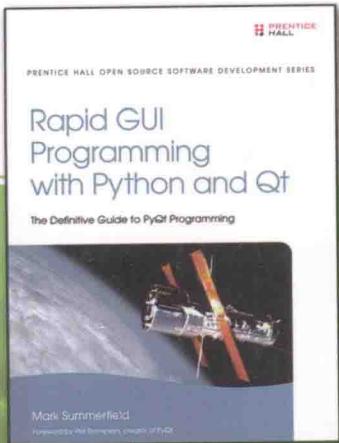


# Python Qt GUI 快速编程 —— PyQt 编程指南

Rapid GUI Programming with Python and Qt  
The Definitive Guide to PyQt Programming



[英] Mark Summerfield 著

闫峰欣 黄琳雅 王军锋 等译



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

# Python Qt GUI 快速编程

## ——PyQt 编程指南

Rapid GUI Programming with Python and Qt  
The Definitive Guide to PyQt Programming

[英] Mark Summerfield 著

闫锋欣 黄琳雅 王军锋 等译

电子工业出版社  
Publishing House of Electronics Industry  
北京 · BEIJING

## 内 容 简 介

本书主要讲述如何利用 Python 和 Qt 开发 GUI 应用程序的原理、方法和关键技术。本书共分四个部分：第一部分主要讲述 Python 基础知识，第二部分通过三个例子给出 PyQt GUI 应用程序的初步印象，第三部分深入讲述窗口部件布局、事件处理、窗口部件子类化、Qt 图形架构和 Qt 的模型/视图等内容，第四部分介绍国际化、网络化和多线程化等内容。

本书结构合理，内容翔实，适合于对 Python、Qt 和 PyQt 编程感兴趣的科教人员和广大的计算机编程爱好者阅读，也可作为相关机构的培训教材。

Authorized translation from the English language edition, entitled Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming, 9780132354189 by Mark Summerfield, published by Pearson Education, Inc., Copyright © 2008 Pearson Education Inc.

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 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2016.

本书中文简体字版专有版权由 Pearson Education(培生教育出版集团)授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

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

版权贸易合同登记号 图字：01-2015-1609

## 图书在版编目(CIP)数据

Python Qt GUI 快速编程：PyQt 编程指南 / (英) 马克·萨默菲尔德 (Mark Summerfield) 著；闫峰欣等译。  
北京：电子工业出版社，2016. 9

书名原文：Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming  
ISBN 978-7-121-29806-6

I. ①P… II. ①马… ②闫… III. ①软件工具—程序设计—指南 IV. ①TP311. 561-62

中国版本图书馆 CIP 数据核字(2016)第 203337 号

策划编辑：冯小贝

责任编辑：李秦华

印 刷：三河市华成印务有限公司

装 订：三河市华成印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：28.75 字数：811 千字

版 次：2016 年 9 月第 1 版

印 次：2016 年 9 月第 1 次印刷

定 价：79.00 元

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

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

本书咨询联系方式：fengxiaobei@phei.com.cn。

## 译者序

Python 是一种面向对象、解释型程序设计语言，由 Guido van Rossum 于 1989 年发明并于 1991 年发布第一个公开发行版。Python 简洁而清晰的语法，丰富而强大的库，使其能够将其他语言编写代码模块（尤其是 C/C++）轻松联结在一起，从而在不影响程序性能的前提下，大大降低整个应用程序的开发成本和资源占用。Python 自 2004 年以来，已成为全球最受欢迎的程序设计语言之一，用户数呈线性激增，被 TIOBE 计算机编程语言排行榜评为 2010 年的年度语言。Python 语言在卡耐基·梅隆大学、麻省理工学院、清华大学等国内外高校和科研机构的用户众多，也进而推动了 Python 的快速发展。

Qt 是一个性能卓越的跨平台 C++ 图形用户界面应用程序开发框架。自 1991 年由奇趣科技（Trolltech）推出至今的 25 年间，深受业界赞誉，并先后于 2008 年、2012 年被诺基亚（Nokia）和大奇（Digia）收购，实现了由单一的桌面计算机和嵌入式应用领域到全 IT 行业的转变。2014 年 4 月，Qt 开发团队正式发布跨平台的集成开发环境 Qt Creator 3.1.0，实现了对 iOS 的完全支持，新增了 WinRT、Beautifier 等插件，集成了基于 Clang 的 C/C++ 代码模块，调整了对 Android 的支持，实现了 iOS、Android、WP 的全面支持。目前，Qt 的用户已经涵盖了全球众多知名厂商，如 Google、Adobe、IBM、华为、中国移动等，而诸如 Google Earth、AutoDesk Maya、Opera、KDE、Skype 这样的应用程序大家也都是耳熟能详的。

PyQt 是 Qt 与 Python 的成功融合，或者也可以认为 PyQt 是 Qt 库的 Python 版本。PyQt 初次发布于 1998 年，创始人是 Phil Thompson。PyQt 的版本包括支持 Qt 1 到 Qt 3 的 PyQt3，支持 Qt 4 的 PyQt4 和现今支持 Qt 5 的 PyQt5。不过，由于 Qt 开发团队已经明确宣布自 2015 年年底后不再支持 Qt 4，所以对于 PyQt 的新手来说，最好能够直接从 PyQt5 开始学习。

本书是迄今为止最受公众认可的 PyQt 编程学习用书之一。作者 Mark Summerfield 在 Qt 公司初创时期就任职于此，近年来也一直作为 Qt 和 Python 开发的知名代码贡献者，在两者中均拥有良好的经验。Mark 在撰写本书的过程中，就不断跟踪 Python 和 Qt 的实时发展动态，因而无论是书中的例子和风格，还是课后习题的设置，始终秉承传授编程思想和原理方法为主、解决和分析技术难点为辅的写作风格，因而他所撰写的多部作品都获得了有软件业界“奥斯卡”之称的“震撼奖”（Jolt Award）。本书与获奖作品《C++ GUI Qt 4 编程》一书的写作风格类似，案例设置通俗易懂，因而是学习 PyQt 不可多得的一本好书。

感谢电子工业出版社的冯小贝编辑。为了能够把握书中的关键技术和发展，译者不得不多方求证、字斟句酌，也形成了近似电影《疯狂动物城》中“闪电”先生的工作模式和工作效率，译稿一拖就是一年多的时间。不过，这一年多的时间中，我核对了书中的每一处链接，验证了它们的有效性；依据作者的勘误信息，订正并更新了本版图书中的相应错误内容，以确保文字的正确性和可读性。因此，无论使用的 Qt 4 还是 Qt 5，无论是 Python 2.x 还是最新的 Python 3.5.x，绝大部分的内容都可以运行无误。

还要感谢参与本书翻译和审校工作的各位战友们，感谢你们的鼓励和支持。大家的工作分工是：西安交通大学的黄琳雅翻译了第 10 章、第 12 章和第 13 章，北京工商大学的张君施翻

译了第 8 章、第 9 章、第 15 章和第 17 章，西南科技大学的王军锋翻译了第 1 章，西北农林科技大学的张雷锋翻译了附录 B 和附录 C，浙江大学的薛一翻译了第 11 章，广东技术师范学院的刘溪翻译了第 14 章、第 16 章、第 18 章和 19 章，西北农林科技大学的闫峰欣翻译了本书第 2 章至第 7 章以及附录 A、前言和致谢等剩余部分。我们还邀请了北京交通大学的王海波、武汉深之度科技有限公司的丁江锋、山东济南初创公司的王翔凯和徐景亮作为外部审稿人，他们的细致和耐心，为我们的工作增色不少。

感谢农业部现代农业装备重点实验室开放课题(项目编号:201603002)和中央高校基本科研业务费项目(项目编号:Z109021423)为译者提供了宽松且安心舒适的工作环境。

书中所用到的示例程序的源代码可从原书站点 [www.qtrac.eu](http://www.qtrac.eu)(英文)下载，也可直接从站点 [www.qtcn.org/pyqtbook](http://www.qtcn.org/pyqtbook)(中文)下载。有关本书的讨论和勘误信息，也会及时在 [www.qtcn.org](http://www.qtcn.org) 网站公布，并在此向网站负责人 XChinux 表示感谢。

由于书中概念和术语数目繁多，加之译者水平所限，译稿中难免存在曲解或误解作者原意的地方，恳请读者谅解。

闫峰欣

2016 年 4 月 24 日

# 序

作为 PyQt 的创始人，非常高兴能够看到本书终于编写完成了。尽管是本书的技术审稿人之一，还是很高兴地要承认，自己的确从书中学到了不少东西。

PyQt 文档中涵盖了 PyQt 中各个层次的全部 API 类。本书则用来讲述如何使用这些类，以及如何将这些类组合起来创建对话框、主窗口和各类应用程序（它们不仅看起来美观漂亮而且功用良好，没有任何不良缺陷），使得大家乐于使用这一编程语言。

我最喜欢这本书的地方在于，即使是用来说明简单知识点的样例都显得极不平凡，并能够以其正确的方式给出极富潜力的应用方法。这些不寻常的方法，将可馈赠那些打算把 PyQt 用于开发更大维度、更高品质应用程序的读者们。

我从事 PyQt 的故事可以追溯到 20 世纪 90 年代。当时我用过一段时间的 Tcl/Tk，但觉得 Tk 应用程序看起来并不美观，尤其是当在第一版 KDE 中看到它运行时所做的那些事时，就打算要使用 Python，于是我认为，是把语言的变化和 GUI 库的变化结合起来的时候了。

起初，使用了一些基于 SWIG 写成的封装类，但随后就说服了自己，应该自己做一套更适合的封装工具<sup>①</sup>。工作就从创建 SIP 开始，并在 1998 年 11 月发布了支持 Qt 1.41 的 PyQt 0.1 版。开发工作自此开始不断正规起来，不仅与新发布的 Qt 保持一致，而且还在不断拓展 PyQt 的应用范围，例如，各类额外的支持工具和改良后的文档。到 2000 年时，PyQt 2.0 就已经可以在 Linux 和 Windows 上同时支持 Qt 2.2 了。对于 Qt 3 的支持始于 2001 年，而 2002 年就支持 Mac OS X 了。PyQt4 系列始于 2006 年 6 月的 PyQt 4.0，它可以支持 Qt 4。

我的基本目标是要让 Python 和 Qt 能够协同工作，这种工作方式要让 Python 编程人员觉得非常自然，同时允许他们能够以 C++ 来做那些像在 Python 中一样可以想做的任何事。达到这一点的关键在于 SIP 的开发工作。这就给予了一个特殊的代码生成器，让我能够完全控制并确保 Python 和 Qt 是始终如一的。

开发和维护 PyQt 的必要步骤目前都已建立完毕。大多数工作现在都可自动完成，这就意味着，与 Trolltech 公司的 Qt 新版本保持同步已不再像从前那样显得是个问题，并可相信，PyQt 在未来几年中会依旧向前发展<sup>②</sup>。

如今，非常欣慰于能够目睹 PyQt 社区在过去的数年中不断成长。如果此书能够把您引入 PyQt 的天地，那么欢迎您！

—Phil Thompson

温伯恩，多赛特，英国

① SWIG 是一种简化脚本语言与 C/C++ 接口的开发工具。简而言之，SWIG 是一个通过包装和编译 C 语言程序来达到与脚本语言通信目的的工具——译者注。

② Trolltech 是挪威的一家公司，Qt 最先源自该公司，先后被诺基亚和大奇 (Digia) 公司收购。国内一般将其称为奇趣科技公司——译者注。

# 前　　言

本书主要讲述如何利用 Python 程序语言和 Qt 应用程序开发框架来开发 GUI 应用程序。仅需要的一点必备知识是，要能够使用一些面向对象编程语言来编程，诸如 C++、C#、Java，当然，也包括 Python 自己。在有关 Rich 文本的章节中，可能还会假定你了解了 HTML 和正则表达式的一些知识；而在数据库和多线程的那些章节中，也还会假定你已了解了相关话题的基本知识。至于 GUI 编程方面的知识就不需要了，因为书中会包含其相关的所有关键概念。

本书将对那些以专业编程为其工作的那些人大有帮助，无论是专业的软件开发人员，还是其他行业的编程人员，如科学家、工程师等，都需要通过编程来支撑自己的工作。本书也同样适用于那些大学生和研究生，他们在课题或者研究工作中上经常需要大量的计算要素。为帮助同学们理解书中讲授的内容，还特地提供了一些练习题（并给出了这些习题的解答思路）。

Python 有可能是世界上广泛应用的最易学习、最漂亮的脚本语言了，而 Qt 则有可能是开发 GUI 应用程序最好的库。Python 和 Qt 的结合，称为 PyQt，使得在所有支持它们的平台，如 Windows、Linux、Max OS X 和类 UNIX 系统的各个现有版本上，开发应用程序并且不做任何改变地运行程序成为可能。无须编译得益于 Python 这一解释性脚本语言，而针对不同操作系统都无须更改源代码的好处则源于 Qt 的抽象方式，能够避开那些与平台相关的细节。我们要做的仅仅就是将写好的一个或者多个源代码文件复制到安装了 Python 和 PyQt 的目标机器上运行即可。

如果对 Python 一无所知，那么欢迎您！您将会探索一门易读易写的语言，其语法简洁而不神秘。Python 支持很多编程范式，但由于我们侧重于 GUI 编程，因而将会在除本书最前面的几章之外的各处均使用面向对象的编程方法。

Python 是一种非常富有表现力的语言，这就意味着，要完成具有相当功能的应用程序，用 Python 编写的代码要比使用其他诸如 C++ 或者 Java 等语言的代码少得多。这就使得通过文本来展示一些小而完整的样例成为可能，也使得 PyQt 成为能够快速并且简易开发 GUI 应用程序的一件理想工具，无论是用做原型设计还是用做最终的产品。

由于本书的重点是有关 GUI 编程的，第一部分会像其他 PyQt 文献一样也给出一个 Python 快速入门教程。在本书的第二部分、第三部分和第四部分，都与 PyQt 相关并会假定读者已经可以用 Python 编程，无论是前期的经验，还是从第一部分中阅读而来的。

在编程时，当有数种可能采用的可行方法时就会经常遇到决策点（decision point）。参阅书籍和网上文献来识别出可用的那些类、方法和函数，还可以参考某些情况下所给出的示例，但那样的文献很少会给出一个上下文背景（broader context）。本书则会给出必要的上下文背景，强调那些用于 GUI 编程的决策点，深入解析器优劣，以便让读者自行决断特定情况下的正确策略。例如，在创建对话框的时候，应该是使用模态（modal）对话框还是非模态（modeless）对话框（请参阅第 5 章中有关这一主题的解释说明和推荐策略中的内容）。

PyQt 可用于编写各类 GUI 应用程序，从会计类应用程序到被科学家和工程师所使用的各种可视化工具。例如，在图 1 中，给出了一个示例 Eric4，这是一个使用 PyQt 编写强大集成开发环境。编写一个仅有 10 行代码长短的 PyQt 应用程序，或者也有可能是编写一个拥有

1000 ~ 10 000 行 PyQt 代码的中型工程，都已司空见惯。某些商业公司利用从一个人到数十人不等的编程团队，已经构建出超过 100 000 行代码的 PyQt 应用程序。许多公司内部使用的工具就是用 PyQt 编写的，但由于这些工具通常直接用于获利的，相关公司一般不会将自己使用 PyQt 的事情公之于众。PyQt 也会广泛用于开源世界，包括游戏、应用设施、可视化工具和各类集成开发环境（IDE）等都会用到它。

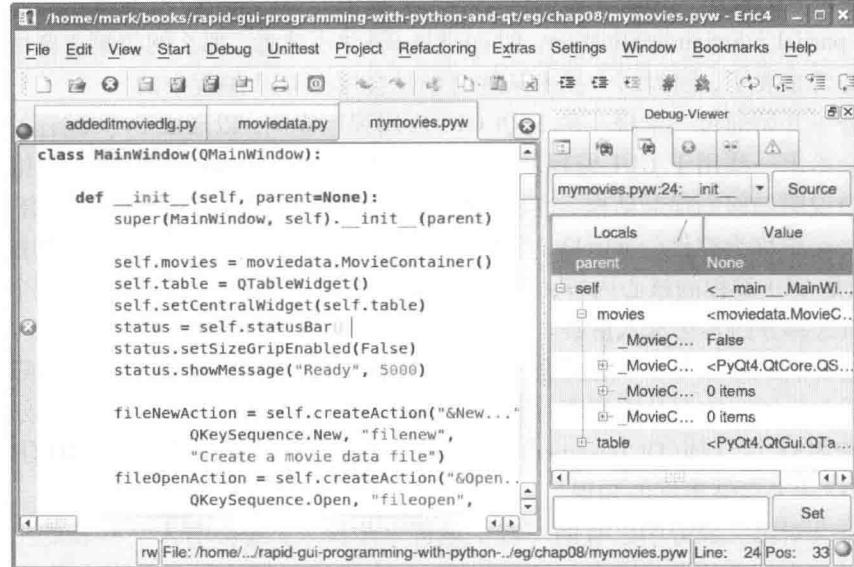


图 1 Eric4 集成开发环境，是一个 PyQt4 应用程序

本书重点放在 PyQt4 和用于 Qt 4 C++ 应用程序开发框架<sup>①</sup>的 Python 绑定上。会以 10 个 Python 模块的形式给出 PyQt4，在这些模块中会大约含有 400 个类和 6000 个左右的方法和函数。所有样例程序均已使用 Python 2.5、Qt 4.2 和 PyQt4.2 在 Windows、Linux 和 Mac OS X 上进行了测试。此外，在 Windows 和 Linux 上，还使用 Qt 4.3 和 PyQt4.3 进行了测试。某些情况下，还有可能可以回溯到以前更早期的版本，但还是建议尽量使用最新版本的 Python、Qt 和 PyQt。

Python、PyQt 和 Qt 可免费用于非商业用途，但 Python 所采用的授权则不同于 PyQt 和 Qt 的授权。基于较为宽泛的授权条款就可以获得 Python，并可将其用于开发商用或者非商用应用程序。而 PyQt 和 Qt 则使用双重授权模式：这就从本质上使得它们只能用于开发非商用应用程序——相应需要使用一种可接受的开源授权，如 GPL (GNU General Public License, GNU 通用公共授权)；或者用于开发商业程序，此时，就必须分别购买商业版的 PyQt 授权和商业版的 Qt 授权。

## 本书的结构

本书共分四个部分。第一部分是快速转换课程，主要面向那些熟悉面向对象语言的非 Python 编程人员，尽管会有一点 PyQt 的内容。这是因为核心的 Python 语言非常简单并且相当

<sup>①</sup> 对于较老的 Qt 3 库也有 Python 绑定，但在新的工程项目中没有理由再去使用该库，特别是自 Qt 4 以来，又提供了更多的功能和更为简单的使用方法。

小巧，这些章节会讲述一些 Python 的基础知识，以便能够为编写真正的 Python 应用程序做好准备。

如果认为自己可以通过阅读一些 Python 程序就能够了解 Python 语法的话，不妨直接跳过第一部分并转入第二部分前面有关 GUI 编程的那些章节。第二部分的开头章节会向前引用一些位于第一部分相关页码的内容，以便为读者选择该方法提供支撑依据。然而，即使是熟悉 Python 的那些读者，也还是建议能够阅读一下第 1 章中 `QString` 的相关内容。如果您对偏函数应用程序 (partial function application，即 currying<sup>①</sup>) 并不熟悉，那么阅读第 2 章中有关小节的相关内容就非常重要了，因为这一技术某些时候是会用在 GUI 编程中的。

第二部分一开始通过三个极小的 PyQt GUI 应用程序样例，以形成 PyQt 编程的一个初始印象。其中也会给出一些用于 GUI 编程基本概念的说明，包括 PyQt 的高级信号和槽的通信机制。第 5 章会说明如何创建对话框，也会说明如何创建和布局对话框中的各个窗口部件 [widget，在 Windows 中称为控件 (control)，是用来构成用户界面的一些图形元素，如按钮、列表框等]。对话框是 GUI 编程的核心内容：多数 GUI 应用程序都有单一的主窗口、数个或者数十个对话框，因而这部分内容会深入讲解。

对话框之后就到了第 6 章，这里会涉及一些主窗口，包括菜单、工具栏、停靠窗口和键盘的快捷键，还会包括一些有关应用程序设置的加载和保存方面的内容。第二部分的剩余章节会讲述如何使用 Qt 设计师 (Qt Designer) 和 Qt 的可视化设计工具，还会讲述如何用二进制、文本和 XML 的格式来存储数据的知识。

第三部分会对第二部分中已有的一些主题进一步深入，还会引入一些新的主题。第 9 章会讲述如何以相当烦琐的方式来布局窗口部件，并会讲述如何处理多文档。第 10 章会讲述一些低级事件处理器 (event handler)，还会讲述如何使用剪切板 (clipboard) 和拖拽、文本、HTML 及二进制数据。第 11 章给出了如何对已有的窗口部件进行修改和子类化，还会讲述如何用 scratch<sup>②</sup> 创建一些全新的窗口部件，并可以完全控制它们的外观和行为。这一章还会给出一些图形基础的示例。第 12 章会讲述如何使用 Qt4.2 的图形视图架构，它尤其适合用来处理大量的独立图形对象。Qt 的 HTML Rich 文本引擎处理能力会在第 13 章中讲述。这一章还会包括一些打印到纸张和 PDF 文件的内容。

第三部分中与模型/视图 (model/view) 相关的章节有两章：第 14 章会介绍该主题，并会展示一些如何使用 Qt 内置视图的方法，用以创建自定义数据模型和自定义代理 (delegate)；第 15 章会讲述如何使用模型/视图架构来执行数据库编程。

第四部分继续模型/视图这一主题，会在第 16 章中给出三个不同的模型/视图的用法。第 17 章的 17.1 节会介绍一些用于提供在线帮助的技术，17.2 节会讲述如何来国际化一个应用程序，包括如何来使用 Qt 的翻译工具创建翻译文件等。Python 标准库提供了一些用于网络化和多线程化的自有类，但在第四部分的最后两章，还是会给出一些网络化和多线程化时使用了 PyQt 的类。

附录 A 会说明哪些地方可以获得 Python、PyQt 和 Qt，并说明如何在 Windows、Mac OS X

---

① 在计算机科学领域，偏函数应用程序是指通过固定原函数的一部分参数生成新函数的过程，新函数的参数数目会少于原函数；currying 是把接受多个参数的函数转换成接受一个单一参数（最初函数的第一个参数）的函数，返回接受余下的参数并返回结果的新函数。currying 和偏函数应用程序有些关系，实为两个并不完全相同的概念——译者注。

② Scratch 是一种新的编程语言，据说是面向中小学生，可认为是一种基于 Squeak 并用其实现的可视化程序设计语言——译者注。

和 Linux 上安装它们。如果安装了 PyQt 并试着通过阅读一些示例代码做些练习，就会发现 PyQt 要更易于学习。附录 B 给出了一些截图和一些选取的 PyQt 窗口部件的简单介绍，这对于那些 GUI 编程新手们会很有帮助。附录 C 展示了一些 PyQt 的关键类的层次示意图，这在要知道 PyQt 提供了哪些类以及它们和谁相关的时候很有用。

如果之前从来就没有用过 Python，那么就应当依次阅读第 1 章到第 6 章。如果已经了解 Python，那么至少要阅读有关字符串方面的规定，然后就可以跳过第 2 章了（除了 2.1 节之外，因为这些内容应当熟练掌握）。要相信你对 lambda 和偏函数应用程序（lambda and partial function application）并不反感，这些内容都在第 2 章讨论。第 3 章很有可能也是可以略过的。然后就可以依次阅读第 4 章至第 6 章了。

一旦涉猎了前六章，那么就有了 Python 的基本知识和 PyQt 的基础。

如果打算知道如何使用可视化工具而不是纯粹的手工编码来创建对话框，那么第 7 章就很有用了，这样就可以节省出大量的时间。对于文件的处理，则至少要阅读过第 8 章的前三节。如果打算对文件进行读写，还需要阅读第 8 章的 8.4 节；而如果打算使用 XML 文件，那么就需要阅读 8.5 节。

对于第三部分，至少需要阅读第 10 章的 10.1 节，即事件处理，以及第 11 章的全部内容。第 12 章和第 13 章的 13.1 节会认为你已经阅读过 PyQt 事件处理的内容了，并同时还会认为也阅读了第 11 章。在这一部分中，第 9 章和第 14 章可以单独阅读，但第 15 章会认为你已经阅读了第 14 章。

对于第四部分，在阅读第 16 章之前，假设已阅读了第 14 章和第 15 章。而其他章节可单独阅读。

如果发现了文字或者示例中的错误，或者其他建议，请发邮件到 [mark@qtrac.eu](mailto:mark@qtrac.eu)，并请在标题栏中备注“PyQt book”字样。在本书的主页会列出一些勘误信息，其中也可以下载各个样例和练习题的答案，主页的网址是：<http://www.qtrac.eu/pyqtbook.html>。

如果打算参与 PyQt 社区，就非常值得加入邮件列表了。可以在 <http://www.riverbankcomputing.com/mailman/listinfo/pyqt> 找到该邮件列表存档的链接，以便可以看到邮件列表的形式，还可以找到加入邮件列表的表单。Python 还有一些邮件列表和其他的活跃社区。要查看它们，可以浏览 <http://www.python.org/community>。

## 致谢

要感谢很多人，那么就先从参与本书的这些人开始吧。

Jasmin Blanchette 是 Trolltech 公司的一名高级软件开发人员，也是一名 Qt 专家，还是一名好编辑和特立独行的作家。我曾与他一起合著过两本 C++/Qt 的书籍。Jasmin 提出了大量的建议和批评，从而极大地提高了本书的质量。

David Boddie 是 Trolltech 公司的一名文档经理，也是一名为 PyQt 做出过很多贡献的 PyQt 活跃开源开发人员。他的投入很大程度上为我保证了所需的各类东西，并会给出一些合理的次序。

Richard Chamberlain 是 Jadu 有限公司的合伙人和首席技术官（CTO），该公司主营内容管理。他的反馈和远见确保了本书能够在尽可能宽泛的领域内获得认可。他还帮助精炼和改良了那些用于示例和练习题中的代码。

Trenton Schulz 是一名 Trolltech 公司的开发人员，也是我之前几本书中不可多得的审阅人员。对于本书，他带来了大量的 Python 和 Qt 知识，对本书手稿给出了相当多的建议。此外，

他还指出了我所遗漏的一些错误。

Phil Thompson 是 PyQt 的创始人和维护人。他从一开始就很支持本书，甚至还将我们针对本书讨论而来的一些结果加入到 PyQt 的新特性和改进之中。他对改进本书提出了无数的建议，并纠正了书中的很多错误和误解。

尤其感谢 Samuel Rolland，让我能够轻松自在地使用他的 Mac 笔记本，安装 PyQt，测试示例和截图。

还要感谢 Python 的创始人 Guido van Rossum，同样要感谢为 Python 发展做出了巨大贡献的 Python 社区，特别是其各个类库，是那么有用，那么好用。

同样要感谢 Trolltech 公司，开发和维护着 Qt，特别感谢 Trolltech 公司中那些过去和现在的开发人员，他们中的很多人我都曾经与之快乐共事，并始终坚信 Qt 会是目前已有的最好跨平台 GUI 开发框架。

特别感谢 Lout 排版语言的创始人 Jeff Kingston。我在自己的全部书籍和大部分著述工程中都使用了 Lout。在这些年中，Jeff 依据用户的建议对 Lout 做出了很多改进并增加了许多新特性，其中就包括了许多我曾自问的东西。也要感谢 James Cloos，他是 DejaVu Sans Mono 字体（派生自 Jim Lyles 的 Vera 字体）及其压缩版的创始人，从该字体派生了本书所使用的 monospaced 字体。

从出版社来说，主编 Karen Gettman 是从最开始就非常支持本书的。特别感谢我的编辑 Debra Williams Cauley 的支持，使得整个出版流程尽可能顺畅自如。还要感谢 Lara Wysong 那么好地管理了出版工作，感谢校对人员 Audrey Doyle 做了那么多的细致工作。

最后，我想感谢我的妻子 Andrea。她的爱、忠诚和支持总是给予我力量和希望。

## 关于作者

**Mark Summerfield** 毕业于威尔士斯旺西大学 (the University of Wales Swansea)，他曾获得计算机科学专业的一等荣誉学位。在进入企业之前，曾进行过为期一年的研究生学习阶段。在加入 Trolltech 公司前的多年间，担任过多个公司的软件工程师。他有大约三年的时间担任 Trolltech 公司的文档管理产品经理，在这段时期内，创立并编写了 Trolltech 公司的技术杂志，*Qt Quarterly*，并与他人共同编写了 *C++ GUI Programming with Qt 3* 一书，之后又编写了 *C++ GUI Programming with Qt 4* 一书。Mark 创立了 Qtrac 有限公司 ([www.qtrac.eu](http://www.qtrac.eu))，在那里他作为独立撰稿人、编辑和技术顾问（特别是有关 C++、Qt、Python 和 PyQt）。

# 目 录

## 第一部分 Python 编程

<b>第1章 数据类型和数据结构 .....</b>	<b>2</b>
1.1 执行 Python 代码 .....	3
1.2 变量和对象 .....	4
1.3 数字和字符串 .....	7
1.3.1 整数和长整型 .....	8
1.3.2 浮点数和小数 .....	9
1.3.3 字节字符串、Unicode 字符串和 QString .....	11
1.4 集合 .....	18
1.4.1 元组 .....	18
1.4.2 列表 .....	20
1.4.3 字典 .....	23
1.4.4 集 .....	24
1.5 内置函数 .....	25
小结 .....	27
练习题 .....	28
<b>第2章 控制结构 .....</b>	<b>30</b>
2.1 条件分支 .....	31
2.2 循环 .....	33
列表解析和生成器 .....	37
2.3 函数 .....	37
2.3.1 生成器函数 .....	40
2.3.2 关键字参数的使用 .....	41
2.3.3 lambda 函数 .....	43
2.3.4 动态函数的创建 .....	43
2.3.5 偏函数应用程序 .....	44
2.4 异常处理 .....	46
小结 .....	51
练习题 .....	51
<b>第3章 类和模块 .....</b>	<b>54</b>
3.1 实例的创建 .....	55
3.2 方法和特殊方法 .....	57
3.2.1 静态数据、静态方法和装饰器 .....	61

3.2.2 例：Length 类 .....	62
3.2.3 集合类 .....	67
3.2.4 例：OrderedDict 类 .....	67
3.3 继承和多态 .....	72
3.4 模块和多文件应用程序 .....	76
小结 .....	78
练习题 .....	79

## 第二部分 GUI 编程基础

<b>第4章 GUI 编程简介 .....</b>	<b>82</b>
4.1 25 行的弹出式闹钟 .....	83
4.2 30 行的表达式求值程序 .....	86
4.3 70 行的货币转换程序 .....	90
4.4 信号和槽 .....	94
小结 .....	101
练习题 .....	102
<b>第5章 对话框 .....</b>	<b>103</b>
5.1 简易对话框 .....	104
5.2 标准对话框 .....	109
OK/Cancel 型模态对话框 .....	110
5.3 智能对话框 .....	115
5.3.1 非模态应用/关闭型对话框 .....	115
5.3.2 非模态的实时对话框 .....	119
小结 .....	121
练习题 .....	122
<b>第6章 主窗口 .....</b>	<b>123</b>
6.1 主窗口的创建 .....	124
6.1.1 动作和按键顺序 .....	127
6.1.2 资源文件 .....	129
6.1.3 创建和使用动作 .....	130
6.1.4 恢复和保存主窗口的状态 .....	135
6.2 用户动作的处理 .....	142
6.2.1 文件动作的处理 .....	142
6.2.2 编辑动作的处理 .....	147
6.2.3 帮助动作的处理 .....	149
小结 .....	150
练习题 .....	151
<b>第7章 使用 Qt 设计师 .....</b>	<b>152</b>
7.1 用户界面的设计 .....	154

7.2 对话框的实现 .....	161
7.3 对话框的测试 .....	165
小结 .....	166
练习题 .....	167
<b>第8章 数据处理和自定义文件格式 .....</b>	<b>169</b>
8.1 主窗口的职责 .....	170
8.2 数据容器的职责 .....	175
8.3 二进制文件的保存和加载 .....	179
8.3.1 用 QDataStream 读写 .....	179
8.3.2 使用 pickle 模块读写 .....	183
8.4 文本文件的保存和加载 .....	185
8.4.1 使用 QTextStream 读写 .....	186
8.4.2 使用 codecs 模块读写 .....	190
8.5 XML 文件的保存和加载 .....	191
8.5.1 XML 的写 .....	191
8.5.2 用 PyQt 的 DOM 类来读取和解析 XML .....	193
8.5.3 用 PyQt 的 SAX 类读取和解析 XML .....	195
小结 .....	198
练习题 .....	199

### 第三部分 中级 GUI 编程

<b>第9章 布局和多文档 .....</b>	<b>202</b>
9.1 布局策略 .....	203
9.2 Tab 标签页窗口部件和堆叠窗口部件 .....	204
9.3 窗口切分条 .....	211
9.4 单文档界面(SDI) .....	213
9.5 多文档界面(MDI) .....	219
小结 .....	227
练习题 .....	228
<b>第10章 事件、剪贴板和拖放 .....</b>	<b>229</b>
10.1 事件处理机制 .....	229
10.2 重新实现事件处理程序 .....	230
10.3 使用剪贴板 .....	235
10.4 拖放 .....	236
小结 .....	240
练习题 .....	241
<b>第11章 自定义窗口部件 .....</b>	<b>242</b>
11.1 使用窗口部件样式表 .....	242
11.2 创建复合窗口部件 .....	245

11.3 子类化内置窗口部件 .....	246
11.4 子类化 QWidget .....	247
11.4.1 例：分数滑块 .....	249
11.4.2 例：流体混合窗口部件 .....	255
小结 .....	260
练习题 .....	261
<b>第 12 章 基于项的图形 .....</b>	<b>262</b>
12.1 图形项的自定义和交互 .....	263
12.2 动画和复杂形状 .....	277
小结 .....	285
练习题 .....	286
<b>第 13 章 Rich 文本和打印 .....</b>	<b>287</b>
13.1 Rich 文本的编辑 .....	288
13.1.1 使用 QSyntaxHighlighter .....	288
13.1.2 Rich 文本的行编辑 .....	293
13.2 文档打印 .....	300
13.2.1 图片的打印 .....	302
13.2.2 使用 HTML 和 QTextDocument 打印文档 .....	302
13.2.3 使用 QTextCursor 和 QTextDocument 打印文档 .....	304
13.2.4 使用 QPainter 打印文档 .....	307
小结 .....	310
练习题 .....	311
<b>第 14 章 模型/视图编程 .....</b>	<b>312</b>
14.1 使用简便项窗口部件 .....	313
14.2 创建自定义模型 .....	320
14.2.1 实现视图逻辑 .....	320
14.2.2 实现自定义模型 .....	323
14.3 创建自定义委托 .....	329
小结 .....	334
练习题 .....	335
<b>第 15 章 数据库 .....</b>	<b>336</b>
15.1 连接数据库 .....	336
15.2 执行 SQL 查询 .....	337
15.3 使用数据库窗体视图 .....	341
15.4 使用数据库表视图 .....	345
小结 .....	356
练习题 .....	356

## 第四部分 高级 GUI 编程

第 16 章 高级模型/视图编程 .....	360
16.1 自定义视图 .....	360
16.2 泛型委托 .....	366
16.3 树中表达表格数据 .....	373
小结 .....	383
练习题 .....	383
第 17 章 在线帮助和国际化 .....	385
17.1 在线帮助 .....	385
17.2 国际化 .....	387
小结 .....	393
练习题 .....	393
第 18 章 网络应用 .....	394
18.1 创建 TCP 客户端 .....	396
18.2 创建 TCP 服务器 .....	400
小结 .....	404
练习题 .....	404
第 19 章 多线程 .....	406
19.1 创建线程服务器 .....	407
19.2 创建和管理次线程 .....	412
19.3 实现次线程 .....	418
小结 .....	422
练习题 .....	423
这并非结束 .....	424
附录 A 安装 .....	425
附录 B PyQt 的部分窗口部件 .....	437
附录 C 部分 PyQt 类的层次 .....	441

# 第一部分

## Python 编程

---

第1章 数据类型和数据结构

第2章 控制结构

第3章 类和模块

# 第1章 数据类型和数据结构

- 执行 Python 代码
- 变量和对象
- 数字和字符串
- 集合
- 内置函数

在这一章，会先从一个 Python 的转专业课程 (conversion course)<sup>①</sup>开始说明非 Python 编程人员是如何用 Python 来编程的。也会引入一些基础数据类型、数据结构和 Python 的编程语法。各处所用的方法主要是想强调一点，那就是理想的代码会和实际用到的代码一样，而不会针对 Python 在线文档 (<https://www.python.org>) 中已有的文档再次给出正式的定义和说明。

如果还没有安装 Python 和 PyQt，那么最好是安装一下：这样就可以测试一下与本书配套的那些示例代码了（可以从 <http://www.qtrac.eu/pyqtbook.html> 下载到这些代码<sup>②</sup>）。附录 A 给出了安装的详细步骤。安装 IDLE 集成开发环境这一软件的好处之一在于，安装时，它会与 Python 一起安装。

## IDLE 开发环境

完整安装 Python 会包括 IDLE 这一简单但却非常有用的集成开发环境。打开 IDLE（在 Windows 系统，可以依次点击开始（start）→所有程序（All Programs）→Python 2.x→IDLE；在 Mac OS X 系统，可以依次点击 Finder→Applications→MacPython 2.x→IDLE；对于 Linux 系统，可以在控制台中输入 run idle &）后，它会显示出 Python Shell 窗口。

如图 1.1 的截图所示，IDLE 会有一个类似 Windows 95 的怀旧界面。这是因为编写它用的是 Tkinter 而不是 PyQt。我们选用 IDLE 是因为 IDLE 会带有标准版 Python 且易学易用。如果打算使用功能更强大、更现代观感的集成开发环境，那么可能会更喜欢使用由 PyQt 写成的 Eric4，或者是其他可用的 Python 集成开发环境。然而，如果是 Python 新手，我们还是建议读者先从最简单的 IDLE 开始，而一旦对 PyQt 使用熟练后，再去尝试并再从其他集成开发环境中挑选出心仪的一个。当然，也可以根本无须使用任何集成开发环境，而是仅仅用一个纯文本编辑器和打印不同状态的方式来进行调试工作。

IDLE 提供了三个关键功能：功能一是，能够输入 Python 表达式和代码并可在 Python Shell 中直接看到结果；功能二是，提供了一个可将 Python 相关语法高亮显示的代码编辑器；功能三是，提供了一个可用于逐步深入代码并帮助认定和杀死 bug 的调试器。在测试那些简单的算

<sup>①</sup> 一些英国大学对某些专业提供转专业课程，这类课程是为了那些本科所学与日后期望的生涯发展不同的大学毕业生所设计的，其入学一般不需要相关的专业背景。课程视课程内容不同其学制也不同，短则 9 个月，长则可达两年，完成之后可以拿到证书（certificate）、文凭（diploma）或硕士学位（master degree）等不同学历资格——译者注。

<sup>②</sup> 与本书配套的国内站点为 <http://www.qtcn.org/pyqtbook/>——译者注。

法、代码片段和正则表达式时，Python Shell 会特别有用，当然，也可以把它当成是功能强大且使用灵活的计算器。

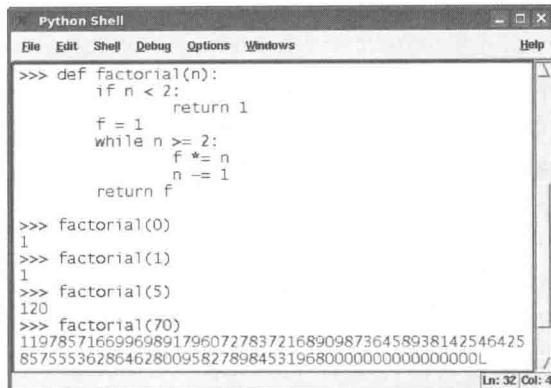


图 1.1 IDLE Python Shell 窗口

## 1.1 执行 Python 代码

在真正开始探索 Python 语言之前，需要知道如何执行 Python 代码。为此，这里会给出一个非常小的示例程序，其长度只有一行。

要编辑 Python 文件需要使用一个纯文本编辑器<sup>①</sup>。在 Windows 平台，可能会用到记事本程序 Notepad，但实际上 IDLE 包含了一个非常适合用于编辑 Python 代码的 Python 编辑器：只需在打开 IDLE 后，点击文件(File)→新窗口(New Window)。

我们会在 hello.py 文件中键入以下代码：

```
print "Hello World"
```

值得注意的是，这里并不需要任何分号(;)：在 Python 中，换行符同样还可以作为声明的分隔符。同理，字符串中也无须使用换行符“\n”，因为打印命令 print 会自动添加一个换行符，除非非要以逗号作为结尾。

假定已经将上述代码保存在文件 hello.py(如果使用的是 Windows，该文件在目录 C:\pyqt\chap01)中了，就可以打开一个控制台窗口(在 Windows XP 系统，可以单击开始→所有程序→附件→控制台，当然，有时候控制台也叫做命令提示符；或者是在 Mac OS X 中，从 /Applications/Utilities 目录下运行 Terminal.app)，然后切换到该目录，并像以下方式这样执行该程序：

```
C:\>cd c:\pyqt\chap01
C:\pyqt\chap01>hello.py
```

只要正确安装了 Python，Windows 就会识别.py 文件后缀并用 python.exe 去执行该文件。

<sup>①</sup> 本书中编写示例程序时用的都是 ASCII 字符，编码方式都是 Unicode。在 Python 程序中，字符串和注释的编码很可能会影响到 Latin-1、UTF-8 或者其他编码方式。这方面的有关说明，请见 Python 官方文档中“编码声明”(Encoding declarations)一节的内容。

这个程序会在控制台中如期打印出“Hello World”的字符<sup>①</sup>。

对于 Mac OS X 和 Linux，需要在控制台提示符后明确地键入解释器(interpreter)的名字和文件的名字才能运行上述代码，如下所示：

```
% python hello.py
```

如果已经安装了 Python 并加到了操作系统的 PATH 环境变量中，这一命令就会得以执行。相应地，对于 Linux 和 Mac OS X 用户，可以增加一行额外的注释 shebang(shell 执行)，用来告诉操作系统要使用 Python 解释器来运行该代码，只有两行代码的 hello.py 文件是这样的

```
#!/usr/bin/env python
print "Hello World"
```

如果在 Mac OS X 和 Linux 上运行，必须要正确地设置文件的权限。例如，在与该文件同目录的控制台提示符下，可以输入 chmod +x hello.py 来使该文件成为可执行文件。

Python 注释从“#”开始并直到该行最后才算结束。这就是说，在所有 Python 程序中添加 shebang 的做法会相当安全，因为在 Windows 中会自动忽略这一行，而在 Linux 则会告诉操作系统要使用 Python 解释器来执行该文件。在附录 A 中，讲述了有关 Mac OSX 系统中如何将 Python 解释器与 .py 和 .pyw 文件关联的方法。

至于说到 Python 程序的执行，其实质上是，Python 会先将 .py(或者 .pyw) 文件读取到内存中并对其进行解析，然后就可以得到一个能够继续执行的字节码程序(bytecode program)<sup>②</sup>。对于由该程序所导入的每一个模块，Python 都会先去检查是否已有预编译过的字节码程序(为 .pyo 或者 .pyc 形式的文件)，因为这里的程序会带有与其 .py 文件相对应的时间戳。如果有，Python 就会使用这些预编译字节码程序；否则，Python 就会解析该模块的 .py 文件，保存到 .pyc 文件中，再来使用这些刚生成的字节码程序。因此，与 Java 不同，无须明确指出该对哪些模块进行字节码编译，无论这些模块是由 Python 提供的，还是由我们自己编写完成的。而且，在大多数的 Python 安装中，作为安装程序的一部分，所提供的各个模块都已被编译过，以避免在任何需要运行它们的时候再去重新编译它们。

## 1.2 变量和对象

在大多数编程语言中，包括 C++ 和 Java，在使用前都需要声明每一个变量，给定变量的类型等。这种变量之所以称为静态类型(static type)，是因为编译器会在编译的时候知道每个变量的类型。与其他大多数的极高阶编程语言(very high level language)一样，Python 会使用不同的变量声明方法：各变量并没有严格的类型限制(是动态类型，dynamic type)，而且也不需要声明<sup>③</sup>。

通过创建和执行一个 Python 文件，就像在上一节 hello.py 中所做的那样，来学习 Python 的变量和标识符。不过，对于只尝试一些很小的代码片段，则根本就不需要单独创建

<sup>①</sup> Mac OS X 用户需要注意的是，无论何时，我们所说的控制台都是与 Mac Terminal 一样的。

<sup>②</sup> 所谓的字节码程序(bytecode program)，是一种由 .py 文件经过编译后生成的二进制文件，常写为 .pyc 格式的文件。.py 文件编译成 .pyc 文件后的加载速度有所提高。.pyc 是一种可由 Python 虚拟机执行的跨平台字节码，既是运行于虚拟机的解释指令，也是一种定义良好的中间表示，已成为当今网络软件和计算设备中广泛使用的重要技术。不同版本编译而成的 .pyc 文件是不同的，如 2.5 版本编译的 .pyc 文件，在 2.4 版本的 Python 中就无法执行——译者注。

<sup>③</sup> 静态类型(static type)和动态类型(dynamic type)的不同之处在于，是在编译期还是在运行期确定变量类型的：静态类型在编译期确定，动态类型则在运行期确定。静态类型的代表有 Java、C++、Haskell 等；动态类型的代表包括 Ruby、Python、Erlang 等——译者注。

一个文件。在 IDLE Python Shell 窗口中，可以直接在 >>> 提示符后输入要执行的那几行代码即可

```
>>> x = 71  
>>> y = "Dove"
```

在赋值运算符 = 前后的空格 (whitespace) 字符并非是必需的，而之所以用到它们，是因为包含了空格的话，代码会具有更好的可读性。作为一种编程风格，将会在所有二元运算符的前后都始终保留一个空格。另一方面，让每个声明语句都占有单独的一行，并且在语句行的前导行处也不带任何多余的空格，这一点很重要。这是因为，Python 会使用缩进和换行来表示块的结构，而不像其他编程语言中是通过括号和分号来表示块的结构的。

现在可以看看这两行到底都做了什么工作。第一行，创建了一个 int 型的对象，该对象的名称是 x<sup>①</sup>。第二行，创建了一个 str 型的对象（是一个 8 位字符串类型），该对象的名称是 y。

一些 Python 程序员更喜欢将名称（如之前用到的 x 和 y）当成对象进行引用，因为对象引用（object reference）指向的是各个对象而不是对象自己。对诸如 int 和 str 这样的基本数据类型，把这些变量看成是“对象”或者看成是“对象引用”并没有多大的差异；它们的表现行为与它们在其他编程语言中的表现行为完全一样

```
>>> x = 82  
>>> x += 7  
>>> x  
89
```

随后还会看到一些例子，在那里将会看到，Python 变量与作为对象引用的不同之处。

### 有关函数、方法和运算符的术语

函数（function）这个术语用来指代一个可独立执行的子程序，而方法（method）这个术语则用来指代一个只有绑定到对象后才能够执行的函数，因此，方法也叫某一特定类的实例。

运算符（operator）可以是独立的，也可以与某一对象绑定，但它又与函数和方法不同，运算符并不会用到括号。运算符的表示不仅有诸如“+”、“\*”和“<”这样较为明显的符号，还会带有诸如 del、print 这种通常称为语句（statement）的名字。

Python 函数并不需要纯粹的数学含义：它们并不一定要返回值，而且还可以修改其参数。Python 函数与 C 语言和 C++ 语言的函数相似，或者像带有 var 参数的 Pascal 函数。Python 方法与 C++ 或 Java 的成员函数类似。

在 Python 中，对象的比较有两种方式：一种是通过标识符（identity），一种是通过值（value）。对象的标识符就是其在内存中的有效地址，这也是对象引用（object reference）所保存的值。如果使用比较运算符，比如“==”和“<”，得到的将是值的比较。例如，如果两个字符串含有同样的文本，就可使用“==”表示它们相等。如果使用 is，就是标识符的比较，这样做的速度会更快，因为此时比较的只是两个地址，而不需要看这两个对象自身的内容。对一个对象引用调用函数 id()，可以得到该对象的标识符。

Python 有一个称为 None 的特殊对象。它可分配给任意变量，也就意味着，这个变量没有

<sup>①</sup> 这与 Java 的赋值运算类似：Integer x = new Integer(71)；对于 C++ 来说，一个近似的等价写法可能是这样的：int xd = 71；int &x = xd；。

具体的值。无论何时，`None` 对象的实例只有一个，因而在对 `None` 对象进行测试时，总是可以使用较为快速的 `is` 和 `is not` 比较。

值得注意的是，在提示符 `>>>` 后只写了一个 `x`。如果在 IDLE 中键入的是一个表达式或者一个变量，就会自动打印输出它的值。在程序中，必须使用显式的 `print` 语句来输出表达式。例如

```
print x
```

Python 的 `print` 语句是运算符，而不是函数，因此，调用它时不需要使用括号（就像在使用“`+`”和其他运算符都没有使用括号一样）。

之前曾讲到，Python 使用动态类型。其中有两个主要原因。第一个原因是，可以把任何对象赋值给任何变量；例如，就可以写成这样

```
x = 47
x = "Heron"
```

在第一行之后，`x` 的类型是整数 `int` 型的；在第二行之后，`x` 的类型是字符串 `str` 型的；由此可见，与变量 `x` 相关的类型是由该名称所绑定的内容来决定的，而与其本身的内部属性无关。因此，就不需要把某一变量的名称与某个特定的类型关联起来。

Python 动态类型的第二个特点是，动态类型是强类型的：Python 不允许在不兼容的类型之间进行运算，在 IDLE 中键入如下内容，可以看到

```
>>> x = 41
>>> y = "Flamingo"
>>> x + y
Traceback (most recent call last):
File <pyshell#2>, line 1, in <module>
  x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

在尝试应用二元运算符 `+` 时，Python 就会抛出一个 `TypeError` 异常并拒绝执行该运算<sup>①</sup>（异常会在第 2 章中介绍）。

如果当时给 `y` 分配的是一个与 `x` 类型相兼容的类型，比如整数 `int` 或浮点数 `float`，该语句就可以正常运行了

```
>>> x = 41
>>> y = 8.5
>>> x + y
49.5
```

尽管 `x` 和 `y` 是不同的类型（一个是 `int`，一个是 `float`），但由于 Python 同样提供了其他语言中也会用到的自动类型转换技术，故而 `x` 还是会被转换成浮点数 `float`，从而使得实际执行的计算过程为  $41.0 + 8.5$ 。

给一个变量赋值称为绑定（binding），因为实际上是把名称绑定到对象上。如果给一个已经存在的变量赋一个新对象，则称为对该名称进行重绑定（rebinding）。这一过程如图 1.2 所示。那么在这一过程中，该名称一开始就绑定的对象会发生什么呢？例如

```
>>> x = "Sparrow"
>>> x = 9.8
```

这里的矩形表示各个对象，圆表示对象引用，结果给出的是上述代码的执行结果。

带有文本“`Sparrow`”的字符串对象 `str` 发生了什么？对象一旦没有具体的名称绑定，对象

<sup>①</sup> `Traceback` 这一行，`File "<pyshell#2>"` 等，每次执行时会有所不同，因而在你的计算机上看到的可能会与这里给出的有所不同。

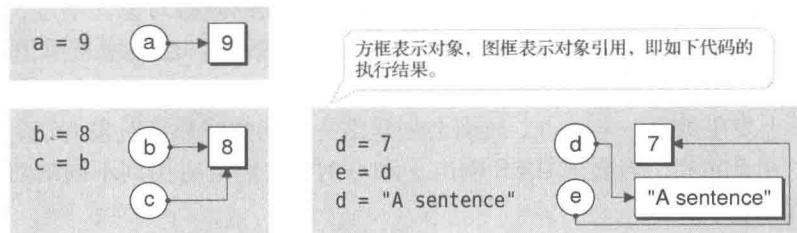


图 1.2 对象引用和对象

所占用的空间就会由垃圾收集程序予以处理，这就意味着它可能会从内存中删除掉。这与 Java 中的工作原理比较类似。

Python 的变量名称由 ASCII 字符、数字和下画线(\_)组成。变量的名称必须以字符开头，且区分大小写(比如，rowan、Rowan 和 roWan 就是三个不同的变量)。Python 的变量名称不可与 Python 的任何关键字名称相同(参见表 1.1)，也不可与 Python 的任何内置常量相同，如 None、True 或者 False。

表 1.1 Python 的关键字<sup>①</sup>

and	class	elif	finally	if	lambda	print	while
as <sup>2.6</sup>	continue	else	for	import	not	raise	with <sup>2.6</sup>
assert <sup>1.5</sup>	def	except	from	in	or	return	yield <sup>2.3</sup>
break	del	exec	global	is	pass	try	

### 1.3 数字和字符串

Python 提供了数个数字类型和两种字符串类型。所有这些类型的共同点在于，它们都是不可变的(imutable)。这就意味着，在 Python 中，数字和字符串不能改变。这听起来相当不灵活，但由于 Python 使用增量赋值运算符(augmented assignment operator，如 +=、\*=，等等)，因而并不会造成什么不便。

在开始学习特定的数据类型之前，先来看看 Python 不变性所造成的一个重要后果。在 IDLE 中，先输入一些简单的表达式

```
>>> x = 5
>>> y = x
>>> x, y
(5, 5)
```

这里先创建了一个值为 5 的 int 型对象，并将名称 x 与其绑定。然后，将 x 赋值给 y，这使得 y 绑定到了与 x 所绑定的相同对象上。因此，当在 IDLE 中输出它们时(如果是在程序中，就需要明确地写成 print x, y 的形式，不过，在 IDLE 中，只需写出表达式，IDLE 就会自动将其输出)，IDLE 会以元组(tuple)的形式输出它们——实质上是这些值的一个只读列表(list)。

现在，来增加一下 y 的值：

```
>>> y += 1
>>> x, y
(5, 6)
```

<sup>①</sup> 在表中关键字旁边的数字，表示引入这些关键字的 Python 版本号。

或许预计 `x` 和 `y` 的值应该都是 6，因为两者引用的都是相同的整数对象。不过，由于 Python 的数字(以及字符串)是不可变的，所以并不会产生上述预期效果。当把增量赋值运算符用于不可变对象时，往往会产生出奇妙的语法甜头<sup>①</sup>(syntactic sugar)：它们并不会改变所应用到的对象。因此，实际上发生的是 `y = y + 1`，故而会创建出一个新的整数型对象(其值为 6)，且 `y` 会绑定到该对象。由此可得，当要求 IDLE 输出 `x` 和 `y` 时，它们会引用到不同的对象，而每个对象都会含有不同的值。

需要时刻牢记的是，“=”运算符执行的是绑定操作，而不是赋值操作。左侧的名称会绑定到右侧的对象上(如果名称已经存在，会重新绑定)。对于不可变对象，貌似绑定和赋值没什么区别，就像稍后所看到的那样。然而，对于可变对象(mutable object)，使用“=”意味着不会形成副本(只是将另一个名称绑定到原来的对象上)，因此在真是需要副本时，必须使用 `copy()` 方法，或者是使用 Python 的 `copy` 模块中的某一函数。稍后会讨论到这一点。

实践应用中，使用不可变的数字和字符串是非常方便的。例如

```
>>> s = "Bath"
>>> t = " Hat"
>>> u = s
>>> s += t
>>> s, t, u
('Bath Hat', ' Hat', 'Bath')
```

值得注意的是，这里将字符串 `s` 赋值给 `u`。直觉看来，`u` 应当拥有赋值给它的值“Bath”，并且，虽然 `s` 和 `u` 引用的是同一字符串，但并不希望应用运算符“`+=`”时会带来任何其他副作用。我们的直觉是正确的：`u` 的值不会改变，因为当“`+=`”应用于 `s` 时，就创建了一个新的字符串对象并绑定到变量 `s`，而 `u` 则绑定到原始字符串“Bath”所引用的唯一对象上。

### 1.3.1 整数和长整型

Python 提供了三种整数类型：布尔型(`bool`)、整型(`int`)和长整型(`long`)。布尔型只有两个值，即 `True` 或者 `False`，它们在上下文中用做数字时，分别可以看做是 1 和 0。长整型可带有一个长度只受机器可用内存限制的整数，因此，可创建和处理具有数百位长度的整数。唯一的缺点是长整型的处理速度要比整型慢。整型 `int` 与大多数编程语言提供的整型是一样的；然而，当整型 `int` 进行运算时，如果其值超过了它的范围(例如，在某些计算机上，它的值会大于  $2^{31} - 1$  或者小于  $-2^{31}$ )，整型 `int` 就会被自动转换为长整型 `long`。

Python 使用后缀 `L` 表示长整型 `long`，如有必要，在代码中也可以用相同的方式来表示长整型 `long`。例如

```
>>> p = 5 ** 35
>>> q = 7L
>>> r = 2 + q
>>> p, q, r
(2910383045673370361328125L, 7L, 9L)
```

整数通常会认为是十进制数(decimal)，除非是以 `0x` 开头的数字，就会被认为是十六进制数(hexadecimal)，例如，`0x3f` 就是一个十六进制数，其十进制值是 63，此外，还有以 `0` 开头的八进制数(octal)。任何类型的整数都可以在其后加上后缀 `L`，使其变为长整型 `long`。

<sup>①</sup> 语法甜头(syntactic sugar)，也叫语法糖、糖衣语法，是英国计算机科学家 Peter J. Landin 给出的一个术语：在计算机语言中添加某一种语法，该语法可使程序员更方便地使用计算机语言开发程序，同时有效提高程序代码的可读性，减少程序出错的机会，而这一语法对计算机语言的功能并没有影响——译者注。

Python 支持常见的数字运算符，包括 +、-、\*、/、% 以及它们的增量运算符 +=、-=、 \*=、/= 和 %=。同时，Python 还提供了 \*\* 运算符，表示数字的平方。

默认情况下，当操作数都是整型 int 时，Python 的除运算符“/”会执行整除运算操作。例如， $5/3$ ，得到的结果是 1。这在大多的数编程语言中都是再正常不过的了，不过这样做在 Python 中却显得非常不方便，因为动态类型意味着一个变量在不同的时候可能是整型 int，也可能是浮点型 float。这一问题的解决方案是，明确告诉 Python，让它总是做“真正的除法”(true division)，这样无论在任何时候，都可以得到浮点型的结果，而如果确实需要出现整除运算时，可以使用“//”运算符。在第 4 章，将会详细看到到底该如何做。

### 1.3.2 浮点数和小数

Python 提供了三种类型的浮点值：float、Decimal 和 complex。float 型浮点数可保存双精度浮点数，其取值范围取决于构建 Python 时所使用的 C(或者 Java)的编译器；这些数具有有限的精度，因而用于数值相等的比较并不可靠。float 型数字可用小数点或者科学记数法表示，例如，0.0、5.7、8.9e-4 等。在 IDLE 中这样键入数字，会显得比较方便

```
>>> 0.0, 5.7, 8.9e-4
(0.0, 5.7, 5.700000000000002, 0.0008999999999999995)
```

这里所出现的不准确性并不是 Python 的问题：计算机是用二进制数来表示浮点数的，这就使得可以准确地表示一些小数(比如 0.5)，但却只能大致估计地表示其他的一些值(比如 0.1)。同时，用固定的位数来表示这些数，就造成所能保存的数字有限。

在实践中，这并不是什么大的问题，因为大多浮点数都是 64 位的，可以满足绝大多数的情况。但如果确实需要很高的精度，就可以使用 Python Decimal 模块中的数字。将会以给定的精度来执行这些运算(默认情况下，精度可以达到 28 位小数)，也可以准确地表示诸如 0.1 这样的周期性数字；但与标准浮点数 float 相比，其处理速度会慢很多。由于较好的准确性，Decimal 数字常用于金融计算中。

在用 Decimal 数字之前，必须先导入 Decimal 模块。这一过程的所使用的语句与在.py 文件中或者在 IDLE 中编写代码没有什么两样，如下所示：

```
>>> import decimal
```

这里是将 decimal 模块导入到 IDLE Shell 的交互窗口中[有关导入语句的解释可以参看后续“对象导入”(Importing Objects)]。整型数字可以传递给 Decimal 的构造函数，但由于 Decimal 的精度高，而浮点数 float 的精度低，所以不能给 Decimal 传递浮点数 float；因此，要给 Decimal 传递浮点数，必须把浮点数当成字符串进行传递。例如

```
>>> decimal.Decimal(19), decimal.Decimal("5.1"),
decimal.Decimal("8.9e-4")
(decimal("19"), Decimal("5.1"), Decimal("0.00089"))
```

数字 decimal.Decimal("5.1") 可以准确保存；但如果作为浮点数 float，它可能会被表示成诸如 5.099999999999996 这样的数字。与此类似，decimal.Decimal("0.00089") 将变为诸如 0.00088999999999995 这样的数字。从 Decimal 转为 float 很简单，尽管这样做可能会丧失部分精度

```
>>> d = decimal.Decimal("1.1")
>>> f = float(d)
>>> f
1.1000000000000001
```

Python 还提供了内置数据类型的复数。这些数字由一个实数和一个虚数组成，后者用后缀 `j` 表示<sup>①</sup>。例如：

```
>>> c = 5.4+0.8j
>>> type(c)
<type 'complex'>
```

这里输入的就是复数(语法形式为：实数部 `real_part + 虚数部 imaginary_part j`)，并通过调用 Python 的 `type()` 函数，来说明 `c` 所绑定的数据类型。

## 对象导入

Python 有一个大而全面的模块库，这些模块提供了大量的预定义功能。基于这一功能性机制，就可以导入所需的常量、变量、函数和类。模块导入的语法一般是

```
import moduleName
```

然后，用点运算符就可以访问模块中的各个对象了。例如，`random` 模块提供了 `randint()` 函数，对其进行导入和使用的方法如下：

```
import random
x = random.randint(1, 10)
```

值得注意的是，通常需要将导入语句 `import` 放到 `.py` 文件的开头，不过，也可以将其放在其他地方，比如，放到某个函数的定义内。

一个使用 Python 模块系统的好处是，每个模块都可以单独看成一个命名空间，因此，就可  
以毫不费力地避免函数名称的冲突。例如，之前可能已经定义过自己的 `randint()` 函数了，但  
这并不会造成函数名称的冲突，因为访问例中所导入的函数时使用的是全名的 `random.  
randint()`。在第 3 章还将会看到，如何创建自定义的模块和如何导入自定义的对象。

各个模块本身可以包含其他一些模块，对于非常大的一些模块来说，将各个对象直接导入到当前的命名空间中更为方便些。Python 为此提供有相应的语法。例如

```
from PyQt4.QtCore import *
x = QString()
y = QDate()
```

这里的做法会导入所有对象，也就是说，`PyQt4` 的 `QtCore` 模块中的所有类都会导入进来，这样也就允许我们使用这些不受任何限制的函数名。这允许我们使用它们的名称。一些开发人  
员对这种用法很是苦恼，但由于几乎所有的 `PyQt` 对象都是以大写字母“Q”开头的，所以如果  
在创建自己的对象时不以大写字母“Q”开头，未来应该就不那么容易出现名称的冲突了。不  
过，对于那些总是喜欢使用全名的开发人员来说，则可以使用像这样的常规导入语法：

```
import PyQt4
x = PyQt4.QtCore.QString()
y = PyQt4.QtCore.QDate()
```

出于便捷性考虑，对于各个 `PyQt4` 模块来说，还是会使用“`from...import`”这样的导入语  
法，尽管在导入其他模块时，使用的仍旧是常规导入语法。

<sup>①</sup> 数学中会用 `i` 来表达虚部数字，但对于工程师们和 Python 来说，用的则是 `j`。

Python 浮点数也有与整型数一样的基本运算，当几种不同类型的数字进行运算时，整型数会自动转换成浮点数进行运算。

### 1.3.3 字节字符串、Unicode 字符串和 QString

在 Python 中，有两种内置类型的字符串：一个是保存字节的 str 字符串，一个是保存 Unicode 字符的 unicode 字符串。这两种字符串类型均支持相同的字符串处理运算法则。像数字一样，Python 的字符串也是不可变的（immutable）。它们也是序列，因此它们也可以当做参数传递给可接收序列的各个函数，还可以使用 Python 的序列运算，例如，`len()` 函数就会返回一个序列的长度。PyQt 则提供了第三种字符串类型 `QString`。

如果只处理 7 位的 ASCII 字符，也就是说，字符的范围在 0 ~ 127 之间，且又想节省一些内存，就可以使用 `str`。然而，如果要使用 8 位字符集，则务必清楚要使用的到底是哪种编码形式。例如，在西欧，8 位字符串的编码形式通常会用 Latin-1 编码。一般来说，对于某个字符串，只是通过检查各个字节来判断用的是哪种编码方式往往不是那么容易。很多现代 GUI 库，包括 Qt，使用的都是 Unicode 字符串，因此，最安全的方法就是，对于 7 位 ASCII 字符使用 `str` 字符串，而对二进制的 8 位字节使用 `unicode` 或者其他诸如 `QString` 这样的字符串。

通过引号可以创建 Python 的字符串

```
>>> g = "Green"  
>>> t = ' trees'  
>>> g + t  
'Green trees'
```

在创建 Python 字符串时，只要在字符串的两端使用了同一种引号，至于是单引号还是双引号，没有什么不同。

要强制将字符串变为 `unicode` 型字符串，只需在引号前加上字母 `u` 即可

```
>>> bird = "Sparrow"  
>>> beast = u"Unicorn"  
>>> type(bird), type(beast), type(bird + beast)  
(<type 'str'>, <type 'unicode'>, <type 'unicode'>)
```

值得注意的是，可用二元运算符“+”来连接各个字符串，不过，如在同一次运算中同时包含了 `str` 和 `unicode`，那么 `str` 就会转换为 `unicode`，且结果也会是 `unicode` 类型（如果 `str` 中包含的字符超出了 7 位 ASCII 字符集的范围，Python 就会发抛出 `UnicodeEncodeError` 异常；有关异常的知识会在第 2 章中看到）。

在 Python 中，没有单独的“字符”类型：一个单字符就是一个长度为 1 的字符串。可以使用 `chr()` 函数，以一个字节为值，获得一个字符，该函数可以接受范围为 0 ~ 255 的整数值。在 Python 文档中，并没有明确指出超过 ASCII 字符范围的那些值该采用哪种编码方式（比如，值大于 127）。对于 Unicode，可以使用 `unichr()` 函数，该函数可以接受范围为 0 ~ 65 535 的整数<sup>①</sup>。要想实现字符串的反向转换，即从字符转换为其整数值（ASCII 值或者 Unicode 的编码值），可以使用函数 `ord()`。例如

① 如果把 Python 配置为 UCS-4 表示方法，其范围可以扩展到 1114 111。

```
>>> euro = unichr(8364)
>>> print euro
€
>>> ord(euro)
8364
```

为什么这里要用 `print` 语句而不让 IDLE 来输出结果呢？这是因为，IDLE 在显示字符串中的非 ASCII 码字符时，会以十六进制转义，所以如果不加 `print`，IDLE 就会输出 `u'\u20ac'`。

还可以通过名称访问 Unicode 字符：

```
>>> euro = u"\N{euro sign}"
>>> print euro
€
```

如果需要在字符串中包含特殊字符，可以通过反斜杠“\”进行转义。在表 1.2 中，给出了一些可用的转义字符；Unicode 转义符只有在 `unicode` 字符串中才会起作用。

表 1.2 Python 中的字符串转义符

转义符	含义
\newline	回车换行(也可忽略)
\	反斜杠(\)
\'	单引号(')
\"	双引号(")
\a	ASCII 字符集的铃(BEL)
\b	ASCII 字符集的退格键(BS)
\f	ASCII 字符集的换页(FF)
\n	ASCII 字符集的换行(LF)
\N{name}	Unicode 字符名
\r	ASCII 字符集的回车键(CR)
\t	ASCII 字符集的 Tab(TAB)
\uhhhh	16 位的十六进制 Unicode 字符
\Uhhhhhhhh	32 位的十六进制 Unicode 字符
\v	ASCII 字符集的竖向 Tab(VT)
\ooo	八进制字符
\xhh	十六进制字符

以下是两个引号转义符用法的例子：

```
"He said \"No you don't!\" again."
'What\'s up with him?'
```

由于双引号的界限，无须对字符串中的单引号进行转义，同时也不必对由单引号界定的字符串中的双引号进行转义。

对于多行字符串，可以使用“三”引号来表示

```
'''This string has three lines in it, with a 'quote',
another "quote", and with just one embedded newline \
since we have escaped one of them.'''
```

这些种类的字符串可像普通字符串一样包含各类转义字符，也可如上述示例一样使用三个单引号或者三个双引号进行界定。在三引号字符串和 Python 代码中，新行符(`newline`)可通过在代码前面加上反斜杠(\)进行转义(即使在 Windows 的行末尾会有两个字符而不是一个字符，这种方法也同样有效)。

Python 字符串是字符的有序集合，可通过位置索引访问字符串中的单个字符，其中，第一个字符的索引值为 0。同样，也可以从字符串的末尾访问每个字符，最后一个字符的索引值是 -1。例如

```
>>> phrase = "The red balloon"
>>> phrase[0], phrase[5], phrase[-1]
('T', 'e', 'n')
```

负的索引值可以用来从右到左访问字符，因为最右端字符的位置索引值是 -1，其左侧字符的位置索引值是 -2，以此类推。

Python 的序列(sequence)还支持切片(slicing)，这也就意味着，可以从一个序列中复制出多个子序列。一个切片由一个、两个或者三个相互独立的部分构成：开始(默认从 0 开始)、结束(默认以序列长度为结束)和一个在此暂不予讨论的部分<sup>①</sup>。切片会从开始索引值处开始并包含开始字符，但不会包含结束索引值处的字符。下面给出了一些示例：

```
>>> phrase = "The red balloon"
>>> phrase[:3]
'The'
>>> phrase[-3:]
'oon'
>>> phrase[4:7]
'red'
```

由于 Python 字符串是不可变的，所以不可能为字符串里的单个字符或者切片重新赋值

```
>>> p = "pad"
>>> p[1] = "o"      # WRONG
Traceback (most recent call last):
File <pyshell#64>, line 1, in <module>
    p[1] = o
TypeError: object does not support item assignment
```

向字符串中插入字符最简单的方式就是使用如下所示的切片语法：

```
>>> p = "pad"
>>> p = p[:1] + "o" + p[2:]
>>> p
'pod'
```

上述方法要明确字符数，看起来很麻烦，不过在实际编程中，一般会通过调用函数来获得索引值，例如，可以使用函数 `find()`。

当然，也有其他的方法。例如

```
>>> p = "pad"
>>> p = "o".join((p[:1], p[2:]))
>>> p
'pod'
```

具有 Pascal 或者 C++ 背景的程序开发人员过去经常使用可变字符串，一开始可能会觉得字符串的不变性很不方便。当然，Python 也确实提供了可变字符串；它们在 `StringIO` 模块和(速

<sup>①</sup> 开始表示切片开始的位置，结束表示切片到哪里结束，第三个部分表示切片的间隔数。如果不指定开始，Python 就从序列首开始；如果不指定结束，Python 会停止在序列尾。值得注意的是，返回的结果序列是从开始位置开始，刚好在结束位置之前结束，即开始位置包含在结果切片序列中，而结束位置被排斥在切片序列之外——译者注。

度更快的) `cStringIO` 模块中。PyQt 的 `QString` 类也是可变的。但在实践中，对不可变字符串的 Python 式处理，特别是上述所给出的处理范式，使用 `join()` 函数进行连接的方法，将很快成为第二直觉式选择。

Python 字符串包含了很多有用方法，但这里将只关注那些最为常用的方法。在 Python 中，方法的调用是通过对对象引用进行点号操作来实现的，而括号()也表明此时正在执行的是一个方法(成员函数)的调用<sup>①</sup>。例如

```
>>> line = "The quick brown fox."
>>> line.find("q")
4
```

`find()`方法会返回字符串中由其参数指定的最左侧字符的位置，但如果找不到给定的字符，将返回 -1。

Python 还提供了 `index()` 方法，也具有完全相同的用法，不过查找失败时，它会抛出 `ValueError` 异常。其他有关序列操作的类(如列表)也有 `index()` 方法，所以字符串操作也就有了该函数，从而保证了一致性。

由于对字符串可以使用 `find()` 或者 `index()`，那么每次使用时，选择哪一个会比使用另一个更好呢？对于一次性搜索，使用 `find()` 并用来检测返回值，会更方便些。但如果是有一大段代码，需要执行大量的多次搜索，使用 `find()` 的话，可以让我们检查每次的搜索返回值，而使用 `index()` 则会假设搜索结果总是有效，但结果如果无效，就需要用单独的异常处理程序来处理这些错误了。当然，如果没有捕捉到这些异常，它们将会被传递到调用栈中，而如果所有地方都没有捕捉到这些异常，就会导致应用程序崩溃。在本书中，将会使用这两种方法，但根据情况的不同，会使用两者中更为合适的一种。

字符串方法也可以用于字符串对象和字符串文本中

```
>>> "malthusian catastrophe".title()
'Malthusian Catastrophe'
```

`title()`方法会返回它所应用到的字符串的副本，但其中的每个单词的第一个字母都会变为大写。Python 提供了字符串的数据格式化语法，类似于 C 语言函数库中的 `printf()` 函数。

表 1.3 中给出了 Python 中的部分字符串操作方法和函数。

表 1.3 部分字符串操作方法和函数

语法	说明
<code>x in s</code>	如果字符串 x 是字符串 s 的一个子串，则返回 <code>True</code>
<code>x not in s</code>	如果字符串 x 不是字符串 s 的一个子串，则返回 <code>True</code>
<code>x + s</code>	返回字符串 x 和 s 连接后的字符串
<code>s * i</code>	返回一个由 i 个字符串 s 相互连接而成的字符串。例如，“ <code>Abc</code> ”* 3 会产生字符串“ <code>AbcAbcAbc</code> ”
<code>len(s)</code>	返回字符串 s 的长度；如果 s 的类型是 <code>str</code> ，返回字节数；如果 s 是 <code>unicode</code> ，返回字符数
<code>s.count(x)</code>	返回字符串 x 在字符串 s 中出现的次数。这个方法以及其他数个方法，都带有可选的开始和结束参数，以便把搜索限定到所调用的字符串切片上
<code>s.endswith(x)</code>	如果字符串 s 以 x 结尾，返回 <code>True</code>

<sup>①</sup> 正如之前提到的，括号是不会用于诸如“+”和 `print` 这样的运算符上的。

(续表)

语法	说明
s.startswith(x)	如果字符串 s 以 x 开头, 返回 True
s.find(x)	返回字符串中出现 x 的最左侧字符的索引值; 如果没找到, 则返回 -1
s.rfind(x)	和 find() 类似, 但是从右往左搜索
s.index(x)	返回字符串中出现 x 的最左端的索引值; 如果没有找到 x, 则抛出 ValueError 异常
s.rindex(x)	和 index() 类似, 但是从右往左搜索
s.isdigit()	如果字符串非空且字符串中包含的一个或者全部字符都是数字, 返回 True
s.isalpha()	和 isdigit() 类似, 但检测的是字符
s.join((x, …))	返回一个字符串, 这个字符串由调用该方法的字符串依次连接各个给定的序列。例如, ":".join(("A", "BB", "CCC")) 调用会返回字符串"A:BB:CCC"。其中的分隔符序列可以为空
s.lower()	返回字符串 s 的全小写副本
s.upper()	返回字符串 s 的全大写副本
s.replace(x, y)	返回字符串 s 的一个副本, 其中, 在字符串 s 中出现字符串 x 的任意位置都用字符串 y 进行替换
s.split()	返回一系列用空格分割的字符串列表。例如, "ab\tc\td\te".split() 会返回 ["ab", "c", "d", "e"] 列表。这个方法中给定的第一个参数, 是将要分割的字符串, 也可以给定第二个参数, 说明最多要分割几个
s.strip()	返回一个字符串的副本, 副本中去掉了字符串开头和串内的空格。可以带一个可选参数, 以说明要从串内移除哪个字符

为获得格式化操作, 可以使用二元运算符“%”, 该运算符的左侧带有一个格式化字符串, 右侧带有一个被格式化的对象(通常是一个对象元组)。例如

```
>>> "There are %i items" % 5
'There are 5 items'
```

字符串中的% i 被替换为数字 5。格式化字符串中% 后面的字母表示对象的类型, 用% i 表示这是一个整数。

这里给出的例子中, 显示了要替换的三种不同类型, 其中的箭头用来说明哪个% 项要被哪个元组元素所代替:

```
>>> "The %i %s cost %f dollars" % (3, "fish", 17.49)
'The 3 fish cost 17.490000 dollars'
```

这里的% 叫做格式指示符(format specifier), 格式化字符串中会至少存在一个这种字符。格式指示符通常由一个百分符号(%)后跟格式化字符构成。百分号自己则用%% 表示。在上面的例子中, 用% i 作为整数 int 的格式指示符, % s 是字符串的指示符, 而% f 则是浮点数 float 的指示符。

之前, 我们看到过如何向字符串中插入一个子字符串。看到了是如何对切片实现这一点的, 并且用到了更 Python 化的字符串 join() 方法。下面给出的是第三种方法, 即使用格式指示符方法:

```
>>> p = "pad"
>>> p = "%$o%$" % (p[:1], p[2:])
>>> p
'pod'
```

这里创建了一个新的字符串, 该字符串由一个字符串(来自字符串 p 的第一个切片)、“o”和另

一个字符串(来自字符串 p 的第二个切片)构成。早前给出的 `join()` 方法用来连接这两个字符串;这个方法可以用来“排布”出不同的字符串。

我们通过在% 和字符之间放置一些信息,来做一些控制格式化% 元素的练习。例如,为了让一个浮点数小数点后只显示两位数字,可以使用指示符`.2f`:

```
>>> "The length is %.2f meters" % 72.8958
'The length is 72.90 meters'
```

下面再给出几个例子,其中两个会给出% 运算符与 `print` 语句联合作用的示例:

```
>>> print "An integer", 5, "and a float", 65.3
An integer 5 and a float 65.3
>>> print "An integer %i and a float %f" % (5, 65.3)
An integer 5 and a float 65.300000
>>> print "An integer %i and a float %.1f" % (5, 65.3)
An integer 5 and a float 65.3
```

许多情况下,只需`%i`(以及它的同义词`%d`)、`%f` 和`%s` 就足够用了。有关格式指示符的详细说明以及如何对其进行特殊修改以得到相应特定结果的方法都会在 Python 文档予以介绍;也就是说,可以在 Python 文档中查找“字符串格式化操作”(String Formatting Operations)即可。也可以用 Python 等其他方法,比如,在 `string` 模块中的 `Template` 类就提供了类 Perl 的解释器,同样可以完成字符串的格式化。甚至还可以使用类C++式的语法;具体这一点可以查看 Python Cookbook 中“Using C++-like iostream Syntax”一节的内容<sup>①</sup>(该书的信息可以参阅“Python 文档”中的内容)。

## Python 文档

Python 提供了数量繁多的文档。大部分的文档质量都很不错,不过也有不少地方涉及的相对较浅。使用中采用 HTML 版本导览文档,会使得文档的组织看起来更像是一本实体书而不是一个在线文档,这样也会使页面之间的交叉链接能够少许多。

Windows 用户要幸运很多,因为给他们提供了 Windows 帮助文件格式的文档。点击操作系统左下角的“开始→所有程序→Python 2.x→Python 手册”,就可以打开 Windows 帮助浏览器。这个工具既提供了索引(Index)的功能,又提供了搜索(Search)的功能,这使得文档的查找工作变得非常轻松。例如,要找到字符串格式指示符的有关信息,只需在索引(Index)的行编辑栏中输入“formatting”,就会出现“formatting, string(%)”的有关条目。

快速浏览一遍文档还是很值得的。建议先看看“库引用”(Library Reference)页面(`lib.html`),从中可以看出 Python 提供了哪些标准库,然后再点击那些感兴趣的主題。这样就可以提供一个哪些可用的初步印象,也有助于让初步知道有哪些文档的内容可能是感兴趣的。

值得注意的是,某些主题的内容下面会涵盖多个标题。例如,在读到有关字符串的内容时,就可以看到“序列类型”(Sequence Types)、“字符串方法”(String Methods)、“字符串格式化操作方法”(String Formatting Operations)和“字符串服务”(String Services)等不同的标题。同样,对于不同的文件和目录,也可以分别看到“访问文件和目录”(File and Directory Access)、“数据的压缩和归档”(Data Compression and Archiving)以及“文件和目录”(Files and Directories)等标题。

对于喜欢印刷读物的读者来说,可以考虑以下几部书籍:

<sup>①</sup> 见 <https://www.safaribooksonline.com/library/view/python-cookbook-2nd/0596007973/ch02s14.html>—译者注。

- 由 Wesley Chun 编著的 *Core PYTHON Programming*。这是一本 Python 教程，可能会适用于那些 Python 新手们，比本书的第一部分所提供的内容还要易读。
- 由 Alex Martelli 编著的 *Python in a Nutshell*。这是一本非常不错的参考书，对 Python 语言和 Python 标准库进行了详细而准确的说明。
- 由 Alex Martelli、Anna Martelli Ravenscroft 和 David Ascher 三人编著的 *Python Cookbook*（第二版）。这本书提供了大量的小型实用函数、类、技巧和思路，非常有助于开阔 Python 程序员使用 Python 开展工作的意识。这些技巧也可以在线找到，参见 <http://aspn.activestate.com/ASPN/Python/Cookbook>。

有关 Python 的在线信息可以先从 <http://www.python.org> 开始。这个网站也是 Python 的主站点。与 PyQt 相关的信息可以参看 <http://www.riverbankcomputing.co.uk>。The PyQt wiki 的网址是 <https://wiki.python.org/moin/PyQt> 或者 <https://en.wikipedia.org/wiki/PyQt>。

值得注意的是，`print` 语句会自动在它所输出的参数之间多输出一个空格。要避免这种情况，可以使用 `sys.stdout.write()` 而不是使用 `print` 接口；更多有关 `write()` 的内容可以参见第 6 章。

在使用 PyQt 时，还可以使用另外一种字符串类型，`QString`。与 Python 的 `str` 和 `unicode` 不同，`QString` 是可变的；这就意味着，我们可以对 `QString` 字符串中的子串进行替换、插入和移除操作，也可以改变 `QString` 字符串中的单个字符。`QString` 还拥有与 `str` 和 `unicode` 不同的 API（Qt 提供 `QString` 是因为 Qt 是用 C++ 编写的，还不能内置支持 Unicode）。

`QString` 可以保存 Unicode 字符，但要取决于所使用的 Python 版本，其内部表示可能会与 Python 的 Unicode 表示有所不同；这并不是什么大问题，因为 PyQt 可以轻松实现 `unicode` 和 `QString` 之间的转换。例如

```
>>> from PyQt4.QtCore import *
>>> a = QString("apple")
>>> b = unicode("baker")
>>> print a + b
applebaker
>>> type(a + b)
<class 'PyQt4.QtCore.QString'>
```

在这里，导入了 `QtCore` 模块中的所有类，从而可用于整个 PyQt4 模块中。在执行 `QString` 和 Python 字符串的有关操作时，正如 `type()` 函数所揭示的那样，生成的字符串总会是 `QString` 类型的。

在使用 PyQt 时，Qt 的许多方法都会带有 `Str`、`unicode` 或 `QString` 类型的字符串参数，而 PyQt 将自动执行一些必要的转换。Qt 方法总会返回 `QString` 型字符串。从 Python 动态类型的角度看，就很容易感到困惑，难以确定返回的字符串到底是 `QString` 还是 Python 字符串。由于这一原因，对于字符串的使用，明智的做法就是制定一个策略，以便能够实时清楚字符串的类型。

本书在使用 PyQt 的基本策略如下：

- 只对严格的 7 位 ASCII 字符串或者原始的 8 位数据，即用原始字符表示的字符串，才使用 `str` 类型。

- 对于那些只被 PyQt 函数所使用的字符串，例如，PyQt 函数返回的字符串只会传给另一个 PyQt 函数，这样就不需要转换这些字符串了。那么，只需简单将其保持为 `QString` 即可。
- 对于其他各类情况，使用 `unicode` 字符串，并尽可能尽快将 `QString` 转换为 `unicode`。换句话说，Qt 函数中一旦返回 `QString` 字符串，总是要即可将其转换为 `unicode` 字符串。

这一策略也就意味着，可以有效避免对 8 位字符串编码做出错误假设（因为我们使用的是 Unicode）。同时，也可以确保传递给 Python 函数的字符串拥有 Python 所期望的方法：`QString` 具有与 `str` 和 `unicode` 不同的方法，因此给 Python 函数传递 `QString` 会出现错误。PyQt 使用 `QString` 而不是 `unicode` 是因为，PyQt 在最初创建时，Python 当初对 Unicode 的支持并不像现在做得那么好。

## 1.4 集合

一旦有了变量，也就是说，有了可以对特定类型对象进行引用的带有名字的单个对象引用，自然就希望能够拥有整个对象引用的集合（collection）。Python 的标准集合类型可以保存对象引用，因此，实际上它们可以保存任何类型对象的集合。使用对象引用的集合所能产生的另一个效果是，它们可以对不同类型的对象进行引用：它们并不需要受内部元素都必须是同一类型这一条件的限制。

内置的一些集合类型有 `tuple`（元组）、`list`（列表）和 `dict`（字典，即 `dictionary`）、`set`（集）和 `frozenset`（原封集）。除 `tuple` 和 `frozenset` 之外，其他的都是可变的，因此，可以对列表、字典和集中的元素进行添加和删除。在 `collections` 模块中，还额外提供了一些可变的集合类型<sup>①</sup>。

Python 在其标准库中有一个集合类型无法保存对象引用；不过，它却可以保存特定类型的数字。这就是 `array`（数组）类型，它主要应用于需要存储大量数据且需要及时高效处理这些数据的地方。

在这一节，将会看到 Python 这些内置的集合类型。

### 1.4.1 元组

元组（tuple）就是一个由零个或者多个对象引用所组成的有序序列。与字符串（也正如很快就会看到的，类似于列表）一样，元组支持诸如 `len()` 这样的序列函数，就像之前见过的切片语法一样。这使得从元组中提取各个元素变得非常容易。然而，元组是不可变的，因此，不能替换或者删除元组中的任何一个元素。但如果确实希望能够对一个有序序列进行修改，只需使用列表而不要使用元组即可；或者，如果早先已经有了一个元组，然后又想修改它，那么只需将它转换成列表，然后再进行更改即可。

之前已经非正式地接触过一些元组，例如，早先在 IDLE 中的某些交互所产生的结果就曾被封装成元组，也曾经使用元组为% 运算符提供过多个参数。

---

<sup>①</sup> Qt 库为自己提供了可用于 C++ 的容器类富集，不过在 PyQt 中无法使用这些容器类，并且在任何时候，Python 自带的集合类都相当好用。

下面的这些例子，展示了如何来构造元组：

