can_count(filename):
            return wordCounter.count(filename)

```

我们把类中的所有方法都做成了静态方法。由于不用关注每个实例的状态, 所以无须创建实例, 直接使用类对象就好。(假如需要追踪每个实例的状态, 那么就将静态方法修改成非静态方法, 并以实例来取代类对象。)

上面这个函数会遍历两个子类对象, 如果其中有子类能够统计单词个数, 那么就令其执行统计操作并返回统计结果。若二者均不能统计, 则默认返回 `None`, 以此表示该文件中的单词个数无法统计。

```

class AbstractWordCounter:
    @staticmethod
    def can_count(filename):
        raise NotImplementedError()

class AbstractWordCounter(
    metaclass=abc.ABCMeta):
    @staticmethod
        @abc.abstractmethod

```

```

        def can_count(filename):
    @staticmethod
    pass
def count(filename):
    raise NotImplementedError()
        def count(filename):
    @abc.abstractmethod
    def count(filename):
        pass

```

AbstractWordCounter 是个纯粹的抽象类，它提供了一套统计单词数的接口，子类必须重新实现接口中的方法。左边这段代码选自 wordcount1.py，它采用传统写法；右边这段代码选自 wordcount2.py，它采用新式写法，其中用到了 abc (abstract base class, 抽象基类) 模块。

```

class PlainTextWordCounter(AbstractWordCounter):
    @staticmethod
    def can_count(filename):
        return filename.lower().endswith(".txt")

    @staticmethod
    def count(filename):
        if not PlainTextWordCounter.can_count(filename):
            return 0
        regex = re.compile(r"\w+")
        total = 0
        with open(filename, encoding="utf-8") as file:
            for line in file:
                for _ in regex.finditer(line):
                    total += 1
        return total

```

上面这个子类用非常简单的方式来定义“单词”这一概念，并以此实现单词计数接口，它假定 .txt 文件是用 UTF-8 标准来编码的（也可以理解成“7-bit ASCII”编码，这种编码方式是 UTF-8 的子集）。

```

class HtmlWordCounter(AbstractWordCounter):
    @staticmethod
    def can_count(filename):
        return filename.lower().endswith((".htm", ".html"))

    @staticmethod
    def count(filename):
        if not HtmlWordCounter.can_count(filename):
            return 0
        parser = HtmlWordCounter._HtmlParser()
        with open(filename, encoding="utf-8") as file:
            parser.feed(file.read())
        return parser.count

```

上面这个子类也实现了单词计数接口，它是为 HTML 文件设计的。该类使用自己私有的 HTML 解析器，这个解析器是 html.parser.HTMLParser 的子类，并嵌套在

HtmlWordCounter 类里面，我们马上就会讲到解析器的代码。准备好私有的 HTML 解析器之后，只需创建解析器实例，并把待统计的 HTML 发给它去解析即可。解析完成之后，把解析器所统计出来的单词数返回给调用者。

为了使范例程序完整一些，我们来看看嵌套在 HtmlWordCounter 里面的 `_HtmlParser` 类，该类用于执行实际的单词统计操作。Python 标准库中的 HTML 解析器的工作原理与 SAX 解析器（SAX 是 Simple API for XML 的所写）相仿，都会遍历待解析的文本，当相关事件发生时（比如找到了“start tag”（起始标签）或找到了“end tag”（结束标签）），会调用与之对应的方法。由此可见，若想使用这种解析器，则必须从中继承子类，并在子类里面重新实现某些方法，以便处理我们所关注的事件。

```
class _HtmlParser(html.parser.HTMLParser):
    def __init__(self):
        super().__init__()
        self.regex = re.compile(r"\w+")
        self.inText = True
        self.text = []
        self.count = 0
```

上面这个嵌套在 `HtmlWordCounter` 之中的私有类是 `html.parser.HTMLParser` 的子类，其中有四项数据。我们用简单的正则表达式描述了对“单词”的定义，并将其放在 `self.regex` 里面（所谓单词，就是由一个或多个字母、数字及下划线所组成的序列）。`self.inText` 是个 `bool` 值，用来表示当前碰到的文本是不是“用户可以看见的文本”（user-visible text），如果是，那么该值就是 `True`，若不是（比如当前文本是 `<script>` 标签或 `<style>` 标签里的文本），则该值为 `False`。`self.text` 用于保存当前标签中的若干条文本，`self.count` 表示已经统计到的单词数。

```
def handle_starttag(self, tag, attrs):
    if tag in {"script", "style"}:
        self.inText = False
```

上述方法的名称及签名都是由基类确定的（所有 `handle_...()` 形式的方法都是如此）。在默认情况下，这些 `handler` 方法不会执行任何操作，所以，如果子类想要处理某事件，那么必须重新实现与该事件相对应的方法。

我们不打算统计“嵌入式脚本”（embedded script）及“样式表”（syle sheet）里的单词，如果遇到这两种标签，那么就把文本统计标志关掉。

```
def handle_endtag(self, tag):
    if tag in {"script", "style"}:
        self.inText = True
    else:
        for _ in self.regex.finditer(" ".join(self.text)):
            self.count += 1
        self.text = []
```

如果到达脚本标签或样式表标签的末尾，那么就重新开启文本统计标志。若是到达其他

标签末尾，则遍历已经收集到的文本，并统计出其中的单词数，然后把 `self.text` 重新设置成空列表。

```
def handle_data(self, text):
    if self.inText:
        text = text.rstrip()
    if text:
        self.text.append(text)
```

如果当前碰到的文本不在脚本或样式表里，那么就把它添加到 `self.text` 之中。

由于 Python 的私有嵌套类非常灵活，而且其标准库里还有功能强大的 `html.parser.HTMLParser`，所以我们可以把那些非常复杂的解析代码都隐藏在 `_HTMLParser` 里面，这样的话，`HtmlWordCounter` 类的用户就无须关注这些细节了。

模板方法模式在某些方面与早前讲过的桥接模式（参见 2.2 节）相似。

## 3.11 访问者模式

“访问者模式”（Visitor Pattern）可以对集合或聚合中的每个对象运用函数。这与迭代器模式的常见用法有所区别：迭代器模式一般是遍历集合或聚合，并在其中每个元素上面调用方法，而访问者模式则不同，它调用的不是方法，而是“外部函数”（external function）。

Python 对该模式提供了原生支持。例如，`newList = map(function, oldSequence)` 这行语句会在 `oldSequence` 中的每个元素上面调用 `function()` 函数，并将每次调用的结果组合成 `newList` 列表。相同的效果也可以通过“列表推导”来实现：`newList = [function(item) for item in oldSequence]`。

如果想把函数套用在集合或聚合中的每个元素上面，那么可通过 `for` 循环来实现：`for item in collection: function(item)`。若元素类型各不相同，则可在 `function()` 函数里用 `if` 语句及内置的 `isinstance()` 函数来区分类型，然后分别执行与元素类型相对应的代码。

某些行为型设计模式是 Python 语言本来就支持的，而剩下的那些也不难实现。责任链模式、中介者模式以及观察者模式都可以用常规方式或基于协程的方式实现出来，而且还有其他一些改编版本，它们都能将通信过程中的各个对象解耦。命令模式可以实现“延迟求值”及“执行 - 撤销”功能。由于 Python 是“字节码形式的解释型语言”（byte-code interpreted language），所以它本身就能实现解释器模式，而且还能把待解释的程序放在单独的进程里执行。迭代器模式与观察者模式都是 Python 的内置功能。备忘录模式很容易就能用 Python 标准库（例如 `pickle` 或 `json` 模块）实现出来。Python 虽然没有对状态模式、策略模式以及模板方法模式提供内置支持，但这三者实现起来都很简单。

设计模式提供了思考、组织及实现代码的有效手段。某些设计模式只针对面向对象编程范式，而另外一些则对过程式及面向对象编程都适用。《设计模式》（《Design Patterns》）一书

出版之后，许多人都开始研究此话题，而且以后还会有更多人投入其中。若想深入学习设计模式，最好是从 Hillside Group 的网站 ([hillside.net](http://hillside.net)) 开始，它是个非营利的教育机构。

下一章将会讲解另一种编程范式：并发 (concurrency)。此范式能够发挥多核硬件的优势以提升程序性能。不过在学习并发编程之前，我们先来看看书中的首个案例研究，也就是图像处理程序包，书中许多地方都会以各种方式使用或引用这个包。

### 3.12 案例研究：图像处理程序包

Python 标准库本身不包含图像处理模块。不过，Tkinter 的 `tk.PhotoImage` 类可以创建、载入并保存图像。（`barchart2.py` 范例程序用的就是这个办法。）然而 Tkinter 只能读写 GIF、PPM 及 PGM 这几种不太常用的图像格式。如果 Python 中的 Tcl/Tk 是 8.6 或更高版本，那么还支持 PNG 格式。即便如此，`tk.PhotoImage` 类也只能用在单线程（也就是主 GUI 线程）里，无法同时处理多张图像。

你当然可以使用 Pillow ([github.com/python-imaging/Pillow](https://github.com/python-imaging/Pillow)) 等第三方图像程序库，或者另选一种 GUI 工具包<sup>⊖</sup>。但笔者决定自己来实现图像处理程序库，这样不仅能用于研究本案例，而且还能以此为基础研究稍后要讲的另一个范例。

这个图像处理程序包应该以一种高效的方式存储图像数据，并且能够直接通过 Python 代码来使用。因此，我们决定把图像中的颜色保存成一维数组。每种颜色（也就是像素点）都用 32 位无符号整数来表示，其中的四个字节分别表示 alpha（透明度）、红、绿、蓝这四个颜色分量，这种表示法有时称为 ARGB 格式。因为使用的是一维数组，所以图像里坐标为  $x$ 、 $y$  的像素在数组中对应的下标就是  $(y \times \text{width}) + x$ 。图 3.8 演示了这种存储方式：这是一张  $8 \times 8$  大小的图像，图中详细展示的那个像素在图像里的坐标是 (5, 1)，因此，它在一维数组里的下标就是  $13 ((1 \times 8) + 5)$ 。

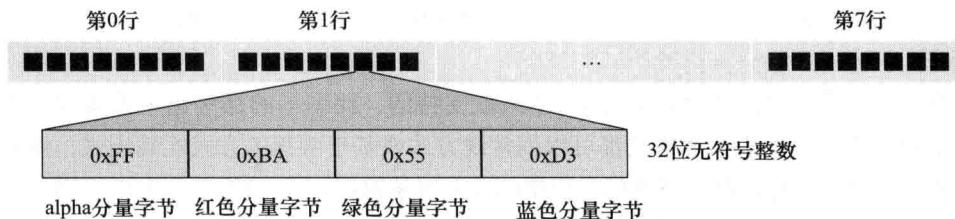


图 3.8 用数组来表示  $8 \times 8$  图像中各像素的颜色

Python 标准库里的 `array` 模块可以处理特定类型的一维数组，所以很符合我们的需求。而第三方的 `numpy` 模块则提供了高度优化过的代码，更擅长处理任意维度的数组，所以在能够使用此模块的情况下，就应该利用这一优势才对。于是，我们这样来设计 `Image`

<sup>⊖</sup> 若想绘制二维数据图，可以使用 `matplotlib` ([matplotlib.org](http://matplotlib.org)) 这个第三方程序包。

包：令其优先使用 numpy 模块，若无法使用该模块，则使用 array 模块。这么做好处是，Image 模块在任何情况下都可以用，但缺点则是无法完全发挥 numpy 的优势，因为编写代码时必须同时顾及 array.array 与 numpy.ndarray。

该程序库要能创建并修改任意图像，同时还要能加载已有图像，并把创建或修改好的图像保存起来。由于加载及保存操作都和具体的图像格式有关，所以我们把处理各种图像时都要用到的那些代码放在一个模块里，然后再为每种图像格式都单独创建一个模块，把实现加载与保存操作所需的代码置于其中。此外，在开发过程中及开发结束之后，如果有人遵照程序包的接口规范编写了新模块，那么程序包还必须自动支持新添加进来的图像格式。

Image 包由四个模块组成。Image/\_\_init\_\_.py 模块提供了通用的功能，其他三个模块分别用来加载与保存三种格式的图像。Image/Xbm.py 适用于 XBM 格式（后缀名为.xbm）的“黑白位图”（monochrome bitmap），Image/Xpm.py 适用于 XPM 格式（后缀名为.xpm）的“彩色位图”（color pixmap），而 Image/Png.py 则适用于 PNG 格式（后缀名为.png）的图像。PNG 格式非常复杂，而且已经有 PyPNG（github.com/drj11/pypng）这个 Python 模块支持它了，因此，如果 Png.py 模块可以使用 PyPNG，那么只需通过适配器模式（参见 2.1 节）对其略加包装即可。

我们首先讲解通用的图像处理模块，也就是 Image/\_\_init\_\_.py。然后看看 Image/Xpm.py 模块，讲解该模块时，我们不谈底层细节。最后，再来完整地学习 Image/Png.py 这个“包装器模块”（wrapper module）。

### 3.12.1 通用的图像处理模块

Image 模块里除了 Image 类之外，还有几个与图像处理有关的便捷函数及常量。

```
try:
    import numpy
except ImportError:
    numpy = None
    import array
```

一个重要的问题是图像数据到底用 array.array 来保存，还是用 numpy.ndarray 保存。照常引入其他模块之后，我们在 try 语句块里引入 numpy 模块。如果引入操作失败了，那么就采取备用方案，将标准库里的 array 模块引进来，以便提供必要的功能，同时还要创建名为 numpy 的变量，并将其值设为 None，因为对程序里的某些地方来说，array 模块与 numpy 模块确实要分开处理。

我们希望用户只需编写一条 import Image 语句就能访问整个图像处理模块。在执行完这条语句后，凡是提供了加载及保存函数的那些模块都可供用户使用，而且与其对应的那些图像格式也将受到支持。这就意味着，用户可以通过下面这段代码来创建并存储一张  $64 \times 64$  的正方形图像。

```
import Image
image = Image.Image.create(64, 64, Image.color_for_name("red"))
image.save("red_64x64.xpm")
```

即便用户没有明确引入 `Image/Xpm.py` 模块，上面这段代码也应该能照常执行才对。而且，如果在初次部署好图像处理程序包之后，`Image` 目录里又有了支持其他图像格式的模块，那么上面这种写法也必须能适用于那些新的图像格式。

为了实现此功能，我们在 `Image/__init__.py` 里编写下面这段代码，试着把与图像格式有关的模块加载进来。

```
_Modules = []
for name in os.listdir(os.path.dirname(__file__)):
    if not name.startswith("__") and name.endswith(".py"):
        name = "." + os.path.splitext(name)[0]
        try:
            module = importlib.import_module(name, "Image")
            _Modules.append(module)
        except ImportError as err:
            warnings.warn("failed to load Image module: {}".format(err))
del name, module
```

上述代码会寻找 `Image` 目录中的模块，如果名字不是 `__init__.py`（或者说，名字不以下划线开头），那么就将其引入，并放进 `_Modules` 列表里。

这段代码首先取得 `Image` 目录在文件系统中的路径，然后遍历该路径下的每个文件。如果文件后缀名是 `.py`，并且不以下划线开头，那么就从文件名中提取模块名称。由于待引入的模块属于 `Image` 包，所以要记得在模块名称前加上“.”。在执行这种“相对引入”（relative import）时，必须把包的名称当作 `importlib.import_module()` 函数的第二个参数传进去。如果此模块能正常引入，那么就把与之对应的 Python 模块对象添加到模块列表里。我们稍后就会讲解这些模块对象的用法。

用完 `name` 与 `module` 变量之后，我们就将其删除，以免扰乱 `Image` 命名空间。

用上面这种办法来引入“插件”（plugin）是非常简单而且很容易理解的，在大多数情况下，这么做都没问题。但如果 `Image` 包放在 `.zip` 文件里，那么这么做就起作用了。（别忘了，Python 也可以引入 `.zip` 文件中的模块，我们只需将 `.zip` 文件的路径添加到 `sys.path` 列表中，并且像普通文件那样引入即可。详情参见：[docs.python.org/dev/library/zipimport.html](https://docs.python.org/dev/library/zipimport.html)。）有一种解决办法是采用标准库的 `pkgutil.walk_packages()` 函数来取代 `os.listdir()`，并修改相应的代码。`walk_packages()` 既可以处理普通的包，也可以处理 `.zip` 文件里的包，此外，它还适用于以 C 语言所写成的扩展以及预编译的字节码文件（后缀名为 `.pyc` 及 `.pyo`）。

```
class Image:
    def __init__(self, width=None, height=None, filename=None,
                 background=None, pixels=None):
        assert (width is not None and (height is not None or
                                       pixels is not None)) or (filename is not None)
        if filename is not None: # From file
            self.load(filename)
        elif pixels is not None: # From data
```

```

        self.width = width
        self.height = len(pixels) // width
        self.filename = filename
        self.meta = {}
        self.pixels = pixels
    else: # Empty
        self.width = width
        self.height = height
        self.filename = filename
        self.meta = {}
        self.pixels = create_array(width, height, background)

```

`Image.__init__()` 方法的签名相当复杂，但是不要紧，我们建议用户采用更为简单的类方法来创建图像，因为那样做更加方便（比如本节早前的范例代码就是用 `Image.create()` 方法来创建图像的）。

```

@classmethod
def from_file(Class, filename):
    return Class(filename=filename)

@classmethod
def create(Class, width, height, background=None):
    return Class(width=width, height=height, background=background)

@classmethod
def from_data(Class, width, pixels):
    return Class(width=width, pixels=pixels)

```

上述三个“工厂式的类方法”（factory class method）都可以创建图像。它们都能在 `Image` 类及其子类上面直接调用（例如：`image = Image.Image.create(200, 400)`）。

`from_file()` 方法可根据文件名来创建图像。`create()` 方法可按照指定的背景色（若没有指定背景色，则为透明）来创建空白图像，而 `from_data()` 方法则可根据图像宽度及一维数组（`array.array` 或 `numpy.ndarray`）中的像素（也就是颜色值）来创建图像。

```

def create_array(width, height, background=None):
    if numpy is not None:
        if background is None:
            return numpy.zeros(width * height, dtype=numpy.uint32)
        else:
            iterable = (background for _ in range(width * height))
            return numpy.fromiter(iterable, numpy.uint32)
    else:
        typecode = "I" if array.array("I").itemsize >= 4 else "L"
        background = (background if background is not None else
                      ColorForName["transparent"])
        return array.array(typecode, [background] * width * height)

```

上面这个函数会创建一个一维数组，其中的元素都是 32 位无符号整数（参见图 3.8）。如果可以使用 `numpy`，并且背景色透明，那么就用 `numpy.zeros()` 工厂函数来创建元素均为 0 的数组（也就是说，数组中的每个元素都是 `0x00000000`）。不管其余三个颜色分量

如何，只要 alpha 分量是 0，那么像素点就是全透明的。若是指定了背景色，则创建一条可以生成 width × height 个值的生成器表达式（这些值都与 background 参数相同），并将其作为迭代器传给 numpy.fromiter() 工厂函数。

要是无法使用 numpy，那么就必须创建 array.array 了。与 numpy 不同，该模块无法精确控制数组中的整数所占据的字节数量，所以我们只能尽量节省空间。首先试着用类型描述符 "I" 来创建数组（数组中的元素都是无符号整数，每个整数最少占用两个字节），然后查询每个数组元素实际占据的字节数量，如果大于或等于四，那么最终就决定创建此类型的数组，否则，就采用类型描述符 "L" 来创建数组（数组中的元素都是无符号整数，每个整数最少占用四个字节）。这种做法所选出来的元素类型既能容纳四个字节，又能尽量保证所占内存空间最小，即便在 64 位计算机上也是如此（64 位计算机一般会用八个字节保存无符号整数，而我们的做法有可能只需四个字节）。确定好类型描述符之后，用它来创建含有 width × height 个元素的数组，数组中的每个元素值都与 background 参数相符。（ColorForName 是个 default dictionary，我们稍后就会讲到。）

```
class Error(Exception): pass
```

上面这个类定义了一种异常类型，名叫 Image.Error。本来我们也可以使用内置的异常类型（例如 ValueError），但现在这种做法的好处在于：Image 程序包的用户既能捕捉到与图像处理有关的异常，又不会错过其他类型的异常。

```
def load(self, filename):
    module = Image._choose_module("can_load", filename)
    if module is not None:
        self.width = self.height = None
        self.meta = {}
        module.load(self, filename)
        self.filename = filename
    else:
        raise Error("no Image module can load files of type {}".format(
            os.path.splitext(filename)[1]))
```

Image.\_\_init\_\_.py 模块并不知道 Image 包到底支持哪几种图像格式，但处理具体事务的模块却知道自己所支持的格式。我们刚才已经把那些模块放在 \_Modules 列表里面了。用不同模块来处理不同图像格式的做法可看作模板方法模式（参见 3.10 节）或策略模式（参见 3.9 节）的变种。

用户通过 filename 参数来指定待加载的图片，load() 方法首先寻找能够加载该图片的模块。如果能找到这种模块，那么就初始化与图像有关的几个实例变量，并命令模块将该图片文件载入。模块的 load() 方法若能顺利执行，则会用图片中每个像素所对应的颜色来填充 self.pixels 数组，并设置好 self.width 与 self.height 的值；若无法执行，则会抛出异常。（3.12.2 节与 3.12.3 节将会讲解具体模块中的 load() 方法。）

```
@staticmethod
def _choose_module(actionName, filename):
```

```

bestRating = 0
bestModule = None
for module in _Modules:
    action = getattr(module, actionPerformed, None)
    if action is not None:
        rating = action(filename)
        if rating > bestRating:
            bestRating = rating
            bestModule = module
return bestModule

```

上述静态方法用于在私有的 `_Modules` 列表中找到能够对文件（其名字以 `filename` 表示）执行特定操作（其名称以 `actionName` 表示）的模块。该方法会遍历每一个加载进来的模块，并试着用内置的 `getattr()` 函数来获取名为 `actionName` 的函数（例如 `can_load()` 或 `can_save()`）。如果找到了这样的函数，那么就以 `filename` 为参数来调用它。

如果模块根本就不支持这种文件，那么 `action` 函数应该返回 0；如果能完美地支持此文件格式，那么会返回 100；若虽能支持，但不尽完美，则返回介于两者之间的值。例如，`Image/Xbm.py` 模块对于扩展名是 `.xbm` 的文件会返回 100，因为它完全支持该格式。而 `Image/Xpm.py` 模块对于扩展名是 `.xpm` 的文件则只会返回 80，因为尽管该模块所测试过的 `.xpm` 都正常，但它却并未完全支持 XPM 规范。

遍历完全部模块之后，返回得分最高者；若找不到合适的模块，则返回 `None`。

```

def save(self, filename=None):
    filename = filename if filename is not None else self.filename
    if not filename:
        raise Error("can't save without a filename")
    module = Image._choose_module("can_save", filename)
    if module is not None:
        module.save(self, filename)
        self.filename = filename
    else:
        raise Error("no Image module can save files of type {}".format(
            os.path.splitext(filename)[1]))

```

上述方法与 `load()` 方法类似：用户通过 `filename` 参数来指定待保存的文件名，而 `save()` 函数则根据文件的扩展名来寻找能够保存这种图像的模块，并执行保存操作。

```

def pixel(self, x, y):
    return self.pixels[(y * self.width) + x]

```

`pixel()` 方法能根据坐标获取像素颜色，并将其按 ARGB 形式返回（也就是将其当成 32 位无符号整数返回）。

```

def set_pixel(self, x, y, color):
    self.pixels[(y * self.width) + x] = color

```

如果用户指定的 `x` 与 `y` 坐标均在图像范围内，那么 `set_pixel()` 方法就把该像素的

颜色设置成 color 参数里的 ARGB 值，否则抛出 IndexError 异常。

Image 模块提供了一些基本的绘制方法，包括 line()、ellipse()、rectangle() 等。我们此处只举一例。

```
def line(self, x0, y0, x1, y1, color):
    Δx = abs(x1 - x0)
    Δy = abs(y1 - y0)
    xInc = 1 if x0 < x1 else -1
    yInc = 1 if y0 < y1 else -1
    δ = Δx - Δy
    while True:
        self.set_pixel(x0, y0, color)
        if x0 == x1 and y0 == y1:
            break
        δ2 = 2 * δ
        if δ2 > -Δy:
            δ -= Δy
            x0 += xInc
        if δ2 < Δx:
            δ += Δx
            y0 += yInc
```

该方法采用“布雷森汉姆直线算法”（Bresenham's line algorithm）来绘制从  $(x_0, y_0)$  到  $(x_1, y_1)$  的线段，此算法只需使用整数运算。由于 Python3 支持 Unicode，所以我们可以把变量名起得更贴切一些，比方说，可以用  $\Delta x$  与  $\Delta y$  来分别表示两个 x 坐标之差及两个 y 坐标之差，用  $\delta$  与  $\delta_2$  来表示误差。

```
def scale(self, ratio):
    assert 0 < ratio < 1
    rows = round(self.height * ratio)
    columns = round(self.width * ratio)
    pixels = create_array(columns, rows)
    yStep = self.height / rows
    xStep = self.width / columns
    index = 0
    for row in range(rows):
        y0 = round(row * yStep)
        y1 = round(y0 + yStep)
        for column in range(columns):
            x0 = round(column * xStep)
            x1 = round(x0 + xStep)
            pixels[index] = self._mean(x0, y0, x1, y1)
            index += 1
    return self.from_data(columns, pixels)
```

上述方法会创建并返回一张新图像，此图像是原图像的缩小版，缩小比例应该是 0.0 至 1.0 之间的数。如果该比例是 0.75，那么新图像的宽度与高度均为原图像的四分之三；如果是 0.5，那么新图像的宽度与高度都是原图像的一半，而其面积则为原图像的四分之一。新图像里的每个像素都对应于原图像某矩形区域内的若干像素，而其颜色则等于那些颜色的平

均值。

图像中像素点的 x 与 y 坐标都是整数，但为了在遍历像素数据的过程中保持精确，我们必须使用浮点运算（也就是用“/”运算符来执行除法，而不能使用“//”运算符）。需要将浮点数表示为整数时，可调用内置的 `round()` 函数。将新图像中每个像素的颜色值都填充到 `pixels` 数组之后，我们调用 `Image.Image.from_data()` 这个便捷的工厂方法（该方法是个类方法）来创建新图像，创建时所传入的 `columns` 参数是提前算好的，它表示新图像的列数。

```
def _mean(self, x0, y0, x1, y1):
    alphaTotal, redTotal, greenTotal, blueTotal, count = 0, 0, 0, 0, 0
    for y in range(y0, y1):
        if y >= self.height:
            break
        offset = y * self.width
        for x in range(x0, x1):
            if x >= self.width:
                break
            alpha, r, g, b = self.argb_for_color(self.pixels[offset + x])
            alphaTotal += alpha
            redTotal += r
            greenTotal += g
            blueTotal += b
            count += 1
    alpha = round(alphaTotal / count)
    r = round(redTotal / count)
    g = round(greenTotal / count)
    b = round(blueTotal / count)
    return self.color_for_argb(alpha, r, g, b)
```

在由 `x0`、`y0`、`x1`、`y1` 所限定的矩形区域内，上述私有方法会把其中每个像素的 alpha、红、绿、蓝这四个颜色分量各自汇总。新图像里的每个像素都相当于原图像里的若干像素，于是我们要将刚才统计出来的那四个汇总值各自与 `count` 相除，并把求出的四个平均值设置成新像素的四个颜色分量。计算过程如图 3.9 所示。

```
MAX_ARGB = 0xFFFFFFFF
MAX_COMPONENT = 0xFF
```

值最小的 32 位 ARGB 颜色是 `0x0`（也就是 `0x00000000`，这是透明色，严格说来，是全透明的黑色）。`Image` 模块定义了上面这两个常量，前者用来表示值最大的 ARGB 颜色（完全不透明的白色），而后者则表示颜色分量的最大取值（255）。

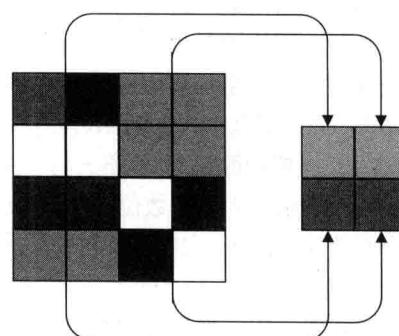


图 3.9 将  $4 \times 4$  的图像按照  $1 : 0.5$  的比例缩小

```

@staticmethod
def argb_for_color(color):
    if numpy is not None:
        if isinstance(color, numpy.uint32):
            color = int(color)
    if isinstance(color, str):
        color = color_for_name(color)
    elif not isinstance(color, int) or not (0 <= color <= MAX_ARGB):
        raise Error("invalid color {}".format(color))
    a = (color >> 24) & MAX_COMPONENT
    r = (color >> 16) & MAX_COMPONENT
    g = (color >> 8) & MAX_COMPONENT
    b = (color & MAX_COMPONENT)
    return a, r, g, b

```

上面这个静态方法（也是模块函数）会根据用户所传入的颜色返回四个分量（分量的值位于 0 至 255 之间）。传入的 `color` 参数可以是 `int`，也可以是 `numpy.uint32`，还可以是由 `str` 所表示的颜色名称。此函数把用户所传入的颜色转换为 `int`，并通过“移位”（bitwise shift, `>>`）及“按位与”（bitwise anding, `&`）操作把四个分量提取到四个字节里。

```

@staticmethod
def color_for_name(name):
    if name is None:
        return ColorForName["transparent"]
    if name.startswith("#"):
        name = name[1:]
        if len(name) == 3: # add solid alpha
            name = "F" + name # now has 4 hex digits
        if len(name) == 6: # add solid alpha
            name = "FF" + name # now has the full 8 hex digits
        if len(name) == 4: # originally #FFF or #FFFF
            components = []
            for h in name:
                components.append([h, h])
            name = "".join(components) # now has the full 8 hex digits
    return int(name, 16)
return ColorForName[name.lower()]

```

上述静态方法（也是模块函数）会把字符串形式的颜色转换成 32 位 ARGB 颜色。`name` 参数如果是 `None`，那么该方法返回透明色。如果 `name` 是以“#”开头的字符串，那么就假定它是个“HTML 风格的颜色”，其格式为“#HHH”、“#HHHH”、“#HHHHHH”或“#HHHHHHHH”，其中的“H”表示十六进制数位。如果 `name` 中的十六进制数位只能构成 RGB 格式的颜色，那么我们会在前面加两个“F”，并将其 `alpha` 通道设为“完全不透明”（`opaque`）。若 `name` 不符合上述两种情况，则从 `ColorForName` 字典里查出与其对应的 ARGB 值，由于 `ColorForName` 是个 `default dictionary`，所以即便待查的键不在字典里，查询操作也总是能够返回值。

```

ColorForName = collections.defaultdict(lambda: 0xFF000000, {
    "transparent": 0x00000000, "aliceblue": 0xFFFF08FF,
    ...
    "yellow4": 0xFF8B8B00, "yellowgreen": 0xFF9ACD32})

```

`ColorForName` 是个 `collections.defaultdict` 对象，可通过颜色名称查出与其对应的 32 位无符号整数值，值里含有颜色的 alpha、红、绿、蓝分量，若颜色名称不在字典中，则默认返回完全不透明的黑色（`0xFF000000`）。虽说 `Image` 程序库的用户可以直接查询该字典，但我们还是推荐使用 `color_for_name()` 函数来做，因为它更方便、更灵活。这些颜色名称取自 X11 的 `rgb.txt` 文件，开头那个“透明色”（“`transparent`”）是笔者自己添加的。

`collections.defaultdict()` 函数的首个参数是工厂方法，其后的参数与普通的 `dict` 一样。如果待查的键不在字典里，那么 `default dictionary` 就会用工厂方法来创建与该键相关联的值，并将其放入字典中。此处所用的工厂方法是个 `lambda` 表达式，它总是返回同一个值（也就是完全不透明的黑色）。构造字典时，本来也可以直接在 `collections.defaultdict()` 函数里传入关键字参数（比如 `transparent=0x00000000`），但 Python 最多只能接受 255 个参数，而要放入字典中的初始颜色却不只 255 个，所以我们先用“{键：值，键，值，值，…}”的写法创建普通字典，然后再把它当成参数转给 `collections.defaultdict()` 函数。用键值对的写法来创建字典时，其数量是没有限制的。

```
argb_for_color = Image.argb_for_color
rgb_for_color = Image.rgb_for_color
color_for_argb = Image.color_for_argb
color_for_rgb = Image.color_for_rgb
color_for_name = Image.color_for_name
```

定义好 `Image` 类之后，我们创建了几个便捷函数，令其指向类中的静态方法。这样做的好处是，用户在执行了 `import Image` 之后，可以直接通过 `Image.color_for_name()` 来调用相关函数，而不用再写成 `Image.Image.color_for_name()` 了（如果有 `Image.Image` 实例，也可以通过 `image.color_for_name()` 来调用）。

至此，我们已经讲完 `Image/_init_.py` 文件中的主要代码了，这些代码是 `Image` 模块的核心。讲解的时候省去了一些内容，比如不太重要的常量、`Image.Image` 类中的几个方法（`rectangle()`、`ellipse()`、`subsample()`）、`size` 属性（该属性返回二元组，其中包含图片宽度与高度）以及很多操作颜色所用的静态方法。现在这个模块已经可以创建、加载、绘制并保存 XBM 及 XPM 格式的图像了，如果安装了 PyPNG 模块，那么还支持 PNG 格式。

接下来我们要讲两个与特定图像格式有关的模块，`Image` 模块要依赖它们。本书不打算讲 `Image/Xbm.py` 模块，一方面是因为其中涉及 XBM 格式的细节，另一方面则是因为通过它所能学到的知识与 `Image/Xpm.py` 模块差不多，所以我们接下来就讲解后者。

### 3.12.2 Xpm 模块概述

如果某模块打算支持特定的图像格式，那么应该提供四个函数，其中的两个是 `can_load()` 与 `can_save()`。若该模块根本无法支持用户所传入的文件格式，则二者均应返

回 0；若完全能够支持，则应返回 100；在虽能支持但却不完美的情况下，应该返回 0 至 100 之间的数。模块还需提供 `load()` 及 `save()` 函数，不过我们假定用户会先通过 `can_load()` 或 `can_save()` 来查询待操作的文件，在得到非 0 的返回值之后，才会调用相应的 `load()` 或 `save()` 函数。

```
def can_load(filename):
    return 80 if os.path.splitext(filename)[1].lower() == ".xpm" else 0

def can_save(filename):
    return can_load(filename)
```

除了某些很少用到的功能之外，`Image/Xpm.py` 模块已经实现了 XPM 规范的大部分要求，所以 `can_load()` 与 `can_save()` 这两个函数的返回值都是 80（这个值的意思是说，该模块对 XPM 格式的支持“尚不完美”(less than perfect)<sup>⊖</sup>。这意味着，如果又多了个能够处理 XPM 图像的模块，而其返回值又大于 80，那么 `Image` 程序库将会改用那个模块来操作 XPM 图像。`Image._choose_module()` 方法负责选取适当的模块，该方法在 3.12.1 节中讲过。)

```
(_WANT_XPM, _WANT_NAME, _WANT_VALUES, _WANT_COLOR, _WANT_PIXELS,
_DONE) = ("WANT_XPM", "WANT_NAME", "WANT_VALUES", "WANT_COLOR",
"WANT_PIXELS", "DONE")
_CODES = "".join((chr(x) for x in range(32, 127) if chr(x) not in '\\'))
```

XPM 是一种纯文本格式（采用 7 位 ASCII 码），我们必须把其中的数据解析出来。这种格式包含了一些“元数据”（metadata，比如宽度、高度、颜色数等）、一张“颜色表”（color table）以及若干像素数据，每个像素都指向颜色表中的一种颜色。由于这些细节与 Python 关系不大，所以我们就不深究了。此处以“硬代码”编个简单的解析器，并用上面这些常量来表示解析器的状态。

```
def load(image, filename):
    colors = cpp = count = None
    state = _WANT_XPM
    palette = {}
    index = 0
    with open(filename, "rt", encoding="ascii") as file:
        for lino, line in enumerate(file, start=1):
            line = line.strip()
            ...
            ...
```

上面是本模块 `load()` 函数的前几行代码。经由 `image` 参数所传进来的对象其类型是 `Image.Image`，此函数会直接设置该对象的 `pixels`、`width`、`height` 这几个 attribute（相关代码没有列出来）。像素数组是用 `Image.create_array()` 函数创建的，所以 `xpm.py` 模块无须关心它究竟是 `array.array` 还是 `numpy.ndarray`，只要该数组是个长度为

---

<sup>⊖</sup> 这两个函数都是通过扩展名来判断文件类型的。另一种办法是读出文件前面几个字节，看它们是否与该类的“魔幻数字”（magic number）相符。比方说，XPM 文件都以 0x2F 0x2A 0x20 0x58 0x50 0x4D 0x20 0x2A 0x2F (“\*XPM\*”) 开头，PNG 文件都以 0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A (“· PNG ···”) 开头。

`width × height` 的一维数组就好。但这同时也意味着模块在访问像素数组时，只能使用这两种数组类型都支持的那些方法。

```
def save(image, filename):
    name = Image.sanitized_name(filename)
    palette, cpp = _palette_and_cpp(image.pixels)
    with open(filename, "w+t", encoding="ascii") as file:
        _write_header(image, file, name, cpp, len(palette))
        _write_palette(file, palette)
        _write_pixels(image, file, palette)
```

XBM 与 XPM 格式都要求在实际文件里写入名称，该名称基于文件名，但同时必须是个有效的 C 语言标识符。我们用 `Image.sanitized_name()` 函数来获取此名称。几乎所有的保存工作都交给私有的辅助函数来完成，而这些函数与本书关系不大，所以没有列出来。

```
def sanitized_name(name):
    name = re.sub(r"\W+", "", os.path.basename(os.path.splitext(name)[0]))
    if not name or name[0].isdigit():
        name = "z" + name
    return name
```

`Image.sanitized_name()` 函数根据参数中的文件名返回调整后的名称，该名称只包含“不带重音符号的拉丁字母”(unaccented Latin letters)、数位及下划线，并且只能以字母或下划线开头。正则表达式中的“`\W+`”可以匹配一个或多个“不能构成英文单词的字符”(non-word character)(也就是不能出现在 C 语言标识符里的字符)。

如果还想令 `Image` 模块支持其他图像格式，那么可在 `Image` 目录下创建对应的模块，并在模块中实现 `can_load()`、`can_save()`、`load()` 及 `save()` 这四个方法，前两个方法要根据给定的文件名返回适当的整数，用以表示能否支持此种文件格式。PNG 是一种很流行的图像格式，但是却非常复杂，好在现有的 PyPNG 模块只需稍加改编，就可以集成到 `Image` 程序库里面。下一小节我们就来讲述具体做法。

### 3.12.3 PNG 包装器模块

PyPNG 模块 ([github.com/drj11/pypng](https://github.com/drj11/pypng)) 对 PNG 图像格式支持得相当好。想令 `Image` 模块支持 PNG 格式，就必须实现模块所要求的那套接口，而 PyPNG 并没有提供接口里的方法。于是，我们将在本节中创建 `Image/Png.py` 模块，通过适配器模式(参见 2.1 节)使 `Image` 模块支持 PNG 图像。上一小节在讲述 `Image/Xpm.py` 模块时，只列出了其中一小部分代码，而这一节则不同，我们会列出 `Image/Png.py` 模块的全部代码。

```
try:
    import png
except ImportError:
    png = None
```

首先用 `import png` 语句试着引入 PyPNG 的 `png` 模块。若操作失败，则创建名为 `png` 的变量，并将其值设为 `None`，稍后我们将检测该变量的值，并据此编写代码。

```

def can_load(filename):
    return (80 if png is not None and
            os.path.splitext(filename)[1].lower() == ".png" else 0)

def can_save(filename):
    return can_load(filename)

```

如果能够引入 png 模块，那么就令 can\_load() 与 can\_save() 函数返回 80，以此表示该模块对 PNG 格式的支持程度。此处返回 80 而不返回 100 是为了使其他模块有机会排在此模块之前。与编写 XPM 模块时的做法类似，我们令 can\_load() 与 can\_save() 函数返回同一个值，它们当然也可以返回不同的值。

```

def load(image, filename):
    reader = png.Reader(filename=filename)
    image.width, image.height, pixels, _ = reader.asRGBAB()
    image.pixels = Image.create_array(image.width, image.height)
    index = 0
    for row in pixels:
        for r, g, b, a in zip(row[::4], row[1::4], row[2::4], row[3::4]):
            image.pixels[index] = Image.color_for_argb(a, r, g, b)
            index += 1

```

这段代码首先以用户所给的 filename 为参数创建 png.Reader 对象，这么做就相当于把 PNG 文件加载到 reader 实例中。然后，提取图像宽度、图像高度以及其中的像素，并丢弃“元数据”。

PyPNG 模块使用 RGBA 格式，而 Image 模块却使用 ARGB 格式，我们必须注意这种区别。通过 png.Reader.asRGBAB() 方法可以解决像素格式转换问题，它能够把所有像素的颜色分量提取到二维数组里，其中的每个一维数组都对应于图像中的某行像素。比方说，某张图片的首行有两个像素，第一个是完全不透明的红色，其右侧是完全不透明的蓝色，那么提取出来的这一行像素数据就会以“0xFF, 0x00, 0x00, 0xFF, 0x00, 0x00, 0xFF, 0xFF”开头。

得到 RGBA 格式的像素数据之后，我们创建大小合适的一维数组，由于数组元素的初始值都是 0，所以其中每个像素都是纯透明的。然后，遍历二维数组中的每个一维数组（也就是图中的每一行像素），并将四种分量各自提取出来。先把 0、4、8、12 等位置上的红色分量提取出来，然后把 1、5、9、13 等位置上的绿色分量提取出来，再把 2、6、10、14 等位置上的蓝色分量提取出来，最后把 3、7、11、15 等位置上的 alpha 分量提取出来。我们通过 Python 内置的 zip() 函数用提取出来的这四组颜色分量来构建四元组。这样的话，首个四元组就是第一行里下标为 0、1、2、3 的那四个元素，第二个四元组就是第一行里下标为 4、5、6、7 的那四个元素，依此类推。我们根据四元组来创建 ARGB 格式的颜色，并将其放入 Image 对象表示像素的一维数组中。

```

def save(image, filename):
    with open(filename, "wb") as file:
        writer = png.Writer(width=image.width, height=image.height,

```

```

        alpha=True)
writer.write_array(file, list(_rgba_for_pixels(image.pixels)))

```

`save()` 函数将大部分工作交由 `png` 模块来做。它首先用适当的元数据来创建 `PngWriter` 对象，然后把 `Image` 里的所有像素都写入其中。由于 `Image` 使用 ARGB 格式而 `png` 模块使用 RGBA 格式，所以我们得编写私有的辅助函数用来转换格式。

```

def _rgba_for_pixels(pixels):
    for color in pixels:
        a, r, g, b = Image.argb_for_color(color)
        for component in (r, g, b, a):
            yield component

```

该函数将遍历由调用者所传进来的数组（在本例中指的就是 `image.pixels`），并将每个颜色值切割成四个分量，然后按照 R、G、B、A 的顺序，通过 `yield` 语句把它们轮流返回给调用者。

上面这些就是 `Image/Png.py` 模块的全部代码了，由于我们把复杂的工作都交给 PyPNG 程序库的 `png` 模块来完成，所以本模块写得很简单。

`Image` 模块提供了一套有用的绘图界面（其中包括 `set_pixel()`、`line()`、`rectangle()`、`ellipse()` 方法），并且能够载入及保存 XBM、XPM 格式的图片，如果安装了 PyPNG，那么还支持 PNG 格式。该模块还提供了 `subsample()` 方法及 `scale()` 方法，前者能以快速而粗略的方式缩小图像，后者则比较平滑。此外，还有一些函数及静态方法，可以非常方便地执行与颜色有关的操作。

`Image` 模块可以在并发环境下使用，比方说，可以用多个线程或进程来创建、加载、绘制并保存图像。这样做要比使用 Tkinter 等办法更方便些，因为 Tkinter 只能在主线程（也就是 GUI 线程）中处理图像。不过，`Image` 模块有个缺点，那就是缩小操作执行得太慢。假如计算机是“多核的”（multi-core），而且有很多张图片需要同时缩小，那么可以通过“并发”（concurrency）来提速，下一章就要讲这个话题。但并发也有其局限，由于“缩小图片”是一种“计算密集型”（CPU-bound）操作，所以并发技术的效果与处理器的数量成正比，例如，在一台四核计算机上，最高速度也只能接近原来的四倍。第五章将会讲解另一种办法（参见 5.3 节），告诉大家如何通过 Cython 显著提升程序的执行速度。

## Python 的高级并发技术

进入 21 世纪后，开发者对并发编程兴趣大增。有三个因素助长了这一趋势。其一，Java 语言令并发编程变得更加普及；其二，拥有多核 CPU 的计算机几乎随处可见；其三，目前大部分语言都支持并发编程。

与“非并发的程序”(nonconcurrent program)相比，并发程序更难编写，也更难维护（有时甚至难度相当大），而且并发程序的运行效率有时比非并发的程序低（甚至低得多）。虽说如此，但优秀的并发程序确实比非并发的程序快很多，所以，为了提升效率，我们还是值得花时间去研究它。

目前大多数编程语言（包括 C++ 及 Java）都直接支持并发，而且其标准库通常还提供了一些封装程度较高的功能。并发可以用多种方式来实现，这些方式之间最重要的区别在于如何访问“共享数据”(shared data)：是通过“共享内存”(shared memory)等方式直接访问，还是通过“进程间通信”(Inter-Process Communication, IPC)等方式间接访问。“基于线程的并发”(threaded concurrency)是指同一个系统进程里有各自独立的若干个线程，它们都在并发执行任务。这些线程一般会依序访问共享内存，以此实现数据共享。程序员通常采用某种“锁定机制”(locking mechanism)来确保同一时间只有一个线程能够访问数据。“基于进程的并发”(process-based concurrency, 或 multiprocessing)是指多个进程独立地执行任务。这些进程一般通过 IPC 来访问共享数据，如果编程语言或程序库支持，那么也可以通过共享内存来实现数据共享。还有一种并发，它基于“并发等待”(concurrent waiting)而非“并发执行”(concurrent execution)，这种方式通常用来实现异步 I/O。

Python 对异步 I/O 提供了某些底层支持 (asyncore 与 asynchat 模块)。Twisted 框架 ([twistedmatrix.com](http://twistedmatrix.com)) 则提供了高层支持。Python 3.4 计划将高层异步 I/O 功能（也包括事

件循环)添加到Python标准库中(参见:[www.python.org/dev/peps/pep-3156](http://www.python.org/dev/peps/pep-3156))。

刚才提到了两种传统的并发方式,也就是基于线程的并发和基于进程的并发,这两种方式Python都支持。Python对多线程的支持方式相当普通,但对于多进程的支持方式则比大多数编程语言或程序库更为高级。此外,Python的多进程与多线程采用同一套机制,使得开发者很容易就能在两套方案中来回切换,至少在不使用共享内存时是如此。

由于有“全局解释器锁”(Global Interpreter Lock, GIL),所以Python解释器在同一时刻只能运行于一个处理器之中<sup>⊖</sup>。Python基本上是用C语言写成的,个别的标准库也是这样,而C语言代码是可以获取及释放GIL的,因此没有这样的限制。即便如此,想通过多线程并发来提升程序速度其效果可能仍然不够理想。

一般来说,计算密集型任务不适合用多线程实现,因为这样做通常比非并发程序还要慢。一种办法是改用Cython(参见5.2节)来编写代码,Cython代码实际上与Python一样,只是多加了一套写法,能够把程序编译成纯C。这种程序执行起来可以比原来快100倍,而用并发则很难达到这样的效果,因为其性能提升程度与处理器核心的数量成正比。如果碰到需要使用并发的场合,而所要执行的任务又是计算密集型的,那么最好避开GIL,改用multiprocessing模块。如果使用多线程,那么同一个进程里的线程在执行时会相互争抢GIL,但如果改用multiprocessing,那么每个进程都是独立的,它们都有自己的Python解释器实例,所以也就不会争夺GIL了。

对于网络通信等“I/O密集型”(I/O-bound)任务来说,并发可以大幅提升程序执行速度。在这种情况下,决定程序效率的主要因素是网络延迟,这与使用线程还是进程来实现并发可能并没有多大关系。

笔者建议,刚开始尽量先按照非并发的方式来写程序。因为这种程序比并发程序简单,写起来快,而且测试起来也比较容易。等写好非并发的程序之后,再看其运行速度是否足够快。如果不快,那么再编写并发版本,然后从运行结果(是否正确)及运行效率两方面来比较二者的优劣。至于并发的类型,笔者推荐采用多进程来实现计算密集型任务,而对于I/O密集型任务来说,多进程与多线程都可以。并发程序的好坏不仅与并发的类型有关,而且也与并发的级别有关。

本书把并发的级别归纳为下列三种:

- **低级并发 (Low-Level Concurrency, 底层并发)**: 就是直接用“原子操作”(atomic operation)所实现的并发。这种并发是给程序库的编写者用的,而应用程序的开发者则不需要它,因为这种写法很容易出错,而且极难调试。虽说Python本身的并发机制一般是用底层操作实现出来的,但开发者不能用Python语言编写这种级别的并发代码。
- **中级并发 (Mid-Level Concurrency)**: 不直接使用原子操作,但却会直接使用

---

<sup>⊖</sup> Jython及其他一些Python解释器无此限制。无论有没有GIL,书中与并发相关的那些范例代码都不会受影响。

“锁”（lock），大多数语言提供的都是这种级别的并发。Python 的 `threading.Semaphore`、`threading.Lock` 及 `multiprocessing.Lock` 等类都支持中级并发。开发应用程序的人一般会使用中级并发，因为他们通常只能使用这个级别的并发功能。

- **高级并发 (High-Level Concurrency)**：既不直接使用原子操作，也不直接使用锁。（锁与原子操作可能在幕后使用，但开发者无须关注这些。）目前已经有编程语言开始支持高级并发了。从 3.2 版本起，Python 提供了支持高级并发的 `concurrent.futures` 模块，此外，`queue.Queue` 及 `multiprocessing` 这两个“队列集合类”（queue collection class）也支持高级并发。

中级并发用起来虽然简单，但却很容易出错，尤其容易出现那种难于追踪而且调试起来非常麻烦的错误，此外，还会导致程序莫名其妙地崩溃或“失去响应”（frozen）。

关键问题出在共享数据上面。如果共享数据可以修改，那么必须用锁来保护，以确保所有线程或进程都要依序存取它（也就是说，同一时刻只能有一个线程或进程访问这份数据）。如果多个线程或进程试图访问同一份共享数据，那么只有其中一个能够获取到，而其他的都会阻塞（也就是进入“空闲”（idle）状态）。这就意味着，当锁定机制生效时，应用程序中只能有一个线程或进程起作用（这就变得和非并发程序类似了），其余的都得等待。由此可见，我们应该尽量少用锁，即便要用，也不要时间太长。最简单的办法是根本不要分享可以修改的数据，这样就不用加锁了，而大部分并发问题也就随之消失了。

有些时候，多个并发的线程或进程确实需要访问同一份数据，但我们不必直接加锁即可解决此问题。一种办法是采用支持“并发访问”（concurrent access）的数据结构。`queue` 模块提供了许多“能在多线程环境下安全使用的”（thread-safe，“线程安全的”）队列，如果是基于多进程的并发，那么可以使用 `multiprocessing.JoinableQueue` 及 `multiprocessing.Queue` 类。前面这种队列能够统一存放待执行的任务，以供并发的线程或进程从中领取，后面那种队列则可用来收集任务的执行结果。在整个过程中，加锁操作都是由数据结构自身负责的。

上面这些支持并发的队列也许不适合用来表述我们所要操作的数据，在这种情况下，如果不想加锁，那么最好的办法就是传递“不可变的数据”（immutable data，比如数字或字符串），或是传递“可变的数据”（mutable data），但却不修改它。假如一定要使用“可变的数据”，那么最安全的办法就是对其“深拷贝”（deep copy），这样可以免去因加锁而带来的开销与麻烦，不过，拷贝操作本身需要占用处理器和内存。另外，如果是多进程并发，那么可以使用支持并发访问的数据类型，特别是 `multiprocessing.Value` 与 `multiprocessing.Array`，前者表示单个可变的值，而后者则是由可变的值所构成的数组，二者都需要通过 `multiprocessing.Manager` 来创建，本章稍后就会演示其用法。

本章前两节通过两个范例程序来研究并发技术，一个是计算密集型程序，另一个是 I/O 密集型程序。我们要使用 Python 的高级并发功能来编写代码，其中既会用到传统的线程安全

队列，又会用到新式的 `concurrent.futures` 模块（Python 3.2 及之后的版本可以使用此模块）。第三节是案例研究，我们将用并发技术编写 GUI（图形用户界面）应用程序，使其在保持响应的同时能够显示进度，而且还带有取消功能。

## 4.1 计算密集型并发

在第三章的案例研究中，我们说过 `Image` 模块可以平滑地缩小图像，但是该操作速度很慢（参见 3.12 节）。本节所研究的范例程序需要处理许多图像，而且打算充分利用多核 CPU 的优势来提升速度。

缩小图像是一种需要大量运算的操作，所以多进程技术应该能够最大限度地提升程序性能，表 4.1 证实了这一点<sup>①</sup>。（在第 5 章的案例研究中，我们将把多进程同 Cython 结合起来，以求进一步提升程序速度。详情参见 5.3 节。）

表 4.1 用各种并发技术来缩小图像所花的时间

| 程 序                            | 并 发         | 执行时间 (秒) | 速度倍数 |
|--------------------------------|-------------|----------|------|
| <code>imagescale-s.py</code>   | 无           | 784      | 基准   |
| <code>imagescale-c.py</code>   | 4 个协程       | 781      | 1.00 |
| <code>imagescale-t.py</code>   | 4 个线程，使用线程池 | 1339     | 0.59 |
| <code>imagescale-q-m.py</code> | 4 个进程，使用队列  | 206      | 3.81 |
| <code>imagescale-m.py</code>   | 4 个进程，使用进程池 | 201      | 3.90 |

`imagescale-t.py` 程序使用了四个线程，其运行结果显然可以说明采用多线程来执行计算密集型任务的效率比非并发程序还要低。这是因为 Python 把所有处理任务都放在了同一个核里，除了执行缩小操作之外，还要在四个线程之间进行“上下文切换”（context switching），这将产生大量开销。采用多进程技术的那两个版本则与之不同，它们都会把任务排布到多个核心上面。使用多进程队列与使用进程池的差别不大，两者都与我们所预估的速度相符（也就是说，速度与 CPU 的核心数成正比）<sup>②</sup>。

所有图像缩小程序都接受命令行参数，我们用 `argparse` 来解析这些参数。所有版本都需要指定如下参数：缩小后的图像尺寸、是否使用平滑缩放（表 4.1 中的程序都使用了平滑缩放）、缩小前的图像所在的文件夹、缩小后的图像所在的文件夹。如果图像本身比指定的尺寸还小，那么就不缩小了，而是直接拷贝过去，不过笔者测试时所用的那些图像都是需要缩小的。采用并发的那几个版本还可以通过 `concurrency` 参数来指定程序所使用的线程或

① 这些时间数据是在一台配有四核 CPU 的计算机上测试得来的，CPU 采用 AMD64 架构，其频率为 3GHz。测试前，CPU 的负载很低。程序处理了 56 张图像，其大小从 1MiB 到 12MiB 不等，这些图像共计 316MiB，处理完后的图像共计 67MiB。

② 与其他操作系统相比，Windows 系统启动新进程的开销特别大。好在 Python 的队列与池都在幕后使用“持久进程池”（persistent process pool），以此避免频繁启动新进程所带来的开销。

进程个数，该参数纯粹是为了调试和统计执行时间而设的。计算密集型程序所使用的线程或进程数量一般与 CPU 的核心数相同。I/O 密集型程序会根据网络带宽状况使用核心数的倍数（比如 2 倍、3 倍、4 倍等）作为线程或进程的数量。为了使范例程序完整，我们列出并发版本的图像缩小程序所用的 `handle_commandline()` 函数。