```
>>> empty = ()
>>> type(empty)
<type 'tuple'>
>>> one = ("Canary")
>>> type(one)
<type 'str'>
>>> one = ("Canary",)
>>> type(one)
<type 'tuple'>
```

创建一个空元组相当简单，但对只有一个元素的元组来说，必须使用逗号来将其与括号表达式进行区分

```
>>> things = ("Parrot", 3.5, u"\u20AC")
>>> type(things)
<type 'tuple'>
```

元组可以保存任何类型的元素；这里就包含了 str、float 和 unicode 几种元素。如果至少含有两个元素并且其含义不会产生歧义，也是可以省略元组中的括号的

```
>>> items = "Dog", 99, "Cow", 28
>>> type(items)
<type 'tuple'>
```

元组可随意进行嵌套和切片，正如下面各例所示：

```
>>> names = "Albert", "Brenda", "Cecil", "Donna"
>>> names[:3]
('Albert', 'Brenda', 'Cecil')
>>> names[1]
'Brenda'
```

首先创建了一个含有名字的元组，然后对前三个名字元素进行了切片，随后检查了索引值为 1 的那个元素。与所有的 Python 序列类似，第一个元素所在的位置是 0

```
>>> names = names[0], names[1], "Bernadette", names[2], names[3]
>>> names
('Albert', 'Brenda', 'Bernadette', 'Cecil', 'Donna')
```

现在已经改变了 names 元组来引用一个新的元组，会在这个新元组中间加入一个元素。此时，或许可以尝试写下 name[:2] 而不是 name[0]、name[1]，也可以类似地对最后的两个名字元素用 name[2:] 来代替，不过，如果要是这样做了，最终将会得到一个三个元素的元组

```
(('Albert', 'Brenda'), 'Bernadette', ('Cecil', 'Donna'))
```

这是因为，当对一个元组进行切片时，所得的各个切片将总是元组自身。

```
>>> names
('Albert', 'Brenda', 'Bernadette', 'Cecil', 'Donna')
>>> names = names[:4]
>>> names
('Albert', 'Brenda', 'Bernadette', 'Cecil')
```

实际上，在这里，通过获取元组的前 4 项，可以切除掉最后一个名字，即这些元组索引位置 0、1、2 和 3。切片中，第一位数字是第一个索引值，这个元素会包含到结果中，第二个数字是最后一个索引值，但这个元素将不会包含到结果中。

```
>>> names
('Albert', 'Brenda', 'Bernadette', 'Cecil')
>>> names = names[:-1]
>>> names
('Albert', 'Brenda', 'Bernadette')
```

切除掉最后一个元素的另一种方法是，从末尾开始索引；这种方法就不需要知道元组的长度。但如果想知道元组的长度，则可以使用 `len()` 函数

```
>>> pets = ("Dog", 2), ("Cat", 3), ("Hamster", 14)
>>> len(pets)
3
>>> pets
(('Dog', 2), ('Cat', 3), ('Hamster', 14))
>>> pets[2][1]
14
>>> pets[1][0:2]
('Cat', 3)
>>> pets[1]
('Cat', 3)
```

如有必要，可以使用多个方括号来实现元组的嵌套和元素的访问。

任何序列都可以传递给元组的构造函数，从而创建出一个元组。例如

```
>>> tuple("some text")
('s', 'o', 'm', 'e', ' ', 't', 'e', 'x', 't')
```

在需要修改对象有序集合的顺序时，元组会非常有用。它们也可以用做一些函数和方法的参数。例如，在使用 Python 2.5 时，`str.endswith()` 方法既可以接受单一的字符串参数（比如，“.png”），也可以接受一个字符串元组（比如，`("png", ".jpg", ".jpeg")`）。

## 1.4.2 列表

`list(列表)` 类型是一个与元组 `tuple` 类似的有序序列。在之前字符串和元组中用到的全部序列函数和切片操作，对于列表也完全有效。列表与元组所不同的地方在于，列表是可变的，有一些可以用来修改列表的方法。然而，创建元组要使用括号，而创建列表则需要使用方括号（或者使用 `list()` 构造函数）。

现在来看一些切片示例，是从列表中提取的一些部分：

```
>>> fruit = ["Apple", "Hawthorn", "Loquat", "Medlar", "Pear", "Quince"]
>>> fruit[:2]
['Apple', 'Hawthorn']
>>> fruit[-1]
'Quince'
>>> fruit[2:5]
['Loquat', 'Medlar', 'Pear']
```

在这里，用到了在字符串和元组中一样的切片语法。

因为列表是可变的，所以可以插入或者删除列表元素。可以通过方法调用来实现这一点，或者也可以在赋值运算符的两侧都使用的切片语法来实现这一点。先来看一下方法调用

```
>>> fruit.insert(4, "Rowan")
>>> fruit
['Apple', 'Hawthorn', 'Loquat', 'Medlar', 'Rowan', 'Pear',
 'Quince']
>>> del fruit[4]
>>> fruit
['Apple', 'Hawthorn', 'Loquat', 'Medlar', 'Pear', 'Quince']
```

使用一次方法调用和一个运算符，我们完成了一个新的元素的插入和该新元素的删除。`del` 语句可以用来移除一个特定索引位置的元素，而 `remove()` 方法则用来移除一个能够

与 `remove()` 方法的参数相匹配的元素。因此，在这个例子中，也可以不那么高效地使用 `fruit.remove("Rowan")` 来删除该元素。

现在，将会使用切片操作来完成同样的事情

```
>>> fruit[4:4] = ["Rowan"]
>>> fruit
['Apple', 'Hawthorn', 'Loquat', 'Medlar', 'Rowan', 'Pear',
'Quince']
>>> fruit[4:5] = []
>>> fruit
['Apple', 'Hawthorn', 'Loquat', 'Medlar', 'Pear', 'Quince']
```

在用“Rowan”赋值时，用到的是方括号，这是因为，这是正在把一个列表切片（一个只有一个元素的列表）插入到另一个列表切片中。如果当时忽略了方括号，Python 将会把单词“Rowan”看成恰如其位的列表，那样就会插进去各个相互分离的“R”、“o”等元素。

在使用多个切片进行插入操作时，源切片和目标切片可以具有不同的长度。如果目标切片的长度为 0，比如 `fruit[4:4]`，那么就会只进行插入操作；不过，如果目标切片的长度大于 0，那么目标切片中相应数量的元素就会被插入切片的那些元素所代替。在这个例子中，我们用一个零元素切片替代了一个单元素切片，自然也就会删除这个单元素切片。

这里给出几个例子：

```
>>> fruit[2:3] = ["Plum", "Peach"]
>>> fruit
['Apple', 'Hawthorn', 'Plum', 'Peach', 'Medlar', 'Quince']
>>> fruit[4:4] = ["Apricot", "Cherry", "Greengage"]
>>> fruit
['Apple', 'Hawthorn', 'Plum', 'Peach', 'Apricot', 'Cherry',
'Greengage', 'Medlar', 'Quince']
>>> bag = fruit[:]
>>> bag
['Apple', 'Hawthorn', 'Plum', 'Peach', 'Apricot', 'Cherry',
'Greengage', 'Medlar', 'Quince']
```

我们用一个长度为 2 的切片替換了一个长度为 1 的切片，`fruit[2:3]`（“Loquat”）。我们也在不移除任何元素的情况下插入过一个三元素切片。在最后一个例子中，我们将 `fruit` 中的元素复制到 `bag` 中；使用 `bag = fruit` 本来也是可以实现复制的，不过在语义上来说，会稍稍有一点点的不同。关于列表复制的更多知识可以参看“浅复制和深复制”（Shallow and Deep Copying）中的内容<sup>①</sup>。

## 浅复制和深复制

之前曾看到过，如果有两个变量同时引用到同一个字符串并修改了其中一个变量，例如，使用 `+=` 进行追加操作——Python 会创建一个新的字符串。发生这样的事情是因为 Python 字符串是不可变的。对于诸如列表（还有字典，很快就会看到）之类的可变类型，事情就会有所不同了。

① 可以简单理解为：Python 中对象之间的赋值是按引用传递的，如果需要复制对象，可以使用标准库中的 `copy` 模块。  
(1) `copy.copy` 是浅复制，此时只会复制父对象，而不会复制对象内部的子对象；(2) `copy.deepcopy` 是深复制，会复制对象及其子对象——译者注。

例如，如果用两个变量创建一个列表，对其进行引用，再通过其中一个变量来修改列表，两个变量就都会引用到这一个修改后的列表上

```
>>> seaweed = ["Aonori", "Carola", "Dulse"]
>>> macroalgae = seaweed
>>> seaweed, macroalgae
(['Aonori', 'Carola', 'Dulse'], ['Aonori', 'Carola', 'Dulse'])
>>> macroalgae[2] = "Hijiki"
>>> seaweed, macroalgae
(['Aonori', 'Carola', 'Hijiki'], ['Aonori', 'Carola', 'Hijiki'])
```

这是因为，默认情况下，在赋值可变数据时，Python 会使用浅复制（shallow copying）机制。通过带一个由整个列表构成的切片可以强迫 Python 执行深复制（deep copying）。

```
>>> seaweed = ["Aonori", "Carola", "Dulse"]
>>> macroalgae = seaweed[:]
>>> seaweed, macroalgae
(['Aonori', 'Carola', 'Dulse'], ['Aonori', 'Carola', 'Dulse'])
>>> macroalgae[2] = "Hijiki"
>>> seaweed, macroalgae
(['Aonori', 'Carola', 'Dulse'], ['Aonori', 'Carola', 'Hijiki'])
```

切片总会复制已切除的各个元素，无论它们是列表的一部分，还是它们本来就是整个列表的，就像在这里所看到的那样。然而，这样做只会产生一个层次的复制，即深复制，如果有一个由多个列表构成的列表，子列表将只会被浅复制。一些其他的集合类型——例如字典 dict——就提供了 copy() 方法，该方法就相当于自己的 [:] 运算符。

对可用于任何深度的深复制来说，必须引入 copy 模块并使用 deepcopy() 函数。尽管在实际应用中，这样做很少出现什么问题，但如果这样做真是出了问题，则可以利用 deepcopy() 的排序输出。

对一个切片使用 del 可以依次删除多个连续元素，或者通过将 0 长度的切片赋值给切片也可以删除其元素。要插入多个元素，可以使用切片操作，也可以用运算符 + 进行切片，还可以使用 extend() 向末尾添加元素。有关列表操作中可用的方法和函数的总结，可以参阅表 1.4。

表 1.4 列表的部分方法和函数

语法	说明
x in L	如果元素 x 在 list L 中，返回 True
x not in L	如果元素 x 不在 list L 中，返回 True
L + m	返回一个含有 list L 和列表 m 全部元素的列表；extend() 方法可以完成同样的事情且效率更高
len(L)	返回 list L 的长度
L.count(x)	返回 list L 中元素 x 出现的次数
L.index(x)	返回 list L 中元素 x 出现处最左侧的位置索引值，或者抛出 ValueError 异常
L.append(x)	把元素 x 追加到 list L 的末尾处
L.extend(m)	把列表 m 的所有元素都追加到 list L 的末尾处
L.insert(i, x)	把元素 x 插入到 list L 中位置索引值为 int i 的地方

(续表)

语法	说明
L.remove(x)	从 list L 中移除最左侧出现的元素 x，或者在没找到 x 的时候，抛出 ValueError 异常
L.pop()	返回并从 list L 中移除最右侧的元素 x
L.pop(i)	返回并从 list L 中移除位置索引值为 int i 的元素
L.reverse()	依次逆序排列 list L 中的各元素
L.sort()	依次排序 list L 中的各元素；这个方法可以带一些可选参数，比如比较函数或者带一个便于进行 DSU(decorate、sort、undecorate，即封装、排序、解封)排序的“键值(key)”

### 1.4.3 字典

字典 dict 类型是一个数据字典，也称为关联数组( associative array )。字典可以保存无序的键-值对(key-value pair)集，并可提供快速的键查询功能。键(key)是唯一的，并且必须是不可变的类型，如 Python 字符串、数字或者元组；值(value)可以是包括集合类型在内的任何类型，因此可以创建出任意的嵌套数据结构。尽管字典不是序列，但正如下一章将要看到的那样，还是可以得到它们的键值序列的。

类似的数据结构也同样存在于其他语言中——例如，Perl 中的 hash、Java 中的 HashMap 以及 C++ 中的 unordered\_map 等。

值得注意的是，元组可以作为字典的键，但列表不可以，因为字典的键必须是不可变的。在那些只可以提供诸如字符串和数字等简单键的语言中，程序开发人员要想获得多元素键，就必须借助于将其多个元素转换成一个字符串，但由于 Python 具有元组这种数据类型，在 Python 中就不再需要做这种转换了。

这里给出了一些例子，给出了如何创建字典并对字典中的元素进行访问的示例：

```
>>> insects = {"Dragonfly": 5000, "Praying Mantis": 2000,
   "Fly": 120000, "Beetle": 350000}
>>> insects
{'Fly': 120000, 'Dragonfly': 5000, 'Praying Mantis': 2000,
 'Beetle': 350000}
>>> insects["Dragonfly"]
5000
>>> insects["Grasshopper"] = 20000
>>> insects
{'Fly': 120000, 'Dragonfly': 5000, 'Praying Mantis': 2000,
 'Grasshopper': 20000, 'Beetle': 350000}
```

从字典中删除各个元素的方法与从列表中删除元素的方法相同，如表 1.5 所示。例如

```
>>> del insects["Fly"]
>>> insects
{'Dragonfly': 5000, 'Praying Mantis': 2000, 'Grasshopper': 20000,
 'Beetle': 350000}
>>> insects.pop("Beetle")
350000
>>> insects
{'Dragonfly': 5000, 'Praying Mantis': 2000, 'Grasshopper': 20000}
```

表 1.5 字典的部分方法和函数

语法	说明
x in d	如果元素 x 在 dict d 中, 返回 True
x not in d	如果元素 x 不在 dict d 中, 返回 True
len(d)	返回 dict d 中元素的个数
d.clear()	从 dict d 中移除全部元素
d.copy()	返回 dict d 的浅复制
d.keys()	返回一个 dict d 中全部键的列表
d.values()	返回一个 dict d 中全部值的列表
d.items()	返回 dict d 中全部 (key, value) 键值对的元组列表
d.get(k)	返回键 k 的值, 或者不存在时返回 None
d.get(k, x)	如果键 k 在 dict d 中, 返回键 k 的值; 否则, 返回 x
d.setdefault(k, x)	与 get() 方法相同, 如果该键 k 没在 dict d 中, 那么就用给定的该键 k 和值 None 作为新的键值对元素插入, 或者在给定值 x 时, 在该键 k 处插入值 x
d.pop(k)	返回并移除键 k 所对应的元素; 如果 dict d 中没有该键, 就抛出 KeyError 异常
d.pop(k, x)	如果键 k 在 dict d, 返回并移除键 k 所对应的元素; 否则, 返回 x

使用构造函数 `dict()` 可以创建字典, 而如果各键恰是有效的标示符(也就是说, 是以字母开头且不带空格的字符数字), 就可以用更为简便的语法了:

```
>>> vitamins = dict(B12=1000, B6=250, A=380, C=5000, D3=400)
>>> vitamins
{'A': 380, 'C': 5000, 'B12': 1000, 'D3': 400, 'B6': 250}
```

之前已经提到, 字典的键值可以是元组; 这里是最后一次用例子给出这种情况的示例:

```
>>> points3d = {(3, 7, -2): "Green", (4, -1, 11): "Blue",
... (8, 15, 6): "Yellow"}
>>> points3d
{(4, -1, 11): 'Blue', (8, 15, 6): 'Yellow', (3, 7, -2): 'Green'}
>>> points3d[(8, 15, 6)]
'Yellow'
```

在第 2 章中, 将会看到如何以任意的顺序来遍历字典, 也会看到如何按照键的顺序进行遍历。

#### 1.4.4 集

Python 提供了两种 `set` 集类型: 集 `set` 和原封集 `frozenset`。两者都是无序的, 因此, 两者都不是序列。集 `set` 是可变的, 所以可进行元素的添加和删除。原封集 `frozenset` 是不可变的, 不可以改变其元素; 不过, 这意味着原封集 `frozenset` 适合用做字典的键。

集 `set` 中的每个元素都是唯一的; 如果想要在集 `set` 中添加一个已经存在了的元素, 调用 `add()` 函数也不会做任何事。如果两个集 `set` 含有相同的元素, 那么不管两个集 `set` 中各个元素插入的顺序是什么, 这两个集 `set` 都是相等的。相反, 列表 `list` 则会保留各个元素插入时的顺序(除非对它们进行了排序), 所以列表允许重复。

原封集 `frozenset` 的创建只需用一个序列参数——例如, 一个元组或者一个列表即可。集 `set` 也可以用相同的方式创建。例如

```
>>> unicorns = set(("Narwhal", "Oryx", "Eland"))
>>> "Mutant Goat" in unicorns
False
>>> "Oryx" in unicorns
True
```

由于这里创建的是一个集 set 而不是一个原封集 frozenset，故而可以添加和移除各个元素。例如

```
>>> unicorns.add("Mutant Goat")
>>> unicorns
set(['Oryx', 'Mutant Goat', 'Eland', 'Narwhal'])
>>> unicorns.add("Eland")
>>> unicorns
set(['Oryx', 'Mutant Goat', 'Eland', 'Narwhal'])
>>> unicorns.remove("Narwhal")
>>> unicorns
set(['Oryx', 'Mutant Goat', 'Eland'])
```

这些集 set 类也支持标准的集运算操作——例如，并集(union)、交集(intersection)以及差集(difference)——Python 为这些运算操作中的一些既提供了方法又提供了运算符，详见表 1.6 所示。

表 1.6 集 set 的部分方法和函数

语法	说明
x in s	如果元素 x 在 set s 中，返回 True
x not in s	如果元素 x 不在 set s 中，返回 True
len(s)	返回 set s 中元素的个数
s.clear()	移除 set s 中的全部元素
s.copy()	返回 set s 的一个浅复制
s.add(x)	如果 set s 中还没有元素 x，把 x 加入 s 中
s.remove(x)	从 set s 中移除元素 x，或者如果 s 中没有 x，抛出 KeyError 异常
s.discard(x)	如果元素 x 在 set s 中，从 s 中移除 x
s.issubset(t)	如果 set s 是 set t 的一个子集，返回 True
s <= t	
s.issuperset(t)	如果 set s 是 set t 的一个超集，返回 True
s >= t	
s.union(t)	返回一个新集，其每个元素全部来自 set s 和 set t
s   t	
s.intersection(t)	返回一个新集，其每个元素是 set s 和 set t 中都有的元素
s & t	
s.difference(t)	返回一个新集，其每个元素是 set s 中有但 set t 中没有的元素
s - t	

## 1.5 内置函数

正如之前已经看到的，Python 有许多内置函数和运算符，例如，`del`、`print`、`len()` 和 `type()`。在表 1.7 至表 1.9 中，分别额外给出了其中一些比较有用的内置函数和运算符，这里仅对其中的部分做些讨论。

表 1.7 与序列相关的部分内置函数和运算符

语法	说明
all(q)	如果 q 中的全部元素都是 True，返回 True；q 是一个迭代器——例如，是一个诸如字符串或者列表之类的序列
any(q)	如果 q 中的任一元素是 True，返回 True
x in q	如果元素 x 在序列 q 中，返回 True；这一条同样适用于字典

(续表)

语法	说明
x not in q	如果元素 x 不在序列 q 中, 返回 True; 这一条同样适用于字典
len(q)	返回序列 q 中元素的数量; 这一条同样适用于字典
max(q)	返回序列 q 中的最大元素
min(q)	返回序列 q 中的最小元素
sum(q)	返回序列 q 中元素的总和

表 1.8 一些有用的内置函数和运算符

语法	说明
chr(i)	返回一个单字符 str, 其 ASCII 值由 int i 给定
unichr(i)	返回一个单字符 unicode, 其 Unicode 码由 int i 给定
ord(c)	如果 c 是一个单字符 str 字符串, 返回其 int 字节值(0 ~ 255), 或者如果 c 是一个单字符 unicode 字符串, 返回 Unicode 代码点的 int
dir(x)	返回一个对象 x 大部分属性的列表, 包括它全部的方法名
help(x)	在 IDLE 中, 输出对象 x 类型的一段简要说明及其包含所有它的全部方法的属性列表
hasattr(x, a)	如果对象 x 有一个 a 属性, 返回 True
id(x)	返回对象引用 x 所引用对象的唯一 ID
isinstance(x, C)	如果 x 是类 C 的一个实例, 或者是类 C 的一个子类, 返回 True
type(x)	返回 x 的类型; 由于可以考虑继承关系, isinstance() 要更好些; type() 更常用于调试
eval(s)	返回计算字符串 s 后的结果, s 可以包含任意的 Python 表达式
open(f, m)	以模式 m 的方式打开字符串 f 所给定的文件, 返回文件句柄; 相关内容见第 6 章
range(i)	返回一个 int 列表, int 成员的编号从 0 到 i - 1; 可选的参数用来说明开始、结束和步进值

表 1.9 与数学相关的部分内置函数和运算符

语法	说明
abs(n)	返回数字 n 的绝对值
divmod(i, j)	返回一个元组, 元组由 i 除以 j 的商数和余数构成
hex(i)	返回一个表示数字 i 十六进制的字符串
oct(i)	返回一个表示数字 i 八进制的字符串
float(x)	返回转换成 float 的 x; x 可能是字符串, 也可能是数字
int(x)	返回转换成 int 的 x; x 可能是字符串, 也可能是数字
long(x)	返回转换成 long 的 x; x 可能是字符串, 也可能是数字
pow(x, y)	返回 x 的 y 次方; 可以接受第三个模组参数——两个参数的形式与使用运算符 ** 的效果一样
round(x, n)	按照 n 给定的位数, 对 float x 小数点之后的数字进行圆整, 返回给圆整值

在 IDLE 中, 或者在直接使用 Python 解释器时, 都可以使用 help() 函数获得一个对象的信息, 或者进入 Python 的交互式帮助系统中。例如

```
>>> help(str)
```

这样就会把 str 类的全部方法及每个方法的简要说明信息都显示出来。这里会给出很多信息, 因此通常需要使用 PageUp 键或者使用滚动条进行翻页。

```
>>> help()
```

这种不带参数的 `help()` 函数，将会进入交互式帮助系统。输入 `quit`，就可以返回到常用的 IDLE 交互窗口中。

一旦对 Python 的各个类有所熟悉，在需要快速回顾时，就可以使用 `dir()` 函数获得类的方法的简要列表，例如

```
>>> dir(str)
```

在第 2 章介绍循环时将会看到 `range()` 函数，在第 6 章介绍文件读取和写入时将会看到 `open()` 函数。在第 3 章还会介绍到 `hasattr()` 和 `isinstance()` 函数。

对于那些与序列相关的函数，`max()` 和 `min()` 可用于含有字符串的序列，也可用于数字序列，但可能会得出惊奇的结果：

```
>>> x = "Zebras don't sail"
>>> max(x), min(x)
('t', ' ')
```

这个顺序主要是根据字符串 `str` 的字节值和 `unicode` 字符串的代码点 (code point) 计算出来的。例如，`ord("z")` 的结果是 90，而 `ord("t")` 的结果是 116。

某些 Python 的内置数学函数可以参阅表 1.9。Python 还提供了一个数学库，其中含有可能需要用到的全部标准函数。通过导入 `math` 模块并使用 `dir()`，可以看出有哪些函数：

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2',
'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod',
'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

前三个元素是些特殊方法(开头和结尾都有两个下画线)；在第 3 章中，还会看到有关特殊方法的更多介绍。除 `math.e` 和 `math.pi` 这两个是常量之外，其他剩下的都是函数。通过交互方式，可以找出某一元素的类型。例如

```
>>> import math
>>> type(math.pi), type(math.sin)
(<type 'float'>, <type 'builtin_function_or_method'>)
>>> math.pi, math.sin
(3.1415926535897931, <built-in function sin>)
>>> math.sin(math.pi)
1.2246063538223773e-16 ①
```

乍一看，用这种交互方式弄明白 Python 提供的东西非常有用，不过，通过阅读 Python 文档，特别是粗略泛读“库引用”(Library Reference)中的内容，将会更进一步有助于理解 Python 的标准库到底应该提供了什么。

## 小结

在这一章，看到了赋值使用运算符 `=`、数值运算使用运算符 `+`(包括整型 `int` 做类型转换为浮点数 `float`)和增量赋值使用 `+=` 的各个用法。此外，也看到了 `print` 运算符，了解到

① 值 `0.00000000000000012246063538223773` 与预期的 0 很接近。

IDLE会自动输出表达式，因此在使用 IDLE 时会很少用到 `print`。还看到注释是以`#`开始的，并会一直持续到该行的结束。实际上，Python 可以用分号进行语句的分隔，但却很少这样做：在 Python 中，一个语句会占用单独的一行；新行就是语句的分割符。

这一章也学习了通过引号创建 Python 的字符串，以及如何使用运算符`[ ]`和`+`对字符串进行切片和连接。也总结了 Python 字符串所提供的一些关键方法；在这一本书中，会有很多有关这些方法应用的例子。我们看到，`QString` 是一个不同的 Unicode 字符串类型，因此需要制定一个策略，以便能够在使用 PyQt 进行编程时统辖 `QString` 字符串和 Python 字符串（通常是 `unicode` 字符串）。

这一章介绍了 Python 的主要集合类型。元组（tuple）为元素组合到一起提供了极好的方法，并可用作字典的关键字。列表（list）是有序的，可以保存重复的元素。列表提供了快速的插入和删除功能，以及基于索引的快速查询功能。字典（dict）是无序的，具有唯一的键。与列表相似，字典提供了快速的插入和删除功能。字典还提供基于键的快速查询功能。集（set）可看成是不带值的字典。在本书的后续内容中，还会大量使用到所有这些类型。

最后，大致看了下 Python 的内置函数和一个数学模块。在第 3 章中，将会看到如何创建我们自己的模块。但在此之前，需要先知道 Python 的控制结构，以便可以对我们的函数进行分支、循环、调用和处理异常——所有的这些将成为下一章的各个主题。

## 练习题

这一章以及整本书中练习题的目的，都是为了鼓励能够大家多去试试 Python，从第二部分 PyQt 以后，各个练习题都是为了让大家掌握一些实践经验。在设计这些练习题时，都会尽量减少代码的输入，其挑战性也会从低到高循序渐进。

这一章的练习题可以直接在 IDLE 中进行试验；从第 2 章开始，它们都会稍稍长一些，可能就需要写入到文件中，这在随后会做进一步的说明。

1. 运行 IDLE，键入如下代码：

```
one = [9, 36, 16, 25, 4, 1]
two = dict(india=9, golf=17, juliet=5, foxtrot=61, hotel=8)
three = {11: "lima", 13: "kilo", 12: "mike"}
```

试着预测一下，`len()`、`max()`、`min()` 和 `sum()` 函数对于三个集合都分别会依次产出什么结果，然后实践一下这些函数并查看它们的结果。它们的结果和预期的一样么？

2. 继续在 IDLE 中进行实验，把字典的键赋给两个变量，然后像下面那样修改其中的一个：

```
d = dict(november=11, oscar=12, papa=13, quebec=14)
v1 = v2 = d.keys()
v1, v2 # This will show the contents of the lists
v1[3] = "X"
```

修改之后，认为 `v1` 和 `v2` 是一样的还是不一样的？为什么？输出 `v1` 和 `v2` 的值看看结果是什么。现在，试试分别给 `v1` 和 `v2` 进行赋值，再次改变其中的一个

```
v1 = d.keys()
v2 = d.keys()
v1[3] = "X"
```

这次的 v1 和 v2 会和之前一样吗？输出它们的结果看看。如果对这些结果中的任何一个感到不解时，可以试着重新阅读“浅复制和深复制”中的内容。

3. 在 Python 文档中，与字符串相关的方法和函数会出现在好几个地方——找到这些页面，读读这些页面，或者大致浏览一下：序列类型（Sequence types）、字符串方法（String methods）、字符串格式化操作方法（String formatting operations）和字符串常量（String constants）。

如果熟悉正则表达式（regular expression），也可以看看正则表达式操作（Regular expression operations）的内容。

还是在 IDLE 中，创建两个浮点数值：

```
f = -34.814  
g = 723.126
```

在阅读字符串格式化相关文档的基础上，使用% 运算符创建个单一格式化字符串，使其能够在应用于 f 时输出 < -34.81 >，而在用于 g 时可以输出 < +723.13 >。

这些练习题的答案以及书中示例的全部源代码均可以在作者的个人网站中下载到，网址是 <http://www.qtrac.eu/pyqtbook.html><sup>①</sup>。在 pyqtbook.zip 文件（以及在 pyqtbook.tar.gz 文件）中会有一些子目录，如 chap01、chap02，等等，在这些子目录中会有相应的例子和答案。这一章的答案是 chap01/answers.txt。

---

<sup>①</sup> 与本书配套的国内站点为 <http://www.qtcn.org/pyqtbook/>——译者注。

## 第2章 控制结构

- 条件分支
- 循环
- 函数
- 异常处理

要编写程序代码需要有数据类型、变量和保存变量的数据结构，也需要控制结构（control structure），比如分支和循环等，以提供程序流程和遍历的控制功能。在这一章中，将学习如何使用 Python 的 `if` 语句，如何使用 `for` 和 `while` 循环进行程序的循环。异常(exception)可以影响到流程的控制，所以也会学习异常的处理和异常的创建。

功能封装的一个基本方法是把功能放到函数和方法之中。这一章会讲述如何定义函数，下一章则会讲述如何定义类和方法。C++ 或者具有类似背景的程序开发人员习惯于把函数只定义一次。在 Python 中也是如此，不过，还有一种例外的可能性：在 Python 中，可以在运行时创建函数，这是一种能够反映当前状态的函数创建方式，所以会在本章的后续内容中看到。

在上一章中，我们使用 IDLE 来试验了一些 Python 代码片段。在这一章，基本仍旧是简单地给出一些本应当写到文件中并作为程序一部分的代码片段。然而，将本章中用到的这些代码片段键入 IDLE 并“实时”看到结果也是完全有可能的，对于那些一时尚难以确定的任何相关内容都这么做，当然也是非常值得的。

Python 的一些函数和运算符可用于布尔值。例如，如果二元运算符 `in` 的左操作数在其右操作数内，就会返回 `True`。同样，`if` 语句和 `while` 语句能够估测传给它们的表达式，这些很快就会看到。

在 Python 中，如果一个值的预定义是常量 `False`、数字 `0`、特殊对象 `None`、空序列（比如，一个空字符串或者一个空列表）、空集合，该值的计算结果都会是 `False`；否则，该值的结果就是 `True`。

在 PyQt 中，一个空 `QString` 和任意的 `null` 对象，也就是说，任何 PyQt 数据类型的对象都有一个 `isNull()` 方法（且这里的 `isNull()` 返回 `True`），计算结果都会是 `False`。例如，一个空的 `QStringList`、一个空的 `QDate`、一个空的 `QDateTime` 以及一个空的 `QTime`，其值都是 `False`。相应地，非空和非 `null` 的 PyQt 对象的结果都是 `True`。

通过将任何对象转换成 `bool` 类型，可以测试并看到这些对象的布尔值。例如

```
from PyQt4.QtCore import *
now = QDate.currentDate()
never = QDate()
print bool(now), bool(never)    # Prints "True False"
```

不带参数的 `QDate()` 构造函数会创建一个空的日期；`QDate.currentDate()` 静态方法会返回当前的日期，当然，这不会是一个空值。

表 2.1 中给出了部分逻辑运算符。

表 2.1 逻辑运算符

组别	运算符	说明
Comparison	< , <= , == , != , >= , >	运算符 < > 还被认为是 != 的同义词，不过已经被废弃不用了
Identity	is , is not	这些用来判断两个对象引用是否引用到了同一个隐含对象上
Membership	in , not in	这些用在列表、字典和字符串上，用法参见第 1 章
Logical	not , and , or	and 和 or 都是短路逻辑 <sup>①</sup> ；它的位的等效数 (bit-wise equivalent) 有：~(not) 、&(and) 、 (or) 和 ^(xor)

## 2.1 条件分支

Python 提供了一个和 C++ 、 Java 等语言具有相同语义的 if 语句，尽管它有着自己的稀疏语法形式

```
if expression1:
    suite1
elif expression2:
    suite2
else:
    suite3
```

首先要提醒 C++ 或者 Java 程序开发人员的是，Python 的 if 语句没有圆括号和大括号。另外一点要注意的是，冒号(:) 是语法的一部分，但在一开始的时候，人们总是很容易忘记它。冒号通常会用于 else 、 elif 以及其他很多地方，以说明后续还有代码块 [ 用 Python 的话来说，称为套件 (suite) ] 。正如所料，还可以有任意数量的 elif (包括 none) ，以及在结尾处还可能会有一个 else 。

与大多数其他编程语言不同，Python 使用缩进来表示它的块结构。一些程序开发人员不大喜欢这一点，至少是在一开始不喜欢，一些人甚至会对这个问题变得相当情绪化。但这只不过是需要些时日去适应罢了，几个月之后，阅读没有大括号的代码貌似就会比带有大括号而显得乱七八糟的那些代码要更简洁、更漂亮。

由于各类套件 (suite) 都是使用缩进来表示的，所以自然就会提出一个问题，“是什么样的缩进呢？” Python 风格指南建议，每个缩进层次用 4 个空格，而且只使用空格 (不要使用 Tab 键) 。大多数的现代文本编辑器都可以进行设置以便能够自动处理这种情况 (当然，IDLE 的编辑器也可以设置) 。任何数量的空格或 Tab ， Python 都可以工作良好，但前提是所有的缩进都使用了一致的数量。在本书中，关于空格的数量将会遵循官方的 Python 指南。

先以一个非常简单的示例开始

```
if x > 0:
    print x
```

在这个例子中，套件只有一个语句 (print x) 。一般来说，一个套件就是一个单语句，或者是一个缩进的语句块 ( 它们本身可能还会含有一些嵌套的套件 ) ，或者是一个的确什么都没做的关键字 pass 。之所以需要关键字 pass 是因为，Python 的语法需要有一个这样的套件，因此，

<sup>①</sup> 短路 (short-circuit) 逻辑又叫惰性求值 (lazy evaluation) ；逻辑表达式 x and y 中，如果 x 为假，表达式立即返回 x 的值；否则，表达式即刻返回 y 的值——这种行为看起来，好像第 2 个值被“短路”了——译者注。

如果希望能够有一个标记，或者是希望说明正在处理一个“什么也没做”的状态，就必须用一些东西来表明这些个情况，因此，Python 就提供了 `pass`；例如

```
if x == 5:  
    pass    # do nothing in this case
```

### 避免空悬 else 陷阱

使用缩进还有一个好处是，在 Python 中几乎不可能存在所谓的“空悬 else”(dangling-else)歧义问题<sup>①</sup>。例如，下面是一些C++代码：

```
if (x > 0)  
    if (y > 0)  
        z = 1;  
    else  
        z = 5;
```

上述代码片段中，如果 `x` 和 `y` 都大于 0，`z` 会被设置为 1，不过看起来好像是，如果 `x` 是小于或等于 0，`z` 会被设置成 5。但事实上，只有在 `x` 大于 0 且 `y` 是小于或等于 0 时，`z` 才会设置成 5。这里给出了用 Python 表示这一含义的代码

```
if x > 0:  
    if y > 0:  
        z = 1  
    else:  
        z = 5
```

同时，如果真是想在 `x` 小于或等于 0 时把 `z` 设置为 5，就应该这样写

```
if x > 0:  
    if y > 0:  
        z = 1  
    else:  
        z = 5
```

正式借助 Python 基于缩进的代码段结构，有效避免了掉进“空悬 else”的陷阱。

通常情况下，不管什么时候，Python 语句中的冒号后面都会跟一个套件，如果这个套件只是一个简单语句，两者可以出现在同一行。例如

```
if x == 5: pass
```

如果语句是两个或者两个以上的语句，就必须在下一个层次的缩进行开始。

Python 支持标准比较运算符，而对于逻辑运算符，Python 会使用名字(`not`、`and`以及`or`)而不是符号进行比较操作。此外，也可以用数学中熟悉的方式对比较运算符进行组合

```
if 1 <= x <= 10:  
    print x
```

在这里，如果 `x` 是处于 1 ~ 10 之间，就输出 `x`。如果 `x` 是一个不带其他作用的表达式，上面的语句就等同于

```
if 1 <= x and x <= 10:  
    print x
```

<sup>①</sup> 空悬 else(dangling-else)问题是指：一个 `else` 子句只能与其代码之前的其中一个 `if` 对应，但如果只有 `if` 而缺少 `else`，或者只有 `else` 而没有与之对应的 `if`，又或者编写代码时没有注意到 `if...else` 两者之间的逻辑对应关系，就可能会产生歧义——译者注。

推荐采用第一种格式：因为这种方式更简洁、更简单、更有效（因为 `x` 可能会是一个含有某种较为复杂计算的表达式），此外，这种方式也更容易维护（还是因为 `x` 只用了一次而不是两次）。

Python 使用 `elif` 和 `else` 来提供多种形式的分支；Python 中没有 `case`（或者 `switch`）语句。

```
if x < 10:
    print "small"
elif x < 100:
    print "medium"
elif x < 1000:
    print "large"
else:
    print "huge"
```

Python 2.5 引入了条件表达式。这是一种可以用于表达式中的 `if` 语句，它相当于其他语言中所使用的三元运算符。不过，该表达式的 Python 语法与使用“`? :`”三元运算符的 C++ 和 Java 语言相当不同，它的形式是：`trueResult if expression else falseResult`（即条件为真时的输出结果，`if` 条件表达式，`else` 条件为假时的输出结果）；因此，表达式是在中间的

```
print "x is zero or positive" if x >= 0 else "x is negative"
```

这行代码会在 `x >= 0` 为 `True` 时，输出“`x is zero or positive`”；否则，它将输出“`x is negative`”<sup>①</sup>。

## 2.2 循环

Python 提供了两种循环（loop）结构。一个是 `while` 循环，其基本语法是

```
while expression:
    suite
```

这是一个例子

```
count = 10
while count != 0:
    print count,
    count -= 1
```

由于在 `print` 语句后面跟的有逗号，所以这里会在同一行打印出“10 9 8 7 6 5 4 3 2 1”。值得注意的是，必须在每个缩进的套件前带一个冒号。

循环可以被提前结束，这可以使用 `break` 语句。这在循环中来说是特别有用的，否则，由于它们的条件表达式总是为 `true`，也就是说，循环会一直进行下去

```
while True:
    item = getNextItem()
    if not item:
        break
    processItem(item)
```

Python 的 `while` 循环也可以与 `else` 语句联合使用，具体可以使用以下语法：

```
while expression:
    suite1
else:
    suite2
```

<sup>①</sup> 文档“What’s New in Python”的作者 Andrew Kuchling 建议在使用条件表达式的地方总是借助括号。不过在本书中，只有在必要的时候，才会用到括号。

`else` 子句(及其相关套件)是可有可无的。如果循环在某一条件下终止, 而不是由 `break` 语句提前终止的, 就会执行 `else` 子句。不过, `else` 并不经常用到, 不过在某些情况下却很有用

```
i = 0
while i < len(mylist):
    if mylist[i] == item:
        print "Found the item"
        break
    i += 1
else:
    print "Didn't find the item"
```

`while` 循环的用途非常广泛, 但由于经常需要用它来遍历列表上的所有元素, 或者对某个特定的数循环多次, 在这些情况下, Python 就提供了另外一种更为简便的循环结构。这就是 `for` 循环, 它的语法是

```
for variable in iterable:
    suite1
else:
    suite2
```

`else` 的工作方式与 `while` 循环中的 `else` 工作方式相同, 也就是说, 如果 `for` 循环完成, 就会执行它的套件, 但如果是被 `break` 语句终止了, 就不会执行它的套件。`iterable` 是一个可被遍历的对象, 比如一个字符串、一个元组、一个列表、一个字典或者是一个迭代器[比如一个生成器(generator), 将在后面讲到]。如果是一个字典, 迭代的是字典的键。

这里对一个字符串进行遍历, 也就是说, 遍历字符串中的每个字符

```
for char in "aeiou":
    print "%s=%d" % (char, ord(char)),
```

这样会打印“`a = 97 e = 101 i = 105 o = 111 u = 117`”。变量 `char` 会依次遍历循环变量中的每个值(在该例中就是先“`a`”, 然后是“`e`”, 以此类推, 一直到“`u`”), 每个遍历值都会执行相应的套件。

`range()` 内置函数会返回一个整数列表, 因而可以方便地用于 `for` 循环中。例如

```
for i in range(10):
    print i,
```

这样就可以打印出“`0 1 2 3 4 5 6 7 8 9`”。默认情况下, `range()` 函数会返回一个从整数 0 开始、每次加 1 直至但不包括给定值的列表。该函数还有两种形式, 分别带有两个或者三个参数

```
range(3, 7)      # Returns [3, 4, 5, 6]
range(-4, 12, 3) # Returns [-4, -1, 2, 5, 8, 11]
```

Python 还提供了一个与 `range()` 函数具有相同语义的 `xrange()` 函数, 只是在 `for` 循环中该函数的内存利用效率能够更高些, 因为在每次调用中, 它都只是简单地进行一次计算, 而不是一次就生成一个完整的整数列表。通常情况下, 都会使用 `range()`, 而只有在需要大幅提升性能时, 才会用 `xrange()` 代替 `range()`。

如果 `for` 循环的遍历值是可变的(比如是列表或字典), 必须保证在循环体内不去改变它的值。如果在遍历列表或字典时又需要更改它们的值, 就必须去遍历由列表索引值构成的列表或者是遍历由字典的键构成的列表, 或者使用浅复制, 而不能是直接对集合本身进行操作。例如

```

presidents = dict(Washington=(1789, 1797), Adams=(1797, 1801),
                  Jefferson=(1801, 1809), Madison=(1809, 1817))
for key in presidents.keys():
    if key == "Adams":
        del presidents[key]
    else:
        print president, presidents[key]

```

这样会从 presidents 字典中移除关键字“Adams”(以及其关联值)并输出

```

Madison (1809, 1817)
Jefferson (1801, 1809)
Washington (1789, 1797)

```

值得注意的是，尽管 Python 通常会使用换行符作为语句的分隔符，但在括号中却不会这么做。当在括号中创建列表或者是在方括号中创建字典时就是如此。这就是为什么 presidents 字典的构造函数可以横跨多行而其中不必使用任何一个能够表明新行的反斜杠(\)转义字符的原因。

由于字典保存着许多键值对，Python 就提供了一些键、值、键值对的遍历方法。为方便起见，如果只是简单遍历字典，甚至都不需要调用 keys() 方法来获得字典的各个键

```

presidents = dict(Washington=(1789, 1797), Adams=(1797, 1801),
                  Jefferson=(1801, 1809), Madison=(1809, 1817))
for key in presidents:
    print "%s: %d-%d" % (key, presidents[key][0], presidents[key][1])

```

这样将会输出(并不一定是指以下的顺序)：

```

Madison: 1809-1817
Jefferson: 1801-1809
Washington: 1789-1797
Adams: 1797-1801

```

在 for 循环中遍历字典，变量就会被依次设置成字典的每个键<sup>①</sup>。字典是无序的，所以就会以不确定的顺序返回字典的键。

要得到值而不是键，可以使用 values() 方法——例如，用 items() 方法和语句“for dates in presidents.values():”可以获得键值对。例如

```

for item in presidents.items():
    print "%s: %d-%d" % (item[0], item[1][0], item[1][1])

```

这会生成与之前例子相同的输出，如下所示：

```

for president, dates in presidents.items():
    print "%s: %d-%d" % (president, dates[0], dates[1])

```

在这里，会打开由 items() 方法返回的每一个键值对，dates 会成为二元素日期元组。

如果想依次序进行迭代，就必须在遍历之前对列表进行排序。例如，要以名字为序来进行遍历，可以这样做：

```

for key in sorted(presidents):
    print "%s: %d-%d" % (key, presidents[key][0], presidents[key][1])

```

无论是 for 循环还是 sorted() 函数都可以对序列或者迭代器进行处理。迭代器(iterator)就是一些能够支持 Python 迭代器协议的对象，这就意味着，它们有 next() 方法，并且可以在没有更多元素的时候抛出 StopIteration 异常。不必奇怪，其实列表和字符串就实现了这一

<sup>①</sup> 需C++/Qt 程序开发人员注意的是：Python 的 for 循环会遍历字典的键，而 Qt 的 foreach 循环则可以遍历 QMap 的值。

协议：列表迭代器(list iterator)能够依次返回列表中的每一个元素，而字符串迭代器(string iterator)则能够依次迭代字符串中的每一个字符。字典也能够支持这一协议：字典可以依次返回字典中的每一个键(会以随机的顺序返回)。因此，当对字典使用 for 循环或者是调用 sorted() 时，实际是对字典的键进行的操作。例如

```
names = list(presidents)
# names == ['Madison', 'Jefferson', 'Washington', 'Adams']
```

因此，在 for 循环中，对 sorted(list(presidents)) 的有效调用就与 sorted(presidents.keys()) 具有同样的效果。如果希望调用能够更清晰明确些，可以把这件事分解成以下几个步骤：

```
keys = presidents.keys()      # Or: keys = list(presidents)
keys.sort()
for key in keys:
    print "%s: %d-%d" % (key, presidents[key][0], presidents[key][1])
```

Python 的 sort() 方法和 sorted() 函数可以带一些可选参数。因此，例如，其实也是可以按照日期来对 presidents 字典进行排序的。

除 keys()、values() 和 items() 方法之外，字典还提供了 iterkeys()、itervalues() 以及 iteritems() 方法。这些额外的方法可以像其他几个普通方法一样使用，只是它们提供了更好的性能而已。然而，它们不能用于遍历这样的字典：在遍历的过程中，该字典的键会发生改变。

就如 while 循环一样，在迭代完成之前可以使用 break 提前结束 for 循环，也可以在两个循环之中使用 continue 而立刻跳到外层的迭代循环中。例如：

```
for x in range(-5, 6):
    if x == 0:
        continue # goes directly to the next iteration
    print 1.0 / x,
```

这样会输出类似这样的东西：“ -0.2 -0.25 -0.33333333333 -0.5 -1.0 1.0 0.5 0.33333333333 0.25 0.2”。不带 continue 的话，甚至可能会最终造成用零除，这样会导致异常。

如前所述，Python 的循环可以带一个可选的 else 子句，该子句只会在循环完成后才得以执行，也就是说，如果在循环中调用了 break 语句，else 子句就不会得到执行。用一个例子就能够把这件事说得更清楚些；这里给出的是一种低效的生成素数列表的方法：

```
primes = [2]
for x in range(2, 50):
    if x % 2:
        for p in primes:
            if x % p == 0:
                break # exits the loop and skips the else
            else:
                primes.append(x)
```

之前在看到% 运算符时，它通常用于对字符串的操作中，能够产生一个格式化字符串。而在这里，% 运算符用于整数的操作，因而在这个上下文环境下，% 运算符执行的整除运算(或者取余)操作，从而会生成一个作为其结果的整数。

在代码的最后，primes 列表的内容是 [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]。只有当 primes 列表迭代完成后才会调用 append() 方法，也就是说，只有在 x 不能被之前的任何素数除尽的时候才会执行它。

## 列表解析和生成器

用 `for` 循环联合 `range()` 生成一个列表非常容易。此外，Python 还提供了另一种备选方法，称为列表解析(list comprehension)——是一些生成列表的表达式。

先来生成一个可以被数字 5 整除的列表

```
fives = []
for x in range(50):
    if x % 5 == 0:
        fives.append(x)
# fives = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

其中，就用到了 `for` 和 `range()` 联合使用的情况。

现在，将会看到如何用列表解析生成一个包含连续数字的简单列表

```
[x for x in range(10)]
```

此时会生成一个列表，`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`。列表解析也可添加一些附属条件：

```
fives = [x for x in range(50) if x % 5 == 0]
```

这样生成的 `fives` 列表与之前的原始 `for` 循环生成的列表相同。利用 `for` 循环嵌套还可以生成更为复杂的列表解析形式，不过在那些情况下，更传统的语法可能会显得更容易理解一些。

列表解析的一个缺点是，运行一次就会生成整个列表，而如果列表很大的话，就会消耗大量的内存空间。在传统语法中也同样存在这个问题，不过可以通过使用 `xrange()` 而不是 `range()` 来解决这个问题。Python 生成器(generator)提供了另外一种解决方案。生成器表达式的用法与列表解析相似，只不过生成器采用了一种较为简易的列表生成方式。

```
fives = (x for x in range(50) if x % 5 == 0)
```

这一句几乎就和列表解析完全相同(唯一显著不同的就是，这里用的是圆括号而不是方括号)，但返回的对象却不再是列表！取而代之，返回的是一个生成器。生成器就是迭代器，因此可以像下面那样进行操作：

```
for x in (x for x in range(50) if x % 5 == 0):
    print x,
```

这样将会输出“0 5 10 15 20 25 30 35 40 45”。

在 Python 编程中，列表解析并不是完全必要的；之所以在这里讲述，只是为了能够尽可能地确保在阅读他人代码时可以分辨列表解析的内容，同时也提供一些尝试 Python 先进性的机会。在稍后使用它们时，通常会给出能够实现诸如与 `for` 循环一样效果的代码。另一方面，由于生成器是 Python 的一个高级而且相对较新的特性，所以不大可能被简单模仿。在下一节，我们会制作一个简单的生成器功能，还会在第 3 章的一个示例类中，创建一些生成器的短方法。

## 2.3 函数

一般来说，函数允许封装和参数化那些常用的功能。Python 提供了三个类型的函数：普通函数(ordinary function)、`lambda` 函数<sup>①</sup>(`lambda` function)和方法(method)。在这一节中，会主要讲述普通函数，也会简单介绍一下 `lambda` 函数；在第 3 章，再讨论方法。

① `lambda` 函数也叫匿名函数，即函数没有具体的名字——译者注。

在 Python 中，每个函数要么具有“全局”(global)作用域，要么拥有“局部”(local)作用域。一般来说，全局作用域意味着函数在定义它的文件中都是可见的，并且对于导入该文件中的任何文件也都是可以访问的。局部作用域则意味着函数是在另一个作用域内(比如，在另一个函数中)定义的，因而也只在闭合的局部作用域内是可见的。在这里，我们不会过多地关注这一问题，不过，在第 3 章还会回过头来继续讨论它。

函数的定义可以使用 `def` 语句，使用语法是

```
def functionName(optional_parameters):
    suite
```

例如：

```
def greeting():
    print "Welcome to Python"
```

函数的名字必须是有效的标识符。函数可用括号进行调用，因此，要执行 `greeting()` 函数，就应该这样做：

```
greeting() # Prints "Welcome to Python"
```

函数的名字是一个指向该函数的对象引用，与其他的对象引用一样，它可以赋值给另一个变量或者存储在某个数据结构中

```
g = greeting
g() # Prints "Welcome to Python"
```

这使得 Python 中保存列表或者函数字典变得非常简单。

接受参数的函数可以通过位置(“位置参数”，positional argument)、名字(“关键字参数”，keyword argument，不过这个关键字与 Python 语言中的关键字没有什么关系)或者两者的组合来给定各个参数的值。让我们来看一个具体的例子吧：Python 没有提供可对各个浮点数 `float` 进行操作的 `range()` 函数，因此，创建一个自己的 `range()` 函数。

```
def frange(start, stop, inc):
    result = []
    while start < stop:
        result.append(start)
        start += inc
    return result
```

如果用 `frange(0, 5, 0.5)` 的形式来调用该函数，可以正如所料得到列表 `[0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]`。

与普通 Python 变量一样，无须指明参数的类型。同时，由于没有给定任何默认参数，所以就必须给定每一个参数，否则，将会得到 `TypeError` 异常。对于那些不太熟悉默认参数的程序开发人员来说，Python 允许在函数的签名中把各值传给每一个参数。每个这样给定的值都是“默认参数”，在函数调用时，如果相应参数的值没有给定，就会用到这个值予以顶替。

在许多情况下，创建的具有一个或者多个参数的函数几乎总是拥有同一个值。对于这种情况，Python 就允许提供默认参数，因而就可以充分地利用这一点，为第三个参数提供一个默认参数，正如这个修订后的 `def` 语句行所显示的那样

```
def frange(start, stop, inc=1.0):
```

这可以很好地运行；例如，由于把增量默认成了 1.0，现在就可以调用 `frange(0, 5)` 来得到 `[0, 1.0, 2.0, 3.0, 4.0]`。与其他语言允许默认参数一样，Python 不允许不带默然参数的参数跟随一个带有默认参数的参数后面；所以，就不应该有 `frange(start = 0, 5)` 函数。

Python 也不允许有重载函数。在实践中，这些限制中哪一个都不是问题，在后续讨论关键字参数的时候，很快就会看到这一点。

遗憾的是，这里的 `frange()` 函数不能像 `range()` 那样提供相同的参数逻辑。对于 `range()` 来说，如果给定一个值，那么它就是上界，如果给定两个值，那么它们就是上界和下界，如果给定三个值，那么它们就是两个边界和一个步长增量。所以，可以创建一个 `frange()` 函数的最终版本，它可以更逼真地模仿出 `range()` 的行为<sup>①</sup>

```
def frange(arg0, arg1=None, arg2=None):
    """Returns a list of floats using range-like syntax

    frange(start, stop, inc)  # start = arg0  stop = arg1  inc = arg2
    frange(start, stop)       # start = arg0  stop = arg1  inc = 1.0
    frange(stop)             # start = 0.0  stop = arg0  inc = 1.0
    """
    start = 0.0
    inc = 1.0
    if arg2 is not None:    # 3 arguments given
        start = arg0
        stop = arg1
        inc = arg2
    elif arg1 is not None:  # 2 arguments given
        start = arg0
        stop = arg1
    else:                  # 1 argument given
        stop = arg0
    # Build and return a list
    result = []
    while start < (stop - (inc / 2.0)):
        result.append(start)
        start += inc
    return result
```

例如，`frange(5)` 会返回 `[0.0, 1.0, 2.0, 3.0, 4.0]`，`frange(5,10)` 会返回 `[5, 6.0, 7.0, 8.0, 9.0]`，而 `frange(2, 5, 0.5)` 则可以返回 `[2, 2.5, 3.0, 3.5, 4.0, 4.5]`。

循环条件与我们之前用过的是不同的。之所以这样做，主要是为了防止由于浮点数的圆整错误而不小心达到了停止值。

`def` 语句行之后，有一个三引号字符串——这个字符串没有给任何东西赋值。一个未赋值的字符串跟在 `def` 语句之后——要么这是 `.py` 或者 `.pyw` 文件中的第一行，要么这应该跟在 `class` 语句之后，这在第一部分的后续内容中将会看到这一点——这称为“文档字符串”(`docstring`)。这在文档函数中是很常见的。按照惯例，第一行会给出一个简短的总结说明，并通过空行与剩下的部分区分开来。

在本书余下内容里给出的大多数例子中，都将忽略文档字符串以便节省空间。不过，在与本书配套的源代码中的适当地方，还是可以看到这些文档字符串的。

使用 `None` 可能会更方便些，或者说，给定一个从 0 到 0 的对象，那么 0 就完全有可能恰好是一个合法的上界了。使用 `arg2 != None` 语句时，本来应当是与 `None` 进行比较的，

<sup>①</sup> 对于更为复杂的 `frange()` 函数可以参看 *Python Cookbook* 一书“Writing a range-like Function with Float Increments”一节中的详细内容(该书第三版的中文译稿可以在这里看到：[http://python3-cookbook.readthedocs.org/zh\\_CN/latest/index.html](http://python3-cookbook.readthedocs.org/zh_CN/latest/index.html))——译者注。

不过使用 `is not` 会更有效率些，而且也是一种更好的 Python 编程风格。这是因为，如果使用 `is`，就可以获得较好的比较一致性而不是仅做值之间的比较，这样就可以比较得更快些，因为此时只是比较两个地址而不需要看对象本身到底是什么。Python 有一个全局的 `None` 对象，因此，使用 `is` 或者 `is not` 与其做比较会非常快。

传递给 Python 各个函数的参数都是一些对象引用。在对诸如字符串和数字这样的不可变对象进行引用时，就可以认为这些参数都是多个对象引用，尽管它们都是按值传递的。这是因为，如果一个不可变参数在函数内发生了“改变”，该参数就会简单绑定到一个新的对象上，而它原来引用到的对象则不会发生变化。

相反，对于那些可变对象，即这些参数就是类似于列表和字典这样的可变类型对象引用，可以在函数内发生改变。这些参数传递的行为与 Java 中的方式一样<sup>①</sup>。

所有的 Python 函数都会返回值。值的返回，要么可以是通过显式地使用 `return` 或者 `yield` 语句（在下一节会谈到），要么是通过非显式的方式，此时 Python 将会返回 `None`。与 C++ 或者 Java 不同，无须专门给定返回的类型：由于返回的是一个与任意类型绑定的对象引用，所以可以返回我们想要的任何类型。Python 函数总是返回一个值，不过由于这个返回值可以是一个元组、一个列表或者是一个其他集合类型，Python 函数就可以返回任意数量的值了。

### 2.3.1 生成器函数

如果像下面的代码片段那样替换 `frange()` 函数结尾处的代码，就可以把函数转换成一个生成器(generator)。生成器没有返回语句；取而代之的是，生成器会有 `yield` 语句。如果一个生成器运行了所有的值，也就是说，如果控制流程一直运行到了函数的结尾，此时不再返回值，相反，Python 会自动抛出一个 `StopIteration` 异常

```
# Build and return a list          # Return each value on demand
result = []                         while start < (stop - (inc / 2.0)):
while start < (stop - (inc / 2.0)):    →      yield start
    result.append(start)           start += inc
    start += inc
return result
```

现在，如果调用 `frange(5)`，将会得到一个生成器对象，而不是一个列表。通过调用 `list(frange(5))`，可以强制让生成器返回一个列表。不过，生成器更为常见的用法是将其用于循环中

```
for x in frange(10):
    print x,
```

不管使用的是哪个版本的 Python，此时都将会输出“0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0”。不过，对于长列表，使用生成器这种形式的效率会更高些，因为使用列表的那种形式会在内存中创建出整个列表，而生成器则每次只创建一个元素。

`yield` 语句的行为和 `return` 语句比较类似，但也有一个至关重要的区别：在 `yield` 返回一个值之后，当下一次再次调用生成器时，生成器会从先前跟在 `yield` 之后的语句处继续执行，而之前的所有状态都不会发生改变。因此，在第一次调用 `frange()` 生成器时，假设调用的形式是 `frange(5)` 时，返回的是 0.0；那么在第二次调用时，它返回的会是 1.0，以此类推。在返回 9.0 之后，`while` 表达式的计算值为 `False`，此时函数终止。

<sup>①</sup> 在 Python 中的可变参数与 Pascal 的 `var` 参数以及与 C++ 的 `non-const` 引用相似。

因为这个函数是生成器(出现这种情况纯粹是因为使用了 `yield`)，当它完成后，并没有返回值，相反，而是抛出了一个 `StopIteration` 异常。在 `for` 循环的上下文中，`for` 会很好地处理这一异常，不会把它当成错误，而是会将其作为迭代完成的说明来加以处理，因此，`for` 循环结束并将控制流转移到 `for` 循环的 `else` 套件中，或者在没有 `else` 套件时继续执行 `for` 循环之后的语句。与此类似，如果强制将生成器放到列表中，列表的构造函数将会自动处理 `StopIteration` 异常。

生成器是一个含有 `next()` 函数的对象，因此，如果愿意，可以继续交互地研究 `frange()` 生成器的行为

```
>>> list(frange(1, 3, 0.75))
[1, 1.75, 2.5]
>>> gen = frange(1, 3, 0.75)
>>> gen.next()
1
>>> gen.next()
1.75
>>> gen.next()
2.5
>>> gen.next()
Traceback (most recent call last):
File <pyshell#126>, line 1, in -toplevel-
    gen.next()
StopIteration
```

使用 `list()` 生成了整个三元素列表，然后就像 `for` 循环一样，使用 `frange()` 返回的生成器生成了每一个后续值。

### 2.3.2 关键字参数的使用

Python 的参数处理能力是非常强大的。到目前为止，使用位置语法(positional syntax)提供过一系列参数。例如，给 `frange()` 函数的第一个参数总是传给 `arg0`，第二个总是传给 `arg1`，而第三个则总是传给 `arg2`。也曾经用到过一些默认参数，以便能够省略一些参数。但是，如果想要传入第一个和第三个参数并接受默认的第二个参数时，会发生什么呢？在下一个示例中，将会看到如何实现这一目标。

Python 提供了 `strip()` 方法从字符串的结尾处去掉空格(或者其他不想要的字符)，但它并没有提供清理字符串内部空格的函数；在从用户那里获得字符串时，这是经常需要做的事情。这里给出的这个函数既可以从字符串的末端去掉空格，又可以用单个空格取代字符串内部的每个空格序列

```
def simplify(text, space=" \t\r\n\f", delete=""):
    result = []
    word = ""
    for char in text:
        if char in delete:
            continue
        elif char in space:
            if word:
                result.append(word)
                word = ""
        else:
            word += char
```

```

if word:
    result.append(word)
return " ".join(result)

```

这个函数会遍历文本字符串中的每一个字符。如果字符是在待删除字符串(默认情况下,该字符串是空的)中,就忽略它。如果该字符是一个“空格”(也就是,它在空格字符串中),添加已构成列表的那些单词中的一个单词,并把下一个单词设置成空白。否则,把该字符添加到正在创建的单词之后。在结尾处,把最后一个处理后的单词添加到单词列表中。最后,使用 `join()` 方法,以单一字符串的形式返回单词列表,该字符串中的每个单词都由一个空格隔开。

现在,来看看该如何使用这个函数

```

simplify(" this and\n that\t too")      # Returns "this and that too"
simplify(" Washington D.C.\n",
         delete=";::")                      # Returns "Washington DC"
simplify(delete="aeiou", text=" disemvoweled ")  # Returns "dsmvwld"

```

在第一种情况中,对于空格和各个删除参数使用的是默认参数。在第二种情况下,使用 Python 的关键字参数语法来给定第三个参数,同时,第二个参数接受默认值。在最后的例子中,对打算用到的两个参数都使用了关键字语法。值得注意的是,如果使用关键字语法,关键字参数的顺序就由自己决定——假设如果也使用了位置参数,这些位置参数要放在关键字参数之前,就如第二种情况所给出的那样。

在 `simplify()` 函数中使用的代码并不是很 Python 化。例如,本应该将单词存储成一个字符列表而不是一个字符串,而且不需要空格参数,因为本来是可以使用字符串对象的 `isspace()` 方法的。在文件 `chap02/simplified.py` 中,包含了这里给出的 `simplify()`,还给出了另一个更 Python 化的类似函数 `simplified()`。正如之前提到的,尽管通常不会在书中给出文档字符串 `docstring`,但在文件中都会给出它们。

Python 的参数传递实际要比我们迄今为止所看到的更复杂。除命名参数外,也可以给 Python 函数传递签名,其中可以接受数量不等的一些位置参数和关键字参数。这要比 C++ 和 Java 的变量参数列表更通用、更强大,不过这一点却很少用到,所以不会过多赘述它。

## 在文件中测试函数

无论是 `frange()` 形式还是生成器形式 `gfrange()`,都在文件 `chap02/frange.py` 中。如果希望能够交互地尝试这些函数或者是任何其他函数,都可以启动 IDLE,把打算使用的文件所在的路径添加到 IDLE 的搜索路径中,例如

```

>>> import sys
>>> sys.path.append("C:/pyqt/chap02")

```

现在,相应的模块就可以加载到 IDLE 中

```
>>> import frange
```

打算导入的文件必须有一个`.py` 后缀,在 `import` 语句中,必定不能包含该后缀。现在,就可以在 IDLE 中使用 `frange()` 和 `gfrange()` 函数了:

```

>>> frange.frange(3, 5, 0.25)
[3, 3.25, 3.5, 3.75, 4.0, 4.25, 4.5, 4.75]

```

第一个名字 `frange` 是模块的名称,并且希望能够在这个模块中访问 `frange` 函数,这就是为

什么可以写成 `frange.frange()` 的原因。刚才就是在导入 `sys` 模块并使用 `sys.path` 访问它的 `path` 列表时，也做过同样的事情。

尽管我们更喜欢使用 IDLE，但也是有可能直接使用 Python 解释器进行交互试验的。如果只是在终端中运行 Python 自身的可执行程序（例如 `Python.exe`），可能会更习惯于使用 `>>>` 提示符，其中是可以交互式地使用解释器的。

### 2.3.3 lambda 函数

到目前为止，一直都是使用 `def` 语句定义函数的，但是，Python 为创建函数还提供了另外一种方法

```
cube = lambda x: pow(x, 3)
```

关键字 `lambda` 用来创建简单的匿名函数。`lambda` 函数既不能包含控制结构（也没有分支或者循环），也不能包含 `return` 语句：返回的值就仅仅是表达式计算后所得到的值。`lambda` 函数可以是闭包，这一点稍后会讨论到<sup>①</sup>。在上面这个例子中，把 `lambda` 函数赋给了变量 `cube`，现在就可以使用了，例如：`cube(3)`，它将返回 27。

一些 Python 程序开发人员不喜欢 `lambda`；当然，`lambda` 函数也不是必需的，因为 `def` 可以用来创建任何我们想要的函数。然而，当开始 GUI 编程后，将会看到 `lambda` 函数还是非常有用的，虽然也可以找到不使用它的备选方案。

### 2.3.4 动态函数的创建

Python 解释器会从 `.py`、`.pyw` 文件的开头开始读取数据。当解释器遇到 `def` 语句时，它就执行该语句，从而创建函数并将其与 `def` 后的名字进行绑定。任何不在 `def` 语句（或者在 `class` 语句中，将会在下一章看到）内的代码都会被直接执行。

Python 不能调用未定义的函数，也不能使用未定义的对象。因此，单独占有一个文件的 Python 程序会具有与 Pascal 类似的结构，从上到下含有许多函数定义，然后在结尾处对其中的一个函数进行调用来启动处理进程。

与 C++ 和 Java 不同的是，Python 程序没有固定的入口点，名字“`main`”也并没有什么特殊之处。Python 解释器只会从遇到的第一行代码开始执行。例如，下面是一个完整的 Python 程序：

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

def hello():
    print "Hello"

def world():
    print "World"

def main():
    hello()
    world()

main()
```

<sup>①</sup> 如果在一个内部函数里可以对外部作用域（但不是全局作用域）中的变量进行引用，那么这个内部函数就被认为是闭包（closure）。分解来说，一个闭包需要包含 3 个条件：(1) 需要有函数嵌套，就是一个函数里面再写一个函数；(2) 外部函数需要返回一个内部函数的引用；(3) 外部函数中有一些局部变量，并且，这些局部变量在内部函数中有所使用——译者注。

解释器执行 `def hello()`，也就是说，会创建 `hello()` 函数，然后创建 `world()` 函数，再然后创建 `main()` 函数。最后，解释器就到了函数调用的地方，在本例中就是 `main()` 这个地方，因此解释器会执行函数调用，通常会把这个地方认为是程序执行的开始点。

Python 程序开发人员通常只会在最上面放置一个语句，以调用打算执行的第一个函数。他们通常称这个函数为 `main()`，并从这个函数调用其他函数，从而产生一种类似用于 C++ 和 Java 中的结构形式。

由于 `def` 语句是在运行时执行的，故而会根据当时的情形而使用不同的定义形式。这种执行方法非常有用，特别是在需要使用 Python 某一早期版本中所没有的功能时，就不必强制用户升级后才能运行程序了。

例如，`sorted()` 函数是在 Python 2.4 中引入的。假设现在有一段代码需要用到 `sorted()`，但有些用户正在使用的是 Python 2.3，有些使用的是 Python 2.4 或者更新的版本，会发生什么呢？此时，对于 Python 2.4 及其更新版本的用户来说，仅需直接借助 `sorted()` 方法即可，但可以为使用旧版本 Python 的用户们提供相类似的函数即可

```
import sys
if sys.version_info[:2] < (2, 4):
    def sorted(items):
        items = list(items)
        items.sort()
        return items
```

一开始是导入 `sys` 模块，这会提供一个 `version_info` 元组。然后，用这个元组就可以获得主副版本号数字。仅仅在版本号数比 2.4 更低时才会定义我们自己的 `sorted()` 函数。需注意的是，可以对元组进行比较：Python 可以比较数据结构，包括嵌套的各类数据结构，前提是它们所包含的数据类型是可以进行比较的。

### 2.3.5 偏函数应用程序

在开始 GUI 编程时将会看到，有些时候，有些地方需要调用特殊的函数，不过在编写那段代码的时候确实知道那些参数中的一个是什么。例如，或许会有数个需要调用同一个函数的按钮，不过，参数化的形式是由能够发起这一调用的那个特定按钮决定的。

最简单的例子是希望能够保存一个可在后续随时调用的函数（也就是，一个指向函数的对象引用）。像这样保存的函数称为回调函数（callback）。这里给出一个简单的示例

```
def hello(who):
    print "Hello", who
def goodbye(who):
    print "Goodbye", who
funclist = [hello, goodbye]
# Some time later
for func in funclist:
    func("Me")
```

这样就会先打印出“Hello Me”，然后再打印出“Goodbye Me”。在这里，就是先保存了两个函数然后在后面调用了它们。值得注意的是，在每次调用 `func()` 函数的时候，我们传递的都是同一个参数，即“Me”。由于之前就已经知道参数是什么，所以最好是把要调用的函数和打算使用的参数都封装进单个可调用对象中。

对于这一问题的一种解决方案是偏函数应用程序[也称为“科里化”(currying)]，这仅意味着，需要一个函数以及该函数的零个、一个或者多个参数，并且将它们封装成一个新的函数，在调用的时候，该函数会用封装了的参数以及调用时的其他参数一起传递给它的最初函数。这样封装的函数称为闭包，因为在创建它们的时候，封装了一些调用上下文的参数。

为了能够感受到这是如何工作的，让我们设想一个非常简单的 GUI 程序，其中有两个按钮，在按下时，将会调用同一个 `action()` 函数(目前先不要担心是如何把按下按钮的动作转换成函数调用的；这非常容易，会在第 4 章中给出详细的解释)。

```
def action(button):
    print "You pressed button", button
```

现在，在创建按钮时，自然知道它们哪个是哪个，因此希望第一个按钮调用 `action("One")` 而第二个按钮调用 `action("Two")`。不过，这样做会带来一个问题。我们知道想调用的是谁，但是不希望直到按钮按下了才让调用发生。因此，例如，希望能够给第一个按钮一个函数，其中封装了 `action()` 和参数"One"，以便能够在按下第一个按钮的时候可以用正确的参数来调用 `action()`。

因此，我们所需要的就是一个函数：它带一个函数、一个参数并返回一个函数，也就是说，在被调用时，它就会用 Python 的初始参数来调用其最初函数。在 Python 2.5 中，可以很轻松地假设出前一个 `action()` 的定义

```
import functools
buttonOneFunc = functools.partial(action, "One")
buttonTwoFunc = functools.partial(action, "Two")
```

`functools.partial()` 函数带一个函数来作为第一个参数，然后是任意数量的其他参数，在返回一个函数，在被调用时，将会用传递的参数和调用时刻所给定的其他任意参数一起来调用那个被传递的函数。

因此，在调用 `buttonOneFunc()` 时，就会正如我们所希望的那样简单调用 `action("One")`。正如之前提到过的那样，函数的名字只是一个用来指向函数的对象引用，因此可以向任何其他的对象引用一样作为参数进行传递。

不过，对于 Python 早期版本的老用户们该如何做这些事呢？本来应当是可以提供一个非常简单、功能也不是那么强的自定义 `partial()` 的。例如

```
def partial(func, arg):
    def callme():
        return func(arg)
    return callme
```

在 `partial()` 函数内，创建了一个内部函数 `callme()`，也就是说，在调用 `partial()` 时，就会调用 `callme()` 函数和传给 `partial()` 函数的参数。在创建 `callme()` 函数后，接着返回一个指向它的对象引用，以便可以在后续调用它。

这就意味着，现在可以这样写

```
buttonOneFunc = partial(action, "One")
buttonTwoFunc = partial(action, "Two")
```

从理想角度看，在 `functools.partial()` 可用时最好是使用 `functools.partial()`，否

则可以回溯到自己的简易 `partial()` 函数上。不错，由于可以在运行时定义这些函数，所以这样做是相当不错的。

```
import sys
if sys.version_info[:2] < (2, 5):
    def partial(func, arg):
        def callme():
            return func(arg)
        return callme
else:
    from functools import partial
```

`if` 语句可以确保在使用 Python 2.5 之前的旧版本时，能够创建一个带有函数和单参数的 `partial()` 函数，并且，`partial()` 函数还可以返回一个函数，就是在被调用时，将会调用参数中传过来的那个函数。不过，如果使用的是旧版本的 Python，可以使用 `functools.partial()` 函数，因此在我们的代码中，可以总是调用 `partial()`，至于创建哪个版本则由用到的来确定。

现在，正如之前那样，可以这样写

```
buttonOneFunc = partial(action, "One")
buttonTwoFunc = partial(action, "Two")
```

只有这一次，这段代码才可以在旧版本和新版本的 Python 上进行工作。

定义完成的 `partial()` 函数可能会是最简单的。创建更为复杂的封装程序也是有可能的，可以在对它进行封装时带有位置和关键字参数，也还可以在每一次对其调用时再带上额外的位置和关键字参数；还可以包括 `functools.partial()` 已经提供了的功能。从第二部分往后，会在不同的地方用到 `partial()`，不过，如果在 Python 2.5 或者后续版本不可用的时候，都会使用在这一节所给出的 `partial()` 的简易版本。

在下一节，将会继续看到函数的一些相对高阶的内容，还会看到一些可能对我们很有用的进行错误处理的方法。

## 2.4 异常处理

很多入门书籍通常都不会先介绍异常处理(exception handling)，经常会将其放在面向对象编程的后面。把异常处理放到控制结构这一章是因为，异常处理在面向过程和面向对象的编程中都是非常重要的，并且异常处理会造成执行流程的动态改变，这当然使得异常处理有资格成为控制结构的一种类型。

异常(exception)是一个能够在某种特定条件下“抛出”(raised)[或者“掷出”(thrown)]的对象。当一个异常被抛出时，标准执行流程就会终止，而解释器就会寻找一个能够处理该异常适当的异常处理程序，以便通过该异常。它先会查看封闭块并试着找出解决该问题的出路。如果在当前函数中没有找到合适的异常处理程序，解释器将会返回到调用栈上，在该函数的调用程序中寻找处理程序，如果在调用函数的调用中也失败了，就以此类推继续寻找。

在解释器搜索适当的异常处理程序的过程中，它可能会遇到 `finally` 块；任何这样的块被执行之后，异常处理程序的搜索都会被重置(可把 `finally` 块用于清理工作——例如，用来确保某个文件是关闭的，很快就会看到这一用法)。

如果找到了处理程序，解释器就把控制权传递给处理程序，执行流程就从那里继续执行。如果一直回溯到了所有顶级的调用栈，还是没有找到处理程序，应用程序就会终止并报告引起该异常的原因。

在 Python 中，异常可以由内置的或者库的函数和方法抛出，也可以通过我们自己的代码抛出。抛出的异常可以是内置的任何异常类型，也可以是自定义的异常类型。

异常处理程序是一些具有如下常规语法的代码块：

```
try:  
    suite1  
except exceptions:  
    suite2  
else:  
    suite3
```

在这里，如果发生异常会执行 *suite1* 中的代码，控制流就会传递给异常语句。如果异常语句正确，就会执行 *suite2*；随后很快会讨论其他情况下所发生的事情。如果没有异常出现，会在 *suite1* 执行完毕后执行 *suite3*。

异常语句不止有一种语法形式，下面是一些例子：

```
except IndexError: pass  
except ValueError, e: pass  
except (IOError, OSError), e: pass  
except: pass
```

在第一种情况下，需要处理 `IndexError` 异常但不需要给出抛出该异常的任何信息。在第二种情况下，会处理 `ValueError` 异常并希望获得该异常对象（放在变量 *e* 中）。在第三情况下，既要处理 `IOError` 异常又要处理 `OSError` 异常，但如果发生其中的任何一个异常，都希望能够得到该异常对象，也是放在变量 *e* 中。最后一种情况最好不要使用，因为它会捕获任何的异常：使用这种如此宽泛的异常处理程序通常是不明智的，因为它将捕获各种异常，包括那些我们并不期望的异常，从而会屏蔽代码中的逻辑错误。由于为各个套件都使用了 `pass`，所以如果捕捉到异常，也不会采取进一步的措施，如果有 `finally` 块的话，执行流程将从该 `finally` 块继续执行，然后再从 `try` 块之后的语句继续执行。

对于一个单独的 `try` 块来说，也可能会有多个异常处理程序

```
try:  
    process()  
except IndexError, e:  
    print "Error: %s" % e  
except LookupError, e:  
    print "Error: %s" % e
```

处理程序的顺序是很重要的。在这种情况下，`IndexError` 是 `LookupError` 的子类，因此如果先有 `LookupError`，控制流程就永远也不会传递给 `IndexError` 处理程序。这是因为，`LookupError` 可同时匹配自身及其所有的子类。就像 C++ 和 Java 一样，当为相同的 `try` 块提供多个异常处理程序时，就会按照它们出现的先后次序依次检查它们。这就意味着，需要按照它们的具体程度，按照从最具体到最不具体依次排列它们。在图 2.1 中，给出了 Python 异常的部分层次；最不具体的异常放在顶部，依次向下，到底部是最具体的异常。

既然已对异常有了初步认识，就来看看使用它们和更传统的错误处理方法之间的差异；这也会进一步加深对异常使用方法和语法的理解。将会看到的是两个含有相同行数并且做了完

全相同事情的代码片段：它们都可以从字符串中提取第一个尖括号元素。在这两种情况下，假设要搜索的字符串是放在变量文本中的。

```
# Testing for errors
result = ""
i = text.find("<")
if i > -1:
    j = text.find(">", i + 1)
    if j > -1:
        result = text[i:j + 1]
print result
```

```
# Exception handling
try:
    i = text.index("<")
    j = text.index(">", i + 1)
    result = text[i:j + 1]
except ValueError:
    result = ""
print result
```

这两种方法都可以确保，如果没有找到尖括号子串，得到的结果都会是一个空字符串。然而，右边的代码片段会默认 `try` 块中的每一行代码都是在前面一行的代码获得了正确执行的基础上——因为，如果没有正确执行，它们就会抛出一个异常，执行流程就会跳到 `except` 块。

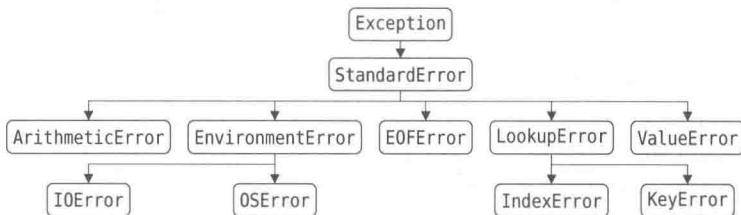


图 2.1 Python 异常的部分层次

如果要搜索的是单个字符串，使用 `find()` 将比使用异常处理机制可能会更方便；不过，如果打算处理两个或者两个以上的事情，那样做就可能会失效了，异常处理，就像这里给出的那样，通常能够在打算执行的代码和处理错误与异常的代码之间获得明显的分界，因而可以得到更简洁的代码。

在编写自己的函数时，可以让它们在出现错误时按照我们的意愿抛出异常；例如，在前一节中，可以把下面几行代码放在所开发的 `simplify()` 函数的开头：

```
def simplify(text, space=" \t\r\n\f", delete ""):
    if not space and not delete:
        raise Exception, "Nothing to skip or delete"
```

这个是可以正常运行的，但遗憾的是，对于这里所要处理的事情来说，这个 `Exception` 类（这通常是 Python 所有异常的基类）是不适用的。相反，通过创建自定义的异常并将其抛出就可以轻松解决这一问题

```
class SimplifyError(Exception): pass
def simplify(text, space=" \t\r\n\f", delete ""):
    if not space and not delete:
        raise SimplifyError, "Nothing to skip or delete"
```

异常都是类的实例，尽管在第 3 章之前都还没有讲到类，不过，创建一个异常类的语法是如此简单，以至于似乎都没有理由不在这里直接将其给出。`class` 语句具有与 `def` 语句相类似的结构，带一个 `class` 关键字，后面跟着名字，不同的是，括号里面放的是基类而不是参数名。之前曾用 `pass` 来表明这是一个空套件，也选择过对 `Exception` 的继承。不过，本来还是可以从 `Exception` 的一个子类进行继承的；例如，`ValueError`。

尽管在实际应用中，在这样的特定例子中抛出异常显得可能有些多余。也应该注意到，该函数将总是用 space 或者 delete 或者两者都非空的形式调用，因而可以对这一信念进行断言而不会使用异常：

```
def simplify(text, space=" \t\r\n\f", delete=""):
    assert space or delete
```

如果 space 和 delete 都是空的，将抛出 `AssertionError` 异常，而且可能会比之前的两种尝试方法都要更清晰地表达出该函数前置条件的逻辑性。如果没有捕捉到异常（也应该不会有断言），该程序就会终止并发出一条错误消息，称 `AssertionError` 是错误的原因并提供一个回溯标记，指出断言失败所在的文件和行。

异常处理的另一个上下文有用之处是，可以用来打破深层嵌套循环。例如，假设有一个三维网格值，希望找到某个特定目标元素第一次出现的位置。这里先给出传统的处理方法：

```
found = False
for x in range(len(grid)):
    for y in range(len(grid[x])):
        for z in range(len(grid[x][y])):
            if grid[x][y][z] == target:
                found = True
                break
        if found:
            break
    if found:
        break
if found:
    print "Found at (%d, %d, %d)" % (x, y, z)
else:
    print "Not found"
```

这段代码的长度是 15 行。它比较容易理解，但是却很难键入，而且也不怎么有效率。现在，使用一种异常处理的方法来完成

```
class FoundException(Exception):
    pass

try:
    for x in range(len(grid)):
        for y in range(len(grid[x])):
            for z in range(len(grid[x][y])):
                if grid[x][y][z] == target:
                    raise FoundException
except FoundException:
    print "Found at (%d, %d, %d)" % (x, y, z)
else:
    print "Not found"
```

这个版本的长度就只有 11 行。如果找到目标，就抛出异常并对其予以处理。如果没有抛出异常，就会执行 `try` 块的 `else` 套件。

在某些情况下，可能需要调用一些清理代码。例如，即使代码出现了错误，也希望能够确保正确关闭了文件、网络或者数据库连接。这可以通过 `try...finally` 块来实现，作为下一个例子展示如下：

```
filehandle = open(filename)
try:
```

```

for line in filehandle:
    process(line)
finally:
    filehandle.close()

```

在这里，用给定名的文件名打开了一个文件并获得了该文件的句柄。然后，对该文件句柄进行了遍历——这是一个生成器，在上下文的 `for` 循环中一次遍历一行。如果有任何异常发生，该解释器都会找出作用域中最近的 `except` 或者 `finally`。在本例这种情况下，它不会找到 `except`，但它确实可以找到一个 `finally`，因此，解释器就会将控制流程转到 `finally` 块并执行其中的语句。如果没有出现异常，将会在 `try` 套件执行完成后执行 `finally` 块。因此，不管是哪一种方式，文件都可以被关闭。

Python 2.5 之前的版本不支持 `try...except...finally` 块。因此，如果既需要 `except` 又需要 `finally`，就必须要使用两个块，一个是 `try...except` 块，另一个是 `try...finally` 块，用一个嵌套到另一个里面。例如，直到 2.4 版的 Python，打开和处理文件时最可靠的方法都是这样的

```

fh = None
try:
    try:
        fh = open(fname)
        process(fh)
    except IOError, e:
        print "I/O error: %s" % e
    finally:
        if fh:
            fh.close()

```

这段代码使用了之前所讨论过的东西，但是，为了能够真正掌握异常处理机制，这里还是打算认真仔细地分析一下这些代码。

如果文件在第一处没有打开，就会执行 `except` 块，然后执行 `finally` 块——这什么也不会做，因为文件没有打开，故而文件的句柄仍然是 `None`。另一方面，如果打开了该文件，而且处理也开始了，但还是可能会出现 I/O 错误。如果发生这种情况，`except` 块就会执行，然后控制流程还是会传递给 `finally` 块，而文件还是会被关闭的。

如果发生的异常不是 `IOError`，或者也不是 `IOError` 的子类，例如，可能会是在 `process()` 函数中出现的 `ValueError`——解释器将会认为 `except` 块不合适，就会寻找最近的、合适的封闭异常处理程序。看起来，解释器将会先遇到 `finally` 块并执行，之后（也就是，关闭文件之后），会接着寻找适当的异常处理程序。

如果文件打开且直到处理完成也没有抛出异常，那么将跳过 `except` 块，但仍会执行 `finally` 块，因为无论发生什么都会执行 `finally` 块。因此，在所有情况下，除了用户直接终止解释器（或者，在非常罕见的情况下，程序崩溃），如果文件打开了，它都会被关闭。

在 Python 2.5 及之后的版本中，可以使用另一种具有相同语义但却更简单的方法，因为可以使用 `try...except...finally` 块了：

```

fh = None
try:
    fh = open(fname)
    process(fh)
except IOError, e:
    print "I/O error: %s" % e

```

```
finally:  
    if fh:  
        fh.close()
```

使用这种语法，仍有可能需要一个 `else` 块以用于没有发生异常的情况；`else` 块会被放置在最后的 `except` 块之后和那个唯一的 `finally` 块之前。在第 6 章讨论文件的话题中，还会再次回顾这一点。

不管用的是什么版本的 Python，也不管异常是否会发生，总是会执行 `finally` 块，确切地说，是执行一次，要么是在 `try` 套件结束之后，要么是在把控制流程切换出 `try` 块抛出异常的时候。

Python 2.6(以及 Python 2.5，会用 `from __future__ import with_statement` 语句)完整提供了另外一种方法：“上下文管理器”(`context manager`)。对于文件的处理，我们更喜欢 `try...finally` 方法，但是在其他情况下，我们更喜欢上下文管理。例如，在第 19 章，会用上下文管理器来说明如何通过线程来锁定和解锁 `read/write` 块。

## 小结

在这一章，看到了如何使用 `if` 进行分支，也看到了如何使用带有 `elif` 或者可选的 `else` 的 `if` 创建多重分支的方法。还看到了如何使用 `while` 和 `for` 进行循环，以及如何使用 `range()` 生成整数列表的方法。学习了一些字典的方法，可以提供字典的键、值以及键值对(元素)，还简要地看了排序。也粗略看到了该如何使用 Python 的列表解析和生成器。

看到了如何使用 `def`(以及用 `lambda`)创建函数。使用位置参数和关键字参数，开发了两个有用的函数，`frange()` 和 `simplify()`。看到了 Python 是如何动态地创建函数，用来读取.py 文件，并看到了如何使用这一动态机制在旧版本的 Python 中提供与新版本 Python 中相类似的功能。此外，还看到了如何使用偏函数应用程序来创建封装器函数，用它来封装函数和函数的参数(闭包)。

还学会了如何抛出异常，如何创建自定义异常以及如何处理异常。明白了如何使用 `finally` 来保证善后工作，还讨论了 Python 可能提供的一些更为复杂的异常处理方法。也看到当在那些可能发生异常的多个地方使用套件，异常处理可以使代码结构更加清晰，以及如何用它们来干净地退出那些深度嵌套的循环集。

创建自定义异常使得我们能够创建出更为简单的类；这些类可以没有属性(没有成员数据)和方法。在接下来的一章中，将会更正式地审视类，并学习如何用任何期望的属性和方法来创建和实例化这些类。

## 练习题

在第 1 章，练习题都很简短以便可以键入 IDLE 中。从现在开始，建议将自己的答案键入到.py 后缀的文件中，并在文件的最后加上一些调用语句。例如，或许可以把文件的结构写成如下类似的形式：

```
#!/usr/bin/env python
def mysolution(arg0, arg1):
    pass      # Whatever code is needed
mysolution(1, 2)      # Call with one set of parameters
mysolution("a", "b")  # Call with another set of parameters
# Additional calls to make sure all boundary cases are tested
```

如果使用的是 Windows，确保是在控制台窗口中运行自己的程序；相应地，Mac OS X 的用户应当使用 Terminal。或许还需要包含有 print 语句，以便可以看到结果（在第二部分的练习题中将会包含 GUI 应用程序）。

如果看过本书的源代码，包括这一章的 answers.py 文件，将发现，这些代码往往都有很长的文档字符串，在很多出现的地方甚至比代码自身都要长。这是因为，文档字符串通常包含了示例的用法说明，其中会身兼测试等多重职责，这可以在第 3 章的“使用 doctest 模块”一节中看到相关说明。

### 1. 编写一个签名函数：

```
valid(text, chars="ABCDEFGHIJKLMNPQRSTUVWXYZ0123456789")
```

这个函数应当返回一个（或者是空的）字符串，它是只含有 char 字符的 text 的副本。例如

```
valid("Barking!")           # Returns "B"
valid("KL754", "0123456789") # Returns "754"
valid("BEAN", "abcdefghijklmnopqrstuvwxyz") # Returns ""
```

该函数应该在 6 行左右完成，可以使用一个 for 循环和一个 if 语句，不必考虑文档字符串，当然也应当写出来。

### 2. 编写一个签名函数

```
charcount(text)
```

这个函数应当返回一个带有 28 键的字典，28 键是“a”，“b”，…，“z”，外加“whitespace”和“others”。对于 text 中的每一个小写字符，如果字符是字母，增加其对应的键的数量；如果该字符是空格，增加“whitespace”键的数量；否则，减少“others”键的数量。例如，如下调用：

```
stats = charcount("Exceedingly Edible")
```

将意味着，stats 是一个带有如下内容的字典：

```
{'whitespace': 1, 'others': 0, 'a': 0, 'c': 1, 'b': 1, 'e': 5,
'd': 2, 'g': 1, 'f': 0, 'i': 2, 'h': 0, 'k': 0, 'j': 0, 'm': 0,
'l': 2, 'o': 0, 'n': 1, 'q': 0, 'p': 0, 's': 0, 'r': 0, 'u': 0,
't': 0, 'w': 0, 'v': 0, 'y': 1, 'x': 1, 'z': 0}
```

使用一个字典和一个 for 循环，应该可以用 12 行左右的代码完成。

### 3. 编写一个签名函数

```
integer(number)
```

参数 number 要么是一个数字，要么是一个可以转换成数字的字符串。本函数应当返回 int 类型的数字，如果传入的数字是 float 应当进行圆整。如果转换失败，要捕获 ValueError 异常并返回 0。确保本函数既可以用于字符串也可以用于纯数字，比如 4.5、32、“23”和“-15.1”，而对于无效数字要能够正确返回 0。本函数应该可以用 5 ~

6行代码完成(提示：要使得能够用于所有情况，就需要总是先转换成 float 类型，也就是，通过对输入调用 float() 来完成)。

#### 4. 现在，编写一个签名函数

```
incrementString(text="AAAA")
```

本函数必须对给定的字符串进行“增量”操作。这里给出一些示例

```
incrementString("A")      # Returns "B"  
incrementString("Z")      # Returns "AA"  
incrementString("AM")     # Returns "AN"  
incrementString("AZ")     # Returns "BA"  
incrementString("BA")     # Returns "BB"  
incrementString("BZ")     # Returns "CA"  
incrementString("ZZA")    # Returns "ZZB"  
incrementString("ZZZ")    # Returns "AAAA"  
incrementString("AAA")    # Returns "AAAB"  
incrementString("AAAZ")   # Returns "AABA"  
incrementString("ABC2")   # Raises a ValueError
```

在 text 中的字符必须是 A ~ Z(或者 a ~ z，此时函数要能够将其大写)；否则，本函数应当抛出 ValueError 异常。

这是一个与之前练习题相比稍具挑战性的题目。尽管本题也可以不使用列表解析的方法完成，但如果使用列表解析的话，应该可以在 20 行以内完成。要想获得正确的结果，还是要有点技巧的(提示：reversed() 函数可以返回一个序列的逆序)。

#### 5. 如果阅读了函数生成器一节的内容，试着编写一个签名函数

```
leapyears(yearlist)
```

参数 yearlist 是一个年数的序列——例如，[1600, 1604, 1700, 1704, 1800, 1900, 1996, 2000, 2004]。以此为输入，其输出应当每次一个，分别为 1600、1604、1704、1996、2000 和 2004。该函数应该用 6 行左右的代码即可完成(提示：闰年是可以被 4 整除的年份，但如果能够被 100 整除，则必须也能够被 400 整除)。

本练习题的参考答案在文件 chap02/answers.py 中。

## 第3章 类和模块

- 实例的创建
- 方法和特殊方法
- 继承和多态
- 模块和多文件应用程序

Python 完全支持过程化编程和面向对象编程，这就为编程时选择其一留足了自由，或者将两者结合起来。到目前为止，完成的都是过程化编程，尽管已经使用过一些 Python 类——例如 `str` 字符串类。但至今还没有定义过自己的类。在这一章，将会了解到该如何创建类和方法，以及如何用 Python 进行面向对象编程。在所有的后续章节中，将都会采用面向对象的方法来编写程序。

假设熟悉面向对象编程——例如使用 C++ 或者 Java——不过，还是要借这个机会澄清一下书中的部分概念术语。书中使用的“对象”(object)一词，以及偶尔用到的“实例”(instance)一词，都可用来指向某一特定的类的实例。对于“类”(class)、“类型”(type)和“数据类型”(data type)三词不做严格区分。属于某一特定实例的变量称为“属性”(attribute)或者“实例变量”(instance variable)。用在方法内部但又不是实例变量的变量称为“本地变量”(local variable)，或者简单叫做“变量”(variable)。用“基类”(base class)这个词表示一个被继承的类；基类既可能是直接先祖，也可能是对继承树的进一步继承。一些人会使用“超类”(super class)这个词来表示这一概念。本书还会使用“子类”(subclass)和“派生类”(derived class)来表示从另一个类中继承而来的类。

在 Python 中，任何方法在其子类中都可以覆盖(override，重新实现(reimplement))；这与 Java 相同(除了 Java 的“final”方法以外)<sup>①</sup>。重载(overload)，换言之，在同一个类中的那些名字相同但参数列表不同的方法，却并不支持，尽管这在实际应用中并没有多大局限性，因为 Python 具有全面的参数处理能力。事实上，潜在的 Qt C++ API 大大扩展了重载的使用范围，但 PyQt 对其的处理是悄无声息的，因此，在实践中，可以调用那些任何“重载”过了的 Qt 方法并依靠 PyQt 做出正确的事情<sup>②</sup>。

本章将以创建类的基本语法开始。然后，研究如何构造和初始化对象，以及如何实现(implement)方法。Python 对于面向对象编程支持最好的特性之一就是，它允许重新实现那些“特殊方法”(special method)。这就意味着，可以让我们自己的类无缝嵌入，从而使这些类具有与内置类一样的行为。例如，可以轻松让我们的类与诸如 `==` 和 `<` 这样的比较运算符一起工作。然后，再来看看数字方面的那些特殊方法：利用这些方法，在创建完整的自定义数据类型时，特别是那些数字性的，可以重载那些诸如 `+` 和 `+=` 这样的有用运算符。如果我们的类是集合，

<sup>①</sup> 用 C++ 术语来说，所有的 Python 方法都是 `virtual` 的。

<sup>②</sup> 在 C++ 程序中，重载(overload)、覆盖(override)和重写(overwrite)的含义并不完全相同，具体来说：重载是将语义、功能相似的几个函数用同一个名字表示，但参数或返回值不同(包括类型、顺序等)；覆盖是指派生类的函数覆盖基类的函数；重写是指派生类的函数屏蔽了与其同名的基类函数——译者注。

还会有一些可以重新实现的额外特殊方法，例如，我们的集合还可以支持 `in` 运算符和 `len()` 函数。本章最后一节，会用 Python 对继承和多态的支持来作为结束。

由于历史原因，Python 提供了两种用户自定义类型（类）：“老式”（old-style）和“新式”（new-style）。唯一显著的不同是，老式类要么没有基类，要么只有旧式的基类。新式类则总是派生自一个新式类，例如，`object`，它是 Python 中最基本的基类。由于再无使用老式类的理由，并且在 Python 3.0 中将会彻底抛弃这些类，所以本书将只会使用新式类。

类的创建的语法很简单

```
class className(base_classes):  
    suite
```

在该类的 `suite` 中，可以使用 `def` 语句；在这一上下文语境中，创建的方法将是封闭类的方法而不再是函数。

也有可能存在“空”类，即它们自己没有方法或者属性（数据成员），正如在上一章的末尾，在派生自定义异常类时所看到的那样。

新式类总是至少会有一个基类——例如，`object`。与 Java 不同，Python 支持多重继承，也就是说，Python 类可以从一个、两个或者多个基类中进行继承。通常要尽量避免应用这种情况，因为这样做将会导致一些不必要甚至易使人混淆的复杂问题。Python 并不支持抽象类（abstract class，是一些不能实例化的类，并且只能从这些类中派生新类——对定义接口很有用），但仍然可以获得抽象类所具有的各类效果。这里将会看到一个有关多重继承的小例子，其中的一个基类就是“抽象”类，只将其用来提供 API（与 Java 的接口很像）。

在 Python 中，可以从类的内部和外部访问所有的方法和属性；并没有诸如“public”和“private”这样的特殊指示符。Python 确实还是有“private”概念的——那些名字以单一下划线开头的对象都可以认为是私有的。就方法和实例变量而言，它们的私有性仅仅是一个希望大家能够遵守的约定而已。而且对模块来说，所谓的私有类和私有函数，就是那些名字以一个下划线开头，在使用时以 `from moduleName import *` 语法无法导入的类和函数。Python 还有一个“隐私”（very private）的概念——那些名字以两个下划线开头的方法和属性。隐私对象仍是可以访问的，不过 Python 解释器会改编（mangle）它们的名字，以便可以避免不小心访问到它们。

现在，对创建类的基本语法已经有了基本了解，并且对 Python 面向对象的特性也有了广泛认识，是时候准备着手创建一个类和一些实例了。

### 3.1 实例的创建

在大多数面向对象语言中，对象的创建有两个步骤：首先，构造对象，其次，初始化对象。一些语言会将这两个步骤合二为一，但 Python 还是会让它们分开的。Python 有一个 `__new__()` 特殊方法，可以用来构造对象，还有一个 `__init__()` 特殊方法，可以用来初始化一个新构造的对象。其实，很少需要我们自己动手实现 `__new__()`；在本书的任何一个自定义类中都没有用到它——老版本的 Python 甚至都没有 `__new__()` 这个特殊方法。Python 完全有能力为我们构建对象，所以几乎在每种情况下，唯一需要我们实现的方法就是 `__init__()`。

考虑到 Python 的两步对象创建方法，通常谈论的会是对象的创建而不是构造。同时，通常会引用类的初始化程序（initializer，它的 `__init__()` 方法），因为这就是在自定义类中通常需要重新实现的方法，也是更接近于 C++ 和 Java 等语言中构造函数思想的一个。

下面来看看实践中是如何创建一个类的。这里将要创建一个保存字符串(一种椅子的名字)和数字的类(这种椅子会有多少条腿):

```
class Chair(object):
    """This class represents chairs."""

    def __init__(self, name, legs=4):
        self.name = name
        self.legs = legs
```

正如之前的代码所示,通常在 `class` 语句之后会跟着一个文档字符串(`docstring`)。本书通常不会显示这些文档字符串,但会在随书使用的示例代码中的适当地方包含这些文档字符串。空行纯粹是为了美观和清晰。

方法名是以两个下画线开始和两个下画线结束的就是一些“特殊”方法。Python 使用这些方法来集成自定义类,以便使得它们可以像内置类一样具有相同的使用模式,这一用法很快就会看到。

`__init__()` 方法,实际上每一个方法都是,带有的第一参数相当于 C++ 或者 Java 的“this”变量,也就是说,是一个指向对象自身的变量。这个变量通常称为 `self`。必须把 `self` 放在每个(非 `static`<sup>①</sup>)方法参数列表中的第一个位置,尽管我们从不需要自己传递这一参数,因为 Python 会自动替我们完成这件事。

尽管名称“`self`”仅仅是规定上的,但我们会一直使用它。在该对象内部,在需要引用实例的方法和属性时,就必须要显式地使用 `self`。例如,在 `Chair` 类的初始化程序中,就曾使用 `self` 创建过两个数据属性。归功于 Python 的动态特性,从而使得在其他方法中创建额外的属性成为可能,如果需要,甚至还可以把额外的属性添加到特定的实例中;不过,可以保守地认为,对于我们所为之努力的 GUI 编程来说,这些足够了。

要创建类的实例,可以使用如下语法:

```
instance = className(arguments)
```

即使不需要传递任何参数,这里的括号也是必需的。这是因为,Python 是通过调用该类的静态 `__new__()` 方法(从 `object` 中继承而来,或者在极少的情况下,是由我们自己实现的)来构造该对象的,然后会对新构造的对象调用 `__init__()`。结果是返回初始化后的对象。

在上述的 `Chair` 类例子中,必须传递一个或者两个参数(Python 会自动传递第一个 `self` 参数);例如

```
chair1 = Chair("Barcelona")
chair2 = Chair("Bar Stool", 1)
```

由于这些属性是 `public`,可以通过点号(.) 运算符进行读取和赋值;例如: `print chair2.name` 将会输出“Bar Stool”,而 `chair1.legs = 2` 将会把 `chair1` 的 `leg` 属性的值从 4 修改为 2。

面向对象编程的卫道士们毫无疑问会对这种从实例外部直接访问属性的方法感到不适应,然而,对那些仅是为了尝试编程的人来说这种形式却相当完美,因为总是可以在后续加入新的访问方式或者新的方法。

现在已经了解了该如何处理构造和初始化过程,还需要了解对象的析构(destruction)。C++ 程序开发人员习惯使用析构函数并依赖于它们可以一次删除所选中的对象。Java 和 Py-

<sup>①</sup> static 静态方法就是一个可在类或者实例上调用且没有 `self` 参数的方法。常规方法都是非静态的,也就是说,它们都有一个 `self` 参数且都必须由实例调用。

thon 的程序开发人员则没有这种特殊的嗜好。相反，它们都有自动进行垃圾收集机制，从总体上看，这使得编程更为容易，但唯一的缺点是在删除对象时不能进行精确的控制。如果资源需要保护，通常的解决方案将是使用 `try...finally` 块来保证善后工作。当一个对象要作为垃圾收集时，将会调用其特殊的 `__del__()` 方法，而 `self` 则是其唯一的参数。在 Python 中（在 Java 中对应的是 `finalize()` 方法）这是惯常做法，但我们自己却很少会使用这一特殊方法。单以本书为例，书中例子和练习题答案中的类超过 170 个，但没有哪一个是对 `__del__()` 进行过重新实现。

现在已经知道如何创建和初始化一个自定义类的对象。接下来，将要看到如何给我们的类提供一些额外的方法而使其具有与众不同的行为。还将学习如何确保我们的类可以和 Python 的其他类较好地集成到一起，使得它们能够像内置类一样在适当的地方起作用。

## 3.2 方法和特殊方法

先从学习一个类开始，该类使用存取器方法(accessor method)而不是使用直接属性访问的形式对属性值进行获取和设置<sup>①</sup>。

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def getWidth(self):
        return self.width
    def setWidth(self, width):
        self.width = width
    def getHeight(self):
        return self.height
    def setHeight(self, height):
        self.height = height
    def area(self):
        return self.getWidth() * self.getHeight()
```

这里对于各个存(getter)和取(setter)函数使用了类 Java 式的命名规范。现在，可以像这样来编写代码

```
rect = Rectangle(50, 10)
print rect.area()      # Prints "500"
rect.setWidth(20)
```

之前，本来也可以像下面那样轻松实现 `area()` 方法：

```
def area(self):
    return self.width * self.height
```

正如在这里所做的那样，编写许多烦琐而简单的存取器方法对于 C++ 等语言来说是不错的方法，因为这样做可以提供最大限度的灵活性，并且编译后的代码也不会有过多开销。而且，如果稍后需要在存取器中执行一些计算，可以在函数中简单添加这一功能而不需要用户对他们的类的代码做任何改变。不过，在 Python 中，没必要编写这样的存取器。相反，可以直接读写

<sup>①</sup> 可以简单地认为，存取器方法中的“存”就是指赋值函数 `setter`，“取”就是指返回值函数 `getter`——译者注。

各个属性，而如果在稍后的阶段需要执行一些计算，可以使用 Python 的 `property()` 函数。这个函数允许我们创建一些命名特性(property)来替代那些属性。特性的访问和属性一样，但实际情况是，它们会调用一些指定用来获取和设置特性值的方法。

下面是 `Rectangle` 类的第二个版本，这一次，对于宽度和高度属性会采用直接属性访问的方法，还有一个用于面积的特性：

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def _area(self):
        return self.width * self.height
    area = property(fget=_area)
```

这样就可以写出像下面的代码：

```
rect = Rectangle(5, 4)
print rect.width, rect.height, rect.area    # Prints (5, 4, 20)
rect.width = 6
```

Python 的 `property()` 函数可用来给定一个存方法、一个取方法、一个删除方法和一个文档字符串。由于只给定了一个存方法，面积特性就成为只读。如果后续在访问 `width` 时需要执行一些计算，可简单地将其转化为特性即可，就像下面这样：

```
def _width(self):
    return self._width
def _setWidth(self, width):
    # Perform some computation
    self._width = width
width = property(fget=_width, fset=_setWidth)
```

值得注意的是，这里已经将实例变量的名字由 `width` 改为了 `_width`，从而避免了与 `width` 特性名字的冲突。一般情况下，把值保存在实例变量中的那些特性都会使用私有名字(名字开头会有两个下画线)来表示实例变量，这样做可以避免与类所使用的特性名字发生冲突。例如，带有 `width` 特性的 `Rectangle` 类的用户可以完全像之前那样获取和设定 `width` 属性，只是到了现在，实际会使用 `_width()` 和 `_setWidth()` 方法来执行这些操作，并且属性的数据会保存在 `_width` 实例变量中。

Python 实际上还提供了远比这里看到的多得多的属性访问控制方法，不过由于这与我们的 GUI 编程目标并没有多大必要，所以将会离开这一主题，如果什么时候有兴趣了回过头来查阅一下即可。这部分内容可以查看 Python 文档中的 `__getattr__()`、`__getattribute__()` 以及 `__setattr__()` 等特殊方法。

Python 方法的机理，包括特殊方法，与函数的机理是完全一样的，不过会外加第一参数 `self`，以及对 `self` 的属性进行访问的能力和调用 `self` 的方法。只需要时刻牢记的是，当我们调用方法或者访问实例变量时，就必须说明让实例使用 `self`。例如，在 `Rectangle` 类的 `setHeight()` 方法中，就是使用 `self.height` 引用实例变量并让 `height` 指向参数的，也就是说，指向一个本地变量。与之相似的是，在 `area()` 方法中，我们调用两个 `Rectangle` 方法时会再次用到 `self`。这一点与 C++ 和 Java 相当不同，C++ 和 Java 是假定已经有了实例的。

在C++中运算符的实现是可能的，也就是说，可能为我们的数据类型提供自定义的运算符实现。C++语法中会使用operator关键字后跟上运算符自身即可——例如，operator +()——不过，在Python中，每个运算符都有名字，所以要在Python中实现一个类的+运算符就应当实现一个方法。实现运算符的所有Python方法都是些特殊方法，这可以通过它们都拥有前后都带两个下画线的名字看出来。

为了更好地将自定义类集成到Python中，或许还有必要重新实现一些额外常用的特殊方法。例如，或许打算支持比较运算符，就是一个用于类的实例的布尔值。实际应用中，就需要再向Rectangle类添加一些方法来说明这是可能的，不过，考虑到简明扼要，这里不会再次重复那些已经实现过的类的声明和方法。比较的开头是这样的

```
def __cmp__(self, other):
    return cmp(self.area(), other.area())
```

假定打算可以自行实现每一个比较运算符的每个特殊方法。例如，如果把“小于”(less than)实现为\_\_lt\_\_()，就可以用<运算符对我们自己的类进行比较。然而，如果不打算单独实现每一个比较运算符，可以像在这里所做的一样简单实现\_\_cmp\_\_()即可。如果这个特殊方法实现了，Python在比较时将会使用这一特定方法，否则就会求助于\_\_cmp\_\_()。所以，只需要实现这一特殊方法，所有的运算符(<, <=, ==, !=, >=, >)将都可用于Rectangle对象之间的比较：

```
rectA = Rectangle(4, 4)
rectB = Rectangle(8, 2)
rectA == rectB      # True because both have the same area
rectA < rectB      # False
```

这里使用内置的cmp()函数来实现了\_\_cmp\_\_(). cmp()函数带有两个对象，如果第一个对象小于第二个对象可以返回-1，如果相等返回0，否则返回1。会使用矩形的面积作为两个对象比较的基础，这就是为什么在例子得到了一个令人吃惊的结果。当然，更严格，或者说是更好的实现方法可能是这样的

```
def __cmp__(self, other):
    if (self.width != other.width):
        return cmp(self.width, other.width)
    return cmp(self.height, other.height)
```

在这里，如果宽度值相等，会返回高度值的比较结果；否则，就返回宽度值比较的结果。

如果任何特殊方法都没有实现，绝大多数情况下，Python也会乐于执行比较，尽管可能不会是我们想要的方式。如果创建的类中的比较可以起作用，就应当实现\_\_cmp\_\_(). 对于其他一些类，要做的最安全的事就是为其\_\_cmp\_\_()实现一个带有NotImplementedError的函数体。

```
def __nonzero__(self):
    return self.width or self.height
```

当对象处于布尔型上下语境时就会调用这个特殊方法；例如，bool(rectA)或者if rectB:，又或者如果对象“非空”就返回True，等等。

```
def __repr__(self):
    return "Rectangle(%d, %d)" % (self.width, self.height)
```

特殊方法的“表示”就是必须返回一个字符串，如果进行求值的话（比如，使用eval()），这个字符串会导致用同样的特性构造对象，就像用这个字符串对该对象进行调用一样。一些对象

要支持这一点会太过复杂，而对另一些对象，比如 GUI 中的一个窗口或者一个按钮，则不算什么；因此，那样的类就不会对 `__repr__()` 提供重新实现。在一个字符串中，% 运算符可以格式化字符串，可使用 %r 获取这一特殊方法结果；也可以使用 `repr()` 函数或者反撇号运算符（` `）。反撇号 ` ` 只是 `repr()` 用法的语法同义词。例如，`repr(x)` 和 `x` 都可以返回同样的结果：由 `x` 的 `__repr__()` 方法所返回的对象 `x` 的表示。

还有一个 `__str__()` 特殊方法，必须返回一个它所应用到的对象的字符串表示（类似于 Java 的 `toString()` 方法），但不像 `__repr__()`，这个表示意味着是人工可以读取的，而不是只能够可供 `eval()` 的。如果，就像这个例子中一样，没有实现 `__str__()` 方法，Python 将会用 `__repr__()` 方法进行代替。例如：

```
rect = Rectangle(8, 9)
print rect    # Prints "Rectangle(8, 9)" using __repr__()
```

如果我们的类需要的是一个人工可读的 Unicode 字符串表示，可以实现 `__unicode__()`。

本来还可以实现一些更多的普通特殊方法，不过它们并不适用于 `Rectangle` 类。在表 3.1 中，给出了一些常用的可被实现的普通特殊方法。

表 3.1 一些基本的特殊方法

方法	语法	说明
<code>__init__(self, args)</code>	<code>x = X()</code>	实现一个新近创建的实例
<code>__call__(self, args)</code>	<code>x()</code>	让实例可调用，也就是说，将它们转换成仿函数（functor）。args 可有可无（高级）
<code>__cmp__(self, other)</code>	<code>x == y</code> <code>x &lt; y</code> # etc	如果 <code>self &lt; other</code> 返回 -1，如果两者相等返回 0，否则返回 1。如果实现了 <code>__cmp__()</code> ，就用其作为那些没有明确实现的任意比较运算符
<code>__eq__(self, other)</code>	<code>x == y</code>	如果 <code>x</code> 等于 <code>y</code> 返回 True
<code>__ne__(self, other)</code>	<code>x != y</code>	如果 <code>x</code> 不等于 <code>y</code> 返回 True
<code>__le__(self, other)</code>	<code>x &lt;= y</code>	如果 <code>x</code> 小于或者等于 <code>y</code> 返回 True
<code>__lt__(self, other)</code>	<code>x &lt; y</code>	如果 <code>x</code> 小于 <code>y</code> 返回 True
<code>__ge__(self, other)</code>	<code>x &gt;= y</code>	如果 <code>x</code> 大于或者等于 <code>y</code> 返回 True
<code>__gt__(self, other)</code>	<code>x &gt; y</code>	如果 <code>x</code> 大于 <code>y</code> 返回 True
<code>__nonzero__(self)</code>	<code>if x: pass</code>	如果 <code>x</code> 非零返回 True
<code>__repr__(self)</code>	<code>y = eval(`x`)</code>	返回 <code>x</code> 的一个可 <code>eval()</code> 的表示。使用反撇号与调用 <code>repr()</code> 是一样的 <sup>①</sup>
<code>__str__(self)</code>	<code>print x</code>	返回 <code>x</code> 的一个可供人工阅读的表示
<code>__unicode__(self)</code>	<code>print x</code>	返回 <code>x</code> 的一个可供人工阅读的 Unicode 表示

此时此刻，C++ 程序开发人员或许会困惑，复制构造函数（copy constructor）和赋值运算符跑哪里去了，而 Java 程序开发人员或许会困惑，`clone()` 方法呢？Python 没有使用复制构造函数，重新实现赋值运算符也没有必要。如果确实希望有一个像使用“=”一样的赋值运算符，Python 将会给已存在的对象绑定一个名称。如果确实需要对象的一个副本，可以使用 `copy` 模块中的 `copy()` 或者 `deepcopy()` 函数，用于对象的第一个函数没有嵌套属性或者足以应付浅复制（shallow copy）的情况，而第二个函数则必须是全面复制。作为选择，可以提供自己的复制方法，通常会调用 `copy()`，因为这是 Python 中的常规做法。

① 最好使用 `repr()` 而不是使用反撇号，因为在 Python 3.0 中会抛弃不用反撇号。

对于数字型类，提供支持诸如`+`和`+=`这样的标准数字运算符功能通常比较容易。而在Python中，可通过实现不同的特殊方法，就很容易做到。如果只实现了“`+`”，Python将会用它提供“`+=`”的功能，但最好还是能够将两者都予以实现，这是因为，实现两者能够让我们获得更好的控制力，且可以更轻松地实现操作的优化。

在表3.2中给出了数字型特殊方法最常见的实现方法。但包括移位运算符和十六进制与八进制转换运算符等在内一些运算符并没有列示在内。

表3.2 部分数字型特殊方法

方法	语法	方法	语法
<code>_float_(self)</code>	<code>float(x)</code>	<code>_int_(self)</code>	<code>int(x)</code>
<code>_abs_(self)</code>	<code>abs(x)</code>	<code>_neg_(self)</code>	<code>-x</code>
<code>_add_(self, other)</code>	<code>x + y</code>	<code>_sub_(self, other)</code>	<code>x - y</code>
<code>_iadd_(self, other)</code>	<code>x += y</code>	<code>_isub_(self, other)</code>	<code>x -= y</code>
<code>_radd_(self, other)</code>	<code>y + x</code>	<code>_rsub_(self, other)</code>	<code>y - x</code>
<code>_mul_(self, other)</code>	<code>x * y</code>	<code>_mod_(self, other)</code>	<code>x % y</code>
<code>_imul_(self, other)</code>	<code>x *= y</code>	<code>_imod_(self, other)</code>	<code>x %= y</code>
<code>_rmul_(self, other)</code>	<code>y * x</code>	<code>_rmod_(self, other)</code>	<code>y %= x</code>
<code>_floordiv_(self, other)</code>	<code>x // y</code>	<code>_truediv_(self, other)</code>	<code>x / y</code>
<code>_ifloordiv_(self, other)</code>	<code>x // = y</code>	<code>_itruediv_(self, other)</code>	<code>x /= y</code>
<code>_rfloordiv_(self, other)</code>	<code>y // x</code>	<code>_rtruediv_(self, other)</code>	<code>y / x</code>

有两个不同的除法运算符的原因是，Python可以要么执行整数除法，要么执行浮点数除法。

某些特殊方法会有两个或者三个版本；例如，`_add_()`、`_radd_()`和`_iadd_()`。带“r”的版本（比如，`_radd_()`），可用于左侧操作数没有合适方法而右侧操作数却有的情形。例如，如果有一个`x + y`的表达式，`x`和`y`的类型分别是`X`和`Y`，Python将先试着通过调用`X._add_(x, y)`来对表达式进行求值。而如果类型`X`没有这个方法，Python将转而试用类型`Y._radd_(x, y)`来求值。如果`Y`也没有那种方法，就会抛出异常。

在“i”的版本中，“i”代表“原地更新”（in-place）。可用于增量赋值运算符中，比如`+=`。很快就会看到一个这些方法在实际应用中的例子，不过，必须应先学会如何创建静态数据和静态方法。

### 3.2.1 静态数据、静态方法和装饰器

在某些情况下，拥有一些能够和类而不是类的实例一起使用的数据会非常有用。例如，如果有一个`Balloon`类，或许就想知道，已经采用了多少种个性化色彩

```
class Balloon(object):
    unique_colors = set()

    def __init__(self, color):
        self.color = color
        Balloon.unique_colors.add(color)

    @staticmethod
    def uniqueColorCount():
        return len(Balloon.unique_colors)

    @staticmethod
    def uniqueColors():
        return Balloon.unique_colors.copy()
```

静态数据是在 class 块内创建的，但却在任何 def 语句之外。为了访问静态数据，必须给定其名字，最简单的方法是使用类名，就像在 Balloon 类的静态方法中所做的那样。在下一个节，将会看到更接近真实上下文语境的静态数据和方法。

@staticmethod 就是一个装饰器。所谓的装饰器(decorator)就是一个函数，它带一个作为参数的函数，用一定的方式封装参数函数，然后再将封装后的函数赋值回初始函数的名字，因此，它具有如下代码所示的同样效果：

```
def uniqueColors():
    return Balloon.unique_colors.copy()
uniqueColors = staticmethod(uniqueColors)
```

符号 @ 用于表明这是一个装饰器。staticmethod() 函数是一个 Python 的内置函数。

可以使用多个装饰器。例如，一个合适的装饰器应当可以写成辅助性的函数或者方法，或者用做记录每次调用的方法。例如

```
@logger
@recalculate
def changeWidth(self, width):
    self.width = width
```

在这里，对象的 width 无论何时发生改变，都会应用到两个装饰器上：logger()，或许会在日志文件或者数据中记录这次改变；recalculate()，或许会更新对象的面积。

除静态方法外，Python 还支持“类方法”(class method)。这与静态方法有些类似，它们都没有第一参数 self，因而可以使用类或者实例对其进行调用。类方法与静态方法的区别之处在于，类方法对它们所调用的类会由 Python 提供第一参数。这通常称为 cls。

### 3.2.2 例：Length 类

现在已经看到过许多 Python 的普通和数字型特殊方法，应该说已经具备了创建一个完整自定义数据类型的能力。将会创建一个 Length 类，用它来保存物理长度。我们希望能够想这样来创建这些长度值：distance = Length("22 miles")。同时，也希望能够以自己喜欢的单位来检索长度值——例如，km = distance.to("km")。这个类一定不能支持长度乘以长度的乘法(因为那样将会生成面积)，不过应当支持数量乘法；例如，distance \* 2。

像往常一样，尽管在 chap03/length.py 源代码中有文档字符串的内容，但还是不会在下面的代码片段中给出它们，一则节省空间，二则可以避免从代码自身上分神。

```
from __future__ import division
```

文件中的第一句就相当有趣。from \_\_future\_\_ import 语句用来打开那些在后续 Python 版本中默认打开的行为。那个语句必须总是放在最前面。在这个例子中，可以说，希望它能够打开 Python 的未来除法行为，即对于除法“/”既要能够形成“正确”的结果，又要能够形成正确的浮点数除法，而不是只像常规的除法那样，也就是说，要能够形成截断除法(truncating division)<sup>①</sup>(“//”运算符可以实现截断除法，如果导入的话，也就的确是我们需要的)。

<sup>①</sup> 这段话的意思，是否打开 Python 精确除法(/)的未来行为：当在程序中没有导入该行为时，“/”运算符执行的是截断除法；当在程序中导入精确除法之后，“/”执行的是精确除法——译者注。

```

class Length(object):
    convert = dict(mi=621.371e-6, miles=621.371e-6, mile=621.371e-6,
                   yd=1.094, yards=1.094, yard=1.094,
                   ft=3.281, feet=3.281, foot=3.281,
                   inches=39.37, inch=39.37,
                   mm=1000, millimeter=1000, millimeters=1000,
                   millimetre=1000, millimetres=1000,
                   cm=100, centimeter=100, centimeters=100,
                   centimetre=100, centimetres=100,
                   m=1.0, meter=1.0, meters=1.0, metre=1.0, metres=1.0,
                   km=0.001, kilometer=0.001, kilometers=0.001,
                   kilometre=0.001, kilometres=0.001)
    convert["in"] = 39.37
    numbers = frozenset("0123456789.eE")

```

在 class 语句的一开始会给出类的名字，提供一个能够在其中创建静态数据和方法的上下文。由于是从 object 中继承的，所以这个类是新式类。然后，创建了一些静态数据。先创建了一个字典，把名字映射到转换因子上。在参数名中不能使用“in”，因为这是 Python 的关键字，所以单独使用[]运算符来将其插入到字典中。还创建了一个用于检测浮点数数字有效性的字符集。

```

def __init__(self, length=None):
    if length is None:
        self.__amount = 0.0
    else:
        digits = ""
        for i, char in enumerate(length):
            if char in Length.numbers:
                digits += char
            else:
                self.__amount = float(digits)
                unit = length[i:].strip().lower()
                break
        else:
            raise ValueError, "need an amount and a unit"
        self.__amount /= Length.convert[unit]

```

在初始化程序中，本地变量 length、digits、i、char 和 unit 的作用一直到了方法的最后。我们引用的只有一个实例变量，self.\_\_amount。这个变量总是以米为单位来保存给定的长度值，无论在初始化程序中使用的单位是什么，任何方法对它都是可访问的。还引用了两个静态变量，Length.numbers 和 Length.convert。

在创建 Length 时，Python 会调用 \_\_init\_\_() 方法。用户有两个选择：不传递参数，此时 length 将会是 0 米，或者传递应字符串，指定一个数量和一个单位，用一个可选的空格符来分割两者。

如果给定了一个字符串，就希望对那些有效的数字进行遍历，然后把余数当成是单位。Python 的 enumerate() 函数会返回一个迭代器，每次遍历它都可以返回一个二元组，一个值是从 0 开始的索引数，一个是来自序列的相应元素。因此，如果该字符串中的 length 是 "7 mi"，那么返回的该元组将会是(0, "7")、(1, " ")、(2, "m") 和(3, "i")。只需在 for 循环中提供足够的变量，就可以解开元组。

一边对 numbers 集中的字符进行检索一边可将它们添加到 digits 字符串中。一旦遇到集中所没有的字符，就试着将 digits 字符串转换成 float，而 length 字符串中剩下的就应

当是单位了。我们会从单位字符串中剥除所有的先导和后置空白字符，并小写该字符串。最后，通过使用静态 `convert` 字典中的转换因子计算出所给的长度是多少米。

把数据属性称为 `_amount` 而不是 `amount`，是因为希望将这个数据作为 `private`。Python 将会对一个类中任何以两个下画线开头（而又不以两个下画线结尾）的名字进行名字改编（name-mangle），会在其前面添加一个下画线和类名，从而让属性的名字变得独一无二。在这个例子中，`_amount` 将会被改编成 `_Length__amount`。当查看这些特殊方法中的一些时，将看到为什么这么用的一个实际好处。

要时刻清醒地认识到，很多事情都是有可能出错的。倘若浮点数转换失败，或者没有给定单位（此时会抛出一个异常，其中会带一个“原因”字符串），又或者所给的单位与 `convert` 字典中的任何一个都不匹配。在这个方法中，就需要提前选定打算抛出的是哪个可能的异常，并在这个方法的文档字符串中进行文档化，以便这个类的用户可以知道有望得到什么。

```
def set(self, length):
    self.__init__(length)
```

由于希望让 `length` 能够可变，所以提供了一个 `set()` 方法。它带有的参数与 `__init__()` 一样，因为 `__init__()` 是初始化程序而不是构造函数，故而可以安全地给它传递要做的工作。

```
def to(self, unit):
    return self.__amount * Length.convert[unit]
```

在这个类中，`length` 是以米为单位保存的。这就意味着，需要维护的仅是一个单浮点数，而不是诸如一个值和一个单位这样的。不过，正如在创建 `length` 时我们可以给定偏爱的单位一样，也希望能够以我们选择的单位返回 `length` 值。这就是 `to()` 方法所要实现的。该方法会使用 `convert` 字典将以米为单位的值转换成给定的单位值。

```
def copy(self):
    other = Length()
    other.__amount = self.__amount
    return other
```

众所周知，如果使用 `=` 运算符，将只会绑定（或者重新绑定）一个名字，因此，如果希望能够对 `length` 进行真正的复制，就需要采用一些方法来实现它。在这里，我们选择提供一个 `copy()` 方法。不过这并不是说必须要这么做：这是因为，本来只需依靠 `copy` 模块就可以了。例如

```
import copy
import length

x = length.Length("3 km")
y = copy.copy(x)
```

这样既可以导入标准的 `copy` 模块，又可以导入我们自己的 `length` 模块（假设 `chap03` 是在 `sys.path` 中的，并且假设这个模块可由 `length.py` 调用）。然后创建了两个独立的 `length`。相反，如果当时做的是 `y=x` 并使用 `set()` 方法修改了 `x` 的值，`y` 可能也会发生改变。当然，由于已经实现了我们自己的 `copy()` 方法，通过用 `y=x.copy()` 应当也是可以实现复制的。

`copy()` 方法应当也可以有不同实现方法的。例如

```
def copy(self):      # Alternative #1
    import copy
    return copy.copy(self)

def copy(self):      # Alternative #2
    return eval(repr(self))
```

这些版本中的第一个版本使用 Python 的标准 copy 模块来实现 copy() 函数。第二个版本使用 repr() 方法为 length 提供了一个可 eval() 的字符串——例如，Length('3000.000000m')——然后，再使用 repr() 计算该代码；在这种情况下，它会构建一个与原始 length 同样大小的新的 length。

```
@staticmethod
def units():
    return Length.convert.keys()
```

之前已经提供过这个静态方法，可以让这个类的用户访问那些支持的单位名字。通过使用 keys()，可以确保返回一个单位名字的列表，而不是返回一个指向静态字典的对象引用。

利用 \_\_init\_\_() 初始化方法的异常，截至目前所看到任何一个方法都没有特殊方法。不过，我们希望 Length 类可以像标准 Python 类一样工作，以便可以使用诸如 \* 和 \*= 这样的运算符，也可以进行比较和转换成适当的兼容类型。通过实现一些特殊的方法可以达到这些目标。先从比较开始。

```
def __cmp__(self, other):
    return cmp(self._amount, other._amount)
```

这个方法的实现比较容易，因为只需要比较一下每个 length 的长度。

对象 other 可以是任何类型的对象。正是有了 Python 的名字改编机制，才使得 self.\_Length\_amount 和 other.\_Length\_amount 之间的比较成为可能。如果 other 对象没有 \_Length\_amount 属性，也就是说，如果它不是 Length，Python 就会抛出一个正是我们想要的 AttributeError 异常。这对于除 self 之外的其他带有 length 属性的全部方法都是真的。

没有名字改编机制，对不是 length 的其他对象就会有一些小危险，同样，对于 \_amount 属性也有同样的可能。还是使用类型测试来避免出现这一危险的，即使这在面向对象编程中的实践性显得稍稍差了些。

```
def __repr__(self):
    return "Length('%.6fm')" % self._amount

def __str__(self):
    return "%.3fm" % self._amount
```

Python 的浮点数精度取决于构建 Python 时所使用的编译器，不过，这种精度已经基本能够满足“representation”方法所要求的精确到小数点后六位的要求了。

对于字符串的表示，既不需要太过精确，也不需要返回一个可以 eval() 的字符串，因此，只需要能够返回原始的 length 并以米为单位即可。如果 Length 类的用户需要一个使用不同单位的字符串表示，可以使用 to()——例如，"% s miles" % length.Length("200 ft").to("miles")。

```
def __add__(self, other):
    return Length("%fm" % (self._amount + other._amount))

def __iadd__(self, other):
    self._amount += other._amount
    return self
```

之前曾用过两个特殊方法来支持加法。第一个特殊方法支持二元 +，两侧的操作数各是一个 length。它可以构建并且返回一个新的 Length 对象。第二个特殊方法支持 +=，可用于一个 length 和另一个 length 的自增。

它们可以允许代码写成如下这样：

```
x = length.Length("30ft")
y = length.Length("250cm")
z = x + y    # z == Length('11.643554m')
x += y      # x == Length('11.643554m')
```

对于混合类型的数学运算也有可能实现 `__radd__()`，不过我们并没有那么做，因为对 `Length` 类来说没有什么意义。

这里会忽略那些用于对减法运算支持的代码，因为它们基本上与加法的代码完全一样（在源文件中给的有）。

```
def __mul__(self, other):
    if isinstance(other, Length):
        raise ValueError, \
            "Length * Length produces an area not a Length"
    return Length("%fm" % (self.__amount * other))

def __rmul__(self, other):
    return Length("%fm" % (other * self.__amount))

def __imul__(self, other):
    self.__amount *= other
    return self
```

对于乘法运算的方法，提供了对一个 `length` 乘以一个数字的支持。如果假设 `x` 是 `length`，`__mul__()` 支持诸如 `x * 5` 和 `__rmul__()` 这样的用法，也支持诸如 `5 * x` 这样的用法。必须明确的一点是，不允许在 `__mul__()` 中乘以 `length`，因为其结果将会是一个面积而不是一个 `length`。在 `__rmul__()` 中并不需要这么做，因为总是会先测试 `__mul__()`，而如果抛出异常了，Python 就不会再测试 `__rmul__()` 了。`__imul__()` 方法支持原位（增量）乘法——例如，`x *= 5`。

```
def __truediv__(self, other):
    return Length("%fm" % (self.__amount / other))

def __itruediv__(self, other):
    self.__amount /= other
    return self
```

除法特殊方法的实现具有与其他数学方法类似的结构。之所以又再次给出它们的一个原因是想提醒我们，“/”和“/=”运算符执行浮点数除法是因为在 `length.py` 文件的一开头，添加了 `__future__ import division` 指示符。也有可能用它重新实现截断除法，不过对于 `Length` 类来说并不合适。

给出它们的另一个原因是，它们与之前看到的各个加法方法还是稍微有那么一点不一样的地方。尽管来说，加法和减法是只能对 `length` 进行操作，乘法和除法是对 `length` 和数字进行操作的。

```
def __float__(self):
    return self.__amount

def __int__(self):
    return int(round(self.__amount))
```

之前已经决定可以支持两种类型的转换，两个类型都要便于书写和理解。之前先实现了的 `__str__()` 方法也是一个类型转换方法（转换成 `str` 类型）。

已经看完了是如何实现一个自定义数据类型的，现在需要将注意力转到如何实现一个自定义集合类（collection class）上。

### 3.2.3 集合类

在 Python 中，集合就是诸如列表和字符串这样的序列，映射就是诸如字典或者集的序列。如果实现自己的集合类，可以使用一些特殊方法，让自己的集合可以像那些内置的集合类型一样使用同样的语法，具有同样的语义。表 3.3 给出了一些常用于集合的特殊方法，这一节将会讨论每种集合类型的一些特性。

表 3.3 集合的部分特殊方法

方法	语法	说明
<code>__contains__(self, x)</code>	<code>x in y</code>	如果 <code>x</code> 在序列 <code>y</code> 中或者如果 <code>x</code> 是 <code>dict y</code> 的一个键，返回 <code>True</code> 。这个方法也可以用于 <code>not</code> 的情况
<code>__len__(self)</code>	<code>len(y)</code>	返回 <code>y</code> 中元素的数量
<code>__getitem__(self, k)</code>	<code>y[k]</code>	返回序列 <code>y</code> 的第 <code>k</code> 个元素或者 <code>dict y</code> 中键 <code>k</code> 的值
<code>__setitem__(self, k, v)</code>	<code>y[k] = v</code>	把序列 <code>y</code> 的第 <code>k</code> 个元素设置成 <code>v</code> ，或者把 <code>dict y</code> 中键 <code>k</code> 的值设置成 <code>v</code>
<code>__delitem__(self, k)</code>	<code>del y[k]</code>	删除序列 <code>y</code> 的第 <code>k</code> 个元素或者 <code>dict y</code> 中键为 <code>k</code> 的元素
<code>__iter__(self)</code>	<code>for x in y: pass</code>	把迭代器返回到集合 <code>y</code> 中

对于序列的情况，通常会实现 `__add__()` 和 `__radd__()` 来支持用“+”的连接，而如果是在可变集合的情况下，则会实现 `__iadd__()` 来支持“+=”的连接。与之相似的是，应当实现 `__mul__()` 方法来支持“\*”，以实现集合的重复。如果给定的索引值无效，应当抛出一个异常。除了这些特殊方法外，一个自定义序列集合则应当实现 `append()`、`count()`、`index()`、`insert()`、`extend()`、`pop()`、`remove()`、`reverse()` 和 `sort()`。

对于映射来说，如果给定的键无效，应当抛出 `KeyError`，而除了这些特殊方法之外，还应当至少实现 `copy()` 和 `get()`，还有 `items()`、`keys()` 和 `values()`，以及它们的迭代器版本，比如 `iteritems()`。一个 Python 迭代器就是一个函数或者方法，可以返回连续值——例如，一个字符串中的每个字符，或者一个列表或者字典中的每个元素。它们通常用生成器实现。

对于集合，如果使用了非法键，也应当抛出一个 `KeyError` 异常；例如，在调用了 `remove()` 时。在集合中，应当分别实现 `issubset()`、`issuperset()`、`union()`、`intersection()`、`difference()`、`symmetric_difference()` 和 `copy()`。对于可变集，还应当提供一些额外的方法，包括 `add()`、`remove()` 和 `discard()`。

### 3.2.4 例：OrderedDict 类

Python 标准库的一个疏忽是缺少排序字典。普通字典可以提供非常快速的查询，但并不提供排序。例如，如果希望能够以键顺序的形式遍历字典的值，就需要把键复制到列表中，对列表进行排序，再对列表进行迭代，使用列表的各个元素来访问字典的值。对于小型字典，或者是偶尔做一下这样的事情，这样的排序倒没什么，不过，当字典非常大或者需要经常排序时，每次都做这样的排序可能就会占用非常大的计算资源。

对这一问题来说，答案显然是创建一个排序字典（ordered dictionary），这也是我们要在这里所做的事情。

深入理解 `OrderedDict` 示例对于学习 GUI 编程并没有什么大的必要，不过，在后续的一些程序中，确实还是会用到在这里所给出的技术和方法。所以，现在，尽可放心地跳到下一节继续阅读，然后再返回这里来理解在那些章节中所遇到的这些技术。

实现排序字典的一种方法可能是从 `dict` 继承，不过这里会用聚合（aggregation，也叫组合（composition））来实现，而把继承延迟到下一节中讲述<sup>①</sup>。

为得到排序字典，需要创建一个保存普通字典的类，同时还要创建一个该字典的键的排序列表。我们会实现全部的 `dict` API，但不会给出 `update()` 或者 `fromkeys()`，因为它们两个都超出了 GUI 编程中需要涉及的内容（当然，这两个方法代码都放在了源代码文件中，参见 `chap03/ordereddict.py`。）

该文件的第一个可执行语句是一个重要的语句：

```
import bisect
```

模块 `bisect` 提供了一些方法，可使用二元切分算法（binary chop algorithm）对诸如列表这样的排序序列进行搜索。很快就会在使用中看到它并对其进行讨论。

对于这个类，我们并不需要任何静态数据，所以将先从 `class` 的声明和 `__init__()` 的定义开始

```
class OrderedDict(object):
    def __init__(self, dictionary=None):
        self.__keys = []
        self.__dict = {}
        if dictionary is not None:
            if isinstance(dictionary, OrderedDict):
                self.__dict = dictionary.__dict.copy()
                self.__keys = dictionary.__keys[:]
            else:
                self.__dict = dict(dictionary).copy()
                self.__keys = sorted(self.__dict.keys())
```

这里创建了一个 `__keys` 列表和一个 `__dict` 字典。如果用另一个字典初始化 `OrderedDict`，就需要得到那个字典的数据。最简单也是最直接的获取数据的方法就是在 `else` 套件中所做的。把对象转换成字典（如果该对象本来就是字典，则不需花费任何代价），对其进行浅复制。然后，就可以得到该字典的键的排序列表。

在 `else` 套件中用到的方法也可用于各类情况，但如果仅是考虑效率，会用 `isinstance()` 引入一种类型测试。如果这个函数的第一个参数是类的一个实例，或者是以数个类（以元组的形式传递）来作为第二个参数，又或者是这些类的基类之一，它可以返回 `True`。因此，如果是从另一个 `OrderedDict`（或者是从 `OrderedDict` 的一个子类）初始化的，只需要浅复制它的字典，这时的代价与之前的一样，会浅复制它的键，这样做的代价低廉，因为它们已经排过序了。

由于字典已经排序，加上普通字典的那些方法，应当是可以以字典中的一种特定索引位置

<sup>①</sup> 严格来讲，聚合 aggregation 和组合 composition 两者稍有异同：聚合较松散，组合较紧密；两者都代表一种关联（association）关系。例如，几个点连在一起组成一个多边形，若多边形不存在，构成多边形的这些点也就不存在了，是一种组合关系；一个订单上包含几个产品，如果订单取消，产品还是要存在的，它们是一种聚合关系——译者注。

值来访问字典元素的各个值。这就是我们打算实现和提供的头两个方法

```
def getAt(self, index):
    return self._dict[self._keys[index]]

def setAt(self, index, value):
    self._dict[self._keys[index]] = value
```

方法 `getAt()` 会返回字典中第 `index` 个元素。它是通过使用列表在第 `index` 个位置所保存的键来访问字典实现这一功能的。方法 `setAt()` 使用同样的逻辑，除了它会设置用于第 `index` 个位置的字典元素的值之外。

```
def __getitem__(self, key):
    return self._dict[key]
```

如果已有一个字典，比如 `d`，并使用了 `value = d[key]` 语句，那么就会调用 `__getitem__()` 特殊方法。只需把工作传递给保存在 `OrderedDict` 类内部的字典即可。如果这个键没有在 `_dict` 中，它就会抛出一个 `KeyError`，这正是想要的，因为我们希望 `OrderedDict` 能够和 `dict` 具有同样的行为，除了在键顺序方面仍不一样之外。

```
def __setitem__(self, key, value):
    if key not in self._dict:
        bisect.insort_left(self._keys, key)
    self._dict[key] = value
```

如果用户使用 `d[key] = value` 给字典赋值，就又一次需要用 `_dict` 来完成工作。但是，如果该键还没有在 `_dict` 中，那么该键就也不能在键列表中，因此就必须把它加上去。

函数 `insort_left()` 带一个排序序列，比如一个排序列表，还带一个要插入的元素。它先会在序列中定位元素应该去的位置来保持序列的次序，再把元素插入到那里。`insort_left()` 函数，就像 `bisect` 模块中的其他所有函数一样，都可以使用二元切分算法，因此，即使是作用于非常长的序列，其性能也相当优秀。

另外一种方法应当就是仅简单追加新键，然后再对列表调用 `sort()`。Python 的排序功能已对部分排序的数据进行了非常高的优化，因此，性能应当不会差，不过我们倾向于能够取得更高效率的解决方案。

```
def __delitem__(self, key):
    i = bisect.bisect_left(self._keys, key)
    del self._keys[i]
    del self._dict[key]
```

元素的删除相当简单。函数 `bisect_left()` 会带一个排序序列，比如一个排序列表，还会带一个元素。它会返回一个元素在序列中的索引位置（或者如果元素在序列中，就返回该元素在序列中应当所在的位置）。我们认为该键是在列表中的，取决于它没在列表中时所抛出的那个异常。根据键列表中的索引值位置，可以删除该键，也可以根据字典中的键删除键值对(`key, value`)。

本来还是可以用一个简单语句，`self._keys.remove(key)`，删除键列表中的一个键，但这种语句应该先用一个较慢的线性搜索。

```
def get(self, key, value=None):
    return self._dict.get(key, value)
```

这个方法会返回给定键的值，除非给定的键没有出现在字典中，在这种情况下，它会返回一个特定的值（默认为 `None`）。由于并没有包含键次序，所以只需把工作传过去即可。

```
def setdefault(self, key, value):
    if key not in self._dict:
        bisect.insort_left(self._keys, key)
    return self._dict.setdefault(key, value)
```

这个方法与 `get()` 相似，不过，也有一个重要的不同之处：如果该键没有在字典中，会用给定的值进行插入。在这种键不在字典中的情况来说，当然必须把它插入到键列表中。

```
def pop(self, key, value=None):
    if key not in self._dict:
        return value
    i = bisect.bisect_left(self._keys, key)
    del self._keys[i]
    return self._dict.pop(key, value)
```

这个方法也与 `get()` 相似，不同的是，如果给定的键在字典中，这个方法就会移除该键元素。自然，如果一个键从字典中移除，也必须从键列表中移除它。

```
def popitem(self):
    item = self._dict.popitem()
    i = bisect.bisect_left(self._keys, item[0])
    del self._keys[i]
    return item
```

这个方法移除和返回一个任意元素，也就是说，一个(`key, value`)元组。先会从字典中移除这个任意元素(因为起先并不知道它是谁)，然后再从键列表中移除它的键，最后再返回所移除的元素。

```
def has_key(self, key):
    return key in self._dict

def __contains__(self, key):
    return key in self._dict
```

`has_key()` 方法可以向下兼容；程序开发人员所使用的 `in`，就是通过 `__contains__()` 特殊方法实现的。

```
def __len__(self):
    return len(self._dict)
```

这会返回字典中元素的数量。本想只像 `len(self._keys)` 一样简单返回即可。

```
def keys(self):
    return self._keys[:]
```

由于是以键列表的浅复制形式返回字典的键，所以它们是带有键顺序的。一个标准的 `dict` 则会以任意的次序来返回其键。

```
def values(self):
    return [self._dict[key] for key in self._keys]
```

按照键的次序返回字典的值。为实现这一点，会通过在一个列表解析中遍历键列表来创建值的列表。使用一个 `for` 循环应当是可以实现这一点的

```
result = []
for key in self._keys:
    result.append(self._dict[key])
return result
```

把代码写成一行而不是四行显然会让列表解析更具吸引力，尽管还是需要慢慢适应这一语法形式的。

```
def items(self):
    return [(key, self.__dict__[key]) for key in self.__keys]
```

对元素的返回与(key, value)元组的返回会使用类似的方法，这里还是会再次使用到传统的循环结构

```
result = []
for key in self.__keys:
    result.append((key, self.__dict__[key]))
return result
```

到现在，虽然列表解析应该才开始，但应该要更易懂些。

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

def iterkeys(self):
    return iter(self.__keys)
```

一个迭代器就是一个“可调用的对象”(callable object)(通常是一个函数或者方法)，可以返回每次调用后的“下一个”元素(那些对象都有一个next()函数，也是Python所调用的)。

通过使用iter()函数，可以获得一个可用于诸如字符串、列表或者元组这样的序列的迭代器，这个iter()就是我们在此要做的事情。对于字典来说，当需要一个迭代器时，就会返回一个字典的键的迭代器，尽管考虑到一致性，dict API也提供了一个iterkeys()方法，因为它也提供itervalues()和iteritems()方法。如果对一个字典调用iter()，比如一个OrderedDict实例，Python就会使用这个\_\_iter\_\_()特殊方法。

```
def itervalues(self):
    for key in self.__keys:
        yield self.__dict__[key]
```

如果调用itervalues()，就必须返回一个能够返回该字典的值的生成器。对于一个普通dict来说，这个生成器会以任意的顺序返回每个值，但对于OrderedDict来说，我们希望能够按照键的次序返回这些值。

包含yield语句的任何函数或者方法都是生成器。yield语句具有return语句类似的行为，不同的是，yield返回一个值之后，当再次调用该生成器时，还会用之前没有改变的状态从yield语句之后的地方继续执行。因而在这个方法中，在返回字典每个值后，for循环中的下一个迭代还会发生，直至返回所有的值。

```
def iteritems(self):
    for key in self.__keys:
        yield key, self.__dict__[key]
```

除返回的是一个(key, value)元组外，这些代码几乎是与itervalues()一模一样的(因为不会产生歧义，不需要使用括号来表明这里是一个元组)。

```
def copy(self):
    dictionary = OrderedDict()
    dictionary.__keys = self.__keys[:]
    dictionary.__dict = self.__dict.copy()
    return dictionary
```

对于复制，对键列表和内部字典执行的是浅复制，因此，花费的代价会与字典的大小成正比。

```
def clear(self):
    self.__keys = []
    self.__dict = {}
```

这是最简单的函数。本来应该使用 `list()` 和 `dict()` 而不是 `[]` 和 `{}` 的。

```
def __repr__(self):
    pieces = []
    for key in self.__keys:
        pieces.append("%r: %r" % (key, self.__dict[key]))
    return "OrderedDict(%s)" % ", ".join(pieces)
```

这里选择为我们的字典提供一种可 `eval()` 的形式(并且因为没有实现 `__str__()`，将会在需要用到字符串而不是字典时仍使用这个函数，例如，在一个 `print` 语句中)。对于每个(`key, value`)对，都会用到`%r`“表示”格式，例如，将要引用的字符串，而数字则不需要。这里会给出两个使用 `repr()` 的实际例子：

```
d = OrderedDict(dict(s=1, a=2, n=3, i=4, t=5))
print repr(d)
# Prints "OrderedDict({'a': 2, 'i': 4, 'n': 3, 's': 1, 't': 5})"
d = OrderedDict({2: 'a', 3: 'm', 1: 'x'})
print `d`    # Same as print repr(d)
# Prints "OrderedDict({1: 'x', 2: 'a', 3: 'm'})"
```

自然而然，本来也是可以使用列表解析来实现这个方法的，不过，在这个例子中，`for` 循环可能更容易理解。

现在就结束了对 `OrderedDict` 类的回顾。看起来，在这里和其他的 Python 集合中好像都遗漏了功能中的一部分，就是文件的加载和保存成文件。实际上，Python 具有集合，包括嵌套集合，加载和保存成字节字符串和文件的能力，前提是它们含有可以表示的对象，比如布尔值、数字、字符串以及那种对象的集合(实际上，Python 甚至可以加载和保存函数、类和在某些情况下的实例)。在第 8 章将会学习到这些功能。

### 3.3 继承和多态

正如期望 Python 能够像那些面向对象的编程语言一样，Python 也支持继承和多态。之前已经使用过继承，因为到目前为止，我们所创建的类都有被继承的 `object`，但在这一节，将会进行更为深入的讨论。Python 的所有方法都是虚拟的，所以如果重新实现基类中的一个方法，这个被重新实现的方法就将会是被调用的方法。很快就会看到如何访问基类的方法，例如，在希望把它们当成重新实现的方法的一部分使用时。

先从一个简单的类开始，这个类中保存了一件艺术作品的一些基本信息

```
class Item(object):
    def __init__(self, artist, title, year=None):
        self.__artist = artist
        self.__title = title
        self.__year = year
```

这里对 `object` 基类进行了继承，并为该类给定了三个私有数据属性。由于属性已经成为私有的，所以就必须要么为它们提供存取器(accessor)，要么创建一些特性(property)，以便可以通过这些特性来访问这些属性。在这个例子中，选择使用存取器的方法

```
def artist(self):
    return self.__artist

def setArtist(self, artist):
    self.__artist = artist
```

由于用于`_title`和`_year`属性的存取器在结构上与`_artist`属性的存取器一样，所以不会在这里给出它们。

```
def __str__(self):
    year = ""
    if self._year is not None:
        year = " in %d" % self._year
    return "%s by %s%s" % (self._title, self._artist, year)
```

如果需要一个字符串表示，若`_year`是`None`可以返回一个“`title by artist`”形式的字符串，否则可以返回一个“`title by artist in year`”形式的字符串。

既然可以封装一件艺术品的基本信息，自然就可以创建一个`Painting`子类，用来保存有关油画方面的一些信息

```
class Painting(Item):
    def __init__(self, artist, title, year=None):
        super(Painting, self).__init__(artist, title, year)
```

之前的代码就是一个完整的子类。由于还没有添加任何的数据属性或者新方法，因此只需使用`super()`内置函数来初始化`Item`基类即可。这个`super()`函数带有一个类并可返回该类的基类。如果这个函数还传递了一个实例（正如这里所做的那样），那么返回的基类对象将绑定到传入的实例上，这就意味着，可以向该实例调用一些（基类）方法。

也有可能通过明确命名基类来调用基类——例如，`Item.__init__(self, artist, title, year)`；值得注意的是，如果使用这种方法，就必须由自己传递`self`参数。

根本就不需要调用基类的`__init__()`——例如，如果基类本来就没有数据属性。而且，如果确实要调用它，那么`super()`调用就不应当是我们做出的第一个调用，尽管它通常就在`__init__()`实现中。

现在来看一个相对稍显复杂的子类

```
class Sculpture(Item):
    def __init__(self, artist, title, year=None, material=None):
        super(Sculpture, self).__init__(artist, title, year)
        self._material = material
```

类`Sculpture`有一个额外属性，因此，在通过基类初始化之后，还需要对这个额外的属性进行初始化。

这里不会给出各个存取器的代码，因为它们在结构上与应用于艺术家的名字上的那些存取器是一样的。

```
def __str__(self):
    materialString = ""
    if self._material is not None:
        materialString = " (%s)" % self._material
    return "%s%s" % (super(Sculpture, self).__str__(),
                     materialString)
```

`__str__()`方法使用基类的`__str__()`方法，并且如果材质已知，就会将它附加到结果字符串的末尾。不能调用`str(self)`，那样做会导致无限递归（一直调用`__str__()`），但是在必要时，明确地调用某一特殊方法并没有什么问题，就像这里所做的那样。

因为 Python 的多态特性，将总是可以调用到正确的`__str__()`方法。例如

```
a = Painting("Cecil Collins", "The Sleeping Fool", 1943)
print a    # Prints "The Sleeping Fool by Cecil Collins in 1943"
b = Sculpture("Auguste Rodin", "The Secret", 1925, "bronze")
print b    # Prints "The Secret by Auguste Rodin in 1925 (bronze)"
```

尽管已经使用特殊方法展示了多态性，但它的工作方式与普通方法的工作完全一样。

Python 使用动态类型，也叫做鸭子类型(duck typing)（“当看到一只鸟走起来像鸭子，游泳起来像鸭子，叫起来也像鸭子，那么这只鸟就可以被称为鸭子”）。这是非常灵活的。例如，假设现在有一个这样的类

```
class Title(object):
    def __init__(self, title):
        self.__title = title

    def title(self):
        return self.__title
```

现在应该就可以这样做

```
items = []
items.append(Painting("Cecil Collins", "The Poet", 1941))
items.append(Sculpture("Auguste Rodin", "Naked Balzac", 1917,
                      "plaster"))
items.append(Title("Eternal Springtime"))
for item in items:
    print item.title()
```

尽管各个元素是不同的类型，但还是会输出每个元素的题名。对于 Python 最重要的是，它们都支持所需的方法，在这个例子中就是 title()。

但是，如果存在一个元素集合但并不能确定这些集合的所有元素是否都支持 title() 方法时，该怎么办？根据这些代码的运行，应该可以得到一个 AttributeError 异常，根据它就可以快速找到不支持 title() 方法的那个元素。所以，一种解决方案就是使用异常处理机制：

```
try:
    for item in items:
        print item.title()
except AttributeError:
    pass
```

这些代码中包含着一个问题，就是说，一旦遇到不合适的元素，循环就会立刻停止。这可能会让我们产生借助 type() 或者 isinstance() 进行类型检查的愿望，例如：

```
for item in items:
    if isinstance(item, Item):
        print item.title()
```

这样，无论是对 Painting 还是对 Sculpture，都可以正常工作，因为它们都是 Item 的子类，但对 Title 对象却不能正常工作。而且，这种方法并不是真正好的面向对象风格。我们真正想做的可以认为是，“能够像鸭子一样叫吗？”，所以可以用 hasattr() 来完成：

```
for item in items:
    if hasattr(item, "title"):
        print item.title()
```

现在，这些元素可以是 Painting、Sculpture、Title 或者甚至是字符串（因为字符串是有 title() 方法的）。

尽管如此，还剩下一个问题：如何才能知道，这个属性就是一个方法——是一个可调用的方法——而不是一个数据属性？一种解决方法是使用 `callable()`。例如

```
for item in items:  
    if hasattr(item, "title") and callable(item.title):  
        print item.title()
```

这仍旧需要使用 `hasattr()`，因为必须对那些已经存在的东西调用 `callable()`（否则，将会收到一个异常），在这个例子中，实例属性就可以作为方法。

Python 的内省功能（introspection）是非常强大的，它拥有许多特性，比已见到的全部特性还要多。不过，除了 `isinstance()` 以外，它的使用是否明智还有待商榷。

有时，定义一个仅用来表示某个特定 API 的抽象基类（一个接口）会很有用。例如，艺术品和其他种类的元素都会有许多尺寸，因此，若能够提供一个带有 `area()` 和 `volume()` 方法的 `Dimension` 接口可能会很有用。尽管 Python 并没有正式支持接口，但通过实现一个没有数据属性的类，且该类的方法可以抛出 `NotImplementedError` 异常，还是可以实现想要的功能的。例如

```
class Dimension(object):  
  
    def area(self):  
        raise NotImplementedError, "Dimension.area()"  
  
    def volume(self):  
        raise NotImplementedError, "Dimension.volume()"
```

这些代码会定义出 `Dimension` 接口，它有两个所需的方法。如果对 `Dimension` 进行多重继承并忘了重新实现这些方法，那么在试图使用它们时，将会收到一个 `NotImplementedError` 异常。下面的代码给出的是 `Painting` 类的新版本，该类就用到了接口

```
class Painting(Item, Dimension):  
  
    def __init__(self, artist, title, year=None, width=None,  
                 height=None):  
        super(Painting, self).__init__(artist, title, year)  
        self.__width = width  
        self.__height = height
```

要计算油画的面积，就需要知道它的宽度和高度，因此会将这些信息加入到构造函数中，并给它们分配适当的属性

```
def area(self):  
    if self.__width is None or self.__height is None:  
        return None  
    return self.__width * self.__height  
  
def volume(self):  
    return None
```

`area()` 和 `volume()` 必须得以实现。尽管 `volume()` 方法对油画没有什么作用，但无论如何还是要实现它（因为接口需要一个这样的函数），因此这里会实现它并返回 `None`。另外，也应该要能够抛出异常——例如，`ValueError` 异常。

重新设计 `Sculpture` 类来接受宽度、高度和厚度参数并提供 `volume()` 实现就再自然不过了。但是，`area()` 的实现对 `Sculpture` 来说或许并不一定有什么意义。当然，这个函数或许意味着整个雕塑的总面积，又或许是从某个角度看一个面的面积。

由于存在歧义，所以要么通过额外传递一个参数来消除歧义，要么放弃并返回 `None` 或者抛出一个异常。

多重继承只是用于类语句中包含两个或者多个基类。在这个例子中，基类出现的顺序并不重要，但在更为复杂的层次结构中，这一点却很重要。

Python 面向对象编程的功能要远远超过本章打算覆盖的内容范围。例如，要尽可能简洁地保存一组固定的属性集，或许可以使用 `_slots_` 类属性。之所以在这里提到这一点，主要是想强调这样一个事实：这与我们在 GUI 章节中将要遇到的 PyQt 的槽(slot，就是一些函数和更为常用的方法)完全不同。又或许可以创建一些元类(meta class)，不过这样做也同样超出了 GUI 编程的内容范围，因此这里不需要讨论这一话题。

### 3.4 模块和多文件应用程序

面向对象编程允许把功能(例如，封装方法和数据属性)进行封装，如封装到类中。Python 模块允许在更高层次上封装那些导入的功能——例如，整个类集，或者全部实例变量。所谓的模块可以认为就是一个`.py`后缀的文件。模块可能会含有导入时要执行的代码，但更为常见的是，它们通常只是在导入时，提供一些函数和实例化的类。之前已经看到过这样的一些例子：在导入`length.py`文件中的`Length`类后，认为它是`length`模块中的类，故而就可以访问了。在导入一个模块时，只需给定不带文件后缀的模块文件名即可。例如

```
import length
a = length.Length("4.5 yd")
```

只有在当前目录下，或者是在 Python 的`sys.path`列表中的模块才能导入。如果需要导入文件系统中其他路径下的模块，就需要将这一路径添加到`sys.path`中。除代码文件外，模块还可以是一个只包含文件目录的文件。在这些情况下，该目录中必须含有一个名为`__init__.py`的文件。这个文件可以是(而且经常是)空的；它只用做标识，用来告诉 Python，这个目录含有一些`.py`文件，且这个目录名就是顶级模块的名字。例如，假设现在创建了一个名为`mylibrary`的目录，并把`length.py`、`ordereddict.py`和`__init__.py`都放在这个目录中。只要将含有`mylibrary`的目录加到 Python 的路径中，应当就可以像下面那样使用了：

```
import mylibrary.length
a = mylibrary.length.Length("14.3 km")
```

在实际应用中，可能更喜欢将`mylibrary.length`起一个更短的别名。例如这样

```
import mylibrary.length as length
a = length.Length("948mm")
```

Python 的模块处理机制要比这里所看到的复杂得多，但这些已经介绍过的内容对我们所关心的 GUI 编程来说已经足够了。Python 和 PyQt 应用程序可以写在一个文件中，或者可以分散在多个文件中。在接下来的几个章节中，将会给出这两种方法。

在 Python 2.7 中，模块导入的语义正在发生一些改变，模块的导入将变成绝对导入而不是相对导入。具体可以参阅 [http://www.python.org/dev/peps/pep-0328<sup>①</sup>](http://www.python.org/dev/peps/pep-0328)。

<sup>①</sup> 简单来说，绝对导入和相对导入的区别就是，Python 在搜索导入模块的文件时路径的默认顺序有所不同，类似于 C 或者 C++ 中`#include`语句后使用“”或者`< >`时，编译器搜索路径就会不同——译者注。

## 使用 doctest 模块

Python 全力支持测试，利用 doctest 和 unittest 模块进行单元测试，利用 test 模块进行回归分析测试。PyQt 还提供了 QTest 模块来实现单元测试的功能。

在创建各个模块时，例如，之前编写的 length 和 ordereddict 模块，可以使用导入应用程序（importing application）来导入它们和它们所提供的对象（例如，Length 和 OrderedDict 类）。但由于.py 文件也可以执行，因此可以轻松实现单元测试代码的包含：当导入该模块时，就会简单忽略单元测试的代码；但在模块运行时，就会执行单元测试。doctest 模块是支持这一方法的。

doctest 模块让单元测试变得尽可能的简单。要使用它，所需要做的就仅是将各个例子加入文档字符串中，看看要在交互式 Python 解释器（或者 IDLE）中输入什么内容以及希望得到什么样的响应即可。例如，这里给出的就是 OrderedDict 类 get() 方法

```
def get(self, key, value=None):
    """Returns the value associated with key or value if key isn't
    in the dictionary

    >>> d = OrderedDict(dict(s=1, a=2, n=3, i=4, t=5, y=6))
    >>> d.get("X", 21)
    21
    >>> d.get("i")
    4
    ...
    return self.__dict__.get(key, value)
```

该文档字符串会包含简要的方法目标描述，然后是一些像要被输入解释器中的例子。先从创建一个 OrderedDict 对象开始；并不需要导入或者限定什么资格，因为现在就是在 OrderedDict 模块中。然后，编写一个能够调用待测试方法以及希望解释器（或者 IDLE）做出何种预期响应的调用函数。接下来，就可以编写另外一个调用和响应了。

doctest 模块之所以使用这样的语法是因为，它对 Python 程序开发人员使用交互式 Python 解释器或者 IDLE，又或者诸如 Eric4 之类的其他 Python 集成开发环境是如此熟悉，以至于它认为可以直接嵌入一个 Python 解释器。当测试运行时，doctest 模块将会先导入模块自身，然后读取每一个文档字符串（使用 Python 的内省功能），再执行以 >>> 开头的每个语句。最后，它会检查执行结果和预期的输出（也可能什么都没有输出）是否相同，并报告所有的失败。

为了使一个模块能够像使用 doctest 模块一样，只需在模块的结尾处添加以下三行代码：

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

无论模块是通过 import 语句导入的，还是在命令行上调用的，模块的所有代码都会得到执行。这样就可以创建出模块的所有函数和类，从而为使用做好准备。

可以说说明一个模块是否被导入了，因为在导入的情况下，它的 \_\_name\_\_ 属性会被设置成该模块的名字。而如果没有导入，如果调用了该模块，它的 \_\_name\_\_ 属性就会被设置成 \_\_main\_\_。

正如之前所看到的那样，可以使用一个 if 语句来判断模块是否被导入，这样的话，就可

以不用再做其他什么了。但是，如果是在命令行上调用的模块，就可以导入 doctest 模块并执行 `testmod()` 函数，它会执行各类测试的。

还可以在控制台窗口中执行测试。例如

```
C:\>cd c:\pyqt\chap03  
C:\pyqt\chap03>ordereddict.py
```

如果测试没有失败，该模块将会安静地运行。如果有任何错误，这些错误就会被输出到控制台。使用 `-v` 标记，可以强迫 doctest 模块给出更为详细的信息

```
C:\pyqt\chap03>ordereddict.py -v
```

这样就可以给出每个单独测试的执行效果，还会在最后给出一个总结。

对于预期可能会出现错误的地方进行测试也是可以的，例如，要能够在可能出现错误的地方抛出一个异常等。对于这些，只需在期望输出的地方写上第一行和最后一行（因为对中间的回溯可能出现不同的结果）并使用一个省略号（…）来表明回溯即可。例如，这是回溯 `OrderedDict` 类 `setAt()` 方法的全部代码

```
def setAt(self, index, value):  
    """Sets the index-th item's value to the given value  
  
    >>> d = OrderedDict(dict(s=1, a=2, n=3, i=4, t=5, y=6))  
    >>> d.getAt(5)  
    6  
    >>> d.setAt(5, 99)  
    >>> d.getAt(5)  
    99  
    >>> d.setAt(19, 42)  
    Traceback (most recent call last):  
    ...  
    IndexError: list index out of range  
    """  
    self._dict[self._keys[index]] = value
```

这里创建了一个含有 6 个元素的 `OrderedDict` 类，但最后的测试会试图设置这个并不存在的第 20 个元素的值。这将导致字典抛出一个 `IndexError` 异常，因此要编写交互式 Python 解释器可能输出的内容，而 doctest 模块可以理解这一点，如果正确抛出了异常，就会通过测试。

doctest 模块并没有比 `unittest` 模块复杂，但它不仅易于使用而且并不唐突。在到目前为止的全部示例中均用到了它，正如在查看书中的源代码时所看到的那样。

## 小结

这一章首先让大家作为类的用户然后再成为类的创造者。可看到如何使用 `__init__()` 特殊方法来初始化和新创建实例，也看到了是如何实现其他特殊方法的，这样可以让自定义的数据类型（类）能够具有与 Python 内置类一样的行为特性。还学习了如何创建普通方法和静态方法，以及如何对每个实例和静态数据进行存储和访问。

这一章回顾两个完整的例子。`Length` 类，是一个数字型数据类型，而 `OrderedDict` 类，是一个集合类。这一章也用到了之前章节中的很多知识，包括 Python 的不少高级特性，比如列解析器和生成器方法等。

这一章既对单一继承做了说明，又给出了多重继承的内容，还用一个例子来说明该如何创建简单的接口类。这一章还学到了许多关于使用 `isinstance()` 进行类型测试以及 `hasattr()` 和鸭子类型的知识。

这一章最后以 Python 模块和多文件应用程序的工作模式概述结尾。此外，还看到了 `doctest` 模块，看到它如何创建单元测试，就像之前在我们的文档字符串中的例子那样简单。

现在已经具备了 Python 语言的一些基础知识。可以创建变量，使用集合，以及创建自己的数据类型和集合类型。可以进行分支、循环、调用函数和方法，还可以抛出和处理异常。显然，还有很多东西需要学习，但本书将会随着需求的增加而不断涵盖所需的知识点。现在已经可以开始进行 GUI 应用程序编程了，因而将在下一章开始这一主题，它也占用本书剩下的全部章节。

## 练习题

- 实现一个 `Tribool` 数据类型。这是一个数据类型，可以拥有下面三个值中的一个：`True`、`False` 或者 `unknown`(对于此种情况，应当使用 `None`)。除 `__init__()` 实现之外，还要实现 `__str__()`、`__repr__()` 和 `__cmp__()`；此外，也要实现用于转换成 `bool()` 的 `__nonzero__()`，用于逻辑非 `not (~)` 的 `__invert__()`、逻辑与 `and (&)` 的 `__and__()` 以及用于逻辑或 `or (|)` 的 `__or__()`。可能有两种逻辑可以使用：传导逻辑(`propagating`)，即任何包含 `unknown`(也就是，`None`) 的表达式都是 `unknown`；非传导逻辑(`nonpropagating`)，即任何包含 `unknown` 并可计算的表达式都是计算后的结果。

表达式	结果	表达式	结果	表达式	结果
<code>~t</code>	<code>False</code>	<code>~f</code>	<code>True</code>	<code>~n</code>	<code>None</code>
<code>t &amp; t</code>	<code>True</code>	<code>t &amp; f</code>	<code>False</code>	<code>t &amp; n</code>	<code>None</code>
<code>f &amp; f</code>	<code>False</code>	<code>f &amp; n</code>	<code>False</code>	<code>n &amp; n</code>	<code>None</code>
<code>t   t</code>	<code>True</code>	<code>t   f</code>	<code>True</code>	<code>t   n</code>	<code>True</code>
<code>f   f</code>	<code>False</code>	<code>f   n</code>	<code>None</code>	<code>n   n</code>	<code>None</code>

例如，使用传导逻辑时，`True | None` 是 `True`，因为只要逻辑 `or` 中的一个操作数是 `True`，这个表达式的结果就是 `True`。但是，`False | None` 的结果是 `None (unknown)`，因为并不能确定这个表达式的结果。

大多数方法都可以通过数行代码实现出来。确保要使用到 `doctest` 模块，并且要为所有的方法都编写单元测试的代码。

- 实现一个 `Stack` 类和一个 `EmptyStackError` 异常类。`Stack` 类应该使用列表来存储它的各个元素，并应该提供 `pop()` 来返回和移除栈顶的元素(最右边的元素)，提供 `top()` 来返回栈顶的元素，还应该提供 `push()` 来将一个新的元素压入栈中。此外，也要提供一些特殊方法，以便让 `len()` 和 `str()` 能够合理工作。要确保 `pop()` 和 `top()` 在调用空栈时，它们可以抛出 `EmptyStackError`。这些方法可以用很少的代码写出来。确保要使用到 `doctest` 模块并为所有的方法都编写了单元测试的代码。

本练习题的参考答案在文件 `chap03/tribool.py` 和 `chap03/stack.py` 中。



## 第二部分

# GUI 编程基础

---

第 4 章 GUI 编程简介

第 5 章 对话框

第 6 章 主窗口

第 7 章 使用 Qt 设计师

第 8 章 数据处理和自定义文件格式

## 第 4 章 GUI 编程简介

- 25 行的弹出式闹钟
- 30 行的表达式求值程序
- 70 行的货币转换程序
- 信号和槽

在这一章，我们会先从简要回顾三段用 PyQt 写成的简单但却很有用的 GUI 程序开始。将利用这一机会，着重强调一些 GUI 编程中可能会涉及的问题，但对其详细的介绍则仍会放到后面的章节中。一旦对 PyQt GUI 编程有了初步感觉，就会来探讨 PyQt 的“信号和槽”机制，这是一个用来响应用户交互操作的高级通信机制，从而允许我们忽略掉那些无关的细节。

尽管 PyQt 在商业上建立的不同应用程序，大小从几百行到 100 000 行不等，但在这一章要介绍的这些程序都会在 100 行以内，它们会说明，即使仅用到很少的代码，也是可以实现很多功能的。

在这一章，我们会仅用手写代码的方式来构建一些用户界面，但是在第 7 章，将会学习如何使用 Qt 设计师 (Qt Designer) 这一可视化图形工具来创建用户界面。

Python 的控制台应用程序和 Python 的模块文件总是会使用 .py 的文件后缀，不过对于 Python GUI 应用程序来说，则会使用 .pyw 的文件后缀。无论是 .py 还是 .pyw，它们在 Linux 平台上用起来都很好；但是在 Windows 平台上，需要确保 Windows 用 pythonw.exe 解释器而不是 pythonw.exe 解释器来运行 .pyw 文件后缀，以便能够让我们在运行 GUI 应用程序时，不会弹出一个无用的控制台窗口<sup>①</sup>。在 Mac OS X 上，一定要使用 .pyw 文件后缀。

PyQt 的文档是以系列 HTML 文件的形式提供的，这些文件独立于 Python 文档之外。最常用到的会是含有 PyQt API 的那些文档。这些文件都是从最初的 C++/Qt 文档转换而来的，classes.html 就是它们的索引页；Windows 用户在系统“开始”按钮的 PyQt 菜单中可以找到指向这个页面的链接。很有必要看一下该页面，从中可以看出哪些类是可用的，当然，也可以对看起来比较感兴趣的那些类仔细阅读一下。

我们即将看到的第一个应用程序会是一个不太寻常的混合程序：一个需要从控制台启动的 GUI 程序，这是因为它运行时需要一些命令行参数。用到这个程序是因为它可以更为简单地说清 PyQt 事件循环机制的工作方式（以及该机制的定义），而不会深入接触其他 GUI 细节。第二个和第三个样例都是非常短小却标准的 GUI 应用程序。它们都展示了是如何来创建和布局窗口部件（Windows 中称为控件，control）的，如标签、按钮、下拉菜单和其他那些用户可以在屏幕上看到且在大多数情况下可以交互的那些元素。这两个例子也说明了该怎样来响应用户的交互，例如，当用户执行某一特定动作时该如何去调用某个特定的函数或者方法。

在最后一节，将会更进一步讲述如何来处理用户的交互，而在下一章，还将会更为全面地讲述布局和对话框的内容。在这一章，只需初步知道这些事情是如何工作的即可，而不必过多

---

<sup>①</sup> 如果使用的是 Windows 并弹出了一个题为“pythonw.exe-Unable To Locate Component”的错误消息对话框，那就很可能意味着是其路径设置不正确。请参阅附录 A 中的相关内容来修正这一问题。

关注各个细节：随手的一些章节将会弥补这些知识欠缺，并且会让你逐渐熟悉那些标准的 PyQt 编程实践。

## 4.1 25 行的弹出式闹钟

第一个 GUI 应用程序看上去有点儿古怪。首先，它必须从控制台启动；其次，它没有装饰栏(decoration)——没有标题栏，没有系统菜单，没有关闭按钮。图 4.1 给出的是示意图。

要得到如图所示的输出，需要在命令行键入如下命令：

```
C:\>cd c:\pyqt\chap04
C:\pyqt\chap04>alert.pyw 12:15 Wake Up
```