```
def handle_commandline():
    parser = argparse.ArgumentParser()
    parser.add_argument("-c", "--concurrency", type=int,
                        default=multiprocessing.cpu_count(),
                        help="specify the concurrency (for debugging and "
                             "timing) [default: %(default)d]")
    parser.add_argument("-s", "--size", default=400, type=int,
                        help="make a scaled image that fits the given dimension "
                             "[default: %(default)d]")
    parser.add_argument("-S", "--smooth", action="store_true",
                        help="use smooth scaling (slow but good for text)")
    parser.add_argument("source",
                        help="the directory containing the original .xpm images")
    parser.add_argument("target",
                        help="the directory for the scaled .xpm images")
    args = parser.parse_args()
    source = os.path.abspath(args.source)
    target = os.path.abspath(args.target)
    if source == target:
        args.error("source and target must be different")
    if not os.path.exists(args.target):
        os.makedirs(target)
    return args.size, args.smooth, source, target, args.concurrency
```

应用程序一般都不会向用户提供并发选项，但它可以用于调试、统计执行时间、测试程序等，所以，我们还是提供了这个选项。`multiprocessing.cpu_count()` 函数返回计算机的核心数（比方说，如果计算机使用一个双核 CPU，那么就返回 2；如果使用两个四核 CPU，那么就返回 8）。

`argparse` 模块采用“陈述式”（declarative）的方式来创建命令行解析器。创建好解析器之后，我们用它来解析命令行，并获取其中的参数，然后会简单地判断一下这些参数是否有效（例如，目标文件夹不得与源文件夹相同）。如果目标文件夹尚未建立，那么就新建一个。`os.makedirs()` 函数与 `os.mkdir()` 类似，但后者只能在现存的文件夹下面新建子文件夹，而前者还可以自动创建子文件夹与现有的上层文件夹之间的那些文件夹。

开始讲解代码之前，请大家先注意下面几条重要规则，它们限定了 `multiprocessing` 模块所能使用的 Python 文件：

- 文件名必须是合法的模块名称。比方说，`my-mod.py` 是个合法的 Python 文件名，但文件名中的 `my-mod` 却不能用作模块名称（因为执行 `import my-mod` 会导致语法错误），而 `my_mod.py` 或 `MyMod.py` 这样的名字则可以正常使用。

- 文件里必须含有“入口点函数”（entry-point function，比如 `main()`），而且其结尾处必须调用入口点。例如，以“`if __name__ == "__main__": main()`”结尾。
- 在 Windows 系统中，Python 文件与 Python 解释器（`python.exe` 或 `pythonw.exe`）应该在同一个驱动器里（比如都在 C: 盘）。

下面两个小节将列出两个多进程版本的图像缩小程序：`imagescale-q-m.py` 与 `imagescale-m.py`。二者都会打印进度（也就是打印出每一张正在缩小的图像名称），而且可以取消任务（例如当用户按下 `Ctrl+C` 组合键时，整个任务就终止了）。

### 4.1.1 用队列及多进程实现并发

`imagescale-q-m.py` 程序创建了两个队列，一个用来保存待执行的任务（也就是待缩小的图像），另一个用来收集任务的执行结果。

```
Result = collections.namedtuple("Result", "copied scaled name")
Summary = collections.namedtuple("Summary", "todo copied scaled canceled")
```

`Result` 是个“具名元组”（named tuple），用来存放单张图片的处理结果。其中前两个元素表示拷贝及缩小过的图片数量，对于单张图片来说，这两个值要么是 1、0，要么是 0、1，第三个元素是缩小后的图片名称。`Summary` 也是个具名元组，用来汇总所有图片的处理结果。

```
def main():
    size, smooth, source, target, concurrency = handle_commandline()
    Qtrac.report("starting...")
    summary = scale(size, smooth, source, target, concurrency)
    summarize(summary, concurrency)
```

所有图像缩小程序的 `main()` 函数都一样，它先用自编的 `handle_commandline()` 函数（此函数前面讲过）读出命令行参数。函数所返回的五个参数分别表示：缩小后的图像尺寸、是否采用平滑缩放、缩小前的图片所在的目录、缩小后的图片所在的目录以及程序所使用的线程或进程个数（只有并发版本才会用到这个参数，其默认值等于 CPU 的核心数）。

程序随后告诉用户，现在开始执行图片缩小操作，然后调用 `scale()` 函数，全部任务都会在这个函数里完成。`scale()` 函数会把执行结果汇总，并返回给调用者，我们用 `summarize()` 函数将其打印出来。

```
def report(message="", error=False):
    if len(message) >= 70 and not error:
        message = message[:67] + "..."
    sys.stdout.write("\r{:70}{}".format(message, "\n" if error else ""))
    sys.stdout.flush()
```

本章与并发相关的所有范例程序都要使用控制台，为了用起来方便一些，我们把 `report()` 函数放在了 `Qtrac.py` 模块里。该函数用 `message` 参数中的消息来覆写控制台当前这一行的文字（如果大于 70 个字符，那么就截取前 67 个），并调用 `flush()` 方法将其

立刻打印到控制台里。如果 message 中的消息是错误信息，那么就在后面加上换行符，以防其他消息覆写此信息。report() 函数不会将错误信息截短。

```
def scale(size, smooth, source, target, concurrency):
    canceled = False
    jobs = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()
    create_processes(size, smooth, jobs, results, concurrency)
    todo = add_jobs(source, target, jobs)
    try:
        jobs.join()
    except KeyboardInterrupt: # May not work on Windows
        Qtrac.report("canceling...")
        canceled = True
    copied = scaled = 0
    while not results.empty(): # Safe because all jobs have finished
        result = results.get_nowait()
        copied += result.copied
        scaled += result.scaled
    return Summary(todo, copied, scaled, canceled)
```

本程序采用基于队列的多进程技术来实现并发式图像缩放，而上面这个函数则是程序的核心，其工作流程如图 4.1 所示。函数首先创建了“joinable queue”（支持 join() 方法的队列），把待处理的任务放入其中。开发者可以在 joinable queue 上面调用 join()，该方法会一直阻塞，直到队列变空后才返回。创建好这个队列之后，scale() 函数又创建了一个 nonjoinable queue，用于保存执行结果。接下来，创建缩小操作所需的进程，调用完 create\_processes() 函数之后，这些进程就能够执行任务了，但目前它们尚处于阻塞状态，因为工作队列里还没有放入任务。然后，调用 add\_jobs() 函数，把全部任务都添加到工作队列里。

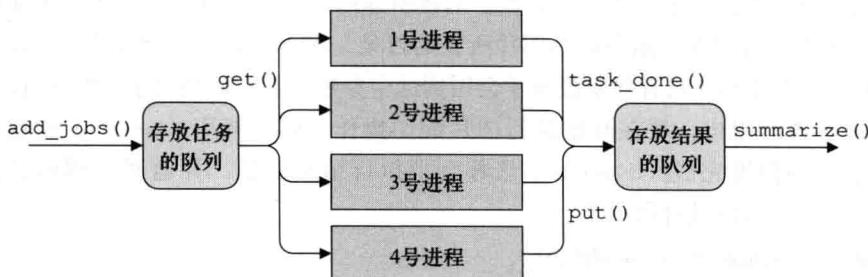


图 4.1 并发处理队列中的任务，并将结果放在另一个队列里

把所有任务都放入队列之后，调用 `multiprocessing.JoinableQueue.join()` 方法，等着队列变空。此方法是在 `try...except` 语句块里调用的，万一用户想取消任务，我们可以从容地处理相关事宜。

所有任务都执行完（或用户取消了整个操作）之后，我们遍历 `results` 队列，以便统

计执行结果。一般情况下，在并发队列上调用 `empty()` 方法所返回的结果是不可靠的，但此处却可以这么做，因为所有的工作进程都已经完工了，该队列的状态不会再改变了。我们在循环体里面调用 `multiprocessing.Queue.get_nowait()` 来获取执行结果，而没有像平常那样调用 `multiprocessing.Queue.get()` 方法，也是由于这个原因。

收集到所有任务的执行结果之后，我们把详细信息放在 `Summary` 具名元组里并返回给调用者。如果程序照常执行了全部任务，那么 `todo` 值是 0，`canceled` 值是 `False`；如果任务取消了，那么 `todo` 值可能不是 0，但 `canceled` 值会是 `True`。

这个函数虽然名叫 `scale()`，但却是个相当通用的“并发式任务执行”函数，它会把待执行的任务分派给各个进程，并收集执行结果。只需稍加改编，就能在其他场合使用。

```
def create_processes(size, smooth, jobs, results, concurrency):
    for _ in range(concurrency):
        process = multiprocessing.Process(target=worker, args=(size,
                                                               smooth, jobs, results))
        process.daemon = True
        process.start()
```

上面这个函数会创建许多进程，这些进程都可以执行任务。创建每个进程时，传入的都是同一个 `worker()` 函数（因为它们要执行的任务都相同，就是缩小图片），此外还传入了任务的细节。细节信息里面包含两个共享队列，一个用于存放任务，另一个用于收集结果。开发者不需要手工给这些队列加锁，因为它们自己会处理同步问题。创建好进程之后，我们将其设为 `daemon`（守护进程）。主进程终止后，所有守护进程也会照常终止（不是守护进程的那些进程还会继续运行，但是在 Unix 系统上，它们会变成“僵尸进程”（zombie process）。

`create_processes()` 函数会创建好所有进程，并将它们都设置成守护进程，每创建完一个进程，就会调用其 `start()` 方法，促使其开始执行 `worker()` 函数。这些进程此时必然都会阻塞，因为任务队列里面还没有任务。它们虽然阻塞了，但主进程与这些进程是各自独立的，所以主进程不会阻塞。于是，主进程很快就能把这些进程都创建完，并且从 `create_processes()` 函数中返回。然后，调用该函数的那段代码会把待处理的任务添加到队列里，于是，刚才那些阻塞的进程就可以领取任务，并继续往下执行了。

```
def worker(size, smooth, jobs, results):
    while True:
        try:
            sourceImage, targetImage = jobs.get()
            try:
                result = scale_one(size, smooth, sourceImage, targetImage)
                Qtrac.report("{} {}".format("copied" if result.copied else
                                           "scaled", os.path.basename(result.name)))
                results.put(result)
            except Image.Error as err:
                Qtrac.report(str(err), True)
        finally:
            jobs.task_done()
```

本来我们可以从 `multiprocessing.Process` 类（或 `threading.Thread` 类）中继承子类，然后用子类实例来实现并发，但是笔者却采用了另一种办法，就是将任务封装在函数里，然后把函数经 `target` 参数传给 `multiprocessing.Process` 对象，这样会稍微简单一些。（创建 `threading.Thread` 对象时，也可以这么做。）

`worker()` 函数是个无限循环，每一轮都试着从共享的工作队列里领取一项任务。此处使用无限循环不会出问题，因为执行 `worker()` 函数的进程都是守护进程，当整个程序终止时，它们也会随之终止。如果队列里没有任务，那么 `multiprocessing.Queue.get()` 方法就会一直阻塞，等获取到任务之后，才会将其返回。返回的值是个二元组，在本例中，二元组的两个元素分别表示源图像和目标图像的名称。

领取任务之后，我们通过 `scale_one()` 函数来缩小（或拷贝）图像，并将操作结果告知用户。然后，把 `result` 对象（该对象的类型是 `Result`）放到保存操作结果的共享队列里。

使用 `joinable queue` 的时候要注意：执行完其中的任务之后，一定要调用 `multiprocessing.JoinableQueue.task_done()` 方法。只有这样，`multiprocessing.JoinableQueue.join()` 方法才能知道队列中的所有任务是不是都执行完了（也就是说，队列里是不是已经没有待执行的任务了）。

```
def add_jobs(source, target, jobs):
    for todo, name in enumerate(os.listdir(source), start=1):
        sourceImage = os.path.join(source, name)
        targetImage = os.path.join(target, name)
        jobs.put((sourceImage, targetImage))
    return todo
```

创建并启动进程之后，它们就处于阻塞状态，所有进程都等着从共享的工作队列里领取任务。

`add_jobs()` 函数会针对每张待处理的图像创建两个字符串：`sourceImage` 表示源图像的完整路径，而 `targetImage` 则表示目标图像的完整路径。函数会把这两个字符串包在二元组里，添加到共享的工作队列中。最后，把待处理的任务数量返回给调用者。

任务队列中有了第一个任务之后，原来处于阻塞状态的某个工作进程就会将其领取，并开始执行此任务，第二个、第三个任务也是如此，最后，所有工作进程都能领到任务。在各进程执行任务时，工作队列里也会有新的任务加进来，只要有进程执行完自己现有的任务，它就可以从队列中再领取一项新任务。这些进程最终会把队列中的任务都领走，它们全部执行完自己的任务之后，就会处于阻塞状态，程序终止时，这些进程也会随之终止。

```
def scale_one(size, smooth, sourceImage, targetImage):
    oldImage = Image.from_file(sourceImage)
    if oldImage.width <= size and oldImage.height <= size:
        oldImage.save(targetImage)
        return Result(1, 0, targetImage)
    else:
```

```

if smooth:
    scale = min(size / oldImage.width, size / oldImage.height)
    newImage = oldImage.scale(scale)
else:
    stride = int(math.ceil(max(oldImage.width / size,
                                oldImage.height / size)))
    newImage = oldImage.subsample(stride)
newImage.save(targetImage)
return Result(0, 1, targetImage)

```

上面这个函数负责执行实际的缩小（或拷贝）操作。它会优先考虑使用 `cyImage` 模块（参见 5.3 节），若无法使用该模块，则使用 `Image` 模块（参见 3.12 节）。假如待处理的图像本身就比目标尺寸小，那么直接将其保存到 `targetImage` 就可以了，此时函数会返回 `Result` 对象。构造该对象时传入的三个参数分别表示本次操作所拷贝的图像数量（1 张）、缩小的图像数量（0 张）以及新图像的名称。如果待处理的图像比目标尺寸大，那么就通过 `oldImage` 对象的 `scale()` 方法或 `subsample()` 方法将其缩小，并把处理后的图像保存成新的图片。在这种情况下，也会返回 `Result` 对象，构造对象时，前两个参数的值会调，表明本次操作没有拷贝图像，但是缩小了一张图像，第三个参数仍然是新图像的名称。

```

def summarize(summary, concurrency):
    message = "copied {} scaled {}".format(summary.copied, summary.scaled)
    difference = summary.todo - (summary.copied + summary.scaled)
    if difference:
        message += "skipped {}".format(difference)
    message += "using {} processes".format(concurrency)
    if summary.canceled:
        message += " [canceled]"
    Qtrac.report(message)
    print()

```

等所有图像都处理完（也就是队列的 `join()` 方法返回的时候），`scale()` 函数会创建 `Summary` 对象，并将其传给 `summarize()` 函数。一般情况下，程序运行完毕后，将会输出下面这种信息（参见第二行）：

```
$ ./imagescale-m.py -S /tmp/images /tmp/scaled
copied 0 scaled 56 using 4 processes
```

如果在 Linux 系统中运行程序，那么只要给命令前面加上 `time`，就能统计出运行时间了。而 Windows 系统则没有自带时间统计命令，不过还是有办法解决的<sup>①</sup>。（在采用多进程的程序里，通过代码来统计时间似乎不够准确。笔者试验过：这么做只能统计出主进程的运行时间，而没有把工作进程所占用的时间计算在内。请注意，Python 3.3 版本的 `time` 模块提供了许多新函数，可以精确计时。）

`imagescale-q-m.py` 与 `imagescale-m.py` 程序都耗时三秒多，两者之间差距不明显，有时前者快，有时后者快。所以，从效果上看，这两个版本没多大区别。

---

① 比方说，可以参考这篇文章的做法：[stack overflow.com/questions/673523/how-to-measure-execution-time-of-command-in-windows-command-line](http://stackoverflow.com/questions/673523/how-to-measure-execution-time-of-command-in-windows-command-line)。

### 4.1.2 用 Future 及多进程实现并发

Python 3.2 新增了 `concurrent.futures` 模块，它提供了一种优雅而高级的方式，可以用多个线程或多个进程来实现并发。本节要讲解 `imagescale-m.py` 程序中的三个函数（其余函数均与上一小节的 `imagescale-q-m.py` 相同）。`imagescale-m.py` 程序会用到 `future` 对象。文档里说，`concurrent.futures.Future` 对象可以“封装 callable，以便异步执行”（encapsulates the asynchronous execution of a callable，参见：[docs.python.org/dev/library/concurrent.futures.html#future-objects](https://docs.python.org/dev/library/concurrent.futures.html#future-objects)）。调用 `concurrent.futures.Executor.submit()` 方法即可创建 `future`，这种对象可以向使用者汇报其状态（取消、运行中、已完成），也可以汇报执行结果或执行时发生的异常。

由于 `concurrent.futures.Executor` 类是个抽象基类，因此不能直接使用。我们必须在它的两个具体子类里选一个来用。`concurrent.futures.ProcessPoolExecutor` 子类可以用多个进程来实现并发。使用这种进程池就意味着相关的 `Future` 对象只能执行或返回“可序列化的对象”（pickleable object），其中当然也包括非嵌套的函数。而 `Executor` 的另一个子类 `concurrent.futures.ThreadPoolExecutor` 则不受此限制，它使用多个线程来实现并发。

从概念上看，用线程池或进程池来实现并发要比用队列简单，如图 4.2 所示。

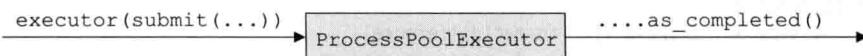


图 4.2 用 `ProcessPoolExecutor` 来处理并发任务，并收集执行结果

```

def scale(size, smooth, source, target, concurrency):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(
        max_workers=concurrency) as executor:
        for sourceImage, targetImage in get_jobs(source, target):
            future = executor.submit(scale_one, size, smooth, sourceImage,
                                      targetImage)
            futures.add(future)
    summary = wait_for(futures)
    if summary.canceled:
        executor.shutdown()
    return summary
  
```

上面这个函数的签名及功能都和 `imagescale-q-m.py` 程序中的那个函数相同，但工作方式却大有区别。它首先创建了空集，并将其赋给 `futures` 变量。然后创建“进程池执行器”（process pool executor），该执行器在幕后会创建许多工作进程。工作进程的具体数量是由“经验算法”（heuristic）<sup>①</sup> 决定的，但笔者在这里手工指定了该值，以便调试程序并统计运行时间。

创建好 `ProcessPoolExecutor` 之后，我们遍历由 `get_jobs()` 函数所返回的每项任务，并将其逐个提交至进程池。调用 `concurrent.futures.ProcessPoolExecutor.`

<sup>①</sup> 也称为“试探法”、“启发式算法”。——译者注

submit()方法时，可以把待执行的任务放在函数里，并通过参数传进去，该方法还能接受一些可选参数，它的返回值是个Future对象。我们把这些Future对象全都放到futures里面。只要进程池里有future，就会开始执行任务。创建好所有的future之后，我们调用自制的wait\_for()函数，把futures变量中的set传进去。wait\_for()函数将会阻塞，直到所有future都执行完毕（或用户取消整个操作）。如果用户取消了整个操作，那我们就手工关闭ProcessPoolExecutor。

```
def get_jobs(source, target):
    for name in os.listdir(source):
        yield os.path.join(source, name), os.path.join(target, name)
```

get\_jobs()函数的功能与上一小节相同，但它这次不将任务添加到队列中，而是变成了生成器函数，该函数会按照需求，通过yield语句来生成任务。

```
def wait_for(futures):
    canceled = False
    copied = scaled = 0
    try:
        for future in concurrent.futures.as_completed(futures):
            err = future.exception()
            if err is None:
                result = future.result()
                copied += result.copied
                scaled += result.scaled
                Qtrac.report("{} {}".format("copied" if result.copied else
                                             "scaled", os.path.basename(result.name)))
            elif isinstance(err, Image.Error):
                Qtrac.report(str(err), True)
            else:
                raise err # Unanticipated
    except KeyboardInterrupt:
        Qtrac.report("canceling...")
        canceled = True
        for future in futures:
            future.cancel()
    return Summary(len(futures), copied, scaled, canceled)
```

创建好所有future对象之后，调用上面这个函数，等待它们执行完毕。concurrent.futures.as\_completed()函数会持续阻塞，直到有future完工（或取消），此时该函数会把这个future返回给调用者。在执行future中的callable任务时，如果发生了异常，那么可以通过Future.exception()方法查询，假如该方法返回None，那就说明没出现异常。在无异常的情况下，我们从future中获取执行结果，并将整个操作的执行进度告知用户。在有异常的情况下，如果异常类型在我们预料之中（也就是说，所抛出的异常是定义在Image模块里面的），那么就告知用户。若发生了预料不到的异常，则用raise语句将其抛出，这种异常表明程序里有逻辑错误，或是用户想通过Ctrl+C组合键取消整个操作。

如果用户想以Ctrl+C组合键来取消整个操作，那么我们就遍历所有future对象，并依次调用其cancel()方法。最后，把目前已经完成的这部分任务汇总起来，报告给用户。

在编写多进程的程序时，可以考虑通过 `concurrent.futures` 或队列来做，这两种办法都比直接加锁要简单得多。在这两种办法之中，`future` 比队列更清晰、更健壮。要想用多线程来取代多进程也很容易，只需把 `concurrent.futures.ProcessPoolExecutor` 改成 `concurrent.futures.ThreadPoolExecutor` 就行了。在多线程的程序里，如果只查询共享数据而不修改，那么需要使用不可变的数据类型，或是对数据执行深拷贝；如果既要查询又要修改，那么就得加锁或使用“线程安全”的数据类型（比如 `queue.Queue`）了，这样才能保证读取及写入操作会按顺序执行，而不会互相重迭。同理，在多进程的程序中，如果只查询共享数据而不修改，那么也需要使用不可变的数据类型，或对数据执行深拷贝；如果既要查询又要修改，那么必须使用“受托管的”（managed）的 `multiprocessing.Value` 及 `multiprocessing.Array`，或是使用 `multiprocessing.Queue`。最理想的情况是：根本不共享数据。若是做不到这一点，那么应该只读取共享数据而不修改它（比方说，使用不可变的数据类型或对数据执行深拷贝）；若是一定要修改共享数据，那么可以使用“能在并发环境下安全使用的”（concurrency-safe，“并发安全的”）队列。这些做法都无须手工加锁，而且比加了锁的代码更容易懂，也更容易维护。

## 4.2 I/O 密集型并发

我们经常需要从因特网中下载一大堆文件或网页。由于网络有延迟，所以通常会同时下载多份文件，这么做要比按顺序逐个下载快很多。

本节要讲解 `whatsnew-q.py` 及 `whatsnew-t.py` 程序。二者都可以下载 RSS feed（RSS 信息源），也就是一种很小的 XML 文档，其中摘录了科技新闻等资讯。这两个程序要从多家网站下载若干个 feed，并将其中的新闻链接统一排布在一份 HTML 网页里。图 4.3 演示了网页中的一部分内容，这张网页的标题是“What's New”。表 4.2 列出了不同版本的程序所耗费的时间<sup>⊖</sup>。尽管“What's new”程序的速度看上去和 CPU 的核心数成正比，但那只是个巧合，这些核心都没有充分利用起来，因为大部分时间耗在了等待网络 I/O 上面。

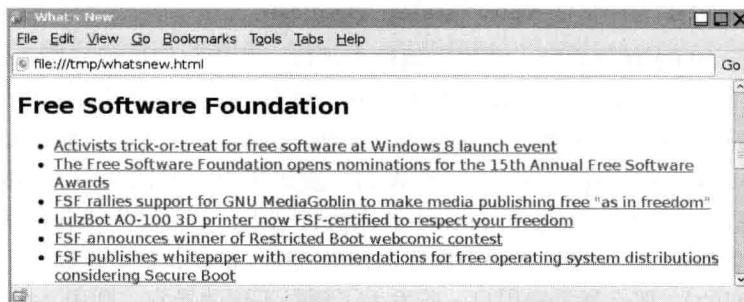


图 4.3 从某个 RSS feed 中获取的科技新闻链接

<sup>⊖</sup> 这些时间数据是在一台配有四核 CPU 的计算机上测试得来的，CPU 采用 AMD64 架构，其频率为 3GHz。测试前，CPU 的负载很低。笔者使用的网络是国内宽带，程序大约要从 200 个网站中下载信息。

表 4.2 下载速度对比

| 程序               | 并发           | 执行时间(秒) | 速度倍数 |
|------------------|--------------|---------|------|
| whatsnew.py      | 无            | 172     | 基准   |
| whatsnew-c.py    | 16个协程        | 180     | 0.96 |
| whatsnew-q-m.py  | 16个进程, 使用队列  | 45      | 3.82 |
| whatsnew-m.py    | 16个进程, 使用进程池 | 50      | 3.44 |
| whatsnew-q.py    | 16个线程, 使用队列  | 50      | 3.44 |
| whatsnew-t.py    | 16个线程, 使用线程池 | 48      | 3.58 |
| gigapixel.py     | 无            | 238     | 基准   |
| gigapixel-q-m.py | 16个进程, 使用队列  | 35      | 6.80 |
| gigapixel-m.py   | 16个进程, 使用进程池 | 42      | 5.67 |
| gigapixel-q.py   | 16个线程, 使用队列  | 37      | 6.43 |
| gigapixel-t.py   | 16个线程, 使用线程池 | 37      | 6.43 |

此表格还对比了另一个程序不同版本的执行时间，这个程序名叫 `gigapixel`，其代码没有列在书中。该程序访问 [www.gigapan.org](http://www.gigapan.org) 网站，获取将近 500 个 JSON 格式的文件，这些文件共计 1.9 MiB，其中的信息是与“超高清图像”（gigapixel image）有关的元数据。各版本的 `gigapixel` 程序其代码都与对应的“what's new”程序相仿，只是 `gigapixel` 程序的提升幅度更大一些。`gigapixel` 程序的效率之所以高，是因为它只访问一个网站，并且那个网站的带宽很大；而“what's new”程序则要访问许多站点，那些站点的带宽也各不相同。

由于网络延迟各有不同，所以并发程序所能提升的速度也会有较大变化，差的时候，只能提升到原来的两倍，而好的时候，则可以提升到 10 倍，甚至更多倍，具体倍数取决于程序所访问的网站、所下载的数据量以及网络连接的带宽。正是由于这一原因，所以多进程与多线程版本之间的差距不是很大，有时前者稍快，有时后者稍快。

表 4.2 的关键之处在于：采用并发技术之后，程序确实执行得比原来快了，只不过每次运行的具体速度有所不同，而且容易受网络环境影响。

### 4.2.1 用队列及线程实现并发

首先来看 `whatsnew-q.py` 程序，该程序要用到多个线程以及两个线程安全的队列。其中一个队列用于存放任务，每项任务都用 URL 来表示，另一个则用来收集结果，每个结果都是二元组。如果首个元素是 `True`，那么第二个元素就是一段 HTML 代码，稍后我们要把这些代码拼成一张 HTML 页面；如果首个元素是 `False`，那么第二个元素就是错误信息。

```
def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    jobs = queue.Queue()
    results = queue.Queue()
    create_threads(limit, jobs, results, concurrency)
    todo = add_jobs(filename, jobs)
    process(todo, jobs, results, concurrency)
```

`main()` 函数会安排好所有事情。首先分析命令行，获取 `limit` 及 `concurrency` 参数，前者表示从每个 URL 中最多读取多少条新闻，后者表示并发级别，我们可以用该参数来调试程序或测试运行时间。然后，程序会告诉用户现在已经开始下载数据了。接下来，把 `whatsnew.dat` 文件的完整路径保存到 `filename` 变量中，该文件列出了每个待下载的 URL 及其标题。

其后，函数创建两个线程安全的队列以及若干个线程，并启动这些线程。它们刚启动时，必然处于阻塞状态，因为还没有工作可以领取，现在我们就把全部任务都放到工作队列里。最后调用 `process()` 函数，它会等待所有任务处理完毕，然后打印出结果。图 4.4 演示了整个程序的并发结构。

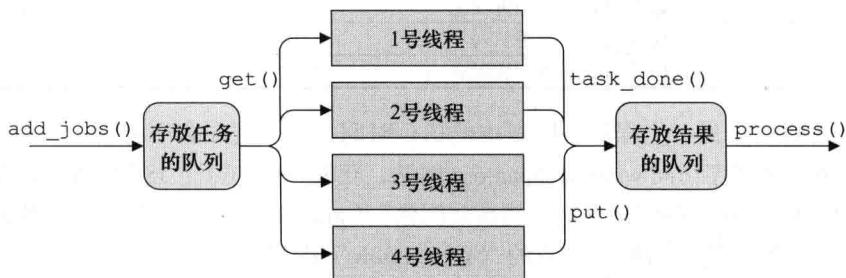


图 4.4 通过队列来处理并发任务，并收集执行结果

顺便说一下，如果要添加的任务很多，或是“添加任务”这项操作本身很耗时，那么最好用单独的线程来做（如果使用多进程技术，那么就在单独的进程里做）。

```

def handle_commandline():
    parser = argparse.ArgumentParser()
    parser.add_argument("-l", "--limit", type=int, default=0,
                        help="the maximum items per feed [default: unlimited]")
    parser.add_argument("-c", "--concurrency", type=int,
                        default=multiprocessing.cpu_count() * 4,
                        help="specify the concurrency (for debugging and "
                             "timing) [default: %(default)d]")
    args = parser.parse_args()
    return args.limit, args.concurrency
  
```

由于“what's new”程序的 I/O 操作很多，所以我们采用默认的并发级别，也就是把线程数量设为 CPU 核心数的倍数，在本例中，是 4 倍<sup>①</sup>。

```

def create_threads(limit, jobs, results, concurrency):
    for _ in range(concurrency):
        thread = threading.Thread(target=worker, args=(limit, jobs,
                                                       results))
        thread.daemon = True
        thread.start()
  
```

① 之所以选定这个倍数，是因为笔者经过测试后发现，使用该值能使程序效率最优。由于每人所使用的计算机配置不同，所以建议大家自己通过试验来决定该倍数。

上面这个函数会按照 `concurrency` 参数的值来创建相同数量的工作线程，创建每个线程时，都指定了 `worker()` 函数以及执行该函数时所需的参数。

与上一节创建进程时的做法一样，我们也把每个线程都设置成“守护线程”（`daemon thread`），这样的话，程序终止的时候，这些线程也会随之终止。创建好线程之后，我们就启动它，此时线程会立即阻塞，因为还没有任务可供领取，但是没关系，阻塞的都是工作线程，而不是程序的主线程。

```
def worker(limit, jobs, results):
    while True:
        try:
            feed = jobs.get()
            ok, result = Feed.read(feed, limit)
            if not ok:
                Qtrac.report(result, True)
            elif result is not None:
                Qtrac.report("read {}".format(result[0][4:-6]))
                results.put(result)
        finally:
            jobs.task_done()
```

工作线程所要执行的 `worker` 函数是个无限循环。此处可以这么做，因为这些线程都是守护线程，当程序终止时，它们也随之终止。

该函数执行到 `jobs.get()` 方法时会一直阻塞，直到队列里有任务可供领取。领到任务之后，它使用 `Feed.read()` 函数来读取 URL 中的文件，该函数定义在我们自编的 `Feed.py` 模块里。每个版本的“What’s new”程序都要依赖这个 `Feed.py` 模块，它会提供迭代器，用以遍历任务文件，而且还会提供 `read()` 函数，用以读取每个 RSS feed。如果读取操作失败了，那么 `ok` 变量就是 `False`，此时我们把 `result` 变量打印出来（变量里面是错误信息）。反之，如果 `ok` 是 `True`，那么 `result` 就应该是由若干个 HTML 字符串所组成的列表。当 `result` 变量里有内容时，我们把列表中的首个元素打印出来（打印的时候省去两端的 HTML 标签），然后将 `result` 变量添加到收集结果所用的队列里。

想在队列上面调用 `join()` 方法，就必须保证 `queue.Queue.get()` 与 `queue.Queue.task_done()` 一一对应。我们用 `try ... finally` 代码块来确保这一点<sup>Θ</sup>。

```
def read(feed, limit, timeout=10):
    try:
        with urllib.request.urlopen(feed.url, None, timeout) as file:
            data = file.read()
            body = _parse(data, limit)
        if body:
            body = ["<h2>{}</h2>\n".format(escape(feed.title))] + body
```

---

<sup>Θ</sup> `queue.Queue` 是个线程安全的队列，而且是个“支持 `join()` 方法的队列”（joinable queue），但是请注意，在多进程环境下，与之对应的队列是 `multiprocessing.JoinableQueue`，而不是 `multiprocessing.Queue`。

```

        return True, body
    return True, None
except (ValueError, urllib.error.HTTPError, urllib.error.URLError,
        etree.ParseError, socket.timeout) as err:
    return False, "Error: {}: {}".format(feed.url, err)

```

`Feed.read()` 函数试着从 URL 中读取并解析数据。如果能够解析出数据，那么就返回 `True`，同时还返回一份含有 HTML 代码片段的列表，其中每个代码片段都包含标题及若干个链接。如果解析不到数据，那么就返回 `True` 和 `None`。若解析时出错，则返回 `False` 及错误信息。

```

def _parse(data, limit):
    output = []
    feed = feedparser.parse(data) # Atom + RSS
    for entry in feed["entries"]:
        title = entry.get("title")
        link = entry.get("link")
        if title:
            if link:
                output.append('<li><a href="{}">{}</a></li>'.format(
                    link, escape(title)))
            else:
                output.append('<li>{}</li>'.format(escape(title)))
        if limit and len(output) == limit:
            break
    if output:
        return ["<ul>"] + output + ["</ul>"]

```

`Feed.py` 模块包含两个版本的私有 `_parse()` 函数。上面列出的这个版本采用第三方的 `feedparser` 模块 ([pypi.python.org/pypi/feedparser](https://pypi.python.org/pypi/feedparser))，能够解析 Atom 与 RSS 格式的信息源。若是无法使用 `feedparser`，则会采取另一种办法（其代码没有列在书中），该办法只能解析 RSS 格式的信息源。

`feedparser.parse()` 函数负责解析具体的 feed。我们只需要遍历该函数的返回值，把标题与每条新闻的链接提取出来，并拼成一份 HTML 代码列表。

```

def add_jobs(filename, jobs):
    for todo, feed in enumerate(Feed.iter(filename), start=1):
        jobs.put(feed)
    return todo

```

`Feed.iter()` 函数的返回值是个二元组，第一个元素是标题，第二个元素是 URL，而 `add_jobs()` 函数则会把这个二元组添加到工作队列里。最后，它会把待处理的任务数量返回给调用者。

在本例中，我们也可以不用 `todo` 变量记录任务数量，而是直接把 `jobs.qsize()` 放在 `return` 语句里返回。但假如 `add_jobs()` 在主线程之外的线程中执行，那么用 `queue.Queue.qsize()` 查出来的任务数量就不可靠了，因为在那种情况下，会有多个线程同时操作队列：有的要从队列中领取任务，有的要给队列里添加任务。

```

Feed = collections.namedtuple("Feed", "title url")

def iter(filename):
    name = None
    with open(filename, "rt", encoding="utf-8") as file:
        for line in file:
            line = line.rstrip()
            if not line or line.startswith("#"):
                continue
            if name is None:
                name = line
            else:
                yield Feed(name, line)
                name = None

```

上面就是 Feed.py 模块的 Feed.iter() 函数。whatsnew.dat 文件应该是  
个 UTF-8 编码格式的纯文本文件，每个信息源都用两行文本表示，第一行是标题（例如：  
The Guardian - Technology），第二行是 URL（例如：[http://feeds.pinboard.in/rss/  
u:guardiantech/](http://feeds.pinboard.in/rss/u:guardiantech/)）。Feed.iter() 函数会忽略空行及评论行（也就是以“#”开头的行）。

```

def process(todo, jobs, results, concurrency):
    canceled = False
    try:
        jobs.join() # Wait for all the work to be done
    except KeyboardInterrupt: # May not work on Windows
        Qtrac.report("canceling...")
        canceled = True
    if canceled:
        done = results.qsize()
    else:
        done, filename = output(results)
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done, todo,
        concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)

```

等所有线程都创建好且所有任务也都添加好之后，main() 函数便会调用上面这个函数，  
而这个函数又会调用 queue.Queue.join() 方法，这个方法将一直阻塞当前线程，只有  
当队列变空（也就是所有任务都处理完毕）或用户取消操作时，该方法才会返回。如果用户  
没有取消整个操作，那么就通过 output() 函数把列表中的链接写入 HTML 文件，然后打  
印一条汇总信息。最后，调用 webbrowser 模块的 open() 函数，在用户默认的浏览器中  
打开 HTML 文件（效果如图 4.3 所示）。

```

def output(results):
    done = 0
    filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
    with open(filename, "wt", encoding="utf-8") as file:
        file.write("<!doctype html>\n")
        file.write("<html><head><title>What's New</title></head>\n")

```

```

file.write("<body><h1>What's New</h1>\n")
while not results.empty(): # Safe because all jobs have finished
    result = results.get_nowait()
    done += 1
    for item in result:
        file.write(item)
    file.write("</body></html>\n")
return done, filename

```

全部任务都处理完之后，将会执行上面这个函数。执行该函数时，`results`参数是个队列，里面存放着任务处理结果。每个处理结果中都包含一份列表，列表里面是许多HTML代码片段（片段中含有标题及若干个链接）。`output()`函数新建名为`whatsnew.html`的文件，并把每个`feed`的标题及新闻链接写入其中。最后，函数把收集到的处理结果的数量（也就是执行完毕的任务数量）及HTML文件的名称返回给调用者。`process()`函数将使用这些数据来打印汇总信息，并且会在用户浏览器中把`output()`函数刚才所写的那份HTML文件打开。

#### 4.2.2 用 Future 及线程实现并发

在 Python 3.2 及后续版本中，我们可以用`concurrent.futures`模块来实现`whatsnew`程序，这样做既不需要使用队列，也不需要手工加锁。本节要讲的`whatsnew-t.py`程序就是用该模块实现的，我们把上一小节已经讲过的那些函数（例如`handle_commandline()`函数及`Feed.py`模块中的函数）省略掉。

```

def main():
    limit, concurrency = handle_commandline()
    Qtrac.report("starting...")
    filename = os.path.join(os.path.dirname(__file__), "whatsnew.dat")
    futures = set()
    with concurrent.futures.ThreadPoolExecutor(
        max_workers=concurrency) as executor:
        for feed in Feed.iter(filename):
            future = executor.submit(Feed.read, feed, limit)
            futures.add(future)
    done, filename, canceled = process(futures)
    if canceled:
        executor.shutdown()
    Qtrac.report("read {}/{} feeds using {} threads{}".format(done,
        len(futures), concurrency, " [canceled]" if canceled else ""))
    print()
    if not canceled:
        webbrowser.open(filename)

```

`main()`函数先创建了空集，稍后我们要把`future`对象放在这个“集合”（`set`）里。接下来，创建“线程池执行器”（`thread pool executor`），这个对象与前面讲过的`ProcessPoolExecutor`相似，只不过它使用线程而非进程来实现并发。在`with`语句块中，我们遍历数据文件里的每

个 feed，并通过 concurrent.futures.ThreadPoolExecutor.submit() 方法来新建相应的 future 对象。该对象会调用 Feed.read() 函数来解析 feed 变量中的 URL，从中获取至多 limit 个链接。创建好 future 之后，我们把它添加到名为 futures 的 set 中。

所有 future 对象都创建完之后，我们调用自编的 process() 函数，该函数会等待这些 future 全部执行完毕（或用户取消整个操作），然后打印出执行结果。如果用户没有取消整个操作，那么还会在用户的网页浏览器中打开程序所生成的 HTML 页面。

```
def process(futures):
    canceled = False
    done = 0
    filename = os.path.join(tempfile.gettempdir(), "whatsnew.html")
    with open(filename, "wt", encoding="utf-8") as file:
        file.write("<!doctype html>\n")
        file.write("<html><head><title>What's New</title></head>\n")
        file.write("<body><h1>What's New</h1>\n")
        canceled, results = wait_for(futures)
        if not canceled:
            for result in (result for ok, result in results if ok and
                           result is not None):
                done += 1
                for item in result:
                    file.write(item)
        else:
            done = sum(1 for ok, result in results if ok and result is not
                      None)
        file.write("</body></html>\n")
    return done, filename, canceled
```

process() 函数先把 HTML 文件的开头写好，然后调用自编的 wait\_for() 函数，等待所有任务处理完毕。如果用户没有取消整个操作，那么函数就遍历各任务的执行结果。执行结果是用二元组来表示的，其中的两个元素可能是 (True, list)、(False, str) 或 (True, None)。process() 函数若发现结果中含有列表（列表里面是标题及若干个链接），则将列表中的元素写入 HTML 文件。

用户如果取消了整个操作，那么我们就统计出程序已经读取了多少个 feed。无论是否取消，函数都要返回三个值，第一个是已经读取的 feed 数量，第二个是 HTML 文件的名称，第三个是 canceled 标志，它表示用户有没有取消整个操作。

```
def wait_for(futures):
    canceled = False
    results = []
    try:
        for future in concurrent.futures.as_completed(futures):
            err = future.exception()
            if err is None:
                ok, result = future.result()
                if not ok:
                    Qtrac.report(result, True)
```

```

        elif result is not None:
            Qtrac.report("read {}".format(result[0][4:-6]))
            results.append((ok, result))
    else:
        raise err # Unanticipated
except KeyboardInterrupt:
    Qtrac.report("canceling...")
canceled = True
for future in futures:
    future.cancel()
return canceled, results

```

上面这个函数会遍历所有 `future` 对象，`for` 语句中的 `concurrent.futures.as_completed()` 方法将导致当前线程阻塞，只有当某个 `future` 对象执行完毕或是用户取消整个操作时，程序才能继续往下执行。在获取到某个 `future` 对象的执行结果之后，函数会根据情况向用户报告错误，或告知用户其中的 `feed` 已经读取出来了。在这两种情况下，函数都要用 `ok` 变量中的 Boolean 值及 `result` 变量的值（这个值可能是字符串列表，也可能是包含错误信息的字符串）来构造二元组，并将这个二元组添加到名为 `results` 的列表中。

如果用户通过 `Ctrl+C` 组合键取消了整个操作，那么我们就调用每个 `future` 对象的 `cancel()` 方法。函数最后返回两个值，一个值表示用户是否取消了整个操作，另一个值是存放运行结果的列表。

与多线程环境类似，`concurrent.futures` 在多进程环境下用起来也很方便。从效率角度来讲，只要在适当的场合下（也就是说，处理的是 I/O 密集型任务，而非计算密集型任务）小心使用，多线程的效率一样会很高。

### 4.3 案例研究：并发式 GUI 应用程序

编写并发式 GUI（图形用户界面，graphical user interface）应用程序是件很麻烦的事情，而要通过 Python 标准的 GUI 程序包 Tkinter 来做就更加困难了。第 7 章简单介绍了如何使用 Tkinter 来编写 GUI 程序，你如果没有开发过 Tkinter 程序，笔者建议先学习那一章，然后再看本节。

要在 GUI 应用程序里实现并发，最容易想到的办法就是使用多线程，但这样做经常会导 致程序变慢，在程序工作量较大时，甚至会使界面“冻结”（frozen，失去响应）。因为 GUI 程序属于计算密集型任务，所以会发生这种状况。另一种办法是使用多进程，但这样做还是会令 GUI 变得很迟钝。

本节讲解 `ImageScale` 程序（该程序在范例代码的 `imagescale` 目录下），其运行效果如图 4.5 所示。它采用了一套精妙的技术，既能实现并发处理，又能令 GUI 及时响应用户操作。程序会报告图片处理进度，而且还带有取消功能。

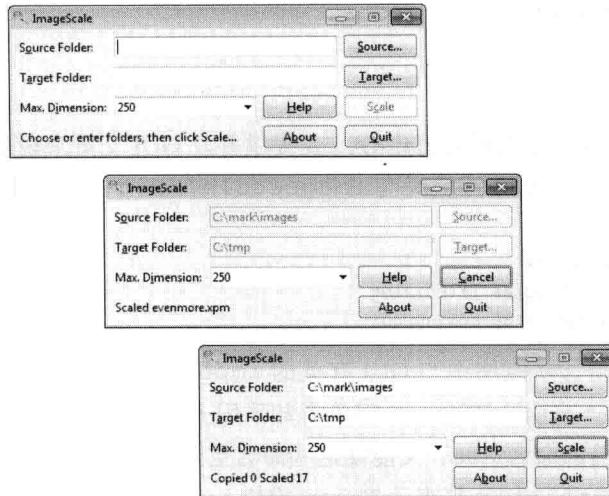


图 4.5 从上到下分别为 ImageScale 程序在缩放图片之前、之中、之后的界面

如图 4.6 所示，本程序会把多线程技术同多进程技术结合起来。它会执行两个线程，一个是主 GUI 线程，另一个是工作线程，而工作线程又会把工作交由进程池来完成。在这种架构下，GUI 总是能及时响应用户操作，因为 GUI 线程和工作线程共享同一个 CPU 内核，所以 GUI 线程能够最大限度地占用“处理器时间”（processor time），工作线程本身几乎无事可做，所以只会占用剩余的那一点点处理器时间。而工作进程则会在各自的核心上执行（假设计算机的 CPU 有多个核心），它们根本不会和 GUI 争抢 CPU 资源。

在 4.1 节中，我们讲解过该程序的控制台版本，也就是 `imagescale-m.py`，那个程序用了大约 130 行代码。而这次要讲的 GUI 版 ImageScale 程序却用了将近 500 行代码，这些代码分布在五个文件中（如图 4.7 所示）。“缩小图像”这一操作本身只用了大约 60 行代码，剩下的全都是 GUI 代码。

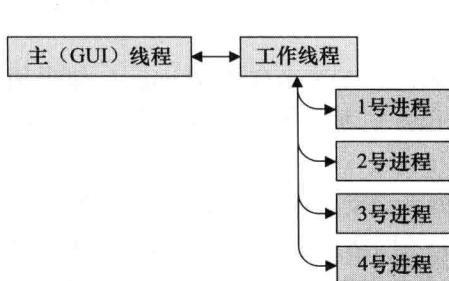


图 4.6 ImageScale 应用程序的并发模型  
(带箭头的线表示通信渠道)

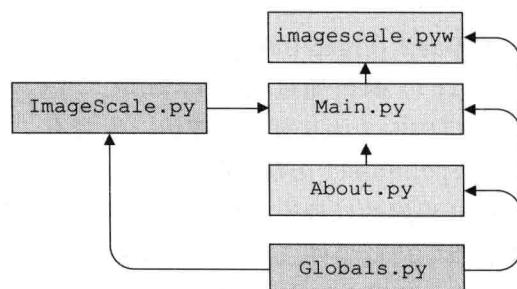


图 4.7 ImageScale 程序各源文件之间的关系  
(箭头表示 import)

本节将讲解与并发 GUI 编程关系最大的那部分代码，同时为了便于理解其原理，我们也

会提供与之相关的其他代码。

### 4.3.1 创建 GUI

本小节要研究的这部分代码对创建 GUI 及提供并发 GUI 支持来说最为重要，它们节录自 `imagescale/imagescale.pyw` 与 `imagescale/Main.py` 文件。