启动后，程序会在后台默默运行，仅是简单地记录时间直到达到给定的时间。此时，程序会弹出一个文本消息窗口。大约窗口显示后一分钟，程序会自动终止。

Wake Up

图 4.1 闹钟程序

这里所给定的时间必须使用 24 小时制。为了测试，可以使用刚刚过去的时间；例如，现在时刻是 12:30 了，但我们可以使用 12:15，此时窗口就会即刻弹出(好吧，暂且认为是一秒之内吧)。

现在已经知道了这个程序能干什么以及该如何去运行它，那么还是先回顾一下它的实现内容吧。这个文件会稍微比 25 行多几行，这是计数时没有包含文件里面的一些注释行和空行，但的确是只有 25 行的执行代码。就先从 import 开始吧。

```
import sys
import time
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

导入 sys 模块，是因为我们想访问包含在 sys.argv 列表中的那些命令行参数。导入 time 模块，则是因为我们需要它的 sleep() 函数；而导入这两个 PyQt 模块，则是因为需要使用 GUI 和 QTime 类。

```
app = QApplication(sys.argv)
```

先从创建 QApplication 对象开始。每个 PyQt GUI 应用程序都必须有一个 QApplication 对象。这个对象会提供访问全局信息的能力，如应用程序的目录、屏幕大小(以及对于多线程系统来说，这个应用程序是在哪个屏幕上)等。这个对象还会提供稍后要讨论的事件循环。

在创建 QApplication 对象时，给它传递了命令行参数；这是因为，PyQt 可以识别一些自己的参数，如 -geometry 和 -style，所以需要给一个让它读到这些参数的机会。如果 QApplication 能够识别任意一个参数，它就会对它们进行一些操作，并将其从给定的参数列表中移除掉。QApplication 能够识别的那些参数列示在 QApplication 的初始化文档中。

```
try:
    due = QTime.currentTime()
    message = "Alert!"
    if len(sys.argv) < 2:
        raise ValueError
    hours, mins = sys.argv[1].split(":")
    due = QTime(int(hours), int(mins))
    if not due.isValid():
```

```

raise ValueError
if len(sys.argv) > 2:
    message = " ".join(sys.argv[2:])
except ValueError:
    message = "Usage: alert.pyw HH:MM [optional message]" # 24hr clock

```

在比较靠后的地方，这个程序需要一个时间点，所以把 `due` 变量值设定为现在的时间。我们还提供了一个默认消息。如果用户没有给定哪怕至少一个的命令行参数值(时间)，就会抛出(`raise`)一个 `ValueError` 异常。这样就会显示一个当前时间，并且会给出含有“用法”错误的消息。

如果第一个参数不含冒号，那么在尝试调用 `split()` 将两个元素解包时，就将触发一个 `ValueError` 的异常。如果小时数和分钟数不是有效的数字，那么 `int()` 将会触发一个 `ValueError` 异常；而如果小时数和分钟数超限，导致成了无效的 `QTime`，那么就需要自己来触发一个 `ValueError` 异常。尽管 Python 提供了自己的 `date` 和 `time` 类，但 PyQt 的 `date` 和 `time` 类通常会显得更为方便些，所以推荐使用 PyQt 的这些类。

如果该 `time` 有效，那么就把这条消息与其他命令行参数(如果还有的话)用空格分隔开；如果没有其他参数，就用开头设置的默认信息“Alert!”来代替之(当一个程序是用命令行方式启动时，会给定一系列的参数，第一个称为调用名，而剩下的其他非空字符，也就是输入命令行中的其他每个单词，都称为字符序列。这些字符序列或许会被 shell 改变，例如，可能会用通配符进行匹配。Python 自己的字符序列位于 `sys.argv` 列表中)。

现在，我们就知道了消息会在何时必须予以显示，以及要显示的信息是什么。

```

while QTime.currentTime() < due:
    time.sleep(20) # 20 seconds

```

把当前时间和目标时间连续不断地进行循环比较。如果当前时间超过了目标时间，循环就会停止。原本是在循环体内放一个 `pass` 声明(statement)的，但如果是这样的话，Python 就会非常快地一遍遍执行这个循环，而过多地占用处理器可不是什么好现象。`time.sleep()` 命令会让 Python 暂停处理一段时间后再继续处理，这里暂停的时间是 20 秒。这样就可以让机器上的其他程序也能够得到更多的运行机会，因为我们也不想在等这个程序到达截止时间的期间什么也不做。

暂且先不提创建 `QApplication` 对象，目前所做的仅仅就是标准的控制台程序设计工作。

```

label = QLabel("<font color=red size=72><b>" + message + "</b></font>")
label.setWindowFlags(Qt.SplashScreen)
label.show()
 QTimer.singleShot(60000, app.quit) # 1 minute
app.exec_()

```

在创建了 `QApplication` 对象后，就会显示一条消息，因为截止时间已到，所以现在就可以创建程序了。一个 GUI 程序需要一些窗口部件，这样就需要用有一个标签(label)来显示该消息。`QLabel` 可以接收 HTML 文本，所以就给它一个 HTML 字符串，让它能够将这段消息显示为加黑、红色、大小为 72 点素(point)<sup>①</sup>。

在 PyQt 中，任何窗口部件都可以作为顶级窗口，即使是按钮和标签都行。当窗口部件这

<sup>①</sup> 在 <http://doc.qt.io/qt-4.8/richtext-html-subset.html> 中给出了所支持的 HTML 标签。

么用的时候，PyQt 就会自动给它加一个标题栏。在这个应用程序中，并不想要一个标题栏，所以要把该标签的各个窗口标记(flag)均设置为闪屏(splash screen)模式，因为这种模式不会有标题栏了。一旦把要作为窗口的标签设置完毕，就可以对其调用 `show()` 函数了。此时，该标签窗口却没有显示出来！`show()` 的调用仅仅是计划执行一个“重绘事件”(paint event)，也就是说，`show()` 会向 `QApplication` 对象的事件队列中添加一个新的事件，以请求对特定的窗口部件进行绘制。

接下来，会设置一个单次定时器(single-shot timer)。这是因为 Python 库的 `time.sleep()` 函数使用的时间是秒，而 `QTimer.singleShot()` 函数使用的是毫秒。给 `singleShot()` 方法两个参数：让这个时间需要持续多长(这里是 1 分钟)，时间到期后调用哪个函数或者方法。

用 PyQt 的术语来说，上述给出的方法或者函数叫做“槽”(slot)，尽管在 PyQt 的文档中，也会用术语“可调用”(callable)、“Python 槽”(Python slot)和“Qt”槽(Qt slot)来区分 Python 的 `_slots_`，这是在 Python 语言指南(Python Language Reference)中给出的一个新特性类。在本书中，会使用 PyQt 的术语，因为本书中就没有用到 `_slots_`。

所以现在要计划执行两个事件：一个是需要立即执行的绘制(paint)事件，一个是需要在一分钟后执行的定时器(timer)超时事件。

调用 `app.exec_()` 会开始执行 `QApplication` 对象的事件循环<sup>①</sup>。第一个事件就是绘制事件，所以标签窗口会带着给定的消息显示在屏幕上。大约一分钟之后，会发生定时器的超时事件，此时会调用 `QApplication.quit()` 方法。这个方法会干净地结束掉该 GUI 应用程序。它会关闭所有已打开的窗口，释放所有占用的资源，然后退出程序。

在各类 GUI 应用程序中都会用到事件循环。用伪代码的形式来表示事件循环会是这个样子

```
while True:  
    event = getNextEvent()  
    if event:  
        if event == Terminate:  
            break  
        processEvent(event)
```

当用户与应用程序交互时，或者是在发生了特定事情时，比如定时器超时或者应用程序的窗口被遮盖(或许是因为关闭了另一个应用程序)等，就会在 PyQt 内部产生一个事件并将其添加到事件队列中去。应用程序的事件循环会持续不断地查看是否还有需要执行的事件，如果有，就执行该事件(或者将其传递给与事件相关联的处理函数或者方法)。

尽管完成这个应用程序只用到了一个简单的窗口部件，并且使用控制台程序看起来确实很有效。然而，还没有给这个应用程序赋予任何可与用户交互的能力。这个应用程序的工作模式看起来也与传统的批处理程序很相似，如图 4.2 所示。调用它，执行一些处理工作(等待，然后显示一条消息)，然后结束。大部分的 GUI 应用程序的工作模式却与之并不相同。一旦调用运行，它们就会执行其事件循环来响应不同的事件。一些事件是用户产生的——例如按下键盘、点击鼠标等——而有些事件则是由系统产生的，例如定时器到达设定时间、窗口重绘等。这些 GUI 应用程序仅会对作为事件结果的请求做出响应，如单击按钮、选择菜单等，并仅会在要求终止时才结束程序。

<sup>①</sup> PyQt 会使用 `exec_()` 而不是用 `exec()` 来避免与 Python 内置的 `exec` 声明之间的冲突。

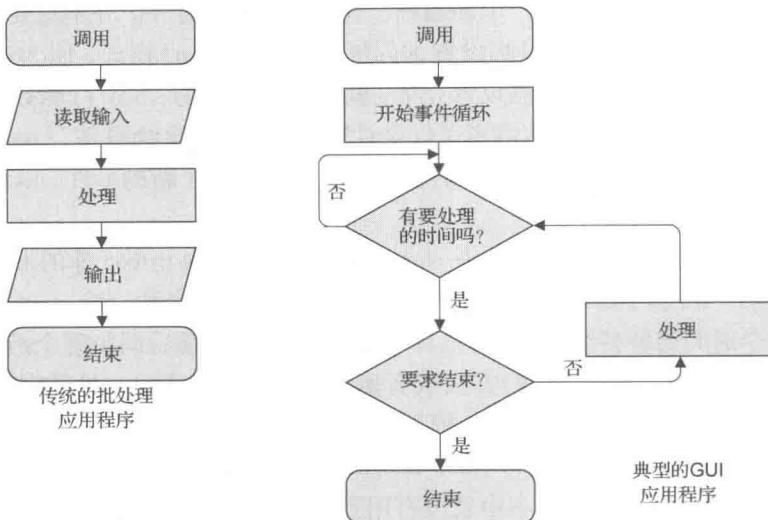


图 4.2 批处理应用程序与 GUI 应用程序

下一个要看到的应用程序将会比我们刚刚看到的这个应用程序更常规些，并且会像一般的小型 GUI 应用程序一样有代表性。

## 4.2 30 行的表达式求值程序

这个用 30 行写成的应用程序纯粹就是一个对话框风格的应用程序(除了空行和注释行之外)。所谓的“对话框风格”，是指这个应用程序没有菜单栏，也不像常规的那样有工具条或者状态栏，因为大部分应用程序都是带有一些按钮的(正如下一节所看到的那样)，而且也没有中心窗口部件(central widget)。相比较而言，“主窗口风格”的应用程序通常会有菜单栏、工具条、状态栏，某些情况下也会有一些按钮；并且，都会有一个中心窗口部件(当然，会包含一些其他的窗口部件)。在第 6 章我们将会看到一些主窗口风格的应用程序。

这个应用程序用到了两个窗口部件：一个是 QTextBrowser，这是一个只读的多行文本框，既可以显示普通文本又可以显示 HTML；还有一个是 QLineEdit，这是一个单行文本框，可用来显示普通文本。在 PyQt 的窗口部件中，所有文本都会采用统一字符编码标准(Unicode)，尽管在必要时也可以将其转码成其他编码形式。

如图 4.3 所示的 Calculate 应用程序可以像其他常规 GUI 应用程序一样通过点击其图标来运行它(或者是双击，这取决于所在的平台和设置情况)(当然，也可以通过控制台来启动它)该应用程序只要一开始运行，用户就可以在单行文本框中轻松输入数学表达式，在输入回车键 Enter(或者是换行键 Return)时，表达式及其结果就会显示在 QTextBrowser 上。如果是表达式无效或者计算规则非法(比如被零除)，就会将其转换成错误消息并直接在 QTextBrowser 上显示此类错误消息。

像平时一样，先来分段浏览一下这些代码。这个例子给出了一些在以后的 GUI 应用程序

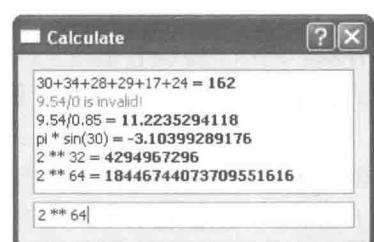


图 4.3 Calculate 应用程序

中都会用到的模式：类会用表单(form)的形式来进行表示，用来响应用户交互的行为会由方法进行处理，而程序的“main”部分则会很短小。

```
from __future__ import division
import sys
from math import *
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

由于我们要做的只是数学计算也不想要其他诸如截断除法<sup>①</sup>这类的算法，只需确保能够实现浮点数除法即可。一般可以通过使用 `import moduleName` 这样的语法来导入非 PyQt 的模块；但由于想让该程序的用户能够用到 `math` 模块里面的全部函数和常量，所以只需把 `math` 模块里面的所有方法导入到当前的命名空间即可。像往常一样，还会导入 `sys` 模块，以便可以获得 `sys.argv` 的参数列表，最后还会导入 `QtCore` 和 `QtGui` 模块里面的所有东西。

```
class Form(QDialog):
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)
        self.browser = QTextBrowser()
        self.lineedit = QLineEdit("Type an expression and press Enter")
        self.lineedit.selectAll()
        layout = QVBoxLayout()
        layout.addWidget(self.browser)
        layout.addWidget(self.lineedit)
        self.setLayout(layout)
        self.lineedit.setFocus()
        self.connect(self.lineedit, SIGNAL("returnPressed()"),
                     self.updateUi)
        self.setWindowTitle("Calculate")
```

正如之前看到的，任何窗口部件都可以用做顶级窗口。但在大多数情况下，我们会通过对 `QDialog` 或者 `QMainWindow`，偶尔也会用到 `QWidget`，子类化的方法来创建自己的顶级窗口。但无论是 `QDialog` 还是 `QMainWindow`，实际上，PyQt 里面的所有窗口部件都是继承自 `QWidget` 的，也都是新类型(new-style)的类。通过继承 `QDialog` 可以得到空白的表单(form)，这是一个灰色的矩形，还可以得到一些方便的行为和方法。例如，如果用户单击关闭按钮 X，那么这个对话框就会关闭掉。默认情况下，关闭一个窗口部件，事实上仅仅是将其隐藏起来了；当然，就像下一张将会看到的那样，我们当然也可以改变这一行为。

令 `Form` 类 `__init__()` 方法默认的父类 `parent` 为 `None`，用 `super()` 方法对其进行初始化。没有父类的窗口部件就会变成顶级窗口，这正是该 `Form` 类所需要的。然后，创建两个窗口部件，为了后续在 `__init__()` 之外仍旧能够访问它们，还需要对它们保持引用。由于这些窗口部件没有给定父类，看上去它们也会变成顶级窗口的，但其实这并不是我们所期待的。在初始化函数中，将会看到它们是如何获得父类的。对于 `QLineEdit` 可以给定一些作为初始文本，并将这些文本全部选中。这样就可以确保当用户开始输入信息时，我们这里所给定的这些文本会被直接覆盖。

我们想让这些窗口部件能够一个接一个地在窗口中竖直显示出来。要达到这一效果，可以通过先创建一个 `QVBoxLayout` 并向其添加我们所创建的那两个窗口部件，然后再对 `Form` 的布局进行设置。如果运行该应用程序并改变该程序的大小，可以发现，那些竖向多

<sup>①</sup> 计算机的整数除整数的规则——译者注。

余的空间都会分配给 `QTextBrowser`, 而这两个窗口部件则都会在水平方向上拉长。这些都会由布局管理器(layout manager)自动处理的, 而通过设置布局策略还可以获得非常精准的调整效果。

使用布局会出现一个重要的副作用: PyQt 会自动将布局中的各个窗口部件重定义其父类。所以, 尽管对于我们的那两个窗口部件并没有给定各自的父类(在该 Form 实例中), 但是当调用 `setLayout()` 的时候, 布局管理器还是会获得这两个窗口部件和该 Form 自身的所有权, 也会获得任何嵌套布局自身的所有权。这样一来, 布局后的窗口部件就没有哪一个再是顶级窗口了, 因为它们都有父类了, 而这也正是我们想要的。因此, 当删除 Form 实例的时候, 它所有的子窗口部件和布局都会随它一起按照正确的顺序删除掉。

对于表单 Form 里面的每个窗口部件都可以使用多种技术进行布局。可以使用 `resize()` 和 `move()` 方法为它们给定绝对大小和位置; 可以重写 `resizeEvent()` 方法来动态地计算它们的大小和位置坐标, 或者也可以直接使用 PyQt 的布局管理器。使用绝对大小和位置会非常不方便。一方面, 需要进行大量的人工计算, 而另一方面, 如果布局改变了, 就要重新计算。动态计算大小和位置是一种更好些的方案, 不过这依然需要我们编写许多冗长枯燥的计算代码。

使用布局管理器会让这一切变得容易很多。同时, 布局管理器也要灵活得多: 会自动适应改变大小的事件并去满足这些改变。任何习惯了不同版本 Windows 下的对话框的人, 相信都会更喜欢大小可以改变(也是很符合实际的), 而不是强制使用一个小小的、不能改变大小的对话框, 因为这样的话会非常不方便, 尤其是当其内容很多而无法满足的时候。布局管理器也会让程序在国际化时变得更为简单, 因为它们可以自动适应内容, 所以翻译后的标签就不会出现因目标语言比源语言冗长而被截断的现象。

---

## 对象的所有权

所有派生自 `QObject` 的 PyQt 类——这就包括了全部的窗口部件, 因为 `QWidget` 就是一个 `QObject` 的子类——都有一个“父类”。这种父—子关系可用于两种互补目的。一个没有父类的窗口部件就会是顶级窗口, 而一个有父类(通常会是另一个窗口部件)的窗口部件就会被包含(或者显示)到它的父类中。这一关系也就定义了所有权, 不同的父类都会拥有它们的孩子。

PyQt 使用父-子拥有权模型来确保如果一个父类(例如, 一个顶级窗口)被删除, 那么它的所有孩子, 即该窗口所包含的全部窗口部件都应当被自动删除掉。为了避免出现内存泄漏, 我们应当总是确保任何 `QObject`, 包括全部的 `QWidget` 都有一个父类, 而只有顶级窗口找一个例外。

大多数的 PyQt `QObject` 子类都会在其构造函数中的最后一个参数处(可选)带一个父类对象。但对于窗口部件一般不会(也不需要)传递这个参数。这是因为, 对话框中的窗口部件是用布局管理器进行布局的, 而一旦出现布局, 各个窗口部件就会自动重定向父类, 指向它们所在的布局, 因而它们会自动正确地改正自己的父类而无须我们做任何其他动作。

在某些情况下, 就必须要明确传递一个父对象——例如, 在创建那些不是窗口部件的 `QObject` 子对象时, 或者是在创建一些不进行布局的窗口部件时(比如是一些停靠窗口部件); 在随后的各个章节中, 将会看到数个这种情况的例子。

最后要说明的一点是, 也可能会发生这样一些情况, 就是 Python 变量正引用的是一个已经不再存在的 PyQt 对象。这一问题会在第 9 章进行讲述。

---

PyQt 提供了三种布局管理器：一是垂直布局，二是水平布局，三是网格布局。这些布局可以相互嵌套，所以即使是非常复杂的布局也有可能做出来。当然，也有其他的布局方式，比如还可以使用分隔符(splitter)或者 tab 窗口部件。所有这些内容会在第 9 章做深入讲解。

需要温馨提示的是，我们打算先让光标从 QLineEdit 开始，为此需要调用 setFocus()。在设置完布局后，就必须要做这一步。

有关 connect() 的调用会在本章稍后的地方深入讲述。简单地说，任何一个窗口部件(以及其他某些 QObject 对象)都可以通过发射(emit)“信号”的方式来声明状态和槽发生了改变。这些信号(与 UNIX 的信号没有任何关系)通常会被忽略掉。然而，我们也可以选择任何感兴趣的信号，通过识别想知道的 QObject 就可以做到这一点，因为该 QObject 所发射的信号就恰是我们想知道的，而在这个信号发射的时候就会调用我们想要调用的函数或者方法。

所以，在这种情况下，当用户在 QLineEdit 上按下回车键 Enter(或者换行键 Return)的时候，returnPressed() 信号将会像往常一样发射出来，但因为有 connect() 调用，当这个信号一发射，就会调用 updateUi() 方法。于是，瞬间我们就会看到发生了什么。

在 \_\_init\_\_ 中做的最后一件事情就是设置窗口的标题。

正如很快就会看到的一样，创建了表单 Form，对其调用了 show() 方法。一旦事件循环开始，表单就会显示出来，好像再没有其他什么会发生了。应用程序会一直运行事件循环，等待用户去按下鼠标或者键盘。而一旦用户开始交互，所有的交互动作就会得以处理。因此，如果用户输入了一个表达式， QLineEdit 就会将用户输入的表达式显示出来，而只要用户按下的回车键，就会调用 updateUi() 方法。

```
def updateUi(self):
    try:
        text = unicode(self.lineEdit.text())
        self.browser.append("%s = <b>%s</b>" % (text, eval(text)))
    except:
        self.browser.append(
            "<font color=red>%s is invalid!</font>" % text)
```

当调用 updateUi() 时，它会获得 QLineEdit 的文本，即刻将它转换为 unicode 对象。然后，使用 Python 的 eval() 函数计算这个作为表达式的字符串的值。如果成功，将计算的结果添加到带有表达式文字、一个等号(=)的 QTextBrowser 的后面，再把结果加粗显示。尽管通常会尽可能地把 QString 转换为 unicode 字符，也可以向那些能够接受 QString 的各个 PyQt 方法传入 QString、Unicode、strs 字符串，而 PyQt 则会自动进行所需的转换工作。如果发生了异常，就把错误信息添加到 QTextBrowser 里面。像这样使用一个捕获所有异常的 except 代码块在实际编程时并不是好方法，不过在这个只有 30 行的程序里，还算情有可原的。

使用 eval() 可以避免由语法和解析所带来的全部工作，而如果使用的是编译型语言的话，这些工作都应自己完成。

```
app = QApplication(sys.argv)
form = Form()
form.show()
app.exec_()
```

现在，Form 类的定义已经完成了，在 calculate.pyw 文件的最后，还需要创建一个 QApplication 对象，以用来对所创建的 Form 类进行实例化，然后将其放入绘制序列中，再开始事件循环。

这样就完成了全部程序。然而，这并不是故事的结局。目前还没有涉及用户该如何来结束这个程序。由于我们的 Form 类是从 QDialog 派生的，因而就会继承一些有用的行为。例如，如果用户点击了 X 关闭按钮，或者是按下了 Esc 键，那么表单 (form) 就会关闭。当一个表单 (form) 关闭的时候，它只不过是隐藏起来了。当表单 (form) 隐藏起来时，PyQt 将会检测该程序不要再有可见的窗口，这样也就不可能再有更多的交互了。PyQt 就会删除该表单 (form) 并对该应用程序执行清空操作。

有些情况下，即使应用程序不可见，但仍旧希望该程序依然能够运行，例如，一个服务器应用程序。对于这些情况，可以调用 QApplication.setQuitOnLastWindowClosed (False) 来实现前述目的。也有可能，尽管这种情况非常少见，在关闭应用程序的最后一个窗口时，要能通知用户。

在 Mac OS X 操作系统上，或者是在某些 X Windows 窗口管理器中，诸如像 twm 这样的应用程序是没有关闭按钮的，同样在 Mac 上，就是选中菜单栏上的退出也是没用的。在这些情况下，就需要按下 Esc 键来终止程序；此外，在 Mac 上还可以使用 Command +。从这个角度来看，对于那些很有可能会用到 Mac 或者类似 twm 这种环境的应用程序，最好还是要提供一个退出按钮 Quit。在本章的最后一节，会给出如何向窗口添加按钮的说明。

现在，我们已经准备好去看这一章最后一个完整的小程序了。这个例子会具有更多的自定义行为，还会有一个更复杂的布局，并且会完成一些更为复杂的处理，但它的基本结构仍旧会与计算程序 Calculate 很像，只不过确实添加了更多的 PyQt 对话框。

### 4.3 70 行的货币转换程序

货币转换程序是一个比较有用的小工具。由于兑换汇率时常变化，也就不能简单地像在前面章节中那样，在类 Length 的货币单位部分仅是简单地用一个静态字典来存储这些信息。幸运的是，加拿大银行 (Bank of Canada) 会提供一个存储了银行汇率的文件，该文件可通过网络获得，并且这个文件使用易于解析的文件格式。这些汇率通常与最新的数据可能会有几天的时差，但这些信息用于预判程序的错误或者是概算支付国际合同的大致数额还是足够了。该应用程序如图 4.4 所示。

这个应用程序必须先下载并解析这些汇率数据。然后，必须要创建一个用户界面，以便用户能够用来给定他们所感兴趣的货币种类和货币的数量。

依照惯例，还是先从这些导入模块看代码

```
import sys
import urllib2
from PyQt4.QtCore import *
from PyQt4.QtGui import *
```

无论是 Python 还是 PyQt 都提供了用于网络的类。在第 18 章中，会用到 PyQt 的类，不过在这 一章，仍将使用 Python 的 urllib2 模块，因为这个模块提供了一个方便从网上抓取文件的函数。

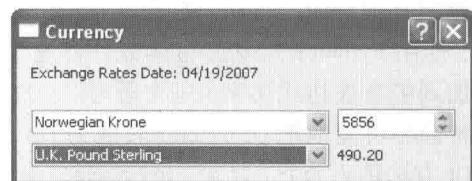


图 4.4 货币转换应用程序 Currency

```

class Form(QDialog):
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)
        date = self.getdata()
        rates = sorted(self.rates.keys())
        dateLabel = QLabel(date)
        self.fromComboBox = QComboBox()
        self.fromComboBox.addItems(rates)
        self.fromSpinBox = QDoubleSpinBox()
        self.fromSpinBox.setRange(0.01, 10000000.00)
        self.fromSpinBox.setValue(1.00)
        self.toComboBox = QComboBox()
        self.toComboBox.addItems(rates)
        self.toLabel = QLabel("1.00")

```

在使用 `super()` 初始化表单后，会调用 `getdata()` 方法。不久就会看到，这个方法会从网上下载读取汇率数据，把它们存储到 `self.rates` 字典里，并返回一个字符串，其中带有这些数据所使用的日期。字典的键值 `key` 是货币的币种，价值 `value` 是货币转换的系数。

为了能够在组合框 (combobox) 里按照顺序显示用户所需的货币类型，可以对字典中的键值 `key` 进行排序并以此作为键值 `key` 的新副本。对于 `date` 和 `rate` 变量和 `dateLabel` 标签，由于只在 `__init__()` 中用到，所以在类的实例中不必再保留对它们的引用。另一方面，由于确实需要能够访问组合框和 `toLabel`，所以可以使用 `self` 来对这些实例中的变量进行引用。

向这两个组合框中添加相同的排序表，再创建一个 `QDoubleSpinBox`，这是一个可以处理浮点型数据的微调框 (spinbox)。需要给这个微调框提供最大值和最小值，还需要设置其初始值。在设置微调框的值之前先设置其取值范围是一种很不错的做法，这是因为，如果先设置值的话，该值就有可能恰好不在微调框的初始范围内，这样该值就会被增加或减小以便能够适应初始范围。

由于在初始化时，这两个组合框都需要显示为相同的币种，且初始值 (value) 均为 1.00，因而在 `toLabel` 里显示的结果也会是 1.00，所以可以对其进行显式设置。

```

grid = QGridLayout()
grid.addWidget(dateLabel, 0, 0)
grid.addWidget(self.fromComboBox, 1, 0)
grid.addWidget(self.fromSpinBox, 1, 1)
grid.addWidget(self.toComboBox, 2, 0)
grid.addWidget(self.toLabel, 2, 1)
self.setLayout(grid)

```

在对这些窗口部件进行布局的时候，使用网格布局 (grid layout) 可能是最简单的方法了。在网格布局中每添加一个窗口部件，都会给定其所应占有的行和列的位置，两者均是从 0 开始。这里的网格布局的原理图如图 4.5 所示。使用网格布局还可以处理更多的事情。例如，可以对不同的行和列进行合并，等等，而这些会在后续的第 9 章中涉及。

dateLabel	(0, 0)	
self.fromComboBox	(1, 0)	self.fromSpinBox (1, 1)
self.toComboBox	(2, 0)	self.toLabel (2, 1)

图 4.5 货币转换程序 Currency 的网格布局

看一下运行效果截屏图，或者是直接运行该程序，很容易看出该网格布局的第 0 列要比第 1 列宽。然而，在现有代码里却找不到任何用以指定这一效果的代码，这是怎么发生的呢？实际上，布局非常灵活，会自动适应所处的环境，无论是有可用的空间还是它所管理的窗口部件中的内容和大小策略。在这种情况下，这些组合框会被水平拉长，以便使其能够足够宽地全部显示出最大文字宽度的货币值，而微调框也会被水平拉伸，以便能够将其最大值显示完全。由于这些组合框是 0 列中最宽的元素，它们的宽度值就实际成为了该列最小宽度的设置值；对于第 1 列中的微调框也是如此。如果运行程序并且把窗口变得更窄些，什么都不会变化，因为窗口实际上已经是其最小宽度了。不过，如果把窗口变得更宽的话，各列都会被拉长以占满那些多余的空间。当然，也可以对布局进行畸变，以便能够把更多的横向空间都给某一列，比如说，把多余的空间都给第 0 列。

没有哪个窗口部件是在初始化的时候就被纵向拉伸了，这是因为都没必要。不过，如果增加窗口的高度，那么多余的全部空间都会跑到 `dateLabel` 这里，因为在当前表单中，它是这里唯一一个可以在各个方向上增加大小的窗口部件，而且也没有其他任何窗口部件会约束它。

现在，既然这些窗口部件已经创建过，初始化过，也进行了布局，那么是设置表单行为的时候了。

```
self.connect(self.fromComboBox,
             SIGNAL("currentIndexChanged(int)"), self.updateUi)
self.connect(self.toComboBox,
             SIGNAL("currentIndexChanged(int)"), self.updateUi)
self.connect(self.fromSpinBox,
             SIGNAL("valueChanged(double)"), self.updateUi)
self.setWindowTitle("Currency")
```

如果用户修改了任意一个组合框的当前元素，相应的组合框就会发射一个 `currentIndexChanged(int)` 带有当前最新元素位置索引的信号。与之相类似的是，如果用户改变了微调框的值，就会发射一个 `valueChanged(double)` 带有最新值的信号。而所有这几个信号都连接到一个 Python 槽上：`updateUi()`。正如即将在下一节要看到的那样，并非所有情况下都必须要这么做，只不过对于这个应用程序而言，这样做是一种比较明智的选择罢了。

在 `__init__()` 的最后，会设置窗口的标题。

```
def updateUi(self):
    to = unicode(self.toComboBox.currentText())
    from_ = unicode(self.fromComboBox.currentText())
    amount = (self.rates[from_] / self.rates[to]) * \
              self.fromSpinBox.value()
    self.toLabel.setText("%0.2f" % amount)
```

为了响应各个组合框发射的 `currentIndexChanged()` 信号和微调框发射的 `valueChanged()` 信号，会调用这个 `updateUi()` 方法。相应的所有信号都会传入一个参数。正如将在下一节看到的，可以忽略信号的那些参数，就像在这里所做的那样。

无论触发的是哪个信号，都会用相同的处理方式。即先提取 `to` 和 `from` 的货币种类，再计算 `to` 的数值，最后设置 `toLabel` 的文本值。之所以给 `from` 文本的名字取做 `from_`，是因为 `from` 是 Python 的关键字而无法使用。在计算出数值后，为了能够让其所在行适应比较窄的页面，这里需要添加一个换行符；实际无论在何种情况下，最好都能限制行的宽度，以便让用户在屏幕上能够更方便地阅读相邻的两个文件。

```
def getdata(self): # Idea taken from the Python Cookbook
    self.rates = {}
    try:
        date = "Unknown"
        fh = urllib2.urlopen("http://www.bankofcanada.ca"
                             "/en/markets/csv/exchange_eng.csv")
        for line in fh:
            if not line or line.startswith("#", "Closing "):
                continue
            fields = line.split(",")
            if line.startswith("Date "):
                date = fields[-1]
            else:
                try:
                    value = float(fields[-1])
                    self.rates[unicode(fields[0])] = value
                except ValueError:
                    pass
        return "Exchange Rates Date: " + date
    except Exception, e:
        return "Failed to download:\n%s" % e
```

这个方法就是获取那些驱动应用程序数据的地方。一开始会创建一个新的 `self.rates` 实例属性 (instance attribute)。与 C++、Java 以及类似的其他语言不同, Python 允许在任何需要的情况下创造实例属性——例如, 在构造函数中, 在初始化函数中, 或者是在任何方法中。甚至可以在运行的时候, 向一些特殊的实例上添加不同属性。

由于在与网络连接时总是有太多出错的可能, 例如, 网络可能会瘫痪, 主机可能宕机, URL 可能改变, 等等, 因而需要让这个应用程序比之前的两个例子更为健壮。另外可能会遇到获取的是非法浮点数的情况, 如当前数据中包含的 NA(不适用, Not Available) 等。可以用内置的 `try...except` 块来捕获非法的数值。所以, 如果把当前币种转换为浮点数时失败, 那么只需忽略掉这一特殊币种并继续执行即可。

通过在 `try...except` 块中封装了几乎整个方法的处理方式, 几乎可以处理其他任何可能的情况(对于大多数应用程序来说, 这一做法就显得有点太过于普通了, 不过对于一个只有 70 行代码的应用程序来说尚可接受)。如果发生问题, 可以抛出异常, 并且将它以字符串的形式返回给调用者 `__init__()`。`getdata()` 返回的字符串会在 `dataLabel` 中显示, 所以通常这个标签会以适当形式显示所用转换利率的日期, 不过在有错误产生的时候, 它会显示错误信息而不是日期。

需要注意的是, 由于 URL 字符串实在是太长了, 所以将其分成两行两个字符串, 但并不需添加回车换行符。这么做之所以可行是因为这些字符串都在圆括号内。如果不是这种情况, 就应要么添加一个回车换行符, 要么用 + 号(并且依然还需要添加一个回车换行符)来连接它们。

初始化 `date` 变量时用了一个字符串, 以用来说明该用哪个日期来计算汇率。然后, 再用 `urllib2.urlopen()` 函数给出一个指向感兴趣文件的文件句柄。可以利用这个文件句柄及其 `read()` 方法一次读取整个文件, 不过在这个例子里面, 我们更推荐使用 `readlines()` 来一行一行地读取文件。

以下是从 `exchange_eng.csv` 文件中得到的某天的数据。这里忽略了一些列和大多数行中的数据, 而仅用省略号来予以表示。

```

...
#
Date (<m>/<d>/<year>),01/05/2007,...,01/12/2007,01/15/2007
Closing Can/US Exchange Rate,1.1725,...,1.1688,1.1667
U.S. Dollar (Noon),1.1755,...,1.1702,1.1681
Argentina Peso (Floating Rate),0.3797,...,0.3773,0.3767
Australian Dollar,0.9164,...,0.9157,0.9153
...
Vietnamese Dong,0.000073,...,0.000073,0.000073

```

`exchange_eng.csv` 文件中有几种不同的行格式。注释和某些空自行会用“#”作为开始，这里将其全部忽略。所有汇兑汇率都是通过汇率名称、汇率值并用逗号分隔的形式列示的。这些汇率是与特定的日期相对应的，而位于最后的日期也是最新的日期。还会有一行是以“Date”为开头的，其中的数值会分别作用于各列。当碰到这一行的时候，我们会选取最后的日期数据，因为这才是与我们正在使用的汇兑汇率相一致的日期。还有一行是以“Closing”开始的，忽略不管。

对于汇兑汇率的每一行，会向 `self.rates` 字典中插入一个项，用当期货币的名称作为键 `key`，并以汇率作为值 `value`。假设文件的编码方式是 7 位 ASCII 或者是 Unicode，但如果不是以上两种之一，可能会认为是编码错误。如果知道用的是何种编码类型，可以在调用 `unicode()` 的时候将其作为第二参数。

```

app = QApplication(sys.argv)
form = Form()
form.show()
app.exec_()

```

在创建 `QApplication` 对象，初始化 `Currency` 应用程序的表单，以及开始时间循环时，都使用了与前面例子完全一样的代码。

对于程序的结束，就像之前的例子那样，因为是通过对 `QDialog` 的子类化，如果用户单击关闭按钮 X 或者是按下 Esc 键，窗口都将会关闭，然后 PyQt 将会结束该应用程序。在第 6 章，还将会看到更多显式结束应用程序的方法，还会看到如何确保让用户有机会来保存那些未保存的改变和程序设置。

目前为止，使用 PyQt 进行 GUI 编程看起来还是比较简单易懂的。尽管随后还会看到一些更为复杂的布局，但从本质上来说，其实并不复杂，同时，因为布局管理器也比较灵活，基本可以处理大多数的情况。自然，就会涉及更多的内容，比如，创建应用程序主窗口的风格，创建用户可用于交互的弹出对话框，等等。然而，还是先从 PyQt 的一些基础开始吧，因为目前还未对信号和槽通信机制加以说明，这就是下一节的主题。

## 4.4 信号和槽

每个 GUI 库都提供了事件发生的不少细节，如鼠标点击、键盘按键等。例如，如果用户点击了一个写有“Click Me”的按钮后，按钮所附带的信息就会变成可用。GUI 库可以告知我们鼠标点击时与按钮的相对坐标，与按钮相应的父窗口部件，还有与屏幕相关的信息；GUI 库还会给出 Shift、Ctrl、Alt 以及 NumLock 键在当时的状态；也会给出按下和松开按钮的精确时间等。如果用户通过非鼠标点击的其他方式按下按钮，也应该能够提供相类似的信息。用户也有可能通过多次使用 Tab 键来把光标移动到按钮上，之后再按下空格，或者是用 Alt + C

等快捷键。尽管所有这些不同方法的结果一样，但按下按钮时这些方法中的每一种都会产生不同的事件和不同的信息。

Qt库曾经是最早认识到几乎在每一种情况下，程序开发人员并非需要甚至根本就不需要知道所有这些底层细节：他们并不关心按钮是如何按下的，他们只是想知道在按钮按下时能够适当响应即可。基于这一原因，Qt和PyQt提供了两种通信机制：低级事件处理机制（low-level event-handling mechanism）和高级机制（high-level mechanism），前者与所有其他GUI库提供的功能类似，后者被奇趣科技（Qt的提出者）称之为“信号和槽”。在第10章会讲述低级事件，第11章会再次讲到，但在这一节，我们仅会关注高级机制。

每个QObject——包括 PyQt 的全部窗口部件，因为它们都派生自 QWidget，这也是 QObject 的一个子类——都会支持信号和槽机制。特别是，它们都有声明状态转换的能力，比如当选中或者不选中复选框（checkbox）时，或者是其他重要事件发生的时候，例如按钮按下（无论是哪种按下方式）。PyQt 的所有窗口部件都有一系列的预定义信号。

无论信号何时发射，默认情况下，PyQt 都只是简单地将其扔掉！要截取一个信号，就必须将它连接到槽上去。在C++/Qt中，槽是必须用特殊语法声明的一些方法，不过在PyQt中，则可以是我们想调用的任何东西（比如，任意的函数或者方法），并且在定义的时候也不需要特殊的语法声明。

大部分的窗口部件也都提前预置了一些槽，所以有些时候可以直接把预置的信号与预置的槽相连接，无须做任何事就可以得到想要的行为效果。从这个方面来看，PyQt要比C++/Qt更为灵便些，因为在PyQt中可以连接的不仅只有槽，还可以是任何可被调用的对象，且从PyQt 4.2以后，还可以向Qobject中动态添加一些预定义的信号和槽。让我们通过图4.6所示的例子，信号和槽（Signals and Slots）程序，来看看信号和槽是如何实际工作的吧。

无论是QDial还是QSpinBox窗口部件都有valueChanged()信号，当这个信号触发时，就会带着新值。这两个窗口部件也都有setValue()槽，带有整数型参数值。因此，可以将这两个窗口部件的这两个信号和槽相互连接起来，无论用户改变了哪一个窗口部件，都会让另一个值做出相应的响应。

```
class Form(QDialog):
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)
        dial = QDial()
        dial.setNotchesVisible(True)
        spinbox = QSpinBox()

        layout = QHBoxLayout()
        layout.addWidget(dial)
        layout.addWidget(spinbox)
        self.setLayout(layout)

        self.connect(dial, SIGNAL("valueChanged(int)"),
                    spinbox.setValue)
        self.connect(spinbox, SIGNAL("valueChanged(int)"),
                    dial.setValue)
        self.setWindowTitle("Signals and Slots")
```

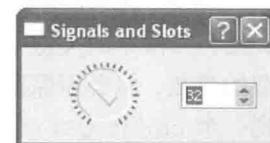


图4.6 信号和槽程序

由于这两个窗口部件是以这种方式连接的，所以如果用户拖动了拨号盘，比如说拨动值为20，此时拨号盘就会发射一个valueChanged(20)这样的信号，相应地，就会对微调框的

`setValue()`槽进行调用，并将 20 作为其参数。不过，在此之后，由于微调框的值被改变了，所以它也会发射一个 `valueChanged(20)` 信号，相应地，拨号盘的 `setValue()` 槽也会将 20 作为参数接受。这样看起来貌似会陷入一个无限的死循环。不过，如果传递的值并未真正改变，`valueChanged()` 就不会再次发射信号。这是因为，编写值变化型槽代码的标准方法就是在执行该方法之前会先检测目标值于当前值之间是否真的存在差异。如果值一直是相同的，那么就什么也不做直接返回；否则，才会执行变化，发射一个声明状态改变的信号。图 4.7 中描绘了这些相互之间的连接关系。

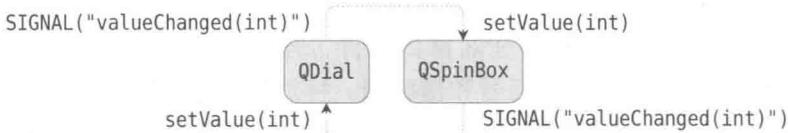


图 4.7 信号和槽的连接关系

现在来看一下连接信号和槽的一般语法形式。假设 PyQt 的各个模块已导入使用有 `from...import *` 的语法，并假定 `s` 和 `w` 都是 `QObject` 对象，一般情况下，这些对象都是窗口部件，通常会用 `self` 来代表 `s`。

```

s.connect(w, SIGNAL("signalSignature"), functionName)
s.connect(w, SIGNAL("signalSignature"), instance.methodName)
s.connect(w, SIGNAL("signalSignature"),
          instance, SLOT("slotSignature"))
    
```

这里的 `signalSignature` 是信号的名字，也是一个(也有可能是空的)用逗号分隔的参数类型的列表。如果该信号是一个 Qt 信号，那么这些类型名必须是 C++ 型类型，比如 `int`、`QString` 等。C++ 型类型名可能会相当复杂，对于每一个类型名都可能会含有一个或者多个 `const`、`*` 和 `&` 的关键字字符。在将这些字符作为信号(槽)名时，可以省略这些 `const` 型和 `&` 型的参数，但必须保留 `*` 型的参数。例如，几乎每一个传递 `QString` 参数的 Qt 信号，都会用到一个 `const QString &` 型参数，不过在 PyQt 中，仅是使用一个 `QString` 就足够了。另一方面，`QListWidget` 有个 `itemActivated(QListWidgetItem *)` 型信号，在编写代码时则必须准确。

PyQt 的各个信号在真正发射时就要定义过，可以带有任意数量和任意类型的参数，这一点稍后就可以看到。

`slotSignature` 与 `signalSignature` 拥有相同的形式，只是前者的名字是一个 Qt 槽。一个槽所拥有的参数数量可能并不比与其相连信号的参数数量多，实际上可能会更少；对于那些多余的参数则会被忽略掉。相互对应的信号和槽则必须具有相同的参数类型，因此可以举一个例子，就不能把 `QDial` 的 `valueChanged(int)` 信号连接到 `QLineEdit` 的 `setText(QString)` 槽上去。

在这个拨号盘和微调框的例子里，在本章之前给出的例子中用到过 `instance.methodName` 的语法。但当该槽实际是个 Qt 槽而不是 Python 方法时，用 `SLOT()` 语法可能就会更高效些

```

self.connect(dial, SIGNAL("valueChanged(int)"),
             spinbox, SLOT("setValue(int)"))
self.connect(spinbox, SIGNAL("valueChanged(int)"),
            dial, SLOT("setValue(int)"))
    
```

正如之前早就看到的那样，一个槽可以被多个信号连接。因而也有可能让一个信号与多个槽相连接。虽然这种情况比较少见，但却有可能会将一个信号与另一个信号相连接：在这些情况下，当第一个信号发射时，就会造成与之相连的信号也发射信号。

使用 `QObject.connect()` 可以建立各类连接，也可以用 `QObject.disconnect()` 来取消这些连接。在实际应用中，很少需要我们自己去取消连接，这是因为，比方说，当删除一个对象后，PyQt 就会自动断开与该对象相关联的所有连接。

迄今为止，我们已经看到了如何连接信号和写槽函数，这些槽就是一些常规的函数或者方法。也知道在说明状态转换或者其他重要事件发生时会发射信号。但如果想创建一个可以发射自定义信号的组件，该怎么办呢？使用 `QObject.emit()` 就可以轻松实现这一点。例如，以下就是一个完整的 `QSpinBox` 子类，它会发射自定义的 `atzero` 信号，还会传递一个数字：

```
class ZeroSpinBox(QSpinBox):
    zeros = 0

    def __init__(self, parent=None):
        super(ZeroSpinBox, self).__init__(parent)
        self.connect(self, SIGNAL("valueChanged(int)"), self.checkzero)

    def checkzero(self):
        if self.value() == 0:
            self.zeros += 1
            self.emit(SIGNAL("atzero"), self.zeros)
```

连接微调框自己的 `valueChanged()` 信号，使其调用这里的 `checkzero()` 槽。如果 `value` 的值刚好为 0，`checkzero()` 槽就会发射 `atzero` 信号，这样就可以计算出一共有多少次 0；像这样来传递一些额外的数据并非都是强制性的。对于信号来说，缺失圆括号非常重要：这会告诉 PyQt，这是一个“短路”信号。

无参数的信号（不带圆括号的信号）是一个短路 Python 的信号（short-circuit Python signal）。在这种信号发射的时候，可以向 `emit()` 方法传递任何额外参数，且这些参数会作为 Python 对象来进行传递。这样就可以避免对参数进行 C++ 类型相互转换时所带来的风险，但也就意味着，可以传递任意 Python 对象，即使这些 Python 对象无法在 C++ 数据类型之间相互转换也可以传递。带有至少一个参数的信号要么是 Qt 信号，要么是 Python 非短路信号（non-short-circuit Python signal）。在这些情况下，PyQt 会检查该信号是否是 Qt 信号，如果不是，就会假定是 Python 信号。无论何种情况，这些参数都会被转换成 C++ 数据类型。

下面就来看看该如何与表单 `__init__()` 方法中的信号相连接：

```
zerospinbox = ZeroSpinBox()
...
self.connect(zerospinbox, SIGNAL("atzero"), self.announce)
```

再说一遍，务必不要带圆括号，因为这是一个短路信号。基于完整性考虑，以下就是与表单相连接的槽：

```
def announce(self, zeros):
    print "ZeroSpinBox has been at zero %d times" % zeros
```

如果在 `SIGNAL()` 方法中使用了不带圆括号的标识符，那么就意味着给定的是一个如前所述的短路信号。使用这种语法既可以用来发射短路信号，也可以用来向其提供连接。例中这两种方法都用到了。

如果用带一个 `signalSignature`（可能是一个不带括号的、逗号分隔的 PyQt 类型列表）的 `SIGNAL()` 函数，可以给定一个 Python 信号或者 Qt 信号（Python 信号会以 Python 代码的形式

式发射；Qt 信号会以隐式C++ 对象的形式发射）。使用这种语法就可以发射 Python 或者 Qt 信号了，也可以连接它们。这些信号可以与任何可调用的函数或者方法相连接，也可以与 Qt 槽相连接；也可以使用带有 *slotSignature* 参数的 *SLOT()* 语法对其予以连接。PyQt 会查看该信号是否是 Qt 信号，如果不是，它会认为该信号是个 Python 信号。如果使用带有括号的指示符，即使是对 Python 信号，这些参数也会被转换成C++ 数据类型。

现在看另外一个例子，一个很小的非 GUI 自定义类，有一个信号和一个槽，这里所探讨的机制并不仅限于 GUI 类，任何 QObject 子类均可以使用这些信号和槽。

```
class TaxRate(QObject):
    def __init__(self):
        super(TaxRate, self).__init__()
        self.__rate = 17.5
    def rate(self):
        return self.__rate
    def setRate(self, rate):
        if rate != self.__rate:
            self.__rate = rate
            self.emit(SIGNAL("rateChanged"), self.__rate)
```

无论是 *rate()* 方法还是 *setRate()* 方法，都可以被连接，因为任何 Python 可调用对象都可以用做槽。如果汇率变动，就可以更新私有的 *\_\_rate* 的值，然后发射自定义的 *rateChanged* 信号，以便能够把新汇率当成参数。也可以使用更为快速的短路语法。如果打算使用标准语法，那么唯一的区别可能就是信号会被写成 *SIGNAL ("rateChanged (float)")*。如果将 *rateChanged* 信号与 *setRate()* 槽连接，因为有 *if* 语句，也就不会发生死循环的情况。回头看一下正在使用的这个类。首先，定义了一个可在汇率发生变动时调用的方法：

```
def rateChanged(value):
    print "TaxRate changed to %.2f%%" % value
```

现在，可以对其测试一下：

```
vat = TaxRate()
vat.connect(vat, SIGNAL("rateChanged"), rateChanged)
vat.setRate(17.5)      # No change will occur (new rate is the same)
vat.setRate(8.5)       # A change will occur (new rate is different)
```

这会导致在控制台里仅输出一行文字：“TaxRate changed to 8.50%”。

在之前的各个例子里，曾经将不同的信号连接在同一个槽上，也并不关心是谁发射了信号。不过有的时候，打算将两个或者更多的信号连接到同一个槽上，并需要根据连接的不同信号做出不同的反应。在本节的最后一个例子中，将会研究这个问题。

图 4.8 中给出的连接程序 Connections 有 5 个按钮和一个标签。当按下任意一个按钮时，就会用信号和槽机制来更新标签上的文本。这里给出的是用表单中的 *\_\_init\_\_()* 方法创建第一个按钮的代码：

```
button1 = QPushButton("One")
```

除了变量的名字和所需传送文本的不同之外，所有其他按钮的创建方式与此类似。



图 4.8 连接程序 Connections

就先从最简单的连接开始，这个连接是用在 button1 中的。以下给出的是 `__init__()` 方法的 `connect()` 调用方式：

```
self.connect(button1, SIGNAL("clicked()"), self.one)
```

对于这个按钮，使用了一种特殊的方法：

```
def one(self):
    self.label.setText("You clicked button 'One'")
```

将按钮的 `clicked()` 信号连接到单一的方法上，以便能够适当地响应该信号，这种做法可能是最常用的一种连接模式。

不过，既然大部分的处理过程都一样，那是否就意味着只需用一些参数就能够按下某个特定的按钮呢？在这些情况下，每个按钮最好都能连接到同一个槽上。实现这一做法可以用两种方法。第一种方法是用偏函数应用程序（partial function application）来封装带一个参数的槽，以便在调用参数按钮时能够调用该槽。另外一种做法是通过 PyQt 来了解是哪个按钮调用了该槽。这里会全部给出这两种方法，先从偏函数应用程序方法开始。

我们曾使用 Python 2.5 的 `functools.partial()` 函数或者是用自己简单的 `partial()` 函数创建过一个封装函数：

```
import sys

if sys.version_info[:2] < (2, 5):
    def partial(func, arg):
        def callme():
            return func(arg)
        return callme
else:
    from functools import partial
```

使用 `partial()` 可以把一个槽和一个按钮名字封装到一起。所以可以试着这样做：

```
self.connect(button2, SIGNAL("clicked()"),
            partial(self.anyButton, "Two")) # WRONG for PyQt 4.0-4.2
```

遗憾的是，在 PyQt 4.3 之前的版本中尚无法实现这一点。该封装函数是在 `connect()` 调用中创建的，不过在 `connect()` 一调用完，就会超出作用范围（go out of scope）并作为垃圾回收掉。从 PyQt 4.3 之后，如果连接时再像这样使用 `functools.partial()` 封装函数，那么就会对它们进行特殊处理。这就意味着，连接的函数就不会被垃圾回收，因而之前给出的代码就可以正确执行了。

对于 PyQt 4.0、4.1 和 4.2 这几个版本，依然可以使用 `partial()`：只需保持一个对封装函数的引用即可，除非在调用 `connect()`，否则不会使用这个引用，但实际上表单（form）实例的属性会确保在其存续期间该引用函数不会超出作用范围，从而也就可以正常工作。因此，该连接看起来可能会是这样的：

```
self.button2callback = partial(self.anyButton, "Two")
self.connect(button2, SIGNAL("clicked()"),
            self.button2callback)
```

当点击 `button2` 后，就会调用 `anyButton()` 方法，会带一个参数，其中含有文本字符串“Two”。该方法的代码看起来是这样的：

```
def anyButton(self, who):
    self.label.setText("You clicked button '%s'" % who)
```

本来也可以把这个槽像之前所给出的 `partial()` 函数那样应用到所有的按钮上。但事实上，

一点不用 `partial()` 函数也可以得到完全相同的结果：

```
self.button3callback = lambda who="Three": self.anyButton(who)
self.connect(button3, SIGNAL("clicked()"),
            self.button3callback)
```

这里会创建一个 `lambda` 函数，用按钮的名字来作为其参数。该函数也可以像 `partial()` 函数那样相同工作，调用同样的 `anyButton()` 方法，仅用 `lambda` 表达式来创建封装函数。

无论是 `button2callback()` 还是 `button3callback()` 都会调用 `anyButton()` 方法；两者的唯一区别是它们的参数不同，一个参数传递“Two”，另一个参数传递“Three”。

如果 PyQt 用的是 4.1.1 或者更高级的版本，就不必保留对自己的引用。这是因为，在连接中使用 `lambda` 表达式创建各个封装函数时，PyQt 会做特殊处理（这与 PyQt 4.3 中扩展 `functools.partial()` 的特殊处理方式一致）。基于这一原因，可以在各 `connect()` 调用中直接使用 `lambda` 表达式。例如：

```
self.connect(button3, SIGNAL("clicked()"),
            lambda who="Three": self.anyButton(who))
```

封装技术可以非常不错地工作，不过这里还会再稍微引入另外一种方法，但确实在某些时候会非常有用，特别是当我们不想封装自己的方法时。这种技术会用在 `button4` 和 `button5` 上。这里给出的是它们的连接：

```
self.connect(button4, SIGNAL("clicked()"), self.clicked)
self.connect(button5, SIGNAL("clicked()"), self.clicked)
```

需要注意的是，这里并没有对这两个按钮所连接的 `clicked()` 方法进行封装，所以第一眼看上去，好像无法区分到底是哪个按钮调用了 `clicked()` 方法<sup>①</sup>。然而，如果确实想要区分出来，只需仔细看下面的实现代码就可以了：

```
def clicked(self):
    button = self.sender()
    if button is None or not isinstance(button, QPushButton):
        return
    self.label.setText("You clicked button '%s'" % button.text())
```

在一个槽的内部，总是可以通过调用 `sender()` 发现到底调用信号是来自哪个 `QObject` 对象的（如果通过使用常规调用方法，调用的槽可能会返回 `None`）。尽管知道连接到这个槽上的仅仅是些按钮，但仍需小心为好。我们曾用过 `isinstance()`，但当时本可以使用 `hasattr(button, "text")` 来替代的。如果把所有的按钮都连接到这个槽上，应该也都是可以正确工作的。

有些编程人员不喜欢使用 `sender()`，因为觉得这不是面向对象的编程风格，所以在需要出现这种情况时，他们会更愿意使用偏函数应用程序的方法。

事实上，确实还有其他的封装技术也可获得对函数和参数的封装。这就是使用 `QSignalMapper` 类，会在第 9 章中给出一个使用该类的例子。

在某些情况下，槽会执行一定的操作，操作结果是调用该槽的信号，无论这种循环调用是直接调用还是间接调用，都会造成调用该槽的信号再次被重复调用，从而造成死循环。在实际应用中，这种循环模式一般较为少见。可以用两种方法降低这种循环发生的可能性。第一种

<sup>①</sup> 把槽和它所要连接的信号命名为相同的名字是 PyQt 编程的一种惯例。

方法是，只在信号真正发生改变的时候才发射该信号。例如，如果用户改变了一个 QSpinBox 的值，或者程序通过调用 `setValue()` 改变其值，那么只有当新数值与当前现有数值不同时，才会发射 `valueChanged()` 信号。第二种方法是，只有是用户的动作才会发射某些信号。例如，只有在用户更改了 QLineEdit 的文本时，它才会发射 `textEdited()` 信号，而如果是在代码中通过调用 `setText()` 更改了文本，则仍旧不会发射 `textEdited()` 信号。

如果看起来确实形成了一个信号-槽循环，通常情况下，要做的第一件事就是去检查代码的逻辑是否正确：是不是像我们想象的那样确实进行了处理？如果逻辑是正确的，也确实是出现了信号-槽循环，或许就需要通过修改那些与其连接的信号来打断这种循环链，例如，将那些程序发生改变就将发射的信号替换为用户出现交互后才发射的信号。如果问题依然存在，可以在代码的某些特定位置用 `QObject.blockSignals()` 来阻止这些信号，所有的 QWidget 类都继承有 `QObject.blockSignals()`，它会传递一个布尔值——`True`，对象停止发射信号——`False`，对象恢复信号发射能力。

到此为止，就完成了有关信号和槽机制的内容。在本书剩余的部分还会给出更多实践中各式各样的信号和槽方面的例子。大部分的 GUI 库都已经以某种或者不同的方式复制了这一机制。这是因为信号和槽机制非常实用，也非常强大，从而使得编程人员能够更灵活地关注于应用程序的逻辑顺序，而不必再依靠自己来考虑针对用户某一具体操作时所需考虑的那些细节了。

## 小结

在这一章中，我们看到是如何创建那些控制台—GUI 混合应用程序的。这当然还可以做得更深一些——例如，如果安装了 PyQt，就可以在一个 `if` 代码块作用域内包含并执行所有的 GUI 代码。这样就可以创建一个 GUI 应用程序，对于那些没有安装 PyQt 的用户，该程序就可以重新回退到“控制台模式”。

我们也看到了，GUI 应用程序不同于普通的批处理程序，会有一个一直在运行的事件循环，检查诸如鼠标点击、键盘按下等用户事件和诸如计时器超时、窗口重绘等系统事件，并且可以在需要程序终止的时候结束程序。

Calculate 应用程序向我们展示了一个非常简单但又非常程序化的典型对话框的 `__init__()` 方法。创建了一些窗口部件，对其进行布局，再创建连接，然后又使用一种或者多种其他方法来响应用户的交互。货币转换应用程序 Currency 使用了相同的方法，只不过是有着更为复杂的用户界面，以及更为复杂的行为和处理方法。货币转换应用程序 Currency 也说明，可以不必受限于常规做法，可以把多个信号连接到同一个槽上。

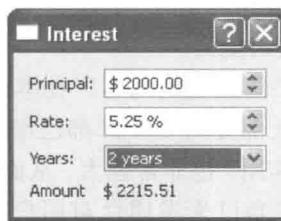
PyQt 的信号和槽机制允许我们用比鼠标点击和键盘按下具体细节更高级的抽象方法来处理用户的交互。这就允许我们可以把关注点放到用户想做什么，而不是关注于该怎么处理他们想做什么。所有的 PyQt 窗口部件都会通过发射信号的方式来声明状态的改变和其他重要事件的发生；大部分时候，这些信号都可以忽略不管。但对于感兴趣的那些信号，就可以方便地使用 `QObject.connect()` 来确保在信号发射的时候，确实调用了打算调用的函数或者方法，以便能够响应这些信号。不像 C++/Qt，必须使用特定的语法声明各个槽，在 PyQt 中，任何可调用的对象，也就是说，任何的函数或者是方法，都可以成为一个槽。

我们还看到了如何将多个信号连接到同一个槽上，以及如何使用偏函数应用程序（partial function application）或者 `sender()` 方法，以便槽能够区分是哪个窗口部件向它发射了信号并对其进行适当地进行响应。

我们还学到了，对于自定义的那些信号，并非一定要声明：可以简单地使用带有欲传递参数的 `QObject.emit()` 来发射它们。

## 练习题

编写一个计算复利<sup>①</sup>（compound interest）的对话框型应用程序。该程序应当与货币转换应用程序 `Currency` 的风格和结构类似，看起来应当与下图类似：



本利之和（amount）应当会在用户修改任何一个变量因子的时候就重新计算，这些变量因子是指：本金（principal）、利率（rate）、年限（years）。年限组合框应当显示可以“1 年期”、“2 年期”、“3 年期”等文本，因而年限值应当是该组合框当前索引值加 1，即 `index + 1`。Python 中复利的计算公式为 `amount = principal * ((1 + (rate / 100.0)) ** years)`。`QDoubleSpinBox` 类两个方法：`setPrefix()` 和 `setSuffix()`，可以分别用来设置“\$”和“%”符号。整个应用程序应当在 60 行之内完成。

提示：可以通过把微调框和组合框的适当信号连接到 `updateUi()` 槽上来实现自动更新操作，具体的计算和本利之和标签的更新可以放到 `updateUi()` 槽中。

本练习题的参考答案在文件 `chap04/interest.pyw` 中。

<sup>①</sup> 复利是指在每经过一个计息期后，都要将所产生的利息加入到本金中，由其再计算下期的利息。这样，在每一个计息期，上一个计息期的利息都将成为生息的本金，即以利生利，俗称“利滚利”——译者注。

## 第5章 对话框

- 简易对话框
- 标准对话框
- 智能对话框

基本上任何一个 GUI 应用程序都有至少一个对话框，且大多数 GUI 应用程序主窗口又都会带有一个或者多个对话框。对话框可以用来声明比放在状态栏或者日志文件里更为重要的那些消息。在这些情况下，这些对话框通常只有一个放置文字的标签和一个在用户阅读文字后按下的 OK 按钮。绝大多数情况下，对话框都用来对用户进行提问。一些对话框则比较简单，只需用户回答一个 yes 或者 no。而其他一些对话框则会让用户做出其他类型的选择——例如，用户打算使用的是哪个文件、文件夹、颜色或者字体。PyQt 对这些东西都提供了内置的对话框。

这一章将关注自定义对话框的创建，以便能够在内置对话框不合适的时候还可以询问用户的需求和喜好。

有个没有说明的问题是，针对特定目的该如何选用哪个窗口部件。例如，当我们希望用户对 3 个选项做出选择时，或许会提供 3 个单选按钮，或者是提供有 3 个项的列表型窗口部件或是一个组合框。当然也可能会是一个三状态复选框。但这些并非是全部。对于 GUI 编程的新手来说，附录 B 选择了一些 PyQt 窗口部件并给出了它们的截屏图和简要描述，或许在做决定时会有所帮助。

Qt 提供了 Qt 设计师 (Qt Designer) 这个可视化设计工具，从而可以轻松地“绘制”出不同类型的对话框，无须编写窗口部件创建和布局的任何代码。也可以用 Qt 设计师来设置对话框的某些行为。随后会在第 7 章介绍 Qt 设计师。在本章，将仅用代码创建所有的对话框。有些程序开发人员会用代码创建其所有对话框，而有些则可能会更喜欢用 Qt 设计师。这两种方法在本书中都会讲到，以便选择更喜欢的编程方式。

划分对话框的一种方式是通过它们的“智能”程度，可根据应用程序写入对话框中数据的多少将其划分为“简易型”(dumb)、“标准型”(standard) 和“智能型”(smart)。这些划分标准会影响到对话框的实现方式和实例化方法(即创建其实例)，因而在本章中也会分为不同小节来予以介绍。在每一节的开始，会给出该标准分类的说明，并且通过具体的例子来解释这一分类的优势和不足。

除按照智能程度划分外，对话框还可以根据模态(modality)进行划分。所谓的应用程序模态对话框(application modal dialog)是指，一旦调用该对话框，它就会成为应用程序唯一能够与用户进行交互的部件。在关闭该对话框之前，用户都不能使用应用程序的其他部件。当然，用户还可以自由使用其他的应用程序，比如，通过单击其他应用程序而使其获得光标。

窗口模态对话框(window modal dialog)与应用程序模态对话框(application modal dialog)的工作方式相似，除了它会阻止与其父窗口、父窗口的父窗口并直至顶层窗口等的交互，当然也会阻止与父窗口同层各兄弟窗口的交互。对于只有一个顶层窗口的那些应用程序来说，在实践上不会存在应用程序模态和窗口模态的区别。在引用“模态”窗口的时候，并不需要说明是哪种类型，通常会认为是窗口模态。

与模态对话框相对应的是非模态对话框(modeless dialog)。当调用非模态对话框的时候，用户可以与该对话框以及应用程序的其他部分交互。这会对如何设计代码产生影响，因为用户可能既从主窗口又从非模态对话框两边同时修改程序的状态，这就可能造成两者相互影响。

对话框编码时的另一个重要问题是进行验证。无论在哪里，都要尽可能通过选择合适的窗口部件并设置这些窗口部件的相关属性来避免自己手写验证代码。例如，如果需要的是整数，就可以选择 QSpinBox 并使用它的 setRange() 方法，以便能够把数值设置在所需的范围内。对用于单个窗口部件的验证可称为“窗口部件级”(widget-level) 验证；而数据库编程人员也会将这种验证称为“域级”(field-level) 验证。有的时候，可能还需要比窗口部件级更高级的验证，尤其是当存在依赖关系的时候。例如，电影院订票系统中可能会有两个组合框，一个用来选择楼层，另一个用于座位所在的行。如果一楼的座位行数为 A-R，二楼的座位行数为 M-T，那么在这种情况下，就只有某些特定的楼层和行数的组合才是有效的。对于这些情况，就必须执行“窗体级”(form-level) 验证；数据库编程人员通常称之为“记录级”(record-level) 验证。

有关验证的另一个要注意的问题是应该在何时进行验证。理想情况下，最好是能够让用户根本就不能输入那些不正确的数据，但有时候确实可以通过一些技巧来实现。可以把验证分为两类：一是“提交后验证”(post-mortem)型，会在用户提交设置数据的时候进行验证；二是“预防式验证”(preventative)型，会在用户编辑窗口部件的时候就进行验证。

利用上述 3 种模态和多种认证策略，就可以为对话框提供不同等级的智能性，从而可以提供出很多种的选择组合。在实际应用中，所选用的组合类型每次基本都差不太多。例如，大多数情况下，或许就会把简易型和标准型对话框做成模态，而把智能型对话框做成非模态。至于验证，就需要根据具体情况来选择合适的验证方案了。本章会给出一些非常常用的案例，而在本书的剩余章节还会给出更高级的各类对话框。

## 5.1 简易对话框

所谓的“简易”对话框(dumb dialog)是指，对话框的调用者会把对话框中的各窗口部件全部设置为初始值，也可由对话框调用者直接获取各窗口部件的最终值。简易对话框不掌握各窗口部件中用于编辑和显示的数据。可以对简易对话框的各窗口部件使用一些基本的验证，不过对一些相互依赖的窗口部件进行验证并不常见(或者说，通常可能不行)；换言之，在简易对话框中通常不会做窗体级验证。简易对话框通常是模态对话框，有一个“接受”按钮(比如 OK)和一个“放弃”(Cancel)按钮。

使用简易对话框最主要的优点是借助 API 后，就无须为其再编写任何代码了，也无须为其他逻辑关系编写任何代码。之所以有这些好处，是因为简易对话框的各个窗口部件都可直接获得。而最主要的不足是使用简易对话框的代码需要与其用户界面相关联(因为是直接访问各窗口部件的)，所以就不大容易实现复杂的验证方案——且如果需要多次应用这个对话框，简易对话框就不如标准对话框或者智能对话框那么方便了。

先从一个具体的例子开始。假设有一个图形应用程序，打算让用户可以设置画笔的属性，例如，画笔的笔尖宽度、样式以及是否对所绘线段进行圆整等。图 5.1 给出了打算获得的效果。

在这个例子中，我们不需要“实时”或者交互式地更新画笔的属性，因而一个模态对话框就足够了。由于所需的验证比较简单，所以在本例中就可以用一个简易对话框。

这样就可以把这个模态对话框的弹出命令放在一个槽上，再把这个槽连接到菜单项、工具栏或者是某个对话框的按钮上。如果用户点击了 OK 按钮，就更新画笔的属性；如果用户点击了 Cancel 按钮，就什么也不做。槽调用看起来会是下面这个样子：

```
def setPenProperties(self):
    dialog = PenPropertiesDlg(self)
    dialog.widthSpinBox.setValue(self.width)
    dialog.beveledCheckBox.setChecked(self.beveled)
    dialog.styleComboBox.setCurrentIndex(
        dialog.styleComboBox.findText(self.style))
    if dialog.exec_():
        self.width = dialog.widthSpinBox.value()
        self.beveled = dialog.beveledCheckBox.isChecked()
        self.style = unicode(dialog.styleComboBox.currentText())
        self.updateData()
```

先从创建 PenPropertiesDlg 对话框开始，稍后会看到对话框的创建细节；现在只需知道的是，该对话框有一个宽度微调框 width、一个圆角复选框 beveled 和一个风格组合框 style。向对话框传递一个 self(窗体调用)作为对话框的父对象，以便能够利用 PyQt 会默认把对话框放在其父对象中间上方的优点，这是因为，对于具有父对象的对话框，系统不会再在任务栏上为其单独添加一个图标。然后就可以直接使用其中的窗口部件了，将其值设置成调用窗体所传递的值。QComboBox.findText()方法可以返回对应文本的下标值。

当对一个对话框调用 exec\_()方法时，该对话框就会以模态形式显示出来。这就意味着，对话框的父窗口以及其兄弟窗口都会被阻塞，直到关闭这个对话框为止。只有当用户关闭对话框(无论是通过点击 Accept 对话框还是通过点击 Rejecting 对话框)，exec\_()的调用才会返回。如果用户点击的是 Accept 对话框，其返回表达式的返回值就会是 True；否则，则返回 False。如果用户点击的是 Accept 对话框，可知用户想使其设置生效，所以就从对话框的窗口部件读取数据并更新应用程序的数据。在最后部分的 updateData() 调用只是一个自定义方法，会在主窗口中显示出画笔的新属性。

在 setPenProperties() 方法的最后，PenPropertiesDlg 会超出作用域范围而成为垃圾回收的对象。所以，无论何时调用 setPenProperties() 方法，都需要先创建一个新的对话框并更新其窗口部件。这种方法可以节省内存，但代价是会迟滞一些速度。诸如此类的小对话框，速度上的这点儿损失非常小，用户也不会察觉到，但在随后我们会给出另一种替代方法，以避免每次都要新建和销毁对话框。

使用简易对话框意味着对话框与应用程序的耦合度很低。通过将窗口部件的各标签全部做成变量则可以完全解耦。然后，只需要简单改变标签的值，就可以用 PenPropertiesDlg 编辑微调框、复选框和组合框等任意类型的数据了。例如，通过读取“Temperature”微调框、“Is raining”复选框和“Cloud cover”组合框的值，就可以用来记录天气了。

看完了如何使用对话框，接下来再看看如何实现这个对话框的代码。PenProperties-Dlg 对话框只有一个 \_\_init\_\_() 方法，会通过几个部分来进行查看。

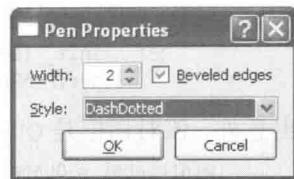


图 5.1 画笔属性对话框

```

class PenPropertiesDlg(QDialog):
    def __init__(self, parent=None):
        super(PenPropertiesDlg, self).__init__(parent)

        widthLabel = QLabel("&Width:")
        self.widthSpinBox = QSpinBox()
        widthLabel.setBuddy(self.widthSpinBox)
        self.widthSpinBox.setAlignment(Qt.AlignRight|Qt.AlignVCenter)
        self.widthSpinBox.setRange(0, 24)
        self.beveledCheckBox = QCheckBox("&Beveled edges")
        styleLabel = QLabel("&Style:")
        self.styleComboBox = QComboBox()
        styleLabel.setBuddy(self.styleComboBox)
        self.styleComboBox.addItem("Solid")
        self.styleComboBox.addItem("Dashed")
        self.styleComboBox.addItem("Dotted")
        self.styleComboBox.addItem("DashDotted")
        self.styleComboBox.addItem("DashDotDotted")
        okButton = QPushButton("&OK")
        cancelButton = QPushButton("Cancel")

```

对于每个可编辑窗口部件，都会创建一个与之对应的标签，以便可以告诉用户他们正在编辑的是哪个属性。当在标签中放入一个与字符(&)时，会产生两种含义。一是这个符号仅仅就只是个字面上的与字符(&)。另一种意思是，这个与字符(&)不会被显示出来，不过紧跟其后的字符会显示为下画线的形式，用以说明这是一个键盘加速键(keyboard accelerator)。例如，在本例的widthLabel中，它的文本是“&Width:”，实际会显示为“Width:”，而其加速键就是Alt+W。在Mac Os X操作系统上，默认会忽略这个加速键行为；因此，PyQt 就不会在这种系统上显示下画线。

字面上与字符(&)与加速键与字符(&)的区别在于，加速键版的与字符(&)的标签上会有一个“伙伴”(buddy)：如果有伙伴，与字符(&)就表示加速键。所谓伙伴，就是在按下加速键时 PyQt 会把键盘光标移到的相应窗口部件。所以，当用户按下Alt+W的时候，键盘的光标会切换到widthSpinBox上去。这一切换意味着用户在按动上、下键或者PageUp、PageDown键时，就会影响widthSpinBox的值，因为该窗口部件拥有光标焦点。

在有多个按钮的情况下，按钮文字上的下画线可用来说明这是一个加速键。所以，在这个例子中，okButton的文字是“&OK”，但会显示成OK，而用户就可以通过鼠标、Tab键、加速键Alt+O等方式按下这个按钮。通常情况下，不需要给Cancel(或者Close)按钮设置加速键，因为它们通常都会与对话框的reject()槽连接在一起，并且QDialog为这种情况专门提供了Esc快捷键<sup>①</sup>。复选框(CheckBox)和单选按钮(Radio Button)与那些文本字符中带有加速键的按钮类似。例如，由于复选框beveled有个带下画线的“B”，用户就可以通过按下加速键Alt+B来切换复选框的选中与否。

不过像这样创建OK和Cancel按钮有个不好之处就是在对它们进行布局的时候，就需要确定一个特定的顺序。例如，可能会把OK放在Cancel的左边。而在某些窗口系统中，这样的顺序就是错误的。PyQt为此提供了一个解决方案，会在对话框按钮布局一栏中讲到。

<sup>①</sup> 使用术语“键盘加速键”(keyboard accelerator)和“加速键”(accelerator)，即“Alt + 字母”形式的按键序列，可用来在不同对话框中点击按钮和切换光标，也可用来弹出菜单栏。而对于其他形式的按键序列则用术语“键盘快捷键”(keyboard shortcut)来表示，例如，按键序列Ctrl+S通常用做文件的保存。在第6章会看到如何创建键盘快捷键。

## 对话框按钮布局

在之前的一些例子中，我们曾经把按钮放到了对话框的右侧，先是 OK 按钮，然后紧接着才是 Cancel 按钮。在 Windows 中这是最常见的布局方式，但这也并非总是正确的。例如，对于 Mac OS X 或者 GNOME 桌面环境的用户来说，他们会将两个按钮的位置进行互换。

如果打算让应用程序能够具有最本地化的观感并期望将其部署到不同的系统平台，诸如按钮次序和位置这样的问题就会让我们非常恼火。Qt 4.2 ( PyQt 4.1 ) 对这一特殊问题提供了一种解决方案：使用 QDialogButtonBox 类。

不是直接创建 OK 和 Cancel 按钮，而是创建一个 QDialogButtonBox。比如这样：

```
buttonBox = QDialogButtonBox(QDialogButtonBox.Ok |  
                           QDialogButtonBox.Cancel)
```

让某个按钮作为“默认”按钮，也就是说，该按钮会在用户按下回车键 Enter 的时候被按下（假设拥有键盘光标的窗口部件不会自行处理回车键 Enter 的键盘按下事件），就可以这样做：

```
buttonBox.button(QDialogButtonBox.Ok).setDefault(True)
```

由于每个按钮盒子<sup>①</sup>( button box )都是一个单独的窗口部件（尽管它可以包含其他窗口部件），就可以直接将其加到对话框中已有的布局中，而不是把它放到自己的布局中再嵌入到对话框的布局中。这里给出的是 PenPropertiesDlg 例子中网格布局可以用到的代码：

```
layout.addWidget(buttonBox, 3, 0, 1, 3)
```

然后，不要连接按钮的 clicked() 信号，而是从该按钮盒子创建各个连接，这样就可以用它自己的信号来响应用户的不同动作：

```
self.connect(buttonBox, SIGNAL("accepted()"),  
            self, SLOT("accept()"))  
self.connect(buttonBox, SIGNAL("rejected()"),  
            self, SLOT("reject()"))
```

尽管仍可以自由连接自己按钮上的各个 clicked() 信号，并且通常对于具有很多按钮的对话框来说，也的确就是这么做的。

QDialogButtonBox 默认会使用水平布局，但也可以通过向其构造函数传递 Qt.Vertical 或者调用 setOrientation() 而形成垂直布局。

多数例子中会使用 QDialogButtonBox，但实际上，如果认为后台的兼容性是个问题时，就应该用 QPushButton 替换它。

我们已将微调框的数字设置为右对齐、垂直居中且有效范围为 0 ~ 24。在 PyQt 中，是允许存在一个宽度为 0 的画笔（比如，线的宽度）的，表示 1 个像素宽，无论对其应用过何种数学变换。对于宽度为 1 或者以上数值的画笔，会按照给定的宽度进行绘制，并会根据相应的数学变换，如比例变形，进行相应的变化。

通过使用微调框并为其设置取值范围，就避免了使用时输入错误而造成画笔宽度错误取值的可能，例如，编辑线的宽度时。很多时候，只需通过选择合适的窗口部件并且为其设置合适的属性值就可以提供所需的全部窗口部件级验证。在使用复选框 beveled 时就可以看

<sup>①</sup> 就是由数个按钮通过组合方式所形成的盒状布局体——译者注。

到：使用画笔绘制的线段要么带圆角，要么不带圆角。同理，在使用线条样式组合框 combobox 时也会用到相同的验证方法——用户只能选择有效的样式，即从所提供的样式表中选择样式。

```
buttonLayout = QHBoxLayout()
buttonLayout.addStretch()
buttonLayout.addWidget(okButton)
buttonLayout.addWidget(cancelButton)
layout = QGridLayout()
layout.addWidget(widthLabel, 0, 0)
layout.addWidget(self.widthSpinBox, 0, 1)
layout.addWidget(self.beveledCheckBox, 0, 2)
layout.addWidget(styleLabel, 1, 0)
layout.addWidget(self.styleComboBox, 1, 1, 1, 2)
layout.addLayout(buttonLayout, 2, 0, 1, 3)
self.setLayout(layout)
```

为了得到想要的布局效果，我们使用过两个布局，一个嵌在另一个里面。先从带有“拉伸”(stretch)的各按钮水平布局开始。拉伸会占用掉尽可能多的空白区域，这样就可以将两个按钮尽可能放在右边，并且空间上依然合适。

在网格布局(grid layout)的三列中，依次放置着宽度标签、宽度微调框和圆角复选框。风格标签和风格组合框放在下一行，且风格组合框占据着合并过的两列。QGridLayout.addWidget()方法的各个参数是窗口部件、行、列，然后是可选项，包括要合并的行的数目，然后跟着是要合并的列的数目。把上述按钮布局作为第三行添加到网格布局中，并让它合并占用所有三列的空间。最后，把整个布局放到对话框上。图 5.2 给出了整个布局的示意图，其中，网格布局用灰色显示。

```
self.connect(okButton, SIGNAL("clicked()"),
            self, SLOT("accept()"))
self.connect(cancelButton, SIGNAL("clicked()"),
            self, SLOT("reject()"))
self.setWindowTitle("Pen Properties")
```

在 \_\_init\_\_() 的最后会创建一些必要的连接。将 OK 按钮的 clicked() 信号连接到对话框的 accept() 槽上：这个槽将会关闭对话框并返回一个 True 值。Cancel 按钮也用相应的方法进行连接。最后，设置一下窗口的标题。

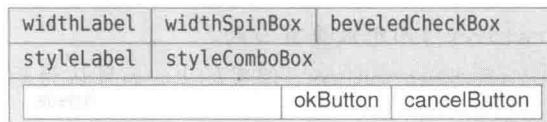


图 5.2 画笔属性对话框的布局

对于那些仅从一个地方调用的小型简易对话框来说，完全有可能无须创建任何对话框类。相反，只需要在调用方法中对这些窗口部件进行创建、布局、连接和 exec\_() 调用。如果 exec\_() 返回 True，就可以从各窗口部件提取数据，然后结束即可。在文件 chap05/pen.pyw 中，给出了画笔属性对话框和一个只有两个按钮的简单程序，其中一个按钮用来调用之前看到的 PenPropertiesDlg 对话框，另一个按钮则仅做些简单的内联事物。创建内联对话框的方法并不值得推荐，所以不会在此查看这些代码，但考虑到完整性，会在 setPenInline() 方法中提到并给出这些代码。

简易对话框较为容易理解和使用，但不推荐使用对话框的窗口部件来设置和获取值，除非

是非常简单的对话框，比如只有一个、两个或者至多只有数个值而已。先给出这些简易对话框的介绍，是为了能够将其作为对话框的简单入门，因为它们的创建、布局以及窗口部件的连接，对于任何对话框来说都是一样的。在下一节，将会介绍标准对话框，既包括模态标准对话框又包括非模态标准对话框。

表 5.1 布局方法摘要

语法	说明
b.setLayout(l)	把 QLayout l 添加到 QVBoxLayout b 中，b 通常是 QHBoxLayout 或者 QVBoxLayout
b.addSpacing(i)	把固定大小为 int i 的 QSpacerItem 条件到布局 b 中
b.addStretch(i)	用最小大小 0 和伸展因子为 int i 的 QSpacerItem 添加到布局 b 中
b.addWidget(w)	向布局 b 中添加一个 QWidget w
b.setStretchFactor(x, i)	把布局 b 中的布局或窗口部件的伸展因子由 x 设置为 int i
g.setLayout(l, r, c)	把 QLayout l 添加到 QGridLayout g 的第 int r 行和第 int c 列；可以额外给定要合并的行数和列数
g.addWidget(w, r, c)	把 QWidget w 添加到 QGridLayout g 的第 int r 行和第 int c 列；可以额外给定要合并的行数和列数
g.setRowStretch(r, i)	把 QGridLayout g 的行 r 拉伸到 int i
g.setColumnStretch(c, i)	把 QGridLayout g 的列 c 拉伸到 int i

## 5.2 标准对话框

如果一个对话框会根据自己的初始化程序或者方法所给定的设置值来初始化各个窗口部件，且各最终值是由调用的方法或者其实例中的变量决定而不是直接通过对对话框的窗口部件来决定的对话框，即可认为是“标准”的对话框。一个标准对话框 (standard dialog) 可同时拥有窗口部件级和窗体级的验证功能。标准对话框既可以是含有“接受”(accept) 和“拒绝”(reject) 按钮的模态对话框，又可以是含有“应用”(apply) 和“关闭”(close) 按钮并可通过信号和槽连接机制通知状态改变的非模态对话框(相对不太常用)。

使用标准对话框的重要好处之一是，调用程序无须知道标准对话框的实现方法，而只需知道该如何设置其初值及在用户点击 OK 按钮时该如何获得结果值。另一个优点，或者说至少对模态对话框是这样的，即由于用户不能与对话框的父窗口和父窗口的兄弟窗口进行交互，可以确保应用程序相关部分的状态不会在对话框应用之前先发生改变。使用标准对话框的最主要缺点是，在必须处理大量不同数据元素时，由于每个数据元素都需要依次提供给对话框且需要确保每次输入都能获得相应地结果输出，就可能需要用到大量的代码。

如上一节一样，我们还是会通过一个例子来进行讲解。在此情况下，这个例子会同时用到这一节和下一节，以便能够更为清晰地看出在处理标准对话框和智能对话框 (smart dialog) 时所用处理方法和折中技术的不同之处。

假设有一个需要通过表格形式显示一些浮点型数据的应用程序，并希望用户能够具有适当控制浮点数格式的能力。实现这一要求的一种方法是，提供菜单项、工具栏按钮或者提供一个可用来调用浮点数格式设置对话框的快捷键。图 5.3 给出的就是一个显示在许多数字上的浮点数格式设置对话框。

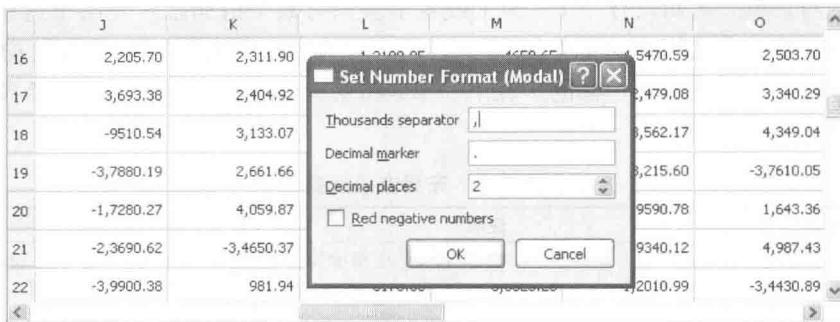


图 5.3 应用中的浮点数格式设置对话框

对于希望用户可以控制的那些数据会放在主窗体的字典中。以下给出了该字典的初始化过程：

```
self.format = dict(thousandsseparator=",",
                   decimalmarker=".",
                   decimalplaces=2,
                   rednegatives=False)
```

像这样使用字典会很方便，并且在添加其他额外数据元素时也会很容易。

在 `numberformatdlg1.py` 文件中可以找到这个数据格式设置对话框，用做数据导入的应用程序放在 `numbers.pyw` 文件中。其中，文件名中的数字“1”是为了与下一节中本对话框的另外两个版本进行区分。

### OK/Cancel 型模态对话框

先从对话框的使用入手；在处理用户的某些操作时，假定调用的是 `setNumberFormat1()` 方法。

```
def setNumberFormat1(self):
    dialog = numberformatdlg1.NumberFormatDlg(self.format, self)
    if dialog.exec_():
        self.format = dialog.numberFormat()
        self.refreshTable()
```

一开始会先创建一个对话框并会为其传递格式字典，而对话框也会根据格式字典自行初始化，这样就将对话框与调用它的窗体绑定到了一起——在其中央上方显示，并且不会在操作系统的任务栏上显示出新的图标。

正如之前提到过的，调用 `exec_()` 会以模态对话框的形式弹出该对话框，因此用户在与对话框的父对象和兄弟对象交互之前，就只能要么选择接受 (`accept`)，要么选择拒绝 (`reject`)。在下一节，我们将会看到使用了非模态形式的对话框，也就不会再强加这样的限制了。

如果用户点击 `OK` 按钮，那么就用对话框中的各个变量值设置格式字典，并且对表格进行更新，以便表格中的浮点数能够显示为新的格式。如果用户取消操作，那就什么都不做。在该方法的最后，对话框会超出作用域范围，因而会进入垃圾回收进程中。

为节省空间并避免无谓的重复，从现在开始，代码中将不会再给出任何的 `import` 导入声明语句，除非在出现时含义不够明显。所以，这个例子中，就不会再给出 `from PyQt4.QtCore import *` 或者 `PyQt4.QtGui` 之类的导入语句了。

现在，来看看对话框是如何实现的吧。

```
class NumberFormatDlg(QDialog):
    def __init__(self, format, parent=None):
        super(NumberFormatDlg, self).__init__(parent)
```

在 `init()` 方法的开始部分，是与之前所看到的其他对话框都一样的内容。

```
thousandsLabel = QLabel("&Thousands separator")
self.thousandsEdit = QLineEdit(format["thousandsseparator"])
thousandsLabel.setBuddy(self.thousandsEdit)
decimalMarkerLabel = QLabel("Decimal &marker")
self.decimalMarkerEdit = QLineEdit(format["decimalmarker"])
decimalMarkerLabel.setBuddy(self.decimalMarkerEdit)
decimalPlacesLabel = QLabel("&Decimal places")
self.decimalPlacesSpinBox = QSpinBox()
decimalPlacesLabel.setBuddy(self.decimalPlacesSpinBox)
self.decimalPlacesSpinBox.setRange(0, 6)
self.decimalPlacesSpinBox.setValue(format["decimalplaces"])
self.redNegativesCheckBox = QCheckBox("&Red negative numbers")
self.redNegativesCheckBox.setChecked(format["rednegatives"])

buttonBox = QDialogButtonBox(QDialogButtonBox.Ok |
                             QDialogButtonBox.Cancel)
```

对于任何希望用户能够改变的格式都会创建一个标签，以便用户能够知道它们要使用哪个编辑窗口部件，以及正在修改的是哪一个属性。由于 `format` 参数是随机的，需假设它可以包含全部变量值，故而可用它来初始化那些编辑窗口部件。还会使用到 `setBuddy()`，以便能够为那些无法使用鼠标的键盘用户提供支持。

表 5.2 选用 QDialogButtonBox 的方法和信号

语法	说明
<code>d.addButton(b, r)</code>	在 QDialogButtonBox 中添加 QPushButton 型按钮 b 和 ButtonRole 型按钮角色 r
<code>d.addButton(t, r)</code>	在 QDialogButtonBox 中添加文本内容为 t 的 QPushButton 型按钮和 ButtonRole 型按钮角色 r
<code>d.addButton(s)</code>	添加一个用于 QDialogButtonBox 中的 QPushButton 型标准按钮 StandardButton s，并返回到 QDialogButtonBox d 所添加的按钮
<code>d.setOrientation(o)</code>	把 QDialogButtonBox 的排列方向设置为 Qt.Orientation o (即竖直或者水平)
<code>d.button(s)</code>	把 QDialogButtonBox 的 QPushButton 型按钮作为 StandardButton s 返回，或者无 QPushButton 型按钮时不返回
<code>d.accepted()</code>	如果点击了 QDialogButtonBox.Accept 角色，就发射本信号
<code>d.rejected()</code>	如果点击了 QDialogButtonBox.Reject 角色，就发射本信号

这里唯一需要验证的是对小数位微调框的有效范围。由于已经选择了提交后验证 (post mortem validation) 的方式，就是说，只有当用户输入数据后，在点击 OK 按钮的当时才会接受编辑的结果并触发验证方法。在下一节，将会看到预防式验证 (preventative validation)，这样就可以让那些不合理的数据无法输入。

```
self.format = format.copy()
```

还需要一份所传入的格式字典的副本，因为我们想在改变对话框内的字典时不会影响原来的初始字典。

```

grid = QGridLayout()
grid.addWidget(thousandsLabel, 0, 0)
grid.addWidget(self.thousandsEdit, 0, 1)
grid.addWidget(decimalMarkerLabel, 1, 0)
grid.addWidget(self.decimalMarkerEdit, 1, 1)
grid.addWidget(decimalPlacesLabel, 2, 0)
grid.addWidget(self.decimalPlacesSpinBox, 2, 1)
grid.addWidget(self.redNegativesCheckBox, 3, 0, 1, 2)
grid.addWidget(buttonBox, 4, 0, 1, 2)
self.setLayout(grid)

```

这个布局的外观与之前见到的画笔属性对话框很相似，只是这次有个 QDialogButtonBox 窗口部件而不是用于各按钮的原有布局。这使得仅用一个 QGridLayout 就有可能创建出整个布局来。

无论是“red negatives”复选框还是这里的按钮框都使用了布局，以便都可以占有一行两列的空间。行和列的合并数是通过 QGridLayout 的 addWidget() 和 addLayout() 方法的最后两个参数给定的。图 5.4 给出了布局的示意图，用灰色网格表示。

```

self.connect(buttonBox, SIGNAL("accepted()"),
            self, SLOT("accept()"))
self.connect(buttonBox, SIGNAL("rejected()"),
            self, SLOT("reject()"))
self.setWindowTitle("Set Number Format (Modal)")

```

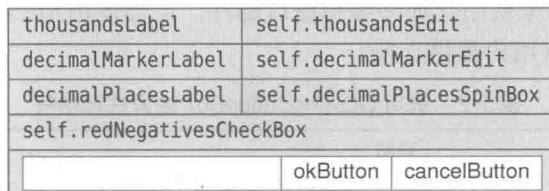


图 5.4 数据格式设置对话框的布局

用于生成连接和设置窗口标题的代码与画笔属性对话框中的那些代码相似，只不过这次使用的是按钮框的信号，而不是与各个按钮直接相连。

```

def numberFormat(self):
    return self.format

```

如果用户点击的是 OK 按钮，对话框会接受设置并返回一个 True 值。这样，通过调用 numberFormat() 方法，窗体调用的方法就会用对话框中的格式字典来覆盖原有的格式字典。由于并没有将对话框的 self.format 属性设置为私有属性（比如，通过调用其 \_\_format\_\_），所以可以在窗体之外对其直接引用；在稍后的例子中，将会用到这一方法。

在用户点击 OK 按钮时，由于使用的是提交后验证方法，可能会出现在某些窗口部件中含有非法数据的情况。为了处理这种情况，就需要重新实现 QDialog.accept() 并在这里提供我们自己的验证方法。由于这一验证方法相当长，所以会分成几个部分来看。

```

def accept(self):
    class ThousandsError(Exception): pass
    class DecimalError(Exception): pass
    Punctuation = frozenset(" ,;.:")

```

一开始，会先创建两个异常类，并在 accept() 方法的内部会用到这两个类。这一做法将有可

能让我们的代码尽可能干净和简洁。也会创建一个字符集，以便用做千位分隔符<sup>①</sup>和小数位的存储。

只需关心的编辑窗口部件是与验证有关的两个行编辑(line edit)窗口部件。这是因为，小数位微调框已经限定为有效范围，而“red negatives”复选框也只有选中和不选两个选项，因而两者都是有效的。

```

thousands = unicode(self.thousandsEdit.text())
decimal = unicode(self.decimalMarkerEdit.text())
try:
    if len(decimal) == 0:
        raise DecimalError, ("The decimal marker may not be "
                             "empty.")
    if len(thousands) > 1:
        raise ThousandsError, ("The thousands separator may "
                               "only be empty or one character.")
    if len(decimal) > 1:
        raise DecimalError, ("The decimal marker must be "
                             "one character.")
    if thousands == decimal:
        raise ThousandsError, ("The thousands separator and "
                               "the decimal marker must be different.")
    if thousands and thousands not in Punctuation:
        raise ThousandsError, ("The thousands separator must "
                               "be a punctuation symbol.")
    if decimal not in Punctuation:
        raise DecimalError, ("The decimal marker must be a "
                             "punctuation symbol.")
except ThousandsError, e:
    QMessageBox.warning(self, "Thousands Separator Error",
                        unicode(e))
    self.thousandsEdit.selectAll()
    self.thousandsEdit.setFocus()
    return
except DecimalError, e:
    QMessageBox.warning(self, "Decimal Marker Error",
                        unicode(e))
    self.decimalMarkerEdit.selectAll()
    self.decimalMarkerEdit.setFocus()
    return

```

从获得两个行编辑窗口部件中的文本开始。尽管有时可能会没有千位分隔符，但小数位分隔符则一定要有，所以先要从检查 decimalMarkerEdit 是否至少拥有一个字符开始。如果有任何字符，就抛出自定义的 DecimalError 异常，以便能够提供适当的错误信息。而如果所有的文本字符中任何一个的长度都不超过一个字符，或者是这些文本字符全部都一样，又或者文本字符中含有非 Punctuation 集合中的字符，就要能够同样抛出异常。这些 if 语句与标点符号不同的地方在于，允许千位分隔符为空，但不允许小数位分隔符也为空。

在两个部分中都使用了圆括号，这样可以把这两部分中的错误信息整合成一个表达式；对于这一语法还有一种替换方法，就是两个部分的链接不用圆括号，中间会使用换行符。

取决于得到的是 ThousandsError 还是 DecimalError，在“警告”消息框中就会显示出适当的错误文本来，如图 5.5 所示。必须把异常对象 e 转换成字符串(可以像之前那样用

<sup>①</sup> 这是指作为数字的千、百万、十亿之间分割的逗号——译者注。

unicode()完成这一工作)，以便可以使其能够成为 QMessageBox 静态方法 warning()的正确参数类型。将来无论是在这一章还是在整本书中，还会用到 QMessageBox 中的更多静态方法，包括额外参数的使用等。

用户一旦获知错误信息并关闭消息框，就可以选中出现错误的行编辑中的非法字符文本，把光标移动到行编辑窗口部件上，以便能够为用户修改做好准备。修改后，再返回，以便能够接受对话框的修改，或者让用户要么修改错误要么点击取消并关闭对话框。

```
self.format["thousandsseparator"] = thousands
self.format["decimalmarker"] = decimal
self.format["decimalplaces"] = \
    self.decimalPlacesSpinBox.value()
self.format["rednegatives"] = \
    self.redNegativesCheckBox.isChecked()
QDialog.accept(self)
```

如果没有抛出异常，任何一个返回语句都不会执行，这样，执行的结果就会到达最后的 accept()方法。在这个方法中，会用编辑窗口部件中的数据更新对话框的 format 字典。之后，对话框窗体就会关闭(也就是隐藏起来)，而 exec\_()语句则返回 True 值。正如之前看到的那样，调用者会收到从 exec\_()语句返回的 True 值，并继续使用 numberFormat()方法获取格式信息。

在最后部分，为什么不用 super()来调用基类的 accept()而是指明用 QDialog 呢？简单的回答是，在这里使用 super()无法得到应有的效果。通过使用 lazy 属性查找，有望进一步提高 PyQt 程序的效率<sup>①</sup>，但结果却是 super()无法在 PyQt 的子类中像我们想象的那样工作(对于其原因的介绍，请参见 PyQt 文档 [pyqt4ref.html](#)，在“super and PyQt classes”一节中)。

尽管只有在对话框被接受(或者被拒绝)时对话框才会隐藏起来，不过它一旦超出作用域，比如在这个例子中，就是调用程序最后的 setNumberFormat1()方法，就会成为垃圾收集处理的对象。

像这样创建一个模态对话框通常较为简单。其中唯一稍为复杂的就是是否需要使用一些布局和确保正确的验证方法，就像我们在这里所做的那样。

某些情况想，用户则希望能够看到做出选择后的结果，可能会不停地做出各种选择改变直到满意为止。对于这些情况，模态对话框可就显得不是很方便了，因为用户必须通过调用对话框，编辑修改，点击确定按钮，查看结果等一遍一遍地重复，直至满意。如果是非模态对话框，用户就可以在不关闭对话框的情况下一直更新程序的状态，这样用户仅需通过一次调用对话框，编辑修改，查看效果，然后再次编辑修改，直到满意：这相较上次会显得方便快捷得多。在下一节，将会看到如何做到这一点；还会看到更为简单、更有交互性的验证方法，即预防式验证(preventative validation)。



图 5.5 QMessageBox 警告示例

<sup>①</sup> 在查询数据的时候，并不需要返回全部字段，这样就能够节省部分运行内存且能进一步提高程序存取数据的速度——译者注。

## 5.3 智能对话框

如果一个对话框能够根据传送到其初始化程序中的引用数据或者数据结构来初始化各个窗口部件，这就使其具有了能够根据用户交互来直接更新数据的能力，则这种对话框即可认为是“智能”的对话框。一个智能对话框(smart dialog)可同时拥有窗口部件级和窗体级的验证能力。智能对话框通常是非模态对话框，具有“应用”(apply)和“关闭”(close)按钮，尽管也可以是“实时”(live)型对话框，此时可以无按钮，可根据所获得的数据来直接反映对窗口部件的变化。非模态智能对话框具有“应用”(apply)按钮，可通过信号和槽连接机制来表明状态的改变。

表 5.3 QDialog 的部分方法

语法	说明
d.accept()	关闭(隐藏)QDialog d，停止其事件循环，exec_()会返回 True 值。如果设置了 Qt.WA_DeleteOnClose，会直接删除该对话框
d.reject()	关闭(隐藏)QDialog d，停止其事件循环，exec_()会返回 False 值
d.done(i)	关闭(隐藏)QDialog d，停止其事件循环，exec_()会返回 int i 值
d.exec_()	模态化显示 QDialog d，阻塞进程直至关闭它
d.show()	非模态化显示 QDialog d；从 QWidget 继承而来
d.setSizeGripEnabled(b)	根据 bool b 的值来决定显示还是隐藏 QDialog d 的尺寸大小拖拽符

使用非模态智能对话框的一个主要好处在于其使用方面。在创建该对话框时，就会给定调用窗体数据结构的引用，以便在调用的地方无须再新增任何代码就可以直接更新该对话框的数据结构。其缺点是，该对话框必须知道调用窗体的数据结构，以便它能够正确反应那些进入自己窗口部件中的数据值，并且只有在这些数据有效时才会应用这些修改值，同时，由于是非模态对话框，也就存在对话框对于正在修改的数据却因用户与应用程序其他部分交互而在后台修改的风险。

这一节，我们还会继续研究数字格式对话框，这样就可以与之前学过的方法进行对比。

### 5.3.1 非模态应用/关闭型对话框

如果希望用户可以重复更改数字格式并能马上看到修改结果，那么就无须不停地调用、接受对话框中数字格式修改所给出的提示，从而会变得非常方便。所需的解决方案是使用非模态对话框，这样用户就可以按照他们喜欢的方式来持续不断地修改数据格式并验证修改的结果了。这类对话框通常都有一个应用(Appl)按钮和一个关闭(Close)按钮。与模态对话框的确定(OK)/取消(Cancel)按钮的风格不同，模态对话框中可以点击取消按钮，然后都会一如之前一样什么都不会发生，而在非模态对话框里，用户一旦点击了应用按钮，改变就会发生，也就无法再撤销到之前的状态。当然，也可以设置一个撤销(Revert)按钮或者是一个默认(Default)按钮，只不过这需要做更多的工作。

从表面上看，模态与非模态对话框的不同之处看似只是按钮上文字的区别。然而，实际上却有两个非常重要的不同：调用窗体的方法在创建和调用对话框时有所不同，且在关闭对话框时，务必要确保删除了对话框，而不是仅仅隐藏了对话框。下面就一起来看看对话框是如何被调用的吧。

```
def setNumberFormat2(self):
    dialog = numberformatdlg2.NumberFormatDlg(self.format, self)
    self.connect(dialog, SIGNAL("changed"), self.refreshTable)
    dialog.show()
```

这个对话框的创建方法与之前模态对话框的创建方法相同，如图 5.6 所示。之后，把对话框的 Python 型信号 `changed` 与对话框调用窗体的 `refreshTable()` 方法进行连接；然后，只需对对话框调用 `show()` 即可。应用程序会继续与对话框一起同时运行，用户则可以与该对话框和应用程序中的其他窗口继续交互。

对话框一旦发射 `changed` 信号，就会调用主窗体的 `refreshTable()` 方法，同时也会用格式字典中的各项设置值来修改表格中所有数字的格式。可以想见，这就意味着，当用户点击应用(`Apply`)按钮的时候，将会更新格式字典并且发射 `changed` 信号。很快就可看到，这到底意味着会发生什么。

尽管对话框变量会超出作用域，但 PyQt 仍会非常智能地保持对非模态对话框的引用，所以该对话框仍会继续存在。不过，在用户点击关闭(`Close`)按钮的时候，对话框通常就会隐藏起来，所以，如果用户一遍又一遍地调用对话框，应用程序就会创建一个又一个的对话框且不会删除之，这样就会造成内存越来越多、不必要的浪费。解决的办法就是确保对话框在关闭后能够删除掉，而不是隐藏起来[后面在介绍“实时”(live)对话框时还会看到另外一种解决方法]。

现在来看看对话框的 `__init__()` 方法。

```
def __init__(self, format, parent=None):
    super(NumberFormatDlg, self).__init__(parent)
    self.setAttribute(Qt.WA_DeleteOnClose)
```

调用完 `super()` 方法之后会调用 `setAttribute()`，以便能够确保对话框在关闭后是被删除了而不仅仅是被隐藏起来了。

```
punctuationRe = QRegExp(r"[ ,;:.]")
thousandsLabel = QLabel("&Thousands separator")
self.thousandsEdit = QLineEdit(format["thousandsseparator"])
thousandsLabel.setBuddy(self.thousandsEdit)
self.thousandsEdit.setMaxLength(1)
self.thousandsEdit.setValidator(
    QRegExpValidator(punctuationRe, self))
decimalMarkerLabel = QLabel("Decimal &marker")
self.decimalMarkerEdit = QLineEdit(format["decimalmarker"])
decimalMarkerLabel.setBuddy(self.decimalMarkerEdit)
self.decimalMarkerEdit.setMaxLength(1)
self.decimalMarkerEdit.setValidator(
    QRegExpValidator(punctuationRe, self))
self.decimalMarkerEdit.setInputMask("X")
decimalPlacesLabel = QLabel("&Decimal places")
self.decimalPlacesSpinBox = QSpinBox()
decimalPlacesLabel.setBuddy(self.decimalPlacesSpinBox)
self.decimalPlacesSpinBox.setRange(0, 6)
```

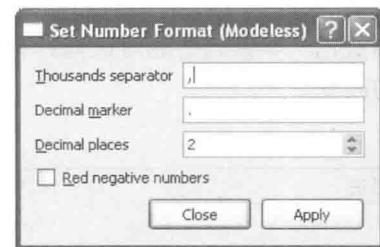


图 5.6 非模态的数据格式设置对话框

```

self.decimalPlacesSpinBox.setValue(format["decimalplaces"])
self.redNegativesCheckBox = QCheckBox("&Red negative numbers")
self.redNegativesCheckBox.setChecked(format["rednegatives"])
buttonBox = QDialogButtonBox(QDialogButtonBox.Apply |
                             QDialogButtonBox.Close)

```

该对话框窗体部件的创建方法与之前用过的方法非常类似，不过这次会在大部分地方使用预防式验证 (preventative validation) 方法。把行编辑窗口部件中的千位和小数位分割符的最大长度均设置为一个字符，并且这两个部分都设置了一个 `QRegExpValidator`。验证器 (validator) 只允许用户输入合法字符，而对于正则验证器则只允许用户输入与正则表达式相符的字符<sup>①</sup>。PyQt 会使用一种正则表达式语法，这是 Python 中 `re` 模块所提供的正则表达式的一个基本子集。

`QRegExpValidator` 初始化程序需要两个参数——一个正则表达式和一个父对象，这就是为什么除了传递正则表达式之外还传递了一个 `self` 对象的原因。

在这个例子中，我们将验证的正则表达式设置为 “[ ,;:. ]” 。这是一个 character class 型表达式，就意味着只有在方括号里面的字符，包括空格、逗号、分号、冒号和句号，才是合法有效的字符。需要注意的是，正则表达式字符串的前面有一个“r”。这表示这是一个“原始” (raw) 字符串，也就是说，字符串中的(几乎)每个字符都是字面意思，没有转义字符。这样就会避免转义诸如“\”之类的字符，尽管在本例中貌似没有什么用。虽然如此，作为较好的编程习惯，还是希望能够在使用正则表达式时总是在表达式前加一个字符“r”。

尽管可以没有千位分隔符，但却必须得有一个小数位分隔符。正是因为如此，所以会用到一个输入掩码 (input mask)<sup>②</sup>。一个“X”掩码表示这里需要一个任意类型的字符，此时就无须考虑输入的字符是否合法，因为前面设置的正则表达式会对其予以验证。在 `QLineEdit.inputMask` 的文档属性中，有关于格式掩码 (format mask) 内容的介绍<sup>③</sup>。

与在模态对话框版本中创建窗口部件方法所不同的唯一之处在于，这里创建的是应用 (Apply) 和关闭 (Close) 按钮，而不是确定 (OK) 和取消 (Cancel) 按钮。

```
self.format = format
```

在模态对话框中我们曾经将调用程序的格式字典做过副本；但在这里，则会对格式字典做一个引用，以便能够在对话框中就可以直接更改它了。

由于这个对话框中的布局与之前介绍的模态对话框的布局完全相同，所以这里就不再给出布局所用到的代码了。

```

self.connect(buttonBox.button(QDialogButtonBox.Apply),
            SIGNAL("clicked()"), self.apply)
self.connect(buttonBox, SIGNAL("rejected()"),
            self, SLOT("reject()"))
self.setWindowTitle("Set Number Format (Modeless)")

```

<sup>①</sup> `QRegExp` 文档对正则表达式给出了简要介绍。对于正则表达式更为深入的内容，可以参阅 Jeffrey E. Friedl 的《精通正则表达式》(Mastering Regular Expressions，有中文版)。

<sup>②</sup> 输入掩码可以控制用户的输入，为开发人员在控制用户输入上提供便利。通过自定义的输入掩码可以控制用户在文本框或字段中只能输入字母、数字以及字母或数字的位数，这样就比用户随意输入后再用代码或有效性规则属性判断输入的有效性要高效很多。例如，输入掩码“###-###-####”可用来限制输入的北美电话号码的格式是否正确——译者注。

<sup>③</sup> 每个 PyQt `QObject` 和 `QWidget` 都有“属性”。这些属性与 Python 的属性类似，只不过是，这些属性可以用 `property()` 和 `setProperty()` 方法进行访问。

会创建两个信号-槽连接。第一个连接是用在应用按钮 Apply 的 clicked() 信号和 apply() 方法之间。为了实现这一连接，必须通过调用 button() 方法重新获得指向按钮框中按钮的引用，并且向其传递一个与之前创建该按钮时所用过的相同的 QDialogButtonBox.Apply 参数。

指向 reject() 的连接会让对话框关闭，且因为有 Qt.WA\_DeleteOnClose 属性，对话框就会被直接删除而不是隐藏起来。还没有连接对话框的 accept() 槽，所以唯一能处理对话框的方法就是关闭这个对话框。如果用户点击了 Apply 按钮，将会调用下面将要给出的 apply() 槽。自然，最后还会设置一下窗口的标题。

这个类中的最后一个方法是 apply()，这里会分成两个部分来查看它。

```
def apply(self):
    thousands = unicode(self.thousandsEdit.text())
    decimal = unicode(self.decimalMarkerEdit.text())
    if thousands == decimal:
        QMessageBox.warning(self, "Format Error",
            "The thousands separator and the decimal marker "
            "must be different.")
        self.thousandsEdit.selectAll()
        self.thousandsEdit.setFocus()
        return
    if len(decimal) == 0:
        QMessageBox.warning(self, "Format Error",
            "The decimal marker may not be empty.")
        self.decimalMarkerEdit.selectAll()
        self.decimalMarkerEdit.setFocus()
        return
```

当两个或多个窗口部件的值相互独立时，窗体级验证通常很有必要。在这个例子中，并不打算允许千位分隔符和小数分隔符一样，所以会在 apply() 方法中进行核查，而如果存在两者一样的情况就通知用户，并把光标放到千位分隔符的行编辑窗口部件上，不使用户的编辑修改起作用并返回。

若能够把两个行编辑窗口部件的信号连接到一个可以进行“核查和修复”的槽上，就可以避免上述情况——在下一个例子中就会这么做。

还必须检查小数分隔符不能为空。尽管小数位分隔符的行编辑窗口部件中的正则表达式验证器只需要一个字符，但也会允许行编辑窗口部件中的内容全部为空。这是因为，对于一个具有有效字符的字符串来说，空字符串可认为是其有效的前缀。归根结底，在用户将光标切换到行编辑窗口部件的时候，其中可能会是全空的。

```
self.format["thousandsseparator"] = thousands
self.format["decimalmarker"] = decimal
self.format["decimalplaces"] = \
    self.decimalPlacesSpinBox.value()
self.format["rednegatives"] = \
    self.redNegativesCheckBox.isChecked()
self.emit(SIGNAL("changed"))
```

如果验证没有问题，return 语句的执行也没有问题，就可以顺利执行到 accept() 槽的最后。这里会更新格式字典。self.format 变量是一个指向调用程序格式字典的引用，所以这些改变会直接应用到调用程序的数据结构上。最后，发射 changed 信号，正如之前看到的，这会调用程序的 refreshTable() 方法，从而也就会用调用程序的格式字典格式化表格中的所有数字。

这个对话框要比在上一节所创建的标准对话框更智能一些。这个对话框会直接作用于调用程序的数据结构(即格式字典)上，并且在数据结构改变后就通知调用程序，以便可以把所做的那些改变应用起来。

在用户打算重复应用修改的情况下，让用户不停地点击应用(Appl)按钮会显得有些不大方便。用户或许只想通过操纵对话框的各个窗口部件后就立即看到修改的效果。下面就来看看该如何实现这一点。

### 5.3.2 非模态的实时对话框

这个将是最后一个数字格式的例子，会看到一个智能化的非模态“实时”(live)对话框——这是一个与我们已见对话框工作方式非常相似的对话框，但该对话框没有任何按钮，且所做的改变会自动、即刻得到应用。对话框如图5.7所示。

在模态对话框版本中曾用过提交后(post-mortem)验证的方法，而在非模态智能对话框版本中则用到了提交后验证和预防式(preventative)验证方法的组合。在这个例子中，将会仅使用预防式验证一种方法。同时，这里不会创建可以让对话框发生改变时通知调用程序的信号-槽机制，而会给对话框一个可以在有改变需要应用时就能够调用的方法，以便可在任何需要时调用该方法。

我们本来也可以像前一个对话框那样完全一样地创建这个对话框，但实际上，这里会给出另外一种完全不同的方法。不是在需要对话框时，先创建再销毁这种模式——每次都用前先创建用后再销毁，而是只创建一次——在第一次确需创建时，然后在用户使用结束后将其隐藏，即每次使用时都是用前先显示用后再隐藏的模式。

```
def setNumberFormat3(self):
    if self.numberFormatDlg is None:
        self.numberFormatDlg = numberformatdlg3.NumberFormatDlg(
            self.format, self.refreshTable, self)
    self.numberFormatDlg.show()
    self.numberFormatDlg.raise_()
    self.numberFormatDlg.activateWindow()
```

在调用窗体的初始化程序中，有这样一个语句：self.numberFormatDlg = None。该语句可以确保在第一次调用这个方法时创建对话框。然后，如之前那样显示对话框。但在这里，在关闭对话框时仅会隐藏起来(因为没有设置Qt.WA\_DeleteOnClose窗口部件属性)。所以，在调用该方法时，会像第一次那样创建并显示对话框，或者会显示之前已经创建过的对话框并随后再隐藏它。考虑到出现第二种情况的可能性，必须应提升(raise，把对话框放到应用程序的所有其他窗口之上)并激活(让对话框获得光标)对话框；在第一次的时候，做这些事情并不会带来什么害处<sup>①</sup>。

而且，这样做会让对话框变得比之前的对话框更为智能，也没有建立信号-槽连接，而只是向对话框传递了一个附加参数，即所绑定的refreshTable()方法。

\_\_init\_\_()方法除了与之前的代码有三处不同之外都完全一样。第一个不同之处是，没有设置Qt.WA\_DeleteOnClose属性，以便在关闭对话框时，对话框会隐藏而不是删除。第

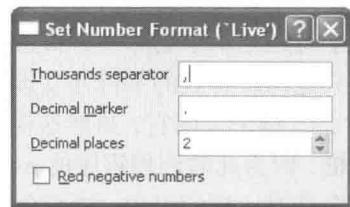


图5.7 数字格式设置“实时”对话框

<sup>①</sup> PyQt会使用raise\_()而不是使用raise()，以避免与内置的raise语句产生冲突。

二点，会保留传递方法的一个副本（比如，会保留一个指向 `self.callback` 中的 `self.refreshTable()` 的引用）；第三，它的信号和槽连接与之前稍有不同。下面是该连接调用的代码：

```
self.connect(self.thousandsEdit,
             SIGNAL("textEdited(QString)"), self.checkAndFix)
self.connect(self.decimalMarkerEdit,
             SIGNAL("textEdited(QString)"), self.checkAndFix)
self.connect(self.decimalPlacesSpinBox,
             SIGNAL("valueChanged(int)"), self.apply)
self.connect(self.redNegativesCheckBox,
             SIGNAL("toggled(bool)"), self.apply)
```

如前一样，可以根据小数位微调框来确保只能设置有效数字的小数位，且与“red negatives”复选框只能处于有效状态相似，任何对这些状态的修改都会被立即应用。

但对于各个行编辑窗口部件来说，现在会连接它们的 `textEdited()` 信号。无论是用户键入字符还是从中删除字符都会发射这些信号。`checkAndFix()` 槽会进一步确保行编辑窗口部件只能保留有效的字符并立即应用这些修改。在该对话框中没有任何按钮：用户可以通过按下 `Esc` 键来关闭它，虽然这样做只会将其隐藏。只有在删除对话框的调用窗体时才会删除对话框，因为此时调用程序的 `self.numberFormatDlg` 实例变量会超出作用域，且也没有其他指向该对话框的引用，故而会成为垃圾收集程序的处理对象。

```
def apply(self):
    self.format["thousandsseparator"] = \
        unicode(self.thousandsEdit.text())
    self.format["decimalmarker"] = \
        unicode(self.decimalMarkerEdit.text())
    self.format["decimalplaces"] = \
        self.decimalPlacesSpinBox.value()
    self.format["rednegatives"] = \
        self.redNegativesCheckBox.isChecked()
    self.callback()
```

`apply()` 方法是目前为止所看到的最为简单的一种。这是因为，只有在所有编辑窗口部件保留有效数据的时候才会调用它，因而也就无须任何提交后验证方法。它不再发射说明状态改变的信号——相反，它会调用所给定的方法，这也会将修改直接应用到调用程序的窗体上。

```
def checkAndFix(self):
    thousands = unicode(self.thousandsEdit.text())
    decimal = unicode(self.decimalMarkerEdit.text())
    if thousands == decimal:
        self.thousandsEdit.clear()
        self.thousandsEdit.setFocus()
    if len(decimal) == 0:
        self.decimalMarkerEdit.setText(".")
        self.decimalMarkerEdit.selectAll()
        self.decimalMarkerEdit.setFocus()
    self.apply()
```

这个方法会使用预防式验证方法来验证用户的键入或者行的编辑修改。我们也仍旧需要依赖于行编辑验证程序、最大长度属性和行编辑窗口部件中小数位的验证，同时再利用这些方法的组合，就可以提供绝大多数情况下的验证需求。不过，用户仍有可能会在两个地方设置同样的文本字符——在这种情况下，会删除千位分隔符并将光标移动到它的行编辑窗口部件上，或者（如果用户总是试图让）小数位分隔符置空——这种情况下，会设置另外一个有效数，选中它，

为其赋予键盘光标。最后可知，这些行编辑窗口部件都是有效的了，所以就可以调用 `apply()` 并应用这些修改了。

使用显示(`show()`)/隐藏(`hide()`)方法的一个好处是，对话框的状态会得到自动维护。如果在每次用到对话框的时候都不得不创建它，那么就必须要能够使用不同的数据来弹出该对话框，但对于这个对话框来说，无论何时显示它(在第一次创建之后)，它都会拥有正确的数据。当然，在这一特定具体的例子中，共有三个编辑同样数据的对话框，这就意味着对话框可能会变得不同步起来；但考虑到在实际的真实应用程序中一般不会出现这种情况，所以这里会暂时先忽略这一问题。

通过传入数据结构(格式字典)和调用程序的更新方法(`refreshTable()`，作为 `self.callback` 进行传递)，使得这个对话框非常智能——并且对它的调用程序来说耦合得也非常紧密。考虑到这一原因，很多编程人员更喜欢使用“折中的”标准对话框——简易对话框有太多限制，使用时不大方便；智能对话框因为与调用程序所施加数据结构有着太过紧密的耦合关系，而显得需要付出太多的维护工作。

## 小结

对话框根据“智能性”可以分为三类，简易型、标准型和智能型；根据显示方式，可以分为模态和非模态两种用法。简易对话框的创建比较简单，也非常适合用做窗口部件级验证。简易对话框通常用做模态对话框，如果细心使用的话，可以广泛使用，因为它们往往与应用程序的逻辑关系耦合比较松散。虽然如此，使用简易对话框通常会让编程人员很有挫败感，结果是需要重新改写为标准对话框或者智能对话框，所以通常最好的办法就是尽量避免使用简易对话框，除非是用在非常简单的只需一个或者两个变量而内置的 `QInputDialog` 静态对话框又不合适的那些情况下。

经常需要在标准模态对话框和智能非模态对话框之间做出抉择，且需要在后者的“应用型”更新和“实时型”更新两类之间做出选择。对程序来说，模态对话框是最简单的，因为会阻塞与对话框的父窗口及父窗口的兄弟窗口之间的其他任何交互，因而可降低对那些正在使用数据在程序后台被修改的风险。但对于一些用户来说却更喜欢非模态对话框，因为在用户决定到底该用哪个方法而不断尝试使用各类不同方法之前会特别方便。如果当模态对话框能够提供某种预览时，它也可以用做相同的目的；例如，字体对话框通常都是模态的，可通过显示样例文本来反映用户修改字体时的字体设置情况。

之前看到的两种验证策略，提交后验证(`post-mortem`)和预防式验证(`preventative`)，可单独或者组合使用。从可用性角度来看，预防式验证通常会优先考虑，尽管它会让用户失望。例如，由于窗体中其他地方的设置而导致某项设置非法时，用户可能就会产生抱怨了(我打算把这个设置成 5，但就是不能如愿)。

也可能会设计一个对话框，以便能够将对话框既用于元素的添加又用于元素的编辑。这些添加/编辑对话框与其他类型的对话框在创建、布局、连接窗口部件等方面没有什么两样。而关键的不同之处在于，根据对话框是用于元素的编辑还是用于元素的添加，这些对话框可能会需要不同的行为方式。在用于元素的编辑时，会利用传入的元素来弹出各个窗口部件，且在编辑时，会用各个窗口部件的初始值弹出。如果接受对话框，或许只会通过那些可以检索的变量设置提供一些简单的存取操作，相关工作仍留给调用程序，或者如果足够智能的话，还可以

直接更新所编辑的各个元素，如果是用户在增添元素就去创建这些新的元素。请参阅文件 chap08/additemmoviedlg.py(其用户界面设计在文件 chap08/additemmoviedlg.ui 中)中的类 AddEditMovieDlg 以及 chap12/pagedesigner.pwy 文件中的类 TextItemDlg，有关添加/编辑元素的对话框。

要避免的另一种可能性是全部使用对话框并让用户可以适当编辑数据，例如，在一个列表或者表格中。这种方法会在模型/视图编程的相关章节中涉及。

## 练习题

编写一个可单独运行的字符串列表编辑对话框。该对话框应当使用 `if __name__ == "__main__"` 的形式，以便可以单独运行和测试该对话框。看起来应该是如图 5.8 所示的样子。

各个字符串应当放到 `QListWidget` 中。由于是用 `Sort` 按钮的 `clicked()` 信号直接连接到 `QListWidget.sortItems()` 方法的，所以 `Sort` 按钮的实现比较简单。

对话框应当作用于它的字符串清单上，该字符串清单要么是传入的副本，要么是自我创建的，在接受的时候，应当发射一个含有该清单(作为 `QStringList` 型)的信号，并要有一个公共存取数据属性，`stringlist`。

槽的实现看起来应该是这样的：

```
def reject(self):
    self.accept()
```

考虑到测试的目的，把以下代码放到文件的最后：

```
if __name__ == "__main__":
    fruit = ["Banana", "Apple", "Elderberry", "Clementine", "Fig",
             "Guava", "Mango", "Honeydew Melon", "Date", "Watermelon",
             "Tangerine", "Ugli Fruit", "Juniperberry", "Kiwi",
             "Lemon", "Nectarine", "Plum", "Raspberry", "Strawberry",
             "Orange"]
    app = QApplication(sys.argv)
    form = StringListDlg("Fruit", fruit)
    form.exec_()
    print "\n".join([unicode(x) for x in form.stringlist])
```

这样会创建一个实例，带一个命名列表中事物类型的字符串，一个字符串列表，然后以模态方式调用它。当用户关闭对话框时，会在控制台中打印字符串清单，以便可以看到编辑的结果。

需要阅读 `QListWidget` 和 `QInputDialog.getText()` 的文档，后者可用于获得待添加的字符串和已有正在编辑的字符串。本练习题应当用 120 行左右的代码完成。

本练习题的参考答案在文件 `chap05/stringlistdlg.py` 中。通过运行程序可以对其进行测试(在 Windows 系统上，应当从控制台运行它；在 Mac OS X 系统上，可以从终端运行它)。



图 5.8 正在添加一个元素的字符串清单对话框

# 第6章 主窗口

- 主窗口的创建
- 用户动作的处理

大多数应用程序都是主窗口型应用程序，也就是说，它们都有一个菜单栏、数个工具栏、一个状态栏、一个中心区并可能还有一些停靠窗口，以便能够为用户提供一个丰富且可导航的全面用户接口。在这一章，将会看到如何来创建一个主窗口型应用程序，其中会给出创建和使用各类特征的做法。

将使用如图 6.1 所示的图片变换应用程序来介绍如何创建主窗口型应用程序。就像大多数这类应用程序一样，都会有一些菜单、工具栏和一个状态栏；还会有一个停靠窗口。除了会看到如何创建所有这些用户接口元素外，还将会介绍如何与它们的用户交互、如何执行相关动作时需要的方法等内容。

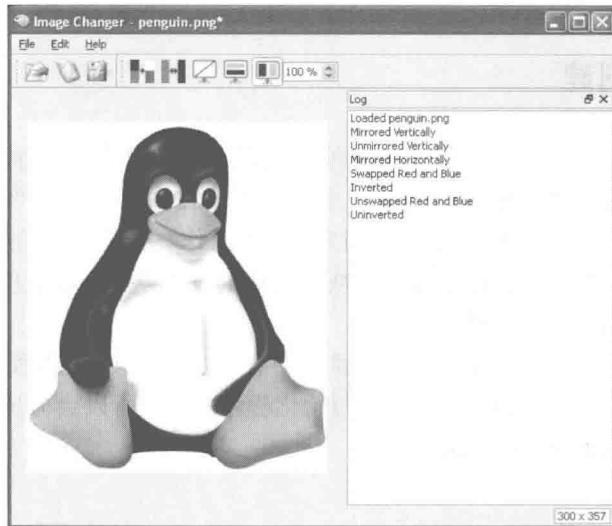


图 6.1 图片变换应用程序

本章还会说明如何处理新文件的创建和已有文件的打开，包括利用应用程序的状态来同步用户界面等。也会涉及如何给用户保存未保存修改的优先权，以及如何管理最近用过的文件清单的方法。还将会介绍如何保存和恢复用户偏好的方法，包括主窗口、工具栏和停靠窗口的大小和位置等。

大多数的应用程序都有用来保存其数据的数据结构，也有可以让用户观察和编辑数据的一个或者多个窗口部件。这里给出的图片变换应用程序 (Image Changer application) 用一个简单的 QImage 来保存着自己的数据，并使用 QLabel 窗口部件作为它的数据观察器。在第 8 章中，将会看到一个用来显示和编辑许多数据元素的主窗口型应用程序；在第 9 章，还会看到如何创建具有多文档处理能力的主窗口应用程序。

在看到如何创建应用程序之前，将讨论一些在用户界面中必须维护的状态。最为常见的是，一些菜单选项和工具栏按钮都是“复选”型，也就是说，它们可能会处于两种状态中的一个。例如，在字处理软件中，用于斜体字的工具栏按钮就应当是可勾选为“开”(on)(按下)或者“关”(off)的状态。如果还有一个斜体字菜单栏选项，那么就必须应确保菜单栏选项和工具栏按钮处于同步状态。幸运的是，PyQt 让这种同步状态变得非常容易。

还有一些选项可能会相互依赖。例如，任何时候，字体对齐只能是左对齐、居中和右对齐三种状态中的一种。因而若用户切换到了居中对齐，那么左对齐、右对齐工具栏按钮和菜单项就必须是关闭的。再次说明一下，PyQt 可以使这种相互依赖的状态同步变得很简单。在本章，既会看到非复选选项操作的内容，如“打开文件”，也会看到独立和相互依赖复选项操作的内容。

尽管一些菜单项和工具栏选项都可以对应用程序的数据产生实时响应，但还是会有其他一些手段，通过调用对话框来进一步精准反映出用户的明确意愿。由于在之前的两章中已经为对话框提供了大量内容，这一章会将重点放到它们的用法而不是它们的创建方法上。这一章还会看到如何调用自定义对话框，以及如何使用许多 PyQt 的内置对话框，包括用于选择文件名的那些对话框、打印对话框以及用于诸如用户查询一个数据元素(如一个字符串或者某个数字等)的对话框。

## 6.1 主窗口的创建

对于大多数主窗口型应用程序，主窗口的创建都会遵循相似的模式。本节将先从一些数据结构的创建和初始化开始，然后会创建一个占有主窗口中心区域的“中心窗口部件”，接着会创建并构建一些停靠窗口。接下来，会创建一些“动作”并将它们插入菜单栏和工具栏中。对于应用程序来说，读取应用程序的设置，恢复用户的工作空间，加载应用程序上次结束时已打开的文件等，这些也都比较常见。

在图 6.2 中给出了构成图片变换应用程序的各个文件。该应用程序的主窗口类在文件 chap06/imagechanger.pyw 中。由于其初始化程序较长，所以会分成几个部分来看。首先先来看看在类定义之前的那些导入语句。

```
import os
import platform
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import helpform
import newimagedlg
import qrc_resources

__version__ = "1.0.0"
```

在本书中，实践应用时会先导入 Python 的标准模块，然后会导入第三方模块(如 PyQt)，最后将是自己的一些自定义模块。在代码中，将会讨论一些来自操作系统和平台模块的元素项。sys 模块会像往常一样提供 sys.argv。helpform 和 newimagedlg 模块会提供 HelpForm 和 NewImageDlg 类。随后还会探讨 qrc\_resources 模块。

带有版本字符串的应用程序司空见惯，通常都会通过调用 \_\_version\_\_ 来实现，会用在应用程序的关于菜单项中。

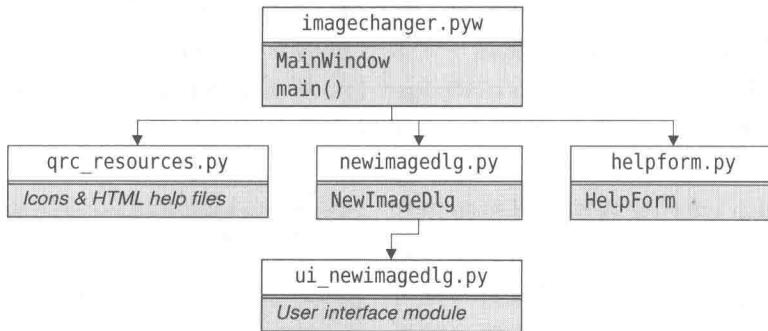


图 6.2 图片变换应用程序的模块、类和函数

现在来看看 MainWindow 类的开头部分：

```

class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.image = QImage()
        self.dirty = False
        self.filename = None
        self.mirroredvertically = False
        self.mirroredhorizontally = False
  
```

初始化程序依照惯例会从 `super()` 调用开始。然后，会创建一个 null 的 `QImage`，以用来保存用户加载或者创建的图片。`QImage` 不是 `QObject` 的子类，所以它并不需要父对象；相反，可以在应用程序结束的时候，将程序的删除交给常规的 Python 垃圾收集程序进行处理。还会创建一些实例变量。用布尔变量 `dirty` 标志作为说明图片是否存在未保存修改的标志。`filename` 会初始化设置为 `None`，以便用来说明是没有图片，还是有尚未保存的新创建图片。

PyQt 提供了一些形式各异的镜像功能，但对于这个例子中的应用程序来说，已经限定为三种可能：图片的竖直镜像、水平镜像和翻转。只需追踪镜像状态以便可以保持用户界面的同步状态，在讨论镜像动作的时候将会看到。

```

self.titleLabel = QLabel()
self.titleLabel.setMinimumSize(200, 200)
self.titleLabel.setAlignment(Qt.AlignCenter)
self.titleLabel.setContextMenuPolicy(Qt.ActionsContextMenu)
self.setCentralWidget(self.titleLabel)
  
```

在一些应用程序中，中心窗口部件是一个复合窗口部件（就是一个由其他数个窗口部件、布局得像对话框一样的窗口部件），或者是一个基于元素的窗口部件（如列表或者表格），但在这里，只需一个 `QLabel` 足矣。`QLabel` 可以显示普通文本、HTML 或者是 PyQt 所支持的任意格式的图片；随后会看到到底有哪些图片格式，因为确实差异比较大。因为初始的标签（label）并没有任何可显示的内容，因而也不会占用任何空间，所以需要为其设置一个最小的尺寸大小，虽然看起来稍微有些奇怪。最后，选中各个图片，将其竖直和水平居中对齐。

PyQt 提供了许多创建上下文菜单的方法，但这里将会使用一种最为简单且最为常用的方法。首先，必须为打算使用上下文菜单的窗口部件设置相应的上下文菜单策略。然后，必须向该窗口部件添加一些动作——这些会在后面深入涉及。当用户调用上下文菜单的时候，菜单就会弹出来，显示添加过的那些动作。

不像在对话框中，我们使用的是布局，在主窗口风格的应用程序中，有的只是一个中心窗口部件——尽管这个窗口部件可能是一个复合型的窗口部件，但在实际应用中则不存在任何限制。只需调用一下 `setCentralWidget()`，所有的事情就都好了。这个方法既可以让主窗口中心区域的窗口部件进行布局，又可以重定义该窗口部件的父对象，以便能够让主窗口掌控它。

工具栏 (toolbar) 可用来容纳工具栏按钮和一些类型的窗口部件，如组合框和微调框。对于比较大的窗口部件、工具库面板或者其他可以让用户从窗口中拖拽出来并以独立窗口的形式自由浮动的窗口部件，使用停靠窗口通常会是比较正确的选择。

停靠窗口 (dock window) 就是一些可以如图 6.3 那种可以出现在停靠区域的窗口。它们会有一个小标题，还有一些恢复、关闭的按钮，可以从一个停靠区域拖拽另一个停靠区域，或者是独立自主地自由浮动于顶级窗口上。停靠在停靠区域后，会自动在它们和中心区域之间提供一个切分窗口，这样就可以非常方便地改变其尺寸大小。

在 PyQt 中，停靠窗口都是类的实例。可以向停靠窗口部件添加一个单一的窗口部件，就像在主窗口的中心区域可以有一个单一的窗口部件一样，与此道理相同，这样的添加操作不会有任何限制，因为所添加的窗口部件可以是复合型的窗口部件。

```
logDockWidget = QDockWidget("Log", self)
logDockWidget.setObjectName("LogDockWidget")
logDockWidget.setAllowedAreas(Qt.LeftDockWidgetArea |
                             Qt.RightDockWidgetArea)
self.listWidget = QListWidget()
logDockWidget.setWidget(self.listWidget)
self.addDockWidget(Qt.RightDockWidgetArea, logDockWidget)
```

各停靠窗口部件不能放进布局中，所以在创建时，除了要提供窗口标题外，还要为其给定父对象。通过父对象设置，可以确保停靠窗口部件不会超出作用域，也不会在错误的时间成为 Python 的垃圾收集程序的处理对象。相反，该停靠窗口部件会在它的父对象，即顶级窗口（主窗口），被删除的时候也删除掉。

每一个 PyQt 对象都可以给一个对象名，尽管到目前为止，还没有这么做过。对象名有时候在代码调试时非常有用，不过在这里设置对象名只是为了让 PyQt 能够保存和恢复停靠窗口部件的尺寸大小和位置，因为可能那里会有许多的停靠窗口部件，PyQt 需要用对象名来从中予以区分出来。

默认情况下，停靠窗口部件都是可移动、可浮动、可关闭且可以拖放到任何停靠区域。由于这里的停靠窗口部件主要是打算保存一个列表（通常是一个又高又窄的窗口部件）通常只愿意停靠（或者浮动）在停靠区域的左边或者右边，所以会用 `setAllowedAreas()` 来限制其停靠区域。停靠窗口部件还有一个 `setFeatures()` 方法，可用来控制停靠窗口部件是否可以移动、浮动、关闭等特性，但这里并不需要使用它，因为仅使用各默认值就可以了。



图 6.3 QMainWindow 的各个区域

停靠窗口部件一旦创建后，就可以创建其中要包含的窗口部件了，在这个例子中是一个列表窗口部件。然后，把创建的列表窗口部件添加到停靠窗口部件中，并把停靠窗口部件添加到主窗口中。在这里，列表窗口部件没有给定父对象是因为，在将其添加到停靠窗口部件中时，停靠窗口部件就会掌控它。

```
self.printer = None
```

希望用户能够打印用户自己的图片。要想这样，就需要创建一个 `QPrinter` 对象。本来是，无论何时需要打印机，都可以随时创建一个打印机并在随后交由垃圾收集程序管理起来即可。但最好是能够保留打印机实例变量，一开始可以初始化为 `None`。在用户第一次要求打印的时候，会创建一个 `QPrinter` 对象并将其赋值给变量。这样做有两个优点。第一，只有在需要打印机对象的时候才会创建该打印机对象，第二，因为保留了打印机对象的印象，它就会一直存在——也会一直保存其前一个状态，如用户选择的打印机、纸张大小等选项。

```
self.sizeLabel = QLabel()
self.sizeLabel.setFrameStyle(QFrame.StyledPanel|QFrame.Sunken)
status = self.statusBar()
status.setSizeGripEnabled(False)
status.addPermanentWidget(self.sizeLabel)
status.showMessage("Ready", 5000)
```

对于应用程序的状态栏来说，希望将常用的消息区放在左侧，并用状态指示器说明当前图片的宽度和高度。这可以通过创建 `QLabel` 窗口部件并添加到状态栏中来实现这一点。还需要关闭状态栏的尺寸大小拖拽符，否则的话，在状态栏中却放一个显示图片尺度的指示器标签看起来会有些不太恰当。在第一次调用 `QMainWindow` 的 `statusBar()` 方法时会自动创建状态栏。如果用字符串调用状态栏的 `showMessage()` 方法，那么该字符串就会显示在状态栏上，并会一直显示到下一个 `showMessage()` 将其排挤掉或者直到调用 `clearMessage()`。之前曾用过双参数窗体，其中的第二个参数是毫秒数(5000，也就是5秒)，即该条消息应当显示的时间；过了这么长的时间后，状态就会将该条消息清除掉。

到目前为止，已经看到了主窗口中心窗口部件、停靠窗口部件的创建和状态栏的设置。现在也基本为菜单栏、工具栏的创建做好了准备，不过还是先应理解什么是 PyQt 的动作，然后再简要看看有关资源的内容。

### 6.1.1 动作和按键顺序

Qt 的设计师程序能够识别用户界面中用户做同样事情时所经常使用的数种不同方法。例如，在许多应用程序中创建一个新文件可以使用 `File→New` 菜单项，或者通过点击 `New File` 工具栏按钮 ，或者直接使用 `Ctrl+N` 键盘快捷键。一般来说，对于用户是如何执行动作我们并不关心，关心的只是到底他们要做什么。PyQt 会使用 `QAction` 类来封装用户的动作。所以，例如，若要创建一个“文件-新建”的动作，可以像这样撰写代码：

```
fileNewAction = QAction(QIcon("images/filenew.png"), "&New", self)
fileNewAction.setShortcut(QKeySequence.New)
helpText = "Create a new image"
fileNewAction.setToolTip(helpText)
fileNewAction.setStatusTip(helpText)
self.connect(fileNewAction, SIGNAL("triggered()"), self.fileNew)
```

这里会假设已有适当的图标和 `fileNew()` 方法。在菜单项文本中的符号意思是该菜单将会显示为 `>New`(除非是在 Mac OS X 系统上，或者是窗口系统设置成了按下才出现下画线)，使用键

盘的用户将可以通过按下 Alt + F、N 的形式调用它，假设 File 菜单的文本是“&File”，那么就可以显示成 File。另外一种方法是，用户还可以使用通过 setShortcut() 创建的快捷键，只需按下 Ctrl + N 即可。

很多按键序列都已经得到了标准化，甚至有些按键序列都已经可以跨越不同窗口系统使用。例如，Windows、KDE 和 GNOME 都会在“新建文件”的时候使用 Ctrl + N，“保存”的时候使用 Ctrl + S。在 Mac OS X 系统上也类似，对于这两个动作会分别使用 Command + N 和 Command + S。在 PyQt 4.2 中的 QKeySequence 类会提供一些标准化按键序列的常量，如 QKeySequence.New。在跨越不同窗口系统的时候，或者是有多个案件次序关联到同一个动作时，这种标准化的按键序列就尤其有用了。例如，如果给 QKeySequence.Paste 设置一个快捷键，PyQt 就会在 Windows 窗口系统上用 Ctrl + V 或者 Shift + Ins 调用“粘贴”动作，在 GNOME 和 KDE 窗口系统上用 Ctrl + V、Shift + Ins 或者 F18，而在 Mac OS X 则用 Command + V。

对于尚未标准化的那些按键序列（或者打算要与 PyQt 之前发行版本向下兼容时），可以以字符串的形式作为快捷键，例如，调用 setShortcut("Ctrl + Q")。本书会使用各标准化的按键序列，否则就会使用字符串作为快捷键。

值得注意的是，需要给 QAction 一个父对象 self（该窗体中的动作是可用的）。每个 QObject 子类（顶级窗口除外）都有一个父对象，这一点非常重要；对于窗口部件来说，通常可以通过对它们进行布局来实现这一点，不过对于纯粹的数据对象，比如 QAction，就必须明确地为其提供父对象了。

表 6.1 QAction 的部分方法

语法	说明
a.data()	返回 QAction a 的 QVariant 型用户数据
a.setData(v)	把 QAction a 的用户数据设置成 QVariant v 型
a.isChecked()	如果 QAction a 是选中的，就返回 True
a.setChecked(b)	根据 bool b 的值，选中或者未选中 QAction a
a.isEnabled()	如果 QAction a 可用，就返回 True
a.setEnabled(b)	根据 bool b 的值，QAction a 为可用或者不可用
a.setSeparator(b)	根据 bool b 的值，将 QAction a 设置为常规动作或者分隔符
a.setShortcut(k)	将 QAction a 的键盘快捷键设置为 QKeySequence k
a.setStatusTip(s)	将 QAction a 的状态提示文本设置为字符串 s
a.setText(s)	将 QAction a 的文本设置为字符串 s
a.setToolTip(s)	把 QAction a 的工具提示文本设置为字符串 s
a.setWhatsThis(s)	把 QAction a 的 What's This? 的文本设置为字符串 s
a.toggled(b)	当 QAction a 的选中状态发生改变时，发射本信号；如果该动作选中，那么 bool b 的值就是 True
a.triggered(b)	当调用 QAction a 时，发射本信号；如果选中 QAction，那么可选项 bool b 的值就是 True

该动作一旦创建，就可以像下面这样将其添加到菜单栏或者工具栏上了：

```
fileMenu.addAction(fileNewAction)
fileToolbar.addAction(fileNewAction)
```

现在，无论用户何时调用“文件→新建”动作（无论通过何种方式），都会调用 fileNew() 方法。

### 6.1.2 资源文件

遗憾的是，我们之前完成的代码有一点小问题。代码假设应用程序的工作目录就是它所在的目录。在 Windows 系统下点击（或者是双击）.pyw（或者是指向它的一个快捷键）运行时，这是常见的情况。但如果是在命令行中的其他不同目录处执行该程序——例如，从./chap06/imagechanger.pyw 目录——那么就不会显示任何图标。这是因为，我们把图标的路径设置为 images，也就是说，是一个与应用程序工作目录相对的目录，所以当从其他地方调用该程序时，就会从./images 目录（或许就根本不存在）中查找这些图标，但实际上，这些图标的目录在./chap06/images。

或许可以用 Python 的 os.getcwd() 函数来尝试解决这一问题；但这样做只会返回调用该应用程序的目录，正如之前提到过的，可能并不是该应用程序实际所在的目录。使用 PyQt 的 QApplication.applicationDirPath() 方法也不会有任何帮助，因为这样只会返回 Python 可执行文件的目录，而不是应用程序自身的目录。一种解决方法是使用 os.path.dirname(\_\_file\_\_) 来为每个图标的文件名提供一个前缀，因为在\_\_file\_\_ 变量中，会含有当前.py 或者.pyw 文件的全名和目录。

另外一种解决方法是，把所有的图标文件（以及帮助文件和其他较小的资源文件）都放到一个.py 模块中，然后再从那里访问它们。这样就不仅解决了目录问题（因为 Python 知道该如何去寻找导入的模块），也意味着再无须使用大量的图标、帮助及类似文件，毕竟这些文件确实可能会造成丢失之类的问题，现在则只需要一个模块就可以包含全部了。

为了能够生成资源模块，必须完成两件事。首先，必须创建一个含有拟使用资源文件细节的.qrc 文件；其次，必须运行 pyrcc4，它会读取一个.qrc 文件并生成资源模块。.qrc 文件是一种简单的 XML 格式文件，可很容易地通过手工书写来完成。下面给出的就是从图片转换应用程序 resources.qrc 文件中提取出来的内容：

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
<file alias="filenew.png">images/filenew.png</file>
<file alias="fileopen.png">images/fileopen.png</file>
...
<file alias="icon.png">images/icon.png</file>
<file>help/editmenu.html</file>
<file>help/filemenu.html</file>
<file>help/index.html</file>
</qresource>
</RCC>
```

上述代码中的省略号表示因版面有限、内容简单而省略的多行内容。每个<file>开头都会含有一个文件名，如果该文件在子目录中，则还会包含其相对目录。现在，如果打算使用“文件→新建”动作的图片，本可以只写为 QIcon(":/images/filenew.png")，不过因为有了别名化方法，就可以将其简写成 QIcon(":/filenew.png") 的形式。开头的“:/”会告诉 PyQt，这是一个资源。资源文件可以像文件系统中的常规文件一样进行处理，唯一的不同点是，它们会带有特殊的目录前缀。但在使用资源文件之前，必须确保已经生成过资源模块并将其导入到了应用程序中。

前面已经看到是如何为图片转换应用程序导入模块的，最后要看到的是 import qrc\_resources。pyrcc4 通过下面的命令行语句就可以生成 qrc\_resources.py 模块：

```
C:\pyqt\chap06>pyrcc4 -o qrc_resources.py resources.qrc
```

无论何时修改了 `resources.qrc` 文件，都必须运行上述命令。

为简便起见，会给出两个小的 Python 程序，以便能够作为 `pyrcc4` 使用的例子，还会给出一些更为简单的其他 PyQt 命令行程序。一个是 `mkpyqt.py`，这是一个命令行程序；另一个是 `Make PyQt`，这是一个用 PyQt4 写成的 GUI 应用程序。这就是说，例如，无须亲自运行 `pyrcc4`，而只需像这样简单键入：

```
C:\pyqt\chap06>mkpyqt.py
```

无论是 `mkpyqt.py` 还是 `Make PyQt` 都可以完成同样的事情：运行 `pyuic4` 和其他 PyQt 工具，它们每个都还会自动使用正确的命令行参数；在下一章会介绍到它们。

### 6.1.3 创建和使用动作

之前在创建“文件→新建”动作的代码中我们已经看到，创建和设置动作需要 6 行的代码。大多数的主窗口型应用程序都有许多动作，因而如果每个动作都有 6 行代码的话，就会变得非常冗长了。基于此，可以创建一个程序助手的函数，以便能够把创建动作的代码量减少到 2~3 行。下面就来看看程序助手，然后再看看是如何将其应用于主窗口初始化程序中的。

```
def createAction(self, text, slot=None, shortcut=None, icon=None,
                 tip=None, checkable=False, signal="triggered()"):
    action = QAction(text, self)
    if icon is not None:
        action.setIcon(QIcon(":/%s.png" % icon))
    if shortcut is not None:
        action.setShortcut(shortcut)
    if tip is not None:
        action.setToolTip(tip)
        action.setStatusTip(tip)
    if slot is not None:
        self.connect(action, SIGNAL(signal), slot)
    if checkable:
        action.setCheckable(True)
    return action
```

这个方法会自动完成“文件→新建”动作中所需的全部手工工作。此外，它还可以处理没有图标的情况，也可以处理“复选型”动作。图标通常是可有可无的，尽管对于一些需要添加到工具栏的动作来说，通常会提供一个图标。如果一个动作具有“开”和“关”的状态，该动作就是复选型，就像字处理程序中通常会提供加粗或者斜体动作一样。

值得注意的是，`QAction` 构造函数的最后一个参数是 `self`；这就是该动作的父对象（即主窗口），可以确保在该动作超出初始化程序的作用域时，该动作不会被垃圾收集程序处理掉。某些情况下，我们会创建一些动作实例变量，以便可以在窗体的初始化程序之外仍旧可以访问它们，但在这个特定的样例中，我们不会这么做。

这里给出的就是如何使用 `createAction()` 程序助手函数来创建“文件→新建”动作的内容：

```
fileNewAction = self.createAction("&New...", self.fileNew,
                                  QKeySequence.New, "filenew", "Create an image file")
```

有个特例是“文件→退出”动作（以及“文件→另存为”动作，对于它们不会提供快捷键），其他

的文件操作的动作会用同样的方式进行创建，所以就不再逐一介绍它们。

```
fileQuitAction = self.createAction("&Quit", self.close,
    "Ctrl+Q", "filequit", "Close the application")
```

QKeySequence 类并没有用于应用程序结束的标准化快捷键，所以我们自己会为其选择一个快捷键并将其作为字符串。当然，也可以仅是简单地使用那些不同的空间，例如，Alt + X 或者 Alt + F4。

close() 槽继承自 QMainWindow。如果是通过调用“文件→退出”动作(之前已将其连接到了 close() 槽)来关闭主窗口的，例如，通过点击文件(File)→退出(Quit)或者按下 Ctrl + Q，就会调用基类的 close() 方法。但如果用户点击了应用程序的关闭按钮 X，就不会调用 close() 方法。

唯一能够确保拦截试图关闭窗口的方法是重新实现关闭事件的处理程序。无论是通过 close() 方法还是借助关闭(close) 按钮来关闭应用程序，总是会调用关闭事件的处理程序。因此，通过重新实现该事件处理程序就可以让用户拥有保存任何尚未保存的编辑修改的优先权，也可以用来保存应用程序的各项设置。

一般情况下，完全可以通过高级的信号和槽机制来实现一个应用程序的行为，但若要使用这种方法，有一点就显得极为重要了，就是必须使用低级的事件处理机制。然而，重新实现关闭事件与重新实现其他方法没有什么两样，并且也并不困难，随着不断深入学习，很快就会看到这一点(事件处理会在第 10 章讲述)。

创建编辑动作的方法类似，不过因为稍有不同，所以会稍稍看些内容

```
editZoomAction = self.createAction("&Zoom...", self.editZoom,
    "Alt+Z", "editzoom", "Zoom the image")
```

让用户能够对图片进行放大或者缩小操作以便更多或者更少地看到细节将会显得非常方便。在工具栏上提供了一个微调框，允许鼠标使用者改变缩放因子(很快会在后面看到)，不过也得能够为键盘用户提供支持，所以会创建一个会加入到编辑(Edit)菜单上的“编辑→缩放”动作。在调用时，与该动作相连接的方法将会弹出一个对话框，用户可以在其中输入缩放比例值。

对于放大和缩小都有标准化的按键序列，但尚缺少常规缩放操作的按键序列，所以在里会选用 Alt + Z 的按键序列(之所以没有使用 Ctrl + Z 是因为在大多数的平台下，这都是用做撤销动作的按键序列)。

```
editInvertAction = self.createAction("&Invert",
    self.editInvert, "Ctrl+I", "editinvert",
    "Invert the image's colors", True, "toggled(bool)")
```

“编辑撤销”(edit invert) 动作是一个切换型动作。本来也是可以使用 triggered() 信号的，不过后面可能还是会通过调用 isChecked() 来查询该动作的状态。使用 toggled(bool) 信号可能会更方便些，这是因为，这个信号不仅会说明该动作已经得到调用，还会说明是否选中了该动作。各个动作也有仅在用户发生修改时才发射的 triggered(bool) 信号，不过在这里，因为无论是通过用户还是通过程序代码来检查是否发生过反转动作(invert action)，都显得有些不合适，我们想要的只是对其进行响应。

由于“编辑→红蓝交换”(edit→ swap red and blue) 这个动作与“编辑→反转”动作类似，所以就不再给出它的代码了。

与“编辑→撤销”动作和“编辑→红蓝交换”动作一样，镜像动作也是可选型动作，只不过不像各自独立的“撤销”和“红蓝交换”动作，会让两个镜像动作之间互斥，也就是任何时候都

仅允许其中之一处于“开”的状态。为了能够得到这一行为，仍会用常规方法创建镜像动作，但会将它们都添加到一个“动作群组”(action group)之中。所谓的动作群组其实是一个类，可以管理一系列的可选型动作，并可确保它所管理的所有动作只要有一个“开”，则其他动作都为“关”。

```
mirrorGroup = QActionGroup(self)
```

动作群组是 QObject 的一个子类，既不是一个顶级窗口也不是一个可以布局的窗口部件，所以必须明确地为其指定父对象，以确保 PyQt 可以在正确的时间删除它。

动作群组一旦创建好，就可以用之前一样的方法来创建这些动作了，只是现在要把它们都添加到动作群组中而已。

```
editUnMirrorAction = self.createAction("&Unmirror",
                                       self.editUnMirror, "Ctrl+U", "editunmirror",
                                       "Unmirror the image", True, "toggled(bool)")
mirrorGroup.addAction(editUnMirrorAction)
```

这里并没有给出“编辑→纵向镜像”或者“编辑→横向镜像”动作的代码，因为这些代码与之前给出的几乎一模一样。

```
editUnMirrorAction.setChecked(True)
```

复选型动作默认为“关”，但如果是诸如这样的群组中，就是其中之一必须为“开”时，首先要做的就是从中选择一个“开”的。在这种情况下，“编辑→取消镜像”(edit→unmirror)动作在初始化时就可能是最应该打开的动作了。选中一个动作会使其发射 toggled() 信号，但此时，QImage 仍旧是空的，所以将会看到，是不会对一个空图片应用任何操作的。

再创建两个动作，“帮助→关于”和“帮助→帮助”，其代码与之前已经见到的代码非常相似。

尽管这些动作现在已经存在了，但目前任何一个都还没有办法实际工作！这是因为，只有在将它们添加到菜单、工具栏或者菜单与工具栏的时候，它们才会变得可操作。

通过访问主窗口的菜单栏会创建菜单栏中的各个菜单(在第一次调用 menuBar() 的时候就会创建主窗口的菜单栏，就像创建状态栏那样)。下面给出了创建编辑(Edit)菜单的代码：

```
editMenu = self.menuBar().addMenu("&Edit")
self.addActions(editMenu, (editInvertAction,
                           editSwapRedAndBlueAction, editZoomAction))
```

在创建编辑(Edit)菜单之后，会使用 addActions() 向其中添加一些动作。要生成图 6.4 中不含镜像选项 Mirror 的编辑(Edit)菜单，这样做就足够了，镜像菜单的创建随后将会看到。

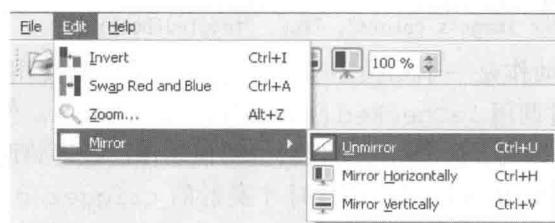


图 6.4 编辑(Edit)菜单和镜像(Mirror)子菜单

使用可以将动作添加到菜单和工具栏上。为减少输入工作，这里会创建一个小型的程序助手函数，可以用来将动作添加到菜单或者工具栏上，也可以用来添加几个分隔符。这里是其代码：

```
def addActions(self, target, actions):
    for action in actions:
        if action is None:
            target.addSeparator()
        else:
            target.addAction(action)
```

target 是一个菜单或者工具栏，actions 是一个列表或者是动作或 None 的元组。本来是可以使用内置的 QWidget.addAction() 方法的，不过在那种情况下，最好还是创建一些分隔符动作（随后会给出）而不是使用几个 None。

在编辑(Edit)菜单上的最后一个选项是镜像 Mirror，在它的右侧带有一个小的三角符号。这个符号表示它会有子菜单。

```
mirrorMenu = editMenu.addMenu(QIcon(":/editmirror.png"),
                               "&Mirror")
self.addAction(mirrorMenu, (editUnMirrorAction,
                           editMirrorHorizontalAction, editMirrorVerticalAction))
```

各个子菜单(submenu)的显示方式与其他菜单的显示方式完全一样，只不过会用 QMenu.addMenu() 将其添加到它们的父菜单上，而不是使用 QMainWindow.menuBar().addMenu() 将其添加到主窗口的菜单栏。在创建了镜像菜单 Mirror 之后，就可以像之前所做的那样用程序助手函数 addActions() 向其添加各动作。

在创建大多数的菜单后，就会用与编辑(Edit)菜单一样的方法进行显示，只不过文件(File)菜单会稍有不同。

```
self.fileMenu = self.menuBar().addMenu("&File")
self.fileMenuActions = (fileNewAction, fileOpenAction,
                       fileSaveAction, fileSaveAsAction, None,
                       filePrintAction, fileQuitAction)
self.connect(self.fileMenu, SIGNAL("aboutToShow()"),
             self.updateFileMenu)
```

我们希望文件(File)菜单能够显示最近用过的那些文件。基于这一原因，我们不会在这里显示文件(File)菜单，相反，会在用户调用它的时候再动态地创建它。这就是为什么创建一个文件(File)菜单的实例变量，也是为什么要用一个实例变量来保存文件(File)菜单的各个动作。这一连接会确保无论何时调用文件(File)菜单，都可以调用 updateFileMenu() 槽。随后会进一步回顾这个槽。

帮助( Help)菜单的创建没有什么新意，与编辑(Edit)菜单的创建方式一样，所以这里就不再给出其说明了。

在依次看过各个菜单后，现在可以转头来看看工具栏。

```
fileToolbar = self.addToolBar("File")
fileToolbar.setObjectName("FileToolBar")
self.addAction(fileToolbar, (fileNewAction, fileOpenAction,
                           fileSaveAsAction))
```

工具栏的创建与菜单的创建有些类似：调用 addToolBar() 可以创建一个 QToolBar 对象，使用 addActions() 可以显示它。对于菜单和工具栏都可以使用 addActions() 方法，因为它们的 API 非常相似，两者都提供了 addAction() 和 addSeparator() 方法。为了能够让 PyQt 存储和恢复工具栏的位置，会设置一个对象名——可以是任意数量的工具栏，PyQt 都可以使用对象名来分清它们，就像是作用于停靠窗口部件一样。图 6.5 中给出了工具栏完成后的效果。



图 6.5 文件(File)工具栏

编辑工具栏和各个可选型动作(“编辑→撤销”、“编辑→红蓝交换”以及镜像动作)在创建时都使用了同样的方法。但正如图 6.6 所示的那样, 编辑工具栏除了工具栏上的各个按钮之外还有一个微调框。为了看到这一点, 这里会尽可能详尽地给出其代码, 但考虑到说明的方便性, 会将其分为两个部分。

```
editToolbar = self.addToolBar("Edit")
editToolbar.setObjectName("EditToolBar")
self.addAction(editToolbar, (editInvertAction,
    editSwapRedAndBlueAction, editUnMirrorAction,
    editMirrorVerticalAction,
    editMirrorHorizontalAction))
```

创建一个工具栏并向其添加一些动作是所有工具栏都一样的创建模式。

为了能够向用户提供一种修改缩放因子的快捷方式, 会通过在编辑工具栏上提供一个微调框来尽可能地做到这一点。在这之前, 我们曾经在编辑菜单 Edit 中单独添加过一个“编辑→缩放”动作, 以用来满足键盘用户的需要。

```
self.zoomSpinBox = QSpinBox()
self.zoomSpinBox.setRange(1, 400)
self.zoomSpinBox.setSuffix("%")
self.zoomSpinBox.setValue(100)
self.zoomSpinBox.setToolTip("Zoom the image")
self.zoomSpinBox.setStatusTip(self.zoomSpinBox.toolTip())
self.zoomSpinBox.setFocusPolicy(Qt.NoFocus)
self.connect(self.zoomSpinBox,
    SIGNAL("valueChanged(int)"), self.showImage)
editToolbar.addWidget(self.zoomSpinBox)
```

向工具栏中添加各个窗口部件的模式基本一致: 创建要添加的窗口部件, 设置一下, 连接到能够处理用户交互的地方, 最后将其添加到工具栏上。创建微调框实例变量是因为需要在主窗口的初始化程序外访问它。addWidget() 调用会向工具栏传递微调框的拥有权。

现在已经把各个动作都放到了菜单和工具栏上。尽管每个动作都添加到了菜单, 有些却还没有添加到工具栏上。这种做法较为常见; 通常情况下, 只会把那些最为常用的动作添加到工具栏上。

之前曾看到过如下的代码:

```
self.imageLabel.setContextMenuPolicy(Qt.ActionsContextMenu)
```

这就告诉 PyQt, 如果动作添加到了窗口部件中, 就可以在上下文菜单中用到它们, 比如在图 6.7 中给出的这些。

```
self.addAction(self.imageLabel, (editInvertAction,
    editSwapRedAndBlueAction, editUnMirrorAction,
    editMirrorVerticalAction, editMirrorHorizontalAction))
```

如果不传递 None 的话, 也可以再次使用 addAction() 方法向标签窗口部件中添加动作, 这是因为 QWidget 没有 addSeparator() 方法。为一个窗口部件设置策略并向其添加动作都是必要的, 以便可以得到该窗口部件的上下文菜单。

QWidget 类都有一个 addAction() 方法, 可被 QMenu、QMenuBar 和 QToolBar 类所继承。这就是为什么可以向这



图 6.6 编辑(Edit)工具栏



图 6.7 图片标签的上下文菜单

些类添加动作的原因。尽管 QWidget 类没有 addSeparator() 方法，但给予方便的原则，在 QMenu、QMenuBar 和 QToolBar 中还是提供了 addSeparator() 方法。如果打算在上下文菜单中添加一个分隔符，就必须通过添加分隔符动作来完成。例如：

```
separator = QAction(self)
separator.setSeparator(True)
self.addAction(editToolbar, (editInvertAction,
    editSwapRedAndBlueAction, separator, editUnMirrorAction,
    editMirrorVerticalAction, editMirrorHorizontalAction))
```

如果需要处理更为复杂的上下文菜单，例如，菜单的动作可根据应用程序状态的不同而有所不同，那么就可以重新实现相应窗口部件的 contextMenuEvent() 事件处理方法。第 10 章会讲述事件处理。

在创建新图片或者加载已有图片的时候，希望用户接口能够恢复到初始状态。特别是，希望“编辑→撤销”和“编辑→红蓝交换”动作能够恢复为“关”，镜像动作可以恢复为“编辑→未镜像”。

```
self.resetableActions = ((editInvertAction, False),
    (editSwapRedAndBlueAction, False),
    (editUnMirrorAction, True))
```

我们已经创建了可保存元组对的实例变量，当创建新图片或者加载图片时，每个元组对都应当保存有一个动作以及图片的选中状态。在查看 fileNew() 和 loadFile() 槽时，先来看看所使用的 resetableActions。

在图片转换应用程序中，任何时候都会使所有动作可用。这样做很好，因为在执行任何动作之前总是会先检查图片是否为空，但也有不好的一点是，例如，在图片为空或者图片还没有发生任何修改时，“文件→保存”会可用，与之相似的是，即使没有任何图片，编辑动作也是可用的。解决这一问题的方法是，根据应用程序的状态来决定各个动作是可用还是不可用，就像在第 13 章中给出的那样。

#### 6.1.4 恢复和保存主窗口的状态

现在，主窗口的用户接口已经设置好了，现在为完成初始化程序方法几乎也做好准备了，但在开始之前，需要恢复应用程序从前一次运行的设置（或者恰好是应用程序第一次运行的话，就使用其默认设置）。

在查看应用程序设置之前，但还必须要绕个小弯，先来看看应用程序对象的创建过程以及到底主窗口自己是如何创建的。在 imagechanger.pyw 文件中最后一个可执行语句仅仅就是一个函数调用：

```
main()
```

如往常一样，需要先给要执行的第一个函数选一个常用的名字。以下就是代码：

```
def main():
    app = QApplication(sys.argv)
    app.setOrganizationName("Qtrac Ltd.")
    app.setOrganizationDomain("qtrac.eu")
    app.setApplicationName("Image Changer")
    app.setWindowIcon(QIcon(":/icon.png"))
    form = MainWindow()
    form.show()
    app.exec_()
```

函数的第一行就是我们之前已经多次见过的内容。后面的三行是新的。使用它们的目的是为

了加载和保存应用程序的设置。如果创建 `QSettings` 对象但并未传递任何参数, 那么就会使用组织名字或者域的名字(取决于所在的平台)以及所设置的应用程序的名字。所以, 通过对应用程序对象实施一次这些设置, 那么无论何时需要 `QSettings` 实例, 都不再需要向其传递这些参数了。

但这些名字意味着什么呢? PyQt 用它们会在合适的地方保存应用程序的设置, 例如, 保存在 Windows 的注册表中, 或者保存在 Linux 系统下的 \$ HOME/. config 目录中, 又或者是在 Mac OS X 系统下的 \$ HOME/Library/Preferences 目录中。注册表键值或者文件和目录的名字会从应用程序对象的名字中派生出来。

可以让图标文件从 `qrc_resources` 模块加载, 因为其目录的开头是带有“:/”的。

在设置了应用程序对象之后, 可以创建和显示主窗口, 开始事件循环, 这一方式就像在之前其他章节中所做的那样。

现在, 可以返回到之前开始的 `MainWindow.__init__()` 方法中, 看看是如何恢复系统设置的。

```
settings = QSettings()
self.recentFiles = settings.value("RecentFiles").toStringList()
size = settings.value("MainWindow/Size",
                      QVariant(QSize(600, 500))).toSize()
self.resize(size)
position = settings.value("MainWindow/Position",
                          QVariant(QPoint(0, 0))).toPoint()
self.move(position)
self.restoreState(
    settings.value("MainWindow/State").toByteArray())
self.setWindowTitle("Image Changer")
self.updateFileMenu()
 QTimer.singleShot(0, self.loadInitialFile)
```

一开始会创建一个 `QSettings` 对象。由于没有传递参数, 应用程序对象所保存的名字会用来定位各个设置信息。先从最近用过的文件清单的检索开始。`QSettings.value()` 方法总会返回 `QVariant` 型结果, 所以必须将其转换成所需的数据类型。

接下来, 使用双参数的 `value()`, 其中, 第二个参数是一个默认值。这就意味着, 在应用程序第一次运行的时候, 根本就没有任何设置, 所以会得到一个 `QSize()` 对象, 其宽度为 600 像素, 高度为 500 像素<sup>①</sup>。在接下来的后续运行中, 返回的尺寸就会变成应用程序结束时的主窗口的尺寸大小——只要应用程序一结束, 主窗口的尺寸大小就会保存下来。一旦有了尺寸大小, 就可以将主窗口重置为给定的尺寸。在获得前一次(或者默认的)尺寸大小后, 就可以采用完全一样的方式来获得并设置位置的值。

没有出现闪屏现象是因为, 重置尺寸大小和位置都是在主窗口的初始化程序中完成的, 也就是在真正显示给用户之前完成的。

Qt 4.2 中为保存和恢复顶层窗口的几何形状引入了两个新的 `QWidget` 方法。遗憾的是, 有 bug 表明在 X11 系统下它们还不够十分稳定, 基于此, 这里会把窗口的尺寸大小和位置作为单独的元素予以存储。Qt 4.3 中则修正了这一 bug, 所以在 Qt 4.3 中(比如, PyQt 4.3), 不需要通过调用 `resize()` 和 `move()` 来检索尺寸大小和位置, 用以下一行代码就都可以完成了:

<sup>①</sup> PyQt 的文档很少给定测量所用的单位, 因为除了 `QPrinter` 使用的是点之外, 其他的测量单位都会假设是像素。

```
self.restoreGeometry(settings.value("Geometry").toByteArray())
```

这里假定在应用程序终止时，几何形状都已被保存，正如在 closeEvent () 中将看到的那样。

QMainWindow 类提供了一个 restoreState () 方法和一个 saveState () 方法；这些方法都可以将内容到 QByteArray 中或者从其恢复内容。它们所保存和恢复的数据包括停靠窗口的尺寸大小和位置，以及工具栏的位置等信息——不过，只有对停靠的那些具有唯一对象名的窗口部件和工具栏来说，它们才可以起作用。

在设置了窗口的标题后，可以调用 updateFileMenu () 来创建文件 (File) 的菜单项。与其他菜单不同，文件 (File) 菜单是动态创建的，以便可以显示任何最近用过的那些文件。从文件 (File) 菜单的 aboutToShow () 信号到 updateFileMenu () 方法的连接意味着，文件 (File) 菜单可以根据用户点击菜单栏上的文件 (File) 菜单或者是按下 Alt + F 而做出实时更新。但是直到该方法首次调用，文件 (File) 菜单都不会存在——这就意味着，用于动作的键盘快捷键都还没有添加到工具栏上，比如用于“文件→退出”的 Ctrl + Q 等都无法工作。从这个角度来看，需要明确调用 updateFileMenu () 以便可以创建一个初始的文件 (File) 菜单，激活各键盘快捷键。

### 开始启动时所做的处理

如果需要在启动的时候处理一些事情——例如，需要加载许多较大的文件，通常总会在一个单独的加载方法中来处理这些事情。在主窗体构造函数的最后，会通过一个 0 时单触发定时器来调用这个加载函数。

如果不使用单发计时器会怎么样呢？设想一下，例如，loadInitialFiles () 可以加载数兆字节的大文件。在主窗口创建时文件的加载就会完成，也就是说，会发生在调用 show () 之前，也在事件循环的开始之前 (exec\_ ())。这就意味着，用户或许需要在从应用程序启动一直到看到应用程序的窗口真正出现在屏幕上经历较长的时间迟滞。同时，文件的加载也可能导致弹出消息对话框（例如，报告一些错误）在主窗口显示后，这些东西可能会更让人敏感，而在事件循环开始运行时亦是如此。

我们希望主窗口能够尽可能快地出现，以便用户可以知道加载成功，并可从中看到那些运行时间较长的进程，比如通过主窗口的用户界面看到大文件加载等。通过使用诸如在图片变换应用程序例子中用过的单触发计时器就可以实现这一点。

之所以可以实现，是因为带有超时时间为 0 的单触发计时器并不会立即执行给定的槽。相反，会把要调用的槽放到事件队列中，然后就返回了。在主窗口初始化程序的最后会实现这一点，同时也会完成初始化。紧接着的下一个语句（在 main () 中）是对主窗口调用 show ()，除了向事件队列中添加一个显示事件之外什么都不做。所以，事件队列现在会有一个事件计时器和一个显示事件。带有 0 时间的事件计时器的意思是，“在事件队列中再没有其他事情可做的时候来完成这件事”，所以在下一个语句 exec\_ () 中，就会到达并开始事件循环，通常是首先选择处理显示事件，以便能够让窗体出现，然后，再没有剩下其他事件后，就会处理单触发计时器的事件，也就是去调用 loadInitialFiles ()。

初始化程序的最后一行看起来有些奇怪。一个带有参数为触发时间（以毫秒为单位）的单触发计时器，一个在超出触发时间后要调用的方法。所以，尽管应当那样写，但这一行看起来可能会被写成这样：

```
    self.loadInitialFile()
```

在这个应用程序中，其中会加载至少一个初始化文件，加载的各个文件迥异，大小约为 1MB，可以使用其中的任何一种方法而不会看到有何不同。虽然如此，直接调用该方法与使用带触发时间为 0 的单触发计时器还是不太一样的，这一点会在“开始启动时所做的处理”中予以阐述。

这样就完成了对主窗口初始化代码的查看，所以现在可以开始看看那些为应用程序提供功能而必须实现的其他方法。尽管图片转换应用程序只是一个特定的例子，但已经从最大的可能让代码做到普适或者易于继承，以便可以将它们用到其他主窗口风格的应用程序中，即使有些代码可能会出现完全不同的情况。

从之前的各个讨论内容来看，似乎对 `loadInitialFile()` 方法的说明已经万事俱备只欠东风了。