```
import tkinter as tk
import tkinter.ttk as ttk
import tkinter.filedialog as filedialog
import tkinter.messagebox as messagebox
```

上面列出了 `Main.py` 模块中与 GUI 有关的 `import` 语句。有些开发者喜欢用 `from tkinter import *` 来引入 Tkinter，但笔者却采用上面这种做法，这样既能把与 GUI 有关的名称放在它们自己的命名空间，又能使这些命名空间用起来方便一些，比如说，在本例中，我们使用 `tk` 而非 `tkinter` 来表示 Tkinter 模块。

```
def main():
    application = tk.Tk()
    application.withdraw() # hide until ready to show
    window = Main.Window(application)
    application.protocol("WM_DELETE_WINDOW", window.close)
    application.deiconify() # show
    application.mainloop()
```

上面就是 `imagescale.pyw` 程序的入口点。在实际的 `main()` 函数中，还有一些与用户配置及应用程序图标有关的代码，这些代码没有列出来。

这段代码的关键步骤是：首先创建顶级 `tkinter.Tk` 对象（它扮演最终的根对象），并暂时将其隐藏，然后创建窗口实例（在本例中，我们自编了 `tkinter.ttk.Frame` 类的子类，并用这个子类来创建实例），最后启动 Tkinter 事件循环。

我们创建完 `application`，立刻就将其隐藏（所以用户此时看不见程序界面），直到完全安排好应用程序的窗口之后，才重新把它显示出来。这样做既可以防止界面闪烁，又能使用户看不到尚在构建中的窗口。

`main()` 函数通过调用 `tkinter.Tk.protocol()` 方法来告诉 Tkinter：如果用户按下窗口中的关闭按钮（此按钮一般带有“ $\times$ ”图案<sup>⊖</sup>），那么就应该执行自编的 `Main.Window.close()` 方法。**4.3.4** 节将会讲解该方法。

GUI 程序的处理流程与某些服务器程序相似：两者启动之后都会等待事件发生，一旦发生，就响应此事件。在服务器程序里，事件可能表示网络连接或网络通信，但在 GUI 程序里，事件则是由用户或系统所触发的。用户敲击键盘或点击鼠标时，会发生用户触发的事件；计时器超时或收到系统将当前窗口带至前台（比方说，原来有别的窗口覆盖在应用程序

---

<sup>⊖</sup> 在 OS X 系统中，关闭按钮通常是个红色的圆，如果应用程序里有尚未保存的内容，那么圆心处会出现一颗黑点。

的窗口之上，而现在那个窗口移走或关闭了) 的消息时，会发生系统触发的事件。图 4.8 演示了 GUI 事件循环。处理事件的办法我们在 3.1 节中讲过。

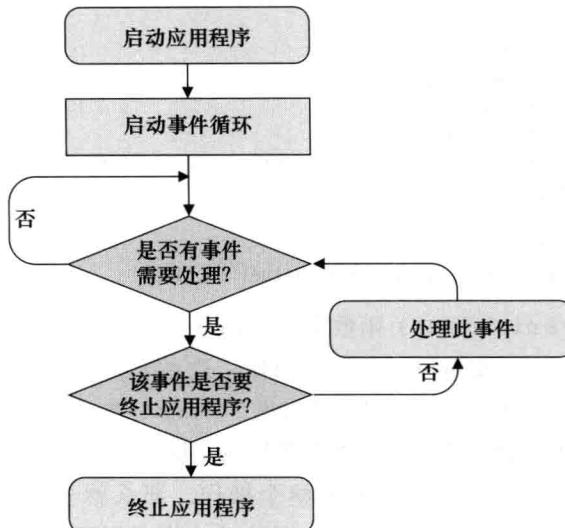


图 4.8 典型的 GUI 事件循环

```

PAD = "0.75m"
WORKING, CANCELED, TERMINATING, IDLE = ("WORKING", "CANCELED",
                                          "TERMINATING", "IDLE")
class Canceled(Exception): pass
  
```

上面这些常量是 ImageScale 程序的 GUI 模块通过 `from Globals import *` 语句引入的。PAD 是排布控件时所用的“内边距”(padding distance)，取值 0.75 毫米。WORKING、CANCELED、TERMINATING、IDLE 这几个常量都是枚举型，表示应用程序的各种状态。Canceled 是个异常，我们稍后将会讲解其用法。

```

class Window(ttk.Frame):
    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.create_variables()
        self.create_ui()
        self.sourceEntry.focus()
  
```

创建 `Window` 对象时，它必须调用基类的 `__init__()` 方法。调用完基类方法之后，我们创建本程序所用的一些变量，然后创建界面本身。最后，把键盘焦点设置到指定源文件夹所用的文本框里。这样的话，用户启动程序之后，直接就可以通过键盘来输入源文件夹的路径。当然了，用户也可以按文本框右边的 Source 按钮，在弹出的对话框里选定文件夹。

笔者不打算列出 `create_ui()` 方法的代码，也不会列出它所调用的 `create_`

`widgets()`、`layout_widgets()`、`create_bindings()` 等方法的代码，因为这些代码都只用于创建 GUI，同并发编程没有关系。（我们会在第 7 章介绍 Tkinter，那时将要举例说明如何创建 GUI。）

```
def create_variables(self):
    self.sourceText = tk.StringVar()
    self.targetText = tk.StringVar()
    self.statusText = tk.StringVar()
    self.statusText.set("Choose or enter folders, then click Scale...")
    self.dimensionText = tk.StringVar()
    self.total = self.copied = self.scaled = 0
    self.worker = None
    self.state = multiprocessing.Manager().Value("i", IDLE)
```

上面是 `create_variables()` 函数中的一部分代码，这些代码与并发编程的关系最大。`tkinter.StringVar` 变量用于保存字符串，这种字符串与用户界面中的控件有关。`total`、`copied`、`scaled` 这三个变量用于计数。`worker` 变量的初始值是 `None`，当用户请求程序处理某些任务时，程序会把工作线程赋给该变量。

如果用户通过点击 `Cancel` 按钮来取消整个操作，那么就会触发程序的 `scale_or_cancel()` 方法（我们稍后会讲解这一过程），该方法会将应用程序的状态设置为 `WORKING`、`CANCELED`、`TERMINATING` 或 `IDLE`；与之类似，如果用户通过点击 `Quit` 按钮来退出应用程序，那么就会触发 `close()` 方法。假如正在缩小图片时用户想取消整个操作或关闭程序，那么我们应该尽快响应这一请求。具体来说，就是把 `Cancel` 按钮的文本换成“`Canceling...`”，并禁用该按钮，同时令工作线程所衍生的那些工作进程不要再继续执行任务了。任务停止之后，还要重新启用 `Scale` 按钮。这一整套设计要求线程与工作进程必须能定期检查程序状态，以判断用户是否想要取消整个操作或退出应用程序。

要想使线程及进程能访问到应用程序的状态，我们可以用变量来表示状态，同时对变量加锁。但这样做意味着每次访问状态变量之前，都要获取这把锁，而且访问完之后，还要把锁释放掉。通过 `context manager` 不难做到这一点，但开发者很容易就会忘记加锁。好在 `multiprocessing` 模块提供了 `multiprocessing.Value` 类，该类实例可以保存特定类型的值，而且可以在并发环境下安全地使用，因为它自己会处理好加锁问题（就像“线程安全的”队列一样）。创建 `Value` 对象时，必须传入类型标识符，此处我们传入“`i`”，表明该值是个 `int`，同时还必须传入初始值，在本例中，我们把 `IDLE` 常量当成初始值，因为应用程序最初的状态就是 `IDLE`。

本例中值得注意的地方是：我们并没有直接创建 `multiprocessing.Value`，而是创建了 `multiprocessing.Manager`，然后通过这个 `Manager` 来创建 `Value`。要想令 `Value` 正常运作，这一步至关重要。（假如要使用多个 `Value` 或 `Array`，那么应该为每个 `Value` 或 `Array` 都创建对应的 `multiprocessing.Manager` 实例，但在本例中无此必要。）

```

def create_bindings(self):
    if not TkUtil.mac():
        self.master.bind("<Alt-a>", lambda *args:
                        self.targetEntry.focus())
        self.master.bind("<Alt-b>", self.about)
        self.master.bind("<Alt-c>", self.scale_or_cancel)
        ...
        self.sourceEntry.bind("<KeyRelease>", self.update_ui)
        self.targetEntry.bind("<KeyRelease>", self.update_ui)
        self.master.bind("<Return>", self.scale_or_cancel)

```

创建 `tkinter.ttk.Button` 对象时，可以关联一项命令（也就是一个函数或方法），当用户点击按钮时，Tkinter 会执行这项命令。关联工作是由 `create_widgets()` 方法完成的，该方法的代码没有列在书中。另外，我们还需向用户提供键盘操作功能。比方说，如果用户点击了 Scale 按钮或在非 OS X 操作系统中按下了键盘的 Enter 键或 Alt+C 键，那么 `scale_or_cancel()` 方法就会执行。

应用程序刚启动时，Scale 按钮应该处于禁用状态，因为此时尚未指定源文件夹及目标文件夹。等两者都指定好之后（用户可以直接在文本框中输入路径，也可以点击文本框右侧的 Source 及 Target 按钮，在弹出的对话框中选择文件夹），我们就需要启用 Scale 按钮了。为了实现这套设计，必须编写 `update_ui()` 方法，令其根据情况来启用或禁用相关控件。只要用户在“Source Folder”或“Target Folder”文本框中输入文字，我们就得调用该方法。

本书范例代码中提供了 `TkUtil` 模块。该模块含有许多工具函数，比如 `TkUtil.mac()`，该函数可用来判断操作系统是不是 OS X。另外，`TkUtil` 模块里还有一些更为通用的功能，比如“‘关于’信息框”（About box）、“模态对话框”（modal dialog）以及其他一些实用功能<sup>⊖</sup>。

```

def update_ui(self, *args):
    guiState = self.state.value
    if guiState == WORKING:
        text = "Cancel"
        underline = 0 if not TkUtil.mac() else -1
        state = "!" + tk.DISABLED
    elif guiState in {CANCELED, TERMINATING}:
        text = "Canceling..."
        underline = -1
        state = tk.DISABLED
    elif guiState == IDLE:
        text = "Scale"
        underline = 1 if not TkUtil.mac() else -1
        state = ("!" + tk.DISABLED if self.sourceText.get() and
                 self.targetText.get() else tk.DISABLED)
    self.scaleButton.state((state,))
    self.scaleButton.config(text=text, underline=underline)

```

---

<sup>⊖</sup> Tkinter 及底层的 Tcl/Tk 8.5 确实能够处理 Linux、OS X 及 Windows 平台之间的某些差异，但有的问题（尤其是涉及 OS X 系统的问题）必须由开发者手工处理。

```

state = tk.DISABLED if guiState != IDLE else "!" + tk.DISABLED
for widget in (self.sourceEntry, self.sourceButton,
               self.targetEntry, self.targetButton):
    widget.state((state,))
self.master.update() # Make sure the GUI refreshes

```

如果程序里发生了可能影响用户界面的情况，那么就会调用上面这个 `update_ui()` 方法。此方法可以由开发者直接调用，也可以由系统在响应某个事件时调用。比如用户通过键盘快捷键选中了按钮，或是用鼠标点击了按钮，那么系统就会调用 `update_ui()` 方法以响应相关事件。在这种情况下，还会向该方法传递若干个参数，不过我们在本例中忽略这些附加参数。

`update_ui()` 方法首先获取 GUI 的状态（状态可能是 WORKING、CANCELED、TERMINATING 或 IDLE），并将其存放到 `guiState` 变量里。其实我们本来不需要创建这个变量，而是可以在每个 `if` 语句里面直接使用 `self.state.value`，但这样做就导致每次访问 `value` 时，该对象都要在底层加锁，为了尽可能缩短加锁时间，最好能像本例这样只访问一次 `value`，并将其值存放到另一个变量里。如果在执行该方法的过程中状态又变了，那也无须担心，因为这会导致系统重新调用此方法。

如果应用程序正在处理图片，那么我们就要把 `scale` 按钮的文本换成 `Cancel`（因为这个按钮身兼“启动任务”与“取消任务”两个职责），并重新启用此按钮。在大部分操作系统中，如果按钮里的某个字符带下划线，那么就表示用户可以通过快捷键来按下此按钮（比方说，用户可以通过 `Alt+C` 键来按下 `Cancel` 按钮），但 OS X 系统却不支持此功能，所以在这种操作系统里，我们必须把 `config()` 方法的 `underline` 参数设置成无效的下标位置。

了解到应用程序的状态之后，我们更新 `scale` 按钮的文本，并对相关字符加下划线，然后根据情况启用或禁用某些控件。最后调用 `update()` 方法，迫使 Tkinter 重新绘制窗口内容，以反映修改之后的样貌。

```

def scale_or_cancel(self, event=None):
    if self.scaleButton.instate((tk.DISABLED,)):
        return
    if self.scaleButton.cget("text") == "Cancel":
        self.state.value = CANCELED
        self.update_ui()
    else:
        self.state.value = WORKING
        self.update_ui()
        self.scale()

```

在程序尚未开始处理图片时，用户可以点击 `scale` 按钮来启动任务，而此时程序会将按钮文本改为 `Cancel`；开始处理图片之后，用户又可以通过该按钮来取消整个操作。如果用户按了 `Alt+C` 键（适用于非 OS X 平台）、`Enter` 键，或通过鼠标点击了 `Cancel` 按钮（也就是刚才那句话里的“`scale` 按钮”），那么就会执行上面这个方法。

如果按钮目前处于禁用状态，那么该方法可以直接返回。（用户虽然无法拿鼠标点击处于禁

用状态的按钮，但他们可能会试着敲击 Alt+C 等键盘快捷键，而这将导致系统调用上述方法。)

该方法若发现按钮处于启用状态且其文本为 Cancel，则会将应用程序的状态改为 CANCELED，并更新用户界面。界面上最显著的变化是：scale 按钮现在已经禁用了，而且其文本变成了 Canceling……。我们稍后就会讲到，在处理图片的过程中，程序要定期检查状态是否改变，如果发现状态变成 CANCELED，那么就会取消整个处理任务，不再继续缩放图片。等取消完整个任务之后，程序又会重新启用 scale 按钮，并将其文本设为 Scale。图 4.5 曾经展示了程序在处理图片之前、之中及之后的界面，而图 4.9 则是程序在取消整个任务之前、之中及之后的界面。

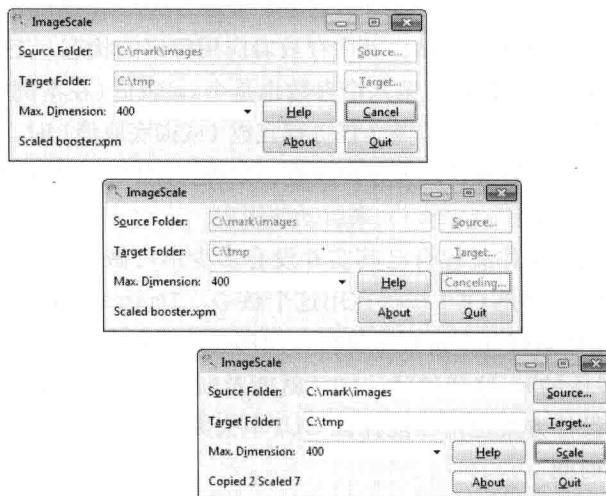


图 4.9 从上到下分别为 ImageScale 程序在取消任务之前、之中、之后的界面

如果按钮的文本是 Scale，那么我们就将应用程序的状态设为 WORKING，并更新用户界面（更新之后，按钮的文本就会变成 Cancel），然后开始缩小图片。

```
def scale(self):
    self.total = self.copied = self.scaled = 0
    self.configure(cursor="watch")
    self.statusText.set("Scaling...")
    self.master.update() # Make sure the GUI refreshes
    target = self.targetText.get()
    if not os.path.exists(target):
        os.makedirs(target)
    self.worker = threading.Thread(target=ImageScale.scale, args=(
        int(self.dimensionText.get()), self.sourceText.get(),
        target, self.report_progress, self.state,
        self.when_finished))
    self.worker.daemon = True
    self.worker.start() # returns immediately
```

scale() 方法首先将三个计数器都设为 0，然后把应用程序的鼠标光标修改成“忙碌”

图样。其后更改 `statusText` 标签的文本，并刷新 GUI，以反映界面中的变化。用户看到刷新之后的界面，就会明白程序已经开始处理图片了。接下来，创建目标目录。如果待创建的目录与其现有的上层目录之间还有目录尚未创建，则一并创建这些目录。

所有事项都准备好之后，我们新建工作线程，并将其赋给 `self.worker`。由于早前的工作线程已经不再受引用，所以系统可将其回收。创建工作线程时我们用的是 `threading.Thread()` 函数，调用该函数时需传入工作线程所要执行的函数以及执行那个函数时所用的参数。工作线程要执行的是 `Image.scale()` 函数，它有六个参数，第一个参数是缩小之后的图片所具备的最大尺寸；第二及第三个参数是源目录和目标目录；第四个参数是 `callable`（在本例中，是 `self.report_progress()`，它是个绑定方法），每执行完一项小任务，都会调用这个 `callable`，第五个参数是 `Value`，其中存放着应用程序的状态，工作进程要定期检测该状态，看用户是否想取消整个操作；第六个参数也是个 `callable`（在本例中，是 `self.when_finished()`，它也是个绑定方法），当整个任务都完成（或彻底取消）时，会调用这个 `callable`。

创建完工作线程后，我们将其设为守护线程，以确保用户在退出应用程序时，线程也能够随之安全退出。最后，调用 `start()` 来启动该线程。

稍后大家就会看到，工作线程自己其实并没有多少事可做，它与 GUI 线程共享同一个 CPU 核心，而大部分时间都是 GUI 线程在用这个核心。`ImageScale.scale()` 函数把工作线程的全部任务都指派给好几个进程来执行，而那些进程各有各的 CPU 核心可供使用（假设计算机的 CPU 是多核的），这样的话，GUI 就能及时响应用户的操作了。即便计算机的 CPU 是单核的，这套设计方案也依然能保证 GUI 不失灵敏性，因为 GUI 线程总能获取到与工作线程相同的 CPU 时间。

### 4.3.2 编写与工作线程配套的 `ImageScale` 模块

我们已经把工作线程所要调用的函数单独放到 `imagescale/ImageScale.py` 模块里面了，本节的范例代码均摘录自该模块。这么做不仅是为了管理起来方便，而且还有更重要的原因。由于 `multiprocessing` 模块要使用 `ImageScale.scale()` 这类函数，所以我们必须把它放在“可供引入的”（`importable`）模块里，而这种模块里的数据又必须“能够序列化”（`pickleable`）才行。包含 GUI 控件或控件子类的模块显然不符合此要求，如果真的把 `ImageScale.scale()` 放在这种模块里引入，那么就会干扰现有的“窗口系统”（`windowing system`）。

`ImageScale` 模块有三个函数，第一个函数名叫 `ImageScale.scale()`，它就是工作线程要执行的那个函数。

```
def scale(size, source, target, report_progress, state, when_finished):
    futures = set()
    with concurrent.futures.ProcessPoolExecutor(
        max_workers=multiprocessing.cpu_count()) as executor:
        for sourceImage, targetImage in get_jobs(source, target):
```

```

future = executor.submit(scale_one, size, sourceImage,
                         targetImage, state)
future.add_done_callback(report_progress)
futures.add(future)
if state.value in {CANCELED, TERMINATING}:
    executor.shutdown()
    for future in futures:
        future.cancel()
    break
concurrent.futures.wait(futures) # Keep working until finished
if state.value != TERMINATING:
    when_finished()

```

上一小节我们曾在 `Main.Window.scale()` 方法里创建了工作线程，并将其赋给 `self.worker`，那个工作线程会执行上面这个 `scale()` 函数。该函数把实际的图像处理工作交给进程池里的进程来做（这就意味着此处采用的是多进程技术，而非多线程技术）。这样设计的好处是：工作线程只需调用 `ImageScale.scale()` 函数，就能把全部图像处理任务都分派到各个进程中去。

`scale()` 函数调用 `ImageScale.get_jobs()`，取出待处理的图像名称及目标图像的名称，然后构建 `future` 对象，该对象将要执行 `ImageScale.scale_one()` 函数，执行的时候还会给函数传入四个参数，分别表示缩放之后的最大“尺寸”（`size`）、源图像、目标图像以及应用程序的状态（该状态放在 `Value` 类型的对象里）。

在 4.2 节中，我们是调用 `concurrent.futures.as_completed()` 函数来等待全部 `future` 对象完工的，但本节不采用这种做法，而是把 `Main.Window.report_progress()` 方法设置成每个 `future` 对象的回调函数，然后通过 `concurrent.futures.wait()` 来等待 `futures` 中的所有 `future` 完工。

把每个 `future` 都添加到 `futures` 之后，我们检测用户是否需要取消整个任务或退出应用程序，如果是这两种情况之一，那么就关闭进程池，并取消所有 `future`。在默认情况下，必须等全部 `future` 对象都完工或取消，`concurrent.futures.Executor.shutdown()` 函数才能返回。

创建好所有 `future` 对象后，`scale()` 函数就阻塞在 `concurrent.futures.wait()` 这个地方了（阻塞的是工作线程，不是 GUI 线程）。如果用户现在决定取消整个操作，那么我们就必须在执行 `future` 对象的 `callable`（本例中，这个 `callable` 指的就是 `ImageScale.scale_one()` 函数）时判断这一情况。

当全部图片都处理完毕或用户彻底取消整个任务之后，如果应用程序没处在退出状态，那么我们就调用 `when_finished()` 这个回调函数，此函数是外界经由 `when_finished` 参数传给 `scale()` 的。执行完 `scale()` 方法之后，工作线程就结束了。

```

def get_jobs(source, target):
    for name in os.listdir(source):
        yield os.path.join(source, name), os.path.join(target, name)

```

上面这个简单的生成器函数会把源图像和目标图像的文件名及完整路径放在二元组里，通过 `yield` 语句返回给调用者。

```
Result = collections.namedtuple("Result", "name copied scaled")
def scale_one(size, sourceImage, targetImage, state):
    if state.value in {CANCELED, TERMINATING}:
        raise Canceled()
    oldImage = Image.Image.from_file(sourceImage)
    if state.value in {CANCELED, TERMINATING}:
        raise Canceled()
    if oldImage.width <= size and oldImage.height <= size:
        oldImage.save(targetImage)
        return Result(targetImage, 1, 0)
    else:
        scale = min(size / oldImage.width, size / oldImage.height)
        newImage = oldImage.scale(scale)
        if state.value in {CANCELED, TERMINATING}:
            raise Canceled()
        newImage.save(targetImage)
        return Result(targetImage, 0, 1)
```

上面这个函数执行实际的缩小或拷贝工作，它优先考虑使用 `cyImage` 模块（参见 5.3 节）来做，如果不可以，就改用 `Image` 模块（参见 3.12 节）。如果能顺利执行完某项任务，那么就返回 `Result` 具名元组；如果在执行过程中用户要取消整个操作，或是要退出应用程序，那么就抛出自编的 `Canceled` 异常。

如果加载、缩小、保存等操作执行到半途而用户决定取消整个操作或退出应用程序，那么上面这个函数会先执行完目前的操作，然后再停止。这意味着用户在发出“取消整个任务”或“退出应用程序”的命令之后，必须等待  $n$  张图片（ $n$  是进程池里的进程数量）完成其加载、缩小、保存等操作，然后整个任务才会彻底取消。由于我们要确保应用程序及时响应用户操作，所以最好的办法也只能是像本例这样：在执行“载入源图像”、“缩小图像”及“保存图像”等“无法中途停止的耗时操作”（expensive non-cancelable computation）之前，先判断用户是不是想取消整个操作或退出应用程序。

不论 `scale_one()` 函数是返回 `Result` 对象还是抛出 `Canceled` 异常，相关的 `future` 都会结束。由于创建每个 `future` 对象时都指定了回调函数，所以此函数会在 `future` 结束时执行，在本例中，回调函数指的就是 `Main.Window.report_progress()` 方法。

### 4.3.3 在 GUI 中显示图像处理进度

本节讲述与 GUI 有关的一些方法，它们的作用是把图像处理进度报告给用户。这些方法节录自 `imagescale/Main.py` 文件。

由于有多个进程要执行 `future`，所以可能导致两个或两个以上进程同时调用 `report_progress()`。但在 `ImageScale` 程序中不可能出现这种情况，因为把 `future` 对象与回调函数

关联起来的那个线程要负责在 future 结束后执行回调函数，具体到本例，就是由工作线程负责调用 report\_progress()。因为本例只有一个工作线程，所以从理论上讲，report\_progress() 方法不可能同时执行许多次。然而这属于实现细节，单就问题本身来说，不给 report\_progress() 方法加锁的做法是非常糟糕的。尽管我们总是想在高层完成并发，但此处却不得不利用加锁这种中层功能。因此，笔者还是决定创建一把锁，以确保 report\_progress() 方法总是依序执行<sup>⊖</sup>。

```
ReportLock = threading.Lock()
```

这把锁写在 Main.py 文件里，本例只有一个方法会用到它。

```
def report_progress(self, future):
    if self.state.value in {CANCELED, TERMINATING}:
        return
    with ReportLock:    # Serializes calls to Window.report_progress()
        self.total += 1 # and accesses to self.total, etc.
        if future.exception() is None:
            result = future.result()
            self.copied += result.copied
            self.scaled += result.scaled
            name = os.path.basename(result.name)
            self.statusText.set("{} {}".format(
                "Copied" if result.copied else "Scaled", name))
            self.master.update() # Make sure the GUI refreshes
```

不论 future 是正常结束还是抛出异常，只要它执行完毕，就会调用上面这个方法。如果用户已经取消了整个操作，那么该方法直接返回即可，因为用户界面稍后自然会由 when\_finished() 方法来更新。如果用户决定退出应用程序，那么就更不用考虑界面的更新问题了，因为应用程序即将终止，到时候用户界面自然随之消失。

report\_progress() 方法的大部分代码都加了锁，如果两个或多个 future 同时结束，那么此时只有其中一个 future 能够完整执行这部分代码，其余的都会阻塞，它们要等着这个 future 把锁释放掉。（我们不用给 self.state 手工加锁，因为该对象的类型是 Value，而这种类型本身就是个“同步类型”（synchronized type），它自己会处理好加锁问题。）对于手工加锁的这部分代码来说，执行的任务越少越好，因为这样可以把阻塞时间降至最小。

加了锁的这部分代码首先递增任务总数，然后看 future 是否抛出异常（比如 Canceled 异常），如果有异常，那么就不用执行后面的代码了。若没有异常，则修改 copied 及 scaled 计数器（如果图片确实经过了缩小处理，那么前者加 0，后者加 1；若图片未经缩小就直接拷贝到目标文件夹，则前者加 1，后者加 0），并修改 GUI 中 statusText 标签的文本。更新 GUI 所用的那些代码一定要加锁，否则，多个 GUI 更新操作就有可能同时执行，那样做也许会导致“未定义的行为”（undefined behavior）。

---

<sup>⊖</sup> always serialized，意思就是必须等某个线程彻底执行完该方法之后，别的线程才能执行，多个线程之间不能穿插交叠。——译者注

```

def when_finished(self):
    self.state.value = IDLE
    self.configure(cursor="arrow")
    self.update_ui()
    result = "Copied {} Scaled {}".format(self.copied, self.scaled)
    difference = self.total - (self.copied + self.scaled)
    if difference: # This will kick in if the user canceled
        result += " Skipped {}".format(difference)
    self.statusText.set(result)
    self.master.update() # Make sure the GUI refreshes

```

工作线程结束的时候，如果用户没有要求终止应用程序，那么上述方法就会执行。此时有可能是全部图片都已经处理完了，也有可能是整个任务都已经取消了。执行此方法时，工作线程及其进程都已结束，所以不需要使用 ReportLock 来加锁。`when_finished()` 方法会把应用程序的状态重新设置成 IDLE，将鼠标光标恢复正常，并修改 `statusText` 标签的文本，告诉用户已经处理完毕的图片数量，如果用户取消了操作，还会给出尚未处理的图片数量。

#### 4.3.4 处理 GUI 程序终止时的相关事宜

要终止并发式的 GUI 程序，不是随便“退出”一下就行了，而是必须先试着停止工作线程，更重要的是停止工作线程所衍生的那些进程。这些进程一定要照常终止，否则就会变成“僵尸进程”，白白占用内存等系统资源。

`imagescale/Main.py` 模块的 `close()` 方法用来处理程序终止时的相关事宜。

```

def close(self, event=None):
    ...
    if self.worker is not None and self.worker.is_alive():
        self.state.value = TERMINATING
        self.update_ui()
        self.worker.join() # Wait for worker to finish
    self.quit()

```

用户点击 Quit 按钮或窗口的关闭按钮（按钮中有“×”图案）之后，上述方法就会执行（在非 OSX 系统中，还可以通过 Alt+Q 键来终止程序）。它首先把用户配置保存起来（这部分代码没有列出），然后判断工作线程是否仍在执行（也就是说，用户是不是在程序正处理图像的时候要求退出）。如果是这种情况，那么该方法会把程序状态设为 TERMINATING，并更新用户界面，使用户知道程序正在取消图片处理任务。由工作线程所衍生的那些进程会定期检查 `Value` 中的状态，一旦看到状态变成 TERMINATING，它们就知道用户想要终止程序，于是会停止工作。而本方法所调用的 `threading.Thread.join()` 则会持续阻塞，直到工作线程及其进程都结束。假如不执行 `join()` 方法，那么就可能会在程序终止之后留下“僵尸进程”，这些进程没有实际用途，但却占据着内存。等 `join()` 方法返回之后，我们调用 `tkinter.ttk.Frame.quit()` 来终止程序。

通过 ImageScale 这个范例，我们学习了怎样将多线程与多进程技术结合起来，以制作一款并发式 GUI 应用程序。这种程序既可以同时处理多项小任务，又能够及时响应用户操作。此外，我们在设计应用程序的架构时，还考虑到了进度汇报与任务取消功能。

编写并发程序时，应该优先考虑“线程安全的队列”以及 `future` 等高层并发功能，并尽量避开加锁这样的中层功能，这种写法要比直接使用底层及中层功能更加容易。此外，一定要保证并发程序的实际速度比非并发版本更快。比方说，在 Python 中处理计算密集型任务时，就不应该采用多线程，因为那样做比非并发的版本还慢。

还需注意的是，不要无意间共享了可变的数据。为了做到这一点，我们在并发程序里应该尽量传递数字及字符串这样的不可变数据。如果确实要使用可变的数据，那么最好能在并发开始之前先把值写进去，并发开始之后，只读取而不写入；另一种办法是，先对可变数据执行深拷贝，然后在拷贝出来的那份数据上面修改。不过正如 ImageScale 程序所示，有时的确需要修改共享数据。在此情况下可以使用 `multiprocessing.Value` 或 `multiprocessing.Array` 等类，这种“受托管的”（`managed`）数据类型无须手工加锁，它们会在底层处理好同步问题。此外，你也可以自己来创建“线程安全的类”，我们将在第 6 章里给出范例（参见 6.2.1 节）。

## 扩充 Python

用 Python 语言编写的大多数程序的运行速度都足够快。在速度达不到要求的情况下，通常可以像上一章那样，用并发来提速。但某些时候，我们确实需要比这还快的运行速度。此时有三种办法提升 Python 程序的速度：第一种办法是使用 PyPy ([pypy.org](http://pypy.org))，它含有内置的“即时编译器”(Just in Time compiler, JIT)；第二种办法是使用 C 或 C++ 代码来编写对“速度要求很高的”(time-critical) 那部分代码；第三种办法是通过 Cython，将 Python 或 Cython 代码编译成 C<sup>⊖</sup>。

一般情况下，我们采用标准的 CPython 解释器来执行 Python 程序，但装了 PyPy 之后，还可以用 PyPy 自己的解释器来执行。对于长期运行的程序来说，提速效果很明显，因为尽管 JIT 编译确实有一些开销，但是编译好之后，所节省的时间却要大于编译时的开销；然而对于执行时间非常短的程序来说，这么做反倒更慢了。

要使用 C 或 C++ 代码，就必须令 Python 程序能访问到这些代码，唯有如此，才能发挥出 C 或 C++ 代码的速度优势，无论是自编代码，还是使用第三方程序库，都需要这么做。如果想自己编写 C 或 C++ 代码，那么比较可行的一种办法是直接使用 Python 的 C 语言接口（参见 [docs.python.org/3/extending](https://docs.python.org/3/extending)）；若想复用既有的 C 或 C++ 代码，则有下面几种方案可供选择。一种方案是用“包装器”(wrapper) 把 C 或 C++ 代码包装起来，并且令包装器根据这些代码产生一套 Python 接口。SWIG ([www.swig.org](http://www.swig.org)) 与 SIP ([www.riverbank-computing.co.uk/software/sip](http://www.riverbank-computing.co.uk/software/sip)) 这两个流行的工具都可以实现此方案。另一种方案是采用 boost::python ([www.boost.org/libs/python/doc/](http://www.boost.org/libs/python/doc/))，该方案只适用于 C++。除了这两种办法之外，还有个新兴的工具，名叫 CFFI (C Foreign Function Interface for Python)，该工具

<sup>⊖</sup> 有一些新式 Python 编译器已经支持此功能了，例如 Numba ([numba.pydata.org](http://numba.pydata.org)) 与 Nuitka ([nuitka.net](http://nuitka.net))。

([bitbucket.org/cffi/cffi](http://bitbucket.org/cffi/cffi)) 虽然面世时间不长，但大家经常使用的 PyPy 项目正是通过它来实现其部分功能的。

### 在 OS X 及 Windows 操作系统中扩充 Python 时的注意事项



尽管本章范例代码只在 Linux 平台测试过，但它们应该都能在 OS X 及 Windows 平台（对于许多使用 `ctype` 与 Cython 的程序员来说，这两种操作系统是他们的主要开发平台）中运行，只是必须根据平台的具体状况稍加调整。大部分 Linux 系统都使用打包好的 GCC 编译器与系统级别的程序库，而且其“字长”(word size) 都设置得与运行该系统的计算机相符。但在 OS X 及 Windows 系统中，情况通常更复杂一些，或者说，情况与 Linux 平台稍有不同。

在 OS X 与 Windows 平台中，编译器与构建 Python 时所用的字长（32 位或 64 位）通常需要和外部共享库（.dylib 或 .DLL 文件）所用的字长以及构建 Cython 代码时所用的字长相符。在 OS X 系统中，可以使用 GCC 做编译器，但目前一般都使用 Clang；在 Windows 系统中，可能会使用某种形式的 GCC，或是由微软等公司发售的某款商业编译器。此外，OS X 与 Windows 系统通常把程序库放在应用程序自己的目录里面，而不是放在整个系统共用的库目录中，头文件也需要单独获取。与平台及编译器有关的配置信息非常繁多，而且出了新版编译器及新版操作系统之后，原有的配置方案可能就会失效，有鉴于此，笔者在本书中只关注 `ctype` 与 Cython 的用法，至于如何在非 Linux 系统上面配置这些工具，请你在准备使用某项具体技术之前，根据自己系统的特殊要求来决定。

刚才描述的那些方案都值得大家深入研究，不过本章专门讲解另外两项技术：一个是 Python 标准库里的 `ctypes` 包 ([docs.python.org/3/library/ctypes.html](http://docs.python.org/3/library/ctypes.html))，另一个是 Cython ([cython.org](http://cython.org))。这两种技术都可以为自编的或第三方的 C 与 C++ 代码提供 Python 接口，而且 Cython 还可以把 Python 和 Cython 代码编译成 C，以便提升程序效率，有些时候，提速效果还是相当显著的。

## 5.1 用 `ctypes` 访问 C 程序库

标准程序库里的 `ctypes` 包可以访问以 C 或 C++ 语言写成的功能。无论这些功能是我们自编的还是由第三方所提供的，甚至是由其他编程语言编译而来的，只要符合“C 语言调用约定”(C calling convention)，就可以使用。此外，编译到“独立共享库”(stand-alone shared library) 里面的功能也可以通过 `ctypes` 包来使用，在 Linux 系统中，这种库文件的后缀名是 .so，在 OS X 系统中是 .dylib，在 Windows 系统中是 .DLL。

我们要创建一个 Python 模块，它可以访问共享库中的某些 C 函数，本节及 5.2.1 节都要用到该模块。待访问的那个共享库叫做 `libhyphen.so`，在某些操作系统里叫做 `libhyphen.uno.so`。（参见前文中提到的注意事项。）此程序库通常会随着 OpenOffice.org 或 LibreOffice 办公套件安装到操作系统中，它里面有个函数，调用者给函数传入单词，而函数会拷贝出一份新的单词，并尽量在词中合适的地方加上“连字符”（`hyphen`），然后返回给调用者<sup>⊖</sup>。这个函数的功能似乎很简单，但其签名却相当复杂，所以在演示 `ctypes` 库的用法时很适合用来做范例。实际上我们需要用三个函数才能完成断字功能：第一个函数用于加载“断字字典”（`hyphenation dictionary`），第二个函数用来执行断字操作，第三个函数用来在完成操作之后释放资源。

使用 `ctypes` 的常见流程是：首先将程序库载入内存，并且把我们要使用的函数保存到相关的引用里面，然后根据需求调用函数。`Hyphenate1.py` 模块就是按这个流程设计的。我们先看看该模块的用法。你可以在交互式 Python 提示符（比方说 IDLE 开发环境就提供了这种提示符）中输入下面这些代码<sup>⊖</sup>：

```
>>> import os
>>> import Hyphenate1 as Hyphenate
>>>
>>> # Locate your hyph*.dic files
>>> path = "/usr/share/hyph_dic"
>>> if not os.path.exists(path): path = os.path.dirname(__file__)
>>> usHyphDic = os.path.join(path, "hyph_en_US.dic")
>>> deHyphDic = os.path.join(path, "hyph_de_DE.dic")
>>>
>>> # Create wrappers so you don't have to keep specifying the dictionary
>>> hyphenate = lambda word: Hyphenate.hyphenate(word, usHyphDic)
>>> hyphenate_de = lambda word: Hyphenate.hyphenate(word, deHyphDic)
>>>
>>> # Use your wrappers
>>> print(hyphenate("extraordinary"))
ex-traor-di-nary
>>> print(hyphenate_de("außergewöhnlich"))
außerge-wöhn-lich
```

在上面这段代码中，我们只调用了一个外部模块中的函数，就是 `Hyphenate1.hyphenate()`，该函数又调用了 `libhyphen` 库中的断字函数。在 `Hyphenate1` 模块中，还有几个私有函数也访问了 `libhyphen` 库中的相关函数。顺便说一下，断字字典的格式与开源的“TEX 排版系统”（`TEX typesetting system`）所使用的格式相同。

实现断字功能所需的全部代码都在 `Hyphenate1.py` 模块里。我们要用到的三个库函

<sup>⊖</sup> 这种操作也称“断字”（`hyphenation`）或“音节划分”（`syllabification`）。——译者注

<sup>⊖</sup> 在某些操作系统中，字典可能存放在 `/usr/share/hyphen` 目录中，请读者根据情况修改 `path` 变量中的路径。也可以把范例代码中提供的字典文件复制到当前目录，并把 `path = "/usr/share/hyph_dic"` 改为 `path = "."`。——译者注

数是：

```
HyphenDict *hnj_hyphen_load(const char *filename);
void hnj_hyphen_free(HyphenDict *hdict);
int hnj_hyphen_hyphenate2(HyphenDict *hdict, const char *word,
    int word_size, char *hyphens, char *hyphenated_word, char ***rep,
    int **pos, int **cut);
```

这几个函数的签名都写在 `hyphen.h` 头文件里。在 C 和 C++ 中，“\*”表示“指针”(pointer)。指针中保存着“内存块”(block of memory)的“内存地址”(memory address)，所谓内存块，指的就是连续的字节块。有些小的块可能只有一个字节，但实际上内存块的大小不限。比方说，由 64 个二进制位所构成的整数就要占用 8 个字节的内存块，而字符串中的每个字符则会占据 1 至 4 个字节(具体字节数与字符在内存中的编码方式有关)，此外还有一些固定开销。

上面列出的第一个函数叫做 `hnj_hyphen_load()`，调用者可以把指向某块内存的指针传给它，而这块内存实际上就是个字节块，里面存放着文件名中的各个“字符”(char)。文件必须是 TEX 格式的断字字典。`hnj_hyphen_load()` 函数会返回指向 `HyphenDict` “结构体”(struct) 的指针，这个结构体不是某个 Python 类的实例，而是一种复杂的“聚合对象”(aggregate object)。所幸我们无须关注 `HyphenDict` 的内部细节，只需在函数间传递指向此结构体的指针即可。

用 C 语言写成的函数所接受的字符串也得是“C 语言风格的字符串”(C-string)，这种字符串实际上就是个指向字符块或字节块的指针。具体来说，有两种传递参数的办法。一种是只传递指针，在这种情况下，指针的最后一个字节必须是 `0x00`('\'0')，这叫做“以 null 结尾的 C-string”(null-terminated C-string)；还有一种办法是同时传递指针及 C-string 所占的字节数。`hnj_hyphen_load()` 函数采用前一种做法，它只接受指针，所以调用者传给它的 C-string 必须以 null 结尾。稍后我们还会看到，调用者给 `ctypes.create_string_buffer()` 函数传递 `str` 之后，函数返回的值也是以 null 结尾的 C-string。

加载进来的每一个断字字典最后都必须释放掉。如果不释放，那么 `hyphenation` 程序库就会无谓地占据着内存。上面列出的第二个函数叫做 `hnj_hyphen_free()`，它的参数是个指向 `HyphenDict` 的指针，函数会把与该指针相关的资源释放掉。此函数没有返回值。释放之后，开发者就不应该再使用这个指针了，这就好比在 Python 里面用 `del` 删掉某变量之后，你就不能再用它了。

第三个函数是 `hnj_hyphen_hyphenate2()`，它会执行实际的断字操作。`hdic` 参数是指向 `HyphenDict` 的指针，这个指针需要通过 `hnj_hyphen_load()` 函数来获取到，而且在使用之前，不能为 `hnj_hyphen_free()` 函数所释放。`word` 参数是待断字的单词，该单词用指针来表示，这个指针指向 UTF-8 编码格式的字节块。`word_size` 是字节块中的字节数。`hyphens` 也是个指向字节块的指针，尽管我们用不到这个参数，但为了使

函数能正常运作，还是得传个有效的指针进去。`hyphenated_word` 是个指向字节块的指针，这个字节块必须足够大，以便容纳添加了连字符之后的单词（实际上，程序库插入的连字符是“=”，而不是“-”），这个字节块也使用 UTF-8 编码格式。字节块中的每个字节其初始值都应该是 `0x00`。`rep` 是个“指向字节块的三重指针”（`a pointer to a pointer to a pointer to a block of bytes`），我们不需要使用此参数，却仍然需要传递有效的指针给它。与之类似，`pos` 与 `cut` 都是“指向 int 的二重指针”（`pointers to pointers to ints`），我们也用不到这两个参数，但还是得把有效的指针传进去。函数的返回值是个 Boolean 型的标志，1 表示失败，0 表示成功。

刚才我们讲解了 `Hyphenate1.py` 模块所封装的 `hyphenation` 库，现在该研究模块本身的代码了。和原来一样，我们还是略去文件开头的 `import` 语句，直接来看查找并加载 `hyphenation` 共享库所用的那段代码。

```
class Error(Exception): pass

_libraryName = ctypes.util.find_library("hyphen")
if _libraryName is None:
    _libraryName = ctypes.util.find_library("hyphen.uno")
if _libraryName is None:
    raise Error("cannot find hyphenation library")

_LibHyphen = ctypes.CDLL(_libraryName)
```

首先创建名为 `Hyphenate1.Error` 的异常类，用来表示与模块相关的异常。这样做可以将其与通用的 `ValueError` 等异常区别开。`ctypes.util.find_library()` 函数用于查找共享库。在 Linux 系统中，函数会给库的名称前面加上 `lib`，后面加上扩展名 `.so`，然后在多个标准路径中寻找名叫 `libhyphen.so` 的库文件。在 OS X 系统中，此函数会搜寻名为 `hyphen.dylib` 的库文件，而在 Windows 系统中，则会查找名叫 `hyphen.dll` 的文件。有的时候，库文件也叫做 `libhyphen.uno.so`，所以如果按原名找不到这个库，那么就按此名称再搜一遍。若是还找不到，则放弃并抛出异常。

如果找到了程序库，那么就用 `ctypes.CDLL()` 函数把它加载到内存中，并且令私有变量 `_LibHyphen` 指向该库。如果想通过 Windows 系统特有的接口来编写只适用于此系统的程序，那么可以用 `ctypes.OleDLL()` 及 `ctypes.WinDLL()` 函数载入 Windows API 库。

将程序库加载进来之后，我们就可以把要用到的那些库函数通过 Python 代码包装起来了。常见的一种做法是：把库函数赋给 Python 变量，并用一份列表来描述函数各参数的类型（列表中的元素都是某种 `ctypes` 类型），然后用一种 `ctypes` 类型来描述返回值的类型。

如果参数个数、参数类型或返回值的类型有错，那么程序会崩溃。与 `ctypes` 相比，`CFI` 包 ([bitbucket.org/cffi/cffi](http://bitbucket.org/cffi/cffi)) 在这方面更为健壮，而且同 PyPy 解释器 ([pypy.org](http://pypy.org)) 结合得更好。

```
_load = _LibHyphen.hnj_hyphen_load
_load.argtypes = [ctypes.c_char_p] # const char *filename
_load.restype = ctypes.c_void_p # HyphenDict *
```

我们在模块中创建名为 `_load()` 的私有函数，调用该函数时，它会在底层执行 `hyphenation` 库中的 `hnj_hyphen_load()` 函数。有了指向库函数的引用之后，还必须指定参数及返回值的类型。这个函数只有一个参数，在 C 语言中，该参数的类型是 `const char *`，而在 Python 语言中，我们则用 `ctypes.c_char_p` (C character pointer, C 语言风格的字符指针) 来表示它。函数的返回值是个指向 `HyphenDict` 结构体的指针。要想用 Python 描述此类型，一种办法是从 `ctypes.Structure` 中继承子类，然后用子类来表示它；另一种办法是把函数返回值声明成 `ctypes.c_void_p` (C void pointer, C 语言风格的 void 指针)，这种指针可以指向任意类型的数据。由于我们只想在函数间传递指针而并不想访问它所指向的内容，因此笔者采用后一种做法。

`_load()` 方法只需要上面这三行代码即可实现（实现该方法之前，还需要用几行代码来寻找并载入程序库），实现好之后，就可以用它来加载断字字典了。

```
_unload = _LibHyphen.hnj_hyphen_free
_unload.argtypes = [ctypes.c_void_p] # HyphenDict *hdict
_unload.restype = None
```

上面这段代码与 `_load()` 方法的套路相同。`hnj_hyphen_free()` 函数只有一个参数，这个参数是指向 `HyphenDict` 结构体的指针。由于我们只是传递该指针，而不关心指针所指的具体内容，所以可把参数定为 `void` 指针，不过要记住：真正传给参数的指针应该是个指向 `HyphenDict` 结构体的指针。此函数没有返回值，在 Python 中为了描述这一现象，我们把 `restype` 设为 `None`。（假如不指定 `restype`，那么 Python 就认定函数返回 `int`。）

```
_int_p = ctypes.POINTER(ctypes.c_int)
_char_p_p = ctypes.POINTER(ctypes.c_char_p)

_hyphenate = _LibHyphen.hnj_hyphen_hyphenate2
_hyphenate.argtypes = [
    ctypes.c_void_p,      # HyphenDict *hdict
    ctypes.c_char_p,      # const char *word
    ctypes.c_int,          # int word_size
    ctypes.c_char_p,      # char *hyphens [not needed]
    ctypes.c_char_p,      # char *hyphenated_word
    _char_p_p,            # char ***rep [not needed]
    _int_p,                # int **pos [not needed]
    _int_p]                # int **cut [not needed]

_hyphenate.restype = ctypes.c_int # int
```

在待包装的几个函数中，上面这个函数最为复杂。`hdic` 参数是指向 `HyphenDict` 结构体的指针，我们在 Python 中把它设定为 C 语言风格的 `void` 指针。第二个参数名叫 `word`，表示待断字的单词，它是个指向字节块的指针，此处我们用 C 语言风格的字符指针来表示。第三个参数是 `word_size`，表示单词在内存中占据的字节数，我们用整数（也就

是 `ctypes.c_int`) 来表示它。第四个参数名叫 `hyphens`, 它是个“缓冲区”(buffer), 我们不需要使用此参数。第五个参数 `hyphenated_word` 也是个 C 语言风格的字符指针。对于第六个参数来说, 由于 `ctypes` 中没有内置“指向字符(即字节)的二重指针”, 所以我们自创了 `_char_p_p` 类型, 该类型是个指针类型, 而这种指针又指向 C 语言风格的字符指针。第七和第八个参数都是指向整数的二重指针, 我们对这两个参数的处理方法与第六个参数类似<sup>②</sup>。

严格来说, 不需要为此函数指定 `restype`, 因为它的返回值是个整数, 而这正与 Python 默认的处理方式相同。但此处为了明确, 我们还是将其手工指定为 `ctypes.c_int`。

我们给 `hyphenation` 程序库里的函数都创建了与之对应的私有包装函数, 使用户无须知晓底层实现细节即可使用 `Hyphenate` 模块。为了配合这一目标, 我们还提供了 `hyphenate()` 这个公有函数, 用户可以把待断字的单词、断字字典以及断字字符传给它。为提升效率, 我们只会把用到的字典加载一次, 而且程序终止的时候, 还会把加载进来的所有字典都释放掉。