```
def loadInitialFile(self):
    settings = QSettings()
    fname = unicode(settings.value("LastFile").toString())
    if fname and QFile.exists(fname):
        self.loadFile(fname)
```

这一方法使用 `QSettings` 对象来获取应用程序中的最后一幅图片。如果有这样一幅图片，并且仍旧还在的话，程序就会试着加载它。在查看有关文件动作的内容时会回头看看 `loadFile()` 函数。

本来也可以像之前所用过的 `fname` 和 `os.access(fname, os.F_OK)` 那样轻松使用它们：这里也没有什么值得关注的不同之处，但对于多人协作的项目来说，在这种情况下，较为明智的策略就是在标准 Python 库的基础上多用 PyQt 或者在用 PyQt 的时候多用标准 Python 库，这样就可以尽可能地让事情变得简单和清晰。

前面曾讨论过应用程序状态的恢复，所以貌似用来恢复关闭事件也是合适的，因为关闭事件中会保存应用程序的状态。

```
def closeEvent(self, event):
    if self.okToContinue():
        settings = QSettings()
        filename = QVariant(QString(self.filename)) \
            if self.filename is not None else QVariant()
        settings.setValue("LastFile", filename)
        recentFiles = QVariant(self.recentFiles) \
            if self.recentFiles else QVariant()
        settings.setValue("RecentFiles", recentFiles)
        settings.setValue("MainWindow/Size", QVariant(self.size()))
        settings.setValue("MainWindow/Position",
                          QVariant(self.pos()))
        settings.setValue("MainWindow/State",
                          QVariant(self.saveState()))
    else:
        event.ignore()
```

如果用户试图关闭应用程序，无论使用的是哪种方法（除非是系统终止或者应用程序崩溃），都会调用 `closeEvent()` 方法。先从自定义的 `okToContinue()` 方法开始；如果用户确实想关闭应用程序，该方法就返回 `True`，否则，就返回 `False`。在 `okToContinue()` 的代码里，还要给用户留下保存那些尚未保存的修改的机会。如果用户确实想关闭程序，可以创建一个全新的 `QSettings` 对象并保存“最后一个文件”（如用户已打开的文件）、最近用过的文件以及主窗口的状态等信息。`QSettings` 类只需读取和写入各 `QVariant` 对象，所以必须小心为

各 QVariant 对象要么提供一个空的 QVariant(用 QVariant() 新创建的), 要么提供所需的正确信息。

如果用户选择不关闭, 就对关闭事件调用 ignore()。这样会告诉 PyQt 只需简单舍弃关闭事件并让应用程序继续运行。

如果使用的是 Qt 4.3(比如, 用 PyQt 4.3)且用 QWidget.restoreGeometry() 保存了主窗口的几何形状信息, 就可以像这样来存储这些信息:

```
settings.setValue("Geometry", QVariant(self.saveGeometry()))
```

如果采用这种方法, 就无须分别保存主窗口的尺寸大小或者位置信息了。

```
def okToContinue(self):
    if self.dirty:
        reply = QMessageBox.question(self,
                                      "Image Changer - Unsaved Changes",
                                      "Save unsaved changes?",
                                      QMessageBox.Yes|QMessageBox.No|
                                      QMessageBox.Cancel)
        if reply == QMessageBox.Cancel:
            return False
        elif reply == QMessageBox.Yes:
            self.fileSave()
    return True
```

closeEvent() 会用到这个方法, 也会被“文件→新建”和“文件→打开”动作用到。如果图片是“脏的”(dirty), 也就是说, 如果有尚未保存的变化, 就会弹出一个消息对话框询问用户该怎么办。如果点击了 Yes, 就把图片保存到磁盘上并返回 True。如果用户点了 No, 只需简单地返回 True 即可, 这样那些未保存的变化就会被丢弃。如果用户点击了 Cancel, 就返回 False, 这就意味着不会保存那些修改, 而当前图片仍旧会保持当前状态, 以便可以在后续加以保存。

表 6.2 QMainWindow 的一些方法

语法	说明
m.addDockWidget(a, d)	向主窗口 QMainWindow m 中的 Qt.QDockWidgetArea a 区域添加一个 QDockWidget d
m.addToolBar(s)	添加并返回一个新的 QToolBar, 其名字由字符串 s 给定
m.menuBar()	返回 QMainWindow m 的 QMenuBar(会在第一次调用该方法时创建)
m.restoreGeometry(ba)	恢复封装在 QByteArray ba 中 QMainWindow m 的位置和大小
m.restoreState(ba)	恢复封装在 QByteArray ba 中 QMainWindow m 的停靠窗口部件和工具栏的状态信息
m.saveGeometry()	返回封装在 QByteArray ba 中 QMainWindow m 的位置和大小
m.saveState()	返回 QMainWindow m 的停靠窗口部件和工具栏的状态, 也就是说, 它们的尺寸大小和位置, 会封装在 QByteArray 中
m.setCentralWidget(w)	把 QMainWindow m 的中央窗口部件设置为 QWidget w
m.statusBar()	返回 QMainWindow m 的 QStatusBar(会在第一次调用这个方法的时候创建)
m.setWindowIcon(i)	把 QMainWindow m 的图标设置成 QIcon i, 这一方法继承自 QWidget
m.setWindowTitle(s)	把 QMainWindow m 的标题设置成字符串 s, 这一方法继承自 QWidget

在本书中, 所有的例子都会使用 yes/no 或者 yes/no/cancel 的消息框, 以给用户保存那些尚未保存的修改机会。某些程序开发人员会喜欢另外一种方法, 就是使用保存(Save)和放弃(Discard)按钮(会分别使用 QMessageBox.Save 和 QMessageBox.Discard 按钮)来代替前一种方法。

最近用过的文件清单是应用程序状态的一部分，不仅是应用程序关闭和执行时必须保存与恢复的状态，而且也会保存当前的运行时间。早前，我们创建过 `fileMenu` 的 `aboutToShow()` 信号与自定义 `updateFileMenu()` 的连接。所以，在用户按下 (Alt + F) 或者点击文件 (File) 菜单时，就会在显示文件 (File) 菜单之前先调用该槽。

```
def updateFileMenu(self):
    self.fileMenu.clear()
    self.addActions(self.fileMenu, self.fileMenuActions[:-1])
    current = QString(self.filename) \
        if self.filename is not None else None
    recentFiles = []
    for fname in self.recentFiles:
        if fname != current and QFile.exists(fname):
            recentFiles.append(fname)
    if recentFiles:
        self.fileMenu.addSeparator()
        for i, fname in enumerate(recentFiles):
            action = QAction(QIcon(":/icon.png"), "&%d %s" % (
                i + 1, QFileInfo(fname).fileName()), self)
            action.setData(QVariant(fname))
            self.connect(action, SIGNAL("triggered()"),
                         self.loadFile)
            self.fileMenu.addAction(action)
        self.fileMenu.addSeparator()
        self.fileMenu.addAction(self.fileMenuActions[-1])
```

先从清空文件 (File) 菜单的所有动作开始。然后，会在最后添加文件菜单动作的初始清单，比如“文件→新建”和“文件→打开”，不过不会添加最后一个，“文件→退出”。接着，会遍历最近用过的文件清单，创建一个只包含在本地文件系统中仍旧存在的一个本地文件清单，但会去除当前文件。尽管这样做好像无关紧要，许多应用程序也会包含有当前文件，通常将其显示在文件清单的第一个。

### QMessageBox 的静态方法

`QMessageBox` 类会提供数个用来弹出带有适当图标和按钮模态对话框的静态简便方法。这些方法在为用户提供仅有一个 OK、Yes、No 或者类似按钮的对话框时非常有用。

最为常用的 `QMessageBox` 静态方法有 `critical()`、`information()`、`question()` 和 `warning()`。这些方法会带有一些参数，包括一个父对象窗口部件 (居中分布于该对象上方)、窗口标题文本、消息文本 (可以是普通文本或者 HTML) 以及 0 个或者多个按钮的说明。如果没有指定按钮，就只提供一个 OK 按钮。

使用常量或者提供自己的文本都可以用来指定各个按钮。在 Qt 4.0 和 Qt 4.1 中，较为常见的是让 OR `QMessageBox.Default` 与 OK 按钮或者 Yes 按钮进行按位或 (即 ‘|’ ) 操作——这就意味着，如果用户按下回车 (Enter) 键就会按下相应的按钮；而如果用户按下 Esc 键，就会让 OR `QMessageBox.Escape` 与取消 (Cancel) 按钮或者否 (No) 按钮进行按位或操作。例如：

```
reply = QMessageBox.question(self,
    "Image Changer - Unsaved Changes", "Save unsaved changes?",
    QMessageBox.Yes|QMessageBox.Default,
    QMessageBox.No|QMessageBox.Escape)
```

该方法会返回所按下的按钮常量。

从 Qt 4.2 以来,为了能够不再具体指明这些按钮并不再使用按位或(OR)操作,对 QMessageBox 的 API 做了大量简化,这样就只需使用各个按钮即可。例如,对于一个 yes/no/cancel 型对话框,只要这样写即可:

```
reply = QMessageBox.question(self,
    "Image Changer - Unsaved Changes", "Save unsaved changes?",
    QMessageBox.Yes|QMessageBox.No|QMessageBox.Cancel)
```

在这个例子中,PyQt 将自动让 Yes(接受)按钮成为默认按钮,会在用户按下回车(Enter)键的激活,并让 Cancel(拒绝)按钮成为取消按钮,会在按下 Esc 键时激活它。QMessageBox 方法还可以确保让这些按钮在不同平台下均能够以正确的顺序显示出来。在本书的例子中,会采用 Qt 4.2 的语法样式。

用户按下“接受”(Accept)按钮(通常是 Yes 或者 OK)或者“拒绝”(Reject)按钮(通常是 No 或者 Cancel)都会关闭消息框。实际上,用户也可以通过点击窗口的关闭按钮 X 或者是按下 Esc 键来按下“拒绝”(Reject)按钮。

如果打算创建一个自定义消息框(例如,使用自定义的按钮文本和自定义的图标)可以通过创建一个 QMessageBox 实例来实现。然后就可以使用诸如 QMessageBox.addButton() 和 QMessageBox.setIcon() 这样的方法,也可以通过调用 QMessageBox.exec\_() 来弹出消息框。

现在,如果在本地文件清单中还有任何最近用过的文件,那么就在菜单栏上添加一个分隔符,然后用只含有文件名(去掉文件目录)的普通文本为每个文件都创建一个动作,并在前面添加一个数字快捷键: 1, 2, …, 9。PyQt 的 QFileInfo 类会提供一些作用于文件的信息,这些信息与 Python 的 os 模块的函数所能提供的功能相似。QFileInfo.fileName() 方法与 os.path.basename() 相当。对于每个动作,还会保存一个“用户数据”项——这里会保存包含文件所在目录的文件全名。最后,把每个最近用过的文件名的动作的 triggered() 信号连接到 loadFile() 槽,再把动作添加到菜单上(在下一节会讲到 loadFile())。在文件的最后地方,另外添加一个分隔符,再添加一个文件(File)菜单的最后一个动作,“文件→退出”。

不过,到底最近用过的文件清单是如何创建和维护的呢?那么,来看一下窗体的初始化程序,其中会对应用程序设置中的 recentFiles 字符串清单进行初始化。可以看到,该清单也会相应地存储在 closeEvent() 中。使用 addRecentFile() 可以把新的文件添加到清单中。

```
def addRecentFile(self, fname):
    if fname is None:
        return
    if not self.recentFiles.contains(fname):
        self.recentFiles.prepend(QString(fname))
        while self.recentFiles.count() > 9:
            self.recentFiles.takeLast()
```

这个方法会预加载给定的文件名,然后从清单的最后移除多出来的文件(最早被加进来的那些文件),以便可以让文件清单中永远都不会超出 9 个文件名。把 recentFiles 变量保存成 QStringList,这就是为什么要对它应用 QStringList 方法而不用 Python 的 list 方法的原因。

addRecentFile() 方法自己会在 fileNew()、fileSaveAs() 和 loadFile() 方法中得到调用,并且也会在 loadInitialFile()、fileOpen() 和 updateFileMenu() 中间调用,这些方法

都会调用或者连接到 `loadFile()`。所以，当第一次保存图片，或者新建了一个文件名，或者创建了一个新图片，又或者是打开了一个已有的图片的时候，都会将文件名添加到最近使用的文件清单中。然而，最新添加的文件名则不会出现在文件(File)菜单中，除非后来创建或者打开了另一个图片，因为 `updateFileMenu()` 方法不会在最近用过的文件清单中显示当前图片的文件名。

这里用于处理最近用过的文件的方法仅是很多处理方法中的一种。另外一种方法是，只创建文件(File)菜单一次，再在最近用过的文件的最后添加一些动作。当更新菜单时，不是清空和重建该菜单，而只是简单地用这些动作隐藏或者显示那些最近用过的文件，此时，会通过文件名的更新来反映最近用过的文件的当前集合。从用户的角度看，不管使用的是哪种方法好像都没什么两样，所以无论哪种情况，文件(File)菜单看起来都会是如图 6.8 所示的样子。

这两种方法都可以用来实现文件(File)菜单中最近用过的文件的功能，在图片转换应用程序中曾在最后加入文件清单，这就像刚才加入 `Quit` 选项那样。它们也都可以用来实现文件(File)菜单中“打开最近的文件”的选项，其中会把所有最近的文件以子菜单的形式展示出来，这在 OpenOffice.org 之类的应用程序中较为常见。在“打开最近的文件”选项前使用分隔符的好处在于，文件(File)菜单总是一样，但在子菜单中却可以给出文件全目录——某种程度上，没有把最近用过的文件直接放到文件(File)菜单是为了避免让其变得过宽(因此，看起来也就会很丑)。



图 6.8 文件(File)菜单中最近用过的文件

## 6.2 用户动作的处理

在上一节，我们创建了主窗口型应用程序的外观并通过创建一系列动作作为应用程序提供了基本的功能行为。还看到了是如何保存和恢复应用程序的设置，以及是如何管理最近用过的文件清单的。

应用程序的一些行为是由 PyQt 自动处理的——例如，窗口的最小化、最大化和尺寸大小的调整等——所以，诸如这些就无须亲自处理了。另外一些行为完全可以通过信号和槽连接来实现。在这一节，会重点关注那些用户已经控制的动作，以及有哪些动作是可以用来查看、编辑和输出数据的。

### 6.2.1 文件动作的处理

文件(File)菜单可能是主窗口型应用程序中实现最多的，在绝大多数应用程序中，都会至少提供“新建”(New)、“保存”(Save)和“退出”(Quit)(或者 Exit)这样的选项。

```
def fileNew(self):
    if not self.okToContinue():
        return
    dialog = newimagedlg.NewImageDlg(self)
    if dialog.exec_():
        self.addRecentFile(self.filename)
        self.image = QImage()
        for action, check in self.resetableActions:
            action.setChecked(check)
        self.image = dialog.image()
```

```

self.filename = None
self.dirty = True
self.showImage()
self.sizeLabel.setText("%d x %d" % (self.image.width(),
                                      self.image.height()))
self.updateStatus("Created new image")

```

当用户要求处理新文件时，先会看看是否“没问题，可以继续”。这样就为用户保存或者取消保存那些未保存修改的机会，或者，也给了他们完全改变原有想法并取消动作的机会。

如果用户继续，就弹出一个模态对话框 NewImageDlg，其中会要求给定预创建图片的尺寸、颜色、图片画笔模式等。这个对话框如图 6.9 所示，其创建和使用的方法就像在上一章中所做的那样。然而，新建图片对话框的用户界面是通过 Qt 设计师 (Qt Designer) 创建的，所以要想让该对话框文件可用，就必须得用 pyuic4 把用户界面文件转换成模块文件。通过运行 pyuic4，或者通过运行 mkpyqt.py 或 Make PyQt，都可以直接完成转换工作，无论采用这两类方法中的哪一个都可以比较容易地自动获得正确的命令行参数。所有相关内容会在下一章讲到。

如果用户接受对话框，就把当前文件名（如果有的话）添加到最近用过的文件清单中。然后，把当前图片设置成空白图片，以确保对复选动作的任何变化都不会影响该图片。接下来，继续遍历各个在新建或者加载图片时而需要重置的动作，把它们每个都设置成较为喜欢的默认值。现在，可以较为安全地为对话框所创建的图片进行设置了。

把文件名设置成 None，把 dirty 标志设置成 True，以确保用户在终止应用程序或者试图创建、加载另一幅图片时，能够提示用户保存该图片，并要求为该图片设置一个文件名。

然后会调用 showImage()，这会显示 imageLabel 中的图片，图片可根据缩放因子进行缩放操作。最后，调用 updateStatus() 来更新状态栏中的尺寸大小标签。

```

def updateStatus(self, message):
    self.statusBar().showMessage(message, 5000)
    self.listWidget.addItem(message)
    if self.filename is not None:
        self.setWindowTitle("Image Changer - %s[*]" % \
                            os.path.basename(self.filename))
    elif not self.image.isNull():
        self.setWindowTitle("Image Changer - Unnamed[*]")
    else:
        self.setWindowTitle("Image Changer[*]")
    self.setWindowModified(self.dirty)

```

先从显示已经传递的消息开始，会带一个 5 秒钟的超时参数。还会向日志窗口部件添加已经发生的每一个动作。

如果用户打开了已有文件，或者保存了当前文件，都将需要有一个文件名。会用 Python 的 os.path.basename() 函数把不带文件目录的文件名放到窗口的标题栏上。本来也可以像之前所做的那样，只是简单地使用 QFileInfo(fname).fileName() 来代替这里的用法。

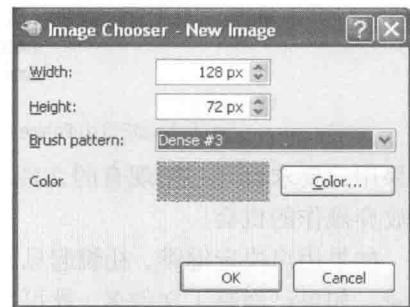


图 6.9 新建图片对话框

的。如果没有文件名且 `image` 变量也不是空的，那么就意味着用户已经创建了一个新图片，只是还没有保存；所以，就可以用一个假文件来暂时代替它，“Unnamed”，即未命名。最后一种情况适用于文件没有打开或者创建。

无论会把窗口标题设置成什么，都可以在标题栏上用一个“[ \* ]”字符。该字符不会原样显示：相反，它只是用来说明文件是否是“脏”的（即有未保存的修改）。在 Linux 和 Windows 系统上，这样做的意思是，该文件不存在未保存的修改，文件名是原样显示，否则，就会在文件名后加( \* )符号来代替“[ \* ]”的字符串。在 Mac OS X 上，如果有未保存的修改，那么会在关闭按钮上带一个圆点。这一机制取决于窗口中改变的状态，所以要确保把状态设置到了 `dirty` 标志上。

```
def fileOpen(self):
    if not self.okToContinue():
        return
    dir = os.path.dirname(self.filename) \
        if self.filename is not None else "."
    formats = ["*.%s" % unicode(format).lower() \
        for format in QImageReader.supportedImageFormats()]
    fname = unicode(QFileDialog.getOpenFileName(self,
                                                "Image Changer - Choose Image", dir,
                                                "Image files (%s)" % " ".join(formats)))
    if fname:
        self.loadFile(fname)
```

如果用户要求打开一个现有的文件，首先要确保用户对未保存修改的保存、放弃保存甚至是完全放弃操作的机会。

如果用户决定继续，礼貌起见，希望能够弹出一个打开文件的对话框，以便能够方便选择目录。如果已经有了文件名，就可以使用它的目录；否则，就使用当前目录“.”。还要选择给文件对话框传入一个用来限制打开文件类型的过滤字符串。该字符串允许对一种文件类型或者多个文件类型同时给定多个扩展符。例如，对于一个文本编辑器来说，可能就会传递这样的一个字符串：

```
"Text files (*.txt)\nHTML files (*.htm *.html)"
```

如果没有更多文件类型，必须用换行符来区分它们。如果一个文件类型可以处理多个文件类型扩展符，就必须用空格来区分这些文件类型扩展符。这里给出的字符串在显示的时候会生成一个组合框，其中有两项，“文本文件”（Text files）和“HTML 文件”（HTML files），这样就可以用来确保在对话框中只有文件类型扩展符为`.txt`、`.htm` 或者 `.html` 的文件才会显示。

在图片转换应用程序的例子中，会使用图片文件类型扩展符来作为该应用程序所用 PyQt 版本所能读取的图片类型。至少，应当会包括`.bmp`、`.jpg`（以及与`.jpeg`一样的`.peg`）和`.png`。该类型清单会遍历各个可读图片文件类型扩展名，创建一个形如“`* .bmp`”、“`* .jpg`”等的字符串列表；这些字符串会用空格分割的形式连接在一起，通过字符串操作函数“`join()`”形成一个单一的字符串。

`QFileDialog.getOpenFileName()`方法会返回一个 `QString`，其中要么保存着文件名（带有全路径），要么是空的（如果用户取消了）。如果用户选择了文件名，那么就调用 `loadFile()` 加载它。

这里，也是整个程序，在需要应用程序名时只是简单地写过。不过，为简化 `QSettings` 的使用，在 `main()` 中已经设置过应用程序对象的名字，无论何时需要，都可以检索该名字即

可。在这里，那么相应的代码应该是这样：

```
fname = unicode(QFileDialog.getOpenFileName(self,
                                             "%s - Choose Image" % QApplication.applicationName(),
                                             dir, "Image files (%s)" % ".join(formats)))
```

应用程序名的使用频率确实惊人。在不超过 500 行的 `imagechanger.pyw` 文件中，应用程序名就用到了数十次。一些应用程序开发人员更喜欢使用可以确保前后一致的调用方法。在第 17 章，涉及国际化的时候，还会进一步讨论到字符串的处理。

如果用户打开文件，会调用 `loadFile()` 方法来真正执行加载操作。这里会分两部分来查看这个方法。

```
def loadFile(self, fname=None):
    if fname is None:
        action = self.sender()
        if isinstance(action, QAction):
            fname = unicode(action.data().toString())
            if not self.okToContinue():
                return
    else:
        return
```

如果该方法是从 `fileOpen()` 方法或者 `loadInitialFile()` 方法中调用的，会传递要打开的文件名。不过，如果是从最近用过的文件动作中调用的，就不会传递文件名。可以使用这一个差异来区分这两种情况。如果调用的是最近用过的文件动作，那么就查找发送对象。这样得到的应该是一个 `QAction`，不过检查应该是安全的，然后提取动作的用户数据，其中会保存最近用过的文件的全名，包括文件所在的路径。由于是以 `QVariant` 的形式保存用户数据的，所以必须将其转换成适当的类型。此时，需要检查是否仍旧可以继续。在“文件→打开”中不必做这样的测试，这是因为，在询问用户要打开的文件名之前就已经进行了测试。所以，现在，如果该方法还没有返回，就知道在 `fname` 中有个必须要试着加载的文件了。

```
if fname:
    self.filename = None
    image = QImage(fname)
if image.isNull():
    message = "Failed to read %s" % fname
else:
    self.addRecentFile(fname)
    self.image = QImage()
    for action, check in self.resetableActions:
        action.setChecked(check)
    self.image = image
    self.filename = fname
    self.showImage()
    self.dirty = False
    self.sizeLabel.setText("%d x %d" %
                           image.width(), image.height())
    message = "Loaded %s" % os.path.basename(fname)
self.updateStatus(message)
```

首先，从把当前文件名置为 `None` 开始，然后试着把图片读取到本地变量中。PyQt 没有使用异常处理机制，所以必须用非直接的方式来查找各个错误。在这种情况下，一个空图片就意味着由于某种原因图片加载失败了。如果加载成功，就可以把新的文件名添加到最近用过的文件

清单中，不过，只有当打开了另一个后续文件，或者当前文件被保存成了另一个文件名的时候，才会把当前这个文件的名字显示出来。然后，把图片变量实例设置为空图片；这就意味着，可以将复选动作任意设置为喜欢的默认值而不会产生任何副作用。这样做是可行的，因为在改变复选动作的时候，尽管根据信号-槽连接机制会调用相关方法，但如果图片为空，本方法仍旧是什么也不做。

初步完成后，可以把本地图片赋值给图片变量实例，把本地文件名赋值给文件名变量实例。接下来，调用 `showImage()` 以当前缩放因子显示图片，清空 `dirty` 标志，更新尺寸大小标签。最后，调用 `updateStatus()` 在状态栏中显示消息，更新日志窗口部件。

```
def fileSave(self):
    if self.image.isNull():
        return
    if self.filename is None:
        self.fileSaveAs()
    else:
        if self.image.save(self.filename, None):
            self.updateStatus("Saved as %s" % self.filename)
            self.dirty = False
        else:
            self.updateStatus("Failed to save %s" % self.filename)
```

`fileSave()` 方法和其他许多方法会作用于应用程序的数据 (`QImage` 实例) 上，不过如果没有任何图片数据，则不起任何作用。基于这一原因，许多方法在没有图片数据时不会做任何事，也不会立即返回结果。

如果有图片数据，文件名也不是 `None`，用户一定会调用“文件→新建”动作，就会第一次保存他们的图片。对于这种情况，可以把工作传给 `fileSaveAs()` 方法。

如果有文件名，可以用 `QImage.save()` 来试着保存图片。这一方法会返回一个布尔型 `success/failure` 标志，以分别作为更新的是哪种状态的响应(这里把有关加载和保存自定义文件格式的内容放到了第 8 章，这是因为在该章，我们主要是希望能够重点关注主窗口的功能)。

```
def fileSaveAs(self):
    if self.image.isNull():
        return
    fname = self.filename if self.filename is not None else "."
    formats = ["*.%s" % unicode(format).lower() \
               for format in QImageWriter.supportedImageFormats()]
    fname = unicode(QFileDialog.getSaveFileName(self,
                                                "Image Changer - Save Image", fname,
                                                "Image files (%s)" % " ".join(formats)))
    if fname:
        if "." not in fname:
            fname += ".png"
        self.addRecentFile(fname)
        self.filename = fname
        self.fileSave()
```

在激活“文件→另存为”时，就会开始检索当前文件名。如果文件名是 `None`，可将其设置为“`.`”，即当前目录。然后就可以使用 `QFileDialog.getSaveFileName()` 对话框来提醒用户输入一个用来保存的文件名了。如果当前文件名不是 `None`，就可以使用默认文件名——如果用户选择的名字与一个已经存在的文件名相同，文件保存对话框就会注意到这一点并给出 yes/no 型警告对话框。对于“文件→”打开动作，本来也是可以在设置文件过滤字符串时使用

相同的技术，但这一次会使用 PyQt 可写版本的图片格式清单（这可能与 PyQt 所能读取的图片格式会有所不同）。

如果用户输入不包含句点的文件名，即没有扩展名，那么就把扩展名设置为.png。然后，把文件名添加到最近用过的文件清单中（以便可后续又打开了一个不同的文件，或者是把当前文件另存为新的名字时，能够将其显示出来），把文件名变量实例设置为该名字，把保存工作传递给刚刚看过的 fileSave() 方法。

文件动作中最后一个需要考虑到的就是“文件→打印”。在调用 filePrint() 方法时会激活这个动作。这个方法会在打印机上绘制该图片。由于这个方法使用了一些我们还未涉及的技术，所以会在后面再进一步深入讲述。所用到的技术会在第 13 章的“打印图片”一节中讲述，也会涉及方法，那里还会探讨打印文档时的一些常规方法。

文件动作中唯一尚未看到的就是“文件→退出”动作。这一个动作与主窗口的 close() 方法相连，因而会在事件队列中添加一个关闭事件。这里会重新实现 closeEvent() 的处理程序，以便可以确保用户能够有机会保存那些尚未保存的修改，用到对 okToContinue() 的调用，其中会保存应用程序的设置。

## 6.2.2 编辑动作的处理

文件动作中的大多数功能都由子类自己提供。传过来唯一要处理的工作就是图片的加载和保存，这些是 QImage 变量实例要求做的事情。这种特殊的在主窗口和保存数据的数据结构之间进行职责分离的做法非常常见。主窗口处理高级的文件新建、打开、保存和最近用过的文件等功能，而数据结构则用来处理加载和保存。

无论是通过视图窗口部件还是通过数据结构，对于要提供的大多数甚至全部的编辑功能来说，这样做也是常见的。在图片转换应用程序中，会通过数据结构（image QImage）处理所有的数据操作，会利用数据查看器（imageLabel QLabel）来显示处理过的数据。此外，这是一种非常常见的职责分离做法。

在这一节，将会查看大多数的编辑动作，但会忽略一些与已给出的几乎完全一样的编辑动作。这里的介绍会非常简要，因为对于图片转换应用程序来说，这些功能都是非常具体的。

```
def editInvert(self, on):
    if self.image.isNull():
        return
    self.image.invertPixels()
    self.showImage()
    self.dirty = True
    self.updateStatus("Inverted" if on else "Uninverted")
```

如果用户调用“编辑→反转”动作，就会选中（或者不选）该动作。无论是哪一种情况，只需使用由 QImage 提供的功能来反转图片的像素，显示变化后的图片，设置 dirty 标志，调用 updateStatus() 在状态栏中简要显示执行过的动作等，此外，还会向日志中添加额外的元素。

editSwapRedAndBlue() 方法与此相同（故这里没有给出），只是它用的是 QImage.rgbSwapped() 方法，且它的状态文字也不大一样。

```
def editMirrorHorizontal(self, on):
    if self.image.isNull():
        return
    self.image = self.image.mirrored(True, False)
    self.showImage()
```

```

self.mirroredhorizontally = not self.mirroredhorizontally
self.dirty = True
self.updateStatus("Mirrored Horizontally" \
    if on else "Unmirrored Horizontally")

```

这一方法在结构上与 `editInvert()` 和 `editSwapRedAndBlue()` 相同。`QImage.mirrored()` 方法带有两个布尔型标志，第一个用于水平镜像，第二个用于竖直镜像。在图片转换应用程序中，故意对所允许的镜像进行了限制，所以用户仅能够无镜像、竖直镜像或者水平镜像，但却不能是既竖直又水平的镜像。还保留了一个可追踪图片是否水平镜像的实例变量。

这里没有给出 `editMirrorVertical()` 方法，因为实质上是完全一样的。

```

def editUnMirror(self, on):
    if self.image.isNull():
        return
    if self.mirroredhorizontally:
        self.editMirrorHorizontal(False)
    if self.mirroredvertically:
        self.editMirrorVertical(False)

```

这个方法会关闭正在应用的镜像，或者在没有镜像图片时什么也不做。它不会设置 `dirty` 标志，也不会更新状态：如果调用了它们中的任何一个，这些动作就会留给 `editMirrorHorizontal()` 或者 `editMirrorVertical()`。

应用程序提供了两种让用户改变缩放因子的方式。用户可以与工具栏中的缩放微调框进行交互——它的 `valueChanged()` 信号连接到了 `showImage()` 槽，该槽很快会看到——或者可以调用编辑(Edit)菜单中的“编辑→缩放”动作。如果使用“编辑→缩放”动作，就会调用 `editZoom()` 方法。

```

def editZoom(self):
    if self.image.isNull():
        return
    percent, ok = QInputDialog.getInteger(self,
        "Image Changer - Zoom", "Percent:",
        self.zoomSpinBox.value(), 1, 400)
    if ok:
        self.zoomSpinBox.setValue(percent)

```

先从用 `QInputDialog` 类的一个静态方法获取缩放因子开始。`getInteger()` 方法带有一个父对象(对话框会居中显示于其上方)、一个标题、一些描述所需数据的文本、一个初始值以及可选的最小值和最大值。

`QInputDialog` 提供一些静态简便方法，包括获得浮点数的 `getDouble()` 方法，从字符串清单中选择字符串的 `getItem()` 以及获取字符串文本内容的 `getText()`。对于所有这些静态简便方法，返回值都是一个二元组，包含一个返回值和一个用来说明用户是否输入并接受了有效值的布尔变量。

如果用户点击 OK，可以把缩放微调框的值设置为给定的整型数。如果该值与当前值不同，微调框就会发射一个 `valueChanged()` 信号。这个信号连接到了 `showImage()` 槽，所以如果用户选择的是一个新的缩放比例值，那么将会调用该槽。

```

def showImage(self, percent=None):
    if self.image.isNull():
        return
    if percent is None:
        percent = self.zoomSpinBox.value()

```

```

factor = percent / 100.0
width = self.image.width() * factor
height = self.image.height() * factor
image = self.image.scaled(width, height, Qt.KeepAspectRatio)
self.imageLabel.setPixmap(QPixmap.fromImage(image))

```

当创建或者加载了新的图片时，就会调用这个槽，无论何时应用过几何变换，以响应缩放微调框的 valueChanged() 信号。用户无论何时修改了工具栏缩放微调框的值，都会发射该信号，无论是直接用鼠标修改，还是间接使用在早前描述的“编辑→缩放”动作修改的。

获得缩放比率并将其转换成缩放因子就可以用来产生图片的新宽度和高度。然后，可以把图片缩放为新的尺寸大小并保留原有的纵横比来创建图片的一个副本，设置 `imageLabel` 来显示该图片。这个标签需要 `QPixmap` 型图片，所以可以使用 `QPixmap.fromImage()` 方法把 `QImage` 转换成 `QPixmap`。

值得注意的是，以这种方式来缩放图片不会对原始图片产生任何影响；这种做法纯粹就是一种视图变换，而不是图片编辑。这就是为什么 `dirty` 标志无须设置的原因。

根据 PyQt 的文档说明来看，`QPixmap` 对屏幕显示的图片进行了优化（因而可用于快速绘制），`QImage` 则对图片编辑进行了优化（这就是为什么我们要用 `QImage` 来保存图片数据的原因了）。

### 6.2.3 帮助动作的处理

在创建主窗口的动作时，为每个动作都提供了帮助文字，并将这些文字设置为它们的状态栏文字和工具栏提示的文字。这就意味着，在用户浏览应用程序的菜单系统时，当前高亮显示的菜单栏选项的状态栏文字就会自动显示在状态栏中。与之类似的是，如果用户把鼠标悬浮在工具栏按钮的上方，相应的工具栏提示文字就会显示为工具栏提示。

对于一个像图片转换程序这么小且简单的应用程序来说，状态栏提示和工具栏提示或许完全已经差不多了。虽然如此，我们还是提供了在线帮助系统来显示它们是如何实现的，尽管这些内容会推迟到第 17 章来讲述。

无论是否提供在线帮助，提供一个“关于”对话框还是不错的主意。这样至少可以显示应用程序的版本和版权信息，如图 6.10 所示。

```

def helpAbout(self):
    QMessageBox.about(self, "About Image Changer",
        """<b>Image Changer</b> v %s
        <p>Copyright © 2007 Qtrac Ltd.
        All rights reserved.
        <p>This application can be used to perform
        simple image manipulations.
        <p>Python %s - Qt %s - PyQt %s on %s"""\ %
        __version__, platform.python_version(),
        QT_VERSION_STR, PYQT_VERSION_STR, platform.system()))

```

`QMessageBox.about()` 静态简便函数会弹出一个带有给定标题和文本内容的模态 OK 型消息框。其中的文本内容可以是像这里给出的 HTML 型。如果消息框没有给定窗口图标，那么就会使用应用程序的窗口图标。

我们显示了应用程序的版本号，还有 Python、Qt 和 PyQt 库的版本信息，也会给出所运行的平台的版本信息。库的版本信息对用户来说可能没有什么直接用途，但对于那些为用户提供帮助的支持人员来说，则可能会非常有用。

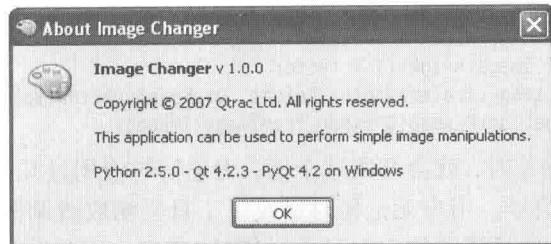


图 6.10 关于图片转换程序的对话框

## 小结

通过对 QMainWindow 的子类化创建了一些主窗口型应用程序。该窗口有一个作为中心窗口部件的简单窗口部件(可能会是一个含有其他窗口部件的复合型窗口部件)。

使用各种动作来向应用程序的用户提供各种功能。这些动作会保留一些作为 QAction 型的对象，它们会有文本(用在菜单中)、图标(用在菜单和工具栏上)、工具栏和状态栏提示信息，还有一些可以在调用的时候连接槽，这些槽会用来执行相应的动作。通常情况下，这些动作都会添加到主窗口的菜单栏，并且大多数都还会添加到工具栏上。为了支持仅使用键盘的用户，我们还为经常用到的动作提供了键盘快捷键，为尽可能快、尽可能方便地使用菜单提供了键盘加速键。

有些动作是复选型的，有些复选型动作的群组可能是互斥的，也就是，任何时候都是有且只有一个是可以选中的。通过设置单一属性，PyQt 支持这些可复选的动作，通过使用一些 QActionGroup 对象，PyQt 也支持互斥的动作群组。

停靠窗口通过使用停靠窗口部件来显示，也可以较为容易地进行创建和设置。任意窗口部件都可以添加到停靠窗口部件和工具栏上，尽管在实际应用中，也只是把一些尺寸较小的或者是像邮箱形状的窗口部件添加到工具栏上。

动作、动作群组以及停靠窗口都必须明确给定一个父对象——例如，主窗口——以便可以确保在适当的时间删除它们。对于应用程序的其他窗口部件和各个 QObject 对象来说，这样做并不是必需的，因为它们要么是由主窗口，要么是由主窗口的任一孩子而拥有的。应用程序剩下的非 QObject 对象可以由 Python 的垃圾收集程序删除掉。

应用程序通常会使用资源文件(一些小的文件，比如图标文件和数据文件)，PyQt 的资源机制使得访问它们非常容易。尽管使用它们的确需要额外的构建步骤，要么使用 PyQt 的 pyrcc4 控制台应用程序，要么使用 mkpyqt.py 或者随本书应用程序所提供的 Make PyQt 程序。

正如像在前一章那样，完全使用代码的形式来创建对话框，也可以像下一章那样使用 Qt 设计师来创建对话框。如果需要在我们的应用程序中包含 Qt 设计师制作的用户界面文件，就像资源文件需要额外的创建步骤一样，可以要么使用 PyQt 的 pyuic4 控制台应用程序，要么再次使用 mkpyqt.py 或者 Make PyQt 程序。

一旦通过设置中心窗口部件并创建菜单、工具栏或许还有停靠窗口等创建了主窗口的可视化界面，就可以重点关注于应用程序各项设置的加载和保存了。在主窗口的初始化程序中通常会加载许多设置，且通常也会在重新实现的 closeEvent() 方法中保存这些设置(也会给用户保存那些为未保存修改的机会)。

如果打算恢复用户的工作空间，加载上一次运行应用程序时已经打开过的各个文件，最好是在主窗口的初始化程序的最后使用单触发计时器来加载这些文件。

大多数的应用程序通常会有个一个数据集和一个或者多个用来显示和编辑这些数据的窗口部件。由于这一章的重心主要是放在主窗口的用户界面的基础知识方面，所以会选用最为简单可行的数据和可视化的窗口部件，不过在后面的章节中，会把重点放到其他方面。

让主窗口关注高级文件的处理、最近用过的文件清单并负责处理数据的加载、保存和编辑的对象来说，都较为常见。

截至目前，已经具有足够的 Python 和 PyQt 知识，可以创建对话框型和主窗口型 GUI 应用程序了。在下一章，将会介绍 Qt 设计师在实际中的应用，这是一个能够用来显著加速对话框开发和维护的应用程序。在第二部分的最后一章，将会探索一些保存和加载自定义文件格式的关键方法，其中会使用到 PyQt 和 Python 库。在第三部分和第四部分，将会更为深入地探索 PyQt，看看事件的处理和自定义窗口部件的创建，例如，更为通用地，学习 PyQt 的模型/视图架构和其他高级特性，包括多线程。

## 练习题

创建如图 6.11 所示的对话框。该对话框的类名应当为 ResizeDlg，其初始化程序应当接受一个初始的宽度值和高度值。对话框应当提供一个称为 result() 的方法，必须返回一个由用户选定的宽度值和高度值二元组。微调框的最小值应当是 4，最大值应当是传入的宽度值(或者高度值)的 4 倍。两个值的显示方式都应当采用右对齐。

修改图片转换应用程序，以便使其可以有一个新的“编辑→重定义尺寸大小”动作。该动作应当出现在编辑(Edit)菜单中(在“编辑→缩放”动作之后)。在 images 子目录中有一个名为 editresize.png 的图标，但需要将其添加到 resources.qrc 文件中。还需要将刚才创建的重定义尺寸大小对话框导入的应用程序中。

应当在 editResize() 槽中使用这个重定义尺寸大小对话框，以便可以让“编辑→重定义尺寸大小”动作可以连接到该槽上。该对话框看起来应该是这样的：

```
form = resizedlg.ResizeDlg(self.image.width(),
                           self.image.height(), self)
if form.exec_():
    width, height = form.result()
```

不像 editZoom() 槽，图片自身应当被修改过，所以如果修改了图片的尺寸大小，那么尺寸大小标签、状态栏、dirty 标志的状态等都应当予以改变。另一方面，如果“新的”尺寸大小与图片的原有尺寸大小一样，就不应该再进行尺寸大小的调整。

重定义尺寸大小对话框可以在 50 行以内编写完成，重定义大小槽应当在 20 行以内，而对于这个新的动作，则只需在主窗口初始化程序的多个地方额外添加一行或者两行代码即可。

本练习题的参考答案在文件 chap06/imagechanger\_ans.pyw 和 chap06/resizedlg.py 中。

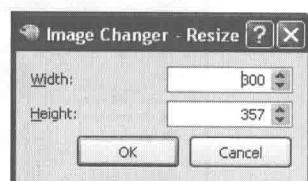


图 6.11 图片转换的重定义尺寸大小对话框

# 第7章 使用 Qt 设计师

- 用户界面的设计
- 对话框的实现
- 对话框的测试

在第5章，我们曾完全用手写代码的方式创建过几个对话框。在初始化程序中，创建了那些需要的窗口部件并为其设置了初始属性。然后，又创建了一个或者多个布局管理器，通过这些窗口部件以获得所需的外观。在那些情况下，当用到数值或者水平布局的时候，曾经添加过一个用来填充不需要空间的“伸展器”(stretch)。在对各个窗口部件进行布局后，又把感兴趣的信号连接到了处理它们的方法上。

一些程序设计人员喜欢用代码来完成所有的事情，然而一些人则更喜欢用可视化的工具来创建自己的对话框。利用 PyQt，就可以选用任何一种方式，也可以两者都用。在上一章的图片转换应用程序中用过两个自定义对话框：一个是 ResizeDlg，纯粹用代码完成的（参见练习题部分），一个是 NewImageDlg，是用 Qt 设计师（Qt Designer）完成的。我们会先给出用纯代码创建的方式，以便能够对布局管理器的工作方式产生较好的认识。不过在这一章，会打算用 Qt 设计师来创建一些对话框，如图 7.1 所示。

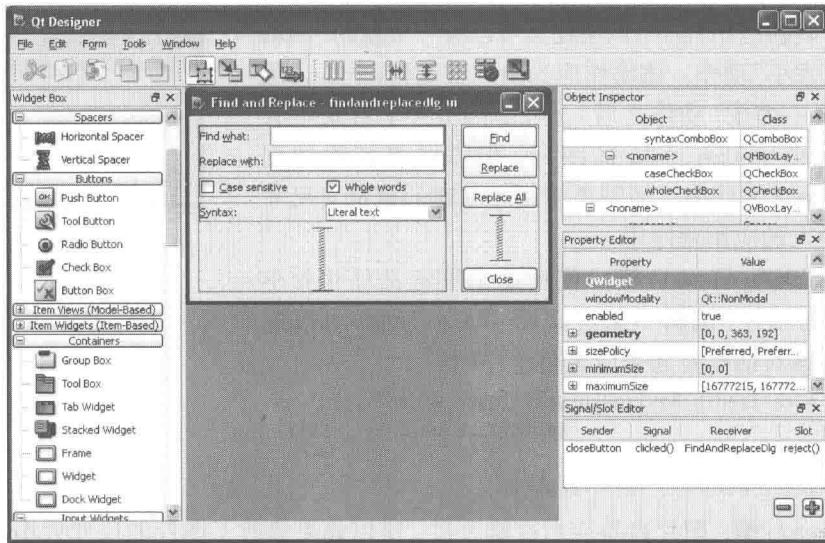


图 7.1 Qt 设计师

Qt 设计师（Qt Designer）可以用来创建对话框、自定义窗口部件、主窗口等用户界面。这里仅会介绍对话框的创建；自定义窗口部件的创建也是一样的，只是它们是基于“窗口部件”模板而没有使用“对话框”模板而已。除了可以方便地使用可视化的 QAction 编辑以外，在主窗口中使用 Qt 设计师提供的好处不多。Qt 设计师也可以用来创建和编辑资源文件（resource file）。

用户界面会保存在.ui文件中，包含了窗体的窗口部件和布局的详细内容。此外，Qt设计师可以在标签和标签的“伙伴”(buddy)之间建立关联，设置Tab键次序，也就是，当用户按下Tab键的时候可以让哪些窗口部件依次获得键盘光标。这也可以用纯代码的QWidget.setTabOrder()完成，但对于手写代码的窗体来说，这样做通常并无必要，因为默认次序会是窗口部件的创建次序，而这通常也正是我们希望的次序。Qt设计器也可以用来建立信号-槽连接，但只能是在内置的信号和槽之间<sup>①</sup>。

一旦设计完成的用户界面保存到.ui文件中，在使用前就必须将其转换成纯代码的形式。使用命令行程序pyuic4可以完成这一过程。例如：

```
C:\pyqt\chap07>pyuic4 -o ui_findandreplacedlg.py findandreplacedlg.ui
```

正如在上一章中提到的那样，可以用mkpyqt.py或者Make PyQt来运行pyuic4。然而，要让界面文件可用，仅仅从一个.ui文件生成一个Python模块(.py文件)还是不够的<sup>②</sup>。值得注意的是，绝不要再去手工编辑所生成的代码(在ui\_\*.py文件中)，因为任何改变都会在再次运行pyuic4的时候被覆盖掉。

从最终用户的角度看，一个对话框的用户界面是用手写代码完成的还是用Qt设计器完成的没有什么两样。然而，在实现一个对话框的初始化程序时却有着显著的不同，因为如果是用手写代码的形式完成的，就必须自己去创建、布局并连接对话框的各个窗口部件，而如果是用Qt设计器完成的用户界面，则只需调用特定的方法就可以达到同样的效果。

使用Qt设计器有一个巨大的好处就是，除了可以方便地可视化设计对话框之外，如果要修改设计结果，只需要重新创建一个用户界面模块即可(可直接使用pyuic4，或者直接借助mkpyqt.py或Make PyQt)，而无须更改代码。而如果需要添加、删除或者重命名一些在代码中要使用的窗口部件，只需花些时间修改一点代码即可。这就意味着，使用Qt设计器会比使用手工编写布局代码更快速、更易于测试设计结果，也有助于实现用Qt设计器进行可视化设计与用代码实现动作行为之间的分离。

在这一章，将会创建一个样例对话框，用它学习如何使用Qt设计器对窗口部件进行创建和布局，设置标签的伙伴并设置Tab键次序，以及完成信号-槽的连接。还将看到，如何使用由pyuic4所生成的用户界面模块，以及如何不必在初始化程序中使用connect()调用就能创建自动连接到自定义槽的方法。

对于这些样例，就曾用过基于Qt4.2早期版本的Qt设计器，它没有QFontComboBox或者QCalendarWidget窗口部件，它的对话框模板(Dialog)使用的是QPushButton而不是QDialogButtonBox。

### mkpyqt.py和Make PyQt

mkpyqt.py控制台应用程序和Make PyQt(makepyqt.pyw)GUI应用程序都可以通过运行PyQt的pyuic4、pyrcc4、pylupdate4和lrelease程序来为我们构建程序。它们既可以做完全一样的工作，又可以自动使用正确的命令行参数来运行PyQt的助手程序，且它们都可以通过检查时间戳来避免做出不必要的工作。

① 某些早期版本的Qt设计器也允许在自定义的信号和槽之间建立连接——译者注。

② 也有可能是可以的，尽管这并不大常见，可以使用PyQt4.uic.loadUi()来直接加载和使用.ui文件。



构建程序会查询.ui 文件并对其运行 pyuic4，产生带有前缀为 ui\_ 的同名但后缀更改为.py 的文件。与之相似的是，也会查找.qrc 文件并对其运行 pyrcc4，产生带有前缀为 qrc\_ 的同名但后缀更改为.py 的文件。

例如，如果在 chap06 目录中运行 mkyqt.py，可以得到：

```
C:\pyqt\chap06>..\mkyqt.py
./newimagedlg.ui -> ./ui_newimagedlg.py
./resources.qrc -> ./qrc_resources.py
```

通过运行 Make PyQt 也可以得到同样的结果：点击 Path 按钮，把目录设置为 C:\pyqt\chap06，然后点击构建 Build 按钮。如果做了任何修改，只需简单地再次运行 mkyqt.py，或者如果使用的是 Make PyQt，只需再次点击构建 Build 按钮，这样就会自动更新那些必要的东西。

这两种构建程序都可以删除生成的各个文件，做好新构建的准备，两者也都可以通过选中 Make PyQt 中的 Recurse 复选框来使用 -r 选项对整个目录树进行递归构建。简单来说，在控制台中也可以运行 mkyqt.py-h。Make PyQt 的各个复选框和按钮都有用法提示。在某些情况下，可能还需要设置各个工具的路径；在第一次使用的时候，可以点击“更多”→“工具路径”(More→Tool paths) 进行设置。

## 7.1 用户界面的设计

在开始之前，必须先启动 Qt 设计师。在 Linux 上，可以在控制台中运行 designer & (假定程序已在启动路径中)，或者从菜单系统调用它。在 Windows XP 中，可以点击 Start→Qt by Trolltech→Designer；在 Mac OS X 上，用 Finder 就可以启动它。Qt 设计师启动时会有打开一个“新窗体”(New Form)对话框；点击“下端带按钮的对话框”(Dialog with Buttons Bottom)，然后再点击“创建”(Create)。这样就可以创建一个带有标题为“未命名”(untitled)的新窗体，图 7.2 给出了 QDialogButtonBox 的示意图。

在 Qt 设计师第一次运行时，它默认为“多顶级窗口”(Multiple Top-Level Windows)模式——除 Mac OS X 用户较为清楚外，可能会让其他用户变得非常困惑。要让所有这些东西都显示为如

图7.1所示的一个窗口，可以点击“编辑→用户界面模式→停靠窗口”(Edit→User Interface Mode→Docked Window)<sup>①</sup>。Qt设计器会记住这一设置值，因而只需一次这样的设置。

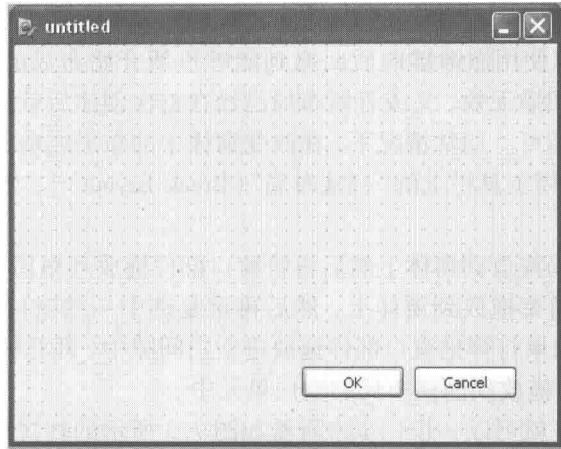


图7.2 下端带按钮的对话框

Qt设计器用起来并不困难，但的确还是需要做些练习的。下面就给出了一些步骤，对正确的开展工作是大有裨益的。对于步骤1和步骤2来说，工作总是从“后”往“前”的，也就是，总会是先从容器(群组框、Tab窗口部件、框架等)开始，然后再对它们所包含的常规窗口部件，即放到它们上面的东西。每次都会从例子的每个步骤开始，不过要知道的是，这里只是对于如何使用Qt设计器创建对话框的简单描述。

1. 拖动一个窗口部件并将其放到窗体上大致正确的位置；无须仔细对它进行调整，通常情况下，只有容器窗口部件才进行尺寸大小的调整。
2. 如果有必要，可以设置窗口部件的属性；如果打算在代码中引用这个窗口部件，那么至少要给它指定一个名字。
3. 重复步骤1和步骤2，直至所需的全部窗口部件都放到了窗体上。
4. 如果还有大的间隙，可以放进一个水平或者竖直分隔符(spacer，看起来好像是蓝色的弹簧)来填充起来；有的时候，在间隙比较明显的时候，会在第1步和第2步中就添加一些分隔符。
5. 选择需要布局的两个或者多个窗口部件(或者分隔符，或者布局；可以在每个上面都先按下Shift键再点击鼠标)，然后再使用布局管理器或者切分窗口(splitter)对它们进行布局。
6. 重复第5步，直至所有的窗口部件和分隔符都布局好为止。
7. 点击窗体(就可以放弃选中的任何东西)并使用布局管理器对其进行布局。
8. 为窗体的标签创建伙伴。
9. 如果按键次序有问题，设置窗体的Tab键次序。
10. 在适当的地方为内置信号和槽建立信号-槽连接。
11. 预览窗体并检查所有东西能否按照设想进行工作。

<sup>①</sup> 从Qt 4.3开始，可以通过点击“编辑→偏好设置”(Edit→Preferences)来获得这一选项的功能。

12. 设置窗体的对象名(会在其类名中用到这个)、窗体的标题并保存它,以便使其拥有一个文件名。例如,如果对象名是“PaymentDlg”,就应当给它的标题命名为“Payment”,文件名为 paymentdlg.ui。

如果布局有错,可以使用撤销键向后回撤到能够重新开始布局的地方,然后再次布局即可。如果无法回撤或者回撤无效,又或者该布局已经在首次创建后就修改过了,只需打乱需要修改的布局并再次布局即可。通常情况下,在改变窗体中的布局之前,很有必要先打乱窗体的布局[点击窗体,然后使用工具栏上的“打乱布局”(Break Layout)];所以,最后必须再次对窗体进行布局。

尽管有可能会将布局拖放到窗体上然后再将窗口部件拖放到布局中,但应用中,最好还是把全部窗口部件和分隔符都拖放到窗体上,然后再重复选中一些窗口部件和分隔符并对其进行布局。这样做的话,如果打算把窗口部件拖放进空白间隙中,即意味着是把窗口部件添加到了已有的布局中,例如,拖放到网格布局的空白单元中。

在有了总体印象后,就可以一步一步来看看如图 7.3 所示的查找和替换对话框了。

基于一种对话框模板(Dialog)来创建一个新的窗体。这样会得到带有一个按钮框的窗体。按钮框中有两个按钮,OK 和 Cancel,并已经为它们设置了相应的信号-槽连接。

点击该按钮框,然后点击 Edit→Delete。这样就可以得到一个空白的窗体。对于这个例子来说,我们会使用 QPushButton 而不是用 QDialogButtonBox。这样在练习的时候,就可以得到比使用 Qt 设计师中的 QDialogButtonBox 更好的控制效果,并为我们提供在 Qt 设计师中创建信号-槽提供了机会。在其他的例子中,以及在练习题中,都会使用 QDialogButtonBox。

默认情况下,Qt 设计师会在左侧有一个称为窗口部件盒(Widget Box)的停靠窗口。其中含有 Qt 设计师所能处理的全部窗口部件。这些窗口部件会分组成不同的部分,向下看到最后会是一个称为显示窗口部件(Display Widgets)的部分;这里就含有标签(Label)的窗口部件(Qt 设计师不会使用类名作为它的窗口部件,至少不会将其用在显示给我们的用户界面中,但几乎在每一种情况下,都会有一个特定的引用名)。

把一个标签(Label)点击并拖拽到窗体上,放到左上角的地方。至于这个标签叫什么我们并不关心,因为不会在代码中引用它,不过它的默认文本“TextLabel”却不是我们想要的。在首次拖放一个窗口部件时会自动选中它,而选中的窗口部件总是会将其各类属性显示在属性编辑器中。切换到属性编辑器(Property Editor)停靠窗口(通常在右侧)并向下滚动到“text”属性。把其中的内容修改为“Find &what:”。这样做并不意味着现在这些显示的文字就会在窗体上被截短;一旦布局该标签,布局管理器就会确保这些文本全部显示出来。

现在,拖放一个 Line Edit 窗口部件(可以从 Input Widgets 部分中找到),并将其放到 Label 的正右侧。切换到属性编辑器,把 Line Edit 的“objectName”(所有窗口部件属性的第一个属性)修改成“findLineEdit”。之所以要给定一个有意义的名字,是因为在代码中还是希望能够引用它的。

现在,在前面那两个窗口部件的下方接着再另外拖放一个 Label 和一个 Line Edit。第二个

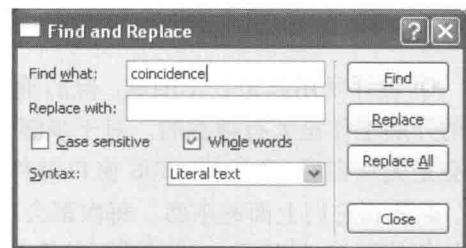


图 7.3 查找和替换对话框

Label的文本应当修改为“Replace w&ith”，第二个Line Edit的文本应当修改为“replaceLineEdit”。窗体现在看起来就与图7.4非常相似了。

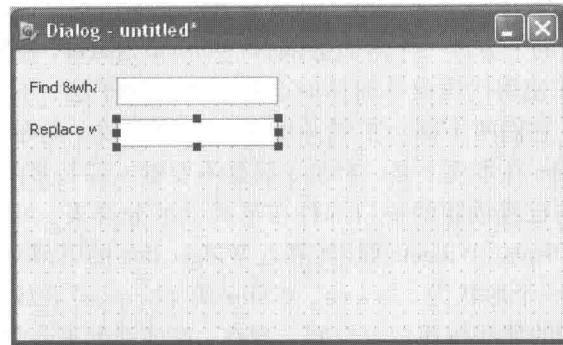


图7.4 两个标签和两个行编辑窗口部件

任何时候，都可以通过按下`Ctrl + S`或者`File→Save`保存该窗体。在保存的时候，会需要文件名`findandreplacedlg.ui`。

每个可编辑属性(以及一些只读属性)都会显示在属性编辑器中。此外，Qt设计器还会提供一个上下文菜单。在上下文菜单中的第一个选项通常会是允许我们修改的最为“重要”的属性(比如，Label或者Line Edit的“text”属性)，而第二个允许修改的选项会是窗口部件的对象名。如果使用上下文菜单来修改复选框、单选框、按钮的文本(text)，那么只需在相应的地方予以编辑即可，此时需要按回车键Enter来完成修改。这里总是说要在属性编辑器中修改各个属性，但如果愿意，也可以在上下文菜单中完成。

现在会添加两个复选框。从按钮群组Buttons group(接近Widget Box的顶部)中拖放一个复选框(Check Box)并反倒第二个标签的下方。使用属性编辑器或者上下文菜单，把它的对象名修改为“caseCheckBox”，把文本(text)修改为“&Case sensitive”。在第一个复选框的右侧再放上第二个复选框：把对象名修改为“wholeCheckBox”，把text修改为“Wh&ole words”，同时把“checked”状态修改为“true”。窗体现在看起来会像是图7.5中的样子了。

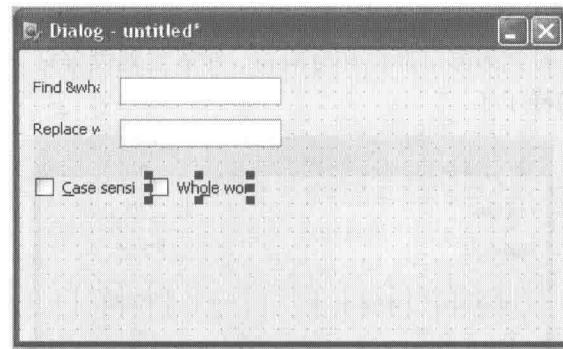


图7.5 两个标签、两个行编辑及两个复选框窗口部件

现在会添加Syntax标签和组合框(ComboBox)。在“Case sensitive”复选框的下方放一个标签，将其text设置成“&Syntax:”。再在Syntax标签的右侧放一个组合框(ComboBox)(在Input Widgets部分可以找到它)。把该组合框的对象名修改为“syntaxComboBox”。在组合框

(Combo Box) 上右击鼠标，选中第一个菜单项 Edit Items。点击“+”图标，键入“Literal text”。重复这些步骤，添加“Regular expression”。点击 OK 按钮完成。

如果用户重新修改了窗体的尺寸大小，如果希望各个窗口部件能够放在一起而不是分散开来，可以在 Combo Box 的下方放一个竖直分隔符(Vertical Spacer，在 Widget Box 顶部的 Spacers 群组中可以找到)。在使用代码设计窗体时会使用伸展因子法，不过在使用可视化方式设计窗体时会用到分隔符：它们两个都会扩展并填充空白的区域。向布局中添加伸展因子与向布局中插入 QSpacerItem 在本质上是一样的，只是不要键入那么多的代码而已。

要让这些按钮能够与已经创建的窗口部件在形式上区分开来，可以在它们和其他窗口部件之间放一个竖线。从 Display Widgets 群组(靠近 Widget Box 的下部)中拖动一根竖直线(Vertical Line，实际上，这是一个形状为 QFrame.VLine 的 QFrame)并放到窗体中所有窗口部件的正右侧，但要给它右侧的那些按钮留点空间。现在，窗体看起来会是图 7.6 中的样子了。

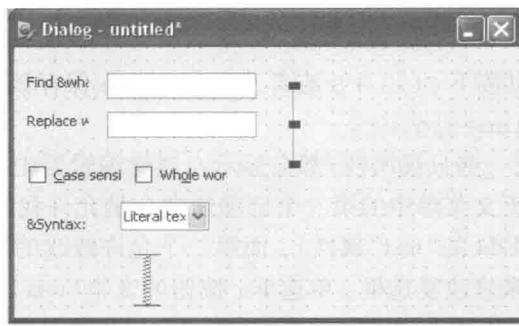


图 7.6 不带任何按钮的查找和替换对话框

现在就为按钮的创建做好准备。在窗体的右上角放一个按钮 Push Button(大约在 Widget Box 上部的 Buttons 群组中)。修改它的属性，把对象名修改为“findButton”，把文本修改为“&Find”。在 Find 按钮的下方另放一个按钮，将其对象名修改为“replaceButton”，把 text 修改为“&Replace”。在 Replace 按钮的下方放上第三个按钮。把对象名修改为“replaceAllButton”，把 text 修改成“Replace &All”。现在，在 Replace All button 的下方放一个竖向分隔符(Vertical Spacer)。最后，在该分隔符的下方放上第四个按钮。该按钮的对象名为“closeButton”，其 text 为“Close”。

这样，就放好了所需的全部窗口部件和分隔符，并适当地设置好它们的所有属性。该窗体现在看起来会是图 7.7 的样子了。

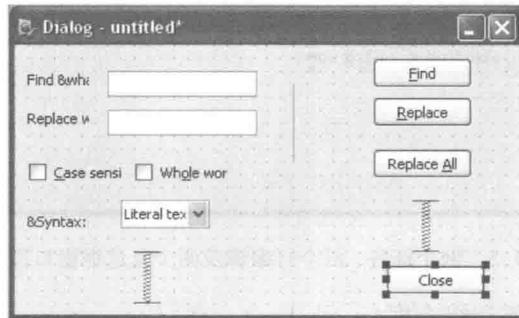


图 7.7 尚未布局的查找和替换对话框

对于该窗体来说，什么样的布局才是最好的布局方式呢？什么才是最好的设计？对于这些问题的回答实际需要考虑一些实际的东西。在这里，我们仅仅会讨论一些机理机制方面的内容，至于美学设计的东西，就留给了读者。

先从前面两个标签和与之对应的行编辑窗口部件开始。点击窗体，不选中任何东西，然后使用Shift键加鼠标点击的方式依次选择各个标签(Label)及其相应的行编辑窗口部件LineEdit，同时选择Replace with标签(Label)及其行编辑窗口部件LineEdit。一旦选中了这4个窗口部件，就可以点击“窗体→网格布局”(Form→Lay Out in a Grid，或者点击工具栏上的相应按钮)。布局会用红色的直线进行标识——在运行的时候，这些红色的布局线是不会显示的。

现在什么都不选择(点击一下窗体)，然后选中两个复选框(Check Box)。点击“窗体→水平布局”(Form→Lay Out Horizontally)。再一次什么都不选，这一次使用水平布局对Syntax标签和ComboBox进行布局。现在应当会有三个布局——一个网格布局，两个水平布局，这样看起来就是图7.8的样子了。

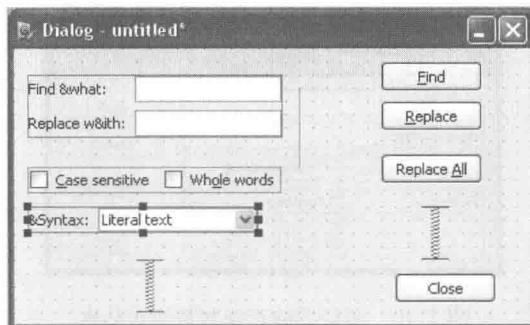


图7.8 带有一些布局的查找和替换对话框

现在可以把布局放到窗体的左侧了。点击窗体，什么都不选。要选中这些布局(而不是选中窗口部件)还是需要些技巧的，但不是用Shift键加鼠标点击的方式选中它们，可以使用选择矩形框。在靠近窗体左下角的地方点击，拖动选择矩形框：这个矩形框不但需要接触要选择的对象，而且还需要向上、向右进行拖拽，以便可以触碰到左手边的竖直分隔符以及其他三个布局——而不需要选择其他东西(如不是竖直线)。选择，释放鼠标，点击“窗体→竖直布局”(Form→Lay Out Vertically)。

也可以使用同样的选择技巧来布局这些按钮。点击窗体，什么也不选。在窗体的右下角点击并拖拽，以便使选择矩形框可以触碰到关闭(Close)按钮、右手边的竖直分隔符和其他三个按钮——而不需选择其他东西。细致，松开鼠标，点击“窗体→竖直布局”(Form→Lay Out Vertically)。现在，就会有一个包含了左侧、右侧布局、中间竖直线中各个窗口部件的窗体，如图7.9所示。

现在算是准备好可以布局窗体了。通过点击窗体，什么也不选。点击“窗体→水平布局”(Form→Lay Out Horizontally)。窗体看起来好像有些太高了，所以只需拖动窗体的底部直到看起来好一些为止。如果拖动得太多了，各个分隔符可能会“消失不见”；其实，它们仍在那里，只是太小以至于无法看见。

现在可以预览一下窗体，以便可以看看布局到底看起来怎么样，且在预览的时候，可以拖动窗体的角来使其变得更大或者更小，以检查窗体的尺寸大小特性是否起作用。要预览窗体，可以点击“窗体→预览”(Form→Preview)(或者按下Ctrl+R)。也可以使用菜单栏中的

“窗体→预览”(Form→Preview)进行预览。窗体此时看起来会像图 7.10 一样。如果无法达到这样的效果，可用“编辑→撤销”(Edit→Undo)回撤所做的修改，然后再次进行布局。如果不得不再次布局，有的时候可能需要对某些窗口部件的位置和尺寸大小进行调整，以便能够为 Qt 设计师提供更多有关设计预期的线索，特别是在使用网格布局的时候。

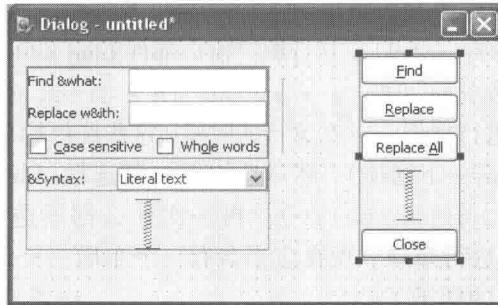


图 7.9 几乎布局好的查找和替换对话框

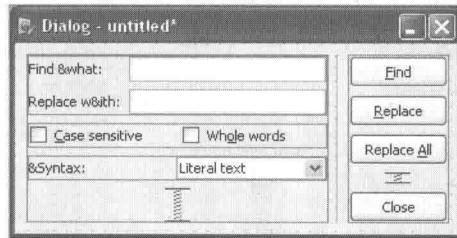


图 7.10 布局后的查找和替换对话框

现在就可以设置标签的伙伴、窗体的 Tab 键次序以及完成所需的任何连接了，最后是对窗体进行命名和保存。

从伙伴设置开始。点击“编辑→编辑伙伴”(Edit→Edit Buddies)切换到伙伴编辑模式。为了设置伙伴关系，可以点击标签(Label)并拖动到打算成为伙伴的窗口部件上。在这个例子中，必须点击 Find what 标签并拖动到行编辑窗口部件“Line Edit”上，然后点击 Replace with 标签和拖动到行编辑窗口部件“Line Edit”上，接着是 Syntax 标签和组合框(Combo Box)。要离开伙伴编辑模式，可以按下 F3 键。现在，各个标签中都不会再看到与字符“&”了。

接下来设置窗体的 Tab 键次序。点击“编辑→编辑 Tab 键次序”(Edit→Edit Tab Order)，然后按照打算设置的按键次序，依次点击每个数字盒。要离开 Tab 键次序模式，可以按下 F3 键。

Find、Replace 和 Replace All 这些按钮需要与我们的自定义方法连接；这些会在 Qt 设计师的外边完成。不过，关闭(Close)按钮可以与对话框的 reject() 槽连接。要实现这一目的，可以点击“编辑→编辑信号/槽”(Edit→Edit Signals/Slots)，然后，从 Close 按钮拖拽到窗体上。在松开鼠标的时候，就会弹出一个 Configure Connection 对话框。从左侧列出的各个信号中点击(无参数的) clicked() 信号，然后再从右侧的槽列表中选择 reject() 槽，接着点击 OK 按钮。要离开信号-槽模式，可以按下 F3 键。

点击窗体，什么也不选。这样做也会产生让属性编辑器显示出窗体各个属性的效果。把对话框的对象名(将会用在它的类名中)设置为“FindAndReplaceDlg”，把“windowTitle”属性设置成“Find and Replace”。点击“文件→保存”(File→Save)保存用户界面，文件名给定为 findandreplacedlg.ui。

在这一节，我们会仅限于用Qt设计器的“Dialog”模板之一来创建自定义对话框，因为这样就足以学到使用Qt设计器所需的基本知识。然而，Qt设计器实际上可以用来创建比在这里任何一个都更复杂的对话框，包括常用的配置对话框，会既带有Tab窗口部件又带有堆窗口部件，从而可以提供十几甚至几十个选项。使用带有自定义窗口部件的插件对Qt设计器进行扩展也是可以的。这些自定义窗口部件通常用C++写成，但自PyQt 4.2以后，也能够兼容使用Python写成的自定义窗口部件。

Qt文档中包含了一个Qt设计器内容全面的使用指南，其中会给出更为深入且覆盖更为广泛的可用内容。然而，在这一节中所给出的内容作为起步来说还是足够了，只是，学习Qt设计器的最佳方式还是去多用它。

在设计了用户界面后，下一步就是写代码以使其变得可用。

## 7.2 对话框的实现

在使用Qt设计器创建用户界面时，利用多重继承的方法会创建一个子类，这样就可以把打算赋予用户界面行为的代码放到这里<sup>①</sup>。要继承的第一个类是QDialog。如果一开始使用的是“Widget”模板，那么这里第一个要继承的类就是QWidget，而如果当时用的是“Main Window”模板，那么这里第一个要继承的类就是 QMainWindow。要继承的第二个类，是在使用Qt设计器设计界面时，用来表示用户界面的那个类。

在上一节，我们创建过一个对象名为“FindAndReplaceDlg”的用户界面，保存在文件名为findandreplacedlg.ui的文件中。必须运行pyuic4（直接运行，或者通过mkpyqt.py或Make PyQt间接运行）才能生成ui\_findandreplacedlg.py模块文件。这个模块中有个类，该类的名字是用窗体的对象名加一个Ui\_前缀，所以，在这里，该类的名字就是Ui\_FindAndReplaceDlg。

我们将会调用自己的FindAndReplaceDlg子类，并将其放到findandreplacedlg.py文件中。

在查看类的声明语句和初始化程序之前，先来看看各个导入语句。

```
import re
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import ui_findandreplacedlg
```

第一个导入语句是要用在代码中的常规表达式模块。对于PyQt编程来说，第二个和第三个导入语句是较为常见的。最后一个导入语句是生成的用户界面模块。现在可以看这个子类了。

```
class FindAndReplaceDlg(QDialog,
    ui_findandreplacedlg.Ui_FindAndReplaceDlg):
    def __init__(self, text, parent=None):
        super(FindAndReplaceDlg, self).__init__(parent)
        self.__text = unicode(text)
        self.__index = 0
        self.setupUi(self)
        if not MAC:
            self.findButton.setFocusPolicy(Qt.NoFocus)
```

<sup>①</sup> 使用其他途径也是有可能的，相关内容参见在线文档。不过，其中的任何一种方法都没有比我们这里所用的方法更为便捷。

```

    self.replaceButton.setFocusPolicy(Qt.NoFocus)
    self.replaceAllButton.setFocusPolicy(Qt.NoFocus)
    self.closeButton.setFocusPolicy(Qt.NoFocus)
    self.updateUi()

```

我们会从 QDialog 和 Ui\_FindAndReplaceDlg 继承。在 Python 编程中，我们很少需要用到多重继承，不过这里用的话会要比不用让事情变得简单得多。这样，子类 FindAndReplaceDlg 就会成为它所继承的两个类的联合体，也就可以直接访问它们的各个属性，当然，记得在前面加上 self。

之前的设置已让初始化程序可以接受对话框处理的数据，也可以接受一个父窗口对象。对第一个继承类 QDialog 会调用 super()。会保留文本的一个副本，也会保留一个位置索引值，以便用户后续又多次点击 Find 来查找同样的文本。

这里对 setupUi() 方法的调用是之前没怎么见过的东西。这个方法由生成模块提供。调用它之后，会创建用户界面文件中给定的全部窗口部件，根据设计布局各个窗口部件，设置其属性和 Tab 键次序，并设置各个连接。换句话说，会重新创建我们在 Qt 设计师中所设计的窗体。

此外，setupUi() 方法会调用 QtCore.QMetaObject.connectSlotsByName()，这是一个静态方法，会在窗体窗口部件的各个信号和我们那个使用了特定命名规范的子类方法之间创建一些信号-槽连接。窗体中任何 on\_widgetName\_signalName 形式的方法名，都会用窗口部件的信号与之相连接。

例如，窗体中有个名叫 findLineEdit 的 QLineEdit 型窗口部件。由 QLineEdit 发射的其中一个信号 textEdited(QString)。所以，如果打算连接这个信号，而在初始化程序中调用 connect() 方法，就可以将相应工作留给 setupUi()。在调用这个打算让信号连接的槽的期间，使用 on\_findLineEdit\_textEdited 的这种方法都可以正常工作。这就是用在窗体的所有连接中的方法，除了在 Qt 设计师中通过可视化方式把 close 按钮的 clicked() 信号进行连接之外。

对于 Windows 和 Linux 用户来说，可以很方便地把按钮的光标策略设置为“不接受光标”(No Focus)。对于鼠标用户来说这样做没什么用，不过对使用键盘的用户来说则很有帮助。这意味着，可以通过按下 Tab 键来在各个可编辑的窗口部件之间移动键盘的光标——在这个例子中，查找行编辑窗口部件、替换行编辑窗口部件、复选框和组合框——这样做就可以避免让 Tab 键经过各个按钮了。键盘用户仍旧可以使用自己的加速键来按下任何键(比如 Esc 键可以认为是关闭按钮)。遗憾的是，各个伙伴和按钮没能为 Mac OS X 的键盘用户提供加速键(在切换到支持辅助设备之前)，所以这些用户需要能够用 Tab 键来遍历所有控件，包括这些按钮。为了能够照顾到所有平台，而不需要在 Qt 设计师中设置按钮的光标策略，可以在用 setupUi() 创建了用户界面之后再手动设置它们。

如果所在的平台是 Mac OS X，那么布尔变量 MAC 就会是 True。文件开头的导入语句之后就会设置这一变量，会使用以下并非什么高深莫测的语句：

```
MAC = "qt_mac_set_native_menuBar" in dir()
```

一种更为简洁清晰的写法是这样的：

```
import PyQt4.QtGui
MAC = hasattr(PyQt4.QtGui, "qt_mac_set_native_menuBar")
```

这样做之所以可行是因为，PyQt4.QtGui.qt\_mac\_set\_native\_menuBar() 函数只存在于 Mac OS X 系统上。在第 11 章，对于 X Window 系统的检测我们也会使用类似的技术。

updateUi()方法调用的最后是我们的自定义方法；用它来启用或者禁用各个按钮，这取决于用户是否输入了要查找的任何文字。

```
@pyqtSignature("QString")
def on_findLineEdit_textEdited(self, text):
    self._index = 0
    self.updateUi()
```

幸而有了setupUi(), findLineEdit的textEdited()信号会自动连接到这个方法上。无论何时，在需要自动连接时，都可以使用@pyqtSignature修饰符来给定信号的参数。使用修饰符的目的是为了区分各个具有同样名字不同参数的信号。在本例中，只有一个textEdited()信号，所以修饰符并不是必需的；不过，经常使用修饰符则是一种非常不错的习惯。例如，如果在PyQt的后续版本中引用了另外一个具有同样名字不同参数的信号，那么使用了修饰符的代码仍旧可以正常工作，但那些没有使用修饰的代码就无法工作了。

由于在用户改变查找文本的时候就会调用这个槽，所以需要把从何处开始搜索的位置索引值重置为0(开头处)。这里，以及在初始化程序中，都会以对updateUi()的调用为结束。

```
def updateUi(self):
    enable = not self.findLineEdit.text().isEmpty()
    self.findButton.setEnabled(enable)
    self.replaceButton.setEnabled(enable)
    self.replaceAllButton.setEnabled(enable)
```

这种类型的方法我们已经见过太多的例子。在这里，如果用户输入了需要查找的文字，就会启用Find、Replace、Replace All按钮。至于是否有需要替换的文本并没什么关系，因为有的时候，确实会存在把一些文字替换成空白的情况，也就是，把找到的文字都删除掉。这个方法就是为什么窗体要启用每个按钮而不启用关闭(Close)按钮的原因。

在用户关闭窗体时，使用text()方法仍旧可以访问所保存的文字(如果用户已经部分或者全部替换了文字，这些文字就会与原文有所不同了)。

```
def text(self):
    return self._text
```

一些Python程序开发人员可能不会为此提供方法；相反，他们可能会用一个self.text变量(而不是self.\_text)，并直接访问该变量。

在用户按下任意按钮[除关闭(Close)按钮之外]调用的那些方法和助手程序方法中，会实现对话框的其他剩余功能。与使用Qt设计器相比，它们的实现并没什么特殊之处，不过考虑到完整性，还是会简单地来看一下的。

```
@pyqtSignature("")
def on_findButton_clicked(self):
    regex = self.makeRegex()
    match = regex.search(self._text, self._index)
    if match is not None:
        self._index = match.end()
        self.emit(SIGNAL("found"), match.start())
    else:
        self.emit(SIGNAL("notfound"))
```

按钮的clicked()信号有个可选的布尔型参数项，我们对其并无兴趣，所以会向@pyqtSignature修饰符提供一个空参数清单。比较一下会发现，之前并没有给on\_findLineEdit\_textEdited()槽的通配符提供空参数清单，是因为textEdited()信号的参数不是可有可无的，所以必须应包含一个。

为了可以执行搜索，会创建一个正则表达式，以说明要查找的文本和查找字符的一些特征。然后，使用该正则表达式从当前索引位置搜索文本。如果找到了一个匹配项，就在匹配项的最后更新索引位置，以便为下一次查找做好准备，并发射一个带有找到该文本的文中位置的信号。

```
def makeRegex(self):
    findText = unicode(self.lineEdit.text())
    if unicode(self.syntaxComboBox.currentText()) == "Literal":
        findText = re.escape(findText)
    flags = re.MULTILINE|re.DOTALL|re.UNICODE
    if not self.caseCheckBox.isChecked():
        flags |= re.IGNORECASE
    if self.wholeCheckBox.isChecked():
        findText = r"\b%s\b" % findText
    return re.compile(findText, flags)
```

先从获取用户输入的查找文本开始。由于只有在有了要查找的文本时才会启用这些按钮[除了关闭(Close)按钮]，可以知道查找文本不能是空的。如果用户选中的只是一些文本字符，就可以用 `re.escape()` 函数匹配正则表达式来处理可能会出现在用户查找文本中的那些元字符(比如“\”)了。然后，就会初始化搜索标志。如果没有选中 `caseCheckBox`，会增加一个 `re.IGNORECASE` 标志。如果用户要求搜索完整的单词，可以在搜索文本之前和之后添加一个 \b：这在 Python 的(和 `QRegExp` 的)正则表达式语言中是用来说明一个单词的边界的。在字符串之前添加一个 `r`，表示这是一个“初始”(raw)字符串，说明诸如“\”的字符是无须转换的。最后，以编译过的形式返回该正则表达式<sup>①</sup>。

如果知道待查找的文字通常是 `QString` 型而不是 `unicode`，或许使用 PyQt `QRegExp` 类要比使用 Python 标准库中的类更好些。

```
@pyqtSignature("")
def on_replaceButton_clicked(self):
    regex = self.makeRegex()
    self.__text = regex.sub(unicode(self.replaceLineEdit.text()),
                           self.__text, 1)
```

这个方法非常简单，因为它会把自己的准备工作传给 `makeRegex()`。我们会使用 `sub` 方法(`substitute`)在第一次出现待查找文本的地方用替换文本代替它。替换文本也可以是空的。1 是要替换的最大次数。

```
@pyqtSignature("")
def on_replaceAllButton_clicked(self):
    regex = self.makeRegex()
    self.__text = regex.sub(unicode(self.replaceLineEdit.text()),
                           self.__text)
```

这个方法与之前的方法几乎一模一样。唯一的不同是，没有给定替换的最大次数，所以只要找到几次(不含一个地方的重复出现)待查找文本，`sub()` 就会替换几次。

`FindAndReplaceDlg` 的实现到此为止。对于 `FindAndReplaceDlg` 对话框其他方法的实现确实与之前所做的没什么不同，除了这里使用修饰符和 `setupUi()` 来提供自动连接之外。

<sup>①</sup> `QRegExp` 文档会提供一个有关正则表达式的简要介绍。在 `re` 模块文档中会介绍 Python 正则表达式引擎。有关更多内容可以参阅 Jeffrey E. Friedl 编写的 *Mastering Regular Expressions* 一书。

为了在应用程序中使用该对话框，必须确保 `ui_findandreplacedlg.py` 已生成了模块文件，并且必须导入了刚刚完成的 `findandreplacedlg` 模块。在下一节，将会看到如何创建和使用窗体。

### 7.3 对话框的测试

由于任何 PyQt 窗口部件，包括任何对话框，都能以其适当的方式用做顶层窗口，通过对其实例化和开始事件循环的方式就可以轻松实现对话框的测试<sup>①</sup>。通常来说，尽管还需要再做一些事情。例如，为了能够看到它们能否正确工作，或许还需要设置一些初始数据，或者提供一些接受对话框信号的方法。

在查找和替换对话框这个例子中，我们需要一些初始文本，还需要检查各个连接的工作状态和查找替换方法的工作状态。

所以，在文件的最后，就额外添加了一些代码。只有在该文件单独执行的时候才会执行这些代码，所以不会影响性能或者在将其用到应用程序时打扰用户与对话框的交互。

```
if __name__ == "__main__":
    import sys

    text = """US experience shows that, unlike traditional patents,
software patents do not encourage innovation and R&D, quite the
contrary. In particular they hurt small and medium-sized enterprises
and generally newcomers in the market. They will just weaken the market
and increase spending on patents and litigation, at the expense of
technological innovation and research. Especially dangerous are
attempts to abuse the patent system by preventing interoperability as a
means of avoiding competition with technological ability.
--- Extract quoted from Linus Torvalds and Alan Cox's letter
to the President of the European Parliament
http://www.eff.org/patentit/patents\_torvalds\_cox.html"""

    def found(where):
        print "Found at %d" % where

    def nomore():
        print "No more found"

    app = QApplication(sys.argv)
    form = FindAndReplaceDlg(text)
    form.connect(form, SIGNAL("found"), found)
    form.connect(form, SIGNAL("notfound"), nomore)
    form.show()
    app.exec_()
    print form.text()
```

首先导入 `sys` 模块，然后创建一段工作的文本。接着，为要连接的对话框的各个信号创建一些简单的函数。

仍旧以常用的方式创建 `QApplication` 对象，然后为我们的对话框创建一个实例，为其传入测试文字。把对话框的两个信号连接到槽上，并调用 `show()`。此时，会开始事件循环。当事件循环结束，打印对话框的文本：如果用户替换过某些文字，显示的文字就会与原文稍有不同。

<sup>①</sup> 在使用 `pyuic4` 时，可以给定一个命令行选项，`-x`，让对话框能够额外再生成一些代码，以便可以进行单独测试。

现在，对话框就可以从控制台运行和测试了。