```
def hyphenate(word, filename, hyphen="-"):
    originalWord = word
    hdict = _get_hdict(filename)
    word = word.encode("utf-8")
    word_size = ctypes.c_int(len(word))
    word = ctypes.create_string_buffer(word)
    hyphens = ctypes.create_string_buffer(len(word) + 5)
    hyphenated_word = ctypes.create_string_buffer(len(word) * 2)
    rep = _char_p_p(ctypes.c_char_p(None))
    pos = _int_p(ctypes.c_int(0))
    cut = _int_p(ctypes.c_int(0))
    if _hyphenate(hdict, word, word_size, hyphens, hyphenated_word, rep,
                  pos, cut):
        raise Error("hyphenation failed for '{}'".format(originalWord))
    return hyphenated_word.value.decode("utf-8").replace("=", hyphen)
```

用户把待断字的单词通过 `word` 参数传给 `hyphenate()` 函数, 而函数则首先声明了 `originalWord` 变量, 令其指向 `word`, 如果断字时发生异常, 那么在错误信息里会用到这个 `originalWord`。然后获取断字字典。私有的 `_get_hdict()` 函数会根据参数里所指定的字典文件名来寻找字典, 并返回指向 `HyphenDict` 结构体的指针。如果字典原先就已经载入了, 那么 `_get_hdict()` 会返回早前创建好的指针; 如果是头一次使用这个字典, 那么就将其加载进来, 并把指向该字典的指针存储起来以备后用, 接着将指针返回给调用者。

传给 `_hyphenate()` 函数的单词必须是 UTF-8 编码格式的字节块, 通过 `str.encode()` 方法很容易就能将字符串转换成这种字节块。此外还需把单词所占的字节数传

---

<sup>②</sup> 作者此处对最后三个参数的解读方式同 `hnj_hyphen_hyphenate2()` 函数的实际情况略有出入, 不过这三个参数在本例中并未使用。——译者注

进去：计算好这个数量之后，我们把 Python 语言的 `int` 转换成 C 语言的 `int`。Python 原生的 `bytes` 对象是不能传给 C 函数的，所以必须创建“字符串缓冲区”（`string buffer`，实际上就是 C 语言的字符块），把表示单词所用的那些字节放在这个缓冲区里。`ctypes.create_string_buffer()` 可以根据给定的 `bytes` 对象或缓冲区大小来创建 C 语言的字符块。尽管用不到 `hyphens` 参数，但也必须传入适当的值才行，`hyphenation` 库的开发文档里面说该参数必须是个指向 C 语言字符块的指针，而且字符块的长度必须比单词所占字节数多 5。于是我们通过 `create_string_buffer()` 来创建适当的字符块。断好字的单词将会放在另一个 C 语言字符块里，而这个字符块需要由调用者传给 `_hyphenate()`，所以我们还需创建大小合适的 `hyphenated_word`。开发文档建议该字符块的长度应为原始单词所占字符块长度的两倍。

`rep`、`pos`、`cut` 这三个参数我们都用不到，但为了使函数正常运作，还是需要传入适当的值。`rep` 参数是个指向 C 语言字符块的三重指针，我们先创建“指向空块的指针”，也就是 C 语言里的“空指针”（`null pointer`）该指针并未指向任何内容，然后创建二重指针，令其指向该空指针，再把整个三重指针赋给 `rep` 变量。接下来，我们为 `pos` 与 `cut` 参数分别创建指向整数 0 的二重指针<sup>⊖</sup>。

所有参数都设置好之后，我们调用私有的 `_hyphenate()` 函数（该函数又会调用 `hyphenation` 库中的 `hnj_hyphen_hyphenate2()` 函数，以完成真正的断字工作），如果返回值不是 0，那么就说明断字操作执行失败，此时 `hyphenate()` 函数将抛出异常。若返回值是 0，则表明断字操作顺利执行完毕，我们在处理过的单词上面通过 `value` 属性获取原始的 `bytes`（这个 `bytes` 是以 `null` 结尾的，也就是说，`bytes` 里最后一个 `byte` 的值是 `0x00`）。然后按照 UTF-8 格式将其解码成 `str`，并且以用户选定的连字符号（默认是“-”）来替换 `hyphenation` 库所插入的“=”号。最后，`hyphenate()` 函数将处理完毕的字符串返回。

请注意，如果 C 函数不使用以 `null` 结尾的字符串，而是采用 `char *` 及 `size` 来表示字符串，那么我们就应该通过 `value` 属性来获取字符串中的原始字节了，而是应该改用 `raw` 属性来获取。

```
_hdicForFilename = {}

def _get_hdic(filename):
    if filename not in _hdicForFilename:
        hdic = _load(ctypes.create_string_buffer(
            filename.encode("utf-8")))
        if hdic is None:
            raise Error("failed to load '{}'".format(filename))
        _hdicForFilename[filename] = hdic
    hdic = _hdicForFilename.get(filename)
    if hdic is None:
        raise Error("failed to load '{}'".format(filename))
    return hdic
```

---

⊖ 作者此处所描述的做法与范例代码略有区别。——译者注

上面这个私有的“辅助函数”(helper function)会把指向 HyphenDict 结构体的指针返回给调用者，如果早前已经加载过相关字典，那么它会直接返回原来创建好的指针。

如果断字字典的文件名不在 `_hdicForFilename` 这个 dict 里，那么就表明该字典是第一次用到，必须立刻加载进来。由于传给 `_load()` 的文件名是个 C 语言的 `const char *`(这种指针所指向的 `char` 是“不可修改的”(immutable))，所以我们只需根据 `filename` 参数直接创建 `ctypes` 字符串缓冲区，并将其传给 `_load()` 即可。假如 `_load()` 函数返回 `None`，那么就说明加载操作失败了，我们把错误信息放在异常里面，并将其抛出。反之，则把指针储存起来以备后用。

不论有没有执行断字字典加载操作，我们都要试着获取与之相应的指针，并将其返回给调用者。

```
def _cleanup():
    for hyphens in _hdicForFilename.values():
        _unload(hyphens)

atexit.register(_cleanup)
```

已经加载好的每个断字字典都会有指针指向它，而这些指针都作为键值对中的值存放在名叫 `_hdicForFilename` 的 dict 里面。程序终止的时候，必须把这些字典全部释放掉。我们创建 `_cleanup()` 这个私有函数，令它遍历 dict 中每一个指向断字字典的指针，并通过私有的 `_unload()` 函数来释放这些指针，而这个 `_unload()` 本身又会调用 `hyphenation` 库中的 `hnj_hyphen_free()` 函数。我们无须手工清除 `_hdicForFilename` 的 dict，因为 `_cleanup()` 函数只会在程序终止的时候执行，而那时系统会自动把 dict 删掉。我们通过标准库的 `atexit` 模块的 `register()` 函数将 `_cleanup()` 注册成“at exit”函数，以确保其能在程序终止时执行。

通过前面所讲的这些代码，我们就可以调用 `hyphen` 库中的相关函数，并且在自编的模块中实现出 `hyphenate()` 函数了。使用 `ctypes` 时确实要注意一些问题，例如要设定参数类型，要初始化参数的值等，不过 `ctypes` 使得我们可以在 Python 程序中调用以 C 或 C++ 语言写成的功能。比方说，我们可能想用 C 或 C++ 编写一些“对速度要求很高的代码”(speed-critical code)，各程序所共享的程序库也需要用这种方式来写，这些代码当然可以直接在自编的 C 或 C++ 程序中调用，然而借助 `ctypes`，我们还可以在 Python 程序中调用它们。`ctypes` 的另一种主要用途是访问第三方共享库中的 C 或 C++ 功能，不过在大多数情况下，我们无须自己来包装这种共享库，因为基本上都可以找到已经包装好的标准库或第三方模块。

`ctypes` 模块里还有好些细节与功能，但限于本书篇幅，我们只能讲这么多了。尽管它的用法比 CFFI 或 Cython 要难，但由于它是 Python 标准库的一部分，所以配置起来似乎更方便些。

## 5.2 Cython 的用法

cython.org 网站上面说 Cython 是一种可为 Python 语言编写“C 扩展”(C extension)的语言，它用起来和 Python 本身一样方便。Cython 有三种用法。第一种用法是包装 C 或 C++ 代码。尽管 `ctypes` 也能实现这一点，但有人觉得用 Cython 更为容易，对熟悉 C 或 C++ 语言的人来说更是如此。第二种用法是把 Python 代码编译成运行速度更快的 C 代码，实际上只要把模块的扩展名从 `.py` 改为 `.pyx` 并编译即可。对于计算密集型任务来说，单凭这种做法就能把速度提升为原来的两倍。第三种做法与第二种类似，但区别在于：刚才那种做法不修改代码，只把文件扩展名改为 `.pyx`，而这种做法还要将代码“Cythonize”，也就是要利用 Cython 所提供的语言扩展来改写代码，使其可以编译为执行效率更高的 C 代码。对于计算密集型任务来说，这种做法可以把速度提升为原来的 100 倍，甚至更高。

### 5.2.1 用 Cython 访问 C 程序库

本节将创建 `Hyphenate2` 模块，其功能与前一节的 `Hyphenate1.py` 模块相同，只是这次要用 Cython 而非 `ctypes` 来做。`ctypes` 版本只有 `Hyphenate1.py` 这一个文件，而 Cython 版本则需要创建目录，并在其中创建四份文件。

首先需要创建 `Hyphenate2/setup.py` 文件。这是个很短小的“基础文件”(infrastructure file)，里面只有一条重要语句，它会把 `hyphenation` 程序库的位置及所要构建的内容告诉 Cython。第二个文件是 `Hyphenate2/__init__.py`。该文件是为了便于用户使用 `Hyphenate2` 模块而设的，里面只有一条语句，用于导出公共的 `Hyphenate2.hyphenate()` 函数及 `Hyphenate2.Error` 异常。第三个文件名叫 `Hyphenate2/chyphenate.pxd`，这个文件也非常小，它会把 `hyphenation` 库及库中需要用到的函数告诉 Cython。第四个文件是 `Hyphenate2/Hyphenate.pyx`，这是个 Cython 模块，我们将用它来实现公共的 `hyphenate()` 函数以及与之配套的私有辅助函数。接下来就逐个讲解这些文件。

```
distutils.core.setup(name="Hyphenate2",
    cmdclass={"build_ext": Cython.Distutils.build_ext},
    ext_modules=[distutils.extension.Extension("Hyphenate",
        ["Hyphenate.pyx"], libraries=["hyphen"])])
```

上面就是 `Hyphenate2/setup.py` 文件的主要内容，`import` 语句没有列在其中。该文件用到了 Python 的 `distutils` 包<sup>⊖</sup>。`name` 是可选的，`cmdclass` 必须采用上述代码中的值。`Extension()` 的第一个参数是个字符串，表示编译好的模块所应具备的名称（比如说，本例编译出来的模块就叫做 `Hyphenate.so`）。其后是一份由 `.pyx` 文件名所组成的列

<sup>⊖</sup> 笔者建议为 Python 安装版本号大于等于 0.6.28 的 `distribute` 包，如果能安装版本号大于等于 0.7 的 `setuptools` 包就更好了（参见 [python-packaging-user-guide.readthedocs.org](http://python-packaging-user-guide.readthedocs.org)）。许多第三方包都必须用新版的包管理工具才能安装，这其中也包括本书要用到的一些包。

表，它指明了待编译的程序代码所在的文件。在这份列表后面，还可以指定另外一份列表，其中的文件名表示外部的 C 或 C++ 程序库。我们在本例中要使用 hyphen 库，所以列表里的名称自然就是 "hyphen" 了。

在包含这些文件的目录（本例指的是 Hyphenate2 目录）里执行下列命令<sup>⊖</sup>，即可构建出 Hyphenate 扩展：

```
$ cd pipeg/Hyphenate2
$ python3 setup.py build_ext --inplace
running build_ext
cythoning Hyphenate.pyx to Hyphenate.c
building 'Hyphenate' extension
creating build
creating build/temp.linux-x86_64-3.3
...
```

如果安装了好几个 Python 解释器，那么应该把上述命令中的“python3”换成你想使用的那个解释器的完整路径。由 Python 3.1 版本的解释器所生成的文件的名字叫做 Hyphenate.so，后续版本的解释器所生成的共享库其文件名随版本而变。比方说，Python 3.3 生成的库文件就叫做 Hyphenate.cpython-33m.so。

```
from Hyphenate2.Hyphenate import hyphenate, Error
```

Hyphenate2/\_\_init\_\_.py 文件只包含上面这行代码。有了这行代码之后，用户只需执行 import Hyphenate2 as Hyphenate，即可调用 Hyphenate.hyphenate() 了。假如没有这行代码，那么用户必须执行 import Hyphenate2.Hyphenate as Hyphenate，才能达成同样效果。

```
cdef extern from "hyphen.h":
    ctypedef struct HyphenDict:
        pass

    HyphenDict *hnj_hyphen_load(char *filename)
    void hnj_hyphen_free(HyphenDict *hdic)
    int hnj_hyphen_hyphenate2(HyphenDict *hdic, char *word,
                               int word_size, char *hyphens, char *hyphenated_word,
                               char ***rep, int **pos, int **cut)
```

上面是 Hyphenate2/chyphenate.pxd 文件的代码。凡是要在 Cython 代码里访问外部的共享 C 或 C++ 程序库，就得创建这种 .pxd 文件。

第一行代码声明了 C 或 C++ 头文件，而该头文件里又声明了我们所要使用的函数及类型。后面的那些代码用于描述程序所用的函数及类型。Cython 提供了“ctypedef struct”这种简便的方式，使我们无须指定 C 或 C++ 结构体的细节，即可将其描述出来。有时我们仅需传递指向某结构体的指针，而不直接访问其中的“字段”(field)，只有在这种情况下才能使用此方式来描述。而这种情况在 Cython 中很常见，我们使用 hyphenation 程序库

---

<sup>⊖</sup> 在某些操作系统中，必须先安装 libhyphen-dev，然后才能执行下面的命令。——译者注

的时候也是如此。接下来那三个函数定义是直接从 C 或 C++ 头文件里复制出来的，但我们都把每条声明语句末尾表示语句结束的分号都删掉了。

.pxd 文件好比一座用 C 代码搭建起来的桥梁，能够把我们编译好的 Cython 程序与 .pxd 文件所引用的那些外部程序库连接起来。

创建好 setup.py、\_\_init\_\_.py 及 chyphenate.pxd 这三个文件之后，该创建最终的 Hyphenate.pyx 文件了。这个文件里面是 Cython 代码，也就是“用 Cython 扩展语法所写成的 Python 代码”（Python with Cython extensions）。我们首先来看 import 语句，然后逐个讲解其中的函数。

```
import atexit
cimport chyphenate
cimport cpython.pycapsule as pycapsule
```

为了保证加载过的断字字典能在程序终止时释放掉，我们需要引入标准库的 atexit 模块。

编写 Cython 代码时，不仅可以用 import 引入普通的 Python 模块，而且可以用 cimport 来引入 Cython 特有的 .pxd 文件（这种文件其实就是把外部的 C 语言程序库包装了一下）。在本例中，我们把 chyphenate.pxd 引入为 chyphenate 模块，这样的话，就可以使用 chyphenate.HyphenDict 类型以及 hyphenation 库里的那三个函数了。

我们需要创建 Python 中的 dict，该 dict 里每个条目的键都是断字字典的文件名，而其值则是指向 chyphenate.HyphenDict 结构体的指针。但问题是，Python 的 dict 不能存放指针（因为指针不是 Python 中的数据类型）。幸好 Cython 所提供的 pycapsule 可以解决此问题。这个 Cython 模块能够把指针封装在 Python 对象里，而封装好的对象自然就能放在任意 collection 里了。我们稍后将要看到，pycapsule 还可以从 Python 对象里提取指针。

```
def hyphenate(str word, str filename, str hyphen="-"):
    cdef chyphenate.HyphenDict *hdic = _get_hdic(filename)
    cdef bytes bword = word.encode("utf-8")
    cdef int word_size = len(bword)
    cdef bytes hyphens = b"\x00" * (word_size + 5)
    cdef bytes hyphenated_word = b"\x00" * (word_size * 2)
    cdef char **rep = NULL
    cdef int *pos = NULL
    cdef int *cut = NULL
    cdef int failed = chyphenate.hnj_hyphen_hyphenate2(hdic, bword,
        word_size, hyphens, hyphenated_word, &rep, &pos, &cut)
    if failed:
        raise Error("hyphenation failed for '{}'".format(word))
    end = hyphenated_word.find(b"\x00")
    return hyphenated_word[:end].decode("utf-8").replace("=", hyphen)
```

上面这个函数的结构与前一节所讲的 ctypes 版本相同。两者最明显的区别是，本函数明确指定了所有参数及相关变量的类型。尽管 Cython 并不强迫开发者必须这么做，但这样

做使得 Cython 能够运用某些代码优化技术来提升程序性能。

`hdict` 是个指向 `HyphenDict` 结构体的指针, `bword` 变量是个 UTF-8 编码格式的 `bytes`, 用来表示待断字的单词。`word_size` 是个 `int`, 该变量很容易就能创建出来。`hyphens` 在本例中没有实际用途, 但为了使 `chypenate.hnj_hyphen_hyphenate2()` 正常运作, 我们还是得创建一块足够大的“缓冲区”(`buffer`, 也就是 C 语言字符块), 用 Cython 创建这种缓冲区是非常简单的, 只需把“空值字节”(null byte) 与缓冲区所占字节数相乘即可。`hyphenated_word` 缓冲区也是用同样的办法创建出来的。

尽管我们不使用 `rep`、`pos` 及 `cut` 参数, 但却必须向其传入适当的值, 否则 `hnj_hyphen_hyphenate2()` 函数就无法正常运行。针对这三个参数, 我们用 C 语言的写法(也就是像“`cdef char **rep`”这种写法)创建三个指针, 这些指针都比实际参数少一级(比方说, `rep` 参数的实际类型是三重指针 `char ***`, 而我们所创建的却是二重指针 `char **`)。在调用 `hnj_hyphen_hyphenate2()` 函数时, 我们用 C 语言的“取地址操作符”(&) 来传递这些指针的地址, 于是, 实际传入的指针又会比我们所创建的多一级。之所以要采取这种迂回的办法, 是因为 `hnj_hyphen_hyphenate2()` 函数要求这三个参数必须是“非空指针”(non-null pointer) 才行。我们不能直接把 C 语言的“空指针”(null pointer, `NULL`) 当成这三个参数本身的价值, 但参数所指向的次一级指针却可以是空指针。在 C 语言中, `NULL` 表示并未指向任何内容的空指针。

设置好所有参数的初始值之后, 我们调用 `hnj_hyphen_hyphenate2()` 函数, 该函数是由 Cython 的 `chypenate` 模块从 `hyphenation` 库里导出的(实际导出工作由 `chypenate.pxd` 文件完成)。如果断字操作失败, 那么就抛出普通的 Python 异常。若是断字成功, 则返回断好的字。为此, 需要在 `hyphenated_word` 缓冲区中寻找首个“空值字节”, 然后把该字节之前的那些字节按 UTF-8 标准解码成 `str`。最后, 以用户指定的断字符号(若未指定, 则默认为“-”)来替换 `hyphenation` 库所插入的“=”号。

```
_hdicForFilename = {}

cdef chypenate.HyphenDict *_get_hdic(
    str filename) except <chypenate.HyphenDict*>NULL:
    cdef bytes bfilename = filename.encode("utf-8")
    cdef chypenate.HyphenDict *hdic = NULL
    if bfilename not in _hdicForFilename:
        hdic = chypenate.hnj_hyphen_load(bfilename)
        if hdic == NULL:
            raise Error("failed to load '{}'".format(filename))
        _hdicForFilename[bfilename] = pycapsule.PyCapsule_New(
            <void*>hdic, NULL, NULL)
    capsule = _hdicForFilename.get(bfilename)
    if not pycapsule.PyCapsule_IsValid(capsule, NULL):
        raise Error("failed to load '{}'".format(filename))
    return <chypenate.HyphenDict*>pycapsule.PyCapsule_GetPointer(capsule,
        NULL)
```

上面这个私有函数的定义以 `cdef` 开头，而不是以 `def` 开头，这说明它是个 Cython 函数，而不是 Python 函数。在 `cdef` 后面，我们指定了函数的返回值类型，本例它是个指向 `chphenate.HyphenDict` 的指针。然后指定函数名和参数，参数应该指明其类型。本例只有一个字符串类型的 `filename` 参数。

由于这个 Cython 函数的返回值是指针，而不是 `object` 这样的 Python 对象，所以通常没办法向调用者报告异常。实际上，这种函数即便发生了异常，也只会打印出一条警告信息而已，打印完之后，系统就把这个异常忽略了。但我们却想令这个函数能够抛出 Python 异常，所以指定了一种特殊的返回值。本例采用指向 `chphenate.HyphenDict` 的空指针来表示异常。

函数首先声明了指向 `chphenate.HyphenDict` 的指针，并将指针值设为 `NULL`（也就是尚未使其指向任何内容），然后在名为 `_hdictForFilename` 的 `dict` 里查找断字字典的文件名。如果找不到，那么就必须用 `hyphen` 库的 `hnj_hyphen_load()` 函数把这个新字典加载进来，该函数是通过 `chphenate` 模块引入的。如果返回的 `chphenate.HyphenDict` 指针不是 `NULL`，那么就说明字典已经加载好了，我们把指针转换成 `void` 指针（也就是可以指向任意数据类型的指针），并新建 `pycapsule.PyCapsule` 对象来保存它。在 Cython 中，“`<类型> 值`” (`<type> value`) 这种写法可以把值从一种类型转换为 Cython 里的另外一种 C 语言类型。比方说，`<int>(x)` 的意思就是把 `x` 值（必须是个数字或 C 语言的 `char`）转换成 C 语言的 `int`。这与 Python 中的 `int(x)` 类似，但是在 Python 中，`x` 应该是 Python 的 `int`、`float` 或包含整数的 `str`（比如 "123"），而且 `int(x)` 返回的是 Python 的 `int`，而非 C 语言的 `int`。

在调用 `pycapsule.PyCapsule_New()` 时，我们可以通过第二个参数（该参数是个 C 语言的 `char *`）给受封装的指针起个名字，第三个参数是指向“析构函数”(destructor function) 的指针。本例用不到这两个参数，所以给它们都传入 `NULL` 就好。我们把封装之后的指针放到 `dict` 里面，与该指针相关联的键是断字字典的文件名。

无论执行 `_get_hdict()` 时有没有加载断字字典，我们都要试着从 `_hdictForFilename` 里面获取 `capsule` 对象，以便提取其中所封装的指针。获取到 `capsule` 之后，还必须把该对象及其名称传递给 `pycapsule.PyCapsule_IsValid()`，以检测其中是否包含有效指针（也就是检测指针是不是“非空指针”）。调用 `PyCapsule_IsValid()` 的时候，我们给表示指针名称的那个参数传入 `NULL`，因为刚才创建 `capsule` 的时候就没有给它起过名字。如果 `capsule` 中的指针有效，那么就用 `pycapsule.PyCapsule_GetPointer()` 函数将其提取出来，提取的时候，还是给表示指针名称的那个参数传入 `NULL`。最后，把提取出来的 `void` 指针转换成 `chphenate.HyphenDict` 指针，并将其返回。

```
def _cleanup():
    cdef chphenate.HyphenDict *hdict = NULL
    for capsule in _hdictForFilename.values():
        if pycapsule.PyCapsule_IsValid(capsule, NULL):
```

```

hdict = (<chyphenate.HyphenDict*>
         pycapsule.PyCapsule_GetPointer(capsule, NULL))
if hdict != NULL:
    chyphenate.hnj_hyphen_free(hdict)
atexit.register(_cleanup)

```

程序终止的时候，经 `atexit.register()` 注册过的那些函数都会执行。由于本例注册了 `_cleanup`，所以系统会于程序终止时调用自编模块中名为 `_cleanup()` 的私有函数。该函数首先声明指向 `chyphenate.HyphenDict` 的指针，并将其设为 `NULL`。然后遍历 `_hdictForFilename` 结构体中每个条目的值，这些值都是 `capsule`，里面含有指向 `chyphenate.HyphenDict` 的无名指针。如果 `capsule` 里面不是空指针，那么就以该指针为参数来调用 `chyphenate.hnj_hyphen_free()` 函数。

在本节中，我们用 Cython 包装了 hyphenation 共享库，其效果和上一节的 `ctypes` 版本非常相似，只是这次需要单独创建目录，而且还需要编写三个短小的“支援文件”(supporting file)。如果仅仅想在 Python 程序里访问 C 或 C++ 程序库，那么使用 `ctypes` 就足够了，只是有些程序员可能觉得 Cython 或 CFFI 用起来会更简单一些。Cython 还有另外一项功能，就是可以把写好的 Cython 代码（也就是用 Cython 扩展语法写成的 Python 代码）编译成执行速度很快的 C 程序。下一小节就来讲这个功能。

### 5.2.2 编写 Cython 模块以进一步提升程序执行速度

绝大多数情况下，Python 代码的执行速度都令人满意，但有的时候，其速度也受制于外部因素（比如网络延迟等），此时无论怎样优化代码，都无法克服这种问题。然而对于计算密集型任务来说，我们可以用“带 Cython 扩展语法的 Python”（Python syntax plus Cython extensions）来编程，使程序能编译成 C 代码，从而提升运行速度。

在着手优化代码之前，应该先分析其性能。大多数程序在运行的时候都把时间花在了一小部分代码上面，所以，优化前必须先把这部分代码找出来，如果优化的不是这部分代码，那么就会徒劳无功。通过“性能分析”(profiling)，我们可以准确找出瓶颈，而找到瓶颈之后，就可以专心优化这部分代码了。另外，通过性能分析，我们还能对比优化之前与优化之后的程序，以评判优化效果。

在 3.12 节的案例分析中，我们曾注意到，`Image` 模块的 `scale()` 方法（该方法采用平滑缩放技术来缩小图片，参见 3.12.1 节）运行得不够快。本节就试着优化这个方法。

```

Scaling using Image.scale()...
 18875915 function calls in 21.587 seconds
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000   21.587   21.587 <string>:1(<module>)
      1   1.441   1.441   21.587   21.587 __init__.py:305(scale)
 786432   7.335   0.000   19.187   0.000 __init__.py:333(__mean)
3145728   6.945   0.000    8.860   0.000 __init__.py:370(argb_for_color)
 786432   1.185   0.000    1.185   0.000 __init__.py:399(color_for_argb)

```

```

1  0.000  0.000  0.000  0.000 __init__.py:461(<lambda>)
1  0.000  0.000  0.002 __init__.py:479(create_array)
1  0.000  0.000  0.000 __init__.py:75(__init__)

```

对 `Image.scale()` 方法进行性能分析之后，即可得出上面这些数据，本书没有列出 Python 内置的函数，因为那些函数我们无法优化。性能分析可以通过标准库的 `cProfile` 模块来做，`benchmark_Scale.py` 程序演示了该模块的用法。缩小一张  $2048 \times 1536$  的彩色照片（共计 3 145 728 像素）耗时超过 21 秒，这显然算不上很快，而且很容易就能看出最费时间的三个方法：一个是 `_mean()`，另两个是静态的 `argb_for_color()` 与 `color_for_argb()`。

我们现在想比较一下纯 Python 程序与 Cython 程序的真实速度相差多少，所以首先把 `scale()` 方法及其辅助方法（比如 `_mean()` 等）复制到 `Scale/Slow.py` 模块中，并将其改为函数。对其进行性能分析，可得出如下结果。

```

Scaling using Scale.scale_slow()...
9438727 function calls in 14.397 seconds
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    1    0.000    0.000   14.396   14.396 <string>:1(<module>)
    1    1.358    1.358   14.396   14.396 Slow.py:18(scale)
786432    6.573    0.000   12.109    0.000 Slow.py:46(_mean)
3145728    3.071    0.000    3.071    0.000 Slow.py:69(argb_for_color)
786432    0.671    0.000    0.671    0.000 Slow.py:77(color_for_argb)

```

与面向对象相关的那些开销这次都没有了，于是 `scale()` 调用函数的次数大约变为原来的一半（原来是 1800 多万次，现在是 900 多万次），但速度仅仅提升到原来的 1.5 倍。虽说如此，但我们毕竟把与性能有关的那些函数都隔离出来了，现在可以专门针对它们来优化，看看优化之后的 Cython 程序能比现在快多少。

我们把 Cython 代码放到 `Scale/Fast.pyx` 模块中，然后用 `cProfile` 来分析新版程序处理照片的性能，并将其与前两版相对比。

```

Scaling using Scale.scale_fast()...
4 function calls in 0.114 seconds
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    1    0.000    0.000    0.114    0.114 <string>:1(<module>)
    1    0.113    0.113    0.113    0.113 Scale.Fast.scale

```

`cProfile` 模块不能分析 `Scale.Fast.scale()` 方法的性能，因为该方法不是 Python 方法，它已经编译成 C 了。不过没关系，优化之后的结果是很好的，速度变为第一版的 189 倍。只用一张图片测试显然不具代表性，然而在测试过大量图片之后，笔者发现，优化后的 `scale()` 方法至少能把速度提升为原来的 130 倍。

如此可观的提速效果是通过多种优化技术达成的，其中有些只适用于 `scale()` 函数及其辅助函数，而另外一些则是通用的。Cython 版的 `scale()` 函数之所以速度快，主要归功于下面这几项优化：

- 把原有的 Python 文件（例如 `Slow.py`）复制为 Cython 文件（例如 `Fast.pyx`），速度就可变为原来的 2 倍。
- 把全部的私有 Python 函数都改写成 Cython 式的 C 函数，又能把速度变为改写前的 3 倍。
- 使用 C 语言 `libc` 程序库中的 `round()` 函数来取代 Python 内置的 `round()` 函数，能把速度再度提升为改写前的 4 倍<sup>⊖</sup>。
- 传递“内存视图”（memory view）而非数组本身，能把速度再度提升为原来的 3 倍。

除了上面这四项主要的优化技术之外，还有一些小的优化技术也能稍稍提升速度。比方说：指明所有变量的具体类型；传递 `struct`（结构体）而非 Python 对象；将小的函数“内联”（inline），等等，另外可以再运用一些常见的优化手法，比如预先计算偏移量。

由上面的数据可知，Cython 的提速效果非常好，现在我们来看看 `Fast.pyx` 模块中的高效代码是怎么写出来的，尤其要看看“Cython 化之后的 `scale()` 函数”（Cythonized versions of the `scale()` function）及该函数的三个辅助函数：`_mean()`、`_argb_for_color()` 与 `_color_for_argb()`。

优化之前的 `Image.scale()` 方法已经在 3.12.1 节里讲过了，我们先将此方法改写成 `Slow.py` 模块里的 `Scale.Slow.scale()` 函数，然后再把该函数改编为 Cython 版本。`_mean()` 函数与 `_argb_for_color()` 函数也是如此，二者所对应的原方法也都在 3.12.1 节里讲过。改编后的函数与原方法相比，在代码上基本相同，区别只在于原方法是通过 `self` 来访问像素数据的，并且会调用其他方法，而改编后的函数则会把像素数据当成参数来传递，并且会调用其他函数。

`Scale/Fast.pyx` 文件会通过 `import` 语句引入它所需要功能，我们先来看看这部分代码。

```
from libc.math cimport round
import numpy
cimport numpy
cimport cython
```

首先引入 C 语言 `libc` 库里的 `round()` 函数，并用它取代 Python 内置的 `round()` 函数。假如这两个函数在程序里都要用到，那么可以把 `from libc.math cimport round` 改成 `cimport libc.math`，这样就可以通过 `libc.math.round()` 来使用 C 语言版的 `round()` 函数了，而 Python 版的函数则依然叫做 `round()`。接下来两行代码分别引入了 NumPy 模块及 `numpy.pxd` 模块，后者使得 Cython 可以在 C 语言级别访问 NumPy。之所以要用 NumPy 来编写 Cython 版的 `scale()` 函数，是因为它能够非常迅速地处理数组。最后，引入 Cython 的 `cython.pxd` 模块，以便使用其中的某些“修饰器”（decorator）。

---

<sup>⊖</sup> 这两个函数未必总能互换，因为它们的行为还是有所区别的。但对于 `scale()` 与 `_mean()` 函数来说，二者是等效的。

```

_dtype = numpy.uint32
ctypedef numpy.uint32_t _dtype_t

cdef struct Argb:
    int alpha
    int red
    int green
    int blue

DEF MAX_COMPONENT = 0xFF

```

前两行代码创建了两种类型，一种是 Python 类型，名叫 `_dtype`，另一种是 C 语言类型，名叫 `_dtype_t`，它们都是 NumPy 模块中“`uint32` 类型”（`unsigned 32-bit integer`, 32 位无符号整数）的别名。然后创建名为 `Argb` 的 C 语言结构体，其中有四个整数型字段，分别叫做 `alpha`、`red`、`green`、`blue`。假如想创建与之等效的 Python 数据结构，则可用 `Argb = collections.namedtuple("Argb", "alpha red green blue")` 语句来实现。我们还用 Cython 的 `DEF` 语句创建了 C 语言常量。

```

@cython.boundscheck(False)
def scale(_dtype_t[:] pixels, int width, int height, double ratio):
    assert 0 < ratio < 1
    cdef int rows = <int>round(height * ratio)
    cdef int columns = <int>round(width * ratio)
    cdef _dtype_t[:] newPixels = numpy.zeros(rows * columns, dtype=_dtype)
    cdef double yStep = height / rows
    cdef double xStep = width / columns
    cdef int index = 0
    cdef int row, column, y0, y1, x0, x1
    for row in range(rows):
        y0 = <int>round(row * yStep)
        y1 = <int>round(y0 + yStep)
        for column in range(columns):
            x0 = <int>round(column * xStep)
            x1 = <int>round(x0 + xStep)
            newPixels[index] = _mean(pixels, width, height, x0, y0, x1, y1)
            index += 1
    return columns, newPixels

```

`scale()` 函数所用的算法与 `Image.scale()` 相同，但它的首个参数是一维数组，其中保存着像素数据，其后两个参数是原图像的宽度与高度，最后一个参数是缩小比例。我们把“边界检查”（bound checking）功能关闭了，不过这样做在本例中并不能提升程序效率。传给 `scale()` 函数的像素数组是个内存视图，这比传递 `numpy.ndarray` 更高效，而且不会有 Python 级别的开销。我们当然也可以在函数里运用图形编程所特有的一些优化手段，比如说把内存向特定的字节边界对齐。然而本例主要是为了演示如何用 Cython 优化程序，所以不打算专门讲解与图形编程有关的内容。

前面说过，`<type>` 可以把一种类型转换为另一种 Cython 类型，而本函数就多次用到了这种转换操作。函数所创建的那些变量都和 `Image.scale()` 方法一样，但是我们要指明

变量所属的 C 语言数据类型（比如整数变量要声明为 int，浮点数变量要声明为 double）。Python 语言里原有的那些写法依然可以照常使用，比如 for...in 循环。

```

@cython.cdivision(True)
@cython.boundscheck(False)
cdef _DTYPE_t _mean(_DTYPE_t[:] pixels, int width, int height, int x0,
                     int y0, int x1, int y1):
    cdef int alphaTotal = 0
    cdef int redTotal = 0
    cdef int greenTotal = 0
    cdef int blueTotal = 0
    cdef int count = 0
    cdef int y, x, offset
    cdef Argb argb
    for y in range(y0, y1):
        if y >= height:
            break
        offset = y * width
        for x in range(x0, x1):
            if x >= width:
                break
            argb = _Argb_for_color(pixels[offset + x])
            alphaTotal += argb.alpha
            redTotal += argb.red
            greenTotal += argb.green
            blueTotal += argb.blue
            count += 1
    cdef int a = <int>round(alphaTotal / count)
    cdef int r = <int>round(redTotal / count)
    cdef int g = <int>round(greenTotal / count)
    cdef int b = <int>round(blueTotal / count)
    return _color_for_argb(a, r, g, b)

```

在缩小之后的图片里，每个像素都对应于原图中的若干像素，所以我们要在 `_mean()` 函数里算出那些像素各颜色分量的平均值，并将其赋给新像素的对应颜色分量。原图的像素是作为内存视图传给 `_mean()` 函数的，这要比传递数组更高效。此参数后面那两个参数分别表示原图的宽度及高度。最后还有四个参数，它们用来界定原图中的某块矩形区域，而 `_mean()` 函数正是要对该区域内的像素求颜色分量均值。

本来可以通过  $(y \times \text{width}) + x$  算出每个像素在 `pixels` 中的位置，但为了提高效率，我们在处理每一行像素之前，都预先把  $y \times \text{width}$  的值算出来，并放在 `offset` 变量里。

另外，我们通过 `@cython.cdivision` 修饰器告诉 Cython：函数里的“/”操作符是 C 语言的除法，而不是 Python 语言的除法。这样做能够令函数稍微执行得快一点。

```

cdef inline Argb _Argb_for_color(_DTYPE_t color):
    return Argb((color >> 24) & MAX_COMPONENT,
                (color >> 16) & MAX_COMPONENT, (color >> 8) & MAX_COMPONENT,
                (color & MAX_COMPONENT))

```

上面这个函数是内联函数，这种函数并没有调用开销，因为函数体中的代码会直接在调用它的地方展开（具体到本例来说，就是函数体中的代码会直接出现在 `_mean()` 里面）。我们想通过内联来进一步提升程序速度。

```
cdef inline _DTYPE_t _color_for_argb(int a, int r, int g, int b):
    return (((a & MAX_COMPONENT) << 24) | ((r & MAX_COMPONENT) << 16) |
            ((g & MAX_COMPONENT) << 8) | (b & MAX_COMPONENT))
```

在缩小后的图片中，每个像素的颜色值都要通过上面这个函数来计算，所以我们将其设为内联函数，以提升效率。

Cython 的 `inline` 指令只是个请求，不是所有的 `inline` 函数都会内联，只有当函数像本例中的 `_argb_for_color()` 与 `_color_for_argb()` 一样简短时，`inline` 请求才会生效。而且要注意，尽管本例可以通过函数内联来提升效率，但有时候，内联反而会降低效率。比方说，如果内联后的代码占用了过多的处理器缓存，那么就会发生这种情况。无论执行何种优化，我们都应该在开发所用的电脑或将要部署的电脑中对比一下优化之前和优化之后的运行效率，据此决定是否选用优化后的版本。

Cython 的功能非常多，本例只用了小小一部分，此外，它的开发文档也非常详尽。Cython 的主要缺点是：无论在哪个平台上构建 Cython 模块，开发者都需要使用一种特定的编译器，而且还要为编译器配置一条“工具链”（tool chain）。不过，只要把相关工具都配置好，我们就能通过 Cython 来优化“计算密集型”代码，从而极大地提升其执行速度。

### 5.3 案例研究：用 Cython 优化图像处理程序包

在第3章的案例研究中，我们曾用纯 Python 编写了 `Image` 模块（参见 3.12 节）。本节将要用 Cython 来编写 `cyImage` 模块，该模块实现了 `Image` 模块的绝大部分功能，而且执行速度比 `Image` 快很多。

表 5.1 用 Cython 优化图像处理程序包所取得的提速效果

| 程 序                           | 并 发       | 是否使用 Cython | 运行时间 (秒) | 速度倍数  |
|-------------------------------|-----------|-------------|----------|-------|
| <code>imagescale-s.py</code>  | 无         | 否           | 780      | 基准    |
| <code>imagescale-cy.py</code> | 无         | 是           | 88       | 8.86  |
| <code>imagescale-m.py</code>  | 进程池中的四个进程 | 否           | 206      | 3.79  |
| <code>imagescale.py</code>    | 进程池中的四个进程 | 是           | 23       | 33.91 |

`Image` 与 `cyImage` 有两个主要区别，第一，前者会自动发现并引入与图片格式有关的全部模块，而对于后者来说，此类模块都是预先写好的，其数量也是固定的；第二，`cyImage` 必须使用 NumPy，否则就无法运作，而 `Image` 则会首先考虑使用 NumPy，如果系统里没有安装 NumPy，那么再切换成备用的 `array`。

笔者分别用 Cython 版的 cyImage 模块与 Python 版的 Image 模块编写了四个图像缩小程序，并将其性能测试结果整理成表 5.1。上一节我们曾经讲过，Cython 能把 scale() 函数的速度提升为优化前的 130 倍，但是整个程序在每个核心中的速度为什么只能提升到原来的 8 倍呢？根本原因在于：用 Cython 优化图像缩放操作之后，该操作本身几乎就不再耗时了，但是“载入源图像”及“保存目标图像”这两个操作却还是要执行的。而对于这种文件处理方面的操作来说，Cython 并不能提升多少速度，因为从 Python 3.1 版本开始，文件处理操作都是用 C 语言来实现的，其执行速度已经很快了。于是，从性能角度看，我们只不过是消除了程序在缩放图片时所遇到的瓶颈而已，但程序目前的瓶颈却在于加载及保存图片，而对于这两项操作，我们没有什么好的办法。

想创建 cyImage 模块，首先要创建 cyImage 目录，并且把 Image 目录下面的模块全都复制过去。第二步是把需要“Cython 化”（Cythonize）的模块改名，在本例中，`__init__.py` 要改为 `Image.pyx`，`Xbm.py` 要改为 `Xbm.pyx`，`Xpm.py` 要改为 `Xpm.pyx`。另外，还需要新建 `__init__.py` 及 `setup.py` 文件。

笔者通过实验发现，如果只是把 `Image.Image.scale()` 方法的代码换成 `Scale.Fast.scale()` 函数中的代码，并把 `Image.Image._mean()` 方法中的代码换成 `Scale.Fast._mean()` 函数中的代码，那么提速效果将很不理想。问题似乎在于，Cython 对方法的提速效果不如对函数的提速效果高。鉴于此，我们把 `Scale.Fast.pyx` 模块复制到 cyImage 目录，并将其更名为 `_Scale.pyx`。然后删去 `Image.Image._mean()` 方法，并修改 `Image.Image.scale()` 方法的代码，令其把全部工作都交由 `_Scale.scale()` 函数来做。这样就能把该方法的速度提升到原来的 130 倍了，不过正如前面所说，整个程序的提速效果还是达不到这么多。

```
try:
    import cyImage as Image
except ImportError:
    import Image
```

虽说 cyImage 并不能完全取代 Image（它不支持 PNG 格式，而且它还必须依赖 NumPy 才能运作），但有时我们只需使用 cyImage 所提供的这些功能就够了，在这种情况下，可以采用上面这种写法来引入该模块。

```
distutils.core.setup(name="cyImage",
                      include_dirs=[numpy.get_include()],
                      ext_modules=Cython.Build.cythonize("*.pyx"))
```

上面这几行就是 cyImage/setup.py 文件的主要代码，文件里还有一些 import 语句没有列出来。这几行代码会把 NumPy 头文件的位置告诉 Cython，并令其构建 cyImage 目录下的所有 .pyx 文件。

```
from cyImage.cyImage.Image import (Error, Image, argb_for_color,
                                    rgb_for_color, color_for_argb, color_for_rgb, color_for_name)
```

在编写 `Image` 模块时，我们曾把所有的通用功能都放在了 `Image/_init_.py` 文件里，但现在编写 `cyImage` 模块时，我们却要把这些功能放在 `cyImage/Image.pyx` 文件里，并另外创建一份 `cyImage/_init_.py` 文件，该文件只有上面这一行代码。这个文件的功能就是引入编译好的那些对象（包括一个异常类、一个普通类以及好几个函数），使用户可以直接通过 `cyImage.Image.from_file()`、`cyImage.color_for_name()` 等方式来使用它们。而且开发者还可以像前面那段代码一样，在 `import` 语句里使用 `as` 子句，这样的话，就可以用 `Image.Image.from_file()` 及 `Image.Image()` 这样的写法来指代 `cyImage` 中的相关事物了。

本节不打算重述 `.pyx` 文件，因为上一小节在讲解如何把 Python 代码转换成 Cython 代码的时候已经说过其中的内容了。然而我们要研究一下 `cyImage/Image.pyx` 文件里的那些 `import` 语句，以及新编的 `cyImage.Image.scale()` 方法。

```
import sys
from libc.math cimport round
from libc.stdlib cimport abs
import numpy
cimport numpy
cimport cython
import cyImage.cyImage.Xbm as Xbm
import cyImage.cyImage.Xpm as Xpm
import cyImage.cyImage._Scale as Scale
from cyImage.Globals import *
```

该文件要使用 C 语言的 `round()` 及 `abs()` 函数，而不使用 Python 中的对应版本。另外，我们不打算像 `Image` 那样动态地找寻并引入相关模块，而是直接引入与各种图像格式相对应的那些模块（也就是直接引入 `cyImage/Xbm.pyx` 及 `cyImage/Xpm.pyx`，或者说得更准确一些，Cython 会把 `.pyx` 编译成 C 语言共享库，而我们引入的实际上就是这些共享库）。

```
def scale(self, double ratio):
    assert 0 < ratio < 1
    cdef int columns
    cdef _DTYPE_t[:, :] pixels
    columns, pixels = Scale.scale(self.pixels, self.width, self.height,
                                   ratio)
    return self.from_data(columns, pixels)
```

上面就是 `cyImage.Image.scale()` 方法的全部代码了。由于它把所有工作都交给 `cyImage._Scale.scale()` 函数来完成，所以该方法本身的代码很少。`cyImage._Scale.scale()` 函数的代码与上一小节所讲的 `Scale.Fast.scale()` 函数相同（参见 5.2.2 节）。

Cython 用起来不如纯 Python 那么方便，我们应该先分析 Python 代码的性能，找出其瓶颈，然后再来判断值不值得将其改写为 Cython。假如瓶颈是由文件 I/O 或网路延迟等因素造

成的，那么即便用了 Cython，可能也帮不上多大忙，此时应该考虑使用并发。反之，如果瓶颈确实位于计算密集型的代码中，那么恐怕就应该用 Cython 来优化了，此时可以考虑安装 Cython 并配置相关的“编译器工具链”(compiler tool chain)。

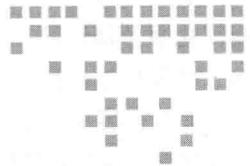
通过性能分析找出待优化的代码之后，最好将这部分低效代码单独提取到某个模块中，然后再次分析程序性能，以确保我们隔离出来的这部分代码正是瓶颈所在。接下来，应该把需要“Cython 化”(Cythonize) 的那些模块复制一份，并将其扩展名由 .py 改为 .pyx，同时创建适当的 setup.py 文件。另外，为了便于开发者使用，我们也可以再创建一份 \_\_init\_\_.py 文件。这时候又需要再次分析程序性能了，我们要看看 Cython 能不能把代码速度提升到原来的两倍。如果可以，那么就反复运用各种 Cython 优化技术来改写代码，比如：明确指定程序里各种数据的类型、使用内存视图来取代数组、把低效方法中代码的全部工作都交给“Cython 化的”(Cythonized) 函数来做。每运用完一项优化技术，就要做一次性能分析。根据性能分析的结果，我们可以把无效的优化代码删掉，把有效的优化代码保留下。然后继续尝试下一项优化，直到所有优化技术都尝试完毕，或程序性能已符合我们要求为止。

高德纳 (Donald Knuth, 唐纳德·克努斯) 在“Structured Programming with go to Statements”一文<sup>⊖</sup>中说过：“别在效率上面斤斤计较。贸然优化就是万恶之源，100 次里有 97 次都是这样。”(We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil, 引自期刊《ACM Computing Surveys》(ACM 计算机概观) 第 6 卷, 第 4 期, 第 268 页, 1974 年 12 月)。此外，算法要是错了，优化得再好也无济于事。但假如算法无误，并且可以通过性能分析查出瓶颈的话，那么用 ctypes 和 Cython 来优化计算密集型程序其效果还是很好的。

如果程序库采用“C 语言调用约定”(C calling convention)，那么借助 ctypes 或 Cython，我们就可以在 Python 程序里通过高级代码来使用程序库中的功能，这些功能是用底层代码写的，运行起来非常快。此外，我们也可以自己编写 C 或 C++ 代码，然后通过 ctypes、Cython 或者直接通过 Python/C API 来调用这些代码。对于计算密集型程序来说，使用并发技术所取得的提速效果在最好的情况下可以与 CPU 核心数成正比。而把纯 Python 代码编译成 C，然后再执行，效率则会非常高，有时可达原来的 100 倍。Cython 使开发者能同时享受到 Python 语言和 C 语言的优点：我们既能用 Python 写出便捷而优雅的代码，又能直接调用 C 程序库里的功能，令自己所编的程序运行起来和 C 程序一样快。

---

<sup>⊖</sup> 文章标题的大意为“使用 go to 语句的结构化编程”，原文参见：<http://cs.sjsu.edu/~mak/CS185CKnuthStructuredProgrammingGoTo.pdf>。——译者注



## 第6章

*Chapter 6*

# Python 高级网络编程

Python 标准库对网络编程的支持非常好，从底层到高层都是如此。底层有 `socket`、`ssl`、`asyncore` 及 `asynchat` 这样的模块可用，中层则有 `socketserver` 等模块支持，而高层的模块就更多了，其中最重要的是 `http` 及 `urllib`。高层的模块所支持的 Internet 协议也有很多种。

支持网络编程的第三方模块数量不少，其中包括 Pyro4（Python remote objects，Python 远程对象，[packages.python.org/Pyro4](http://packages.python.org/Pyro4)）、PyZMQ（Python bindings for the C-based 0MQ library，对基于 C 的 0MQ 库所做的 Python 绑定，[zeromq.github.com/pyzmq](https://zeromq.github.com/pyzmq)）及 Twisted（[twistedmatrix.com](http://twistedmatrix.com)）。如果你只想针对 HTTP 及 HTTPS 协议来开发，那么可以考虑 `requests`（[python-requests.org](http://python-requests.org)），这是个简单易用的第三方程序包。

本章讲解两个高级的网络编程模块，一个是标准库里的 `xmlrpc` 模块（XML Remote Procedure Call，XML 远程过程调用），另一个是第三方的 RPyC 模块（Remote Python Call，远程 Python 调用，[rpyc.sourceforge.net](http://rpyc.sourceforge.net)）。这两个模块功能都很强大，而且用起来也很方便，开发者无须关注底层及中层细节。

本章要演示两个客户端与一个服务器程序，而且要用 `xmlrpc` 及 RPyC 模块分别实现这套程序。在这两个版本的程序中，服务器与客户端的任务基本相同，所以便于我们对比其效果。服务器用来管理某种“仪表的读数”（meter readings，比如电表、水表、煤气表等），而“抄表员”（meter reader）则可以用客户端来领取抄表任务，并汇报自己抄到的数据，如果没能抄到数据，那么需要提交未能抄到数据的原因。

这两套程序最重要的区别是：用 `xmlrpc` 所编的服务器程序不采用并发技术，而用 RPyC 制作的服务器则是并发的。稍后大家就会看到，由于实现方式不同，所以服务器管理

数据的方式也会有很大差别。

为了尽可能简化服务器程序，我们把仪表读数的管理工作单独提取成模块（非并发版本叫做 `Meter.py`，并发版本叫做 `MeterMT.py`）。这样划分还有个好处：假如要管理的数据同本例差别很大，那么很容易就能用自编的模块把本例的读数管理模块替换掉，我们只需稍加改编，就能令客户端与服务器程序适用于其他场合。

## 6.1 编写 XML-RPC 应用程序

如果要用底层协议实现网络通信，那么就得亲自管理需要传递的每份数据。我们必须把数据“封包”（*package up*）并将其发送给接收方，接收方收到之后要“解包”（*unpack*），并对其中数据执行某种操作，以便响应发送方。假如反复用代码来实现这个过程，那么就显得十分枯燥，而且容易出错。一种办法是采用 RPC 程序库（RPC 是 *remote procedure call*（远程过程调用）的缩写）。我们只需把函数名称及参数（比如字符串、数字、日期等）发给这个程序库就行了，而“封包”、“发送”、“解包”、“执行操作”（也就是调用相关函数）等事项都由 RPC 库来做。XML-RPC 是一种比较流行的标准 RPC 协议，实现该协议的程序库会把数据（也就是函数名称及参数）编码成 XML 格式，并且使用 HTTP 机制来传输。

在 Python 标准库中，`xmlrpclib.server` 与 `xmlrpclib.client` 模块用于支持该协议。协议本身与编程语言无关，所以，用 Python 语言写成的 XML-RPC 服务器照样可以与用其他编程语言所写的客户端相通信，只要那些语言支持此协议即可。反过来也一样，用 Python 语言写成的 XML-RPC 客户端也能连接到用其他语言写成的 XML-RPC 服务器上面。

`xmlrpclib` 模块还提供了某些与 Python 相关的扩展功能。比方说，我们可以在网络中传递 Python 对象，但这要求客户端与服务器必须都是用 Python 开发的。本节的范例程序不打算运用这项特性。

除了 XML-RPC 之外，还有一种轻量级的替代方案，叫做 JSON-RPC。它的功能与 XML-RPC 同样多，但采用的数据格式却非常精简，这种格式所占字节数不多，在网络中传输时所需的额外开销远远低于 XML 格式。Python 标准库的 `json` 模块可以把 Python 数据编码成 JSON 格式，也可以从 JSON 中解码 Python 数据，但是 Python 并未提供用于开发 JSON-RPC 客户端或服务器的模块。不过，有很多第三方 Python 模块支持 JSON-RPC（参见 [en.wikipedia.org/wiki/JSON-RPC](http://en.wikipedia.org/wiki/JSON-RPC)）。如果客户端与服务器都打算用 Python 开发，那么可以考虑 RPyC，我们将在下一节（6.2 节）讲解它。

### 6.1.1 数据包装器

客户端与服务器所要处理的数据封装在 `Meter.py` 模块中。该模块提供了 `Manager` 类，用于存放仪表读数，并且提供了相关的方法，抄表员可通过这些方法来登录抄表系统、

领取抄表任务、提交抄表结果。如果要管理的数据与本例完全不同，那么你可以用自编的模块来替换该模块。

```
class Manager:
    SessionId = 0
    UsernameForSessionId = {}
    ReadingForMeter = {}
```

成功登录的用户都会获得“唯一的会话标识符”（unique session ID），而上述代码中的 SessionId 则用来提供此标识符。

这个类里还有两个静态字典，其中一个以会话 ID 为键，以用户名为值，另一个以仪表编号为键，以仪表读数为值。

这些静态数据都无须做成“线程安全的”，因为 xmlrpc 服务器不是并发式服务器。MeterMT.py 版本的模块支持并发，我们将在 6.2.1 节里讲解它与 Meter.py 之间的区别。

在更真实的应用场景中，数据可能会存放在 DBM 文件或数据库里，而不会存放在字典里，但过我们很容易就能用符合实际需求的数据存储形式来替换本例所用的字典。

```
def login(self, username, password):
    name = name_for_credentials(username, password)
    if name is None:
        raise Error("Invalid username or password")
    Manager.SessionId += 1
    sessionId = Manager.SessionId
    Manager.UsernameForSessionId[sessionId] = username
    return sessionId, name
```

抄表员在领取抄表任务或提交抄表结果之前，必须先以用户名和密码登录。

如果用户名和密码正确，那么就返回唯一的 session ID 以及用户的真实姓名（这个名字可以显示在用户界面中）。每次登录成功后，程序都会赋予用户唯一的 session ID，并把它添加到 UsernameForSessionId 字典里。调用其他方法时，都必须传入有效的 session ID。

```
_User = collections.namedtuple("User", "username sha256")
def name_for_credentials(username, password):
    sha = hashlib.sha256()
    sha.update(password.encode("utf-8"))
    user = _User(username, sha.hexdigest())
    return _Users.get(user)
```

上面这个函数在执行的时候会根据传入的 password 参数算出“SHA-256 哈希码”（SHA-256 hash），如果 username 与哈希码都和模块私有的 \_Users 字典（该字典的内容没有列出来）中的某个条目相符，那么就返回与 username 相对应的用户真名，否则返回 None。

在 \_Users 字典中，每个条目的键都是个 \_User 元组，其中包括用户登录时使用的 username（例如 carol），还包括根据密码所算出的 SHA-256 哈希码；而每个条目的值则是与 username 相对应的用户真名（例如“Carol Dent”）。这种存储方案无须保存实

际密码<sup>Θ</sup>。

```
def get_job(self, sessionId):
    self._username_for_sessionid(sessionId)
    while True: # Create fake meter
        kind = random.choice("GE")
        meter = "{}{}".format(kind, random.randint(40000,
            99999 if kind == "G" else 999999))
    if meter not in Manager.ReadingForMeter:
        Manager.ReadingForMeter[meter] = None
    return meter
```

抄表员登录之后，可以调用上面这个方法来领取仪表编号，领到编号之后，他们就可以去抄录相关仪表的读数了。该方法首先判断 session ID 是否有效，如果无效，那么 \_username\_for\_sessionid() 方法会抛出 Meter.Error 异常。

由于我们并未用真实的数据库来存放待抄的仪表，所以在抄表员请求领取抄表任务时，get\_job() 方法会虚构一个仪表出来。首先创建仪表编号（例如“E350718”或“G72168”），然后判断 ReadingForMeter 字典里有没有该编号，如果没有，那么就将其添加进去，并把与之关联的读数设为 None。

```
def _username_for_sessionid(self, sessionId):
    try:
        return Manager.UsernameForSessionId[sessionId]
    except KeyError:
        raise Error("Invalid session ID")
```

如果 session ID 有效，那么上面这个方法就会返回与之相关的 username；若 ID 无效，则会把通用的 KeyError 转换成自编的 Meter.Error 异常，并将其抛出。

在与业务有关的代码中，最好能把内置的异常转换为自编的异常，因为这样一来，调用该代码的人就能根据其需要来捕获这些自编的异常了，而无须把通用的异常一并捕获进来。如果程序抛出了通用的异常，那就说明代码有逻辑错误，这种异常留着不捕获反而好一些。

```
def submit_reading(self, sessionId, meter, when, reading, reason=""):
    if isinstance(when, xmlrpc.client.DateTime):
        when = datetime.datetime.strptime(when.value,
            "%Y%m%dT%H:%M:%S")
    if (not isinstance(reading, int) or reading < 0) and not reason:
        raise Error("Invalid reading")
    if meter not in Manager.ReadingForMeter:
        raise Error("Invalid meter ID")
    username = self._username_for_sessionid(sessionId)
    reading = Reading(when, reading, reason, username)
    Manager.ReadingForMeter[meter] = reading
    return True
```

---

<sup>Θ</sup> 本例所用的这种方法仍不安全。要想更安全些，可以给每个密码附加一段“salt”（“盐”，干扰文本）文本，这样的话，相同的密码就不会产生相同的哈希值了。还有个更好的办法，就是使用第三方 passlib 库 ([code.google.com/p/passlib](http://code.google.com/p/passlib))。

上面这个方法的第二个参数是 session ID，其后是仪表号码（例如“G72168”）、抄表日期与时间、所抄录的仪表读数（该参数一般是正整数，若没能抄到读数，则应传入 -1），以及未能抄到读数的原因（如果未能抄到读数，那么应该给该参数传入非空的字符串）。

开发者可以令 XML-RPC 服务器使用 Python 内置的类型，但在默认情况下服务器不会这么做，而且我们在本例中也确实没有这么做。由于 XML-RPC 协议与编程语言无关，因此我们的 XML-RPC 服务器不应该只服务于 Python 客户端，还应该向用其他语言所写的客户端提供服务，只要那些语言也支持 XML-RPC 就行。不使用 Python 类型的缺点是：日期/时间对象无法以 `datetime.datetime` 的形式传输，而要表示为 `xmlrpclib.DateTime`，数据接收方又必须将其再转回 `datetime.datetime`。（还有一种办法是传递 ISO-8601 格式的日期/时间字符串。）

把抄表员提交的数据验证好之后，我们根据本方法的 `sessionId` 参数来获取抄表员的 `username`，并用它来创建 `Meter.Reading` 对象。该对象是个简单的具名元组：

```
Reading = collections.namedtuple("Reading", "when reading reason username")
```

在 `submit_reading()` 方法最后，我们把抄表员提交的仪表读数存放到 `ReadingForMeter` 里，然后返回 `True`（如果不写 `return` 语句，那么就会返回默认的 `None`）。之所以要返回 `True`，是因为 `xmlrpclib.server` 模块在默认情况下不支持 `None`，如果令其支持 `None`，那么客户端就只能用 Python 来写了，而这又与我们的需求不符：我们要令服务器能够服务于用各种语言所编写的客户端。（假如改用 RPyC，那么就可以令本方法返回 Python 语言特有的值了。）

```
def get_status(self, sessionId):
    username = self._username_for_sessionid(sessionId)
    count = total = 0
    for reading in Manager.ReadingForMeter.values():
        if reading is not None:
            total += 1
            if reading.username == username:
                count += 1
    return count, total
```

抄表员提交完读数后，可能还想了解一些相关信息，比方说，自己提交了多少次读数，服务器自启动之后总共处理了多少笔数据。上面这个方法可以统计出这两项数字，并将其返回给调用者。

```
def _dump(file=sys.stdout):
    for meter, reading in sorted(Manager.ReadingForMeter.items()):
        if reading is not None:
            print("{}={}@{}[{}]{}, {}, {}, {}".format(meter, reading.reading,
                reading.when.isoformat()[:16], reading.reason,
                reading.username), file=file)
```

上面这个方法纯粹是为了调试而设的，我们可以用它来检查仪表读数是不是都正确地存

放起来了。

Meter.Manager 提供了用于登录的 login() 方法，还提供了获取及设置数据所用的一些方法，像这样的类通常可以充当“数据包装类”(data-wrapping class)，以供服务器所用。如果要管理的数据与本例完全不同，那么只需把该类替换掉就可以了，而本章所讲的客户端与服务器程序则基本上可以沿用。唯一要注意的就是：假如服务器是并发的，那么必须对共享数据加锁，或使用“线程安全的类”来管理那些数据，6.2.1 节将谈到这个问题。

## 6.1.2 编写 XML-RPC 服务器

有了 xmlrpc.server 模块，我们很容易就能编写出 XML-RPC 服务器。本节的代码选自 meterserver-rpc.py 文件。

```
def main():
    host, port, notify = handle_commandline()
    manager, server = setup(host, port)
    print("Meter server startup at {} on {}:{}".format(
        datetime.datetime.now().isoformat()[:19], host, port, PATH))
    try:
        if notify:
            with open(notify, "wb") as file:
                file.write(b"\n")
        server.serve_forever()
    except KeyboardInterrupt:
        print("\rMeter server shutdown at {}".format(
            datetime.datetime.now().isoformat()[:19]))
        manager._dump()
```

main() 函数获取到主机名 (hostname) 与“端口号”(port number) 之后，创建 Meter.Manager 与 xmlrpc.server.SimpleXMLRPCServer 对象，并启动服务器，开始向客户端提供服务。

如果 notify 变量之中有文件名，那么服务器就会创建该文件，并向其中写入“换行符”(newline)。用户手工启动服务器的时候是不需要指定文件名的，但我们在 6.1.3 节会看到，如果服务器是由 GUI 客户端启动的，那么客户端就会给服务器传递文件名，而服务器程序则会把该名称放入 notify 变量里。GUI 客户端会一直等着服务器把该文件创建好，创建好之后，客户端就知道服务器已经完全启动了，于是它会删掉该文件，并开始与服务器通信。

若想终止服务器，你可以按下 Ctrl+C 组合键，或向其发送 INT 信号（在 Linux 操作系统中，可以通过 kill -2 pid 命令来做），而 Python 解释器则会把这两种行为转换成 KeyboardInterrupt 异常。如果用户是通过上述两种方式来终止服务器的，那么我们就通过 manager.\_dump() 来收集其中的仪表读数信息，以便查验。（正因为要收集数据，所以我们才会用到 manager 实例。）

```
HOST = "localhost"
PORT = 11002
```

```

def handle_commandline():
    parser = argparse.ArgumentParser(conflict_handler="resolve")
    parser.add_argument("-h", "--host", default=HOST,
                        help="hostname [default %(default)s]")
    parser.add_argument("-p", "--port", default=PORT, type=int,
                        help="port number [default %(default)d]")
    parser.add_argument("--notify", help="specify a notification file")
    args = parser.parse_args()
    return args.host, args.port, args.notify

```

笔者专门把上面这个函数列出来讲，是因为我们想令用户通过 -h 选项来设定主机名（另外也可以用 --host 来指定）。在默认情况下，argparse 模块会预留 -h 与 --help 选项，如果用户在输入命令时指定了这两个选项之一，那么程序会显示出自己的命令行用法，然后终止。可是我们此处却要重新定义 -h 选项的含义，同时又要保持 --help 选项的语义不变，所以，必须在构建 ArgumentParser 对象时指定 conflict\_handler 参数，以便覆盖 -h 选项。

不巧的是，argparser 模块迁移到 Python 3 之后，却保留了 Python 2 时代以 “%” 符号来格式化字符串的旧办法，而没有改成 Python 3 里 str.format() 方法所用的那种花括号方式。有鉴于此，帮助文本里面的默认值必须按照 %(default)t 的格式来写，其中 t 是值的类型（d 表示十进制整数，f 表示浮点数，s 表示字符串）。

```

def setup(host, port):
    manager = Meter.Manager()
    server = xmlrpclib.server.SimpleXMLRPCServer((host, port),
                                                requestHandler=RequestHandler, logRequests=False)
    server.register_introspection_functions()
    for method in (manager.login, manager.get_job, manager.submit_reading,
                   manager.get_status):
        server.register_function(method)
    return manager, server

```

上面这个函数用于创建 manager 对象，以便管理数据（比如本例中的仪表数据），另外还会创建服务器对象。调用了 register\_introspection\_functions() 方法之后，客户端就能访问 system.listMethods()、system.methodHelp() 及 system.methodSignature() 这三个“introspection function”（内容探查函数、“内省”函数）了。（本例中的 XML-RPC 客户端不会用到这三个函数，但对于更复杂的客户端来说，也许用得到。）如果想令客户端能够访问 manager 对象的某个方法，那么必须先在服务器上注册它。我们通过 register\_function() 方法来注册这些方法。（注册的时候，方法名前面要加上“manager.”，具体原因请参见 2.5 节所讲的“绑定方法与非绑定方法”。）

```

PATH = "/meter"

class RequestHandler(xmlrpclib.server.SimpleXMLRPCRequestHandler):
    rpc_paths = (PATH,)

```

在本例中，服务器并不需要对客户端发来的请求做特殊处理，所以我们只需创建最简

单的 request handler (请求处理器) 就好：这个 handler 直接继承自 `xmlrpclib.server.SimpleXMLRPCRequestHandler` 类，而且设定了一条特殊的路径 (PATH)，客户端需要通过此路径向服务器发送请求。

创建好服务器之后，我们来创建访问该服务器的客户端。

### 6.1.3 编写 XML-RPC 客户端

本节将创建两种不同的客户端，一种是基于控制台的客户端，它总是假定服务器已经运行起来了；另一种是 GUI 客户端，如果有服务器已运行，那么它就直接与其通信，若尚未有服务器运行，则自行启动之。

#### 1. 基于控制台的 XML-RPC 客户端

开始讲解代码之前，我们先看看用户是如何通过控制台与服务器互动的。在执行下列交互命令之前，必须先启动 `meterserver-rpc.py` 服务器。

```
$ ./meterclient-rpc.py
Username [carol]:
Password:
Welcome, Carol Dent, to Meter RPC
Reading for meter G5248: 5983
Accepted: you have read 1 out of 18 readings
Reading for meter G72168: 2980q
Invalid reading
Reading for meter G72168: 29801
Accepted: you have read 2 out of 21 readings
Reading for meter E445691:
Reason for meter E445691: Couldn't find the meter
Accepted: you have read 3 out of 26 readings
Reading for meter E432365: 87712
Accepted: you have read 4 out of 28 readings
Reading for meter G40447:
Reason for meter G40447:
$
```

假设有位名叫 Carol 的抄表员启动了上述客户端，那么客户端首先会提示她输入用户名，并且在方括号内给出默认的用户名。由于她在抄表服务器里的用户名恰好与操作系统里的用户名一样，都是“carol”，所以只需按下“回车”(Enter) 键即可。然后，客户端会提醒 Carol 输入密码，在她输入密码的过程中，控制台不会把这些密码回显出来。输入的密码经服务器验证无误之后，客户端会给出欢迎消息，并打印出用户的全名。接下来，客户端会向服务器发送请求，以领取抄表任务，并提示 Carol 把她所抄到的读数录入控制台。如果她输入的是数字，那么客户端就接受这项输入，并把此值发给服务器。若是输入有误（比如上面第二项数据“2980q”就输错了），或由于其他原因而输入了无效的读数，那么客户端就会打印出一条信息，并再次提示用户输入读数。只要 Carol 输入的读数（或无法抄得仪表读数的原因）为

服务器所接受，那么控制台就会打印出她在这次会话中提交了多少条有效数据，同时还会打印出服务器在这次会话中总共收集了多少条有效数据（这其中也包括其他抄表员提交至该服务器的数据）。如果 Carol 在输入仪表读数时直接按了“回车”，那么控制台会询问未能抄到读数的原因。如果此时不输入原因而继续按“回车”，那么客户端就终止了。

```
def main():
    host, port = handle_commandline()
    username, password = login()
    if username is not None:
        try:
            manager = xmlrpclib.ServerProxy("http://{}:{}{}".format(
                host, port, PATH))
            sessionId, name = manager.login(username, password)
            print("Welcome, {}, to Meter RPC".format(name))
            interact(manager, sessionId)
        except xmlrpclib.Fault as err:
            print(err)
        except ConnectionError as err:
            print("Error: Is the meter server running? {}".format(err))
```

上面这个 `main()` 函数先获取服务器的主机名及端口号（若用户未指定，则采用默认值），然后获取用户的用户名及密码。接下来，针对服务器所要使用的 `Meter.Manager` 实例来创建 proxy，并将其赋给 `manager` 变量。（2.7 节讲解了“代理模式”（Proxy Pattern）。）

创建好 proxy 之后，我们通过 `manager.login()` 来登录服务器，然后开始与服务器交互。如果服务器尚未运行，则会出现 `ConnectionError` 异常（在 Python 3.3 之前的版本中，是 `socket.error` 异常）。

```
def login():
    loginName = getpass.getuser()
    username = input("Username [{}]: ".format(loginName))
    if not username:
        username = loginName
    password = getpass.getpass()
    if not password:
        return None, None
    return username, password
```

我们通过 `getpass` 模块的 `getuser()` 函数来获取用户在登录操作系统时所用的用户名，并将该名称用作登录抄表服务器时的默认用户名。然后用 `getpass()` 函数来提示用户输入密码，用户所输的密码不会“回显”（echo）到控制台。由 `input()` 与 `getpass.getpass()` 所返回的字符串都不包含末尾的换行符。

```
def interact(manager, sessionId):
    accepted = True
    while True:
        if accepted:
            meter = manager.get_job(sessionId)
            if not meter:
```

```

        print("All jobs done")
        break
    accepted, reading, reason = get_reading(meter)
    if not accepted:
        continue
    if (not reading or reading == -1) and not reason:
        break
    accepted = submit(manager, sessionId, meter, reading, reason)

```

登录成功之后，客户端调用上面这个函数来处理它与服务器之间的通信。此函数会反复从服务器端领取任务（在本例中指的就是抄表任务），然后获取抄表员所输入的读数或未能抄到读数的原因，并将数据提交给服务器，直到用户既不输入读数，又不给出原因为止。

```

def get_reading(meter):
    reading = input("Reading for meter {}: ".format(meter))
    if reading:
        try:
            return True, int(reading), ""
        except ValueError:
            print("Invalid reading")
            return False, 0, ""
    else:
        return True, -1, input("Reason for meter {}: ".format(meter))

```

上面这个函数必须处理三种情况：用户输入了有效读数（也就是输入了整数）；用户输入了无效读数；用户根本没输入读数。用户如果没有输入读数，那么应该给出未能抄到读数的原因，如果不给出原因，那么就意味着用户想要退出客户端。

```

def submit(manager, sessionId, meter, reading, reason):
    try:
        now = datetime.datetime.now()
        manager.submit_reading(sessionId, meter, now, reading, reason)
        count, total = manager.get_status(sessionId)
        print("Accepted: you have read {} out of {} readings".format(
            count, total))
    return True
    except (xmlrpc.client.Fault, ConnectionError) as err:
        print(err)
        return False

```

`get_reading()` 函数如果从抄表员那里获取了读数或未能抄到读数的原因，那么就会调用上面这个函数，该函数会通过 `manager.submit_reading()` 方法把数据提交给服务器（`manager` 是个针对 `Meter.Manager` 的 proxy）。将读数或原因提交上去之后，函数会调用 `manager.get_status()` 来查询服务器的状态，也就是查询当前这位用户已经提交了多少笔有效数据，以及服务器自启动之后总共收到多少笔有效数据。

客户端的代码虽然比服务器的代码长，但是读起来却非常直观。此外，由于我们使用了 XML-RPC 协议，所以客户端也可以改用支持该协议的其他语言来编写。你还可以选用别的技术来制作客户端的用户界面，比方说，用 Urwid ([excess.org/urwid](http://excess.org/urwid)) 这样的程序库来制作基

于 Unix 控制台的用户界面，或用 Tkinter 等 GUI 工具包来制作图形化的用户界面。

## 2. 带图形界面的 XML-RPC 客户端

本书第 7 章将会讲解如何用 Tkinter 编写 GUI 程序，你如果还不熟悉 Tkinter，那么可以先学习第 7 章，然后再回到这里。在这里，我们专门讲解 meter-rpc.pyw 这个 GUI 程序是如何与 meter 服务器通信的。程序界面如图 6.1 所示。

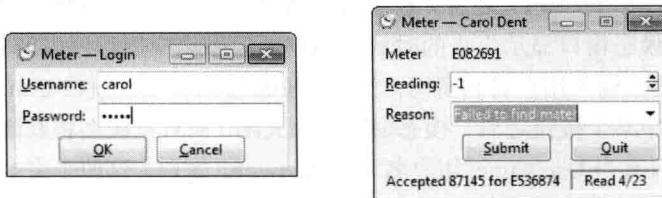


图 6.1 带图形界面的 XML-RPC 客户端，左右两张小图分别是登录窗口与主窗口在 Windows 操作系统中的样貌

```
class Window(ttk.Frame):
    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.serverPid = None
        self.create_variables()
        self.create_ui()
        self.statusText.set("Ready...")
        self.countsText.set("Read 0/0")
        self.master.after(100, self.login)
```

创建主窗口的时候，我们把服务器进程的 ID 设为 `None`（进程的 ID 简称 PID，是 Process ID 的缩写），主窗口如果构建好了，那么程序就会在 100 毫秒后调用 `login()` 方法。之所以要有 100 毫秒的延迟，是为了给 Tkinter 留出绘制主窗口的时间。不过用户现在并不能直接和主窗口交互，因为我们还创建了应用程序级别的“模态窗口”（modal window）供用户登录服务器。如果应用程序里面有模态窗口，那么用户只能通过这种窗口与应用程序交互。具体到本例来说，尽管用户可以看到主窗口，但却不能操作它，而是必须先通过模态窗口登录服务器，登录完成之后，模态窗口会消失，此时用户才可以操作主窗口。

```
class Result:
    def __init__(self):
        self.username = None
        self.password = None
        self.ok = False
```

上面这个类的代码很少（该类位于 `MeterLogin.py` 文件里），它用来存放用户同模态登录对话框窗口之间的交互结果。我们把指向 `Result` 实例的引用传给模态对话框，这样的话，即便对话框关闭了或是从内存中删除了，我们也依然能获取到用户所输入的内容。

```

def login(self):
    result = MeterLogin.Result()
    dialog = MeterLogin.Window(self, result)
    if result.ok and self.connect(result.username, result.password):
        self.get_job()
    else:
        self.close()

```

上述方法首先创建 `result` 对象，然后创建应用程序级别的模态窗口，供用户登录。调用完 `MeterLogin.Window()` 之后，模态窗口就会显示出来，而程序将会阻塞，直到模态窗口关闭为止。在模态窗口显示出来的时候，用户无法操作应用程序内的其他窗口，他们此时只有两种选择：要么输入用户名和密码，并按下 OK 按钮，要么就按 Cancel 按钮退出程序。

用户点击了其中某个按钮之后，模态窗口就会关闭（而且系统会将其删除）。如果用户点击的是 OK 按钮（只有当 `Username`(用户名) 和 `Password`(密码) 这两个文本框中都有内容时，才可以点击 OK 按钮），那么客户端就会尝试连接服务器，并领取第一份抄表任务。如果用户按的是 Cancel 按钮，或是连接服务器的过程出错，那么主窗口就关闭了（同时系统会把它删掉），而应用程序也随之终止。

```

def connect(self, username, password):
    try:
        self.manager = xmlrpclib.ServerProxy("http://{}:{}{}".format(HOST, PORT, PATH))
        name = self.login_to_server(username, password)
        self.master.title("Meter \u2014 {}".format(name))
        return True
    except (ConnectionError, xmlrpclib.Fault) as err:
        self.handle_error(err)
        return False

```

用户输入了用户名和密码之后，如果按下 OK 按钮，那么上述方法就会执行。它首先针对服务器的 `Meter.Manager` 实例创建 proxy，并尝试登录服务器。登录成功后，该方法会修改应用程序的标题：它会在应用程序本身的名称后面加个 em-dash 符号（—，该符号的 Unicode code point (Unicode 编码点) 是 U+2014)，然后再写上用户的全名。修改完标题之后，该方法返回 `True`。

如果连接服务器时出错了，那么就会弹出写有错误文本的消息框，并返回 `False`。

```

def login_to_server(self, username, password):
    try:
        self.sessionId, name = self.manager.login(username, password)
    except ConnectionError:
        self.start_server()
        self.sessionId, name = self.manager.login(username, password)
    return name

```

如果 meter 服务器已经运行起来了，那么上述方法在第一次调用 `self.manager.login()` 时就能登录服务器，并获取到 session ID 与用户的全名。如果第一次登录时抛出了 `ConnectionError` 异常，那么应用程序就会认为该异常是由于服务器尚未启动而

导致的。于是它会启动服务器，并再次尝试登录。若是第二次登录又失败了，那么抛出的 ConnectionError 就会传播到该方法的调用者那里（在本例中指的就是 self.login() 方法），而调用者会捕获此异常，并通过消息框向用户报告错误信息，待用户按下消息框中的 OK 按钮之后，应用程序就会终止。

```
SERVER = os.path.join(os.path.dirname(os.path.realpath(__file__)),
                      "meterserver-rpc.py")
```

上面这个常量保存了服务器程序的完整路径及其文件名。我们在这里假设服务器程序与 GUI 客户端位于同一目录下。虽说客户端与服务器通常会分别运行在两台电脑中，但有些应用程序确实像本例这样分成服务器与客户端两部分编写，然后又放在同一台电脑中运行。

如果想把应用程序的功能同用户界面隔离开，那么就可以像本例这样分成两部分来设计。这样设计的缺点是必须要有两份可执行文件，而合起来设计的话，则只需一份可执行文件，另外，还会有网络通信方面的开销，不过客户端与服务器如果运行在同一台电脑中，那么这部分开销是很小的，用户应该不会觉察。这样设计的优点是，客户端与服务器都可以独立开发，而且很容易就能把这种应用程序移植到新的平台上面。因为服务器是用“与平台无关的代码”（platform-independent code）编写的，所以我们只需把绝大部分精力放在移植客户端上面即可。此外，这种设计方案也使得我们可以运用新的用户界面技术（例如一套新的 GUI 工具包）来移植客户端。也许还有个好处，那就是可以实现更精细的安全管控，比方说，可以设定服务器程序的权限，使系统里只有一部分人有权运行它，而同时使客户端程序以用户权限来运行。

```
def start_server(self):
    filename = os.path.join(tempfile.gettempdir(),
                           "M{}.$${".format(random.randint(1000, 9999)))
    self.serverPid = subprocess.Popen([sys.executable, SERVER,
                                       "--host", HOST, "--port", str(PORT), "--notify",
                                       filename]).pid
    print("Starting the server...")
    self.wait_for_server(filename)
```

上面这个方法通过 subprocess.Popen() 函数启动服务器。本例以这种方式来启动子进程（并用该进程运行服务器程序）是不会一直阻塞到子进程执行完毕的。

假如我们要用子进程去执行普通的程序，并且要等那个程序终止之后再往下执行，那么可以在此处阻塞，一直等着子进程里的程序运行完。但本例不是这种情况，我们要启动的是个服务器程序，该程序只有等客户端终止之后才会终止，所以创建子进程的时候不能一直等着它执行完。此外，还必须给服务器以启动的机会，因为只有服务器启动起来了，我们才能登录进去。笔者所用的办法很简单：创建“伪随机的”（pseudo-random）文件名，并启动服务器，启动的时候，将文件名作为 notify 参数传过去。服务器启动的时候会创建该文件，而客户端则会持续检查该文件系统里是否有该文件，一旦发现有这个文件，那么就表明服务器那边已经准备好了。

```

def wait_for_server(self, filename):
    tries = 100
    while tries:
        if os.path.exists(filename):
            os.remove(filename)
            break
        time.sleep(0.1) # Give the server a chance to start
        tries -= 1
    else:
        self.handle_error("Failed to start the RPC Meter Server")

```

上面这个方法最多会把用户界面阻塞（也就是“冻结”）10秒钟（每0.1秒尝试一次，总共尝试100次），不过实际阻塞时间通常不会超过一秒。客户端如果发现服务器已经把notify文件创建好了，那么就将其删除，并开始照常处理GUI界面中的事件。在本例中，客户端删掉文件之后，会以用户所输入的用户名和密码来登录服务器，登录之后，会显示主窗口，供用户输入仪表读数。假如服务器没能启动，那么while循环中的break就一直没有机会执行，程序在尝试100次之后将转入else子句。

“轮询”（Polling）不是个好办法，尤其在GUI应用程序中更应该少用。但我们此处要编写的是跨平台的应用程序，而该程序又必须在服务器启动之后才能继续执行，所以只得采用这套轮询的办法了，对于本例来说，它是个既简单而又合理的解决方案。

```

def get_job(self):
    try:
        meter = self.manager.get_job(self.sessionId)
        if not meter:
            messagebox.showinfo("Meter \u2014 Finished",
                               "All jobs done", parent=self)
            self.close()
        self.meter.set(meter)
        self.readingSpinbox.focus()
    except (xmlrpclib.Fault, ConnectionError) as err:
        self.handle_error(err)

```

登录到服务器之后，客户端程序会调用上述方法来领取第一份抄表任务（客户端若发现服务器尚未启动，则会先启动它）。self.meter变量的类型是tkinter.StringVar，该变量同显示仪表编号的那个标签相关联。

```

def submit(self, event=None):
    if self.submitButton.instate((tk.DISABLED,)):
        return
    meter = self.meter.get()
    reading = self.reading.get()
    reading = int(reading) if reading else -1
    reason = self.reason.get()
    if reading > -1 or (reading == -1 and reason and reason != "Read"):
        try:
            self.manager.submit_reading(self.sessionId, meter,
                                         datetime.datetime.now(), reading, reason)

```

```

        self.after_submit(meter, reading, reason)
    except (xmlrpc.client.Fault, ConnectionError) as err:
        self.handle_error(err)

```

如果用户输入的仪表读数不是 -1，那么应用程序就会启用 Submit 按钮（如果输入的是 -1，那么只有在用户填写了抄不到读数的原因之后，Submit 按钮才会启用），一旦用户点击该按钮，上述方法就会执行。该方法从用户界面的控件中获取仪表编号、读数（以 int 表示）及原因，然后通过 manager.submit\_reading() 方法将其提交到服务器（manager 是个针对 Meter.Manager 的 proxy）。如果服务器接受了用户提交的读数，那么客户端程序就会调用 after\_submit() 方法；如果服务器不接受，那么客户端程序会把捕获到的错误传给 handle\_error() 方法。

```

def after_submit(self, meter, reading, reason):
    count, total = self.manager.get_status(self.sessionId)
    self.statusText.set("Accepted {} for {}".format(
        reading if reading != -1 else reason, meter))
    self.countsText.set("Read {}/{}".format(count, total))
    self.reading.set(-1)
    self.reason.set("")
    self.get_job()

```

上面这个方法通过 manager.get\_status() 来查询服务器当前的状态，并据此更新用户界面的状态栏及计数标签。该方法也会把输入读数及原因所用的那两个文本框重置，然后从服务器中领取下一项抄表任务。

```

def handle_error(self, err):
    if isinstance(err, xmlrpc.client.Fault):
        err = err.faultString
    messagebox.showinfo("Meter \u2014 Error",
                       "{}\nIs the server still running?\n"
                       "Try Quitting and restarting.".format(err), parent=self)

```

如果客户端程序出错了，那么就会执行上面这个方法。它会将错误信息放在应用程序级别的模态消息框里，并将其显示给用户。该消息框中只有一个 OK 按钮。

```

def close(self, event=None):
    if self.serverPid is not None:
        print("Stopping the server...")
        os.kill(self.serverPid, signal.SIGINT)
        self.serverPid = None
    self.quit()

```

用户关闭应用程序的时候，我们要判断 meter 服务器是由客户端启动的还是本来就已经在运行了。如果是前者，那么客户端程序会向服务器发送中断信号（Python 会将此信号转换为 KeyboardInterrupt 异常），使其安全终止。

os.kill() 函数可以根据调用者所指定的 process ID 向相关程序发送信号，所发送的信号应该是 signal 模块中的常量。在 Python 3.1 中，该函数只适用于 Unix 系统，但从

Python 3.2 开始，也支持 Windows 操作系统了。

`meterclient-rpc.py` 这个控制台界面的客户端只有大约 100 行代码，而带有图形界面的 `meter-rpc.pyw` 客户端则有大约 250 行代码（另有约 100 行代码写在 `MeterLogin.py` 文件里，该文件用于实现登录对话框）。这两种客户端用起来都很简单，而且移植起来也很容易。此外，由于 Tkinter 提供了主题方面的相关支持，所以运行在 OS X 及 Windows 系统里的 GUI 客户端看起来都和该操作系统中的原生应用程序差不多。

## 6.2 编写 RPyC 应用程序

如果服务器与客户端都决定用 Python 来写，那么我们未必要使用 XML-RPC 这样复杂的协议，而是可以直接选一套 Python 专用的协议。有很多程序包都支持 Python 客户端与 Python 服务器之间的“远程过程调用”（Remote Procedure Call，RPC），而本节要使用的则是 RPyC([rpyc.sourceforge.net](http://rpyc.sourceforge.net))。该模块提供了两种模式：一种比较传统，叫做“经典模式”(classic mode)；另一种比较新颖，叫做“基于服务的模式”(service-based mode)。本例采用后者。

默认情况下，RPyC 服务器是并发的，所以我们不能使用上一节那种非并发的“数据包装器”(data wrapper，参见 6.1.1 节的 `Meter.py`)，而是要新编一个名叫 `MeterMT.py` 的模块。该模块包含两个新类，一个是 `ThreadSafeDict`，另一个是 `_MeterDict`，原来的 `Manager` 类还在，但这次不能使用标准的 `dict` 了，而要改用刚才说的那两种字典。

### 6.2.1 线程安全的数据包装器

`MeterMT` 模块中的 `Manager` 类可以“在并发环境下使用”(concurrency-supporting)，而且模块里还有两种“线程安全的”字典。我们首先讲解 `Manager` 类的静态数据，然后看看该类的哪些方法与前一节的 `Meter.Manager` 不同。

```
class Manager:
    SessionId = 0
    SessionIdLock = threading.Lock()
    UsernameForSessionId = ThreadSafeDict()
    ReadingForMeter = _MeterDict()
```

为了支持并发，`MeterMT.Manager` 类必须对其静态数据加锁，以便使多个线程能按顺序访问它们。在这些静态数据中，`SessionId` 可直接加锁，而两个字典中的数据则要用自编的“线程安全字典”(thread-safe dictionary) 来加锁，我们稍后再讲解那两种字典。

```
def login(self, username, password):
    name = name_for_credentials(username, password)
    if name is None:
        raise Error("Invalid username or password")
    with Manager.SessionIdLock:
        Manager.SessionId += 1
```

```

sessionId = Manager.SessionId
Manager.UsernameForSessionId[sessionId] = username
return sessionId, name

```

上面这个方法与早前版本的唯一区别在于：它是在加锁的情况下递增静态变量 `SessionId` 并为其设置新值的<sup>①</sup>。如果不对这一操作加锁，那么就有可能出现下面这种情况：线程 A 已经递增了 `SessionId`，但还没等赋值完成，线程 B 又再次递增了它，此时线程 A 和线程 B 所看到的 `SessionID` 都是递增了两次的值，而不是像我们预期的那样，看到各自不同的值。

```

def get_status(self, sessionId):
    username = self._username_for_sessionid(sessionId)
    return Manager.ReadingForMeter.status(username)

```

`get_status()` 几乎把所有工作都交给自编的 `_MeterDict.status()` 方法来完成，我们稍后再讲这个方法。

```

def get_job(self, sessionId):
    self._username_for_sessionid(sessionId)
    while True: # Create fake meter
        kind = random.choice("GE")
        meter = "{}{}".format(kind, random.randint(40000,
            99999 if kind == "G" else 999999))
        if Manager.ReadingForMeter.insert_if_missing(meter):
            return meter

```

上述方法的最后两行代码和早前版本不同。我们要判断虚构的仪表在不在字典中，如果不在，那么就把它插入字典，并将其初始读数设为 `None`。这样就能保证原先在抄表任务里出现过的仪表不会再度出现在其他抄表任务中。在早前的版本中，我们用两条语句来完成“检测”及“插入”操作，但在并发环境下却不能这么做，因为也许有别的线程会在两条语句之间执行。在本例中，我们把这套操作交给自编的 `_MeterDict.insert_if_missing()` 方法来完成，该方法会告诉我们 `_MeterDict` 里是否发生了插入操作。

```

def submit_reading(self, sessionId, meter, when, reading,
    reason=""):
    if (not isinstance(reading, int) or reading < 0) and not reason:
        raise Error("Invalid reading")
    if meter not in Manager.ReadingForMeter:
        raise Error("Invalid meter ID")
    username = self._username_for_sessionid(sessionId)
    reading = Reading(when, reading, reason, username)
    Manager.ReadingForMeter[meter] = reading

```

上面这个方法与 XML-RPC 版本很像，但这次不用再转换 `when` 中的日期 / 时间了，而且也不用返回 `True`，直接使用默认的返回值 `None` 就好，因为 RPyC 完全可以接受这种返回值。

---

<sup>①</sup> 指的就是 `Manager.SessionId += 1` 这行代码，其中的 “`+=`” 操作符可以视为“递增并赋值”操作。——译者注

## 1. 简单的线程安全字典

如果使用 CPython（也就是标准的 Python 解释器，它是用 C 实现的），那么由于有 GIL（Global Interpreter Lock，全局解释器锁），所以 dict 在理论上似乎是线程安全的。无论有多少 CPU 核心，Python 解释器在同一时刻都只能执行在一个线程上面，因此对于单个方法来说，这种操作都是“原子操作”（atomic action）。然而本例却不是这样，因为我们要执行的“单一原子操作”是由 dict 中的好几个方法构成的，所以还是无法保证整套操作是“线程安全的”。而且无论何时，我们都应该依赖这种实现细节，因为毕竟还有很多 Python 解释器是没有 GIL 的（比如 Jython 和 IronPython），在这些解释器中，就连单个的 dict 方法都无法保证其原子性。

如果真想使用线程安全的字典，那么要么采用第三方程序库，要么自己来实现。自己实现其实不难，我们可以用现有的 dict 为基准，把外界对其的访问行为封装到自编的“线程安全方法”（thread-safe method）里，然后令外界通过这些方法来执行原有的访问操作。本节将会讲解 ThreadSafeDict 类，它是个能在多线程环境下使用的字典，其接口虽是 dict 接口的“子集”（subset），但却足以提供 meter 程序所需的字典功能。

```
class ThreadSafeDict:
    def __init__(self, *args, **kwargs):
        self._dict = dict(*args, **kwargs)
        self._lock = threading.Lock()
```

ThreadSafeDict “聚合”（aggregate）了 dict 与 threading.Lock 对象。由于我们只是想令外界对 self.\_dict 的访问保持“串行化”（serialized，也就是同一时刻只能有一个线程访问 self.\_dict），所以 ThreadSafeDict 不用继承 dict，只需将其聚合进来即可。

```
def copy(self):
    with self._lock:
        return self.__class__(**self._dict)
```

由于 Python 的锁支持 context manager 协议，因此加锁很简单，只需多写一条 with 语句即可。锁会在无用时自行释放，即便 with 块里发生了异常，这把锁也一定能释放掉。

某线程在执行到 with self.\_lock 语句时，如果其他线程取得了锁，那么该线程就会阻塞，直到获得这把锁之后，才能继续执行 with 块里的语句。也就是说，只有当其他线程没有占据这把锁时，本线程才能执行 with 块里的语句。有鉴于此，在加锁时，执行的语句应该越少越好，而且越快越好。但在本例中，with 块里的操作却开销很大，这是不得已的，因为没有更好的办法了。

如果某个类实现了 copy() 方法，那么该方法应该给调用者返回本实例的一份拷贝。我们不能直接返回 self.\_dict.copy()，因为那是个普通的 dict，而不是 ThreadSafeDict。返回 ThreadSafeDict(\*\*self.\_dict) 倒是可以，但如果 ThreadSafeDict

还有子类，而且子类又没有重新实现 `copy()` 方法，那么就不能这么做了。于是笔者选用了 `self.__class__(**self._dict)` 这一返回值，它既适用于 `ThreadSafeDict`，又适用于其子类。（“\*”及“\*\*”的用法参见 1.2 节“序列与映射的解包操作”。）

```
def get(self, key, default=None):
    with self._lock:
        return self._dict.get(key, default)
```

上述方法“忠实地”实现了线程安全版的 `dict.get()` 方法。

```
def __getitem__(self, key):
    with self._lock:
        return self._dict[key]
```

实现了上面这个特殊方法之后，我们就可以通过键来访问字典里的值了，比如：`value = d[key]`。

```
def __setitem__(self, key, value):
    with self._lock:
        self._dict[key] = value
```

上面这个特殊方法使开发者可以用“`d[key] = value`”这种写法向字典中插入新的条目，也可以修改既有条目中的值。

```
def __delitem__(self, key):
    with self._lock:
        del self._dict[key]
```

上面这个特殊方法使 `ThreadSafeDict` 支持 `del` 语句，也就是说，开发者可以用 `del d[key]` 来删除字典中的条目。

```
def __contains__(self, key):
    with self._lock:
        return key in self._dict
```

如果字典里某个条目的键与 `key` 参数相符，那么上面这个特殊方法应该返回 `True`，否则应返回 `False`。实现该方法之后，我们就可以在 `in` 关键字里使用 `ThreadSafeDict` 了，例如：`if k in d: ...`。

```
def __len__(self):
    with self._lock:
        return len(self._dict)
```

上面这个特殊方法返回字典中的条目数量。实现了该方法之后，我们就可以用 Python 内置的 `len()` 函数来查询 `ThreadSafeDict` 的条目数量，例如：`count = len(d)`。

`ThreadSafeDict` 没有提供 `dict` 类里的 `clear()`、`fromkeys()`、`items()`、`keys()`、`pop()`、`popitem()`、`setdefault()`、`update()` 及 `values()` 方法。在这些方法中，大部分都很好实现。但是返回“视图”（view）的那几个方法要特别注意，比如 `items()`、`keys()`、`values()` 等。最简单、最安全的办法就是根本不要去实现它们。还有一种办法是把字典数据拷贝到 `list` 里面，然后把那个 `list` 返回给调用者，例如，我们可以

用 `with self._lock: return list(self._dict.keys())` 当作方法体来实现 `keys()` 方法。如果字典很大，那么这种做法将耗费大量内存，而且一旦有线程执行此方法，想执行该方法的其他线程就会阻塞。

还有个办法也能实现线程安全的字典，那就是先在某个线程里创建一份普通的字典，然后非常小心地操作这个字典，只在创建它的那个线程里执行写入操作（或是设置一把锁，只有获得了锁的线程，才能向字典里写入数据），并通过 `types.MappingProxyType` 类（它是 Python 3.3 新加入的类）向其他欲操作此字典的线程提供“只读视图”（`read-only view`，也就是 `thread-safe view`（线程安全的视图））。

## 2. 继承自 `ThreadSafeDict` 的 `_MeterDict` 类

我们不打算用普通的 `ThreadSafeDict` 对象来表示保存仪表编号及仪表读数的那个字典，而是新建名为 `_MeterDict` 的私有子类，并为其添加两个方法<sup>Θ</sup>。

```
class _MeterDict(ThreadSafeDict):
    def insert_if_missing(self, key, value=None):
        with self._lock:
            if key not in self._dict:
                self._dict[key] = value
                return True
        return False
```

上面这个方法会把给定的键和值放到字典里，并返回 `True`；假如键（也就是虚构的仪表编号）已经在字典里了，那么该方法就不修改字典了，而是直接返回 `False`。这样做可以确保每次领取抄表任务时，任务里的仪表编号都是新的，而且各不相同。

`insert_if_missing()` 方法所执行的代码实际上就相当于：

```
if meter not in ReadingForMeter: # WRONG!
    ReadingForMeter[key] = None
```

由于 `ReadingMeter` 是个 `_MeterDict` 实例，而 `_MeterDict` 又继承了 `ThreadSafeDict`，所以 `ReadingMeter` 能够使用 `ThreadSafeDict` 的所有功能。在上面这两行代码中，`in` 语句会调用 `ReadingForMeter.__contains__()` 方法，而 “[ ]” 操作则会调用 `ReadingForMeter.__setitem__()` 方法，尽管这两个方法各自都是线程安全的，但它们合起来却不能保证线程安全。因为在某个线程执行完 `if` 语句而又尚未执行 “`ReadingForMeter[key] = None`” 这条赋值语句时，另一个线程可能突然闯入，并访问 `ReadingForMeter` 字典。笔者采用的解决办法是把这两个操作放在同一个加锁范围内执行，这正是 `insert_if_missing()` 方法中那段代码的含义。

---

<sup>Θ</sup> 如果不实现 `_MeterDict` 的 `items()` 方法，那么 `meterserver-rpyc.py` 服务器程序在停止时可能会出错。读者可以自己来实现这个方法。——译者注

```

def status(self, username):
    count = total = 0
    with self._lock:
        for reading in self._dict.values():
            if reading is not None:
                total += 1
                if reading.username == username:
                    count += 1
    return count, total

```

上面这个方法的开销可能比较大，因为它要在加锁的情况下遍历底层 `_dict` 中的每个值。还有个办法，是通过 “`values = self._dict.values()`” 语句把这些值都赋给 `values` 变量，并且只对这一句加锁，把锁释放掉之后，再通过 `values` 来遍历（也就是说，在不加锁的情况下执行遍历操作）。到底是在加锁时先把值拷贝到 `values` 里，然后在不加锁的情况下遍历比较快，还是直接在加锁的情况下遍历比较快，这要根据具体环境来定。如果想知道确切结果，那当然只能在真实的应用场景中对这两种办法做性能分析了。

## 6.2.2 编写 RPyC 服务器

早前大家已经看到，用 `xmlrpclib.server` 模块来创建 XML-RPC 服务器（参见 6.1.2 节）是非常简单的。通过本节我们就会发现，用 RPyC 来创建服务器也同样简单。

```

import datetime
import threading
import rpyc
import sys
import MeterMT

PORT = 11003

Manager = MeterMT.Manager()

```

上面这几行代码是 `meterserver-rpyc.py` 的开头部分。我们首先引入两个 Python 标准库里的模块，然后引入 `rpyc` 模块，最后引入线程安全的 `MeterMT` 模块。虽说本程序的端口号是固定的，但你也可以像早前的 XML-RPC 版本那样，使用户能够通过命令行选项来指定端口号，然后在程序里用 `argparse` 模块将其解析出来。设定好端口号之后，我们创建 `MeterMT.Manager` 实例。RPyC 服务器里的线程将共享这一实例。

```

if __name__ == "__main__":
    import rpyc.utils.server
    print("Meter server startup at {}".format(
        datetime.datetime.now().isoformat()[:19]))
    server = rpyc.utils.server.ThreadedServer(MeterService, port=PORT)
    thread = threading.Thread(target=server.start)
    thread.start()
    try:
        if len(sys.argv) > 1: # Notify if called by a GUI client

```

```

        with open(sys.argv[1], "wb") as file:
            file.write(b"\n")
        thread.join()
    except KeyboardInterrupt:
        pass
server.close()
print("\rMeter server shutdown at {}".format(
    datetime.datetime.now().isoformat()[:19]))
MeterMT.Manager._dump()

```

上面列出了服务器程序结尾部分的代码。我们引入 `rpyc.utils.server` 模块并打印消息，告诉用户服务器启动了。然后创建 `ThreadedServer` 实例，创建该实例的时候，我们传入了 `MeterService` 类。服务器将会根据需要来创建 `MeterService` 类的实例，我们马上就会讲到这个类。

创建好 `server` 之后，我们本来可以直接写一句 `server.start()`，然后就不管了。这么做确实能启动服务器，但会令它“永远”执行下去（`run "forever"`）。而我们却想使用户能通过 `Ctrl+C` 组合键（或是 `INT` 信号）来停止服务器，并且还想使服务器在停止时能够打印出它所收集到的读数。

为了实现这一功能，我们把服务器放在它自己的线程里启动，这样的话，它就会在那个线程里创建“线程池”（`thread pool`），以管理客户端发来的连接。启动了服务器之后，我们在当前线程里调用 `thread.join()`，这会导致本线程阻塞，直到服务器线程结束为止。如果服务器中断了，那么我们可以捕获到 `KeyboardInterrupt` 异常。忽略掉该异常之后，我们调用 `server.close()`，以便关闭服务器。`close()` 方法会阻塞当前线程，直到服务器里的每条线程都处理完各自的连接为止。接下来，我们向控制台里打印消息，告诉用户服务器已关闭，然后把客户端提交给服务器的仪表读数打印出来。

如果服务器是由 GUI 客户端启动的，那么客户端应该会通过 `notify` 参数向服务器传入文件名，在这种情况下，应该有且仅有 `notify` 这一个参数。服务器若发现指定了 `notify` 参数，则会创建相应的文件，并向其中写入换行符，而客户端一旦发现该文件创建好，就会认为服务器已经启动并正常运行了。

采用“基于服务的模式”（`service-based mode`, `service mode`）来创建 RPyC 服务器时，应该为其指定 `rpyc.Service` 的子类，服务器会把这个子类当成“类工厂”（`class factory`），用它来创建相关“服务”（`service`）的实例。（第 1 章的 1.1 节与 1.3 节分别讲解了两种与工厂相关的设计模式。）我们在程序开头创建了 `MeterMT.Manager` 实例，而本例所用的 `MeterService` 类只不过是对这个 `Manager` 略加包装而已。

```

class MeterService(rpyc.Service):
    def on_connect(self):
        pass

    def on_disconnect(self):
        pass

```

客户端连接到某个 service 时，系统会执行它的 `on_connect()` 方法。与之类似，如果某个连接已完成，那么系统会调用 `service` 的 `on_disconnect()` 方法。本例不需要处理这两种情况，所以我们把两者设计成“什么事都不做的方法”(do nothing method)。如果某个方法根本就用不到，那么完全可以不去实现这个方法。笔者把这两个方法列在上面，只是为了使读者看到其签名而已。

```
exposed_login = Manager.login
exposed_get_status = Manager.get_status
exposed_get_job = Manager.get_job
```

`service` 可以将方法（或类及其他对象）“展示”(`expose`)给客户端。只要类名或方法名以 `exposed_` 开头，那么客户端就可以访问，而且访问的时候加或不加这个前缀都行。比方说，RPyC 客户端想登录 meter 服务器的时候，既可以调用 `exposed_login()`，又可以调用 `login()`。

我们令 `exposed_login()`、`exposed_get_status()` 及 `exposed_get_job()` 这三个方法分别指向 `MeterMT.Manager` 实例中的相关方法。

```
def exposed_submit_reading(self, sessionId, meter, when, reading,
                           reason=""):
    when = datetime.datetime.strptime(str(when)[:19],
                                      "%Y-%m-%d %H:%M:%S")
    Manager.submit_reading(sessionId, meter, when, reading, reason)
```

对于上面这个方法来说，我们必须把 `Manager.submit_reading()` 方法略加包装才行。原因在于，RPyC 传给该方法的那个 `when` 变量是经过 `netref` 包装过的 `datetime.datetime` 对象，而不是普通的 `datetime.datetime`。在绝大多数情况下，这都不会出问题，但我们想在字典里存放实际的 `datetime.datetime`，而不是指向远端（也就是客户端）`datetime.datetime` 对象的引用。所以，我们要把包装在 `netref` 里的日期 / 时间转换成 ISO 8601 格式的日期 / 时间字符串，并将其解析为服务器端的 `datetime.datetime` 对象，然后传给 `MeterMT.Manager.submit_reading()` 方法。

上面列出的这些就是 RPyC 服务器的全部代码了，如果不写 `on_connect()` 与 `on_disconnect()` 方法，那么还能再省几行代码。

### 6.2.3 编写 RPyC 客户端

由于 RPyC 客户端与 XML-RPC 客户端非常相似，所以本节只讲解两者的不同之处。

#### 1. 基于控制台的 RPyC 客户端

与 XML-RPC 客户端一样，RPyC 客户端的使用者也必须单独去启动及停止服务器，而且只有在服务器已经运行起来的情况下，RPyC 客户端才能正常运作。

`meterclient-rpyc.py` 程序与早前讲过的 `meterclient-rpc.py` 程序（参见

6.1.3 节) 几乎完全一样, 只是 main() 与 submit() 函数不同。