```
C:\pyqt\chap07>python findandreplacedlg.py
```

除非使用自动测试工具，向对话框添加一些测试功能通常会更有帮助。撰写这些代码无须花太多的时间或者耗费太大的努力，对话框的逻辑无论在何时发生变化都会运行这些代码，这样也会有助于降低引入缺陷的可能性。

有时，向对话框传递一些复杂的对象看起来会让测试无法进行。不过，要感谢 Python 的鸭子类型(duck typing)<sup>①</sup>，可以让我们总能够创建一些有助于测试的行为模仿为类。例如，在第 12 章，会用到一个属性编辑器对话框。这个对话框会对一个“Node”对象进行操作，故而在测试代码中，会创建一个 `FakeNode` 类，它能够提供一些用于设置和获取对话框会用到的节点属性的方法(相关文件请参阅文件 `chap12/propertiesdlg.ui` 和实现 `PropertiesDlg` 的文件 `chap12/propertiesdlg.py`，其中，`ui_propertiesdlg.py` 是生成的)。

## 小结

Qt 设计师提供了一种创建用户界面的快速、简易方式。使用可视化设计工具使得直接看到设计能否“工作”更为简单。Qt 设计师的另一个好处是，如果修改了设计，加入没有在代码中添加、移除、重命名任何窗口部件，那么就无须修改任何代码。并且，即使添加过、重命名过、移除过任何窗口部件，这些改变对于我们的代码来说都可能是非常小的，因为 Qt 设计师会为我们处理所有的创建和布局工作。

使用 Qt 设计师的一些基本原理都总是一样的：先把窗口部件拖动到窗体、容器(比如框架、群组框、Tab 窗口部件等)上，然后，设置这些窗口部件的各个属性。接着，添加一些分隔符来占用空隙。再然后，选择一些特殊的窗口部件、分隔符、布局等，然后对它们应用布局功能，重复这一步骤，直至布局了所有一切。再次，对窗体自身进行布局。最后，设置伙伴、按键次序和信号-槽连接。

实现带有由 Qt 设计师创建的对话框的界面与通过手工代码实现的方法类似。最大的不同之处在于初始化程序，在其中，会简单调用 `setupUi()` 来创建和布局各个窗口部件，也会创建各个信号-槽连接。实现这些的方法仍旧可以与之前实现它们的方法相似(它们的代码也不会有任何不同)，但通常会使用 `on_widgetName_signalName` 命名规范，还会使用 `@pyqtSignature` 修饰符来充分利用 `setupUi()` 自动创建连接的功能。

使用中有一点目前并未涉及，就是使用“Widget”模板创建复合窗口部件(由两个或者更多其他窗口部件一起布局而构成的窗口部件)。在某些情况下，这窗口部件的设计可用于整个窗体，而在另一些情况下，它们则会用做窗体的组件——例如，仅是作为 Tab 窗口部件的一个页面或者堆窗口部件的一页。或者，两个或者多个符合窗口部件可以在一个窗体中布局在一起，创建出更为复杂的窗体来。这样，使得利用 Qt 设计师进行设计并以常规方式来生成 Python 模块的这种用法就成为可能。于是，可以导入生成的这些模块，并在窗体类中，调用每个自定义

---

<sup>①</sup> Python 的鸭子类型(duck typing)这个术语的核心概念是指，一个对象的有效语义不是继承自特定的类或者实现方法，而是由当前属性和方法的集合来决定的。举个例子来说，当看到一只鸟时，它走起来很像鸭子，游起泳来很像鸭子，叫起来也像鸭子，那么，就可以称这只鸟是一只鸭子。在鸭子类型中，关注的不是对象的类型本身，而是如何使用它的一译者注。

窗口部件的 `setupUi()` 方法创建用户界面。

至于说，对于诸如到底使用 Qt 设计师所创建的对话框比手工代码创建智能多少、应当采用何种模态形式、如何进行验证工作等问题，实际都没什么两样。但唯一例外的是，在 Qt 设计师中我们是可以设置窗口部件属性的——例如，可以设置微调框的范围和初始值。当然，可以用代码完成同样的事情，不过，对于那些只需要进行简单验证的窗口部件来说，在 Qt 设计师中完成这样的工作通常会更为简便。

必须用 `pyuic4` 把 Qt 设计师的 `.ui` 文件转换成 Python 的模块，无论是直接用 `pyuic4`，还是通过 `mkpyqt.py` 或 `Make PyQt`，如果给了 `.qrc` 文件，这两种方法都可以为源文件生成 Python 模块。

如果没有使用测试工具，可以添加一些只有在窗体单独运行时才运行却不影响对话框表现的测试代码，这样的话，无论是对话框的开发还是后期维护，都会非常方便。如果没有对话框所需要的复杂对象，通常可以创建一个能够提供与复杂对象同样方法的“伪”类，测试时，可以向该伪类传递一个实例即可。

完全可以通过手写代码的方式完成各类 PyQt 程序，这样的话，什么时候都用不上 Qt 设计师。然而，使用可视化设计工具设计对话框也很有用，因为可以即时看到设计结果，也可以快速、简洁地实现对话框的设计和修改。使用 Qt 设计师的另外一个好处是，相当多用于创建、布局、连接窗口部件的重复性代码都可以自动产生而无须手工完成。在这一章和前面一章中，都用到了拿 Qt 设计师来创建对话框。在接下来的各章中，还会看到更多使用 Qt 设计师所创建的对话框示例。

## 练习题

使用 Qt 设计师创建如图 7.11 所示用户界面的一种，或者创建一个自己的设计。可能会用到网格布局，还会用到数值和水平布局。对于网格布局，或许还要多尝试几次，可能需要通过调整窗口部件的尺寸大小和位置来让 Qt 设计师创建出所希望的效果来。对于按钮可以使用 `QDialogButtonBox`。

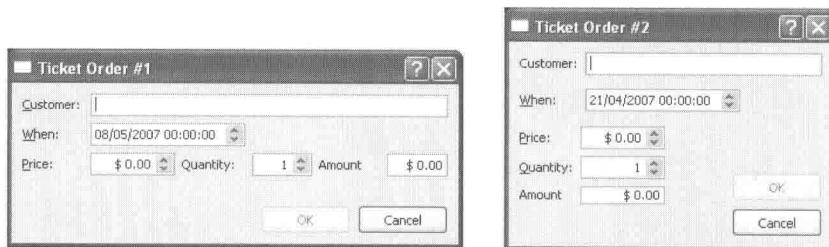


图 7.11 两种不同设计的对话框

价格 (Price) 微调框应当具有的取值范围为  $0.00 \sim 5\,000.00$ ，右对齐，前缀为“\$”，如图 7.11 所示。数量 (Quantity) 微调框应当具有的取值范围为  $1 \sim 50$ ，同样也是右对齐。如果不喜欢默认值，应当可以把日期格式设置为任何值。

在代码中需要引用的窗口部件应当具有有意义的名字，例如，`customerLineEdit` 和 `priceSpinBox`。

设置适当的伙伴关系，也就是，从“Customer”标签到它的行编辑窗口部件，从“When”标签到日期时间编辑窗口部件，等等。还要确保 Tab 键的次序依次是 Customer、When Date、Price、Quantity，最后是按钮框中的 OK 和 Cancel。

使用用户界面创建一个子类。其代码应当确保，如果客户行编辑窗口部件非空，数量比 0 大，那么 OK 按钮就要可用。为了访问 QDialogButtonBox 中的按钮，可以使用带有按钮常量作为参数的 button() 方法——例如，buttonBox.button(QDialogButtonBox.Ok)。

在用户每次修改任何一个微调框的值时，数量(Amount)都应当在数量标签处重新计算和显示。把时间(When)的范围设置为从明天开始，一直到明年的今天。给用户提供一个可以返回四元组(unicode、datetime.datetime、float、int)的 result() 方法，还有时间、价格和数量(如果使用的是 4.1 之前的 PyQt 版本，把时间返回类型设置为 QDateTime；否则，可以使用 QDateTime.toPyDateTime() 方法获得 datetime.datetime)。

在创建和显示 TicketOrderDlg 的最后留上足够的测试代码，以便可以与其交互。在事件循环之后，通过可在控制台上运行的 result() 方法，打印所返回的四元组。

含有测试代码的子类应当不超过 60 行。如果是第一次使用 Qt 设计师，可能会需要 15 ~ 20 分钟来得到正确的设计结果，不过，实际使用中，像这样的对话框应当在几分钟内就可以完成了。

本练习题的参考答案在文件 chap07/ticketorderdlg1.ui 和 chap07/ticketorderdlg2.ui 中，带有测试程序的参考答案放在了 chap07/ticketorderdlg.py。

## 第8章 数据处理和自定义文件格式

- 主窗口的职责
- 数据容器的职责
- 二进制文件的保存和加载
- 文本文件的保存和加载
- XML 文件的保存和加载

大多数的应用程序需要加载和保存数据。通常，数据格式会预先给定，因为应用程序读取的数据多是由不受控制的其他应用程序产生的。不过，对于那些可以创建自己文件格式的应用程序来说，还是有不少可用选项的。

在第6章，我们创建过一个主窗口型应用程序，从中学到了如何创建菜单栏和工具栏，还学到了如何处理文件的加载和保存。在这一章，将会看到另外一种主窗口型应用程序，但这一次，将会把重点放到应用程序的数据上。

用做例子的应用程序的名字叫 My Movies，如图 8.1 所示。该程序会用来存储收藏集中相关电影的基本信息。这个应用程序将允许我们对自定义 Movie 对象(或者也叫 records，正如平时的称呼那样)进行查看和编辑，也允许以不同的文件格式把这些记录从硬盘加载或者保存到硬盘上。

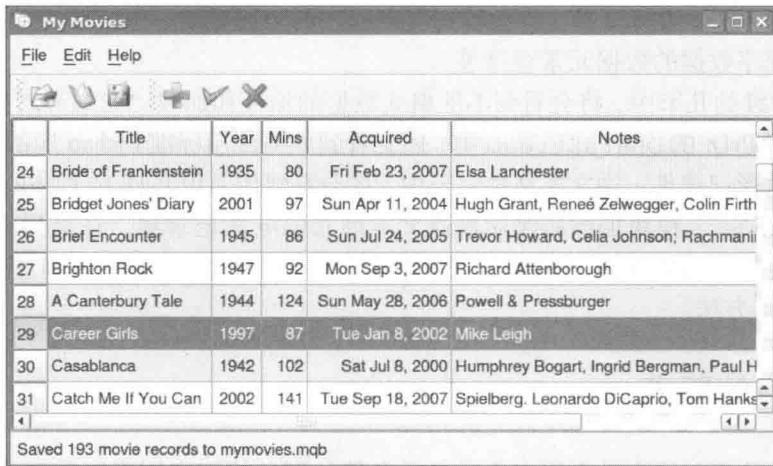


图 8.1 My Movies 应用程序

如果只是打算看看文件处理的细节，则可以直接跳到相关小节即可。

在所有之前的例子中，我们总是会保存尽可能多的 Python 型数据，然后在必要时再转换成 PyQt 型数据，或者从 PyQt 型数据转换成 Python 型数据。对于特殊的字符串，会采用一种策略，就是尽可能把 QString 转换成 unicode 字符串，这样就可以总是对 unicode 字符串进行处理。不过，在这一章会打算采用相反的方法，把数据全部保存为 PyQt 型，并只在必要的

时候才会转换成 Python 型。之所以这样做的原因是，PyQt 对二进制数据提供了非常优秀的支持，并且使用了与 C++/Qt 同样的二进制数据格式，这样就会非常有益于处理一些必须被 C++ 和 Python 程序同时访问的文件。另一个原因是，这样做还会为我们理解每种方法的优劣之处提供一个比较，以便可以为将来需要处理的应用程序做出正确决断。

采用 PyQt 类型保存数据的一大直接好处是，不必把用来查看和编辑的数据从窗口部件转换过来再转换过去。在处理大的数据集时，这将显著地降低数据的存储压力。

在加载和保存自定义数据时，可供使用的选项有 5 种。可以使用二进制、普通文本、XML 文件，或者可以使用带有明确文件名的 `QSettings` 对象，又或者可以使用数据库。在这一章，将会讲述前三种方式，简要提到第 4 种，`QSettings`。有关数据库的内容则会放到第 15 章。

除了 `QSettings` 之外的其他方法都可以使用 Python 标准库或者 PyQt 进行实现。在这一章，既会讨论二进制格式，又会讨论文本格式的加载和保存，以便可以对它们进行比较和对照。对于 XML，文件的加载和检索将会使用 PyQt，并由我们自己完成保存。Python 的标准库也会提供有限的 XML 支持，不过，并不会给出相关内容，因为无法用 PyQt 的 XML 类来完成相关事情。

在第 6 章，看到是如何使用 `QSettings` 对象来保存和加载用户设置，比如主窗口的尺寸大小和位置、最近用过的文件清单等。该类会把全部数据保存成 `QVariants` 类型，但对于少量数据来说，还是可以接受的。可以用这个类保存 `QSettings` 带文件名实例的自定义数据，例如，`iniFile = QSettings("curvedata.ini", QSettings.IniFormat)`。现在，可以使用 `iniFile` 对象 `setValue()` 写入数据，使用 `value()` 读取数据，这两种情况，都会在 `QVariant` 和相关类型之间进行数据转换。

在接下来的一节(8.1 节)中，将会看到高级的文件处理方法和由应用程序主窗口子类所执行的数据表示方法。在 8.2 节，将会看到应用程序的数据模块，包括单一数据元素的实现，以及保存应用程序数据的数据元素容器等。

然后，在后续的几节中，将会看到不同格式数据的保存和加载。在二进制文件一节中，将会看到如何使用 PyQt 的 `QDataStream` 类，还会看到如何使用标准 Python 库的 `cPickle` 模块来加载和保存电影记录集。在文本文件一节中，将会看到如何用 PyQt 的 `QTextStream` 和 Python 标准库的 `codecs` 模块加载和保存普通文本形式的电影记录集。在最后一节中，将会手工编写一些把电影记录集保存为 XML 文件的代码，还会看到如何使用 DOM 和 SAX 解析器来读回 XML 数据的方法。

## 8.1 主窗口的职责

主窗口通常给定的职责是为用户提供高级文件处理动作和应用程序数据的表示。在这一节，将会特别关注于文件相关的动作，因为它们会与第 6 章的图片转换应用程序有所不同，特别是在大型应用程序中的性能表现等方面。还会看到数据是如何展示给用户的。在 My Movies 应用程序中，数据保存在“容器”(container)(名字是 `MovieContainer`)中，保存和加载(以及导出和导入)的全部工作都由主窗口传递给该容器。下一节将会深入研究这个容器，并会在随后的各节中看到该容器保存和加载方面的代码。

源代码放在了 `chap08` 目录中，其中还包括了利用 Qt 设计师完成的用户界面，可用于电影记录的添加和编辑。图 8.2 给出的是本应用程序的各 Python 模块。

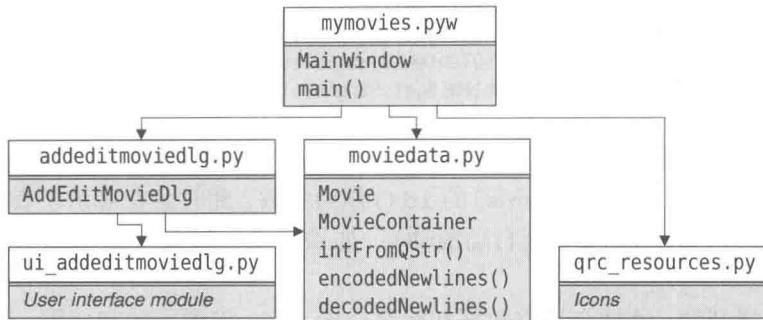


图 8.2 My Movie 应用程序的模块、类和函数

现在需要在保存和导出、加载和导入之间做一个区分。在加载一个文件时，文件名会用来作为应用程序保存时的当前文件名。如果保存了一个文件，就会使用应用程序的当前文件名，所以后面的保存仍将会使用同样的文件。通过“另存为”(save as)动作可以修改当前文件名。在导入一个文件时，会清空当前文件名；这就意味着，如果用户打算保存它，就必须给这些数据一个新的文件名。如果用户导出数据，就会要用户提供一个新的文件名，且当前文件名不受影响。

现在，已经为查看主窗口的文件处理功能做好了准备。就先从主窗口初始化程序的开头开始，来看看数据保存电影容器（movie container）的创建方法和 QTableWidgetItem 的数据表示方法。

```
class MainWindow(QMainWindow):  
    def __init__(self, parent=None):  
        super(MainWindow, self).__init__(parent)  
        self.movies = moviedata.MovieContainer()  
        self.table = QTableWidget()  
        self.setCentralWidget(self.table)
```

调用 `super()` 后，会创建一个新的空电影容器（会简单查看一下 `Movie` 和 `MovieContainer` 类）。然后，会创建一个 `QTableWidget`。这个窗口部件用来表达和部分用于编辑表格数据。表单的设置和显示会放到 `updateTable()` 中。初始化程序的其他部分会忽略掉，因为我们已经从第 6 章知道该如何设置状态栏，创建文件、编辑和帮助动作，显示菜单栏和工具栏，以及如何从前一个进程的设置中恢复应用程序的状态。

考虑到完整性，会简单看看 `updateTable()`，以便知道这里的表单窗口部件是如何设置和显示的（如果只是纯粹对文件处理感兴趣，可以向前跳转到 `fileNew()` 方法。）`updateTable()` 方法的使用非常简单和直接。PyQt 还提供了包括清单、表格、树等基于元素窗口部件更为复杂的显示和编辑方法，其中会使用 PyQt 的模型/视图架构——这些内容会在第 14 章中学到。

```

self.table.setAlternatingRowColors(True)
self.table.setEditTriggers(QTableWidget.NoEditTriggers)
self.table.setSelectionBehavior(QTableWidget.SelectRows)
self.table.setSelectionMode(QTableWidget.SingleSelection)
selected = None

```

这个方法相当长，所以会分成三个部分来查看。调用时可以不带参数，这时只是简单现实表格；或者调用时带一个当前电影(Movie)的 id() 作为参数，此时会在显示了表格后，选中所给定电影的行并使其可见(如有必要会自动滚动)。如果刚添加过或者编辑过一部电影，就会传递这个当前电影。

先从清空表格开始；这样会清空数据和标题。接下来，设置行数和列数，还有列标题。我们会设置表格的各个属性，以便用户不能随意编辑数据，因为在这个特殊的应用程序中，更希望用单独的添加/编辑对话框来编辑数据。还要确保用户每次仅能够选择单独一行。如果当前只有一部电影的话，selected 变量就会保存带有当前电影的题名和返回 QTableWidget-Item 的 id()。

```

for row, movie in enumerate(self.movies):
    item = QTableWidgetItem(movie.title)
    if current is not None and current == id(movie):
        selected = item
    item.setData(Qt.UserRole, QVariant(long(id(movie))))
    self.table.setItem(row, 0, item)
    year = movie.year
    if year != movie.UNKNOWNEYEAR:
        item = QTableWidgetItem("%d" % year)
        item.setTextAlignment(Qt.AlignCenter)
        self.table.setItem(row, 1, item)
    minutes = movie.minutes
    if minutes != movie.UNKNOWNMINUTES:
        item = QTableWidgetItem("%d" % minutes)
        item.setTextAlignment(Qt.AlignRight|Qt.AlignVCenter)
        self.table.setItem(row, 2, item)
    item = QTableWidgetItem(movie.acquired.toString(
        moviedata.DATEFORMAT))
    item.setTextAlignment(Qt.AlignRight|Qt.AlignVCenter)
    self.table.setItem(row, 3, item)
    notes = movie.notes
    if notes.length() > 40:
        notes = notes.left(39) + "..."
    self.table.setItem(row, 4, QTableWidgetItem(notes))

```

QTableWidget 中的每个单元格都由 QTableWidgetItem 表示。这些元素可以保留可显示诸如“用户”数据的文本。我们会遍历电影容器中的每部电影，为每部电影都创建含有数个元素的一行。在每行第一个单元格(元素)中保存电影的题名，对这个元素的用户数据进行设置以保存电影的 id()。必须把 ID 转换为 long，要确保这一转换正确地保存在 QVariant 中。该元素一旦创建并设置完成，就可以将其放到表格中的适当行和列中了。

如果有年代和分钟数的数据，可以只显示这两个数据单元。对于备注来说，如果其中的数据太长，可以对其截短并添加一个省略号，因为备注部分可能会有多达好几段的文字。

```

self.table.resizeColumnsToContents()
if selected is not None:
    selected.setSelected(True)
    self.table.setCurrentItem(selected)
    self.table.scrollToItem(selected)

```

一旦添加过全部表格元素，就可以调整表格的各列，以匹配其中的内容。

在遍历电影容器中的各部电影时，这些电影会以字幕顺序（但会忽略开头的“A”、“An”和“The”单词）的形式返回。如果用户添加了一部新电影，或者编辑了一部已有的电影，就要确保添加的或者是编辑的这部电影的可见性。通过在添加或者编辑之后，调用带有该电影 ID 的 updateTable()，就可以实现这一点。在 updateTable() 的最后，如果传入了电影 ID，selected 变量就会保存该元素所对应的电影的标题单元格，并把这个单元格（以及该单元格的行）变成当前和选中的，如有必要，将会滚动这个表格窗口部件，以确保该行对用户可见。

```
def fileNew(self):
    if not self.okToContinue():
        return
    self.movies.clear()
    self.statusBar().clearMessage()
    self.updateTable()
```

这个方法与第6章图片转换应用程序中的同名方法类似。关键的不同之处在于，除了主窗口负责数据之外，相关工作会移交给保存在 self.movies 中的电影容器。当调用 updateTable() 时，将不会有电影记录，所以该窗口部件除了只是显示列标题之外，再无其他。

okToContinue() 方法与在图片转换应用程序中同名方法几乎完全一样。唯一不同之处在于，不是检查 self.dirty 的状态（因为图片转换应用程序的主窗口保存了应用程序的数据），而是调用 self.movies.isDirty()，因为在这个应用程序中，电影容器会保存相应数据。

```
def fileOpen(self):
    if not self.okToContinue():
        return
    path = QFileInfo(self.movies.filename()).path() \
        if not self.movies.filename().isEmpty() else "."
    fname = QFileDialog.getOpenFileName(self,
        "My Movies - Load Movie Data", path,
        "My Movies data files (%s)" % \
            self.movies.formats())
    if not fname.isEmpty():
        ok, msg = self.movies.load(fname)
        self.statusBar().showMessage(msg, 5000)
        self.updateTable()
```

打开文件的方法与我们之前的方法在结构上是一样的。电影容器会把当前文件名保存为 QString。通常，每个应用程序都会有一种自定义的文件格式，不过，为了说明 My Movie 应用程序能够支持多种文件格式，会提供一个 formats() 方法，由其返回可用的文件扩展名。

主窗口子类会把加载工作传到电影容器上。我们已经设计了电影容器的加载和保存方法，会返回一个布尔型的 success/failure 标志和一条消息。这条消息要么是一个错误消息，要么是一个报告，说明加载或者保存了多少电影记录。在 My Movies 应用程序中，只会使用这种消息。

如果加载成功，电影容器将会包含这些新的电影记录并用 updateTable() 显示它们。如果加载失败，电影容器仍旧为空，updateTable() 将会只显示列的标题。

```
def fileSave(self):
    if self.movies.filename().isEmpty():
        self.fileSaveAs()
    else:
        ok, msg = self.movies.save()
        self.statusBar().showMessage(msg, 5000)
```

再次说明的是，这个方法的逻辑与之前看到的方法一样。用于保存和加载的代码取决于文件扩展名，将会在稍后看到。

这里会跳过 `fileSaveAs()` 的代码；它的代码与图片转换应用程序中的一样，除了文件名用的是 `QString` 而不是 `unicode` 方法之外，默认的扩展名是 `.mql`（意思是：Qt 二进制格式的 My Movies）。

```
def fileImportDOM(self):
    self.fileImport("dom")

def fileImportSAX(self):
    self.fileImport("sax")

def fileImport(self, format):
    if not self.okToContinue():
        return
    path = QFileInfo(self.movies.filename()).path() \
        if not self.movies.filename().isEmpty() else "."
    fname = QFileDialog.getOpenFileName(self,
        "My Movies - Import Movie Data", path,
        "My Movies XML files (*.xml)")
    if not fname.isEmpty():
        if format == "dom":
            ok, msg = self.movies.importDOM(fname)
        else:
            ok, msg = self.movies.importSAX(fname)
        self.statusBar().showMessage(msg, 5000)
        self.updateTable()
```

通常，应当只提供单一的导入方法，使用 SAX 或者 DOM 解析器。这里会给出这两种解析器的用法，所以会提供两个单独的导入动作。两者会产生同样的结果。

文件动作中用于导入的代码与“文件→打开”动作非常相似，只是会使用由用户给定的导入解析器。与所有其他文件处理代码一样，会把工作传递给电影容器。

```
def fileExportXml(self):
    fname = self.movies.filename()
    if fname.isEmpty():
        fname = "."
    else:
        i = fname.lastIndexOf(".")
        if i > 0:
            fname = fname.left(i)
        fname += ".xml"
    fname = QFileDialog.getSaveFileName(self,
        "My Movies - Export Movie Data", fname,
        "My Movies XML files (*.xml)")
    if not fname.isEmpty():
        if not fname.contains("."):
            fname += ".xml"
        ok, msg = self.movies.exportXml(fname)
        self.statusBar().showMessage(msg, 5000)
```

只提供一种 XML 导出方法。其代码与“文件→另存为”动作类似。值得注意的是，必须使用 `QString` 方法来确保文件带有 `.xml` 扩展名，而不是像在图片转换应用程序中所用到的 `unicode` 方法，因为文件名会保存为 `QString`。

## 8.2 数据容器的职责

应用程序的数据容器负责保存全部的数据元素，也就是电影记录，并可用来把电影记录保存到硬盘或者从硬盘加载。在上一节查看 `MainWindow.updateTable()` 方法时已经看到，是如何使用 `for` 循环对容器中的全部电影进行遍历以便可以显示在应用程序的 `QTableWidget` 中的。在这一节，将会看到由 `moviedata` 模块提供的功能，包括用于保存电影数据的数据结构，如何对排序后的遍历操作提供支持等，但另一方面，不会包含真实的保存和加载代码，因为这些代码会在随后的小节中看到。

为什么要使用一个全新的自定义数据容器？毕竟，简单使用一种 Python 的内置数据结构也是可以的，比如清单或者字典。我们更愿意采用把标准数据结构封装到自定义容器类中这样的方法。这样可以确保对数据的访问都能够被我们的类所控制，这就有助于维护数据的一致性。这样也会让容器功能的扩展并在未来不改动已有代码的情况下替换相关数据结构变得更加容易。换句话说，这是一种面向对象的方法，避免了只是使用，比如，清单和一些全局函数所带来的弊端。

先从 `moviedata` 模块的导入语句和常量开始。

```
import bisect
import codecs
import copy_reg
import cPickle
import gzip
from PyQt4.QtCore import *
from PyQt4.QtXml import *
```

电影标题名会按照其字母顺序进行保存，并会忽略名字中的大小写和开头的“A”、“An”和“The”单词。为了降低插入操作的成本和查询的次数，我们会使用 `bisect` 模块来维护这一次序，并使用与第 3 章中实现 `OrderedDict` 相同的技术。

使用特殊文本编码来读取和写入 Python 文本文件时，`codecs` 模块是必需的。`copy_reg` 和 `cPickle` 模块会用来保存和加载 Python 的“pickle 文件”——这是一些包含任意 Python 数据结构的文件。`gzip` 模块用来压缩数据；我们会用它压缩和解压缩那些 pickle 过的数据。`PyQt4.QtCore` 的导入与之相似，不过，还必须导入 `PyQt4.QtXml` 模块来访问 PyQt 的 SAX 和 DOM 解析器。在随后的各节中，还会看到使用中的所有这些 `PyQt4.QtGui` 模块，因为 `moviedata` 模块只是一个不带 GUI 功能的纯数据处理模块。

```
CODEC = "UTF-8"
NEWPARA = unichr(0x2029)
NEWLINE = unichr(0x2028)
```

对于文本文件，我们打算使用 UTF-8 编码。这是一种 8 位的 Unicode 编码方式，每个 ASCII 字符都会占用一个字节，而对于其他任何字符都会使用两个或者更多的字节。在各类文件中，这可能是应用最为广泛的 Unicode 字符编码方式。通过使用 Unicode，就可以像今天使用中的任何人类语言那样存储各个写入文本。

尽管 `\n` 是一个有效的 Unicode 字符，在使用 XML 时，将需要使用与 Unicode 相关的段落分隔符和换行符。这是因为，XML 解析器通常无法区分各个 ASCII 空白字符，比如换行符，还有另一个，比如空格，如果打算保留用户的换行符和段落分隔符的话，就会显得不够方便了。

```

class Movie(object):
    UNKNOWNYEAR = 1890
    UNKNOWNMINUTES = 0

    def __init__(self, title=None, year=UNKNOWNYEAR,
                 minutes=UNKNOWNMINUTES, acquired=None, notes=None):
        self.title = title
        self.year = year
        self.minutes = minutes
        self.acquired = acquired \
            if acquired is not None else QDate.currentDate()
        self.notes = notes

```

Movie 类用来保存一部电影相关的数据。我们会直接使用其实例变量，而不是提供简单的 get 函数和 set 函数。电影标题和备注会存储成 `QString`，日期会由 `QDate` 获取。电影的发行年代和影片时长会保存为 `int`。会提供两个静态常量来说明一部电影的发行年代和影片时长是未知的。

现在可以查看电影容器类了。这个类保存了一个排了序的电影清单，为不同格式类型的电影保存和加载(还有导出和导入)提供函数功能。

```

class MovieContainer(object):
    MAGIC_NUMBER = 0x3051E
    FILE_VERSION = 100

    def __init__(self):
        self.__fname = QString()
        self.__movies = []
        self.__movieFromId = {}
        self.__dirty = False

```

在使用 PyQt 的 `QDataStream` 类时，`MAGIC_NUMBER` 和 `FILE_VERSION` 可用来存储和加载文件。

文件名会存储为 `QString`。`__movies` 清单中的每个元素自身都是一个二元素清单，第一个元素用做排序键，第二个元素是 `Movie`。这就是该类的主要数据结构，并用来依次保存电影。`__movieFromId` 字典的键是 `Movie` 对象的 `id()`，值是 `Movie`。正如在第 1 章看到的那样，每个 Python 对象都会非常方便地拥有一个唯一的 ID，可以通过调用 `id()` 来获得该 ID 值。在知道一部电影的 ID 后，就可以用这个字典来快速查询电影。例如，主窗口会在 `QTableWidgetItem` 的第一列，把电影的 ID 保存为“用户”数据。没有数据的备份，当然，因为这两个数据结构确实保存了对 `Movie` 对象的引用而不是对 `Movie` 对象自己的引用。

```

def __iter__(self):
    for pair in iter(self.__movies):
        yield pair[1]

```

在 `MainWindow.updateTable()` 使用 `for` 循环遍历电影容器时，Python 则会用容器的 `__iter__()` 方法。在这里可以看到，会遍历排序后的 `[key, movie]` 清单，并返回每个电影元素。

```

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

```

这个方法允许我们对电影容器应用 `len()` 函数。

在接下来的几节中，将会看到以不同格式在电影容器中加载和保存电影的代码。不过，首先会看到电影容器是如何清空的，以及电影是如何添加、删除、更新等，以便可以对容器的工作方式有个印象，特别是有关容器的排序。

```
def clear(self, clearFilename=True):
    self._movies = []
    self._movieFromId = {}
    if clearFilename:
        self._fname = QString()
    self._dirty = False
```

这个方法用来清空全部数据，可能包括文件名。它是由 `MainWindow.fileNew()` 调用的，的确会清空文件名，也可以从不同的保存和加载方法中调用，这样就会保留文件名不动。电影容器会维护 `dirty` 标志，以便可以总是知道是否还存在未保存的修改。

```
def add(self, movie):
    if id(movie) in self._movieFromId:
        return False
    key = self.key(movie.title, movie.year)
    bisect.insort_left(self._movies, [key, movie])
    self._movieFromId[id(movie)] = movie
    self._dirty = True
    return True
```

第一个 `if` 语句可以确保没有把同一部电影添加两遍。使用 `key()` 方法可以生成一个适当的键值顺序，使用 `bisect` 模块的 `insort_left()` 函数可以向 `_movies` 清单中插入二元组 `[key, movie]` 清单。这种做法速度很快，因为 `bisect` 模块使用的是二进制分割算法 (binary chop algorithm)。也可以保证 `_movieFromId` 字典是最新的，并把容器设置为 `dirty` 状态。

```
def key(self, title, year):
    text = unicode(title).lower()
    if text.startswith("a "):
        text = text[2:]
    elif text.startswith("an "):
        text = text[3:]
    elif text.startswith("the "):
        text = text[4:]
    parts = text.split(" ", 1)
    if parts[0].isdigit():
        text = "%08d" % int(parts[0])
        if len(parts) > 1:
            text += parts[1]
    return u"%s\t%d" % (text.replace(" ", ""), year)
```

这个方法会产生一个适合用于电影数据排序的键值字符串。我们不会保证键的唯一性，因为排了序的数据结构可以是一个清单，其中重复的键也不会有问题。由于这些代码专用于英语，故而可以清除电影标题中那些明确和不明确的文章。如果电影标题是由数字开头的，那么就把数字前加 0 并按照大小进行排序，例如，“20”就会放在“100”之前。但并不需要对年代数字加前导 0，因为年代都肯定是 4 位数字。所有其他数字都会存储为 PyQt 数据类型，不过，我们为键值字符串选用了 `unicode`。

```
def delete(self, movie):
    if id(movie) not in self._movieFromId:
        return False
    key = self.key(movie.title, movie.year)
    i = bisect.bisect_left(self._movies, [key, movie])
    del self._movies[i]
    del self._movieFromId[id(movie)]
    self._dirty = True
    return True
```

为了删除一部电影，就必须将其从各个数据结构中移除，在`_movies`清单这个例子中，必须先找到电影的索引位置。

```
def updateMovie(self, movie, title, year, minutes=None,
               notes=None):
    if minutes is not None:
        movie.minutes = minutes
    if notes is not None:
        movie.notes = notes
    if title != movie.title or year != movie.year:
        key = self.key(movie.title, movie.year)
        i = bisect.bisect_left(self._movies, [key, movie])
        self._movies[i][0] = self.key(title, year)
        movie.title = title
        movie.year = year
        self._movies.sort()
    self._dirty = True
```

如果用户编辑了一部电影，应用程序总是会根据用户的修改来调用这个方法。如果传递的分钟数或者备注是`None`，那么就认为它们都没有被修改。如果电影的标题或者年代改变了，那么就表明，现在，电影或许在`_movies`清单的位置错误了。在这些情况下，就会用电影的原始标题和年代来查找该电影，为其设置新的标题和年代，然后再次重新排序该清单。实际上，这样做的代价并不大，因为是在首次出现时就做了的。该清单将至少包含一个不正确的排序元素，且 Python 的排序算法已经为部分排序数据进行了高度优化。

如曾发现这里有性能问题，那么就应当用`delete()`和`add()`重新实现`updateMovie()`。

```
@staticmethod
def formats():
    return ".*.mqb *.*.mpb *.*.mqt *.*.mpt"
```

通常，我们会为应用程序提供一个，或者至多两个自定义数据格式，不过，为了展示用法，这里会用 4 种文件扩展符提供 3 种数据格式。扩展名`.mqb`是 Qt 库格式，它用的是`QDataStream`类；扩展名`.mpb`是 Python 的`pickle`格式（使用`gzip`进行压缩）；扩展名`.mqt`是 Qt 文本格式，它使用`QTextStream`类；而扩展名`.mpt`则是 Python 文本格式。这两种文本格式都是一样的，只是使用了不同的文件扩展名，为了能够对比，可以使用不同的保存和加载代码。

```
def save(self, fname=QString()):
    if not fname.isEmpty():
        self._fname = fname
    if self._fname.endsWith(".mqb"):
        return self.saveQDataStream()
    elif self._fname.endsWith(".mpb"):
        return self.savePickle()
    elif self._fname.endsWith(".mqt"):
        return self.saveQTextStream()
    elif self._fname.endsWith(".mpt"):
        return self.saveText()
    return False, "Failed to save: invalid file extension"
```

在用户调用“文件→保存”动作时，也希望能够调用数据容器的`save()`方法。这正是需要在`My Movies`中发生的，也是常用的套路。然而，在这里，不是执行`save`自身，而是`save()`方法会把工作交给与文件扩展名相关的方法。这样做足矣，以便说明如何保存为不同的格式；不过，在实际应用程序中，通常只会用到一种格式。

还有一个相应的 `load()` 方法，具有与 `save()` 方法相同的逻辑，会把工作交给与文件扩展名相关的加载方法。所有的加载和保存方法都会返回一个二元组，其中，第一个元素是布尔型的 `success/failure` 标志，第二个元素是一条消息，要么是一条错误消息，要么是报告成功发生了什么事情。

目前已经看完了 My Movies 应用程序有关文件处理的基础架构，也看到了容器在内存中保存数据时的数据结构。在接下来的几节中，将会看到容器数据向硬盘保存和从硬盘加载的执行代码。

## 8.3 二进制文件的保存和加载

无论是 PyQt 还是 Python 标准库都会为二进制文件的读写提供便利。PyQt 使用 `QDataStream` 类，Python 标准库使用 `file` 类，要么直接，要么与 `pickle` 模块联合。

二进制格式不适合人类阅读，但却是编码最简单、从硬盘读和往硬盘写速度最快的。就连检索也不是必需的：数字、日期以及多种 PyQt 类型，包括图片，都可以不经归一化就能用来读写。PyQt 对于二进制文件的支持非常强：PyQt 可以确保二进制文件与平台无关，而且对二进制文件进行版本识别也并不困难，这样就可以在必要时及时扩展文件的格式。Python 标准库的 `pickle` 模块（与之对应的是更快的 `cPickle`）也可以提供与平台无关的快速加载和保存，不过，对于处理诸如图片这样的复杂 PyQt 类型还是赶不上 PyQt 的 `QDataStream`。

### 8.3.1 用 `QDataStream` 读写

`QDataStream` 类可以读写 Python 的布尔型和数值型数据，也可以读写 PyQt 类型的数据，包括二进制格式的图片。通过 `QDataStream` 写的文件都是平台无关的；该类还会自动处理字节顺序和字的大小<sup>①</sup>。

几乎每个版本的 PyQt 都有 `QDataStream` 类，都会为数据存储提供新的二进制格式——之所以这样做，是为了 `QDataStream` 可以适应新的数据类型，并提升对已有数据类型的支特。这并不是问题，因为每个版本的 `QDataStream` 都可读取用以前版本格式所存储起来的数据。此外，`QDataStream` 总会用同样的方式存储整数，无论用的是哪个版本的 `QDataStream`。

```
def saveQDataStream(self):
    error = None
    fh = None
    try:
        fh = QFile(self._fname)
        if not fh.open(QIODevice.WriteOnly):
            raise IOError, unicode(fh.errorString())
        stream = QDataStream(fh)
        stream.writeInt32(MovieContainer.MAGIC_NUMBER)
        stream.writeInt32(MovieContainer.FILE_VERSION)
        stream.setVersion(QDataStream.Qt_4_2)
```

<sup>①</sup> 摘自维基百科：“`endian`”一词来源于乔纳森·斯威夫特的小说《格列佛游记》。小说中，小人国为水煮鸡蛋应该从大的一端(`big-end`)剥开还是从小的一端(`little-end`)剥开而争论不休，争论的双方分别被称为“大头邦”和“小头邦”。1980 年，一位网络协议的早期开发者 Danny Cohen 在其著名的论文 *On Holy Wars and a Plea for Peace* 中，为平息一场关于字节该以什么样的顺序传送的争论而第一次引用了该词——译者注。

由于 PyQt 使用返回值而不是异常，如果文件无法打开就会自行抛出一个异常，因为我们更喜欢使用基于异常的方法来处理错误。如果打开了文件，就可以创建一个写入的 `QDataStream` 对象。

PyQt 不能猜测打算用来存储 `int` 型和 `long` 型整数的尺寸大小，所以必须使用 `writeIntn()` 和 `writeUIntn()` 方法写入整数值，其中的 `n` 为 8、16、32 或者 64，也就是，用来存储整数值的字节位数。对于浮点型数字，`QDataStream` 提供了 `writeDouble()` 和 `readDouble()` 方法。这些方法可操作 Python 的 `float` 数字(相当于 C 和 C++ 的 `double`)，将其以 IEEE -754 格式存储为 64 位数据。

第一个要写的整数是“幻数”(magic number)。这是一个任意数字，用来识别 My Movies 的数据文件。这个数字绝不会改变。我们应当在任何自定义二进制数据文件中都给定一个唯一的幻数，因为在识别文件类型时，文件扩展名并不总是靠得住。接下来会写入“文件版本号”(file version)。这是我们文件格式的版本(会将其设置成 100)。如果决定在随后更改文件格式，仍旧会保持幻数不变(尽管，文件中仍旧会保存这电影数据)，不过，文件格式将会发生改变(比如，改变为 101)，以便可以加载不同格式时考虑执行不同的代码。

由于各个整数总是会存储为同样的格式，所以就可以在设置 `QDataStream` 之前，先安全地写入它们。不过，一旦写入了幻数和文件版本号，就应当把 `QDataStream` 的版本号设置成 PyQt 可用来读写剩余数据的相应版本。如果打算充分利用后续版本，就应当使用 `Qt_4_2` 的原始文件格式，而为后续版本再用另一个文件格式。然后，在加载数据时，就应当根据文件的格式数字来设置 `QDataStream` 的版本号。

设置 `QDataStream` 的版本号非常重要，因为这可以确保正确地保存和加载任何类型的 PyQt 数据。如果只是保存和加载整数，那么版本号的设置与否都是无关紧要的，因为它们的表示从来就没有改变过。

```
for key, movie in self._movies:
    stream << movie.title
    stream.writeInt16(movie.year)
    stream.writeInt16(movie.minutes)
    stream << movie.acquired << movie.notes
```

现在遍历电影数据，把每部电影的数据都写入到数据流中。数据的格式列示在了图 8.3 中。对于许多 PyQt 类来说，`QDataStream` 类会重载 `<<` 运算符，包括如，`QString`、`QDate` 和 `QImage`，所以必须使用类 C++ 数据流语句来写入数据。`<<` 运算符会把它的右侧操作数写入到它左侧操作数的数据流中。也可以对同一数据流重复使用这一运算符，因为它只是返回它所应用到的数据流，不过，对于整数，则必须使用 `writeIntn()` 和 `writeUIntn()` 方法。

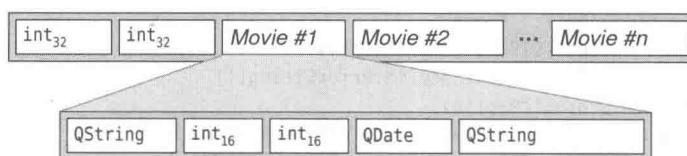


图 8.3 My Movies 中 `QDataStream` 的文件格式

由于写入的是二进制数据，也就无须做任何格式化操作。要确保的只是在加载回数据时，使用的是同一个版本的 `QDataStream`，且读入的数据类型能够与存储时的数据类型一样、次序一致。所以，在这种情况下，将会加载回两个整数(幻数和文件版本号数)，然后是任意数量的电影记录，每条记录都由一个字符串、两个整数、一个日期和一个字符串构成。

```

except (IOError, OSError), e:
    error = "Failed to save: %s" % e
finally:
    if fh is not None:
        fh.close()
    if error is not None:
        return False, error
    self._dirty = False
return True, "Saved %d movie records to %s" % (
    len(self._movies),
    QFileInfo(self._fname).fileName())

```

如果出现任何错误，只需简单地放弃并返回一个失败标志和一条错误信息即可。否则，就清空 dirty 标志并返回一个 success 标志和一条说明保存了多少条记录的消息。

相应的加载方法就很简单，尽管它确实有更多的错误处理方法。

```

def loadQDataStream(self):
    error = None
    fh = None
    try:
        fh = QFile(self._fname)
        if not fh.open(QIODevice.ReadOnly):
            raise IOError, unicode(fh.errorString())
        stream = QDataStream(fh)
        magic = stream.readInt32()
        if magic != MovieContainer.MAGIC_NUMBER:
            raise IOError, "unrecognized file type"
        version = stream.readInt32()
        if version < MovieContainer.FILE_VERSION:
            raise IOError, "old and unreadable file format"
        elif version > MovieContainer.FILE_VERSION:
            raise IOError, "new and unreadable file format"
        stream.setVersion(QDataStream.Qt_4_2)
        self.clear(False)

```

创建 QFile 对象和 QDataStream 对象与之前的方法一样，只是这次会使用 ReadOnly 而不是 WriteOnly 模式。然后，会读入幻数。如果这个 My Movies 数据文件的数字不唯一，那么就抛出异常。接下来，读取文件版本号，确保这是一个我们能够处理的文件。此时，根据文件版本号就可以分开处理了，如果使用中的这种文件格式超过一种的话。然后，设置 QDataStream 的版本号。

接下来是清空电影数据结构。尽可能晚些做这一步，以便能够尽可能早地抛出异常，从而避免损伤原始数据。False 参数告诉 clear() 方法清空 \_movies 和 \_movieFromId，但不会清空文件名。

```

while not stream.atEnd():
    title = QString()
    acquired = QDate()
    notes = QString()
    stream >> title
    year = stream.readInt16()
    minutes = stream.readInt16()
    stream >> acquired >> notes
    self.add(Movie(title, year, minutes, acquired, notes))

```

本应该在文件的开头，文件版本之后，就来存储电影数量的。不过，相反，我们仅会遍历数据

流直到最后。对于非数字型数据，必须创建一些保存正确类型的空变量。然后，使用 `>>` 运算符，会带一个作为左侧操作数的数据流和一个作为右侧操作数的变量；它会从数据流中读取一个右侧操作数类型的值，并将该值放到右侧操作数中去。该运算符会返回文件流，所以它可以多次重复应用。

表 8.1 QDataStream 的部分方法

语法	说明
<code>s.atEnd()</code>	如果已经到达 <code>QDataStream s</code> 的最后，返回 <code>True</code>
<code>s.setVersion(v)</code>	把 <code>QDataStream s</code> 的版本号设置为 <code>v</code> ，其中， <code>v</code> 是 <code>Qt_1_0</code> 、 <code>Qt_2_0</code> 、 <code>…</code> 、 <code>Qt_4_2</code> 或者 <code>Qt_4_3</code> 之一
<code>s &lt;&lt; x</code>	把对象 <code>x</code> 写入到 <code>QDataStream s</code> 中； <code>x</code> 可以是 <code>QBrush</code> 、 <code>QColor</code> 、 <code>QDate</code> 、 <code>QDateTime</code> 、 <code>QFont</code> 、 <code>QIcon</code> 、 <code>QImage</code> 、 <code>QMatrix</code> 、 <code>QPainterPath</code> 、 <code>QPen</code> 、 <code>QPixmap</code> 、 <code>QSize</code> 、 <code>QString</code> 、 <code>QVariant</code> 等
<code>s.readBool()</code>	从 <code>QDataStream s</code> 中读取一个 <code>bool</code>
<code>s.readDouble()</code>	从 <code>QDataStream s</code> 中读取一个 <code>float</code>
<code>s.readInt16()</code>	从 <code>QDataStream s</code> 中读取一个 16 位 <code>int</code> 。另外还有 <code>readUInt16()</code> 方法
<code>s.readInt32()</code>	从 <code>QDataStream s</code> 中读取一个 32 位 <code>int</code> 。另外还有 <code>readUInt32()</code> 方法
<code>s.readInt64()</code>	从 <code>QDataStream s</code> 中读取一个 64 位 <code>int</code> 。另外还有 <code>readUInt64()</code> 方法
<code>x = QString()</code>	从 <code>QDataStream s</code> 中读取对象 <code>x</code> ； <code>x</code> 必须是已经存在的(以便数据流知道要读入的是何种数据类型)，并且可以是任何能够被 <code>&lt;&lt;</code> 写入的类型
<code>s &gt;&gt; x</code>	
<code>s.writeBool(b)</code>	把 <code>bool b</code> 写入到 <code>QDataStream s</code> 中
<code>s.writeDouble(f)</code>	把 <code>float f</code> 写入到 <code>QDataStream s</code> 中
<code>s.writeInt16(i)</code>	把 <code>int i</code> 当成 16 位 <code>int</code> 写入到 <code>QDataStream s</code> 中。另外还有 <code>writeUInt16()</code> 方法
<code>s.writeInt32(i)</code>	把 <code>int i</code> 当成 32 位 <code>int</code> 写入到 <code>QDataStream s</code> 中。另外还有 <code>writeUInt32()</code> 方法
<code>s.writeInt64(l)</code>	把 <code>int i</code> 当成 64 位 <code>int</code> 写入到 <code>QDataStream s</code> 中。另外还有 <code>writeUInt64()</code> 方法

## 文件错误处理的各种方法

在这一章中用于处理文件错误的方法具有如下左侧所示的结构。另外一种在 `chap09/textedit.py` 和 `chap14/ships.py` 中使用的相等有效方法如下右侧所示。

<pre> error = None fh = None try:     # open file and read data except (IOError, OSError), e:     error = unicode(e) finally:     if fh is not None:         fh.close()     if error is not None:         return False, error     return True, "Success" </pre>	<pre> exception = None fh = None try:     # open file and read data except (IOError, OSError), e:     exception = e finally:     if fh is not None:         fh.close()     if exception is not None:         raise exception </pre>
---	---

在调用的地方，假设要处理的是 `load()` 方法，或许会用到与左侧所示方法相类似的代码。

```

ok, msg = load(args)
if not ok:
    QMessageBox.warning(self, "File Error", msg)

```

对于右侧的方法我们可以像这样来使用代码：

```
try:  
    load(args)  
except (IOError, OSError), e:  
    QMessageBox.warning(self, "File Error", unicode(e))
```

另外一种方法，分别用于 chap09/sditexteditor.pyw 和 chap12/pagedesigner.pyw 中，会在文件处理方法的内部由其自身处理全部的错误。

```
fh = None  
try:  
    # open file and read data  
except (IOError, OSError), e:  
    QMessageBox.warning(self, "File Error", unicode(e))  
finally:  
    if fh is not None:  
        fh.close()
```

在调用的地方，只需简单调用 load(args)，让 load() 方法自己去向用户报告任何错误。

对于整数，必须总是使用 readIntn() 和 readUIntn() 方法进行读取，且必须与写入时所给定的位数一样。

一旦读入一部电影的数据，就可以创建一个新的 Movie 对象，并且立即使用上一节中看到的 add() 方法将其添加到容器的数据结构中。

```
except (IOError, OSError), e:  
    error = "Failed to load: %s" % e  
finally:  
    if fh is not None:  
        fh.close()  
    if error is not None:  
        return False, error  
    self._dirty = False  
    return True, "Loaded %d movie records from %s" % (  
        len(self._movies),  
        QFileInfo(self._fname).fileName())
```

错误处理和最后的 return 语句在结构上与之前在 save 方法中用到的一样。

使用 PyQt 的 QDataStream 类写入二进制数据在原理上与使用 Python 的 file 类没有什么两样。使用正确的 QDataStream 版本号必须要小心，也要使用幻数和文件版本号，或者某些类似的方法。《》和《》运算符的使用显得不是那么 Python，但却很容易理解。

本应当把写入电影的代码放到 Movie 类自身中，或许用一个带有 QDataStream 参数的方法并把电影数据写给它就行了。实践中，让数据容器负责文件的处理而不是负责单一的数据元素，这样做通常会更方便些，基本上也是最灵活的。

### 8.3.2 使用 pickle 模块读写

Python 的标准 pickle 模块，以及与之相对应的速度更快的 cPickle，都可以向硬盘存储和从硬盘读取任意的 Python 数据结构。这些模块可以提供完全一致的函数和功能。两者唯一的区别在于，pickle 模块完全是由 Python 实现的，而 cPickle 模块则是用 C 实现的。这些模块只可以支持 Python 标准库内的数据类型和据此构建的类。如果打算用 PyQt 4.3 之前的 PyQt 版本来 pickle 与 PyQt 相关的数据类型，就必须告诉 pickle(或者 cPickle) 模块该如何处理它们。

```

def _pickleQDate(date):
    return QDate, (date.year(), date.month(), date.day())
def _pickleQString(qstr):
    return QString, (unicode(qstr),)
copy_reg.pickle(QDate, _pickleQDate)
copy_reg.pickle(QString, _pickleQString)

```

copy\_reg 模块用来说明如何读取和写入非标准化数据类型。通过带有两个参数的 copy\_reg.pickle() 可以提供相应的信息。第一个参数是打算要 pickle 的新型类，第二个参数是函数。这个函数必须只带一个参数，就是打算 pickle 的类的实例，且应当返回一个二元组，其中，第一个元素是该类，第二个元素是一个标准 Python 类型的元组，其类型可以传入类的构造函数中创建一个实例，它与所传入的实例具有相同的值。

利用这一信息，pickle 模块可以通过把类名存储为文本、把参数存储为标准 Python 类型元组的方式存储各个类的实例。然后，在打算 unpickle(加载)回数据时，Python 可以使用 eval() 重新创建各个实例。

PyQt 4.3 支持对基本 Qt 数据类型的 pickle，这些类型包括 QByteArray、QChar、QColor、QDate、QDateTime、QKeySequence、QLine、QLineF、QMatrix、QPoint、QPointF、QPolygon、QRect、QRectF、QSize、Q.SizeF、QString、QTime，以及全部的 PyQt 枚举变量。这就意味着，无须写入和注册自己的 pickle 函数，就可以对这些任意类型进行 pickle。

```

def savePickle(self):
    error = None
    fh = None
    try:
        fh = gzip.open(unicode(self.__fname), "wb")
        cPickle.dump(self.__movies, fh, 2)
    except (IOError, OSError), e:
        error = "Failed to save: %s" % e
    finally:
        if fh is not None:
            fh.close()
        if error is not None:
            return False, error
        self.__dirty = False
    return True, "Saved %d movie records to %s" % (
        len(self.__movies),
        QFileInfo(self.__fname).fileName())

```

我们可以轻松把任何 Python 数据结构保存成 pickle，包括递归数据结构。通过以二进制模式打开文件并使用 dump() 函数，就可以做到这一点。在这个例子中，已经选择保存压缩了的 pickle(这或许可以降低文件 50% 的大小)，但应当避免像这样使用压缩算法：

```
fh = open(unicode(self.__fname), "wb")
```

必须把文件名转换成 unicode，因为它会被保存成 QString。给 open() 的 wb 参数的意思是“二进制写”。dump() 函数带一个 Python 数据结构，在这个例子中，[key, movie] 清单是一个文件句柄，也是一个格式代码。我们总是用格式代码 2，就意味着 pickle 二进制格式。

由于各个键可由 key() 方法产生，真正需要保存的只是一些 Movie 实例，而不是这些 [key, movie] 清单。如果磁盘空间的优先级高，或许会这样做，不过，好像在加载数据时，需要重新生成各个键，所以会在磁盘空间和保存与加载速度之间取得平衡。考虑到更快、更简单地存储与加载，我们选择了牺牲磁盘空间的做法。

```
def loadPickle(self):
    error = None
    fh = None
    try:
        fh = gzip.open(unicode(self._fname), "rb")
        self.clear(False)
        self._movies = cPickle.load(fh)
        for key, movie in self._movies:
            self._movieFromId[id(movie)] = movie
    except (IOError, OSError), e:
        error = "Failed to load: %s" % e
    finally:
        if fh is not None:
            fh.close()
        if error is not None:
            return False, error
        self._dirty = False
    return True, "Loaded %d movie records from %s" % (
        len(self._movies),
        QFileInfo(self._fname).fileName())
```

unpickle 几乎与 pickle 一样简单。必须记得用 gzip 打开文件，以便可以解压。open() 的参数 rb 的意思是“二进制读”。我们用 pickle 的 load() 函数检索数据；它带有一个文件句柄，并可返回整个数据结构。我们会把数据结构直接赋给 \_movies 清单。然后，遍历各部电影，显示 \_movieFromId 字典：这不能保存，因为需取决于 Movie 的 id()，而在应用程序运行时 id() 每时每刻都不一样。

pickle 和 unpickle 是存储和加载二进制数据最为简单的方法，也最适合用于数据保存在标准 Python 数据类型中的情况。如果把数据保存为 PyQt 的数据类型，通常最好的方式是使用 QDataStream。在存储诸如图片（因为在 pickle 时，目前尚没有任何一种能用的转换方法）这样的复杂 PyQt 数据类型方面，这个类要比 pickle 模块更为有效，它能够产生比 pickle 模块更高压缩的文件（除非要 pickle 的数据已经压缩过了）。这个类还可以轻松在 C++/Qt 应用程序之间提供无缝数据格式。

## 8.4 文本文件的保存和加载

PyQt 和 Python 标准库提供了一些读写文本文件的便捷方法。PyQt 使用 QTextStream 类，Python 标准库会使用 codecs 模块。

普通文本格式通常是人类可以用诸如文本编辑器的软件阅读的，通常也很容易写入。任何类型的数据都可以写成一种形式或者另一种形式的普通文本。利用数字和日期的字符串表示形式，可以相当轻松和紧凑地对其进行表示，而其他类型，诸如图片，则可以写成相当冗长的形式——例如，可以使用 .xpm 格式写成普通文本。

普通文本的读取也要包括对非文本性数据的读取，或者说对它结构的读取（例如，一条记录的结构），意味着必须编写一个解析程序，但这样的话会非常困难，特别是对于复杂数据或者复杂的数据结构。普通文本格式也可以一定方式相当灵活地进行扩展为与之前的格式相兼容，不过，由于在读取编码和写入编码方式之间可能存在些许差异，会造成相关数据的误读，毕竟用户或许会使用一个与文本实际使用的编码方式并不相同的文本编辑程序。这些格式对

于存储简单数据类型的简单文件结构会非常有用<sup>①</sup>。

需要写入的数据会只包含一些简单类型：字符串、整数和日期。不过，还需要对这些文本文件给予一定的结构，以便每条电影记录都可以相互区分开，我们也必须认识到，这些文本记录或许会占用多行。

我们选用的数据结构如图 8.4 所示。在左侧的格式(Format)列中，空格用“`\u2225`”表示，回车用“`\u2226`”表示。

这些记录可能会跨过多行，不过已经假设，每条记录都不会用`\{\{ENDMOVIE\}\}`文本作为开头。在其中引入转义字符或许会是一种更为健壮的解决方案。例如，本来可以假设为开头的任何行，比方说，“`\`”开头的行，都会忽略这个“`\`”并把行中剩下的部分作为纯文本。这样，只需写成“`\{\{ENDMOVIE\}\}`”，就可以在行中包含“`\{\{ENDMOVIE\}\}`”这样的文本了。

Format	Example
<code>\{\{MOVIE\}\} \u2225 title \u2226</code>	<code>\{\{MOVIE\}\} 12 Monkeys</code>
<code>year \u2225 minutes \u2225 acquired \u2226</code>	<code>1995 129 2001-06-21</code>
<code>\{\{NOTES\}\} \u2226</code>	<code>\{\{NOTES\}\}</code>
<code>notes \u2226</code>	<code>Based on La Jet\u00e9e</code>
<code>\{\{ENDMOVIE\}\} \u2226</code>	<code>\{\{ENDMOVIE\}\}</code>

图 8.4 My Movies 文本格式

#### 8.4.1 使用 QTextStream 读写

使用了 `QTextStream` 并以文本格式写操作的代码与之前使用 `QDataStream` 写操作的代码非常相似。

```
def saveQTextStream(self):
    error = None
    fh = None
    try:
        fh = QFile(self._fname)
        if not fh.open(QIODevice.WriteOnly):
            raise IOError, unicode(fh.errorString())
        stream = QTextStream(fh)
        stream.setCodec(CODEC)
        for key, movie in self._movies:
            stream << "\{\{MOVIE\}\}" << movie.title << "\n" \
                << movie.year << " " << movie.minutes << " " \
                << movie.acquired.toString(Qt.ISODate) \
                << "\n\{\{NOTES\}\}"
            if not movie.notes.isEmpty():
                stream << "\n" << movie.notes
            stream << "\n\{\{ENDMOVIE\}\}\n"
```

有两个重点需要注意。第一，必须指明打算使用的编码方式。在各种情况下都会使用 UTF-8；(CODEC 会保存 UTF-8 文本。)如果不这样做，PyQt 将会使用本地的 8 位编码，这样可能就会是任意类型的 ASCII、Latin-1 或者美式 UTF-8、Latin-1 或者西欧 UTF-8、EUC-JP、JIS、Shift-JIS 或者日式 UTF-8。通过指明编码方式，可以确保总是能够用给定的编码方式进行编码，这样就不会错误解释各个字符了。遗憾的是，我们无法保证每名用户在编辑文件时都能够用到正确的编码格式。如果打算要编辑文件，就可以在第一行用 ASCII 码写下编码方式——例如，可以写成与 XML 相似的方式，`encoding = "UTF-8"`——这样至少可以给编辑器一个提示。这一问题应当会在未来几年中逐渐得到解决，因为 UTF-8 正在逐步变成文本文件编码的全球标准。

<sup>①</sup> 如果格式非常简单，或许最简单的方法就是使用 `QSettings` 对象，由其读写特定的文件而不是用手工编码的方式。

第二点应该会明显得多：所有数据都会被写成文本。`QTextStream` 会重载运算符 `<<`，以便自动处理布尔值、数字和 `QString`，但对于其他数据类型则必须转换成它们的字符串表示形式。对于日期，以选择使用 ISO(YYYY-MM-DD) 格式。如果备注部分为空，我们也已经找到了在 `{NOTES}` 标记之后能够避免出现空行的方法。

我们略过了 `except` 和 `finally` 代码块，因为这些代码块与之前多次看过的一样——例如，都在 `saveQDataStream()` 方法中。

尽管以文本格式写出非常直观，但要读取回来就没那么容易了。一方面，将不得不把每一个要读取的整数（年、分钟和所需日期的组成内容）当成文本，还要将这些整数转换成文本的表达方式。但这一主要的问题是，必须正确地解析文件，挑拣出每部电影的各个记录属性值。

整数的处理并不算困难，因为 `QString` 提供了一个 `toInt()` 方法；不过，这个方法会返回一个 `success/failure` 标志而不是抛出异常，每处理一个数字就检查一次，这就意味着每个数字都需要三行代码而不是一行代码。基于此，我们就不得不为整数的读取创建一个更 Python 化的封装函数。

```
def intFromQStr(qstr):
    i, ok = qstr.toInt()
    if not ok:
        raise ValueError, unicode(qstr)
    return i
```

这个函数仅会调用 `QString.toInt()` 并在失败时抛出一个异常，或者在成功时返回该整数。

为了解析电影文本文件，将使用一个有限状态自动机<sup>①</sup> (finite state automaton) 来收集每部电影的数据。用于单部电影解析的自动机如图 8.5 所示。这就意味着，在读取每行文本之前，都会有一个本行包含什么内容的期望。如果没有满足该期望，就会出错；否则，就读取期望数据，把期望设置成下一行文本所应包含的内容。

```
def loadQTextStream(self):
    error = None
    fh = None
    try:
        fh = QFile(self.__fname)
        if not fh.open(QIODevice.ReadOnly):
            raise IOError, unicode(fh.errorString())
        stream = QTextStream(fh)
        stream.setCodec(CODEC)
        self.clear(False)
        lineno = 0
```

该方法一开始足够熟悉。一旦打开了文件，就创建 `QTextStream`，设置编码格式，清空已有的电影数据，为从磁盘读取数据做好准备。

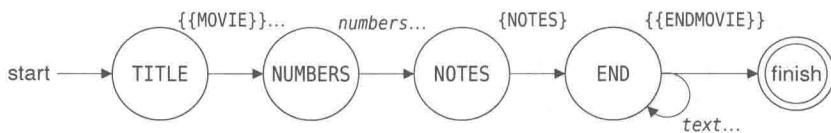


图 8.5 My Movies 文本格式中用于每部电影的有限状态自动机

<sup>①</sup> 有限状态自动机，简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型——译者注。

对于每部电影，首先希望“title”行中含有{{MOVIE}}，后跟以空格，然后是这部电影的标题，再接着是“numbers”行，其中含有年、分钟以及获取日期，然后是“notes”行，只含有{{NOTES}}，再然后是零和多行的备注文本内容，最后是“end”行，只含{{ENDMOVIE}}。我们先从“title”行开始。

为帮助用户发现格式错误，会在lino变量中追踪当前行的行数，该变量会用在错误消息中。

用来读取文件的while循环的主体相当长，所以查看时会分成几个部分。

```
while not stream.atEnd():
    title = year = minutes = acquired = notes = None
    line = stream.readLine()
    lino += 1
    if not line.startsWith("{{MOVIE}}"):
        raise ValueError, "no movie record found"
    else:
        title = line.mid(len("{{MOVIE}}")).trimmed()
```

先从变量初始化开始，将会把一部电影的各个属性值保存为None，以便可以轻松说明我们是否已经读取到了全部内容。

遍历文件的每一行。与Python标准库的file.readline()方法不同，PyQt的QTextStream.readLine()会除去这一行的行末换行符。每次在读取一行时，让lino加1。

任何一部电影的第一行都希望能够以{{MOVIE}}标记开始。如果这一行是错误的，就抛出一个带有错误消息的异常；这个异常处理程序将会在传递给用户的消息中添加行号。如果一行正确，就在这一行的开头处通过读取{{MOVIE}}标记之后的文本来提取电影的标题，同时去掉任何连字符和行末换行符及空格。

QString.mid(n)方法相当于unicode[n:]，QString.trimmed()与unicode.strip()一样。

现在，为读取“numbers”行做好准备了。

```
if stream.atEnd():
    raise ValueError, "premature end of file"
line = stream.readLine()
lino += 1
parts = line.split(" ")
if parts.count() != 3:
    raise ValueError, "invalid numeric data"
year = intFromQStr(parts[0])
minutes = intFromQStr(parts[1])
ymd = parts[2].split("-")
if ymd.count() != 3:
    raise ValueError, "invalid acquired date"
acquired = QDate(intFromQStr(ymd[0]),
                 intFromQStr(ymd[1]),
                 intFromQStr(ymd[2]))
```

开始时，检查是否已经提前到了文件的末尾了，如果是，抛出异常。否则，读入“numbers”行。这一行应当会有一个整数(年)、一个空格、一个整数(分钟数)、一个空格和一个采用YYYY-MM-DD格式的获取日期。一开始，先把这一行在空格字符处分开，这样会得到三个字符串，年、分钟数和获取日期。使用intFromQStr()函数把字符转换成它所表达的整数；如果有任何转换失败，就抛出一个异常，并将其传给该方法的异常处理程序。可以直接转换年和分钟数，但对于获取日期来说，必须再次分割该字符串，这个时间是一个十六进制字符，于是就可以用从每个部分中提取出来的整数值创建一个QDate。

现在，可以读取 { NOTES } 标记了，这是一个可选项，位于这一行的 notes 部分，最后将会是 { { ENDMOVIE } } 标记。

```

if stream.atEnd():
    raise ValueError, "premature end of file"
line = stream.readLine()
lino += 1
if line != "{NOTES}":
    raise ValueError, "notes expected"
notes = QString()
while not stream.atEnd():
    line = stream.readLine()
    lino += 1
    if line == "{{ENDMOVIE}}":
        if title is None or year is None or \
           minutes is None or acquired is None or \
           notes is None:
            raise ValueError, "incomplete record"
        self.add(Movie(title, year, minutes,
                       acquired, notes.trimmed()))
        break
    else:
        notes += line + "\n"
else:
    raise ValueError, "missing endmovie marker"

```

我们希望获得含有 { NOTES } 标记的单行。此时，会把 notes 变量设置为空 QString。即使没有添加任何备注文本，实际上也可以得到一个 QString 而不是一个 None，这足以告诉我们读取到了备注部分，即使这部分的确就是空的而已。

表 8.2 QTextStream 的部分方法

语法	说明
s.atEnd()	如果到达了 QTextStream s 的最后，返回 True
s.setCodec(c)	把 QTextStream s 的文本编码模式设置成 c 所给定的——可以是一个字符串（例如，“UTF-8”），也可以是一个 QTextCodec 对象
s << x	把对象 x 写入到 QTextStream s 中；x 可以是任意类型的 bool、float、int、long、QString、str、unicode 或者一些其他类型
s.readLine()	读取一行，以 QString 形式返回，去掉任何的行结束字符
s.readAll()	读取整个文件，将其以 QString 形式返回

现在有两种情况。或者有 { { ENDMOVIE } } 标记，或者正在读取备注内容的一行。对于后一种情况，只需把这行添加到已有的备注部分即可，并在最后重新添加一个被 PyQt 的 readLine() 方法剔除掉的换行符。然后就可以继续循环，要么遇到 { { ENDMOVIE } } 标记，要么是备注的另一行。

如果得到标记，可以核查任何一个变量都不是 None，以保证读取了一条电影记录的所有数据，然后用搜集到的数据创建并添加一部新的电影。现在，打破内部的 while 循环，准备读取另一部电影，或者，如果刚刚读取的电影是文件中的最后一条记录，就结束。

如果就从来没有得到过 { { ENDMOVIE } } 标记，有些时候正好会处于文件的最后，内部的 while 循环就会结束。如果发生了这些，将会执行 while 循环的 else 选项并抛出一个适当的异常。仅在循环结束，才会执行 while 或者 for 循环的 else 选项，否则，就会用 break 语句来结束循环。

```

except (IOError, OSError, ValueError), e:
    error = "Failed to load: %s on line %d" % (e, lineno)
finally:
    if fh is not None:
        fh.close()
    if error is not None:
        return False, error
    self._dirty = False
return True, "Loaded %d movie records from %s" % (
    len(self._movies),
    QFileInfo(self._fname).fileName())

```

错误的处理与之前所见到的几乎完全一样，只是这一次，包含了错误出现的行号。

#### 8.4.2 使用 codecs 模块读写

使用 PyQt 类的另外一种方法是使用 Python 用于文本文件读写的各个内置类和标准库类。使用这些文件类，可以直接读写文件，不过，如果打算指定编码方式，就必须用 `codecs` 模块来代替了。

```

def saveText(self):
    error = None
    fh = None
    try:
        fh = codecs.open(unicode(self._fname), "w", CODEC)
        for key, movie in self._movies:
            fh.write(u"{{MOVIE}} %s\n" % unicode(movie.title))
            fh.write(u"%d %d %s\n" % (movie.year, movie.minutes,
                                      movie.acquired.toString(Qt.ISODate)))
            fh.write(u"{{NOTES}}")
            if not movie.notes.isEmpty():
                fh.write(u"\n%s" % unicode(movie.notes))
            fh.write(u"\n{{ENDMOVIE}}\n")

```

在用 `QTextStream` 写文件时，会使用与之前用过的完全一样的文件格式，所以代码会与 `saveQTextStream()` 非常相似。会使用 `codecs.open()` 函数而不是 `open()` 函数打开文件；并不需要指明“binary”标志。我们忽略了从 `except` 代码段到最后的代码，因为与之前看过的是一样的。

```

def loadText(self):
    error = None
    fh = None
    try:
        fh = codecs.open(unicode(self._fname), "rU", CODEC)
        self.clear(False)
        lineno = 0
        while True:
            title = year = minutes = acquired = notes = None
            line = fh.readLine()
            if not line:
                break
            lineno += 1
            if not line.startsWith("{{MOVIE}}"):
                raise ValueError, "no movie record found"
            else:
                title = QString(line[len("{{MOVIE}}") : ].strip())

```

这里只会给出 `loadText()` 方法前面几行与 `saveText()` 相关的代码。这是因为，该方法使

用了同样的算法，使用了与 `loadQTextStream()` 方法几乎一样的代码。唯一的显著不同之处在于，考虑到要像 Python 的 `unicode` 一样读取几行内容，就必须把标题和备注都转换成 `QString`。另外，Python 会保留换行符而不是抛弃它们，然后返回一个空字符串来说明已经到了文件的结尾处，所以考虑到这一点，就必须稍稍修改一下代码。对于整数，可以使用 Python 的 `int()` 函数而不是使用真正用于 `QString` 的 `intFromQStr()` 函数。

往回读取会选用 `rU` 模式，这代表“读取一般的换行符”，而不是选择 `r` 模式，这表示“读”。这就意味着，这些行都将会得到正确读取，即使是在，比如说是在 Linux 上写入的，又比如说是在 Windows 下，往回读取的，甚至是在这两个操作系统上使用了不同的行结束规则，都可以正确读取。

## 8.5 XML 文件的保存和加载

无论是 PyQt 还是 Python 标准库，都可以读写 XML 文件。PyQt 提供两种读取解析程序，可以用它的 `QDomDocument` 类写 XML 文件。PyQt 4.3 添加了两个新的 XML 类。`QXmlStreamReader` 类与 SAX 一样轻便，使用也非常容易，`QXmlStreamWriter` 类在写 XML 文件时，要比手工写入或者使用 DOM 写入都更容易和高效。Python 标准库还对 XML 提供扩展支持，不过在这一节，仅会将用法限定到 PyQt 库所提供的功能上，因为 Python 的 XML 类在 Python 的文档、*Python and XML* 和 *XML Processing in Python* 等书中均有较为详细的介绍。

XML 格式与普通文本格式相比显得有些冗长，也不太容易由人读取。另一方面，需要关注的编码问题，所以手工编码能够比普通文本更可靠些，在解析整个文件结构的时候，使用 XML 库通常要比普通文本更容易些。XML 格式的扩展性通常要比二进制和普通文本格式更简单些，尽管在写 XML 时还是要稍加注意，以确保数据中没有包含 XML 的元字符。写 XML 比较简单直接，不过读取还是需要使用解析程序的。有两种非常不同并广泛使用的 XML 解析程序 API：一是 DOM（文档对象模型，Document Object Model），会把整个 XML 文档加载到内存中，适用于编辑文档的结构；二是 SAX（XML 简单 API，Simple API for XML），采用递增式工作方式，所以为少资源饥渴型且适用于 XML 文档的搜索和处理。这里会给出实际应用中的这两种解析程序。

### 8.5.1 XML 的写

如果已经把 XML 文档读取到 `QDomDocument` 中，或者在程序中已经创建了或者加载了 `QDomDocument`，把文档保存到硬盘去的最简单办法就是使用 `QDomDocument.toString()`，可以把整个文档形成一个 XML 格式的 `QString`，然后再把这个字符串保存到硬盘去。实际应用中，尽管经常仅会把 XML 用做数据交换格式，但也还是会把数据保存成自定义数据结构的。这些情况下，就需要写入 XML 自身了，这也是一会将要看到的。

在 XML 中，“空白”字符的顺序，比如换行符、Tab、空格等，通常都会被当成单一空格。这样做很方便，不过并不适用于我们的电影备注，因为对于备注来说，打算保留用户插入的换行符和段落标记。

```
def encodedNewlines(text):
    return text.replace("\n\n", NEWPARA).replace("\n", NEWLINE)

def decodedNewlines(text):
    return text.replace(NEWPARA, "\n\n").replace(NEWLINE, "\n")
```

在之前的两个函数可以用来保留用户的段落标记和换行符。第一个函数会用 Unicode 字符来编码和表示段落标记和换行符，第二个函数会用 Unicode 字符来把段落标记和换行符解码为类似于\n的字符。

利用这两个可用的函数，就可以来看看该如何以 XML 格式输出电影数据。从查看打算产生的文件格式开始：

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE MOVIES>
<MOVIES VERSION='1.0'>
...
<MOVIE YEAR='1951' MINUTES='100' ACQUIRED='2002-02-07'>
<TITLE>The African Queen</TITLE>
<NOTES>
Katherine Hepburn, Humphrey Bogart
</NOTES>
</MOVIE>
...
</MOVIES>
```

为了节省空间，这里省略了用来表达电影记录的各个椭圆，加之它们也并不属于格式的一部分。尽管总是会以同样的顺序写出<MOVIE>标签的各个属性，到目前为止，XML 解析程序主要担心这些顺序出现凌乱。这些属性值不应当包含单个或者两个引用、XML 元字符、<、>以及&<sup>①</sup>。对于各个属性值就意味着，要么对其转义，要么确保只使用这些我们已知不会包含那些字符的值——例如，数字、日期和 ISO 格式的时间以及布尔值等。对于诸如标题和备注的字符串，可以包含引用，因为只有元字符是不允许出现的。

```
def exportXml(self, fname):
    error = None
    fh = None
    try:
        fh = QFile(fname)
        if not fh.open(QIODevice.WriteOnly):
            raise IOError, unicode(fh.errorString())
        stream = QTextStream(fh)
        stream.setCodec(CODEC)
        stream << ("<?xml version='1.0' encoding='%s'?>\n"
                  "<!DOCTYPE MOVIES>\n"
                  "<MOVIES VERSION='1.0'>\n" % CODEC)
```

在我们的 XML 文件中，写入选择使用的是 PyQt 的 QTextStream；本来也可以像在 codecs 模块中那样的简单用法，尽管那时本应当把 QString 转换成 unicode 的<sup>②</sup>。

截至目前，这个方法仍以比较熟悉的方式开始。一旦创建了 QTextStream，就可以像往常一样将其编码设置为 UTF-8，然后输出前面的三行——这些总是一样的。

```
for key, movie in self._movies:
    stream << ("<MOVIE YEAR='%d' MINUTES='%d' "
               "ACQUIRED='%s'>\n" % (
               movie.year, movie.minutes,
```

<sup>①</sup> 为了转义 XML 文本，必须把<to &lt;,>转换成&gt;，把&转换成&amp;。在属性值中，除了这些转义字符外，还必须把'转换成&apos;，把"转换成&quot;。

<sup>②</sup> PyQt 4.0 有个缺陷，会阻止用 QTextStream 正确写入，所以对于 PyQt 4.0 用户来说，就必须使用 codecs 模块。这个问题在 PyQt 4.1 以后的版本中已经不复存在。

```

        movie.acquired.toString(Qt.ISODate))) \
    << "<TITLE>" << Qt.escape(movie.title) \
    << "</TITLE>\n<NOTES>"
if not movie.notes.isEmpty():
    stream << "\n" << Qt.escape(
        encodedNewlines(movie.notes))
stream << "\n</NOTES>\n</MOVIE>\n"
stream << "</MOVIES>\n"

```

我们仍旧用之前所用一样的方法遍历电影数据。`Qt.escape()` 函数带有一个 `QString` 并会以适当转义后的任意 XML 元字符返回它。使用自己的 `encodedNewlines()` 函数把备注部分中的所有段落标记和换行符都转换成相应的 Unicode 字符。对于各个属性则不会执行任何转义，因为我们知道，它们中不会包含任何不可接受的字符。我们会忽略这个方法的最后部分，因为异常处理和返回与之前我们看过的在结构上都是一样的。

### 8.5.2 用 PyQt 的 DOM 类来读取和解析 XML

PyQt 的 `QDomDocument` 类可以一次读入整个(结构良好的)XML 文档。不过，一旦拥有了 `QDomDocument`，就必须要学会使用它。一些应用程序会把文档反应成窗口部件，通常是 `QTreeWidget`，然而有些情况，如 `My Movies`，则会纯粹用 XML 来作为数据交换的格式，通览整个文档后，把数据依照其结构予以展示。

```

def importDOM(self, fname):
    dom = QDomDocument()
    error = None
    fh = None
    try:
        fh = QFile(fname)
        if not fh.open(QIODevice.ReadOnly):
            raise IOError, unicode(fh.errorString())
        if not dom.setContent(fh):
            raise ValueError, "could not parse XML"
    except (IOError, OSError, ValueError), e:
        error = "Failed to import: %s" % e
    finally:
        if fh is not None:
            fh.close()
        if error is not None:
            return False, error
    try:
        self.populateFromDOM(dom)
    except ValueError, e:
        return False, "Failed to import: %s" % e
    self.__fname = QString()
    self.__dirty = True
    return True, "Imported %d movie records from %s" % (
        len(self.__movies), QFileInfo(fname).fileName())

```

加载方法的第一部分看起来好像很熟悉，不过需要注意的是，会在调用 `QDomDocument.setContent()` 的时候读取整个文件。如果这个方法调用成功(返回 `True`)，就可以知道，XML 成功得到了解析。

一旦拥有 `QDomDocument`，就需要用它提取数据，这就是 `populateFromDOM()` 所要做的事情。

这个方法的结尾与我们之前所见到的不大一样。会清空文件名并把 dirty 标志设置为 True。这样将确保，如果用户试图保存已经导入了的 XML 格式的电影数据，或者如果用户试图退出应用程序，都会给他们一个“另存为”对话框，以便使用户能够有机会保存应用程序的二进制或者文本格式的数据。

之前已经提到过，QDomDocument 是可以轻松写入文件的。这里会给出如何使用它的说明，假设 dom 的类型是 QDomDocument，filename 是个有效的文件名，但不会带错误检验：

```
codecs.open(filename, "w", "utf-8").write(unicode(dom.toString()))
```

这样将会生成一个文件，与通过 exportXml() 生成的文件会有些许不同。例如，QDomDocument.toString() 方法会对齐各个标记，对于属性会使用双引号而不是单引号，从而让各属性次序略有不同。此外，生成的 XML 文档则会与 exportXml() 生成的文档保持严格一致。

一旦 QDomDocument 读入 XML 文件，就需要遍历其内容以显示应用程序的数据结构，要完成这些则可以从对文档调用 populateFromDOM() 开始。

```
def populateFromDOM(self, dom):
    root = dom.documentElement()
    if root.tagName() != "MOVIES":
        raise ValueError, "not a Movies XML file"
    self.clear(False)
    node = root.firstChild()
    while not node.isNull():
        if node.toElement().tagName() == "MOVIE":
            self.readMovieNode(node.toElement())
        node = node.nextSibling()
```

开始时，要先检查所读取的文件确实是一个 XML 电影文件。如果不是，抛出一个异常，如果是，清空数据结构，以便可以为显示从 DOM 文档中提取的数据做好准备。

DOM 文档由“节点”(node) 构成，每个节点都可以表示成一个 XML 标签或者带有两个标签的文本。一个节点或许会有孩子，也或许会有兄弟。在 XML 格式的电影中，会有一些 <MOVIE> 兄弟节点，它们带有 <TITLE> 和 <NOTES> 子节点，这些子节点依次带有文本子节点。所以，为了提取数据，就需要遍历这些 <MOVIE> 节点，并且每遇到一个，都要提取它的属性值和它的子节点中的数据。

```
def readMovieNode(self, element):
    def getText(node):
        child = node.firstChild()
        text = QString()
        while not child.isNull():
            if child.nodeType() == QDomNode.TextNode:
                text += child.toText().data()
            child = child.nextSibling()
        return text.trimmed()
```

readMovieNode() 由一个嵌套函数定义开始。getText() 函数的参数是一个节点——例如，是一个 <TITLE> 或者 <NOTES> 为开头的节点——会遍历其子文本节点，不断积攒文本。最后，它会返回这些文本，并从每个结尾处都去掉空白字符。正如之前提示过的，QString.trimmed() 方法会和 unicode.strip() 做同样的工作。

```
year = intFromQStr(element.attribute("YEAR"))
minutes = intFromQStr(element.attribute("MINUTES"))
ymd = element.attribute("ACQUIRED").split("-")
```

```

if ymd.count() != 3:
    raise ValueError, "invalid acquired date %s" % \
        unicode(element.attribute("ACQUIRED"))
acquired = QDate(intFromQStr(ymd[0]), intFromQStr(ymd[1]),
                  intFromQStr(ymd[2]))

```

`readMovieNode()`方法自身从提取`<MOVIE>`标签的属性数据开始，会将其中的年代和分钟数从文本转换成`int`，把获取日期转换成`QDate`。

应该也可以避免自己亲自处理获取日期的细节，只需把相关工作推给解析程序即可。例如，不必用`ACQUIRED`属性，对于每个整数值只要用`ACQUIREDYEAR`、`ACQUIREDMONTH`和`ACQUIREDDAY`即可。用这三个属性，就无须再将那些十六进制数字各个分开了，不过这个格式可能就会更复杂些吧。

```

title = notes = None
node = element.firstChild()
while title is None or notes is None:
    if node.isNull():
        raise ValueError, "missing title or notes"
    if node.toElement().tagName() == "TITLE":
        title = getText(node)
    elif node.toElement().tagName() == "NOTES":
        notes = getText(node)
        node = node.nextSibling()
    if title.isEmpty():
        raise ValueError, "missing title"
self.add(Movie(title, year, minutes, acquired,
               decodedNewlines(notes)))

```

每个`<MOVIE>`节点都有两个子节点，分别是`<TITLE>`和`<NOTES>`。尽管总是会在`exportXml()`方法中依次写出它们，但还是不希望强制让这些子节点带有特定次序。基于这一原因，会遍历这些子节点，并使用嵌套方法`getText()`收集它所遇到的每个子节点中的文本。

在最后，假定电影有标题，会创建一个新的`Movie`对象，并立即使用`add()`将其添加到我们的数据结构中。

使用DOM解析程序把XML导入自定义数据结构的工作方式不错，尽管总是需要编写一些诸如`getText()`这样的小助手函数。在希望利用`QDomDocument`保存和管理XML数据的地方，DOM将是最好的方式，而不是将其转换成数据结构。

### 8.5.3 用PyQt的SAX类读取和解析XML

使用SAX解析程序导入XML与使用DOM解析程序的工作方式相当不同。利用SAX，会定义一个处理器类，能够只执行感兴趣的那些方法，然后会传给SAX解析程序一个该处理器的实例，以用做XML的解析。使用DOM的话，无法完成XML的解析，但可以利用处理器遇到数据时所调用的这些方法一块接一块地解析数据。我们没有实现的那些方法都会由基类提供，在这些情况下，比较安全的做法是什么都别做。

```

def importSAX(self, fname):
    error = None
    fh = None
    try:
        handler = SaxMovieHandler(self)
        parser = QDomSimpleReader()

```

```

parser.setContentHandler(handler)
parser.setErrorHandler(handler)
fh = QFile(fname)
input = QXmlInputSource(fh)
self.clear(False)
if not parser.parse(input):
    raise ValueError, handler.error
except (IOError, OSError, ValueError), e:
    error = "Failed to import: %s" % e
finally:
    if fh is not None:
        fh.close()
    if error is not None:
        return False, error
    self._fname = QString()
    self._dirty = True
return True, "Imported %d movie records from %s" % (
    len(self._movies), QFileInfo(fname).fileName())

```

先从自定义 `SaxMovieHandler` 实例和 SAX XML 解析程序的创建开始。这个解析程序可以带一个内容处理程序、一个错误处理程序和一些其他的处理程序；我们已经选择只创建一个处理程序，这是一个既可以处理内容又可以处理错误的处理程序，所以可以为这两个目的而设置成同样的处理程序。

我们得到了一个 `QFile` 文件处理程序，并把它变成了 XML 的“输入源”。此时，可以清空数据结构，还是尽可能晚一些，然后告诉解析程序去解析 XML 文件。解析程序依然会在成功时返回 `True`，失败时返回 `False`。

处理数据结构时没有明确的阶段分割界限，这是因为所有这些解析过程都是在 `SaxMovieHandler` 类中完成的。最后，清空文件名，把 `dirty` 标志设置成 `True`，就像之前在 `importDOM()` 的最后所做的那样。

`SaxMovieHandler` 类是 `QXmlDefaultHandler` 的一个子类。对于内容的处理，它通常至少会实现 `startElement()`、`endElement()` 和 `characters()` 这几个方法，以便能够处理标签的开始、属性、结束以及两个标签之间的文本内容。如果在处理错误时也使用同样的处理程序，还必须至少要实现 `fatalError()` 方法。

```

class SaxMovieHandler(QXmlDefaultHandler):
    def __init__(self, movies):
        super(SaxMovieHandler, self).__init__()
        self.movies = movies
        self.text = QString()
        self.error = None

```

对 `super()` 的调用可以确保基类得到了适当的初始化。`movies` 参数就是电影容器自身。`text` 实例变量用来保存两个标签之间的文本——例如，标题或者备注的文本——而如果出现错误，则会给 `error` 变量一个错误消息。

```

def clear(self):
    self.year = None
    self.minutes = None
    self.acquired = None
    self.title = None
    self.notes = None

```

第一次调用这个方法时，会为每个电影的属性创建一个实例变量。每次调用它时都会把变量设置成 None；这样会让测试读取电影属性是否完全变得很容易。

```
def startElement(self, namespaceURI, localName, qName, attributes):
    if qName == "MOVIE":
        self.clear()
        self.year = intFromQStr(attributes.value("YEAR"))
        self.minutes = intFromQStr(attributes.value("MINUTES"))
        ymd = attributes.value("ACQUIRED").split("-")
        if ymd.count() != 3:
            raise ValueError, "invalid acquired date %s" % \
                unicode(attributes.value("ACQUIRED"))
        self.acquired = QDate(intFromQStr(ymd[0]),
                              intFromQStr(ymd[1]),
                              intFromQStr(ymd[2]))
    elif qName in ("TITLE", "NOTES"):
        self.text = QString()
    return True
```

这个方法是对基类中该方法的重新实现，基于这一原因，就必须使用相同的名称。我们只对最后两个参数感兴趣：qName（给定的名字）会保存标签的名字，attributes 会保存标签的属性数据。只要一遇到新的开始标签，就会调用这个方法。

如果遇到了一个新的 < MOVIE > 标签，就会清空相应的实例变量，然后就从标签的属性数据中加载年代、分钟数和获取日期。这样会让 title 和 notes 变量成为 None。

如果标签是 < TITLE > 或者 < NOTES >，就会希望能够获得它的相应文本（如果有的话），所以把 text 变量设置成空字符串，以便能够为后加内容做好准备。

每个重新实现的方法都必须在成功时返回 True，或在失败时返回 False；所以，最后我们会返回 True。

```
def characters(self, text):
    self.text += text
    return True
```

只要在两个标签之间一有文本，就会调用 characters() 方法。这里只是简单地将其中的文本添加到 text 变量中。结束标签将会告诉我们是否继续收集标题或者备注的文本。

```
def endElement(self, namespaceURI, localName, qName):
    if qName == "MOVIE":
        if self.year is None or self.minutes is None or \
           self.acquired is None or self.title is None or \
           self.notes is None or self.title.isEmpty():
            raise ValueError, "incomplete movie record"
        self.movies.add(Movie(self.title, self.year,
                              self.minutes, self.acquired,
                              decodedNewlines(self.notes)))
        self.clear()
    elif qName == "TITLE":
        self.title = self.text.trimmed()
    elif qName == "NOTES":
        self.notes = self.text.trimmed()
    return True
```

一遇到结束标签就会调用这个方法。标签的名字在 qName 参数中。如果结束标签是 < /MOVIE >，那么电影数据的任何实例变量都应当设置为 None（尽管标题或者备注可能是空的 QString）。

如果任何一个变量都不是 `None`, 那么假设电影有标题, 就可以创建一个新的 `Movie` 对象, 并且立即将其添加到电影容器中。

如果已经到了标题或者备注的结束标签, 那么可知, 其中的文本 `QString` 已经添加到了相应的开始和结束标签中, 所以会对这些文本做相应赋值。如果没有文本, 那么这个赋值将会是一个空的 `QString`。

```
def fatalError(self, exception):
    self.error = "parse error at line %d column %d: %s" % (
        exception.lineNumber(), exception.columnNumber(),
        exception.message())
    return False
```

如果解析出现错误, 就会调用 `fatalError()` 方法。我们会重新实现它以显示处理程序的 `error` 文本, 并返回 `False` 来说明解析失败。这样将会造成解析程序结束解析过程, 并向其调用程序返回 `False`。

使用 PyQt 的 SAX 解析程序需要我们创建至少一个单独的处理程序子类。这样做并不困难, 尤其是因为只需重新实现打算使用的这些方法。使用 SAX 与使用 DOM 的解析过程相比, 前者也是内存需求不严苛的, 因为 SAX 是逐级递增的工作方式, 且速度上显得更快, 特别是对于大型文档来说。

## 小结

假如所有选择可用, 使用时哪个格式才是最好的, 且我们是应该用 Python 还是用 PyQt 的类?