```
def main():
    username, password = login()
    if username is not None:
        try:
            service = rpyc.connect(HOST, PORT)
            manager = service.root
            sessionId, name = manager.login(username, password)
            print("Welcome, {}, to Meter RPYC".format(name))
            interact(manager, sessionId)
        except ConnectionError as err:
            print("Error: Is the meter server running? {}".format(err))
```

RPyC 客户端的 main() 方法与原来相比, 第一个区别在于主机名与端口号都是用“硬编码”(hard-code) 直接写在程序里的。如果你要像 XML-RPC 客户端那样, 使用户可以配置这两个选项, 那么只需略微修改代码即可。第二个区别在于: 原来我们是先针对服务器端的 Manager 创建 proxy, 然后再通过 proxy 与服务器通信, 而这次则是先连接到提供服务的服务器上面。在本例中, 服务器只提供一种服务, 也就是 MeterService 服务, 我们可以通过它做中介与服务器端的 MeterMT.Manager 通信。至于其他代码, 也就是登录服务器、领取抄表任务、提交读数、获取服务器状态所用的那些代码, 都和原来相同, 唯一例外的是 submit() 函数, 它这次要捕获的异常和 XML-RPC 客户端不一样。

保持主机名与端口号同步是个很费劲的工作, 端口号发生冲突时更是如此, 我们有时不得不选一个平常很少用到的端口。此问题可以用 registry server(注册服务器) 来解决。如果采用这个办法, 那么就必须在网络中启动由 RPyC 所提供的 registry\_server.py 服务器。用 RPyC 编写的服务器程序在启动时会自动找寻 registry server, 如果找到了, 那么会把自己所能提供的服务注册到 registry server 里。然后, 客户端就不用通过 rpyc.connect(host, port) 来连接服务器了, 而是可以改用 rpyc.connect\_by\_service(service) 语句, 例如: rpyc.connect\_by\_service("Meter")。

## 2. 带图形界面的 RPyC 客户端

图 6.2 演示了 meter-rpyc.pyw 程序运行时的样子, 这是个带图形界面的 RPyC 客户端。实际上, 在同一个操作系统里, 我们可以通过外观区分出 RPyC 客户端与 XML-RPC 客户端。

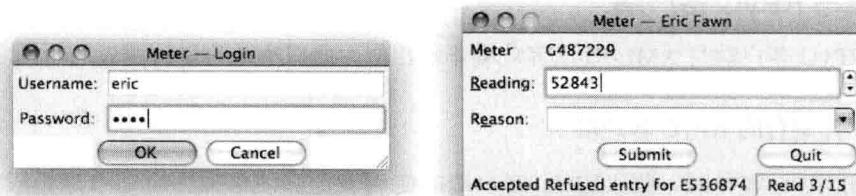


图 6.2 带图形界面的 RPyC 客户端在 OS X 系统中的样貌, 左侧是登录窗口, 右侧是主窗口

我们想用 Tkinter 创建 GUI 式的 RPyC 客户端，如果启动时已有 meter 服务器运行，那么它会自动与之连接，若没有，则会自行启动之。这种客户端的代码实际上与早前 GUI 式的 XML-RPC 客户端几乎一样，只是有两个方法和几条 import 语句需要修改，另外还要稍微改动几个常量，而且 except 子句中的异常也有所不同。

```
def connect(self, username, password):
    try:
        self.service = rpyc.connect(HOST, PORT)
    except ConnectionError:
        filename = os.path.join(tempfile.gettempdir(),
                               "M{}.$$.format(random.randint(1000, 9999)))
        self.serverPid = subprocess.Popen([sys.executable, SERVER,
                                           filename]).pid
        self.wait_for_server(filename)
        try:
            self.service = rpyc.connect(HOST, PORT)
        except ConnectionError:
            self.handle_error("Failed to start the RPYC Meter server")
            return False
        self.manager = self.service.root
    return self.login_to_server(username, password)
```

抄表员在登录窗口里输入用户名和密码并按下 OK 按钮之后，客户端程序就会执行上面这个方法，该方法试着连接服务器并登录 MeterMT.Manager。

如果连接服务器的操作失败了，那么我们就认为服务器还没启动，于是客户端会尝试启动服务器，并把某个文件名通过 notify 参数传过去。虽然启动服务器的操作本身不会阻塞（也就是说，“启动服务器”是个“异步”（asynchronous）操作），但我们必须等服务器完全启动好之后才能连接它。wait\_for\_service() 方法就是用来做这件事的，该方法和早前那个版本（参见 6.1.3 节）几乎完全相同，只是这次可能会抛出 ConnectionError，而不是像原来那样，在无法连接服务器时调用 self.handle\_error() 方法。如果能够连接到服务器，那么我们就针对服务器一方的 MeterMT.Manager 来创建 proxy，并把它赋给 self.manager，然后通过 self.manager 来登录 meter 服务器。

```
def login_to_server(self, username, password):
    try:
        self.sessionId, name = self.manager.login(username, password)
        self.master.title("Meter \u2014 {}".format(name))
        return True
    except rpyc.core.vinegar.GenericException as err:
        self.handle_error(err)
    return False
```

如果抄表员输入的用户名和密码是正确的，那么我们就设置好 self.sessionId 的值，并且根据抄表员的全名来修改应用程序的“标题栏”（title bar）。若登录失败，login\_to\_server() 则返回 False，这会使应用程序终止。如果服务器是由 GUI 应用程序启动的，那么这也将使服务器终止。

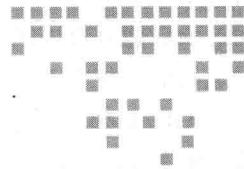
本章的范例程序都没有采用加密技术，所以有人可能会窃听到客户端与服务器之间的通信。对于这种不传输“私人数据”（private date）的应用程序来说，这可能根本不算什么问题，如果客户端和服务器都在同一台电脑中运行，而这台电脑又处在防火墙保护之下，或两者之间的网络通信本身就是加密的，那么也同样不会有危险。如果一定要加密的话，也完全有办法实现。对于 XML-RPC 程序来说，一个办法就是使用第三方的 PyCrypto 包（[www.dlitz.net/software/pycrypto](http://www.dlitz.net/software/pycrypto)），它可以把经由网络所传输的全部数据都加密。另外一种办法是采用 Transport Layer Security (“secure sockets”) 协议<sup>②</sup>来传输，Python 的 `ssl` 模块支持该协议。对于 RPyC 来说，想实现加密就更简单了，因为它内置了加密功能。RPyC 可以通过“密钥”和“证书”来进行 SSL 通信，也可以通过“SSH tunneling”（SSH 隧道，SSH 是 Secure Shell 的简称）技术对网络通信加密，这种做法要比使用 SSL 简单许多。

Python 对网络编程的支持非常好，从底层到高层都是如此。标准库里几乎包含了所有流行的“高层协议”（high-level protocol），比如传输文件所用的 FTP 协议，收发电子邮件所用的 POP3、IMAP4 及 SMTP 协议，进行网络通信所用的 HTTP 及 HTTPS 协议，等等，此外当然也包括 TCP/IP 及其他“底层 socket 协议”（low-level socket protocol）。Python 里有个名叫 `socketserver` 的中层模块，可以用作实现高层服务器的基础，不过 Python 也直接支持许多高层服务器技术，比方说，可以用 `smtpd` 模块来创建电子邮件服务器，可以用 `http.server` 模块来创建 Web 服务器，也可以像本章这样，用 `xmlrpc.server` 模块来创建 XML-RPC 服务器。

有许多第三方的网络编程模块可供开发者选择，特别是有很多支持 Python WSGI（Web Server Gateway Interface（Web 服务器网关接口），参见：[www.python.org/dev/peps/pep-3333](http://www.python.org/dev/peps/pep-3333)）的 Web 框架。[wiki.python.org/moin/WebFrameworks](http://wiki.python.org/moin/WebFrameworks) 页面详细列出了第三方的 Python Web 框架，有关 Web 服务器的更多信息请参阅 [wiki.python.org/moin/WebServers](http://wiki.python.org/moin/WebServers)。

---

<sup>②</sup> Transport Layer Security 协议中文称为“传输层安全协议”，简称 TLS 协议；Secure Sockets Layer 协议中文称为“安全套接层协议”，简称 SSL 协议。——译者注



## 第 7 章

*Chapter 7*

# 用 Tkinter 开发图形用户界面

精心设计的 GUI (graphical user interface, 图形用户界面) 应用程序可以向用户呈现出一套非常直观、非常新颖同时又极易使用的操作界面。应用程序越复杂，对 GUI 的要求也就越高，尤其是当 GUI 里面包含应用程序专用的“自定义控件”<sup>①</sup> (custom widget) 时，就更要注重 GUI 的设计了。与 GUI 程序相比，web 应用程序的界面就不那么直观，浏览器里显示出来的菜单、工具栏和控件都是如此。除非 HTML5 的 canvas 技术得以广泛运用，否则 web 应用程序绘制自定义控件的手段就非常有限。此外，web 应用程序的运行效率也没有“本机应用程序”(native application)<sup>②</sup>高。

在智能手机上，越来越多的应用程序都可以通过语音来控制了，不过在台式电脑、笔记本电脑及平板电脑上面，大部分应用程序依然是传统的 GUI 程序（需要通过鼠标、键盘或语音来控制）或触摸式程序。笔者编写本书时，“触控设备”(touch-controlled device) 基本上都使用了“专属的程序库”(proprietary library)<sup>③</sup>，而且开发者必须用特定的编程语言及开发工具才能为这种设备开发应用程序。所幸有个第三方的开源程序库叫做 Kivy ([kivy.org](http://kivy.org))，使开发者可以用 Python 代码实现跨平台的触摸式程序，这样一来就可以解决刚才提到的那个问题了。虽说如此，但大部分“基于触摸的界面”(touch-based interface) 都是为处理器能力较低且屏幕较小的设备而设计的，在这种设备上，用户同一时刻也许只能看到一个应用程序。

桌面操作系统的用户和电脑玩家们更想完全发挥出大屏幕及高端处理器的优势，而传统

① 在 Windows 操作系统中开发 GUI 的程序员经常使用“control”、“container”或“form”等词来描述 GUI 对象。本书采用更为通用的“widget”一词，它是 Unix GUI 编程中的术语。

② 也称“本地应用程序”。此处大致可以理解为直接运行在操作系统里的应用程序。——译者注

③ 也叫做“专有的程序库”、“有专利的程序库”。——译者注

的 GUI 应用程序最容易做到这一点。此外，某些操作系统（比如新版的 Windows）提供了语音控制功能，而这项功能也可以同现有的 GUI 应用程序相配合。用 Python 开发的命令行程序是可以跨平台运行的，与之类似，用 Python 开发的 GUI 程序也能跨平台运行，只要选对了“GUI 工具包”（GUI toolkit）就行。有很多 GUI 开发工具包可供选择。笔者简述下列四组工具包，它们都已移植到 Python 3，而且至少能在 Linux、OS X 及 Windows 这三个操作系统中运行，其界面与直接运行在相关系统中的普通应用程序相仿。

- **PyGtk 与 PyGObject**：PyGtk ([www.pygtk.org](http://www.pygtk.org)) 是个稳健而成功的项目，不过，在 2011 年停止了开发。后继者名叫 PyGObject ([live.gnome.org/PyGObject](http://live.gnome.org/PyGObject))。但不巧的是，在笔者编写本书时，PyGObject 并不能算是完全跨平台的，因为开发出来的程序似乎只能在基于 Unix 的操作系统里运行。
- **PyQt4 与 PySide**：PyQt4 ([www.riverbankcomputing.co.uk](http://www.riverbankcomputing.co.uk)) 为“Qt4 GUI 应用程序开发框架”（Qt 4 GUI application development framework, [qt-project.org](http://qt-project.org)）提供了 Python 式的绑定机制。PySide ([www.pyside.org](http://www.pyside.org)) 项目出现得比较晚，它高度兼容于 PyQt4，而且授权协议更为宽松。PyQt4 也许是最稳定、最成熟的跨平台 Python GUI 工具包了<sup>⊖</sup>。
- **Tkinter**：Tkinter 提供了对 Tcl/Tk GUI 工具包 ([www.tcl.tk](http://www.tcl.tk)) 的绑定。Python 3 附带的 Tcl/Tk 一般是 8.5 版本，Python 3.4 或后续的 Python 版本可能会将其换成 Tcl/Tk 8.6 版。与这里提到的其他工具包不同，Tkinter 只提供了非常基本的功能，而没有提供“工具栏”（toolbar）、“可停靠式窗口”（dock window）及“状态栏”（status bar）（我们可以自己设法创建这些控件）。另外，其他工具包都会自动处理与具体操作系统有关的很多特性，比如 OS X 系统的“universal menu bar”（通用菜单栏），而 Tkinter 则需要开发者自己去处理各种操作系统之间的许多差别，至少在 Tcl/Tk 8.5 版是如此。Tkinter 的主要优势在于它是 Python 标准库的一部分，而且与其他工具包相比，它非常轻便。
- **wxPython**：wxPython ([www.wxpython.org](http://www.wxpython.org)) 提供了对 wxWidget 工具包 ([www.wxwidgets.org](http://www.wxwidgets.org)) 的绑定。wxPython 项目已经运作多年了，不过还需要重新编写大量代码才能移植到 Python 3，本书发行的时候，大家应该就能看到成果了<sup>⊖</sup>。

除了 PyGObject 之外，上面列出的这些工具包都提供了用 Python 语言构建跨平台 GUI 程序所需的功能。如果我们只想针对某个特定的平台来开发，那么应该能找到专门针对该平台 GUI 程序库的 Python 绑定库（参见 [wiki.python.org/moin/GuiProgramming](http://wiki.python.org/moin/GuiProgramming)），我们也可以直接使用针对特定操作系统的 Python 解释器，例如 Jython 或 IronPython。如果想绘制三维图形，那么通常可以直接用这些 GUI 工具包来画。另外也可以考虑使用 PyGame ([www.pygame.org](http://www.pygame.org))

<sup>⊖</sup> 向大家透露个消息：笔者曾是 Qt 的“文档管理员”（documentation manager），并且写过一本有关 PyQt4 编程的书：《Rapid GUI Programming with Python and Qt》（详见附录 B）。

<sup>⊖</sup> wxPython 项目于 2013 年底移植到了 Python 3。——译者注

([pygame.org](http://pygame.org)), 若需求相当简单, 则可直接使用 Python 的某种 OpenGL 绑定库来做, 下一章将会讲解此话题。

由于 Tkinter 是标准的 Python 程序库, 所以我们创建出来的 GUI 应用程序部署起来很容易 (在必要时甚至可以把 Python 和 Tcl/Tk 直接同应用程序一起打包发布; 比方说, 可以使用这个工具: [cx-freeze.sourceforge.net](http://cx-freeze.sourceforge.net))。与命令行程序相比, 这种应用程序更直观, 而且更容易为用户所接受, 对使用 OS X 和 Windows 操作系统的用户来说, 更是如此。

本章要演示三个范例程序。首先是非常小的 “hello world” 程序, 其次是简单的货币转换器, 最后是内容较为丰富的 Gravitate 游戏。Gravitate 游戏与 “方块下落式游戏” (TileFall) 或 “同色消除式游戏” (SameGame) 相似, 只不过本游戏会以 “向窗口中心聚拢”的方式来填补消掉的方块所留下的空隙, 而别的游戏则以下落及左移的方式填补。通过 Gravitate 游戏, 大家会明白如何用 Tkinter 来创建 “主窗口风格的” (main-window-style) 应用程序, 并在其中加入菜单、对话框、状态栏等常用控件。7.2.2 节会讲解 Gravitate 游戏的对话框, 而 7.3 节则将剖析 Gravitate 主窗口的架构。

## 7.1 Tkinter 简介

编写 GUI 程序并不比编写其他类型的程序难, 而且还有可能制作出界面美观并深受用户欢迎的产品来。

请大家注意, GUI 编程是个非常庞大的话题, 只用一章篇幅不可能讲得太深, 如果真要深谈 GUI 开发, 那么至少需要一整本书才行。本竟能做的就是讲解 GUI 编程中的几个重点问题, 尤其是如何实现 Tkinter 所没有提供的那些功能。首先我们来看最经典的 “hello world” 程序, 它的源代码写在 `hello.pyw` 文件里, 其运行效果如图 7.1 所示。

```
import tkinter as tk
import tkinter.ttk as ttk
class Window(ttk.Frame):
    def __init__(self, master=None):
        super().__init__(master) # Creates self.master
        helloLabel = ttk.Label(self, text="Hello Tkinter!")
        quitButton = ttk.Button(self, text="Quit", command=self.quit)
        helloLabel.pack()
        quitButton.pack()
        self.pack()

window = Window() # Implicitly creates tk.Tk object
window.master.title("Hello")
window.master.mainloop()
```



图 7.1 对话框式的 “Hello World” 程序在 Linux、OS X 及 Windows 系统中的样貌

上面就是 `hello.pyw` 程序的全部代码了。许多 Tkinter 开发者喜欢把 Tkinter 中的全部名称都引入程序的命名空间里（例如，以 `from tkinter import *` 语句来引入 Tkinter），但是笔者选择把它们留在各自的命名空间中（只是引入后的名称比原来短了：`tkinter` 引入为 `tk`, `tkinter.ttk` 引入为 `ttk`），这样我们就能知道某个类来自哪里了。（`ttk` 模块是个包装器，它是针对 Tcl/Tk 的官方 Tile 扩展。）我们本来可以只写第一行引入语句，然后使用 `tkinter.Frame` 来取代 `tkinter.ttk.Frame`，并且把原来使用 `ttk` 的地方都改用 `tk` 来做，但是 `tkinter.ttk` 版本的类提供了主题方面的支持，所以应该优先选用，尤其是在 OS X 及 Windows 操作系统中。

大部分普通的 `tkinter` 控件都有带主题的 `tkinter.ttk` 版本与之对应。由于“普通控件”（plain widget）与“带主题的控件”（themed widget）未必总是具有相同的界面，而且在某些场合只能使用普通控件，所以一定要阅读开发文档。（如果你能看懂 Tk/Tk 代码，那么笔者推荐 [www.tcl.tk](http://www.tcl.tk) 网站上的文档；否则请访问 [www.tkdocs.com](http://www.tkdocs.com) 网站，这里有一些以 Python 及其他语言写成的范例，另外也可以去 [infohost.nmt.edu/tcc/help/pubs/tkinter/web](http://infohost.nmt.edu/tcc/help/pubs/tkinter/web) 看看，这里提供了实用的 Tkinter 教程及参考资料。）另外，`tkinter-ttk` 里某些带主题的控件（例如 `tkinter.ttk.Combobox`、`tkinter.ttk.Notebook`、`tkinter.ttk.Treeview` 等）没有与之对应的普通控件。

本书所采用的 GUI 编程风格是为每个窗口都创建对应的类，而且通常会把该类放到各自的模块中。对于“顶级窗口”（top-level window，也就是应用程序的主窗口）来说，我们通常会像本例这样，用 `tkinter.Toplevel` 或 `tkinter.ttk.Frame` 的子类来表示。Tkinter 维护了一套“所有权体系”（ownership hierarchy），该体系由“父控件”（parent widget）与“子控件”（child widget）之间的上下级关系构成，父控件与子控件也称“主控件”（master widget）与“从控件”（slave widget）。然而总体来说，我们不用担心这个问题，如果从某个控件类中继承了子类，那么只需在其 `__init__()` 方法中调用内置的 `super()` 函数即可。

大多数 GUI 应用程序都遵循标准的创建流程：先创建一个或多个类，用以表示程序中的窗口，其中之一是主窗口。然后针对每个窗口类，创建此窗口需要用到的变量（`hello.pyw` 文件里没有这样的变量），创建控件，调整控件布局，并指定一些方法，用以响应各种事件（例如鼠标点击、键盘敲击、超时等）。在本例中，我们把用户点击 `quitButton` 按钮的行为同继承下来的 `tkinter.ttk.Frame.quit()` 方法关联起来，该方法会关掉窗口，由于此窗口是应用程序中唯一的顶级窗口，所以关掉它之后，整个应用程序就会照常终止。等所有窗口类都准备好之后，再来创建“应用程序对象”（application object）（在本例中，系统会自动创建该对象）并启动 GUI 事件循环。4.3.1 节中的图 4.8 演示了事件循环。

大部分 GUI 程序的代码肯定要比 `hello.pyw` 更长，而且更复杂。不过，其窗口类的创建流程依然与此处相同，只是需要创建的控件和需要关联的事件远远多于本例。

大多数时下流行的 GUI 工具包都使用“布局”（layout）来排布控件，而不会把控件的大

小及位置写成固定值（hard code，“硬代码”）。这么做的好处是，每个控件既可以保持与其他控件的相对位置关系，同时又能自动扩大或缩小尺寸，以便与其内容（比如标签或按钮中的文本）相吻合，即使内容变了，也依然可以做到这一点。采用布局来设计 GUI，可以为程序员免去许多乏味的计算工作。

Tkinter 提供了三种“布局管理器”（layout manager）：第一种是“place”（放置），就是用固定值来描述控件位置，这种布局很少使用；第二种是“pack”（填充），就是根据某个假想的“中心点”（central cavity）来排布控件；第三种是“grid”（网格），就是把控件按行列位置摆放在网格里，这种布局最为流行。本例采用 pack 式布局：我们先调用标签和按钮的 pack() 方法，然后调用整个窗口的 pack() 方法。对于这种非常简单的窗口来说，采用 pack 式布局比较合适，但在后面的例子中大家就会看到，grid 布局用起来是最为简单的。

GUI 应用程序分为两大类：“对话框式”（dialog style）程序和“主窗口式”（main-window style）程序。前者是既没有菜单又没有工具栏的窗口，用户一般会通过窗口中的按钮及“下拉列表框”（combobox，组合框）等控件与之交互。对于“小工具”（small utility）、“媒体播放器”（media player）及某些游戏来说，由于其用户界面非常简单，所以很适合设计成对话框式的应用程序。而主窗口式的应用程序则有“中心区域”（central area），区域上方通常有菜单与工具栏，下方有状态栏，而且还可能带有“可停靠式窗口”（dock window）。对于复杂一些的应用程序来说，使用主窗口式风格更为合适，这些程序经常需要把选项整理成菜单或把相关按钮放在工具栏里，当用户点击这些菜单项或按钮时，会弹出对话框。这两类程序本章都要讲到，不过我们先从对话框式的程序开始，因为这部分内容基本上也适用于主窗口式的程序所弹出的对话框。

## 7.2 用 Tkinter 创建对话框

对话框按照“模态”（modality）可以分为四种，从智能程度上也可以分为不同级别。我们先来简述这四种 modality，然后再讨论智能程度。

- **全局模态（Global Modal）：**全局模态窗口会阻塞整个操作系统的用户界面，使用户只能与本窗口相交互，而不能操作其他应用程序。除了操作本窗口，用户既不能切换到其他应用程序，也不能做别的事情。这种窗口有两个常见的用法：一个是作为操作系统启动时的登录对话框，另一个是从上了密码的屏幕保护程序中跳出时所显示的解锁框。普通应用程序的开发者决不应该使用全局模态窗口，因为一旦其中有 bug，那么就将导致整台电脑无法使用。
- **应用程序级模态（Application Modal）：**应用程序级别的模态窗口会阻止用户操作程序里的其他窗口，但用户仍然可以切换至系统里的其他应用程序。模态窗口的代码要比非模态窗口的好写，因为用户无法以开发者不可预见的方式改变应用程序状态。但是，某些用户会觉得这种窗口操作起来不太方便。

- **窗口级模态 (Window Modal)**: 窗口级别的模态与应用程序级别的模态类似，但它并不能完全阻止用户操作应用程序里的所有其他窗口，而是只能阻止用户操作位于“同一窗口体系”(same window hierarchy) 里的其他窗口。这种模态很有用，比方说，用户打开了两个顶级窗口，每个窗口里都显示了一份文档，如果某窗口弹出了对话框，那么我们可以把该对话框设为窗口级别的模态对话框，这样一来，该对话框就只能阻止用户操作当前这个文档窗口，而不能阻止用户操作另一个文档窗口；但假如把它设置成应用程序级别的模态对话框，那么用户就连另一个文档窗口都无法操作了。
- **非模态 / 无模态 (Modeless)**: 非模态对话框既不会阻塞本应用程序中的窗口，也不会阻塞其他应用程序中的窗口。非模态对话框编写起来要比模态对话框困难，因为程序员必须考虑到用户对应用程序内其他窗口所做的操作，那些操作也许会改变应用程序的状态，而这些状态有可能正是非模态对话框所要依赖的。

用 Tcl/Tk 的术语来说，全局模态窗口具备 “global grab”，而应用程序级模态窗口与窗口级模态窗口（这两类窗口一般统称为“模态窗口”(modal window)）则具备 “local grab”。在 OS X 操作系统中，用 Tkinter 开发的某些模态窗口看起来和 sheet 一样<sup>⊖</sup>。

“dumb”（朴拙的）对话框通常是供用户向应用程序里输入数据的，这种对话框并不知道与应用程序有关的信息。用户在登录应用程序时通常会遇到这种对话框，它只接受用户名和密码，并将其传给程序。（在前一章的 6.1.3 节里，我们看到过这样的对话框，其代码位于 MeterLogin.py 文件中。）

“smart”（智能的）对话框对应用程序中的信息有所了解，它们甚至会收到一些引用，这些引用指向应用程序里的变量或数据结构，从而使对话框可以直接操作应用程序里的数据。

模态对话框既可以拙，又可以巧，还可以介乎二者之间。有这样一种相当智能的模态对话框：它不仅知道用户所应输入的每一条数据是什么意思，而且还能判断出这些数据组合起来是否符合应用程序的要求。比方说，输入起止日期的对话框如果足够智能的话，那么就应该检查用户所输入的结束日期是不是比起始日期还早，如果是，那么就拒绝这一输入。

非模态对话框一般来说都是 smart 对话框，它们通常可划分成两种风格：“apply/close”（应用 / 关闭）与 “live”（即时）。“apply/close 式对话框”允许用户操作其中的控件，并点击 Apply 按钮，然后用户就能在主窗口里看到效果了。而 “live 式对话框”则会在用户操作其控件时立即把修改之后的值运用到程序中去，这在 OS X 系统中很常见。非模态对话框如果比较智能，那么会提供“撤销 / 重做”(undo/redo) 功能或 “Default” 按钮（可将控件重设为应用程序的默认值），而且可能还有 “Revert” 按钮（可将控件归复到本对话框初次弹出时的值）。非模态对话框也可以做成 dumb 对话框，这种对话框只是提供信息而已，例如显示帮助信息所用的 Help 对话框。这些对话框通常只有 “Close” 按钮。

---

<sup>⊖</sup> “sheet” 是 Mac OS X 系统对某些模态窗口的称谓，这种窗口会从其父窗口中“冒出来”。——译者注

非模态的对话框在改变颜色、字型（font）、格式或模板时尤其有用，因为用户可以看到修改之后的效果，若是不满意，则可以当场尝试其他效果。同样的功能假如改用模态对话框来做，那么用户就必须打开对话框，做出修改，确认并关闭对话框，然后观看效果，若是不满意，则又要重复这个过程，直到满意为止。

对话框式的应用程序的主窗口就是个非模态对话框，而主窗口式的应用程序则通常会同时用到模态对话框与非模态对话框：用户选中某个菜单项或点击工具栏里的某个按钮之后，可能需要弹出模态对话框，也可能需要弹出非模态对话框。

### 7.2.1 创建对话框式应用程序

本节将讲解一个简单而实用的对话框式应用程序，它是个货币换算器，其源代码位于 currency 目录中，应用程序运行效果如图 7.2 所示。



图 7.2 对话框式的 Currency 应用程序在 OS X 及 Windows 系统中的样貌

应用程序里有两个下拉列表框，用于选择货币名称（名称后面写有“货币标识符”（currency identifier）），还有个“数值调整框”（spinbox），用于输入待换算货币的数量，另外有个标签，用于显示把上面那种货币换算成下面这种货币之后的数量。

应用程序的代码分布在三份 Python 文件中：currency.pyw 里是我们要执行的程序，Main.py 表示 Main.Window 类，而 Rates.py 则提供了 Rates.get() 函数，我们在 1.5 节曾经讲过这个函数。此外还有 currency/images/icon\_16x16.gif 及 currency/images/icon\_32x32.gif 这两个图标文件，它们是应用程序在 Linux 及 Windows 操作系统下所使用的图标。

Python 的 GUI 程序可以使用标准的 .py 后缀名，也可以使用 .pyw 后缀名，但在 OS X 及 Windows 系统中，.pyw 后缀名通常与另一个 Python 解释器相关联（比方说，那个解释器可能是 pythonw.exe 而不是标准的 python.exe）。那种解释器可以在不启动控制台的情况下运行应用程序，这对用户来说更友好一些。但对开发者来说，最好还是在控制台里通过标准的 Python 解释器来执行 GUI 程序，因为这样可以看到 sys.stdout 和 sys.stderr 的输出信息，而这些信息有助于调试程序。

#### 1. Currency 程序的 main() 函数

对于比较大的程序来说，最好是把“可执行的”（executable）模块设计得非常小，然后把其他代码分别放在各自的 .py 模块文件里（这些模块是大是小都无所谓）。在配置比较高的电脑上第一次运行程序时，似乎看不到这样做的好处，但是程序运行了一次之后，所有 .py

模块文件（除了那个“可执行的”文件之外）都会编译成字节形式的 .pyc 文件。以后运行程序时，只要 .py 文件没有改动，那么 Python 就可以直接使用相应的 .pyc 文件了，这样一来，程序的启动速度就会比初次运行的时候快。

currency 应用程序的可执行文件叫做 currency.pyw，其中只包含一个小函数，即 main() 函数。

```
def main():
    application = tk.Tk()
    application.title("Currency")
    TkUtil.set_application_icons(application, os.path.join(
        os.path.dirname(os.path.realpath(__file__)), "images"))
    Main.Window(application)
    application.mainloop()
```

函数首先创建 Tkinter 的“应用程序对象”（application object）。该对象其实是个顶级窗口，它一般是不可见的，并且会充当应用程序最终的“父控件”（parent widget，也叫“主控件”（master widget）或“根控件”（root widget））。在 hello.pyw 程序中，我们并没有明确创建该对象，而是由 Tkinter 为我们自动创建的，但通常来说，最好是自己创建此对象，以便调整某些会影响到整个应用程序的设置。比方说，本例就通过该对象把应用程序的标题改成了“Currency”。

本书范例程序里附带了 TkUtil 模块，其中内置了某些支持 Tkinter 编程的便捷函数，另外还有一些模块等遇到的时候再讲。此处我们使用了 TkUtil 模块里的 set\_application\_icons() 函数。

设置好标题及图标之后（OS X 系统会忽略我们所设置的图标），创建应用程序主窗口实例，并把应用程序对象当成父控件（或主控件）传给它，接下来启动 GUI 事件循环。如果事件循环终止了（比方说，我们调用了 tkinter.Tk.quit()），那么应用程序也会随之终止。

```
def set_application_icons(application, path):
    icon32 = tk.PhotoImage(file=os.path.join(path, "icon_32x32.gif"))
    icon16 = tk.PhotoImage(file=os.path.join(path, "icon_16x16.gif"))
    application.tk.call("wm", "iconphoto", application, "-default", icon32,
                        icon16)
```

为了使范例程序看起来完整一些，我们把 TkUtil.set\_application\_icons() 函数的代码列在上面。tk.PhotoImage 类可以加载 PGM、PPM 及 GIF 格式的 pixmap 图像。（Tcl/Tk 8.6 版本应该还支持 PNG 格式。）创建好这两张图像后，我们调用 tkinter.Tk.tk.call() 函数，该函数实际上就相当于发送一条 Tcl/Tk 命令。一般情况下应该尽量避免直接编写底层代码，但本例却必须这么做，因为 Tkinter 没有提供与该功能对应的 Python 绑定。

## 2. Currency 程序的 Main.Window 类

currency 程序的主窗口是按照早前所说的那套流程来创建的，该类的 \_\_init\_\_() 方法

所调用的那些函数就清晰地体现了这一点。本节的所有代码都节选自 currency/Main.py。

```
class Window(ttk.Frame):
    def __init__(self, master=None):
        super().__init__(master, padding=2)
        self.create_variables()
        self.create_widgets()
        self.create_layout()
        self.create_bindings()
        self.currencyFromCombobox.focus()
        self.after(10, self.get_rates)
```

如果我们的类继承自某个控件类，那么就必须在 `__init__()` 方法里调用 Python 内置的 `super()` 函数。本例中，我们不仅把 `master` (`master` 就是由应用程序的 `main()` 函数所创建的那个“应用程序对象”，其类型为 `tk.Tk`) 传给了 `super().__init__()`，而且还传入了值为 2 的 `padding` 参数。`padding` 参数的单位是像素，它表示应用程序窗口的“内边框”(inner border) 与窗口里的控件之间的距离。

接下来，我们创建窗口变量（也就是应用程序要用到的变量）及控件，并把控件排布好。然后，创建“事件绑定”(event binding)，在这之后，我们把键盘焦点交给顶部的下拉列表框，使用户可以选择待换算的货币类型。最后，调用从 Tkinter 继承下来的 `after()` 方法，该方法有两个参数，第一个参数是以毫秒为单位的延迟时间，第二个参数是个 callable，该方法保证至少会经过指定的延迟时间之后再去调用 callable。

由于要从互联网下载汇率数据，所以可能得花上几秒钟才能下载好。但我们还想先把应用程序显示出来，因为如果不这么做的话，那么用户就会认为程序没有启动，于是可能会再次启动它。于是，我们要把下载汇率这个操作延后执行，以便给应用程序留出足够的时间，使其能把界面显示出来。

```
def create_variables(self):
    self.currencyFrom = tk.StringVar()
    self.currencyTo = tk.StringVar()
    self.amount = tk.StringVar()
    self.rates = {}
```

`tkinter.StringVar` 型变量里存有字符串，而这些字符串可以同控件相关联。关联起来之后，如果 `StringVar` 中的字符串变了，那么控件内容也会自动跟着 `StringVar` 一起改变，反之亦然。其实 `self.amount` 的类型本来可以声明成 `tkinter.IntVar`，但是 Tcl/Tk 在内部几乎都使用字符串，所以把它声明成 `tkinter.StringVar` 用起来通常会更方便些，即便本来要操作的就是数字，也还是应该将其声明成 `StringVar`。`rates` 变量是个 `dict`，其中每个条目的键都是货币名称，而值则是汇率。

```
Spinbox = ttk.Spinbox if hasattr(ttk, "Spinbox") else tk.Spinbox
```

`tkinter.ttk.Spinbox` 控件并未包含在 Python 3 的 Tkinter 中，但有可能会在 Python

3.4 版本加入。上面这行代码会判断 Python 里是否包含此控件，如果有，就用它，否则就使用无主题版本的 `tk.Spinbox` 控件。由于这两个控件的接口并不相同，所以开发时要小心，我们只应该使用两者都支持的那些功能。

```
def create_widgets(self):
    self.currencyFromCombobox = ttk.Combobox(self,
                                              textvariable=self.currencyFrom)
    self.currencyToCombobox = ttk.Combobox(self,
                                              textvariable=self.currencyTo)
    self.amountSpinbox = Spinbox(self, textvariable=self.amount,
                                 from_=1.0, to=10e6, validate="all", format="%0.2f",
                                 width=8)
    self.amountSpinbox.config(validatecommand=(
        self.amountSpinbox.register(self.validate), "%P"))
    self.resultLabel = ttk.Label(self)
```

每个控件在建立的时候都要指明其 `parent` (或 `master`)，只有 `tk.Tk` 对象例外，该对象通常是个 `window` 或 `frame`，其他控件可以放在其中。上面这段代码创建了两个下拉列表框，并把每个列表框同各自的 `StringVar` 关联起来。

上面这段代码还创建了“数值调整框” (`spinbox`)，并将其与 `StringVar` 相关联，同时还设置了最小值与最大值。创建 `spinbox` 时的 `width` 参数表示控件宽度，单位是字符，而 `format` 参数则是格式化控件内容所用的字符串，它采用 Python 2 时代的旧格式 (`"%0.2f"` 就相当于 `str.format()` 方法所使用的格式化字符串 {:0.2f})。`validate` 参数设为 `"all"` 的意思是：无论用户是直接输入数字，还是通过“微调按钮” (`spin button`) 来改变数值，只要 `spinbox` 的值变了，就要进行验证。创建好 `spinbox` 之后，我们注册验证时要用到的 `callable`。Python 在执行这个 `callable` 的时候，会给它传入一个参数，该参数的格式是 `%P`，这是个 `Tcl/Tk` 的格式化字符串，而不是 Python 的格式化字符串。顺便告诉大家，如果没有明确设置 `spinbox` 的初始值，那么其值会自动设为最小值，也就是 `from_` 参数的值 (在本例中是 1.0)。

最后，把显示换算结果所用的标签控件创建出来。我们不给它设置初始文本。

```
def validate(self, number):
    return TkUtil.validate_spinbox_float(self.amountSpinbox, number)
```

上面这个方法就是我们向 `spinbox` 所注册的 `callable`，它负责验证控件内容。在验证的时候，`Tcl/Tk` 的 `"%P"` 格式化字符串表示 `spinbox` 里的文本。这样一来，只要 `spinbox` 的值变了，Python 就会调用上面这个方法，并把 `spinbox` 里的文本当成 `number` 参数传进去。我们把实际的验证工作交给通用的便捷函数来完成，此函数位于 `TkUtil` 模块中。

```
def validate_spinbox_float(spinbox, number=None):
    if number is None:
        number = spinbox.get()
    if number == "":
        return True
```

```

try:
    x = float(number)
    if float(spinbox.cget("from")) <= x <= float(spinbox.cget("to")):
        return True
except ValueError:
    pass
return False

```

调用者应该给上面这个函数传入 spinbox 及数值（数值可以是个字符串，也可以是 None）。如果 number 参数里没有值，那么函数就会自己从 spinbox 中获取文本。spinbox 中的文本若是空字符串，函数便返回 True，以此表示控件值“有效”(valid)。之所以要这样做，是为了使用户能够把 spinbox 里的值删掉，然后重新开始输入新值。如果 spinbox 中的文本不是空字符串，那么就将其转为浮点数，并判断它是否处在 spinbox 的取值范围内。

每个 Tkinter 控件都有 config() 方法，该方法可以接受一个或多个 key=value 形式的参数，用于设置控件“属性”(attribute)，另外还有 cget() 方法，可以用 key 参数为键，查出与之关联的“属性值”(attribute value)。这些控件也有 configure() 方法，但它只是 config() 的别名而已。

```

def create_layout(self):
    padWE = dict(sticky=(tk.W, tk.E), padx="0.5m", pady="0.5m")
    self.currencyFromCombobox.grid(row=0, column=0, **padWE)
    self.amountSpinbox.grid(row=0, column=1, **padWE)
    self.currencyToCombobox.grid(row=1, column=0, **padWE)
    self.resultLabel.grid(row=1, column=1, **padWE)
    self.grid(row=0, column=0, sticky=(tk.N, tk.S, tk.E, tk.W))
    self.columnconfigure(0, weight=2)
    self.columnconfigure(1, weight=1)
    self.master.columnconfigure(0, weight=1)
    self.master.rowconfigure(0, weight=1)
    self.master.minsize(150, 40)

```

由上面这个方法所排布的控件位置如图 7.3 所示。每个控件都放在网格中特定的单元格里，而且会在“东西方向”(West and East directions) 上保持“黏性”(sticky)，这就是说，当窗口宽度改变时，控件也会在水平方向上随之扩大或缩小，而当窗口高度改变时，控件却不会改变其高度。控件在 x 方向与 y 方向上都有 0.5 毫米的间距，意思就是每个控件周围都有 0.5 毫米空白。（“\*\*”的用法请参见 1.2 节。）

|                             |                      |
|-----------------------------|----------------------|
| (0, 0) currencyFromCombobox | (0, 1) amountSpinbox |
| (1, 0) currencyToCombobox   | (1, 1) resultLabel   |

图 7.3 Currency 应用程序主窗口内各控件的布局

所有控件都排布好之后，我们把窗口自身也排布到一张仅含一个单元格的网格中，并令其在所有方向（也就是北、南、东、西四个方向）上伸缩。然后配置每一列的权重，这些权重也叫做“缩放系数”(stretch factor)。本例把第 0 列的权重设为 2，把第 1 列的权重设为 1，

意思就是如果窗口横向拉伸，那么数值调整框与标签控件每变宽一个像素，下拉列表框的宽度就要增加两个像素。对于窗口本身所在的那个单元格，其列权重与行权重都要设成非 0 的值，以便窗口中的内容能够随窗口大小而调整其尺寸。最后，我们要给窗口设置一套合理的小尺寸，否则用户可能会把窗口缩小得几乎看不见了。

```
def create_bindings(self):
    self.currencyFromCombobox.bind("<<ComboboxSelected>>",
                                   self.calculate)
    self.currencyToCombobox.bind("<<ComboboxSelected>>",
                                 self.calculate)
    self.amountSpinbox.bind("<Return>", self.calculate)
    self.master.bind("<Escape>", lambda event: self.quit())
```

上述方法会把事件与相关操作绑定起来。在本例中，我们关注两种事件，第一种叫做“虚拟事件”(virtual events)，也就是由某些控件所产生的“自定义事件”(custom event)，第二种叫做“真实事件”(real events)，也就是用户界面中所发生的事情，例如敲击按键或调整窗口大小。虚拟事件的事件名称其左右两侧各有两个“尖括号”(angle bracket)，而真实事件的事件名称其左右两侧各有一个尖括号。

当下拉列表框的值有变化时，它会给事件循环的事件队列里添加<<Combobox Selected>>虚拟事件。对于本例的这两个下拉列表框来说，我们都把此事件绑定到self.calculate()方法，以便重新计算货币换算后的结果。而对于数值调整框来说，只有当用户按下Enter或Return键时，我们才执行self.calculate()。用户按下Esc键之后，我们就调用继承下来的tkinter.ttk.Frame.quit()方法，以终止应用程序。

```
def calculate(self, event=None):
    fromCurrency = self.currencyFrom.get()
    toCurrency = self.currencyTo.get()
    amount = self.amount.get()
    if fromCurrency and toCurrency and amount:
        amount = ((self.rates[fromCurrency] / self.rates[toCurrency]) *
                  float(amount))
        self.resultLabel.config(text="{:.2f}".format(amount))
```

上面这个方法会获取两种货币名称以及待转换的货币总量，然后执行转换。最后，它会把转换好的结果设置成标签控件的文本，并用半角逗号做“千位分隔符”(thousands separator)，同时还会显示出小数点后的两位数字。

```
def get_rates(self):
    try:
        self.rates = Rates.get()
        self.populate_comboboxes()
    except urllib.error.URLError as err:
        messagebox.showerror("Currency \u2014 Error", str(err),
                            parent=self)
        self.quit()
```

Currency程序是通过“计时器”(timer)来调用上面这个方法的，这样做是为了给窗口留

出足够的时间，使其能够把自己绘制出来。`get_rates()`方法试着获取包含汇率数据的字典（字典里每个条目的键都是货币名称，值都是“转换系数”(conversion factor)），并用该字典来生成下拉列表框里的选项。如果无法获取汇率数据，那么就弹出一个写有错误信息的“消息框”(message box)，用户关闭该消息框（比方说按下对话框里的OK按钮）之后，应用程序就终止了。

`tkinter.messagebox.showerror()`函数的前两个参数表示窗口标题文本及信息文本，另外还可以指定`parent`参数，如果指定了这个参数，那么消息框就会出现在`parent`的正中间。由于Python3的源文件使用UTF-8编码，所以我们也可以把em dash符号(—)以字面量的形式直接写在代码里<sup>⊖</sup>，但由于印刷本书代码所用的“等宽字型”(monospaced font)中没有这个字符，所以笔者只好用Unicode转义符来表示它了。

```
def populate_comboboxes(self):
    currencies = sorted(self.rates.keys())
    for combobox in (self.currencyFromCombobox,
                     self.currencyToCombobox):
        combobox.state(("readonly",))
        combobox.config(values=currencies)
    TkUtil.set_combobox_item(self.currencyFromCombobox, "USD", True)
    TkUtil.set_combobox_item(self.currencyToCombobox, "GBP", True)
    self.calculate()
```

上述方法用货币名称来生成下拉列表框中的各选项，这些选项按字母表顺序出现。我们把两个下拉列表框都设为“只读的”(read-only)。接下来，把顶部那个下拉列表框中的货币设为美元，把底部的设为英镑。最后，调用`self.calculate()`方法，把首次换算的结果显示在窗口右下角的标签里。

每个带主题的Tkinter控件都有`state()`方法，可以把一个或多个状态设置给该控件，同时还有`instate()`方法，用于判断控件是否处于特定的状态。最常用的几种状态是“disabled”、“readonly”和“selected”。

```
def set_combobox_item(combobox, text, fuzzy=False):
    for index, value in enumerate(combobox.cget("values")):
        if (fuzzy and text in value) or (value == text):
            combobox.current(index)
            return
    combobox.current(0 if len(combobox.cget("values")) else -1)
```

上面这个通用函数写在TkUtil模块里。下拉列表框中如果有某个选项的文本与`text`参数相同，那么此函数就会将下拉列表框的当前值设置成该选项。如果`fuzzy`参数是`True`，那么不一定要求完全相同，只要条目中的文字包含`text`即可。

这个简单而实用的currency程序需要大约200行代码（不包括标准库里的模块及本书范例代码里附带的TkUtil模块）。对于这种小的GUI工具来说，其代码量一般都会远远超过等效的命令行版本。但当应用程序变得越来越复杂、功能变得越来越多时，代码量之间的差

---

<sup>⊖</sup> 也就是写成“Currency – Error”。——译者注

异很快就会消失。

## 7.2.2 创建应用程序中的对话框

对于小工具、媒体播放器及某些游戏来说，我们可以将其做成独立的对话框式应用程序，这样既简单又方便。但对于更为复杂的应用程序来说，通常应该有主窗口及若干辅助的对话框。本节将会讲解如何创建模态对话框与非模态对话框。

对于控件、布局及事件绑定来说，模态对话框与非模态对话框之间没有区别。但模态对话框一般会把用户输入的内容赋给相关变量，而非模态对话框则通常会调用应用程序的方法或修改应用程序的数据，以便响应用户操作。此外，模态对话框显示出来以后，就会阻塞用户的其他操作，而非模态对话框则不会，我们在编程时必须注意这一重要区别。

### 1. 创建模态对话框

这里讲解 Gravitate 游戏程序的 Preferences 对话框。其代码位于 `gravitate/Preferences.py` 中，对话框样貌如图 7.4 所示。

在 Linux 及 Windows 系统中，当用户点击 Gravitate 游戏的“File”菜单下的“Preferences”菜单项时，程序就会执行 `Main.Window.preferences()` 方法，而该方法将会弹出上面这个模态的 Preferences 对话框。但在 OS X 系统中，依照惯例，用户必须点击应用程序菜单中的“Preferences”选项或按下“⌘”键，才能弹出此对话框。（不巧的是，Tkinter 并不会自动处理这两种不同的操作方式，所以我们必须手工来处理。7.3.2 节将会讲到这个问题。）

```
def preferences(self):
    Preferences.Window(self, self.board)
    self.master.focus()
```

上面这个方法位于主窗口类中，它会把 Preference 对话框显示出来。这是个 smart 对话框，所以我们不用像 dumb 对话框那样，先给它传入一些值，等用户点击 OK 按钮之后再去更新应用程序的状态，而是可以直接把应用程序里的对象传给它。在本例中，此对象指的就是 `self.board`，其类型为 `Board`，这是个用于显示二维图像的 `tkinter.Canvas` 子类。

`preferences()` 方法会新建 Preferences 对话框窗口。调用该方法之后，对话框就会显示出来，并阻塞其他操作（因为该对话框是模态的），直到用户点击 OK 或 Cancel 按钮。由于对话框本身已经足够“智能”（smart），可以在用户点击 OK 按钮时更新 `Board` 对象，所以我们不用再做任何处理，只需在对话框关闭之后把键盘焦点设置到主窗口即可。

Tkinter 自带的 `tkinter.simpledialog` 模块中提供了几个基类，可供开发者据此来创

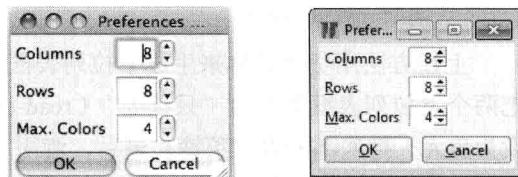


图 7.4 Gravitate 游戏程序的模态 Preferences 对话框在 OS X 及 Windows 系统中的样貌

建自定义的对话框，另外也提供了一些现成的便捷函数，比如 `tkinter.simpledialog.askfloat()` 等，它们可以弹出对话框，使用户能够通过对话框来输入单个数值。这些现成的对话框提供了某些内置的“挂钩”(hook)，开发者只要继承这些类，就可以轻松定制自己想要的控件。但笔者在编写本书时发现，这些类和函数很久没有更新了，而且也没有使用带主题的控件。于是，本书的范例代码中提供了 `TkUtil/Dialog.py` 模块，该模块里有个基类，可用来编写带主题的自定义对话框，其使用方式与 `tkinter.simpledialog.Dialog` 类相似，而且模块里也提供了一些便捷函数，例如 `TkUtil.Dialog.get_float()`。

本书所有对话框都是用 `TkUtil` 模块编写的，而没有使用 `tkinter.simpledialog`。这样做是为了使用带主题的控件，从而令 Tkinter 对话框在 OS X 及 Windows 系统里看起来和原生应用程序一样。

```
class Window(TkUtil.Dialog.Dialog):
    def __init__(self, master, board):
        self.board = board
        super().__init__(master, "Preferences \u2014 {}".format(APPNAME),
                        TkUtil.Dialog.OK_BUTTON|TkUtil.Dialog.CANCEL_BUTTON)
```

上面这个对话框类的 `__init__()` 方法接受两个参数，`master` 参数表示父对象，`board` 参数是个 `Board` 实例。该实例可以为对话框中的控件提供初始值，而当用户点击对话框的 OK 按钮时，对话框便会把用户所选定的控件值赋给该实例的相关属性，然后摧毁自己。`APPNAME` 常量（该常量没有列出来）的值是字符串 "Gravitate"。

凡是继承自 `TkUtil.Dialog.Dialog` 的类都必须提供 `body()` 方法，该方法用于创建对话框中除了按钮之外的控件，而按钮则由基类负责创建。子类还应提供 `apply()` 方法，用户点击对话框中的“确认”(accept) 按钮时，程序会调用该方法，至于确认按钮到底是 OK 按钮还是 Yes 按钮，这要由开发者来定。子类也可以有 `initialize()` 方法及 `validate()` 方法，但本例并不需要它们。

```
def body(self, master):
    self.create_variables()
    self.create_widgets(master)
    self.create_layout()
    self.create_bindings()
    return self.frame, self.columnsSpinbox
```

上面这个方法必须创建对话框所要使用的变量，排布对话框里的控件，并设置好事件绑定（按钮及按钮的事件绑定除外）。它必须把包含其他所有控件的那个大控件（通常是个 `frame`）返回给调用者，此外，还可以返回第二个值，用以表示对话框刚显示出来的时候具备键盘焦点的那个控件。在本例中，我们把其他控件都放在 `frame` 里，并将这个 `frame` 返回，同时把初始的键盘焦点设置到第一个 `spinbox` 控件上面。

```
def create_variables(self):
    self.columns = tk.StringVar()
```

```

    self.columns.set(self.board.columns)
    self.rows = tk.StringVar()
    self.rows.set(self.board.rows)
    self.maxColors = tk.StringVar()
    self.maxColors.set(self.board.maxColors)

```

这个对话框很简单，只用到了“标签”(label)控件和“数值调整框”(spinbox)控件。对于每一个 spinbox 来说，我们都创建与之关联的 `tkinter.StringVar`，并根据传进来的 Board 实例，为 `StringVar` 设置初始值。你可能觉得应该使用 `tkinter.IntVar`，但 Tcl/Tk 在内部其实只会使用字符串，所以我们还是把它声明成 `StringVar` 比较好。

```

def create_widgets(self, master):
    self.frame = ttk.Frame(master)
    self.columnsLabel = TkUtil.Label(self.frame, text="Columns",
        underline=2)
    self.columnsSpinbox = Spinbox(self.frame,
        textvariable=self.columns, from_=Board.MIN_COLUMNS,
        to=Board.MAX_COLUMNS, width=3, justify=tk.RIGHT,
        validate="all")
    self.columnsSpinbox.config(validatecommand=(
        self.columnsSpinbox.register(self.validate_int),
        "columnsSpinbox", "%P"))
    ...

```

上述方法用于创建对话框中的控件。我们首先创建外围的 frame，该方法在最后要把这个 frame 当成其他所有控件的父控件（也就是“容器控件”(containing widget)）返回给调用者。这个 frame 的父控件必须是对话框经由 `master` 参数传给该方法的那个控件，而对话框里的其他控件则必须把 frame 当作其直接或间接的父控件。

书中只列出了调整游戏中的方块列数所用的那两个控件，而调整行数及最大颜色数的那些控件其结构则与此相同。每次我们都是先创建好 label 和 spinbox，然后将 spinbox 与对应的 `StringVar` 关联起来。`width` 属性表示 spinbox 应该占据多少个字符的宽度。

顺便说一下，我们在创建 label 时用的是 `TkUtil.Label`，而不是 `tkinter.ttk.Label`，因为这样就无须每次都编写 `underline=-1 if TkUtil.mac() else 0` 语句了。

```

class Label(ttk.Label):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        if mac():
            self.config(underline=-1)

```

上面这个类的代码非常少，开发者可以为这种控件里的字符加下划线，以表示“键盘快捷键”(keyboard shortcut)，同时又无须担心代码是否运行在 OS X 系统上，因为该类在 OS X 系统上会自动把 `underline` 参数设为 -1，从而禁用下划线功能。`TkUtil/__init__.py` 模块里还有 `Button`、`Checkbutton` 及 `Radiobutton` 类，它们也具有和此处相同的 `__init__()` 方法。

```
def validate_int(self, spinboxName, number):
    return TkUtil.validate_spinbox_int(getattr(self, spinboxName),
                                       number)
```

我们在前面讲过如何验证 spinbox 里的数值，也讲过 TkUtil.validate\_spinbox\_float() 函数的用法。与早前的 validate() 方法相比，上面这个 validate\_int() 除了方法名和所要验证的数据类型不同之外（原来验证的是浮点数，这次验证的是整数），还有个区别：早前我们是把待验证的 spinbox 控件直接传给 TkUtil.validate\_spinbox\_float() 函数，而这次则是先通过 spinboxName 找到相关的 spinbox 控件，然后再把它传给 TkUtil.validate\_spinbox\_int()。

注册过的“验证函数”（validation function）在执行时会收到两个字符串，第一个是待验证的 spinbox 控件的名称，第二个是经 Tcl/Tk 格式化过的字符串。当控件内容需要验证时，程序就会把相关的两项数据交由 Tcl/Tk 来解析，并把解析后的值当成参数传给验证函数。Tcl/Tk 并不会修改 spinbox 控件的名称，但是会把注册时所指定的 "%P" 替换成 spinbox 控件里的字符串值。TkUtil.validate\_spinbox\_int() 函数需要用 spinbox 控件及字符串值做其参数。于是，我们把对话框本身（用 self 表示）及 attribute 名称（用 spinboxName 来表示）传给 Python 内置的 getattr() 函数，以此查出具备该名称的 spinbox 控件，然后把指向此控件的引用与待验证的 number 一起传给 TkUtil.validate\_spinbox\_int()。

```
def create_layout(self):
    padW = dict(sticky=tk.W, padx=PAD, pady=PAD)
    padWE = dict(sticky=(tk.W, tk.E), padx=PAD, pady=PAD)
    self.columnsLabel.grid(row=0, column=0, **padW)
    self.columnsSpinbox.grid(row=0, column=1, **padWE)
    self.rowsLabel.grid(row=1, column=0, **padW)
    self.rowsSpinbox.grid(row=1, column=1, **padWE)
    self.maxColorsLabel.grid(row=2, column=0, **padW)
    self.maxColorsSpinbox.grid(row=2, column=1, **padWE)
```

上面这个方法会按照图 7.5 中的版式来排布控件。这套布局非常简单，所有的 label 控件都向左对齐（sticky=tk.W，也就是向西对齐），而所有的 spinbox 控件都会填满水平方向剩下来的那些空间，此外，每个控件周围都有 0.75 毫米的空白

（这个常量保存在 PAD 里，它没有列在上述代码中）。（“\*\*”的用法请参阅 1.2 节“序列与映射的解包操作”。）

```
def create_bindings(self):
    if not TkUtil.mac():
        self.bind("<Alt-l>", lambda *args: self.columnsSpinbox.focus())
        self.bind("<Alt-r>", lambda *args: self.rowsSpinbox.focus())
        self.bind("<Alt-m>",
                 lambda *args: self.maxColorsSpinbox.focus())
```

|                       |                         |
|-----------------------|-------------------------|
| (0, 0) columnsLabel   | (0, 1) columnsSpinbox   |
| (1, 0) rowsLabel      | (1, 1) rowsSpinbox      |
| (2, 0) maxColorsLabel | (2, 1) maxColorsSpinbox |

图 7.5 Gravitate 游戏的 Preferences 对话框  
主体部分各控件的布局

在非 OS X 操作系统中，我们想令用户能够通过键盘快捷键来切换 spinbox 并点击按钮。比方说，用户按下 Alt+R 组合键之后，键盘焦点就会切换到调整方块行数的那个 spinbox 中。按钮的快捷键不用在上面这个方法里设置，因为它们由基类负责。

```
def apply(self):
    columns = int(self.columns.get())
    rows = int(self.rows.get())
    maxColors = int(self.maxColors.get())
    newGame = (columns != self.board.columns or
               rows != self.board.rows or
               maxColors != self.board.maxColors)
    if newGame:
        self.board.columns = columns
        self.board.rows = rows
        self.board.maxColors = maxColors
        self.board.new_game()
```

上述方法只有当用户点击了对话框中的“确认”按钮（按钮文字可能是“OK”或“Yes”）之后才会执行。我们获取各 StringVar 的值，并将其转换为 int（这个转换操作应该总是能顺利执行）。然后把这些 int 赋给 Board 实例中的相关属性。只要有值发生变化，我们就重新开始一盘游戏，以便使用户能看到自己所做的设定。

在比较复杂的大型应用程序里，开发者通常需要操作很多步才能找到自己想测试的对话框，这中间可能要点击几个菜单项，还有可能会从对话框里再弹出对话框。为了使测试过程方便一些，通常应该在包含窗口类的模块末尾加上 `if __name__ == "__main__":` 语句，并把一段能够弹出受测对话框的代码置于其中。比方说，在 `gravitate/Preferences.py` 模块中，这段代码就是：

```
def close(event):
    application.quit()
application = tk.Tk()
scoreText = tk.StringVar()
board = Board.Board(application, print, scoreText)
window = Window(application, board)
application.bind("<Escape>", close)
board.bind("<Escape>", close)
application.mainloop()
print(board.columns, board.rows, board.maxColors)
```

首先我们创建一个 `close()` 函数，该函数能够终止应用程序。然后创建 `tk.Tk` 对象，此对象将是应用程序里所有控件最终的父控件，一般情况下，我们应该把它隐藏起来，但在测试 Preferences 对话框时，我们会把它显示出来。另外，我们还会把 Esc 键同 `close()` 函数绑定起来，这样的话，用户就能按 Esc 键关闭窗口了。

如果按常规方式启动游戏，那么主窗口会在弹出 Preferences 对话框时，把 Board 实例传过去，但此处我们要单独测试对话框，所以必须自己创建 Board 实例。

创建好 Board 实例之后，我们创建对话框，该对话框会阻塞游戏中的其他操作，直到

用户点击OK或Cancel按钮为止。实际上，只有当事件循环开始之后，对话框才会真正显示出来。对话框一旦关闭，我们就把Board实例中可供对话框修改的那些属性打印出来。关闭对话框时，如果用户点击的是OK按钮，那么打印出来的就应该是修改后的值；但若点击的是Cancel按钮，则会打印对话框中的初始值。

## 2. 创建非模态对话框

这里将要讲解Gravitate游戏中的“Help对话框”，它是个非模态的对话框，如图7.6所示。

早前我们说过，在创建控件、排布控件与绑定事件这三件事上，模态对话框与非模态对话框没有区别。这两种对话框之所以不同，是因为模态对话框会阻塞用户操作，而非模态对话框则不会，此类对话框的调用者（比如主窗口）可以继续执行事件循环并接受用户操作。我们首先来看弹出对话框所用的代码，然后再看对话框本身的代码。

```
def help(self, event=None):
    if self.helpDialog is None:
        self.helpDialog = Help.Window(self)
    else:
        self.helpDialog.deiconify()
```

上面就是Main.Window.help()方法的代码。Main.Window中有个名叫self.helpDialog的实例变量，`__init__()`方法会把它设为None（该方法的代码没有列出来）。用户首次打开Help对话框时，我们要创建这个对话框，并把主窗口当成父控件传给它。“创建对话框”这一行为会使对话框弹出来，并显示在主窗口上方，而这个对话框是非模态的，所以主窗口的事件循环还会继续，这使得用户既可以操作对话框，又可以操作主窗口。

由于我们已经持有指向Help对话框的引用，所以如果用户以后还要打开Help对话框，那么只需调用tkinter.Toplevel.deiconify()将其重新显示出来就好。这样做之所以可行，是因为用户关闭对话框之后，该对话框只是隐藏了起来，但它并没有把自己摧毁。而另外一种做法则是：每次使用对话框时都重新创建并将其显示出来，等用户关闭对话框之后就将其摧毁。本例所采用的方式与之不同：我们只在首次使用对话框时创建它，然后将其显示来，用户关闭对话框之后，我们只将其隐藏起来，但却并不把它摧毁，以后再用到这个对话框时，只需重新将其显示出来就好。这样做要比刚才说到的那种方法快。另外，对话框隐藏了之后，其状态会保留下，等下次显示出来的时候，这些状态将会保持不变。

```
class Window(tk.Toplevel):
    def __init__(self, master):
        super().__init__(master)
```

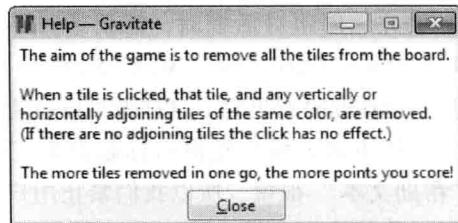


图7.6 Gravitate游戏的非模态Help对话框在Windows操作系统中的样貌

```

    self.withdraw()
    self.title("Help \u2014 {}".format(APPNAME))
    self.create_ui()
    self.reposition()
    self.resizable(False, False)
    self.deiconify()
    if self.winfo_viewable():
        self.transient(master)
    self.wait_visibility()

```

非模态对话框通常继承自 `tkinter.ttk.Frame` 或 `tkinter.Toplevel`, 本例也是如此。创建对话框时, 需要通过 `__init__()` 方法的 `master` 参数指定父控件。调用 `tkinter.Toplevel.withdraw()` 方法之后, 对话框窗口就会立刻隐藏起来, 从而使用户不会在此时看见它, 这样做可以避免创建窗口时发生闪烁现象。

接下来, 我们把窗口标题设为 “Help—Gravitate”, 然后创建对话框里的控件。由于 “帮助文本” 很短, 所以我们禁止用户调整对话框的大小<sup>⊖</sup>, 而是令 Tkinter 自行确定窗口尺寸, 使其恰好能够容纳帮助文本与 Close 按钮。若是帮助文本比较多, 则可用 `tkinter.Text` 的子类设计一种控件把文本置于其中, 并为控件加上滚动条, 同时允许用户调整对话框尺寸。

把所有控件都创建完并将其排列好之后, 我们调用 `tkinter.Toplevel.deiconify()` 来显示对话框窗口。如果窗口是 “可视的” (`viewable`, 意思就是说, 系统的 “窗口管理器” (`window manager`) 会把这个窗口显示出来), 那么我们就通知 Tkinter: 该窗口对于它的父控件来说是 “暂时的” (`transient`)。这等于是提示窗口管理系统: 这个 “暂时的窗口” (`transient window`) 也许稍后就不见了。等该窗口隐藏或摧毁之后, 父控件里原来遮住的部分内容就会显示出来, 因而需要重新绘制, 而窗口管理系统则可根据这项提示对重绘操作进行优化。

`__init__()` 方法最后调用的那个 `tkinter.Toplevel.wait_visibility()` 方法会阻塞程序, 直到窗口显示出来为止 (阻塞时间很短, 用户察觉不到)。默认情况下, `tkinter.Toplevel` 窗口都是非模态的, 但是只要在最后加两行语句, 就能将其变成模态窗口。这两行语句是 `self.grab_set()` 与 `self.wait_window(self)`。第一条语句会限制应用程序的焦点 (在 Tk/Tcl 的术语里, “焦点” 称为 “grab”), 将其锁定到本窗口, 这样它就变成模态窗口了。而第二条语句则会阻塞程序, 直到本窗口关闭为止。本书在讨论模态对话框时并没有提到过这两条命令, 是因为一般情况下我们都通过继承 `tkinter.simpledialog.Dialog` (在本书中, 继承的是 `TkUtil.Dialog`) 来创建模态对话框, 而 `tkinter.simpledialog.Dialog` 及 `TkUtil.Dialog` 都已经把这两条语句写好了。

用户现在就可以在这个 `Preferences` 对话框里面进行操作了, 同时还可以操作游戏的主窗口, 另外, 如果游戏里还弹出了其他非模态窗口, 那么用户也同样可以操作它们。

---

⊖ 原文是 “set the dialog to be nonresizable” (把对话框设为 “大小不可变”)。——译者注

```

def create_ui(self):
    self.titleLabel = ttk.Label(self, text=_TEXT, background="white")
    self.closeButton = TkUtil.Button(self, text="Close", underline=0)
    self.titleLabel.pack(anchor=tk.N, expand=True, fill=tk.BOTH,
                         padx=PAD, pady=PAD)
    self.closeButton.pack(anchor=tk.S)
    self.protocol("WM_DELETE_WINDOW", self.close)
    if not TkUtil.mac():
        self.bind("<Alt-c>", self.close)
    self.bind("<Escape>", self.close)
    self.bind("<Expose>", self.reposition)

```

Preferences 窗口的用户界面非常简单，只需用上面这个函数即可创建好。首先，我们创建标签控件，以便显示帮助文本（帮助文本写在 \_TEXT 常量里，该常量没有列在上面），然后创建 Close 按钮。我们使用 TkUtil.Button（该类继承自 tkinter.ttk.Button）来创建按钮，这样就能在 OS X 系统里正确处理下划线了。（早前我们讲过的那个 TkUtil.Label 类的编写方式几乎与此处的 TkUtil.Button 类完全相同。）

由于对话框窗口里只有两个控件，所以不妨使用最简单的布局管理器来排布它们。我们调用 helpLabel 的 pack() 方法，把标签控件安排在窗口顶部，并令其能够在水平方向和垂直方向上延伸，然后调用 closeButton 的 pack() 方法，将按钮控件安排在窗口底部。

如果当前操作系统不是 OS X，那么我们就给 Close 按钮设置 Alt+C 快捷键。另外，无论在何种操作系统中，我们都把 Esc 键同窗口关闭操作绑定起来。

用户关闭这个非模态的对话框窗口之后，我们并不将其摧毁，而是把它隐藏起来，这样等到下次需要弹出 Help 对话框时，就不用重新创建了，只需要把它显示出来就好。由于采用了这种方式，所以用户在打开 Help 对话框并将其关闭（也就是隐藏）之后，可能会移动游戏的主窗口，然后再度打开 Help 对话框。在这种情况下，我们完全可以把对话框显示在它一开始出现的那个位置上（或是用户上次移动到的那个位置上）。但由于对话框里的文本很少，所以笔者觉得还是每次显示时重新计算位置比较好。于是，我们把 <Expose> 事件（当窗口必须重新绘制自身内容时，就会发生该事件）与自编的 reposition() 方法绑定起来。

```

def reposition(self, event=None):
    if self.master is not None:
        self.geometry("{}+{}".format(self.master.winfo_rootx() + 50,
                                     self.master.winfo_rooty() + 50))

```

上面这个方法会把 Help 窗口移动到与父控件（也就是游戏的主窗口）相同的位置，但是会向右方及下方各偏移 50 像素。

从理论上说，我们并不需要在 \_\_init\_\_() 方法中明确调用 reposition() 方法，但实际上我们还是这样做了，因为这可以确保 Help 窗口在显示出来之前能够先摆放在正确的位置上。如果不这样做，那么窗口有可能在显示出来之后突然跳到别处。本例不会发生这

种情况，因为窗口在显示出来之前，其位置已经设定好了。

```
def close(self, event=None):
    self.withdraw()
```

如果用户通过 Esc 键、Alt+C 组合键、Close 按钮或带有“×”图样的关闭按钮关掉了 Help 对话框，那么程序就会调用上述方法。该方法只会把窗口隐藏起来，并不会摧毁它。调用 `tkinter.Toplevel.deiconify()` 方法可以重新将其显示出来。

### 7.3 用 Tkinter 创建主窗口式应用程序

本节要讲解如何创建 Gravitate 这个主窗口式的应用程序中最通用的那些方面。该程序的运行效果如图 7.7 所示，Gravitate 游戏的相关信息请参见标题为“Gravitate”的边栏。程序的用户界面里包含一些常见的标准元素，例如菜单栏、中心控件（central widget）、状态栏、对话框等。Tkinter 对菜单提供了内置的支持，但是窗口中心的控件和状态栏则必须由我们自行创建。很容易就能把 Gravitate 程序改编成其他的主窗口式应用程序：总架构无须修改，只把菜单栏、状态栏及窗口中间的控件改一下就行。

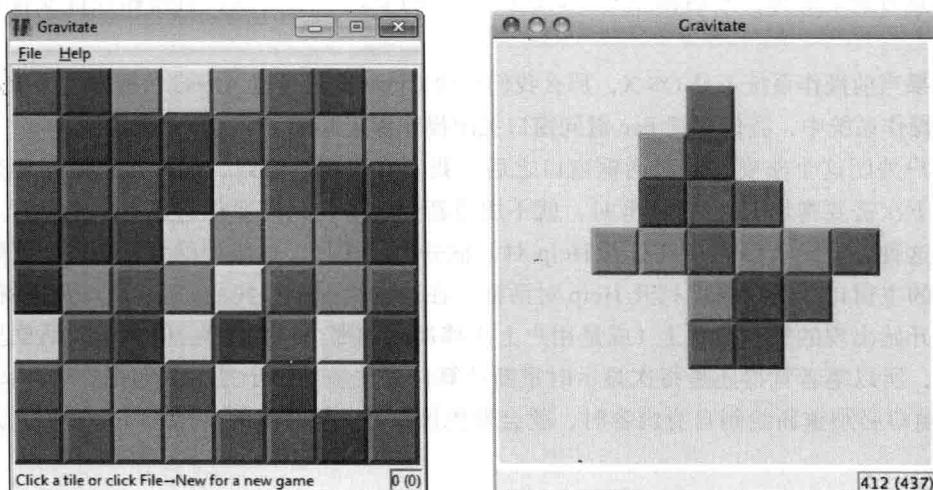


图 7.7 Gravitate 游戏程序在 Windows 及 OS X 系统中的运行截图

Gravitate 由 7 份 Python 文件及 9 个图标图像构成。应用程序的“可执行文件”是 `gravitate/gravitate.pyw`，其主窗口类写在 `gravitate/Main.py` 文件里。另有 3 份文件用于实现 3 种对话框：`gravitate/About.py` 用于实现 About 对话框，本书不打算讲解它；`gravitate/Help.py` 用于实现 Help 对话框，它在 7.2.2 节已经讲过了；`gravitate/Preferences.py` 用于实现 Preferences 对话框，它在 7.2.2 节讲过了。主窗口的中心区域由 Board 控件所占据，与该控件相对应的 Board 类写在 `gravitate/`

Board.py文件里，这个类继承自tkinter.Canvas（由于篇幅所限，本书就不讲解tkinter.Canvas了）。

### Gravitate



游戏目标是将所有方块消除。玩家点击某个方块后，垂直方向与水平方向上与之同色且邻近的方块都会消去。（如果这两个方向上没有同色且邻近的方块，那么这次点击无效。）一次点击所消去的方块数量越多，得分也就越高。

Gravitate游戏的逻辑和Tile Fall（方块下落）或Same Game（同色消除）类游戏相似。但关键区别是：在那两种游戏中，玩家消去了方块之后，其余方块会下落并左移，以填补空缺，而在Gravitate里，游戏画面的中心点会把四周的方块“吸”过来。

本书范例中包含三个版本的Gravitate游戏。第一个版本位于gravitate目录下，本节所要讲解的就是这一版。第二个版本放在gravitate2目录中，其游戏逻辑和前者相同，而且还带有可以隐藏的工具栏及功能更为丰富的Preferences对话框。这个对话框里的选项比第一个版本多，例如，玩家可以选择方块的形状，也可以选择缩放比例，以便将方块放大或缩小。另外，它还能在下次启动游戏程序时记住上次的最高分。除了用鼠标操作之外，这个版本还支持键盘，玩家可通过方向键选择方块，并按空格键试着将其移除。第三个版本是三维的，我们放在第8章的8.2节来讲。此外www.qtrac.eu/gravitate.html还有个在线版本。

```
def main():
    application = tk.Tk()
    application.withdraw()
    application.title(APPNAME)
    application.option_add("*tearOff", False)
    TkUtil.set_application_icons(application, os.path.join(
        os.path.dirname(os.path.realpath(__file__)), "images"))
    window = Main.Window(application)
    application.protocol("WM_DELETE_WINDOW", window.close)
    application.deiconify()
    application.mainloop()
```

上面就是gravitate/gravitate.pyw文件的main()函数。它首先创建顶级的tkinter.Tk对象，该对象一般都是隐藏的，本例也是如此：我们调用application.withdraw()，立刻将应用程序隐藏起来，这样的话，屏幕就不会在创建主窗口时闪烁了。在默认情况下，Tkinter程序的菜单带有tear-off风格<sup>⊖</sup>，它是一种类似Motif GUI的复古风格，而时下流行的GUI程序几乎都不使用此风格，所以我们将其关闭。接下来，用7.2.1节讲过的TkUtil.set\_application\_icons()函数来设置应用程序的图标，然后创建程序主窗口，并告诉Tkinter：如果用户点击了带有“×”图样的关闭按钮，那么就调用Main.

<sup>⊖</sup> tear-off是“可以沿虚线撕下的纸片”，在这种风格的菜单中，菜单标题与菜单里的各选项之间有一条虚线，如果用户点击了该虚线，那么整个菜单可能会以窗口形式展示出来。——译者注

`Window.close()` 方法。最后，我们把应用程序显示出来（或者说得更准确些，我们给事件循环里面添加一个事件，稍后处理这个事件的时候，程序就会显示出来了），并启动事件循环。此时程序会出现在屏幕上。

### 7.3.1 创建主窗口

从原则上讲，Tkinter 的主窗口与对话框没有区别。但在实际的应用程序中，主窗口一般都会有菜单栏及状态栏，而且通常会有工具栏，有时还会有“可停靠式窗口”（dock window）。这些窗口的中心通常会有一个控件，对于字处理程序来说，应该是个文本编辑器控件，对于电子表格程序来说，应该是个表格控件，而对于游戏、“仿真程序”（simulation）和“可视化程序”（visualization）来说，应该是个能够显示图形的控件。Gravitate 游戏的主窗口同时具备菜单栏、中心控件以及状态栏。

```
class Window(ttk.Frame):
    def __init__(self, master):
        super().__init__(master, padding=PAD)
        self.create_variables()
        self.create_images()
        self.create_ui()
```

Gravitate 游戏的 `Main.Window` 类继承自 `tkinter.ttk.Frame`，它把大部分工作都交给基类和三个辅助方法来完成。

```
def create_variables(self):
    self.images = {}
    self.statusText = tk.StringVar()
    self.scoreText = tk.StringVar()
    self.helpDialog = None
```

有的文本消息只需要暂时显示在状态栏中，这种消息存放在 `self.statusText` 里面；而当前分数及最高分数则需要一直显示，这些文本用 `self.scoreText` 来保存。`helpDialog` 的初始值是 `None`，我们在 7.2.2 节讲过这个 `Help` 对话框。

GUI 应用程序的菜单项旁边经常会有图标，而且工具栏里的按钮也需要图标。我们在编写 Gravitate 游戏时，把所有图标图像都放在 `gravitate/images` 子目录中，并且定义了一系列常量，以表示其名称，比方说，`NEW` 常量的值是字符串 "New"，它代表 New 菜单项旁边的那个图标。`Main.Window` 在创建的时候，会调用 `create_images()` 方法，把程序所需的图像都加载进来，并把它们作为值放在 `self.images` 字典里。通过 Tkinter 载入图像之后，我们必须持有指向这些图像的引用，否则系统就会把它们当成垃圾回收了（也就是说，这些图像会消失不见）。

```
def create_images(self):
    imagePath = os.path.join(os.path.dirname(
        os.path.realpath(__file__)), "images")
```

```

for name in (NEW, CLOSE, PREFERENCES, HELP, ABOUT):
    self.images[name] = tk.PhotoImage(
        file=os.path.join(imagePath, name + "_16x16.gif"))

```

由于我们打算用 $16 \times 16$ 像素的图像来表示菜单项旁边的图标，因此会遍历每个常量(NEW、CLOSE等)，并把与该操作相对应的图像加载进来。

Python内置的\_\_file\_\_常量含有表示当前文件的文件名及其路径。我们用os.path.realpath()去掉路径中的“..”及“符号链接”(symbolic link)，以获取绝对路径，然后取出表示目录的那一部分(也就是把后面的文件名丢掉)，将其与“image”字符串拼接，这样就得到了应用程序里images子目录的绝对路径。

```

def create_ui(self):
    self.create_board()
    self.create_menuBar()
    self.create_statusbar()
    self.create_bindings()
    self.master.resizable(False, False)

```

笔者大幅度地重构了游戏代码，使得上面这个方法把它的任务交给四个辅助方法来完成。等用户界面创建好之后，它会调用self.master.resizable(False, False)，禁止玩家调整游戏窗口的尺寸。由于方块的大小是固定的，所以调整窗口尺寸没什么意义。(Gravitate 2程序也不允许玩家调整窗口尺寸，但是它允许玩家修改方块的大小，修改之后，游戏窗口就会按照新的方块大小自动调整其尺寸了。)

```

def create_board(self):
    self.board = Board.Board(self.master, self.set_status_text,
                           self.scoreText)
    self.board.update_score()
    self.board.pack(fill=tk.BOTH, expand=True)

```

上面这个方法会创建Board实例(Board类是tkinter.Canvas的子类)并把self.set\_status\_text()方法传给该实例，这样一来，就可以把临时消息显示在主窗口的状态栏中了，另外，创建实例时也会传入self.scoreText变量，这使得Board控件可以更新玩家当前的分数及最高分数。

创建完board之后，我们调用update\_score()方法，使状态栏中持续显示的计分器变成“0(0)”。然后调用pack()方法，把board排布在主窗口中，并令其在水平方向和垂直方向上延展。

```

def create_bindings(self):
    modifier = TkUtil.key_modifier()
    self.master.bind("<{}-n>".format(modifier), self.board.new_game)
    self.master.bind("<{}-q>".format(modifier), self.close)
    self.master.bind("<F1>", self.help)

```

上面这个函数绑定了三项键盘快捷键：按Ctrl+N(或⌘N)可以开始新游戏，按Ctrl+Q(或⌘Q)可以退出游戏，按F1可以“弹出”(如果窗口已隐藏，那么就是“显示”)非模态

的 Help 窗口。TkUtil.key\_modifier() 方法会根据当前的操作系统返回“快捷键修饰符”(shortcut modifier) 的名称 ("Control" 或 "Command")。

### 7.3.2 创建菜单

在 Linux 及 Windows 系统中，Tkinter 会按照传统方式把菜单显示在窗口的标题栏下方。但在 OS X 系统中，Tkinter 则把菜单“集成”(integrate) 到屏幕顶端那个菜单里面。稍后大家就会看到，这项集成工作需要我们手工完成。

“菜单”(menu) 与“子菜单”(submenu) 都是 tkinter.Menu 的实例。其中必须有一个 Menu 实例用作顶级窗口的菜单栏(也就是主窗口的菜单栏)，其余所有菜单都是这个菜单栏的子菜单。

```
def create_menubar(self):
    self.menubar = tk.Menu(self.master)
    self.master.config(menu=self.menubar)
    self.create_file_menu()
    self.create_help_menu()
```

上面这个方法先创建了空菜单，并将其设为主窗口的子控件，然后把主窗口的 menu 属性(该属性代表菜单栏) 设置成这个空菜单(也就是 self.menubar)。接下来开始在空白的菜单栏中添加子菜单。Gravitate 游戏有两个子菜单，我们将在稍后讲解。

#### 1. 创建 File 菜单

大部分主窗口式应用程序的 file 菜单下面都有“创建新文档”、“打开已有文档”、“保存当前文档”及“退出应用程序”等菜单项。但对于游戏来说，很多菜单项都用不到，Gravitate 游戏的菜单项很少，如图 7.8 所示。

```
def create_file_menu(self):
    modifier = TkUtil.menu_modifier()
    fileMenu = tk.Menu(self.menubar, name="apple")
    fileMenu.add_command(label=NEW, underline=0,
                         command=self.board.new_game, compound=tk.LEFT,
                         image=self.images[NEW], accelerator=modifier + "+N")
    if TkUtil.mac():
        self.master.createcommand("exit", self.close)
        self.master.createcommand("::tk::mac::ShowPreferences",
                                 self.preferences)
    else:
        fileMenu.add_separator()
        fileMenu.add_command(label=PREFERENCES + ELLIPSIS, underline=0,
                             command=self.preferences,
                             image=self.images[PREFERENCES], compound=tk.LEFT)
        fileMenu.add_separator()
        fileMenu.add_command(label="Quit", underline=0,
```

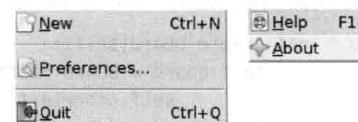


图 7.8 Gravitate 游戏的菜单在 Linux 操作系统中的样貌

```

        command=self.close, compound=tk.LEFT,
        image=self.images[CLOSE],
        accelerator=modifier + "+Q")
    self.menuBar.add_cascade(label="File", underline=0,
                            menu=fileMenu)

```

上面这个方法用于创建 file 菜单。常量名的所有字母均大写，除非另有说明，否则常量中的字符串就与常量名称相同，例如，NEW 常量中的字符串就是 "New"。

`create_file_menu()` 方法首先获取键盘快捷键的“修饰符”(modifier，在 OS X 系统中是  $\text{⌘}$  键，在 Linux 与 Windows 系统中是 Ctrl 键)。然后创建 file 菜单，并将其设为窗口菜单栏的子菜单。`file` 菜单的 `name` 属性是 "apple"，这就相当于告诉 Tkinter：在 OS X 系统中，这个菜单应该集成到应用程序的“应用程序菜单”(application menu) 里面(实际效果就相当于把该菜单当成应用程序的 application menu)，而其他操作系统则会忽略此属性。

`tkinter.Menu.add_command()`、`tkinter.Menu.add_checkbutton()` 及 `tkinter.Menu.add_radiobutton()` 这三个方法都可以添加菜单项，但我们在 Gravitate 程序中只使用第一个方法。`tkinter.add_separator()` 方法用于添加“分隔线”(separator)。`underline` 属性在 OS X 操作系统中不起作用，而在 Windows 操作系统中，只有当开发者明确指定要把下划线显示出来或是用户按着 Alt 键不放时，它才会显示出来。我们用 `fileMenu.add_command()` 方法来添加 file 菜单中的每个菜单项，调用该方法时，把菜单项的文本传给 `label` 参数，把下划线的位置传给 `underline` 参数，把用户点击菜单项时所需执行的命令传给 `command` 参数，并把菜单项所对应的图标传给 `image` 参数。菜单项会根据 `compound` 属性来决定应该如何处理其图标与文本，我们所采用的值是 `tk.LEFT`，它的意思是应该把图标与文本都显示出来，并将图标置于文本左侧。添加菜单项的时候，我们也给它设定了快捷键，比方说，在 Linux 及 Windows 操作系统中，用户可通过 `Ctrl+N` 来选择 File 菜单下的 New 菜单项，而在 OS X 系统中，则可以通过  $\text{⌘}N$  来做。

在 OS X 操作系统中，当前应用程序的 Preferences 与 Quit 菜单项会显示在“应用程序菜单”里面，这个菜单位于 apple 菜单右侧，而应用程序的 file 菜单又位于“应用程序菜单”右侧。为了把游戏程序自身的 Preferences 与 Quit 菜单项同 OS X 系统相集成，我们通过 `tkinter.Tk.createcommand()` 方法将 Tcl/Tk 的 `::tk::mac::ShowPreferences` 与 `exit` 命令绑定到 Gravitate 程序的相关方法上面。对于其他操作系统来说，我们还是照常把 Preferences 及 Quit 当成普通的菜单项添加到 file 菜单里。

完全创建好 file 菜单之后，我们把它添加为菜单栏的 `cascade` 菜单(级联菜单，也就是“子菜单”(submenu))。

```

def menu_modifier():
    return "Command" if mac() else "Ctrl"

```

上面这个小函数写在 `TkUtil/__init__.py` 里面，游戏程序在设置菜单项里的文本时会用到它。OS X 操作系统会对 "Command" 做特殊处理，把它显示成  $\text{⌘}$  符号。

## 2. 创建 Help 菜单

游戏程序的 help 菜单里只有两个菜单项：Help 和 About。但是 OS X 系统处理它们的方式却与 Linux 及 Windows 系统不同，所以我们必须自己编写代码来分别应对。

```
def create_help_menu(self):
    helpMenu = tk.Menu(self.menuBar, name="help")
    if TKUtil.mac():
        self.master.createcommand("tkAboutDialog", self.about)
        self.master.createcommand "::tk::mac::ShowHelp", self.help)
    else:
        helpMenu.add_command(label=HELP, underline=0,
                             command=self.help, image=self.images[HELP],
                             compound=tk.LEFT, accelerator="F1")
        helpMenu.add_command(label=ABOUT, underline=0,
                             command=self.about, image=self.images[ABOUT],
                             compound=tk.LEFT)
    self.menuBar.add_cascade(label=HELP, underline=0,
                            menu=helpMenu)
```

上面这段代码首先创建 help 菜单，并将其 name 设为 "help"。Linux 与 Windows 系统都会忽略这个 name，但是 OS X 系统若发现 name 的值是 "help"，则会将其集成到操作系统的 help 菜单里。如果这段代码在 OS X 系统中运行，那么就用 `Tk.createcommand()` 方法把 Tcl/Tk 的 `tkAboutDialog` 及 `::tk::mac::ShowHelp` 命令同 Gravitate 程序里的相关方法绑定起来。在其他操作系统中，我们依然按照传统方式创建 Help 及 About 菜单项。

### 7.3.3 创建带计分器的状态栏

Gravitate 游戏程序的状态栏比较常见：其左侧用于显示暂时性的消息，而右侧则会一直显示游戏分数。在图 7.7 中，左边那张截图里的状态栏就同时显示了计分器以及一条暂时出现的消息。

```
def create_statusbar(self):
    statusBar = ttk.Frame(self.master)
    statusLabel = ttk.Label(statusBar, textvariable=self.statusText)
    statusLabel.grid(column=0, row=0, sticky=(tk.W, tk.E))
    scoreLabel = ttk.Label(statusBar, textvariable=self.scoreText,
                          relief=tk.SUNKEN)
    scoreLabel.grid(column=1, row=0)
    statusBar.columnconfigure(0, weight=1)
    statusBar.pack(side=tk.BOTTOM, fill=tk.X)
    self.set_status_text("Click a tile or click File-New for a new "
                         "game")
```

为了创建状态栏，我们首先新建 frame。然后向其中添加 label，并把 `self.statusText`（类型为 `StringVar`）同这个 label 绑定起来。现在可以通过修改 `self.statusText` 变量来设置状态栏里的文本了，不过在本例中，我们是通过调用 `set_status_text()` 方法来设置的。接下来，我们继续向 frame 中添加另一个 label，用于显示玩家当前的分数及最高分

数，这个 label 与 self.scoreText 变量相关联，而且会一直显示在状态栏中。

我们通过 grid() 方法把两个 label 控件分别放在状态栏（状态栏本身是个 frame 控件）中的两个单元格里，并且令 statusLabel 控件（该控件用于显示暂时性的消息）在水平方向上尽量延伸。然后调用 statusBar 的 pack() 方法，将状态栏安排在主窗口底部，同时令其在水平方向上延伸，以占满整个窗口宽度。最后，调用自编的 set\_status\_text() 方法来设定游戏刚开始时所要显示的那一条暂时消息。

```
def set_status_text(self, text):
    self.statusText.set(text)
    self.master.after(SHOW_TIME, lambda: self.statusText.set("")))
```

上述方法会根据 text 参数（该参数可以是空字符串）来设置 self.statusText 控件的文本，并在 SHOW\_TIME 毫秒之后（本例是在 5000 毫秒（也就是 5 秒）之后）将其清除。

虽说本例的状态栏中只有标签控件，但你完全可以添加其他类型的控件，例如“下拉列表框”（combobox）、“数值调整框”（spinbox）、按钮等。

因本章篇幅所限，我们只能演示 Tkinter 的一些基本用法。自从 Python 采用 8.5 版的 Tcl/Tk（该版本首次开始使用“主题”（theming）之后，Tkinter 就变得更加流行了，因为用它所编写的程序在 OS X 及 Windows 系统中的界面看上去和原生应用程序一样。Tkinter 有许多非常实用且极为灵活的控件，其中最值得一提的就是 tkinter.Text 和 tkinter.Canvas，前者可以编辑并展示“带有样式及格式的文本”（styled and formatted text），而后者则用来绘制二维图形（Gravitate 及 Gravitate 2 都用到了它）。另外还有三个控件也很有用：tkinter.ttk.Treeview 能够把一系列条目以表格或树状图的形式显示出来，tkinter.ttk.Notebook 控件用于实现“分页”（tab，Gravitate 2 的 Preferences 对话框用到了这个控件），tkinter.ttk.Panedwindow 控件用于实现带“分隔条”（splitter）的窗口。

尽管 Tkinter 并没有像其他 GUI 工具包那样提供某些高级功能，但通过本章我们已经看到，创建能显示暂时消息与常驻计分器的状态栏控件还是非常简单的。Tkinter 对菜单的支持比本章所需的更为丰富，例如有子菜单、“二级子菜单”（subsubmenu，子菜单下的子菜单）等，另外还可以像“复选框”（checkbox）或“单选按钮”（radiobutton）那样，把若干菜单项归为一组，令用户在其中多选或单选，这叫做 checkable 菜单项。此外，创建“关联菜单”（context menu，又称上下文菜单、快捷菜单、环境菜单）也相当容易。

在本章没有讲到的那些流行功能中，开发者最需要用到的就是工具栏。这种控件创建起来很容易，但在实现一些功能时要多加注意，例如我们也许需要令其能在隐藏与可见状态之间切换，而且还要在窗口改变尺寸之后自动适应新的窗口大小。另外还有一个比较流行的功能也是很多应用程序都需要用到的，那就是“可停靠式窗口”。我们可以创建出能在隐藏与可见状态之间切换的可停靠式窗口，而且用户可以把这种窗口从一个“停靠区域”（dock area）拖放到另一个停靠区域，还可以把它从停靠区域里拖出来，使其变成可以自由浮动的窗口。

本书范例代码中还有两个应用程序，分别是 texteditor 与 texteditor2，但由于

篇幅所限，没有在本章讲解。图 7.9 演示了 `texteditor2` 的运行效果。这两个程序都实现了能够在隐藏与可见状态之间来回切换的工具栏，这种工具栏能够随窗口尺寸自动改变其大小。这两个程序里都有子菜单、“复选框式菜单项”（checkbox-style menu option）、“单选按钮式菜单项”（radiobutton-style menu option）、环境菜单以及“最近打开的文件列表”（recent files list），另外，它们也都实现了如图 7.10 所示的“可扩展式对话框”（extension dialog）<sup>①</sup>。通过这两个程序，你可以学到 `tkinter.Text` 控件的用法，还可以学会怎样同“剪贴板”（clipboard）相交互。`texteditor2` 程序还演示了如何实现可停靠式窗口（不过在 OS X 系统中，这些窗口的自由浮动功能有些问题）。

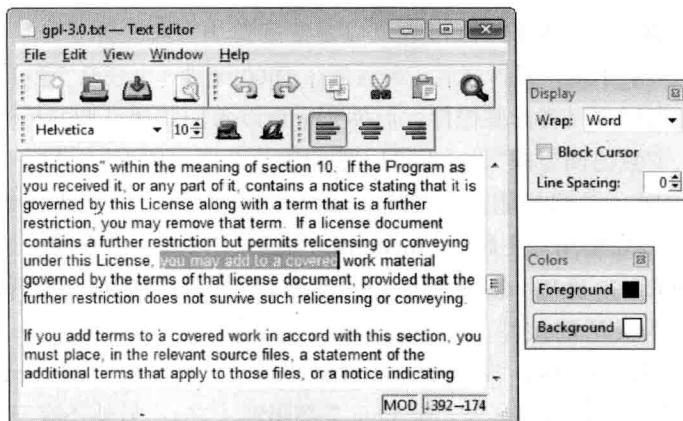


图 7.9 Text Editor 2 应用程序在 Windows 系统中的运行截图

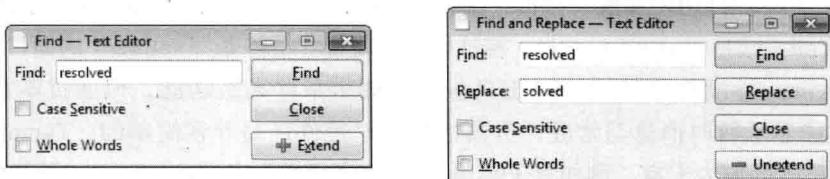
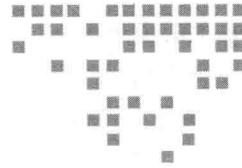


图 7.10 Text Editor 应用程序的“可扩展式对话框”在 Windows 系统中的样貌

若想把 Tkinter 的功能变得像其他 GUI 工具包一样丰富，那么肯定要投入大量精力来开发基本控件才行。但 Tkinter 的好处则在于它并不会对程序开发施加任何限制，只要我们精心设计好必备的基本控件（例如工具栏及可停靠式窗口），就可以在编写其他应用程序时复用这些控件。由于 Tkinter 非常稳定，而且又是 Python 标准库中的一部分，所以很适合用来开发易于部署的 GUI 程序。

<sup>①</sup> 所谓“可扩展”，指的是用户可以点击“Find”对话框（“查找”对话框）右下角的 Extend 按钮，将其扩展成“Find and Replace”对话框（“查找与替换”对话框），其后可以点击 Unextend 按钮，将其收缩成原来的“Find”对话框。——译者注



## 用 OpenGL 绘制 3D 图形

很多流行的应用程序，像是“设计工具”（design tool）、“数据可视化工具”（data visualization tool）、游戏等，都需要用到 3D 绘图功能，这其中当然也包括“屏幕保护程序”（screensaver）。上一章提到的那些 Python GUI 工具包全都支持 3D 绘图：有些本来就支持，另外一些可通过“附加元件”（add-on）来支持。这些工具包之所以能实现 3D 绘图，基本上都是因为包含了一套可以调用系统 OpenGL 库的接口。

另外也有许多支持 3D 绘图的 Python 程序包，它们能够提供一套高级接口，以简化 OpenGL 编程。比方说 Python Computer Graphics Kit ([cgkit.sourceforge.net](http://cgkit.sourceforge.net))、OpenCASCADE ([github.com/tenko/occmodel](https://github.com/tenko/occmodel)) 及 VPython ([www.vpython.org](http://www.vpython.org))。

除了使用高级接口之外，也可以采用更为直接的方式来调用 OpenGL。采用这种方式来实现 3D 绘图的两大开发包分别是 PyOpenGL ([pyopengl.sourceforge.net](http://pyopengl.sourceforge.net)) 与 pyglet ([www.pyglet.org](http://www.pyglet.org))。它们都“忠实地”（faithfully）包装了 OpenGL 库，使得开发者很容易就能把 C 语言范例代码改写成 Python（OpenGL 的原生开发语言就是 C 语言，而且很多教科书也是用 C 语言来描述 OpenGL 的）。这两个包都可以创建独立的 3D 程序，如果用 PyOpenGL 来开发，那么就要用到它所提供的包装器，这个包装器封装了 OpenGL 的 GLUT GUI 库；而若用 pyglet 开发，则需通过它自己的事件处理机制及顶层窗口来实现。

对于独立的 3D 程序来说，最好是用现成的 GUI 工具包搭配着 PyOpenGL 来做（Tkinter、PyQt、PySide、wxPython 以及其他一些 GUI 工具包都可以和 PyOpenGL 搭配着使用），但如果程序的 GUI 特别简单的话，那么用 pyglet 来开发就足够了。

OpenGL 有许多版本，而且有两种非常不同的使用方式。传统方式又称“立即模式”（direct mode），它适用于所有版本。这种方式会直接调用 OpenGL 函数，而函数则会立即执

行。另一种比较新的方式是从 2.0 版本开始支持的，它用传统方式把“场景”（scene）设定好，然后用“OpenGL 着色语言”（OpenGL Shading Language，是一种特殊的 C 语言）来编写 OpenGL 程序。系统会把这种程序以纯文本的形式传送给 GPU，由 GPU 来负责编译并执行。与传统方式相比，这种方式写出来的程序运行速度更快，而且用途更多，但它尚未受到广泛支持。

本章将要讲解一个 PyOpenGL 程序与两个 pyglet 程序，这些范例程序会演示 3D OpenGL 编程的许多基本功能。这一章的范例都采用传统方式编写，因为这样更容易看到通过调用函数来绘制 3D 图形的过程，而不用再去学习 OpenGL Shading Language，因为这毕竟是一本讲 Python 语言的书，所以我们的重点仍然放在 Python 编程上面。由于笔者假定你已对 OpenGL 编程有一定的了解，因此本章大部分 OpenGL 调用都不会加以解释。如果还不熟悉 OpenGL，那么可以先看看附录 B 中提到的《OpenGL SuperBible》一书，这是一本实用的教程。

在 OpenGL 编程中要注意的一个重要问题就是“命名约定”（naming convention）。许多 OpenGL 函数名的末尾都带有数字，数字后面跟着若干字母。数字表示参数的个数，而字母表示参数类型。比方说，`glColor3f()` 函数就用三个浮点类型的参数来设置当前颜色，这三个参数依次表示红、绿、蓝颜色分量，每个参数的取值范围是 0.0 至 1.0，而 `glColor4ub()` 函数则用四个无符号字节参数来设置颜色，这四个无符号字节分别表示红、绿、蓝、alpha（透明度）分量，每个参数的取值范围是 0 至 255。在编写 Python 代码时，我们可以照常使用任意类型的数字，在必要的时候，Python 会自动帮我们转换。

把三维场景投影到二维平面（例如电脑屏幕）通常有两种方式：正交投影和透视投影。“正交投影”（orthographic projection）能够保持物体大小不变，CAD（computer-aided design，电脑辅助设计）工具经常使用这种方式。而“透视投影”（perspective projection）则会把离观察者近的物体投射得大一些，把远的物体投射得小一些。这么做更逼真，尤其是在绘制景观时。本章的两节将分别用透视投影及正交投影来创建场景。

## 8.1 用透视投影法创建场景

本节将创建 Cylinder 程序，其运行效果如图 8.1 所示。这两个程序都会显示三条坐标轴以及一个有光照效果的中空圆柱体。PyOpenGL 版本（左侧截图）更符合 OpenGL 接口，而 pyglet 版本（右侧截图）的代码写起来更加容易，而且效率也稍微高一点。

这两个程序的大部分代码都相同，另外有一些方法只是名称不同而已。鉴于此，我们在 8.1.1 节中会完整地讲解 PyOpenGL 版本，而在 8.1.2 节里讲解 pyglet 版本时，则只讲述其中与 PyOpenGL 版本不同的那些内容。后面的 8.2 节还会出现大量 pyglet 代码。

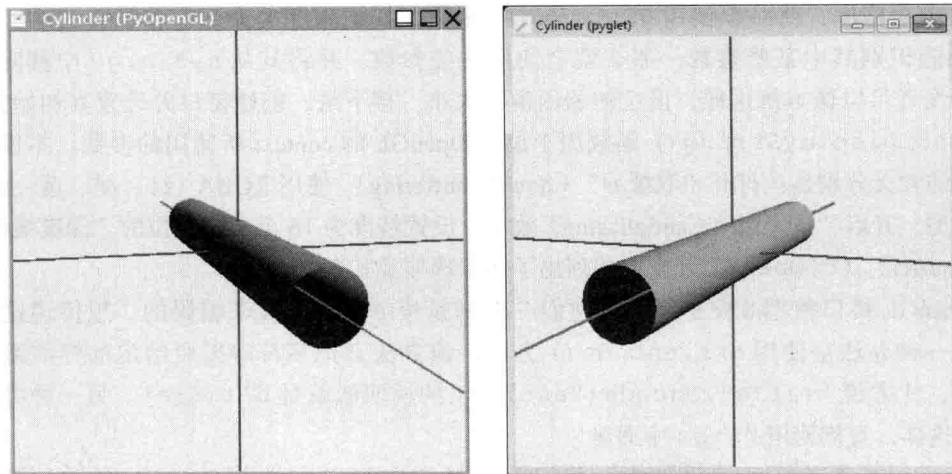


图 8.1 Cylinder 程序在 Linux 及 Windows 系统中的运行截图

### 8.1.1 用 PyOpenGL 编写 Cylinder 程序

cylinder1.pyw 程序创建了一个简单的场景，用户可以分别围绕着 x 轴与 y 轴来旋转场景。当窗口尺寸改变时，其中的场景也会随之缩放。

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLU import *
```

该程序会用到 OpenGL 的 GL 库（这是“核心库”(core library)）、GLU 库（这是个“实用工具库”(utility library)）以及 GLUT 库（这是个“窗口工具包”(windowing toolkit)）。一般情况下，最好不要用 `from module import *` 语句引入程序库，但对于 PyOpenGL 来说，这么做是合理的，因为引入的所有名称都带有 gl、glu、glut、GL 等前缀，所以很容易判断某个名称来自哪个模块，因此不太可能产生冲突。

```
SIZE = 400
ANGLE_INCREMENT = 5

def main():
    glutInit(sys.argv)
    glutInitWindowSize(SIZE, SIZE)
    window = glutCreateWindow(b"Cylinder (PyOpenGL)")
    glutInitDisplayString(b"double=1 rgb=1 samples=4 depth=16")
    scene = Scene(window)
    glutDisplayFunc(scene.display)
    glutReshapeFunc(scene.reshape)
    glutKeyboardFunc(scene.keyboard)
    glutSpecialFunc(scene.special)
    glutMainLoop()
```

与其他的 GUI 工具包一样，GLUT 程序库也提供了事件处理机制和顶级窗口。要想

使用这个程序库，就必须调用 `glutInit()` 函数，并把程序的命令行参数传过去，该函数如果能识别其中某些参数，那么就会使用这些参数，并将其从 `sys.argv` 中移除。然后，开发者可以像本例这样，设定初始的窗口大小。接下来，创建窗口并设置其初始标题。`glutInitDisplayString()` 函数用于设置 OpenGL 的 context 所使用的参数，本例中四个参数的含义分别是：打开“双缓冲”（double-buffering）、使用 RGBA（红、绿、蓝、alpha）颜色模型、开启“抗锯齿”（antialiasing）效果、设置精度为 16 个二进制位的“深度缓冲区”（depth buffer）。（PyOpenGL 开发文档列出了所有选项及其含义。）

OpenGL 接口使用 8 位字符串，它们一般都是按 ASCII 标准来编码的。想传递这种字符串，一种办法是使用 `str.encode()` 方法，该方法会把字符串按照给定的标准编码成 `bytes`，比方说 `"title".encode("ascii")` 的返回值就是 `b'title'`。另一种办法是像本例这样，直接使用 `bytes` 字面量。

`Scene` 是笔者自己编写的类，我们将通过它向窗口中绘制 OpenGL 图像。创建好 `Scene` 之后，我们把该类中的一些方法注册成 GLUT 回调函数，也就是说，如果发生了某种事件，那么 OpenGL 就会调用相应的函数来响应此事件。首先注册的方法是 `Scene.display()`，当窗口显示出来的时候（有可能是启动程序之后首次显示出来，也有可能是挡在它前面的其他窗口移开或关闭了），OpenGL 就会调用该方法。然后注册 `Scene.reshape()` 方法，当窗口尺寸改变时，OpenGL 会调用该方法。第三个注册的是 `Scene.keyboard()` 方法，如果用户按了键盘中的键（某些键除外），那么 OpenGL 就会调用此方法。最后注册 `Scene.special()` 方法，如果刚才注册的“键盘函数”（keyboard function）无法处理用户所按的键，那么 OpenGL 就会调用这个方法。

创建完窗口并注册完回调函数之后，启动 GLUT 事件循环。该循环会持续运行，直到程序终止。

```
class Scene:
    def __init__(self, window):
        self.window = window
        self.xAngle = 0
        self.yAngle = 0
        self._initialize_gl()
```

`Scene` 类的 `__init__()` 方法首先将 `self.window` 这个引用指向 OpenGL 窗口，然后把 x 轴与 y 轴的旋转角度设为 0。最后，把与 OpenGL 相关的所有初始化工作都交给 `self._initialize_gl()` 函数来做。

```
def _initialize_gl(self):
    glClearColor(195/255, 248/255, 248/255, 1)
    glEnable(GL_DEPTH_TEST)
    glEnable(GL_POINT_SMOOTH)
    glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)
    glEnable(GL_LINE_SMOOTH)
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST)
```

```

glEnable(GL_COLOR_MATERIAL)
glEnable(GL_LIGHTING)
glEnable(GL_LIGHT0)
glLightfv(GL_LIGHT0, GL_POSITION, vector(0.5, 0.5, 1, 0))
glLightfv(GL_LIGHT0, GL_SPECULAR, vector(0.5, 0.5, 1, 1))
glLightfv(GL_LIGHT0, GL_DIFFUSE, vector(1, 1, 1, 1))
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50)
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, vector(1, 1, 1, 1))
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)

```

上面这个方法只会在设定 OpenGL 的 context 时调用一次。首先把“清除色”(clear color, 也就是“背景色”(background color)) 设置为浅蓝, 然后启用多项 OpenGL 功能, 其中最重要的就是创建“光源”(light)。圆柱体表面的颜色之所以不均一, 就是由于场景里面有光源。此外, 我们令开发者可以通过调用 glColor...() 函数来改变圆柱体的“基本颜色”(basic color, 也就是未受光源照射时的颜色), 比方说, 启用了 GL\_COLOR\_MATERIAL 选项之后, 如果调用 glColor3ub(255, 0, 0) 函数把当前颜色设为红色, 那么这就会影响到“材质颜色”(material color, 在本例中指的是圆柱体的颜色)。

```

def vector(*args):
    return (GLfloat * len(args))(*args)

```

上面这个辅助函数用来创建 OpenGL 的浮点数数组 (数组中每个浮点数的类型都是 GLfloat)。

```

def display(self):
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    glPushMatrix()
    glTranslatef(0, 0, -600)
    glRotatef(self.xAngle, 1, 0, 0)
    glRotatef(self.yAngle, 0, 1, 0)
    self._draw_axes()
    self._draw_cylinder()
    glPopMatrix()

```

如果场景窗口是第一次显示, 或者覆盖在上面的其他窗口移走或关闭了, 那么 OpenGL 就会调用上面这个方法。它首先把场景沿着 z 轴向后移动, 使用户能够从正前方看到圆柱体, 然后根据用户的操作来旋转 x 轴及 y 轴。(一开始, 这两个轴的旋转角度都是 0 度。) 将场景平移并旋转之后, 我们把坐标轴及圆柱体本身绘制出来。

```

def _draw_axes(self):
    glBegin(GL_LINES)
    glColor3f(1, 0, 0)      # x-axis
    glVertex3f(-1000, 0, 0)
    glVertex3f(1000, 0, 0)
    glColor3f(0, 0, 1)      # y-axis
    glVertex3f(0, -1000, 0)
    glVertex3f(0, 1000, 0)
    glColor3f(1, 0, 1)      # z-axis

```

```
glVertex3f(0, 0, -1000)
glVertex3f(0, 0, 1000)
glEnd()
```

在 OpenGL 的术语中，三维空间里的点叫做 vertex。我们绘制三条坐标轴所用的方式是一样的：每次都是先设定坐标轴颜色，然后给出待绘制线段的“起点”(start vertex) 和“终点”(end vertex)。glColor3f() 与 glVertex3f() 函数都需要用三个浮点数做参数，不过我们在程序中直接给这些参数传入 int 就好，转换工作会由 Python 来完成。

```
def _draw_cylinder(self):
    glPushMatrix()
    try:
        glTranslatef(0, 0, -200)
        cylinder = gluNewQuadric()
        gluQuadricNormals(cylinder, GLU_SMOOTH)
        glColor3ub(48, 200, 48)
        gluCylinder(cylinder, 25, 25, 400, 24, 24)
    finally:
        gluDeleteQuadric(cylinder)
    glPopMatrix()
```

GLU 这个实用工具库本身就可以创建一些基本的 3D 形状，其中也包括圆柱体。首先，我们把坐标原点继续沿着 z 轴向远处推。然后创建 quadric (二次曲面)，此对象可用来渲染多种 3D 形状。我们用三个无符号字节来指定颜色 (这三个字节分别表示红、绿、蓝颜色分量，其取值范围都是 0 至 255)。gluCylinder() 函数有六个参数，第一个参数是刚才创建的那个通用的 quadric 对象，第二与第三个参数表示圆柱体上下底面的半径 (在本例中，这两个参数相同)，第四个参数是圆柱体的高度，最后两个参数是两个“粒度因子”(granularity factor)，这两个值越高，绘制效果就越平滑，但是绘制时间也会随之变长。最后，我们不用等待 Python 进行垃圾回收，而是明确地删掉这个 quadric，以尽量降低程序的资源占用量。

```
def reshape(self, width, height):
    width = width if width else 1
    height = height if height else 1
    aspectRatio = width / height
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(35.0, aspectRatio, 1.0, 1000.0)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
```

当场景窗口的尺寸改变时，上述方法就会执行。它把几乎全部工作都交给 gluPerspective() 函数来做。实际上，凡是采用透视投影的场景，都可以用这段代码作为起点来编写。

```
def keyboard(self, key, x, y):
    if key == b"\x1B": # Escape
        glutDestroyWindow(self.window)
```

如果用户按下的键不是功能键、方向键、Page Up、Page Down、Home、End 或 Insert 键，那么上面这个方法就会执行（我们早前曾用 glutKeyboardFunc() 将其注册过）。我们在这个方法里判断用户按的是不是 Esc 键，如果是，那么就将窗口删除，此时由于没有其他窗口，所以程序会终止。

```
def special(self, key, x, y):
    if key == GLUT_KEY_UP:
        self.xAngle -= ANGLE_INCREMENT
    elif key == GLUT_KEY_DOWN:
        self.xAngle += ANGLE_INCREMENT
    elif key == GLUT_KEY_LEFT:
        self.yAngle -= ANGLE_INCREMENT
    elif key == GLUT_KEY_RIGHT:
        self.yAngle += ANGLE_INCREMENT
    glutPostRedisplay()
```

上面这个方法是通过 glutSpecialFunc() 函数注册的，只有当用户按下功能键、方向键、Page Up、Page Down、Home、End 或 Insert 键的时候，这个方法才会执行。如果用户按的是方向键，那么就把场景与 x 轴或 y 轴的夹角递增或递减，然后告诉 GLUT 工具包窗口内容需要重新绘制。这会使早前通过 glutDisplayFunc() 注册的那个 callable 得到调用，在本例中，这个 callable 就是 Scene.display() 方法。

我们现在已经看到 PyOpenGL 版 cylinder1.pyw 程序的全部代码了。熟悉 OpenGL 编程的读者应该对这些代码非常熟悉，因为用 Python 语言来调用 OpenGL 函数和用 C 语言来调用几乎完全一样。

### 8.1.2 用 pyglet 编写 Cylinder 程序

从结构上看，pyglet 版的 cylinder2.pyw 程序和 PyOpenGL 版的 cylinder1.pyw 非常相似。关键区别在于，pyglet 本身就提供了事件处理机制及窗口创建接口，所以我们无须再调用 GLUT 中的函数了。

```
def main():
    caption = "Cylinder (pyglet)"
    width = height = SIZE
    resizable = True
    try:
        config = Config(sample_buffers=1, samples=4, depth_size=16,
                        double_buffer=True)
        window = Window(width, height, caption=caption, config=config,
                        resizable=resizable)
    except pyglet.window.NoSuchConfigException:
        window = Window(width, height, caption=caption,
                        resizable=resizable)
    path = os.path.realpath(os.path.dirname(__file__))
    icon16 = pyglet.image.load(os.path.join(path, "cylinder_16x16.png"))
    icon32 = pyglet.image.load(os.path.join(path, "cylinder_32x32.png"))
```

```
window.set_icon(icon16, icon32)
pyglet.app.run()
```

在上一小节中，我们曾把一些 bytes 形式的字符串传给相关函数，以便配置 OpenGL 的 context，而本节则不用这样做，因为 pyglet 允许开发者直接通过 pyglet.gl.Config 对象来指明需求。我们先把自己想要的配置放在 config 里，然后根据 config 来创建自编的 Window 对象（Window 类是 pyglet.window.Window 的子类），若该操作失败，则根据默认的配置来创建窗口。

pyglet 有一项功能很棒，那就是可以设置应用程序的图标，该图标会显示在标题栏的某一角，而且还会在“任务切换器”(task switcher) 里面显示。创建完窗口并设置好图标之后，我们启动 pyglet 的事件循环。

```
class Window(pyglet.window.Window):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.set_minimum_size(200, 200)
        self.xAngle = 0
        self.yAngle = 0
        self._initialize_gl()
        self._z_axis_list = pyglet.graphics.vertex_list(2,
            ("v3i", (0, 0, -1000, 0, 0, 1000)),
            ("c3B", (255, 0, 255) * 2)) # one color per vertex
```

上面这个方法与上一小节 Scene 类的 `__init__()` 方法相似，区别在于它设定了窗口的最小尺寸。我们稍后就会看到，pyglet 能够以三种不同的方式画线。第三种方式会根据预先设定好的列表来绘制线段，而列表中有两个 pair，用来表示线段的两个端点及其颜色。我们通过 `pyglet.graphics.vertex_list()` 函数来创建这样的列表。该函数的第一个参数表示 pair 的数量，其后跟着如数的 pair。每个 pair 里面有两个值，第一个值是格式字符串，第二个值是序列。在本例中，第一个 pair 中的格式字符串是 "v3i"，意思是“由三个整数坐标所描述的点”(vertices specified by three integer coordinates)，由于每个点需要用三个整数来描述，而我们需要描述两个点，所以这个 pair 的序列里面需要有六个整数。第二个 pair 所用的格式字符串是 "c3B"，意思是“由三个无符号字节所描述的颜色”(colors specified by three unsigned bytes)，在本例中，我们需要描述两个点的颜色，每个点的颜色要用三个无符号字节，而这两个点的颜色又相同，于是这个 pair 里的序列就是 (255, 0, 255) \* 2。

本节没有列出 `_initialize_gl()`、`on_draw()`、`on_resize()` 及 `_draw_cylinder()` 方法。`_initialize_gl()` 方法与 cylinder1.pyw 所用的同名方法非常相似。pyglet 在显示 Window 的时候（Window 是 pyglet.window.Window 的子类），会自动调用 `on_draw()` 方法，而该方法的代码与 cylinder1.pyw 程序的 `Scene.display()` 方法相同。与之类似，窗口尺寸改变时所调用的 `on_resize()` 方法其代码也和 cylinder1.

pyw 程序的 Scene.reshape() 方法相同。另外，Scene 的 \_draw\_cylinder() 方法与 Window 的 \_draw\_cylinder() 也一样。

```
def _draw_axes(self):
    glBegin(GL_LINES)           # x-axis (traditional-style)
    glColor3f(1, 0, 0)
    glVertex3f(-1000, 0, 0)
    glVertex3f(1000, 0, 0)
    glEnd()
    pyglet.graphics.draw(2, GL_LINES, # y-axis (pyglet-style "live")
        ("v3i", (0, -1000, 0, 0, 1000, 0)),
        ("c3B", (0, 0, 255) * 2))
    self._z_axis_list.draw(GL_LINES) # z-axis (efficient pyglet-style)
```

我们用三种不同的办法来绘制三条坐标轴，以此演示 pyglet 的各种画线方式。x 轴是通过调用传统的 OpenGL 函数来绘制的，这与 PyOpenGL 版本的程序所用的办法完全相同。而 y 轴的绘制办法则是把两个点（点的个数其实并不限于两个）的坐标及颜色告诉 pyglet，令其在两点之间画线。这种方式的效率要比传统方式高一点，尤其是在绘制大量线段的时候。绘制 z 轴所用的办法效率最高：我们事先把包含点及颜色信息的若干 pair 放到 pyglet.graphics.vertex\_list 形式的列表里面，然后调用这个 vertex\_list 的 draw() 方法在点之间画线。

```
def on_text_motion(self, motion): # Rotate about the x or y axis
    if motion == pyglet.window.key.MOTION_UP:
        self.xAngle -= ANGLE_INCREMENT
    elif motion == pyglet.window.key.MOTION_DOWN:
        self.xAngle += ANGLE_INCREMENT
    elif motion == pyglet.window.key.MOTION_LEFT:
        self.yAngle -= ANGLE_INCREMENT
    elif motion == pyglet.window.key.MOTION_RIGHT:
        self.yAngle += ANGLE_INCREMENT
```

如果用户按了方向键，并且 Window 中又定义了名叫 on\_text\_motion() 的方法，那么程序就会调用该方法。上面这个方法所执行的操作与上一小节 cylinder1.pyw 程序的 special() 方法相同，只不过这次在表示按键的时候，用的不是 GLUT 里面的常量，而是 pyglet 里面的常量。

假如 Window 类还定义了 on\_key\_press() 方法，那么当用户按了键盘上的普通按键时，程序就会调用那个方法。pyglet 默认会在用户按下 Esc 键的时候关闭窗口，窗口关闭之后，应用程序也会随之终止，而这正是我们想要的效果，于是，就不需要再去实现 on\_key\_press() 方法了。

两个版本的 Cylinder 程序都用了 140 行左右的代码。但是通过 pyglet.graphics.vertex\_list 与其他 pyglet 扩展，我们可以写出既简洁又高效的代码，尤其在事件及窗口的处理上面，pyglet 比 PyOpenGL 更方便。

## 8.2 用正交投影法制作游戏

在第 7 章中，我们讲解了如何编写 2D 的 Gravitate 游戏，但却没说游戏中的方块是怎么画出来的。实际上，方块的画法是先画个正方形，然后在其上、下、左、右四个方位分别画四个等腰梯形。上方和左方的梯形颜色要比方块颜色浅一些，而下方和右方的梯形颜色则要比方块颜色深一些，这样看起来就有 3D 效果了（游戏画面参见图 7.7，游戏规则参见 7.3 节中题为“Gravitate”的部分）。

3D 版 Gravitate 游戏的运行效果如图 8.2 所示，本节会讲解该程序的大部分代码。这个程序用球体而非方块来表示游戏中的元素，并且在相邻的球体之间留有一定的空白，使玩家在沿着 x 轴与 y 轴旋转游戏场景时，能够看清立体结构。我们把重点放在 GUI 和 3D 代码上，与游戏逻辑有关的一些底层代码就不讲了。完整的代码位于 gravitate3d.pyw 文件里。

3D 版游戏程序的 main() 函数（本书没有列出这个函数的代码）和 cylinder2.pyw 程序的 main() 函数几乎完全相同，区别只在于窗口标题和图标图像的名称不同。

```
BOARD_SIZE = 4 # Must be > 1.
ANGLE_INCREMENT = 5
RADIUS_FACTOR = 10
DELAY = 0.5 # seconds
MIN_COLORS = 4
MAX_COLORS = min(len(COLORS), MIN_COLORS)
```

上面这些是程序要用到的常量。BOARD\_SIZE 表示每条坐标轴上的球体个数。如果将其设为 4，那么就表示 x 轴、y 轴、z 轴上各有 4 个球体，而整个场景里总共会有 64 个球体 ( $64 = 4 \times 4 \times 4$ )。ANGLE\_INCREMENT 设为 5 的意思是用户每按一下方向键，游戏场景就要旋转 5 度。当用户选中某个球体并再次点击它的时候，与该球临近的同色球以及与那些球临近的同色球都会消去，而消去之后，其他球体会向场景中心聚拢，以填补空隙。在“消去”与“聚拢”操作之间会有延迟，其时间长短由 DELAY 常量决定。COLORS 常量（该常量没有列在上面的代码中）是个“列表”(list)，其中每个元素都是“元组”(tuple)，而每个元组里面又有三个整数，其取值均位于 0 至 255 之间。这个列表里的每个元组都表示一种颜色。

如果玩家所点击的球体处于“尚未选中”(unselected) 状态，那么就将其选中，而玩家原

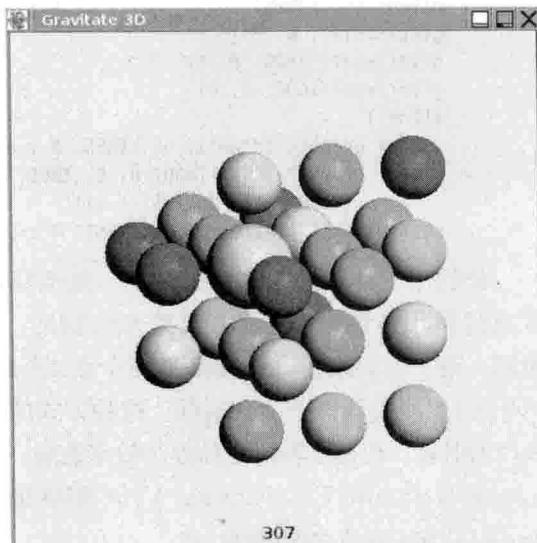


图 8.2 Gravitate 3D 游戏在 Linux 系统中的运行截图

来所选中的那个球体则会恢复到尚未选中时的状态。当前选中的这个球体其半径会比其他球体大，两者的半径之差用 RADIUS\_FACTOR 常量表示。若玩家所点击的球体已经处在“选中”(selected) 状态，则判断是否有同色的球与之临近（所谓临近，是从 90 度方向判断的<sup>⊖</sup>，斜方向的不算）。如果有，那么就将本球体及所有与之临近的同色球体全部删去；如果没有，那么就把当前球体恢复到尚未选中时的状态。

```
class Window(pyglet.window.Window):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.set_minimum_size(200, 200)
        self.xAngle = 10
        self.yAngle = -15
        self.minColors = MIN_COLORS
        self.maxColors = MAX_COLORS
        self.delay = DELAY
        self.board_size = BOARD_SIZE
        self._initialize_gl()
        self.label = pyglet.text.Label("", bold=True, font_size=11,
                                       anchor_x="center")
        self._new_game()
```

与 Cylinder 程序的 `__init__()` 方法相比，上面这个 `__init__()` 方法用的语句更多，因为要设置颜色、延迟及游戏画面尺寸，而且还要在启动游戏之前先旋转场景，使玩家直接能看到三维效果。

pyglet 提供了一项非常有用的功能，就是可以设置文本标签。本例中，我们创建空标签，并令其在场景底部居中。这个标签可用来显示消息及当前分数。

自编的 `_initialize_gl()` 方法（该方法的代码没有列出来，但是与早前看到的同名方法类似）设置了背景色及光源。按照程序的逻辑与所需的 OpenGL 功能设置好各项参数后，调用 `_new_game()` 方法，开始新一局游戏。

```
def _new_game(self):
    self.score = 0
    self.gameOver = False
    self.selected = None
    self.selecting = False
    self.label.text = ("Click to Select • Click again to Delete • "
                      "Arrows to Rotate")
    random.shuffle(COLORS)
    colors = COLORS[:self.maxColors]
    self.board = []
    for x in range(self.board_size):
        self.board.append([])
        for y in range(self.board_size):
            self.board[x].append([])
```

---

<sup>⊖</sup> 也就是说，某球体与本球体的连线必须与某条坐标轴平行。——译者注

```

for z in range(self.board_size):
    color = random.choice(colors)
    self.board[x][y].append(SceneObject(color))

```

上面这个方法会从 COLORS 列表中随机选取颜色来创建游戏画面中的每个球体，COLORS 列表中最多会有 self.maxColors 种颜色用于创建球体。游戏画面用 board 来表示，它是个“三重列表”（a list of lists of lists），最里面那一层的列表中存放的是 SceneObject 对象。SceneObject 对象有两个颜色，一个是经由构造器传过去的 color，另一个是由程序自动生成的 selection color，此颜色用于判定玩家是否选中了这个球体，我们将在 8.2.2 节里解释它。

由于我们修改了标签中的文本，所以 pyglet 会调用 on\_draw() 方法，以重新绘制游戏场景。此时玩家就能看到新的游戏画面，并且可以开始操作了。

### 8.2.1 绘制游戏场景

如果场景是首次显示出来的，或是因挡在上面的窗口关闭或移走而重新显示出来的，那么 pyglet 就会调用 on\_draw() 方法。场景尺寸（也就是其窗口尺寸）有变化时，pyglet 会调用 on\_resize() 方法。

```

def on_resize(self, width, height):
    size = min(self.width, self.height) / 2
    height = height if height else 1
    width = width if width else 1
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    if width <= height:
        glOrtho(-size, size, -size * height / width,
                size * height / width, -size, size)
    else:
        glOrtho(-size * width / height, size * width / height,
                -size, size, -size, size)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

```

Gravitate 3D 程序的游戏场景是用正交投影法绘制的。凡是想用正交投影法来绘制场景的程序都可以采用上面这段代码来写。（假如采用 PyOpenGL 来开发，那么可以把上面这个方法改名为 reshape()，并且用 glutReshapeFunc() 函数来注册它。）

```

def on_draw(self):
    diameter = min(self.width, self.height) / (self.board_size * 1.5)
    radius = diameter / 2
    offset = radius - ((diameter * self.board_size) / 2)
    radius = max(RADIUS_FACTOR, radius - RADIUS_FACTOR)
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)

```

```

glPushMatrix()
glRotatef(self.xAngle, 1, 0, 0)
glRotatef(self.yAngle, 0, 1, 0)
with Selecting(self.selecting):
    self._draw_spheres(offset, radius, diameter)
glPopMatrix()
if self.label.text:
    self.label.y = (-self.height // 2) + 10
    self.label.draw()

```

在采用 PyOpenGL 来开发 3D 程序时，我们曾通过 `glutDisplayFunc()` 函数注册了 `display()` 方法，而在采用 pyglet 来开发程序时，则可以实现上面这个等价的方法。我们想令游戏画面尽量填满窗口空间，同时球与球的间距又不能拉得太大，否则场景旋转之后，有些球体就会超出窗口范围。另外，还需要计算一个“偏移量”（`offset`），以确保游戏画面能在窗口里居中。

把准备工作处理好之后，我们旋转场景以反映玩家的操作（比方说玩家按下了方向键），然后在自编的 `Selecting context manager` 范围内绘制球体。绘制时要分两种情况处理：一种情况是直接把场景绘制给玩家看，而另一种情况则是把场景绘制到玩家看不到的缓冲区里，以判断玩家是否选中了某个球（8.2.2 节将会讲解具体的判定办法）。这个 `context manager` 能根据上述两种情况来开启或关闭某些特定的设置。

由于窗口大小可能会变，所以如果标签里面有文本的话，那么就要确保标签的 `y` 坐标始终位于窗口底部，然后调用标签的 `draw()` 方法，以绘制其中的文本。

```

def _draw_spheres(self, offset, radius, diameter):
    try:
        sphere = gluNewQuadric()
        gluQuadricNormals(sphere, GLU_SMOOTH)
        for x, y, z in itertools.product(range(self.board_size),
                                         repeat=3):
            sceneObject = self.board[x][y][z]
            if self.selecting:
                color = sceneObject.selectColor
            else:
                color = sceneObject.color
            if color is not None:
                self._draw_sphere(sphere, x, y, z, offset, radius,
                                  diameter, color)
    finally:
        gluDeleteQuadric(sphere)

```

`quadric` 既可以用来绘制圆柱体，也可以用来绘制球体。早前讲过的 `Cylinder` 程序只需绘制一个圆柱体，而本例则要绘制多达 64 个球体。我们采用相同的 `quadric` 来绘制每个球体。

刚才在填充名为 `board` 的三重列表时，我们采用三层循环（也就是 `for x in range(self.board.size): for y in range(self.board.size): for z in range(self.board.`

`size)` 语句) 来确定每个球体的 `x`、`y`、`z` 下标, 但上面这个方法却不这样做, 而是将 `for` 语句和 `itertools.product()` 函数搭配起来使用, 于是只需一层循环就能达到相同效果。

我们根据 `x`、`y`、`z` 下标从三重列表里取出对应的 `sceneObject`, 然后根据情况来设定 `color` 变量: 如果当前正在判断玩家是否点击了某个球体, 那么就将 `color` 设为 `sceneObject.selectColor`; 若是正在把游戏场景直接绘制给玩家看, 则将 `color` 的值设为 `sceneObject.color`。如果 `color` 变量是 `None`, 那么就表示该球体已经从画面中移除了; 但如果不是 `None`, 那么我们就要把这个球体画出来。

```
def _draw_sphere(self, sphere, x, y, z, offset, radius, diameter,
                 color):
    if self.selected == (x, y, z):
        radius += RADIUS_FACTOR
        glPushMatrix()
        x = offset + (x * diameter)
        y = offset + (y * diameter)
        z = offset + (z * diameter)
        glTranslatef(x, y, z)
        glColor3ub(*color)
        gluSphere(sphere, radius, 24, 24)
        glPopMatrix()
```

我们用上面这个方法来绘制每个球体。与球体在三维空间里的正确位置相比, 绘制出来的位置应该有所偏移, 偏移量由 `offset` 参数决定。如果待绘制的球体正处于“受选”(`selected`) 状态, 那么我们会增加其半径。`gluSphere()` 函数的最后两个参数是“粒度因子”(`granularity factor`), 它们的值越大, 绘制出来的效果就越平滑, 但是所需的处理时间也会随之增加。

## 8.2.2 判断用户是否选中了场景里的物体

在以二维平面的形式所呈现的三维空间里, 要想判断用户是否选中了某个物体可不是一件容易的事。这些年来, 有许多种技术都致力于解决这一问题, 而 Gravitate 3D 游戏所采用的这种判定技术似乎是相当可靠的, 并且其适用范围也非常广。

这项技术的工作原理如下。当用户点击场景时, 我们把场景重新绘制到一块“离屏缓冲区”(off-screen buffer) 里面, 绘制的时候, 每个物体所用的颜色互不相同, 而用户则看不到这块缓冲区。绘制好之后, 我们从缓冲区里查出用户所点击的那个像素是什么颜色, 然后用该颜色与场景里每个物体的颜色相比对, 以确定用户到底选中了哪个物体。为了使这项技术奏效, 我们在绘制场景时必须关闭“抗锯齿”(antialiasing)、“光照”(lighting) 及“贴图”(texture) 功能, 以确保绘制每个物体时所用的颜色都互不相同, 而且都没有经过额外处理。

我们首先来看 `SceneObject`, 每个球体都是用这样一个对象来表示的, 其后, 我们来讲 `Selecting context manager`。

```

class SceneObject:
    __SelectColor = 0
    def __init__(self, color):
        self.color = color
        SceneObject.__SelectColor += 1
        self.selectColor = SceneObject.__SelectColor

```

每个 SceneObject 对象都有两种颜色，一种是绘制时所用的显示颜色（以 self.color 来表示），另一种则是判断是否受选时所用的颜色（以 self.selectColor 来表示），前者未必互不相同，而后者则肯定独一无二。私有的静态变量 \_\_SelectColor 是个整数，每新建一个 SceneObject，它的值就会加 1，而我们正是根据这个值来设置 selectColor 的，这样设置能使每个 SceneObject 的 selectColor 互不相同。

```

@property
def selectColor(self):
    return self.__selectColor

```

上面这个属性会返回 SceneObject 的 select color<sup>⊖</sup>，如果该对象已从画面中移除，那么返回的就是 None，否则返回的是“三元组”(3-tuple)，其中每个元素代表一种颜色分量，其取值位于 0 至 255 之间。

```

@selectColor.setter
def selectColor(self, value):
    if value is None or isinstance(value, tuple):
        self.__selectColor = value
    else:
        parts = []
        for _ in range(3):
            value, y = divmod(value, 256)
            parts.append(y)
        self.__selectColor = tuple(parts)

```

selectColor 属性所对应的 setter 的 value 参数可以接受 None 或 tuple(元组)，如果 setter 发现 value 参数不是这两种值，那么就会把它当成整数，并据此算出一个 tuple 来，然后把这个 tuple 赋给 \_\_selectColor，只要给定的 value 不同，那么算出来的 tuple 也就互不相同。程序所创建的首个 SceneObject 会给 value 参数传入 1，所以 setter 算出来的颜色就是 (1, 0, 0)。第二个 SceneObject 会给 value 参数传入 2，所以算出来的颜色就是 (2, 0, 0)，一直到第 255 个，此时算出来的颜色是 (255, 0, 0)。第 256 个 SceneObject 的颜色是 (0, 1, 0)，第 257 个是 (1, 1, 0)，第 258 个是 (2, 1, 0)，依此类推。这个颜色生成算法可以应对 1600 多万个对象，并保证每个对象的颜色都不同，这对大多数程序来说已经足够用了。

---

<sup>⊖</sup> select color 是为了判定玩家所送的球体而使用的一种特殊渲染颜色。——译者注

```
SELECTING_ENUMS = (GL_ALPHA_TEST, GL_DEPTH_TEST, GL_DITHER,
    GL_LIGHT0, GL_LIGHTING, GL_MULTISAMPLE, GL_TEXTURE_1D,
    GL_TEXTURE_2D, GL_TEXTURE_3D)
```

我们需要根据情况来决定是否开启抗锯齿、光照、材质以及其他会影响物体颜色的选项。如果是把场景绘制给玩家看，那么就开启相关选项；如果是把场景绘制到离屏缓冲区，以判断玩家是否点击了其中的球体，那么就关闭相关选项。上面列出了可能会影响 Gravitate 3D 游戏中物体颜色的各种 OpenGL enum。

```
class Selecting:
    def __init__(self, selecting):
        self.selecting = selecting

    def __enter__(self):
        if self.selecting:
            for enum in SELECTING_ENUMS:
                glDisable(enum)
            glShadeModel(GL_FLAT)

    def __exit__(self, exc_type, exc_value, traceback):
        if self.selecting:
            for enum in SELECTING_ENUMS:
                glEnable(enum)
            glShadeModel(GL_SMOOTH)
```

进入 context manager 的范围之后，如果是为了检测受选球体而绘制，那么就把 OpenGL 状态中可能导致颜色改变的那些选项禁用，并切换到“平面着色模型”(flat shading model)。

```
def __exit__(self, exc_type, exc_value, traceback):
    if self.selecting:
        for enum in SELECTING_ENUMS:
            glEnable(enum)
        glShadeModel(GL_SMOOTH)
```

离开 context manager 的范围时，如果刚才是为了检测受选球体而绘制，那么就将 OpenGL 状态中可能导致颜色改变的那些选项重新启用，并切换回“平滑着色模型”(smooth shading model)。

如果想看一下判定受选球体所用的离屏缓冲区是什么样子，那么只需修改两处源代码即可。首先，把 SceneObject.\_\_init\_\_() 方法中的 `+= 1` 改为 `+= 500`。然后，把 Window.on\_mouse\_press() 方法（这个方法将在下一小节讲解）中的 `self.selecting = False` 语句注释掉。现在可以运行程序，并随意点击某个球体，然后程序就会把离屏缓冲区里的内容当成游戏场景绘制到窗口里面了，除此之外，游戏的其他功能都和修改前一样。

### 8.2.3 处理用户操作

Gravitate 3D 游戏基本上是通过鼠标来操作的。不过玩家也可以用方向键来旋转游戏画面，并通过其他按键来开始新游戏或退出游戏。

```

def on_mouse_press(self, x, y, button, modifiers):
    if self.gameOver:
        self._new_game()
        return
    self.selecting = True
    self.on_draw()
    self.selecting = False
    selectColor = (GLubyte * 3)()
    glReadPixels(x, y, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, selectColor)
    selectColor = tuple([component for component in selectColor])
    self._clicked(selectColor)

```

如果 Window 类提供了 `on_mouse_press()` 方法，那么 pyglet 就会在用户点击鼠标按钮时调用它。如果在完成一盘游戏之后继续点击鼠标，那么我们就再开启一盘新的游戏。如果在当前这盘游戏还未完成时点击鼠标，那么我们就把 `selecting` 设为 `True` 并重新绘制场景（这时将把场景绘制到用户看不见的离屏缓冲区里），绘制完之后，我们再把 `selecting` 重新设置为 `False`。

`glReadPixels()` 函数可以获取一个或多个像素的颜色，在本例中，我们把玩家点击鼠标的位置传给 `glReadPixels()`，以获取离屏缓冲区中对应坐标上的像素颜色，获取的颜色用 RGB 格式表示，这种格式由三个无符号字节组成，每个字节的取值范围都是 0 至 255。获取到颜色之后，我们把这些字节转换成含有三个整数的三元组，以便与缓冲区里每个球体的 `select color` 相比对（每个球体的 `select color` 都互不相同）。

请注意，调用 `glReadPixels()` 函数时，我们假定坐标系原点位于左下角且 y 轴向上延伸（pyglet 所使用的坐标系正是如此）。如果坐标系的原点在左上角，且 y 轴向下延伸，那么调用函数之前还需加上 `viewport = (GLint * 4)()` 及 `glGetIntegerv(GL_VIEWPORT, viewport)` 这两条语句，并且要把传给 `glReadPixels()` 的参数 `y` 改成 `viewport[3] - y`。

```

def _clicked(self, selectColor):
    for x, y, z in itertools.product(range(self.board_size), repeat=3):
        if selectColor == self.board[x][y][z].selectColor:
            if (x, y, z) == self.selected:
                self._delete() # Second click deletes
            else:
                self.selected = (x, y, z)
    return

```

如果玩家点击鼠标时不需要新开启一盘游戏，那么程序就会调用上面这个方法。我们通过 `itertools.product()` 函数来生成 `(x, y, z)` 三元组，并以此为下标来遍历 `board` 中的球体。对于每一个球体来说，我们都把用户所点击的那个像素的颜色同球体的 `select color` 相比对。如果二者一致，那么就能确定玩家肯定点击了当前这个球体。若该球体早前已处在选定状态，则说明玩家第二次点击了它，于是，我们把该球体及临近的同色球体都从画面里删去。若该球体尚未处在选定状态，则将其选中；如果原来有其他球体受选，那

么就将那个球体恢复到“未受选”(unselected)状态。

```
def _delete(self):
    x, y, z = self.selected
    self.selected = None
    color = self.board[x][y][z].color
    if not self._is_legal(x, y, z, color):
        return
    self._delete_adjoining(x, y, z, color)
    self.label.text = "{:,}.".format(self.score)
    pyglet.clock.schedule_once(self._close_up, self.delay)
```

上面这个方法用于消去受选球体及临近的同色球体(那些同色球体附近如果还有同色球体,也一并消去)。首先我们将受选球体解除选定,然后判断是否有同色球体可供消去(也就是说,本球体旁边至少要有1个同色球体方能执行消去操作)。如果有可以消除的同色球体,那么就调用`_delete_adjoining()`方法来消除,而`_delete_adjoining()`方法又会调用其辅助方法以完成操作(本书没有列出`_delete_adjoining()`方法及其辅助方法的代码)。接下来,我们更新标签内容,把玩家通过消去操作所得的分数加到现有分数上面,然后计划在半秒钟之后调用`self._close_up()`方法(该方法的代码没有列出来)。消去同色球体之后,其他球体会向场景中心聚拢,以填补空隙,而有了这半秒钟的延迟,玩家就可以在其他球体聚拢之前,先看清自己刚才到底消去了哪些球体。(这个聚拢过程还可以展示得更加细腻一些,也就是说,不要立刻就聚拢,而是每次向场景中心移动几个像素,分步骤完成聚拢操作,以实现动画效果。)

```
def on_key_press(self, symbol, modifiers):
    if (symbol == pyglet.window.key.ESCAPE or
        ((modifiers & pyglet.window.key.MOD_CTRL or
         modifiers & pyglet.window.key.MOD_COMMAND) and
         symbol == pyglet.window.key.Q)):
        pyglet.app.exit()
    elif ((modifiers & pyglet.window.key.MOD_CTRL or
           modifiers & pyglet.window.key.MOD_COMMAND) and
           symbol == pyglet.window.key.N):
        self._new_game()
    elif (symbol in {pyglet.window.key.DELETE, pyglet.window.key.SPACE,
                    pyglet.window.key.BACKSPACE} and
          self.selected is not None):
        self._delete()
```

玩家可以点击带有“x”图样的按钮来终止游戏程序,不过我们也允许玩家按 Esc 键或 Ctrl+Q 组合键(或⌘Q)来终止。在当前这盘游戏已经完成的情况下,玩家可以通过点击鼠标来启动新游戏,此外,无论何时,都可以通过按 Ctrl+N(或⌘N)组合键来开始新游戏。玩家首次选中某个球体之后,可以通过再次点击它来消去本球体及其临近的同色球体,除了这种方式之外,我们也允许玩家通过 Del、Space 或 Backspace 键来执行消去操作。

Window 类里也有个名叫 `on_text_motion()` 的方法，用于处理旋转操作，使玩家可以通过方向键将游戏场景绕着 x 轴或 y 轴旋转。由于这个方法的代码和 8.1.2 节中的同名方法一样，所以我们没有将其列出来。

Gravitate 3D 游戏的代码到这里就讲完了。没有提到的那些方法都与游戏的逻辑细节有关，尤其是用于消去临近的同色球（将球体的 `color` 与 `select_color` 设为 `None`）以及使其他球体向场景中心聚拢所用的那些方法更是如此。

用程序来创建 3D 场景是相当有挑战性的，其中一个重要原因在于：传统的 OpenGL 接口完全是“过程式的”（procedural，也就是说，是“基于函数的”（based on functions））而非“面向对象的”（object oriented）。但由于有了 PyOpenGL 及 pyglet 等程序库，所以我们能够把用 C 语言写成的 OpenGL 程序直接移植到 Python，移植的时候还可以调用 OpenGL 接口中的所有功能。此外，用 pyglet 来处理事件及创建窗口也是非常方便的，而 PyOpenGL 的优点则在于它能同 Tkinter 等许多 GUI 工具包相集成。

## 结 束 语

本书讲解了许多实用的技巧，也介绍了许多有用的程序库，但愿大家在阅读过程中能够产生一些想法，并得到一些启发，从而用 Python 3 ([www.python.org](http://www.python.org)) 编写出更优秀的程序来。

Python 语言正变得越来越流行，而且其应用领域也在逐渐扩大，世界各地的开发者都开始用这门语言来写程序了。从编程范式上来说，Python 支持过程式、面向对象式及函数式三种风格；从语法上来看，Python 代码是清晰、简洁而协调的，因此，Python 可称作理想入门语言。同时，对于资深开发者来说，它又是一门优秀专业语言，比方说，Google 公司近年来的表现就已经证明了这一点。Python 之所以能成为一门专业语言，是因为它既支持“快速开发”(rapid development)，又能够方便地调用各种功能强大的程序库。这些程序库未必都是 Python 库，它们也可以是用 C 语言来开发的，还可以是用遵从“C 语言调用约定”的其他编译型语言来开发的。

对 Python 语言的学习和探索是永无止境的。这门语言既很好地满足了初学者的需要，又提供了许多高深的功能，即便是对编程语言要求非常高的专家也会觉得满意。从授权协议及源代码的可用性角度来看，Python 是一门“开放”(open)的语言，但它的开放却并不止于此，你甚至可以直接深入字节码层面来查看其内部结构。另外，对于愿意贡献源代码的开发者来说，Python 社区的门当然是敞开的([docs.python.org/devguide](http://docs.python.org/devguide))。

编程语言可能有上千种，但真正被广为接受的却并不多，Python 就是其中之一，而且目前非常流行。笔者从事计算机研究已有几十年，也用过了许多语言，但经常对雇主所指定的那些语言感到失望。虽然从那些语言里面学到或听别人说到一些非常优秀的特性，但笔者和其他计算机研究人员一样，也认为那些语言都有各种各样的问题，而且用起来不是

很方便，于是笔者打算创建这么一种语言，它既能克服其他语言的缺点，又能将其他语言的优点全部囊括进来。可是，每次构思这门“理想语言”的时候，笔者都会意识到：它其实就是 Python。尽管 Python 本身也有个别地方设计得有点儿别扭（比如对常量的支持，对“可选的类型检查”（optional typing）的支持，以及对访问控制（也就是私有属性）的支持），但总的来说，笔者觉得自己无须再去发明“理想的编程语言”了，直接使用 Python 就好。感谢吉多·范罗苏姆（Guido van Rossum）先生以及诸位曾经或将要给 Python 贡献源代码的开发者，多谢大家为这个世界创造出这样一门无比强大而又实用的编程语言，Python 及其“生态系统”（ecosystem）在每个领域都表现得相当出色。用 Python 来编程，真是种享受。

## 附录 B

# 参考书目摘录<sup>⊖</sup>

### *C++ Concurrency in Action: Practical Multithreading*

Anthony Williams (Manning Publications, Co., 2012, ISBN-13: 978-1-933988-77-1)

这是一本份量很重的 C++ 并发教程，它描述了许多可能会影响并发程序的问题与陷阱，并给出了避免这些状况的办法。这些内容对其他编程语言同样适用。

### *Clean Code: A Handbook of Agile Software Craftsmanship*

Robert C. Martin (Prentice Hall, 2009, ISBN-13: 978-0-13-235088-4)

该书讲解了编程中的许多“战术问题”(tactical issue)，诸如怎样起名、怎样设计函数、怎样重构等。书里提出的很多想法应该能够帮助开发者改善其代码风格，并写出更易维护的程序。(书中范例是用 Java 语言描述的。)

### *Code Complete: A Practical Handbook of Software Construction, Second Edition*

该书会告诉大家如何构建坚实的软件，它不仅谈了编程语言的细节，而且还切入到编程思想、编程原则与编程实践领域。书中包含大量范例，可促使程序员更为深入地思考他们所从事的程序设计工作。

### *Design Patterns: Elements of Reusable Object-Oriented Software*

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995, ISBN-13: 978-0-201-63361-0)

该书是近些年最有影响力的编程教材之一。在日常编程中，设计模式既非常迷人，又特别实用。(书中范例大多以 C++ 代码写成。)

---

<sup>⊖</sup> 与中文版有关的信息为译者所加。——译者注

(《设计模式——可复用面向对象软件的基础》, 李英军 马晓星 蔡敏 刘建中 等译, 吕建 审校, 机械工业出版社, 2000 年)

*Domain-Driven Design: Tackling Complexity in the Heart of Software*

Eric Evans (Addison-Wesley, 2004, ISBN-13: 978-0-321-12521-7)

这本软件设计教程写得非常有趣, 尤其适用于多人参与开发的大型项目。该书的核心内容是: 创建及完善一套能够反映系统设计思路的领域模型, 并创立一门“通用语言”(ubiquitous language), 使得包括软件工程师在内的每一位系统设计人员都可以借此来交流想法。

*Don't Make Me Think!: A Common Sense Approach to Web Usability, Second Edition*

Steve Krug (New Riders, 2006, ISBN-13: 978-0-321-34475-5)

这本短小而实用的教程根据研究成果和经验来讲解“网页易用性”(web usability)。书中思想浅显易懂, 可以套用在任何规模的网站上。

*GUI Blooper 2.0: Common User Interface Design Don'ts and Dos*

Jeff Johnson (Morgan Kaufmann, 2008, ISBN-13: 978-0-12-370643-0)

这本书古怪的书名似乎有点儿吓人, 其实它是本严肃的教程, 所有 GUI 程序员都应该来看看。你未必会接受作者所提的每一项建议, 但是看完全书之后, 你在设计用户界面时就会变得更加谨慎, 判断力也会变得更加敏锐。

(《GUI 设计禁忌 2.0》, 盛海艳 等译, 机械工业出版社, 2008 年)

*Java Concurrency in Practice*

Brian Goetz, et. al. (Addison-Wesley, 2006, ISBN-13: 978-0-321-34960-6)

该书精彩地讲解了 Java 并发编程。书中的很多并发编程技巧也适用于其他语言。

(《Java 并发编程实战》, 童云兰 等译, 机械工业出版社, 2012 年)

*The Little Manual of API Design*

Jasmin Blanchette (Trolltech/Nokia, 2008)

这是本很薄的手册, 可以从 [www21.in.tum.de/~blanchet/api-design.pdf](http://www21.in.tum.de/~blanchet/api-design.pdf) 免费下载。书里讲述一些与 API 设计有关的思路和见解, 其中大部分范例都取自 Qt 工具包。

(《API 设计小手册》, vrcats 译, <http://moto.debian.tw/viewtopic.php?f=28&t=14492>)

*Mastering Regular Expressions, Third Edition*

Jeffrey E.F. Friedl (O'Reilly Media, 2006, ISBN-13: 978-0-596-52812-6)

该书是学习正则表达式的标准教程, 它透彻地讲解了许多实用的范例, 读者查阅起来也非常方便。

*OpenGL® SuperBible: Comprehensive Tutorial and Reference, Fourth Edition*

Richard S. Wright, Jr., Benjamin Lipchak, and Nicholas Haemel (Addison-Wesley, 2007, ISBN-13: 978-0-321-49882-3)

没有 3D 图形开发经验的程序员很适合用这本 OpenGL 教程来学习 3D 绘图技术。书中的范例是用 C++ 语言写的, 但由于 pyglet 及其他 Python OpenGL 绑定库都“非常忠实地

再现了”(pretty faithfully reproduce) OpenGL API, 所以这些范例只需稍加改编, 即可适用于 Python。

*Programming in Python 3: A Complete Introduction to the Python Language, Second Edition*

Mark Summerfield (Addison-Wesley, 2010, ISBN-13: 978-0-321-68056-3)

只要你会用常见的过程式或面向对象式语言来编程 (Python 2 当然也算), 那么就可以通过该书来学习 Python 3。

*Python Cookbook, Third Edition*

David Beazley and Brian K. Jones (O'Reilly Media, 2013, ISBN-13: 978-1-4493-4037-7)

该书全面讲解了 Python 3 编程, 里面有很多有趣而实用的想法。它非常适合与本书搭配着看。

*Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*

Mark Summerfield (Prentice Hall, 2008, ISBN-13: 978-0-13-235418-9)

该书讲解如何使用 Python 2 及 Qt 4 工具包进行 GUI 编程。Python 3 版本的范例可从作者网站下载。虽说是为 PyQt 而写, 但其中绝大部分内容也适用于 PySide。

*Security in Computing, Fourth Edition*

Charles P. Pfleeger and Shari Lawrence Pfleeger (Prentice Hall, 2007, ISBN-13: 978-0-13-239077-4)

该书风趣而务实地讲解了计算机安全领域的诸多事项, 解释了攻击原理及防御策略。

*Tcl and the Tk Toolkit, Second Edition*

John K. Ousterhout and Ken Jones (Addison-Wesley, 2010, ISBN-13: 978-0-321-33633-0)

该书是 Tcl/Tk 8.5 的标准教程。Tcl 与传统的编程语言不同, 它的“语法相当随意”(syntax-free), 而 Python/Tkinter 程序的开发者通常却又需要查看 Tcl/Tk 开发文档, 于是可以通过该书来学会如何阅读这些文档。

无论是有经验的老程序员，还是正在寻求自我提升的新手，都能通过书中的宝贵建议及实用范例来提高编程技能。作者引领大家从不同的角度思考问题，介绍各种开发工具并详细讲述各项开发技巧，使大家能够创建出更为高效的解决方案。

—— Doug Hellmann, DreamHost公司资深开发者

只要你有一定的Python编程经验，本书就能帮助你改进Python程序的代码质量、稳定性、运行速度、可维护性及可用性。

Mark Summerfield是Python开发者社区的知名技术作家，在本书中，他借助高质量的范例代码和3个完整案例研究，全方位系统讲解如何借助设计模式写出优雅的代码，如何通过并发技术及编译过的Python程序（也就是Cython）来提升处理速度，如何进行高级网络编程，以及如何绘制图形界面与三维图形，并且还解释了为何某些面向对象的设计模式在Python里用不到。此外，作者还用多个实例证明Python并不是一门效率低下的语言——只要善用并发和Cython技术，就能完全发挥多核CPU的优势，从而编出运行速度非常快的Python程序来。

本书涉及的所有代码都已在Linux操作系统里测试过，而且绝大部分代码也在OS X及Windows系统中测试过。

## 本书主要内容：

- 在Python程序中有效地利用创建型设计模式、结构型设计模式和行为型设计模式的优势。
- 用多进程技术、多线程技术和concurrent.futures模块来编写并发式Python程序。
- 在无须手工加锁的前提下，使用“线程安全的队列”及future来避免并发环境中的各种问题。
- 用xmlrpclib和RPyC等高级模块来简化网络编程。
- 用Cython、基于C的Python模块、性能分析工具等技术来提升Python代码的执行速度。
- 用Tkinter创建新式GUI应用程序。
- 通过pyglet和PyOpenGL来调用OpenGL API，以发挥出计算机硬件所具备的强大图形处理能力。

PEARSON

[www.pearson.com](http://www.pearson.com)

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：[www.hzbook.com](http://www.hzbook.com)

网上购书：[www.china-pub.com](http://www.china-pub.com)

数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)



上架指导：计算机/程序设计

ISBN 978-7-111-47394-7



9 787111 473947 >

定价：69.00元