二进制格式最适合用于性能和平台独立, 也是最易于实现的方式。使用普通文本格式特别适合用于小文件, 特别是只保存简单数值, 如字符串、数字和日期, 以及那些适合手工编辑的东西。即使这样, 还是会存在风险是, 用户的文本编辑器可能会使用与实际不同的编码方式。我们建议对普通文本格式使用 UTF-8, 因为它正在成为事实的标准编码。XML 的读写要比二进制文件的读写慢很多(除非是小文件, 即小于 1 MB 左右的文件), 不过也还是值得使用的, 至少在作为导出和导入格式时还是不错的。尽管如此, 如果用户可以导出和导入自己的 XML 格式的数据, 也就使其具有导出数据的能力, 利用其他一些工具处理数据后, 再导回那些处理过了的数据, 就无须知道或者关心应用程序所通常使用的二进制格式的细节。

至于是使用 Python 还是使用 PyQt 类, 对于保存简单数据类型的小文件来说好像根本无关紧要。如果打算降低编程难度, `cPickle` 模块或许会是最简单的方法。不过, 如果有许多大文件(数百万字节)或者使用了诸如 `QBrush`、`QCursor`、`QFont`、`QIcon`、`QImage` 等复杂的 PyQt 类型, 最简单、最有效的方法仍旧是使用 `QDataStream`, 因为它可以直接读写所有这些类型。使用二进制格式的一个缺点是, 如果打算修改格式, 就至少应修改加载方法以便它可以加载新的和旧的二进制格式。实践中这不并是什么问题, 因为截至目前, 我们都会在文件的幻数之后、数据开始之前的地方包含一个文件版本号, 而我们就可以利用这个版本号来判断到底应该使用哪个加载代码了。

在这一阶段, 我们介绍了 GUI 编程的基础知识。可以创建出带有菜单、工具栏、停靠窗口的主窗口应用程序, 也可以创建和弹出喜欢的任何类型的对话框。也知道了该如何使用 Qt 设计师来简化和加速对话框的设计, 还看到了如何加载和保存不同格式的应用程序数据。在第

三部分，将会进一步深入和拓宽有关 GUI 编程的知识，学习如何处理多文档和如何创建可受用户管理的复杂对话框。还会探索一些 PyQt 的主结构特性，从它的底层事件处理机制到自定义窗口部件的创建，包括 2D 图形等知识，再到包括基于元素的图形、Rich 文本(HTML)处理、模型/视图编程等高级特性。

## 练习题

修改 My Movies 应用程序，以便每个 Movie 对象都可以存储一个额外的信息片段：一个叫做“位置”(location)的 QString，用来说明电影位于什么地方——例如，影厅和楼层。假定只保存和加载二进制的 Qt 格式.mqb 文件，也只导出和导入 XML 文件，所以可以去掉一些用于保存和加载 pickle 与文本文件的代码。要确保新的 My Movies 应用程序仍旧可以读取原应用程序的.mqb 和.xml 文件，即那些没有位置数据的文件。

Moviedata 模块的 Movie 类需要一个叫做“location”的额外 QString 属性。MovieContainer 类将需要数个小的修改。将需要一个旧的和一个当前的文件版本号，以便可以知道正在处理的是哪一个。formats() 方法现在应当只返回字符串 \* .mqb，或者完全忽略干净。save() 和 load() 方法需要能够只处理.mqb 文件，但也必须能够考虑位置和其他不同的文件版本。与 exportXml() 方法和两个导入 XML 方法相似的是，还应考虑出现 <LOCATION> 标签的可能性。对于用户界面的改变是显然的，所以就不单独列出来说明了。

没有哪个修改会需要很多行的代码，不过其中一些还是很繁琐的，可能需要细心些才能得到正确结果。确保测试过自己的修改。例如，可以加载一个旧版本的.mqb 格式，然后再导入一个旧格式的.xml 文件。添加一些位置信息，用新的.mqb 文件保存这些数据，并以 XML 格式导出这些数据。最后再把这两个数据格式读取回去，以检查所有工作都正确无误。

本练习题的参考答案放在几个文件中：chap08/mymovies\_ans.pyw、chap08/movie-data\_ans.pyw、chap08/addeditmoviedlg\_ans.ui 和 chap08/addeditmoviedlg\_ans.py。



# 第三部分

## 中级 GUI 编程

---

- 第 9 章 布局和多文档
- 第 10 章 事件、剪贴板和拖放
- 第 11 章 自定义窗口部件
- 第 12 章 基于项的图形
- 第 13 章 Rich 文本和打印
- 第 14 章 模型/视图编程
- 第 15 章 数据库

# 第9章 布局和多文档

- 布局策略
- Tab 标签页窗口部件和堆叠窗口部件
- 窗口切分条
- 单文档界面(SDI)
- 多文档界面(MDI)

到目前为止，在创建的每一个对话框中，所有的窗口部件都是同时显示的。但是，在某些情况下，如在复杂的配置对话框或者是属性编辑器中，这么多的窗口部件一次同时显示出来可能会让用户产生不小的困惑。在这种情况下，就可以使用 Tab 标签页窗口部件(Tab widget)或者堆叠窗口部件(stacked widget)，这些窗口部件可以将那些有联系的窗口部件都组织在一起，而每次只是同时显示一组相关的窗口部件的集合，或者，也可以采用扩展对话框(extension dialog)的形式按需显示那些额外的选项。所有这些技术都可以使得对话框在为用户提供更小巧、更简单的导览和应用功能；在 9.2 节，将会讲述到这些技术。

在之前所创建的主窗口风格应用程序中，都有一个中央窗口部件(central widget)。不过，在某些情况下，可能会需要在这个中央区域显示两个或者多个窗口部件，并且还经常需要让用户能够对窗口部件的相应大小进行自由控制。要实现这一功能，一种方法是，使用带有多个停靠窗口的单中央窗口部件；这种方法在第 6 章中已经看到过。另外一种方法是，使用窗口切分条(splitter)，会在本章的第 3 节介绍这一内容。

主窗口风格应用程序中的另外一个问题是，到底该如何处理多个文档。对此有 4 种解决方法。第 1 种解决方法是，使用应用程序的多个实例。在这一解决方法中，用户会为打算处理的每一个文档都启动一个应用程序实例。理论上，这样做根本不需要再做任何编程处理，但实际上，可能还是需要实现一些诸如文件锁定的机制(file-locking scheme)，或者是实现一种基于跨进程通信的机制，以确保用户不会对同一文档同时启动多个应用程序。到目前为止所创建的全部应用程序都属于这一种，尽管其中的任何一个都尚未用到文件锁定机制<sup>①</sup>。

第 2 种解决方法是使用单文档界面 SDI(Single Document Interface)。这就意味着，希望用户能够只运行应用程序的一个实例，不过可以使用这个应用程序的实例来创建任意多的主窗口来分别处理所有打算处理的文档(确保用户在同一时刻只启动了一个应用程序的实例是可以实现的，但是这种技术在不同平台会有所不同，这些内容已经超出了本书的范围)。这一方法相当流行，在 *Apple Human Interface Guidelines* 中的“文档风格”一节推荐的就是这种方案。这一方法会在 9.4 节介绍到。

第 3 种解决方法是使用多文档界面 MDI(Multiple Document Interface)。同样，仍旧是希望用户能够只运行应用程序的一个实例，不过在这里，所有的文档同时都会保存在同一个“工作区”(workspace)中，也就是说，各个子窗口都在主窗口的中央区域内。MDI 没有 SDI 那么流

---

<sup>①</sup> 对于文件锁定的代码，可以参阅 *Python Cookbook* 一书中“File Locking Using a Cross-Platform API”一节的内容。

行，也没有那么占用资源。对于 MDI 应用程序来说，只有一个主窗口，无论正在处理的文档是多少个，而 SDI 则对每个文档来说都有一个带有菜单栏、工具栏等的主窗口。在这一章的最后一节，将会介绍如何实现多文档 MDI 应用程序。

第 4 种可选解决方法是使用一个 Tab 标签页窗口部件，每个文档只占据一个自己的标签页。在当下的许多 Web 浏览器中一般都会使用这一解决方法。这里只会介绍对话框中 Tab 标签页窗口部件的用法，不过在练习题中还是有机会去创建一个基于 Tab 标签页窗口部件的主窗口应用程序的，并且也许会惊奇地发现，所需的代码将会与 MDI 应用程序中的代码非常相似。

但是，在介绍 Tab 标签页窗口部件和堆叠窗口部件以及多个文档的处理之前，将会先深入介绍一些关于布局的相关知识，这些内容将会比之前第一次遇到布局时所介绍的内容要深一些。

## 9.1 布局策略

在之前的章节中，曾经看到过很多 PyQt 布局管理器应用的例子。在 PyQt 中，可以将窗口部件设置成特定的尺寸大小和具体的位置，也可以通过重新实现每个窗口部件的 `resizeEvent()` 处理程序来实现布局的手动处理。不过，使用布局管理器是目前最为简单的方法，并且相较于手动处理它可以提供不少额外的好处。

布局管理器允许窗口部件通过增长和收缩来充分利用自己可用的空间，以动态响应用户对窗体尺寸大小的改变。基于全部窗口部件的最小尺寸大小，布局管理器会为窗口提供一个最小尺寸大小。这样就可以确保窗体不至于太小而不能使用，并且也不会固定尺寸大小，而是会根据窗口部件内容的变化而变化——例如，一个标签 (label) 或许需要更多或者更少的空间，都取决于它要显示的文本是英语还是德语。

`QVBoxLayout`、`QHBoxLayout` 和 `QGridLayout` 布局管理器的功能非常强大。这种框式布局 (box layout) 可以借助“伸展”来消耗那些位于窗口部件之间的空间来避免窗口部件过高或者过宽。网格布局 (grid layout) 可以让窗口部件跨越多行和多列。所有的布局管理器都可以相互嵌套，因此可以轻松创建出非常复杂的布局。

尽管如此，仅使用布局管理器有时还是无法完全得到想要的布局效果。为布局管理器提供协助功能的一种简单解决方法就是给布局管理器中那些不太满意布局效果的窗口部件设置其尺寸大小的策略 (size policy)。每个窗口部件都可以单独设置其横向和纵向尺寸大小策略。(每个窗口部件的尺寸大小也都有一个固定的最大值和最小值，但使用尺寸大小策略通常可以提供更好的尺寸大小行为)。另外，有两个尺寸大小是与每个窗口部件都相关的：一个是尺寸大小提示 (size hint)，一个是最小尺寸大小 (minimum size)。前者是窗口部件的期望尺寸大小，后者则是窗口部件压缩时所能够被压缩到的最小尺寸大小。在表 9.1 中，给出了尺寸大小策略中所用到的这两个尺寸值。

例如，假定有一个 `QLineEdit` 的默认策略是水平为 `Expanding`、纵向为 `Fixed`。这就意味着，这个行编辑窗口部件将会占用尽可能的横向空间，并且会始终保持纵向尺寸固定不变。每个内置的 PyQt 窗口部件都设置了一个合理的尺寸大小提示和尺寸大小策略，所以通常在需要调整布局时，只需要改变其中的一个或两个窗口部件的设置即可。

表 9.1 PyQt 的尺寸大小策略

策略	效果
Fixed	 窗口部件具有其尺寸大小提示所给定的尺寸且尺寸再不会改变
Minimum	 窗口部件的尺寸大小提示就是它的最小尺寸；该窗口部件再不能被压缩得比这个值还小，但它可以变得更大
Maximum	 窗口部件的尺寸大小提示就是它的最大尺寸；该窗口部件再不能变得比这个值还大，但它可以压缩到它的最小尺寸大小提示值
Preferred	 窗口部件的尺寸大小提示就是它期望的尺寸；该窗口部件可以缩小到它的最小尺寸大小提示值，或者也可以变得比它的尺寸大小提示还要大
Expanding	 窗口部件可以缩小到它的最小尺寸大小提示值，或者它也可以变得比它的尺寸大小提示值还大，但它希望能够变得更大

尺寸大小策略还为每个策略存储了一个伸展因子(stretch factor)。该因子可以用来说明布局管理器是如何分配窗口部件之间的空间的。例如，如果在一个 QVBoxLayout 中含有两个 QListWidget，它们两个也都希望能够朝两个方向变化。但是，如果希望底部的能够比顶部的变化的快一些，就可以将顶部的伸展因子设置为 1，而把底部的伸展因子设置为 3。这样就可以确保用户调整尺寸大小时，两个窗口部件之间多出来的额外空间将会按照 1:3 的比例进行分配。

如果借助对尺寸大小策略和伸展因子进行设置依然不够，则总是可以通过创建一个子类并重新实现 `sizeHint()` 和 `minimumSizeHint()` 方法来返回所期望的尺寸大小。在第 11 章将会看到这样的一些例子。

## 9.2 Tab 标签页窗口部件和堆叠窗口部件

一些对话框可能需要用许多窗口部件来显示所有可用的选项，但这样会让用户难以理解。最显而易见的方式就是创建两个或者多个对话框来分割这些选项，即每个对话框只显示一部分选项。在可行的情况下，这将是一种不错的解决方法，可以有效简化用户的需求，并且相较于单个的复杂对话框来说，从维护的角度来看，这样做貌似也会更容易些。但是，往往很多情况下需要的只能是一个对话框，因为其中显示给用户的多个选项可能是相互关联并且是需要同时显示在一起的。

在必须使用单个对话框时，一般可以使用两种群组方法来考虑这些选项。一种方法是只群组那些相关的选项。通过使用 `QTabWidget` 可以相当轻松地处理这一方法。一个 Tab 标签页窗口部件根据需要可以有多个“页面”(page)(是一些子窗口部件和一些 Tab 标签页标题)，每个页面都可以排放一些需要显示的相关选项。在图 9.1 中给出了一个 PaymentDlg 对话框，这是一个使用 Qt 设计师(Qt Designer)软件创建的具有 3 个页面的 Tab 标签页窗口部件。

Tab 标签页窗口部件(Tab Widget)在 Qt 设计师的 Widget Box Container 中。可以像其他的窗口部件一样将其拖动到窗体上。它与大多数的容器窗口部件相似，而不是像大多数其他窗口部件那样，在将其拖放到窗体上之后，通常需要手动调整 Tab 标签窗口部件的尺寸大小。在 Qt 设计师里，Tab 标签页窗口部件会有一个上下文菜单选项，可以用来删除和增加页面。通过点击相关的标签可以设定当前页面，也可以通过设置 `currentIndex` 属性来设定当前页。当前页面的标签文本可以通过 `currentTabText` 属性来进行设置。

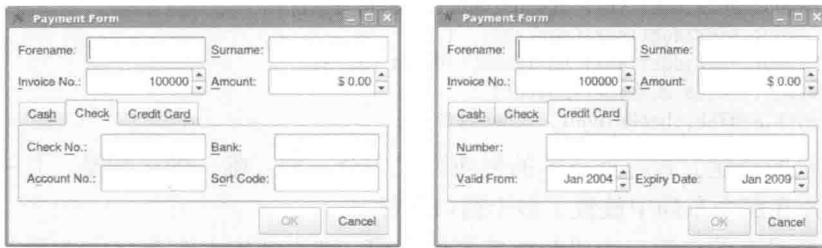


图 9.1 使用 Tab 标签页窗口部件提供支付方式选择的对话框

一旦将一个 Tab 标签页窗口部件拖放到一个窗体上并调整其尺寸大小后，就可以把其他窗口部件拖动到这个标签页里。这些窗口部件都可以以常用方式进行布局，而每个标签页也可以采用相同的方式对自己进行布局：先取消选中的所有窗口部件，然后点击该标签页，再对其应用布局管理即可。

借助每个 Tab 标签页的标记文字，Tab 标签页窗口部件对用户来说都是显而易见的了，因为每个 Tab 标签页面都可以拥有更多的选项，并可以为用户提供一种更为简单的在不同页面之间轻松导航的方法。Tab 标签页窗口部件的 Tab 标签可以是圆角也可以是尖斜角的，也可以将 Tab 标签放到顶部、底部、左侧或者右侧。

尽管在创建对话框时使用 Qt 设计师要比手动处理更为简单快捷，但知道该如何用纯手写代码的形式实现同样的功能却是相当有趣且有益的事情。这里不会再介绍该如何创建那些常规的窗口部件，因为之前已经介绍过多次，相反，这里将会重点关注 Tab 标签页窗口部件和窗体的整体布局。下面的代码片段就是从 chap09/paymentdlg.pyw 中 PaymentDlg 初始化程序中抽取出来的（Qt 设计师版的文件在 paymentdlg.ui 和 paymentdlg.py 中）。

```
tabWidget = QTabWidget()
cashWidget = QWidget()
cashLayout = QHBoxLayout()
cashLayout.addWidget(self.paidCheckBox)
cashWidget.setLayout(cashLayout)
tabWidget.addTab(cashWidget, "Cas&h")
```

Tab 标签页窗口部件的创建方式同其他窗口部件的创建方式一样。Tab 标签页窗口部件中的每个页面都必须包含一个窗口部件，所以这里会先创建一个新的窗口部件 cashWidget，并且还会为其创建一个布局。然后，再向该布局中添加一些相关的窗口部件——在这个例子中，只会添加一个窗口部件，即 paidCheckBox——接着会让该布局应用到所包含的窗口部件上。最后，把包含的窗口部件作为新的 Tab 标签放到 Tab 标签页窗口部件中，同时记得设置 Tab 标签的标签文本<sup>①</sup>。

```
checkWidget = QWidget()
checkLayout = QGridLayout()
checkLayout.addWidget(checkNumLabel, 0, 0)
checkLayout.addWidget(self.checkNumLineEdit, 0, 1)
checkLayout.addWidget(bankLabel, 0, 2)
checkLayout.addWidget(self.bankLineEdit, 0, 3)
checkLayout.addWidget(accountNumLabel, 1, 0)
```

<sup>①</sup> 在 PyQt 文档以及某些扩展中，QTabWidget 的 API 中的“Tab”仅用来指代一个单独的 Tab 标签，也用来指代一个 Tab 的标签和页面组合。

```
checkLayout.addWidget(self.accountNumLineEdit, 1, 1)
checkLayout.addWidget(sortCodeLabel, 1, 2)
checkLayout.addWidget(self.sortCodeLineEdit, 1, 3)
checkWidget.setLayout(checkLayout)
tabWidget.addTab(checkWidget, "Chec&k")
```

这个 Tab 标签页的创建方式与上一个的创建方式完全一样。唯一的区别是，上一个曾用过一个网格布局，并在那个布局中放置了多个窗口部件。

而在这里则没有给出第三个 Tab 标签页的代码，因为结构上它与已经看到过的 Tab 标签页完全一样。

```
layout = QVBoxLayout()
layout.addLayout(gridLayout)
layout.addWidget(tabWidget)
layout.addWidget(self.buttonBox)
self.setLayout(layout)
```

考虑到完整性，这里给出了对话框布局的主要代码，但仅会忽略网格布局的代码，该布局创建于窗体的顶部，其中含有标签、行编辑框以及微调框。各个按钮由 QDialogButtonBox 提供，而一个窗口部件也可以像其他的窗口部件一样进行布局。最后，把整个窗体布局到一个纵向布局框中：首先是顶部的网格，然后是中间的 Tab 标签页窗口部件，接着是底部的按钮框。

另外还有一种群组选项的方法，只是这一方法仅适用于某些特定情况。一种较为简单的情况是，让一组选项变为可用或者不可用，可以用一个可选的 QGroupBox 来实现这一点。如果用户没有选中该群组框，那么这个群组框内所有的窗口部件都会被禁用。这也就意味着，即使这些选项都不可用，用户也仍然可以看到这个群组框内所包含的那些选项，这通常会很有用。在其他一些情况下，可能会有两个或者多个群组选项框，但每次只能有一个群组选项框可以使用。对于这种情况，QStackedWidget 将会是个不错的选择。从概念上来说，一个堆叠窗口部件(stacked widget)就是一个不带 Tab 标签页的 Tab 标签页窗口部件。因此，用户对于这些堆叠显示的窗口部件就没有什么视觉可见的线索来定位它们，也就没有办法在这些堆叠的窗口部件的各个页面之间进行导览了。

要使用堆叠窗口部件，在 Qt 设计师中只需把堆叠窗口部件(Stacked Widget)拖放到窗体上并像对 Tab 标签页窗口部件一样对其调整尺寸大小就可以了。在 Qt 设计师中，会在一个堆叠窗口部件的右上角处分别显示两个小箭头来说明这是一个堆叠窗口部件。这两个小箭头也会在窗体预览时显示出来，不过在运行时它们是不会显示出来的——在图 9.2 的前两个截图中，在窗体上部的颜色组合框中都显示了这样的两个小箭头。与 Tab 标签页窗口部件完全一样，各个窗口部件也可以被拖放到堆叠窗口部件的堆叠页面中。堆叠了的窗口部件都有一个上下文菜单，其中会给出增加和删除页面的选项，就像 Tab 标签页窗口部件一样，在各个页面之间还会有一些额外的选项来进行页面的导航和页面顺序的编辑。

由于堆叠窗口部件没有 Tab 标签，就必须给用户提供一种在不同页面之间进行导航的方法。在图 9.2 中所给出的 VehicleRentalDlg 中，车辆类型组合框就会用做页面选择器。要使其生效，在 Qt 设计师中就需要将该组合框的 currentIndexChanged(int) 信号连接到堆叠窗口部件的 setCurrentIndex(int) 槽上。另外一种常用的解决方法是，要让用户能够看到所有可用的页面，可以使用一个包含了每个页面名字的 QListWidget，就像连接组合框的信号一样，把它的 currentRowChanged(int) 信号连接到堆叠窗口部件的 setCurrentIndex(int) 槽上。

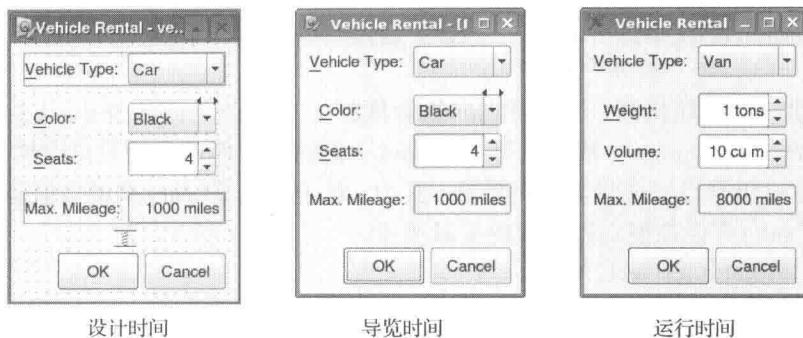


图 9.2 使用了堆叠窗口部件的对话框

现在，将会看看该如何通过代码来创建一个堆叠窗口部件。下面的代码片段就来自文件 chap09/vehiclerentaldlg.pyw 中 VehicleRentalDlg 类的初始化程序 (Qt 设计师版的文件分别放在 vehiclerentaldlg.ui 和 vehiclerentaldlg.py 中)。

```
self.stackedWidget = QStackedWidget()
carWidget = QWidget()
carLayout = QGridLayout()
carLayout.addWidget(colorLabel, 0, 0)
carLayout.addWidget(self.colorComboBox, 0, 1)
carLayout.addWidget(seatsLabel, 1, 0)
carLayout.addWidget(self.seatsSpinBox, 1, 1)
carWidget.setLayout(carLayout)
self.stackedWidget.addWidget(carWidget)
```

向一个堆叠窗口部件添加一个“页面”与向一个 Tab 标签页窗口部件添加一个页面是一样的。先从创建一个普通窗口部件开始，接着会为其创建一个布局并将这些窗口部件布局成我们想要的样子。然后，把该布局应用到这个普通窗口部件上并将这个窗口部件添加到堆叠窗口部件中。这里没有给出 vanWidget 的代码，因为它的代码结构与之前的完全一致。

```
topLayout = QHBoxLayout()
topLayout.addWidget(vehicleLabel)
topLayout.addWidget(self.vehicleComboBox)
bottomLayout = QHBoxLayout()
bottomLayout.addWidget(mileageLabel)
bottomLayout.addWidget(self.mileageLabel)
layout = QVBoxLayout()
layout.addLayout(topLayout)
layout.addWidget(self.stackedWidget)
layout.addLayout(bottomLayout)
layout.addWidget(self.buttonBox)
self.setLayout(layout)
```

考虑到完整性，这里又一次给出了整个对话框布局的代码。先从带有组合框的顶部布局开始，该布局用于设置该堆叠窗口部件的当前窗口部件。接着，会创建一个包含多个英里数标签的底部布局，再为这些按钮创建一个按钮布局。接下来，把所有这些布局，连同堆叠窗口部件自身一起添加到一个纵向布局框中。

```
self.connect(self.buttonBox, SIGNAL("accepted()"), self.accept)
self.connect(self.buttonBox, SIGNAL("rejected()"), self.reject)
self.connect(self.vehicleComboBox,
            SIGNAL("currentIndexChanged(QString)"),
            self.onVehicleTypeChanged)
```

```

        self.setWidgetStack)
self.connect(self.weightSpinBox, SIGNAL("valueChanged(int)"),
             self.weightChanged)

```

必须为用户提供一种导航机制，通过将纵向组合框的 `currentIndexChanged()` 信号连接到自定义的 `setWidgetStack()` 槽上就可以实现这一目的。后面这个槽只是窗体验证功能的一部分；这个槽在这里会设置成最大的英里数，其中，对于小汽车（car）来说该值是个固定值，但对于厢式货车（van）来说该值则需要取决于其质量。

```

def setWidgetStack(self, text):
    if text == "Car":
        self.stackedWidget.setCurrentIndex(0)
        self.mileageLabel.setText("1000 miles")
    else:
        self.stackedWidget.setCurrentIndex(1)
        self.weightChanged(self.weightSpinBox.value())

def weightChanged(self, amount):
    self.mileageLabel.setText("%d miles" % (8000 / amount))

```

`setWidgetStack()` 槽会让适当的窗口部件可见，同时会处理英里数的设置值，因为该值取决于汽车是小汽车（car）还是厢式货车（van）。

这里会使用组合框的当前文本内容来决定该显示哪个窗口部件。一种更为健壮的解决方法可能是，让每个数据元素和组合框的每个元素相互关联起来（可以使用双参数的 `QComboBox.addItem()` 方法），并使用当前元素的数据项来选择该显示哪个窗口部件。

## 扩展对话框

对于复杂对话框还可以采用另外一种解决方法：扩展对话框（extension dialog）法。对于那些具有“简单”模式和“高级”模式的对话框来说，是最为典型的使用场景。初始对话框只会显示一些简单的选项，但会提供一个切换按钮，以用来显示或者隐藏那些高级的选项。

在图 9.3 中给出了一个扩展对话框，当按下切换按钮 More 时，就会显示那些额外的复选框选项。通过调用 `QPushButton` 的 `setCheckable(True)`，任何 `QPushButton` 都可以变成一个切换按钮，或者把在 Qt 设计师中设置 `QPushButton` 的 `checkable` 属性修改成 `True`。

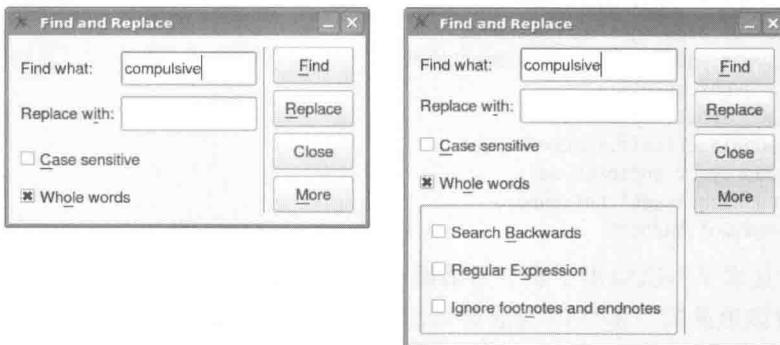


图 9.3 一个扩展对话框

要让扩展对话框能够工作就需要用到两项技术。第一项技术是将所有高级选项的窗口部件都放到一个 `QFrame` 中。这就意味着只需要隐藏或者显示这个框架即可，因为 PyQt 会自动隐藏或者显示在该框架中的全部窗口部件，以反映出该框架的可见性状态。如果在可见时不

希望用户看到框架的外部轮廓，可以吧它的“frameShape”属性设置成 QFrame.NoFrame。

第二项技术是保持对话框的尺寸大小不变。这样做将可以确保对话框能够变得尽可能小（同时可以保持其空白），从而来保证对话框内容可见。这样做将可以保证当对话框的高级选项隐藏时，能够让对话框尽可能小一些，并且可以在展开各个高级选项时，对话框又能够尽可能大。在创建对话框时，也必须隐藏该框架。这里给出的就是这个对话框初始化程序的代码（从文件 chap09/findandreplacedlg.py 提取而来）：

```
class FindAndReplaceDlg(QDialog,
    ui_findandreplacedlg.Ui_FindAndReplaceDlg):

    def __init__(self, parent=None):
        super(FindAndReplaceDlg, self).__init__(parent)
        self.setupUi(self)
        self.moreFrame.hide()
        self.layout().setSizeConstraint(QLayout.SetFixedSize)
```

不过，该如何让 More 按钮与高级选项框架的显示/隐藏状态联系起来呢？只需将 moreButton 的 toggled(bool) 信号连接到 moreFrame 的 setVisible(bool) 槽即可。值得注意的是：如果这个连接是在 Qt 设计师中建立的，就必须将其配置连接对话框（Configure Connection dialog）中的“Show all signals and slots”复选框选中；否则，可能不会显示 setVisible() 槽。

作为这一节的最后一个例子，会再次回顾一下使用代码来实现布局的方法。不像先前的两个布局所给出的新窗口部件（QTabWidget 和 QStackedWidget）的用法，这个对话框的布局则仅用到了之前所看到的那些窗口部件——只不过是以新的方式而已。下面的代码片段是从 FindAndReplaceDlg 类的初始化程序中提取而来的，它在文件 chap09/findandreplacedlg.pyw 中（Qt 设计师版本的文件分别放在 findandreplacedlg.ui 和 findandreplacedlg.py 中）。

这里仅将介绍窗体中与扩展对话框有关的那些窗口部件创建的内容。该窗体的布局如图 9.4 所示。

```
moreFrame = QFrame()
moreFrame.setFrameStyle(QFrame.StyledPanel|QFrame.Sunken)
```

这里会创建一个框架并在其中放置一些额外的复选框。如果没有调用过 setFrameStyle()，该框架将会没有轮廓。

```
line = QFrame()
line.setFrameStyle(QFrame.VLine|QFrame.Sunken)
```

这里“画出来”的这条直线可以显式地将对话框左侧的主窗口部件与右侧的按钮区分开来，它也是一个框架。水平横线可以通过 QFrame.HLine 框架样式创建出来。

```
moreButton = QPushButton("&More")
moreButton.setCheckable(True)
```

More 按钮在某些方面与其他按钮是不同的：它是可选择的。这就意味着，它可以像切换按钮一样工作，当第一次点击时会按下，然后再次点击就会弹起来，以此类推。

各个标签和行编辑框会放到一个网格布局中；这里不会给出这些代码，因为之前已经多次看到过这种类型的布局。

```
frameLayout = QVBoxLayout()
frameLayout.addWidget(self.backwardsCheckBox)
frameLayout.addWidget(self.regexCheckBox)
frameLayout.addWidget(self.ignoreNotesCheckBox)
moreFrame.setLayout(frameLayout)
```



图 9.4 查找替换对话框的布局

还是希望能够在 moreFrame 中对这些额外的复选框进行布局的。为此，这里会创建一个布局，在这个例子里，就是创建一个纵向布局框，并且把这些复选框添加到该布局框中。然后，将这个布局应用于该框架上。在之前的那些例子中，曾经将布局应用于布局上来获得布局的嵌套效果，不过这里会通过将布局应用于框架上而获得布局嵌套。因此，除了可以将一个布局放到另一个布局里面获得嵌套布局之外，也可以将嵌套框架和群组框放到布局里面，这样就可以为用户创造出极大的灵活性。

```
leftLayout = QVBoxLayout()
leftLayout.setLayout(gridLayout)
leftLayout.addWidget(self.caseCheckBox)
leftLayout.addWidget(self.wholeCheckBox)
leftLayout.addWidget(moreFrame)
```

左边的布局是一个纵向布局框，它会被添加到一个网格布局(用一些标签和行编辑窗口部件)中，这个例子中就是大小写敏感复选框和全字匹配复选框，然后再添加 moreFrame(其中会在纵向布局框中含有那些额外的复选框)。

```
buttonLayout = QVBoxLayout()
buttonLayout.addWidget(self.findButton)
buttonLayout.addWidget(self.replaceButton)
buttonLayout.addWidget(closeButton)
buttonLayout.addWidget(moreButton)
buttonLayout.addStretch()
```

这里的按钮布局与之前见过那些布局都一样，只是这一次使用的是纵向布局框而不是横向布局框。

```
mainLayout = QHBoxLayout()
mainLayout.setLayout(leftLayout)
mainLayout.addWidget(line)
mainLayout.setLayout(buttonLayout)
self.setLayout(mainLayout)
```

对话框的主布局是一个横向布局框，由左边的布局、中间是分割线和按钮布局构成。分割线可以随着 moreFrame 的显示与隐藏而在竖直方向上变大或缩小。

```
moreFrame.hide()
mainLayout.setSizeConstraint(QLayout.SetFixedSize)
```

一开始会隐藏 moreFrame(因而会隐藏它所包含的那些窗口部件)，通过将尺寸大小设置成固定(fixed)不变就可以确保对话框能够根据 moreFrame 的可见与否来自动调整自己的尺寸大小。

```
self.connect(moreButton, SIGNAL("toggled(bool)"),
             moreFrame, SLOT("setVisible(bool)"))
```

最后要做的是，必须把 More 按钮的 toggled() 信号连接到 moreFrame 的 setVisible()

槽上。当隐藏(或显示)moreFrame 时, 就可以隐藏(或显示)那些在该框架中的全部窗口部件, 因为当对一个窗口部件调用 show() 或者 hide() 时, PyQt 就会自动将这些调用传播到该窗口部件的所有子窗口部件上。

大家可能已经注意到, 这一节所创建的每个对话框都有两个版本。一个版本是纯手写代码的形式, 比如, paymentdlg.pyw, 而另外一个版本则是 Qt 设计师用户界面的形式, 是个带有代码的模块文件, 例如, paymentdlg.ui 和 paymentdlg.py。通过全部以代码的形式对 Qt 设计师版本的文件(.pyw)和模块文件(.py)的对比可以清楚地发现, 通过使用 Qt 设计师, 到底有多少代码是可以避免使用手写的。使用 Qt 设计师的另外还有一个好处, 特别是对于那些复杂的对话框, 对使用 Qt 设计师完成的设计进行修改比对用纯手写代码完成的设计进行修改要容易得多。

### 9.3 窗口切分条

一些主窗口风格的应用程序可能需要在其中央区域显示多个窗口部件。有两种典型的应用程序就需要这样做, 一个是邮件客户端, 一个是网络新闻阅读器。对于此需求通常会有三种解决方案。第一种解决方案是创建一个由多个其他窗口部件(就像一个对话框一样进行创建和布局, 只不过是继承自 QWidget 而不是 QDialog)构成的复合窗口部件, 然后再把这个复合窗口部件作为主窗口的中央窗口部件。另外一种解决方案是只有一个中央窗口部件, 而把其他的窗口部件都放到各个停靠窗口中——在第 6 章就已经看到过这种形式。第三种解决方案就是使用窗口切分条(splitter), 而这也正是这一节的主要内容。

图 9.5 给出的是一个模拟的新闻阅读器应用程序, 而图 9.6 则给出了窗体的窗口切分条和窗口部件之间的关系。Qt 设计师可以完全支持窗口切分条, 并且它的使用方式与横向和纵向布局框是完全一样的: 选中两个或者多个窗口部件, 点击 Form→Lay Out Horizontally in Splitter(在窗口切分条中横向布局)或者 Form→Lay Out Vertically in Splitter(在窗口切分条中纵向布局)。

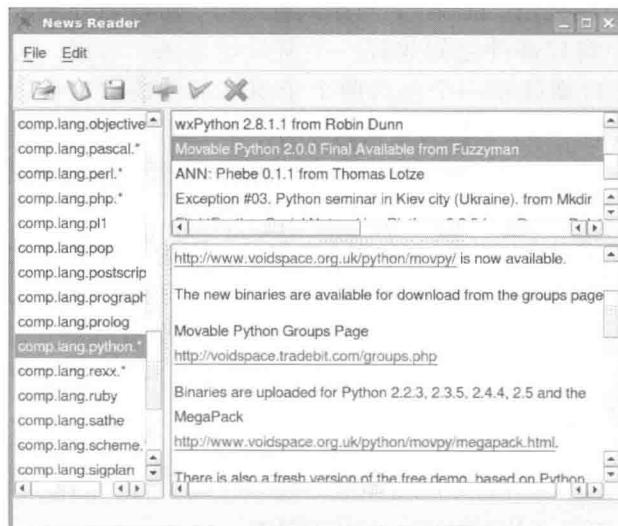


图 9.5 模拟的新闻阅读器应用程序

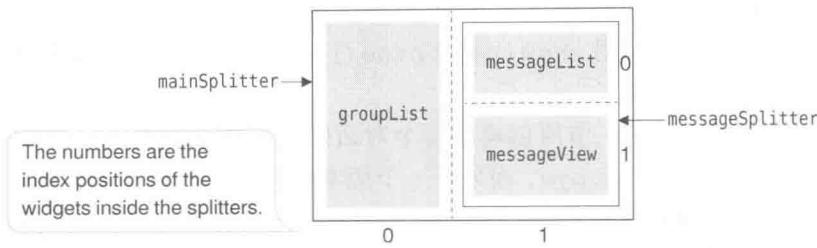


图 9.6 新闻阅读器的各个窗口切分条和子窗口部件

在这一节将会介绍如何用手写代码的形式来创建窗口切分条，包括该如何保存和恢复它们的相对位置。这里先从模拟的新闻阅读器初始化程序的有关部分开始。

```
class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.groupsList = QListWidget()
        self.messagesList = QListWidget()
        self.messageView = QTextBrowser()
```

按照惯例，这个初始化程序会先调用 `super()`。随后的三行代码则稍与常规不同，因为尽管这是一个主窗口，但这里还是会去创建三个窗口部件而不是只创建一个。

```
self.messageSplitter = QSplitter(Qt.Vertical)
self.messageSplitter.addWidget(self.messagesList)
self.messageSplitter.addWidget(self.messageView)
self.mainSplitter = QSplitter(Qt.Horizontal)
self.mainSplitter.addWidget(self.groupsList)
self.mainSplitter.addWidget(self.messageSplitter)
self.setCentralWidget(self.mainSplitter)
```

现在要创建两个窗口切分条。第一个是 `messageSplitter`；这个窗口切分条中在纵向方向上分别保存着消息列表和消息查看两个窗口部件，一个在上，另一个在下。第二个窗口切分条是 `mainSplitter`，可以在水平方向上进行切分，其左边是群组列表窗口部件 `groupsList`，右边是消息切分条 `messageSplitter`。与布局框一样，窗口切分条可以容纳多个子窗口部件，此时就会在每两个窗口部件之间放置一个窗口切分条。所以，这里的主切分条 `mainSplitter` 由一个子窗口部件和一个包含两个子窗口部件的窗口切分条组成，最终 `mainSplitter` 会包含所有的一切，又因为这些窗口切分条也是窗口部件（但不同于布局框，因为各个布局框都是布局），就像这里所做的一样，可以把一个窗口切分条当成一个主窗口的中央窗口部件。

一些用户可能会发现窗口切分条也有恼火的地方，因为只有通过鼠标才能够对它们进行尺寸大小的调整。这里会通过保存和恢复用户的窗口切分条尺寸大小的方式来尽可能地减少这一令人郁闷的问题。这样做很有效果，因为用户只需对它们设置一次，从此以后，这些尺寸大小就可以一直有效了。

```
settings = QSettings()
size = settings.value("MainWindow/Size",
                      QVariant(QSize(600, 500))).toSize()
self.resize(size)
position = settings.value("MainWindow/Position",
                          QVariant(QPoint(0, 0))).toPoint()
self.move(position)
```

```
self.restoreState()
    settings.value("MainWindow/State").toByteArray())
self.messageSplitter.restoreState()
    settings.value("MessageSplitter").toByteArray())
self.mainSplitter.restoreState()
    settings.value("MainSplitter").toByteArray())
```

保存用户主窗口的各个状态设置的代码可以与之前见过的用于恢复窗口所拥有的任意工具栏和各个停靠窗口的尺寸大小及位置的那些代码相似。窗口切分条也有状态，可以像主窗口的状态一样进行状态的保存和恢复。

```
def closeEvent(self, event):
    if self.okToContinue():
        settings = QSettings()
        settings.setValue("MainWindow/Size", QVariant(self.size()))
        settings.setValue("MainWindow/Position",
            QVariant(self.pos()))
        settings.setValue("MainWindow/State",
            QVariant(self.saveState()))
        settings.setValue("MessageSplitter",
            QVariant(self.messageSplitter.saveState()))
        settings.setValue("MainSplitter",
            QVariant(self.mainSplitter.saveState())))
    else:
        event.ignore()
```

在主窗口的关闭事件中，开头的代码仍旧与先前的代码一样，只不过现在不仅要保存主窗口的尺寸大小、位置和状态，还要保存窗口切分条的相应状态。

在第一次启动新闻阅读器应用程序时，根据默认值，主窗口切分条 mainSplitter 会严格将其一半宽度给群组列表窗口部件 groupsList，并将另一半宽度给消息窗口切分条 messageSplitter。相应地，消息窗口切分条 messageSplitter 会将其一半高度给消息列表窗口部件 messagesList，而将另一半高度给消息查看窗口部件 messageView。要想改变这些比例，比如要让群组列表更窄、消息查看器更高，就可以通过对它们使用伸展因子来实现。例如：

```
self.mainSplitter.setStretchFactor(0, 1)
self.mainSplitter.setStretchFactor(1, 3)
self.messageSplitter.setStretchFactor(0, 1)
self.messageSplitter.setStretchFactor(1, 2)
```

setStretchFactor()的第一个参数是从 0 开始的窗口部件的位置索引值（各个窗口部件的顺序都是从 0 开始的，且是从左到右或者从上到下的），第二个参数是要应用的伸展因子值。在这个例子中，之前曾说过第 0 个窗口部件（即群组列表窗口部件 groupsList）的伸展因子应该是 1 且第 1 个窗口部件（消息窗口切分条 messageSplitter）的伸展因子应该是 3。相应地，对于消息窗口切分条来说，在竖直方向上会将空间切分成 1:2 的比例，即消息查看窗口部件 messageView 的高度比例为 2。由于可以保存和恢复窗口切分条的尺寸大小，这些伸展因子只会在应用程序第一次启动时才会有效。

## 9.4 单文档界面(SDI)

对于某些应用程序而言，用户希望能够用其处理多个文档。通常可以通过运行该应用程序的多个实例来实现这一功能，但是这样将会耗用大量资源。这一做法的另外一个缺点是，将

无法通过同一个通用菜单来在各个不同的文档之间进行导航。

对于这一问题通常有三种解决方法。第一种方法是，在单个主窗口中使用 Tab 标签窗口部件，在每个标签页面仅处理一个文档。这种解决方法在 Web 浏览器中非常流行，但在进行文档的编辑处理时并不方便，因为每次不可能同时查看两个或者多个文档。这里不会再对这种解决方法进行介绍，因为有关 Tab 标签窗口部件的内容在 9.2 节中的介绍已经足够多了，并且在课后的练习题中还留有动手练习的题目。另外的两种解决方法将分别是在这一节介绍的单文档界面 SDI 和下一节介绍的多文档界面 MDI。

创建一个 SDI 应用程序的关键在于创建一个窗口子类，使其处理所有操作，包括加载、保存以及清空，从而减少应用程序中那些个基本的窗口集。

这里将先从查看 SDI 文本编辑器(Text Editor)初始化程序的代码开始；这个应用程序如图 9.7 所示。

```
class MainWindow(QMainWindow):
    NextId = 1
    Instances = set()

    def __init__(self, filename=QString(), parent=None):
        super(MainWindow, self).__init__(parent)
        self.setAttribute(Qt.WA_DeleteOnClose)
        MainWindow.Instances.add(self)
```

这里的静态变量 `NextId` 用来提供一个表示新建空白窗口数量的数字：就是“`Unnamed-1.txt`”、“`Unnamed-2.txt`”等新建窗口中的数字。

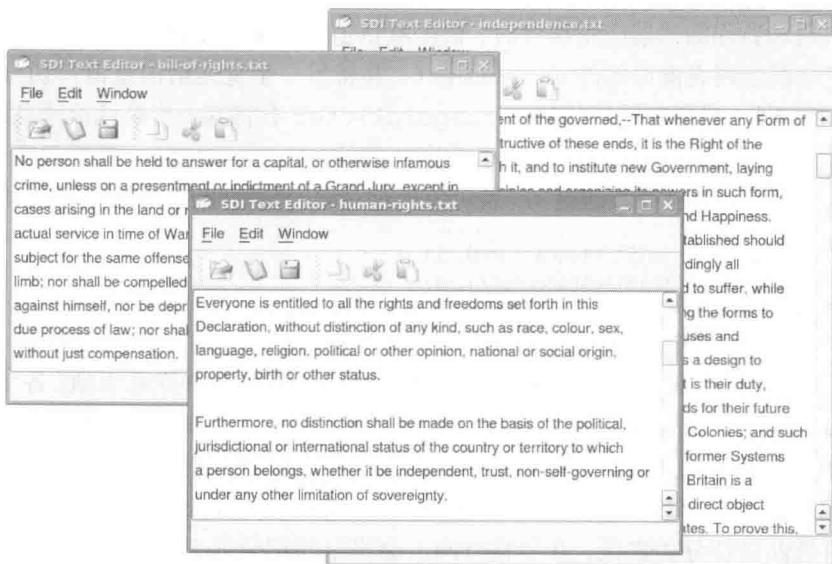


图 9.7 打开了三个文档的 SDI 文本编辑器

这个应用程序由一个或者多个 `MainWindow` 实例构成，每个实例都必须能够独立运行。然而，要从所有这些实例中的任何一个来访问全部这些实例，通常会有三种常见情况。第一种情况是，提供一个“`save all`”选项；另外一种是，提供一个 `Window` 菜单，用户可以通过这个菜单在任意两个窗口实例之间进行切换；还有一种是，提供一个“`quit`”选项，让用户可以借助它

来终止应用程序，这样也就隐含着必须要关闭每一个窗口了。这里的静态变量 Instances 可以用来记录这些全部实例。

在创建一个新窗口实例时，会对其进行设置，以便可以在关闭时能够删除自己。这就意味着，这些窗口可以直接被用户或者间接被其他实例（例如，当关闭该应用程序时）所关闭。一种情况是使用 Qt.WA\_DeleteOnClose，会让该窗口自动注意保存那些未保存的修改并清空自己。也可以把该窗口添加到窗口实例的静态集中，以便可以让任意一个窗口实例都可以访问其他的全部窗口。后面将会进一步深入研究这些问题。

```
self.editor = QTextEdit()
self.setCentralWidget(self.editor)
```

QTextEdit 是最为理想的中央窗口部件，能够把一些动作直接传递给它，这一点很快就可以看到。现在先看一些这样的动作，并会跳过之前已经看到过的 createAction()方法的创建过程。

```
fileSaveAllAction = self.createAction("Save A&ll",
                                      self.fileSaveAll, icon="filesave",
                                      tip="Save all the files")
```

这个动作与其他所有的文件操作动作都相似，会用一个连接将其与 MainWindow 子类的一些方法相连接。

```
fileCloseAction = self.createAction("&Close", self.close,
                                     QKeySequence.Close, "fileclose",
                                     "Close this text editor")
```

“Close”动作和之前看到的那些关闭动作也相似。通常，不需要重新实现 close() 方法，而是直接重新实现 closeEvent() 处理程序，以便可以截获全部的窗口干净关闭事件。不同之处在于，这个关闭动作仅关闭当前窗口，而不会关闭该应用程序（除非这个窗口是该应用程序的唯一一个窗口）。

```
fileQuitAction = self.createAction("&Quit", self.fileQuit,
                                    "Ctrl+Q", "filequit", "Close the application")
```

“Quit”动作会终止应用程序，因而也就会关闭 SDI 文本编辑器应用程序的每一个窗口，正如在查看 fileQuit() 方法时所看到的结果一样。

```
editCopyAction = self.createAction("&Copy", self.editor.copy,
                                    QKeySequence.Copy, "editcopy",
                                    "Copy text to the clipboard")
```

这个动作会连接到 QTextEdit 的相关槽上。这个动作与“cut”和“paste”动作一样。

除了窗口的菜单之外，窗口中的菜单、工具栏和状态栏的创建方式与先前的创建方式完全一样，现在就来看看窗口 window 的菜单吧。

```
self.windowMenu = self.menuBar().addMenu("&Window")
self.connect(self.windowMenu, SIGNAL("aboutToShow()"),
            self.updateWindowMenu)
```

并没有向窗口 Window 菜单中添加任何动作。相反，这里只是简单地将菜单的 aboutToShow() 方法连接到很快将要看到的自定义 updateWindowMenu() 方法上，该方法会将菜单装配到所有的 SDI 文本编辑器应用程序的窗口上。

```
self.connect(self, SIGNAL("destroyed(QObject*)"),
            MainWindow.updateInstances)
```

当用户关闭一个窗口时，借助于 Qt.WA\_DeleteOnClose 标志，将会删除该窗口对象。不过，

由于之前曾对静态集 `Instances` 中该窗口进行过引用，所以这个窗口对象不会被作为垃圾回收掉。因此，需要把该窗口的 `destroyed()` 信号连接到一个能够对 `Instances` 集进行更新的槽函数上，而这个槽函数会移除那些已经关闭了的窗口对象。在接下来查看 `updateInstances()` 方法的时候，将会更深入地对此予以讨论。

由于每个窗口都会负责一个单独的文件，故而可以使每个文件名只关联一个窗口。这个文件名将被作为参数传递给窗口的初始化程序，默认情况下这个文件名会是一个空的 `QString` 字符串。初始化程序的最后几行就是对文件名的处理。

```
self.filename = filename
if self.filename.isEmpty():
    self.filename = QString("Unnamed-%d.txt" % \
                           MainWindow.NextId)
    MainWindow.NextId += 1
    self.editor.document().setModified(False)
    self.setWindowTitle("SDI Text Editor - %s" % self.filename)
else:
    self.loadFile()
```

如果这个窗口没有文件名，要么是因为应用程序刚刚启动，要么是因为用户调用了“file new”动作，这样就需要给窗口创建一个合适的标题；否则，就会载入给定的文件。

这里的 `closeEvent()`、`loadFile()`、`fileSave()` 和 `fileSaveAs()` 方法都与之前见过的相似，所以这里不再赘述（当然，这些方法的源代码可以在文件 `chap09/sditexteditor.pyw` 中找到）。相反，这里将会重点关注这些方法在单文档界面程序中的特殊之处。

```
def fileNew(self):
    MainWindow().show()
```

当用户调用“file new”动作时，就会调用这个 `fileNew()` 方法。还会再创建一个该类的实例，并会对其调用 `show()`（因而会显示成非模态的形式）。在该函数的最后，由于这个窗口没有任何的 PyQt 父对象，并且也不是实例变量，所以希望能够在超出作用域后就销毁掉这个窗口。不过，在主窗口的初始化程序中，该窗口对象会将自己添加到静态集 `Instances` 中，因而依然存在一个指向该窗口的对象引用，所以也就不会销毁该窗口。

```
def fileOpen(self):
    filename = QFileDialog.getOpenFileName(self,
                                           "SDI Text Editor -- Open File")
    if not filename.isEmpty():
        if not self.editor.document().isModified() and \
           self.filename.startsWith("Unnamed"):
            self.filename = filename
            self.loadFile()
        else:
            MainWindow(filename).show()
```

这个方法和之前看到的那些方法稍有区别。如果用户给定了一个文件名，并且当前文档既没有修改过也没有重命名过（也就是说，是一个新的空白文档），那么就将文件载入到已有的窗口中；否则，会创建一个新的窗口，并把要载入的文件名传递给它。

```
def fileSaveAll(self):
    count = 0
    for window in MainWindow.Instances:
        if isAlive(window) and \
           window.editor.document().isModified():
```

```

if window.fileSave():
    count += 1
    self.statusBar().showMessage("Saved %d of %d files" % (
        count, len(MainWindow.Instances)), 5000)

```

基于对用户的负责，这里会提供一个 Save All 菜单选项。在它被调用时，就会遍历 Instances 集中的每一个窗口，并保存其中每一个处于“存活”(alive)和修改状态的窗口。

如果一个窗口没有被删除，那么这个窗口就是存活的(alive)。遗憾的是，这并不像看起来那么简单。与一个 QWidget 相关的生命周期有两个：一个生命周期是引用这个窗口部件的 Python 变量(在这个例子中，就是 Instances 集中的每一个 MainWindow 实例)；另一个生命周期是底层的 PyQt 对象，这是一个由计算机窗口系统一直关注的窗口部件。

通常情况下，PyQt 对象及其 Python 变量的生命周期是完全一样的。但在这里或许并不是那么回事。例如，假设启动程序并点击 File→New 多次以便可以创建 3 个窗口，然后导航到其中的一个窗口并将其关闭。这时，关闭的窗口就会同时被删除掉(这要归功于 Qt.WA\_DeleteOnClose 属性)。

实际上，PyQt 实际是对删除了的窗口调用了 deleteLater() 方法。这样就会给这个窗口一个结束所有正在执行的操作机会，以便可以让这个窗口能够干净地删除掉。所有这些操作通常可以在 1 毫秒内完成，此时，底层的 PyQt 对象就会从内存中删除掉，也不会再存在了。但是，Instances 集中的 Python 引用将依然存在，只是现在，所引用的 PyQt 对象不存在而已。因此，在访问 Instances 集中的任何窗口之前，都必须先检查其中窗口的存活状态。

```

def isAlive(qobj):
    import sip
    try:
        sip.unwrapinstance(qobj)
    except RuntimeError:
        return False
    return True

```

sip 模块是 PyQt 所支持的模块之一，通常不需要直接访问这个模块。但在某些需要深入使用的情况下，这个模块会非常有用。在这里，这个方法就可以用来尝试访问一个变量的底层 PyQt 对象。如果这个对象已经被删除了，就会抛出一个 RuntimeError 异常，此时返回的就会是 False；否则，这个对象依然存在，并且返回 True<sup>①</sup>。通过执行这一检查过程，就可以确保不会无意中访问一个已被关闭并且删除了的窗口，即使指向该窗口的变量还没有被删除掉。

```

@staticmethod
def updateInstances(qobj):
    MainWindow.Instances = set([window for window \
        in MainWindow.Instances if isAlive(window)])

```

一个窗口无论何时关闭(并且因而会删除掉)，都会发射一个 destroyed() 信号，这样就可以把这个信号连接到初始化程序中的 updateInstances() 方法上。这个方法会用一个只含有依旧存活的窗口实例集来重写 Instances 集。

那么，为什么在遍历这些窗口实例时还需要检查各个窗口的存活状态——例如，在 file-

<sup>①</sup> isAlive() 函数是基于 Giovanni Bajo 的 PyQt(随后是 PyKDE) 邮件列表中发布的帖子，“How to detect if an object has been deleted”。这个邮件列表既用于 PyQt 也用于 PyKDE。

`SaveAll()`方法中——因为这个方法确保的 `Instances` 集已经是最新的了，并且该集也只会保存那些仍旧存活的窗口。这是因为，理论上仍旧存在一种可能，就是在窗口关闭与恰好更新 `Instances` 集的时间之间，该窗口却被其他的方法遍历了。

用户在任何一个 SDI 文本编辑器窗口中点击了其中的 `Window` 菜单，就会出现一个所有当前窗口的列表。之所以会出现这个，是因为 `windowMenu` 的 `aboutToShow()` 信号连接到了 `updateWindowMenu()` 槽上，而该槽正是用来更新菜单的。

```
def updateWindowMenu(self):
    self.windowMenu.clear()
    for window in MainWindow.Instances:
        if isAlive(window):
            self.windowMenu.addAction(window.windowTitle(),
                                      self.raiseWindow)
```

首先，清空所有已经存在的菜单条目；不过，其中总会至少有一个条目，即当前窗口。接下来，遍历所有窗口实例并向每个仍旧存活着的窗口添加一个动作。这个动作的文本内容会含有一个说明窗口的标题（也就是文件名）和一个当用户调用菜单项时所执行的槽——`raiseWindow()`。

```
def raiseWindow(self):
    action = self.sender()
    if not isinstance(action, QAction):
        return
    for window in MainWindow.Instances:
        if isAlive(window) and \
           window.windowTitle() == action.text():
            window.activateWindow()
            window.raise_()
            break
```

这个方法应当可以被 `Window` 菜单中的任何一个条目所调用。先从完整性检查开始，然后会遍历所有窗口实例，从中找出哪个窗口实例标题的文本能够与这个动作的文本内容相匹配。如果能够找到一项匹配，就让该窗口关注该“激活”窗口（应用程序中拥有键盘焦点的顶层窗口），并将该窗口置顶到其他的窗口之上，以便能够让用户看到它。

在接下来的 MDI 一节将会看到如何创建一个更为复杂的 `Window` 菜单，其中还会带有一些快捷键和额外的菜单选项

```
def fileQuit(self):
    QApplication.closeAllWindows()
```

PyQt 提供了一种快捷方法来关闭应用程序的所有顶层窗口。这个方法会对所有窗口调用 `close()`，从而会让每个窗口都产生一个 `closeEvent()`。在这个事件（这里没有给出）中，会检查 `QTextEdit` 的文本是否存在有尚未保存的修改，而如果有，就会弹出一个对话框来询问用户是否打算保存。

```
app = QApplication(sys.argv)
app.setWindowIcon(QIcon(":/icon.png"))
MainWindow().show()
app.exec_()
```

在 `sditexteditor.pyw` 文件的最后，会创建一个 `QApplication` 实例和一个单独的 `MainWindow` 实例，然后就会开始事件的循环。

SDI 方法的使用很流行，但也有不少弊端。由于每个主窗口都有自己的菜单栏、工具栏和

一些停靠窗口，相较于单个主窗口程序而言，会耗用更多的资源<sup>①</sup>。同时，尽管使用 Window 菜单可以很方便地在不同窗口之间进行切换，但如果打算更精细地控制窗口的尺寸大小和位置，就必须自己手动编写代码。这些问题将可以在接下来要介绍的不那么流行的 MDI 方法中得到解决。

## 9.5 多文档界面(MDI)

多文档界面相较于单文档界面或者是运行多个应用程序实例这两种解决方案而言，具有诸多优点。多文档界面应用程序资源占用更少，能够为用户提供更为简单的、可实现多个文档窗口之间相互关联的布局功能。不过，它也有一个小缺点，就是无法在多文档界面窗口之间使用 Alt + Tab 键（在 Mac OS X 上是 Command + Tab 键）进行切换，尽管在实际使用中这并不是什么问题，因为对于多文档界面应用程序来说，程序开发人员总是会不约而同地都会用一个 Window 菜单来实现各个窗口之间的导航。

创建一个多文档界面应用程序的关键是创建一个窗口部件子类，并在这个窗口部件子类中实现包括加载、保存以及清空等一切功能，同时在应用程序的多文档界面“工作区”（workspace）中保存一些窗口部件，再给它们传递一些窗口部件相关的动作。

在这一节，将会创建一个能够提供与上一节 SDI 文本编辑器（Text Editor）一样功能的文本编辑器，只是在这一次，要做的将是一个 MDI 应用程序。这个应用程序的效果如图 9.8 所示。

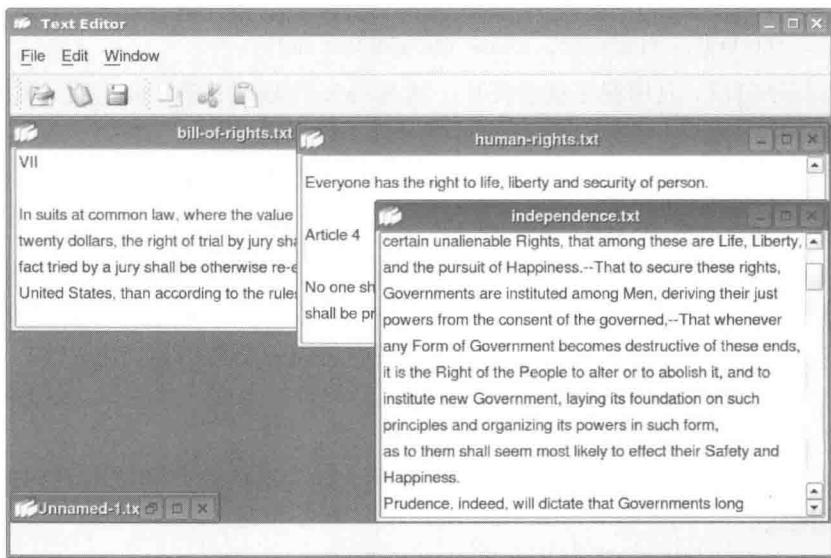


图 9.8 含有 4 个文档的多文档编辑器

显示和编辑的每个文档都使用了一个自定义的 QTextEdit 窗口部件，这是 QTextEdit 的一个子类。这个窗口部件拥有 Qt.WA\_DeleteOnClose 属性设置，也有一个文件名实例变量，还可以加载和保存成文件名所给定的文件。如果该窗口部件被关闭（因而也会被删除掉），它的关闭事件处理程序就会给用户一个保存那些尚未保存的修改的机会。TextEdit 的实现

<sup>①</sup> 在 Mac OS X 上只有一个菜单栏，位于屏幕的上方。它会发生变化来反映出是哪个窗口当前正拥有着键盘焦点。

程序非常简单，与之前所见过的代码相当类似，因此这里就不再回顾了；它的源代码放在文件 chap09/textedit.py 中。

用于这个应用程序的代码相应放在文件 chap09/texteditor.pyw 中。这里将会查看一下这些代码，先从 `MainWindow` 子类的初始化程序的代码开始。

```
class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.mdi = QWorkspace()
        self.setCentralWidget(self.mdi)
```

PyQt 的多文档界面窗口部件称为 `QWorkspace`<sup>①</sup>。与 Tab 标签页窗口部件或者堆叠窗口部件一样，也可以向一个 `QWorkspace` 中添加多个窗口部件。这些窗口部件会在工作区中进行布局，简直就像一个操作系统的微型桌面，用户可以对工作区内的这些窗口部件进行平铺、层叠、图标化或者拖拽以及重新改变尺寸大小等操作。

通过调用 `QWorkspace.setScrollBarsEnabled(True)`，有可能让一个工作区比其窗口拥有更大的空间。工作区的背景颜色可以通过背景画刷进行设定。

```
fileNewAction = self.createAction("&New", self.fileNew,
    QKeySequence.New, "filenew", "Create a text file")
```

绝大部分文件动作的创建和先前看到的创建方式一样。但是很快将会看到，多文档编辑器和单文档编辑器一样，没有 `okToContinue()` 方法，因为每个文档窗口都会自己管理自己。

```
fileQuitAction = self.createAction("&Quit", self.close,
    "Ctrl+Q", "filequit", "Close the application")
```

如果关闭应用程序窗口，应用程序就会终止。所有的文档窗口也将会被关闭，而任何尚未保存的修改都会负责提示用户并在需要保存的时候对其予以保存。

```
editCopyAction = self.createAction("&Copy", self.editCopy,
    QKeySequence.Copy, "editcopy",
    "Copy text to the clipboard")
```

在 SDI 单文档编辑器中，会把“copy”、“cut”和“paste”动作传递到每个窗口的 `QTextEdit` 上进行处理。但是，在多文档编辑器应用程序中，这样做是无法实现的，因为当用户在触发这些动作中的任何一个时，都必须将这个动作应用到其中任何一个文档窗口上。

`TextEdit` 窗口处于激活状态。为此，主窗口必须自己完成一些工作，在查看这些动作的实现代码时将会看到这一点。

这里没有给出用于其他文件和编辑动作的代码，因为这些代码和之前看到的那些动作采用的是同样的模式。

```
self.windowNextAction = self.createAction("&Next",
    self.mdi.activateNextWindow, QKeySequence.NextChild)
self.windowPrevAction = self.createAction("&Previous",
    self.mdi.activatePreviousWindow,
    QKeySequence.PreviousChild)
self.windowCascadeAction = self.createAction("Casca&de",
    self.mdi.cascade)
self.windowTileAction = self.createAction("&Tile",
    self.mdi.tile)
```

<sup>①</sup> 自 Qt 4.3 以来，多文档界面是由 `QMdiArea` 类提供的，其 API 与 `QWorkspace` 的 API 相似。

```

self.windowRestoreAction = self.createAction("&Restore All",
                                             self.windowRestoreAll)
self.windowMinimizeAction = self.createAction("&Iconize All",
                                              self.windowMinimizeAll)
self.windowArrangeIconsAction = self.createAction(
    "&Arrange Icons", self.mdi.arrangeIcons)
self.windowCloseAction = self.createAction("&Close",
                                          self.mdi.closeActiveWindow, QKeySequence.Close)

```

所有的窗口动作都会被创建成实例变量，因为要能够用其他方法访问它们。对于某些动作，可以将工作直接传递给 mdi 工作区实例，但是对于所有多文档界面窗口的最小化和恢复操作都必须要靠我们自己进行处理。

```

self.windowMapper = QSignalMapper(self)
self.connect(self.windowMapper, SIGNAL("mapped(QWidget*)"),
             self.mdi, SLOT("setActiveWindow(QWidget*)"))

```

在接下来即将创建的 Window 菜单中，需要有一些可以让用户选择的窗口变成激活窗口的方法。在上一节中曾经介绍过该问题的一种简单解决方法。另外一种解决方法是使用偏函数应用程序 (partial function application)，把每一个窗口动作都用相应的 `TextEdit` 作为参数并将其连接到 `QWorkspace.setActiveWindow()` 上。这里会采用一种纯 PyQt 的解决方法，并使用之前曾用到过的 `QSignalMapper` 类。在接下来查看 `updateWindowMenu()` 方法的代码时，将会解释这个类的使用方法。

```

self.windowMenu = self.menuBar().addMenu("&Window")
self.connect(self.windowMenu, SIGNAL("aboutToShow()"),
            self.updateWindowMenu)

```

对 `aboutToShow()` 信号的连接可以保证 `updateWindowMenu()` 方法能够在菜单显示之前得到调用。

```

self.updateWindowMenu()
self.setWindowTitle("Text Editor")
QTimer.singleShot(0, self.loadFiles)

```

在构造函数的最后部分，通过调用 `updateWindowMenu()` 来强制创建 Window 菜单。这看上去有些奇怪；毕竟，只要用户一尝试着去使用 Window 菜单，无论如何都会创建它，那为什么还要现在就这样做呢？理由是，如果在应用程序启动时就自动加载了一些文档，用户或许就希望能够使用快捷键 (F6 和 Shift + F6) 来在这些文档之间进行导航，但是，这些个快捷键可能只有在创建了菜单之后才会生效。

```

def closeEvent(self, event):
    failures = []
    for textEdit in self.mdi.windowList():
        if textEdit.isModified():
            try:
                textEdit.save()
            except IOError, e:
                failures.append(str(e))
    if failures and \
        QMessageBox.warning(self, "Text Editor -- Save Error",
                            "Failed to save%s\nQuit anyway?" % \
                            "\n\t".join(failures),
                            QMessageBox.Yes|QMessageBox.No) == QMessageBox.No:
        event.ignore()

```

```

        return
    settings = QSettings()
    settings.setValue("MainWindow/Size", QVariant(self.size()))
    settings.setValue("MainWindow/Position",
                      QVariant(self.pos()))
    settings.setValue("MainWindow/State",
                      QVariant(self.saveState()))
    files = QStringList()
    for textEdit in self.mdi.windowList():
        if not textEdit.filename.startsWith("Unnamed"):
            files.append(textEdit.filename)
    settings.setValue("CurrentFiles", QVariant(files))
    self.mdi.closeAllWindows()

```

在应用程序终止时，要给用户一次保存那些尚未保存修改的机会。然后会保存主窗口的尺寸大小、位置和状态。还同样需要保存一个含有全部多文档窗口所有文件名的列表。最后，调用 `QWorkspace.closeAllWindows()`，它会让每个窗口都收到一个关闭事件。

如果有任何东西出现保存失败，那么就将其记录在案，而在所有文件处理完毕之后，如果还是存在错误，那么就弹出一个消息框，通知用户并给他们一个能够取消终止应用程序的机会。

在 `TextEdit` 的关闭事件中，有一些代码用来给用户提供一次保存那些尚未保存修改的机会，不过在这里，却没有任何这样的代码，因为就是在这个方法的一开始，就已经有了对那些未保存的修改进行保存的处理。在 `TextEdit` 的关闭事件中有这些代码是因为用户可以在任何时候关闭任何窗口，因而每个窗口都必须要能够应付被迫关闭的问题。但是，当应用程序终止时不会这样做，相反，对于那些修改了的文件还会调用 `save()`，因为希望保有一个当前文件列表，而要实现这一点，在执行到保存当前文件列表和调用 `save()` 代码之前，每个文件就必须都要有一个适当的文件名，这样才能够实现这一功能。

```

def loadFiles(self):
    if len(sys.argv) > 1:
        for filename in sys.argv[1:31]: # Load at most 30 files
            filename = QString(filename)
            if QFile.exists(filename):
                self.loadFile(filename)
                QApplication.processEvents()
    else:
        settings = QSettings()
        files = settings.value("CurrentFiles").toStringList()
        for filename in files:
            filename = QString(filename)
            if QFile.exists(filename):
                self.loadFile(filename)
                QApplication.processEvents()

```

对这个应用程序进行这样的设计是为了能够让它加载该应用程序上一次运行时所打开的全部文件。不过，如果用户是在命令行中给定一个或者多个文件的，就需要忽略先前打开的那些文件，然后再只打开用户所给定的那些文件。在这个例子中，已经设定打开文件的上限为 30，要避免用户在一个含有成百上千文件的目录中无意识地使用通配符 `*.*` 来作为文件的规格说明。

这里的 `QApplication.processEvents()` 调用会暂时将控制权返还给事件循环，以便使那些累积的任何事件（比如一些绘制事件）都可以得到处理。然后，会从下一个语句开始恢

复处理进程。在这个运行的应用程序中之所以这样做的效果是，在加载每个文件之后都会即刻弹出一个文档编辑窗口，而不是在所有的文件都加载完毕后才显示出所有的窗口。这样就可以让用户非常清楚地知道应用程序正在处理的事情，而一开始就出现一个很长的延时，可能会让用户觉得应用程序已经崩溃了。使用 `processEvents()` 的另外一个好处是，用户的鼠标和键盘事件将可以获取一些处理器时间片，即使同时还有大量的其他进程正在运行，也能够让应用程序保持响应。

使用 `processEvents()` 来让应用程序在耗时操作期间保持响应要比使用线程进程简单得多。虽然如此，也必须要谨慎使用，因为它可能会导致那些需要长时间运行的操作被当成是造成问题的处理事件。有助降低这一风险的一种解决方法是传递一些额外的参数——例如，对应当处理的事件的类型标志的限制，或者是花在事件处理上的最大时间等。在第 12 章还会看到另外一个使用 `processEvents()` 的例子；多线程则是第 19 章的主题。

```
def loadFile(self, filename):
    textEdit = textedit.TextEdit(filename)
    try:
        textEdit.load()
    except (IOError, OSError), e:
        QMessageBox.warning(self, "Text Editor -- Load Error",
                            "Failed to load %s: %s" % (filename, e))
        textEdit.close()
        del textEdit
    else:
        self.mdi.addWindow(textEdit)
        textEdit.show()
```

在载入一个文件时，要么是 `loadFiles()`，要么是 `fileOpen()` 的执行结果，会用给定的文件名创建一个新的 `TextEdit`，并让该编辑器再载入这个文件。如果文件载入失败，会用一个消息窗口通知用户，并关闭和删除该编辑器。如果文件载入成功，该编辑器就会添加到工作区并予以显示出来。这里不再需要用静态变量 `instances` 来让各个 `TextEdit` 实例持续存活，因为 `QWorkspace` 会自动处理这些。

```
def fileNew(self):
    textEdit = textedit.TextEdit()
    self.mdi.addWindow(textEdit)
    textEdit.show()
```

这个方法仅创建了一个新的编辑器，将其添加到工作区并予以显示。这个编辑器的窗口标题为“`Unnamed-n.txt`”，这里的 `n` 是一个从 1 开始的递增整数值。如果用户输入了任何文本并试图关闭或保存该编辑器，就会提示用户要选择一个合适的文件名。

```
def fileOpen(self):
    filename = QFileDialog.getOpenFileName(self,
                                           "Text Editor -- Open File")
    if not filename.isEmpty():
        for textEdit in self.mdi.windowList():
            if textEdit.filename == filename:
                self.mdi.setActiveWindow(textEdit)
                break
        else:
            self.loadFile(filename)
```

如果用户选择打开一个文件，就需要先检查一下，看这个文件是否已经存在于工作区中的任一编辑器之中。如果已经存在，只需激活所在的编辑器窗口。否则，就将该文件载入到一个新的

编辑器窗口中。如果用户希望能够多次载入同一个文件(例如, 想查看一个长文件的不同部分), 那么只需每次简单调用 `loadFile()`, 而不需要担心该文件是否已经存在于已有的编辑窗口之中。

```
def fileSave(self):
    textEdit = self.mdi.activeWindow()
    if textEdit is None or not isinstance(textEdit, QTextEdit):
        return
    try:
        textEdit.save()
    except (IOError, OSError), e:
        QMessageBox.warning(self, "Text Editor -- Save Error",
                            "Failed to save %s: %s" % (textEdit.filename, e))
```

当用户触发“file save”动作时, 通过调用 `QWorkspace.activeWindow()` 可以判定他们打算保存的是哪个文件。如果返回的是一个 `TextEdit`, 就对其调用 `save()` 即可。

```
def fileSaveAll(self):
    errors = []
    for textEdit in self.mdi.windowList():
        if textEdit.isModified():
            try:
                textEdit.save()
            except (IOError, OSError), e:
                errors.append("%s: %s" % (textEdit.filename, e))
    if errors:
        QMessageBox.warning(self, "Text Editor -- Save All Error",
                            "Failed to save\n%s" % "\n".join(errors))
```

为方便起见, 这里提供了一个“save all”动作。由于可能会有许多个窗口, 如果在保存一个文件时出现了问题(例如, 磁盘空间不足), 这个问题就可能会影响到许多窗口。因此, 不是在每个 `save()` 失败时都提供一个错误消息, 而是将这些错误都收集在一个错误列表中, 如果最后确实有一些错误了, 就一起显示出来。

```
def editCopy(self):
    textEdit = self.mdi.activeWindow()
    if textEdit is None or not isinstance(textEdit, QTextEdit):
        return
    cursor = textEdit.textCursor()
    text = cursor.selectedText()
    if not text.isEmpty():
        clipboard = QApplication.clipboard()
        clipboard.setText(text)
```

这个方法与前一个方法的开头一样——对那些应用到某个特定窗口的所有方法的开头都一样——利用获取用户正在工作的编辑器。`QTextEdit.textCursor()` 所返回的 `QTextCursor` 在程序语义上相当于用户使用的光标, 但它又与用户的光标无关; 这个类会在第 13 章中给出完全深入的讨论。如果有选中的文本, 就将这些文本复制到操作系统的全局剪贴板中<sup>①</sup>。

```
def editCut(self):
    textEdit = self.mdi.activeWindow()
    if textEdit is None or not isinstance(textEdit, QTextEdit):
        return
```

<sup>①</sup> X Window 操作系统会使用两个剪贴板: 一个是默认剪贴板, 一个是鼠标选择剪贴板。Mac OS X 也有一个“查找”(Find) 剪贴板。PyQt 会通过对 `setText()` 和 `text()` 使用一个作为第二个可选参数的“mode”来提供对所有可用剪贴板的访问。

```

cursor = textEdit.textCursor()
text = cursor.selectedText()
if not text.isEmpty():
    cursor.removeSelectedText()
    clipboard = QApplication.clipboard()
    clipboard.setText(text)

```

这个方法基本上与 copy 方法完全一样。唯一的区别是如果有选中的文本，就将这些文本从编辑器中移除掉。

```

def editPaste(self):
    textEdit = self.mdi.activeWindow()
    if textEdit is None or not isinstance(textEdit, QTextEdit):
        return
    clipboard = QApplication.clipboard()
    textEdit.insertPlainText(clipboard.text())

```

如果剪贴板中有文本，无论这些文本是来自于这个应用程序中的复制或者剪切操作，还是来自于另外一个应用程序，都会将这些文本插入到编辑器的当前光标位置处。

多文档窗口的所有操作都可以由 QWorkspace 的各个槽提供，因此这里不再需要提供平铺、层叠或者窗口的导航。但是，确实还是需要实现那些用来提供所有窗口最小化和恢复功能的代码。

```

def windowRestoreAll(self):
    for textEdit in self.mdi.windowList():
        textEdit.showNormal()

```

windowMinimizeAll()方法(这里没有给出)一样也是需要重新实现的，只是这里调用的是 showMinimized()方法而不是 showNormal()。

每当调用 QSignalMapper 对象的 map() 槽时都会发射一个 mapped() 信号。传递给 mapped() 信号的参数就是那个对其响应调用了 map() 槽的 QObject。这里使用信号映射程序来将 Window 菜单中的各个动作与 QTextEdit 窗口部件关联起来，以便在用户选择某个特定窗口时，相应的 QTextEdit 能够成为激活窗口。这可以通过两个地方进行设置：一个是在窗体的初始化程序中，另一个是在 updateWindowMenu() 方法内，如图 9.9 所示。

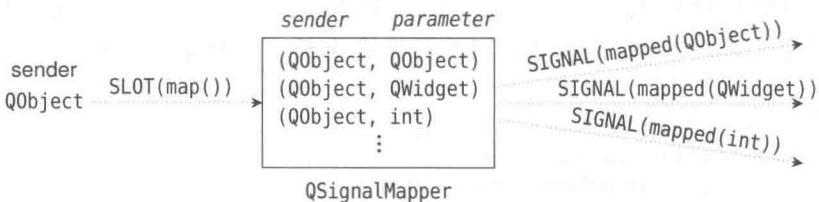


图 9.9 QSignalMapper 的一般操作

在窗体的初始化程序中，创建了一个从信号映射程序的 `mapped(QWidget*)` 信号到 MDI 多文档界面工作区的 `setActiveWindow(QWidget*)` 槽之间的信号-槽连接。要使其起作用，就需要信号映射程序必须能够发射一个信号来响应用户从 Window 菜单中选择的多文档窗口，这可以在 `updateWindowMenu()` 方法中进行设置。MDI 文本编辑器的信号映射程序如图 9.10 所示。

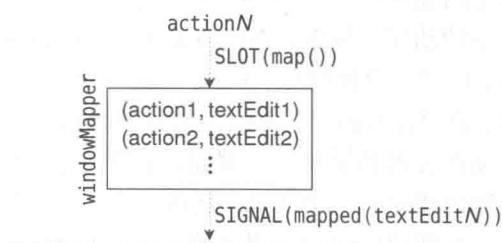


图 9.10 MDI 编辑器的信号映射程序

```

def updateWindowMenu(self):
    self.windowMenu.clear()
    self.addActions(self.windowMenu, (self.windowNextAction,
                                      self.windowPrevAction, self.windowCascadeAction,
                                      self.windowTileAction, self.windowRestoreAction,
                                      self.windowMinimizeAction,
                                      self.windowArrangeIconsAction, None,
                                      self.windowCloseAction))
    textEdits = self.mdi.windowList()
    if not textEdits:
        return

```

这里先从 Window 菜单的全部动作清空开始，然后会重新添加所有的标准动作。接着，获取 QTextEdit 各个窗口的列表；如果一个也没有，那么只需结束并返回；否则，必须为每个窗口添加一个菜单条目。

```

    self.windowMenu.addSeparator()
    i = 1
    menu = self.windowMenu
    for textEdit in textEdits:
        title = textEdit.windowTitle()
        if i == 10:
            self.windowMenu.addSeparator()
            menu = menu.addMenu("&More")
            accel = ""
            if i < 10:
                accel = "&%d " % i
            elif i < 36:
                accel = "&%c " % chr(i + ord("@") - 9)

```

这里遍历了所有的窗口。对于前面的 9 个窗口，会创建一个 &1、&2 这样的“accel”加速键字符串，以便可以生成 1、2、…、9 这样的数字。如果有 10 个或者更多的窗口，可以创建一个文本内容为“More”的子菜单，然后再把第 10 个以及后续的窗口添加到这个子菜单中。对于第 10 个到第 36 个窗口，可以创建一个 &A、&B、…、&Z 这样的“accel”加速键字符串；对于其他的任意窗口都不要提供“accel”加速键字符串（%c 格式的字符串用来说明一个单独的字符）。More 子菜单的加速键仅限于英文；其他语言或许需要不同的处理方法。

```

        action = menu.addAction("%s%s" % (accel, title))
        self.connect(action, SIGNAL("triggered()"),
                     self.windowMapper, SLOT("map()"))
        self.windowMapper.setMapping(action, textEdit)
        i += 1

```

这里用加速键（也可能为空）文本和标题文本（窗口的标题，就是不带路径的文件名）来创建一个新的动作。接着，连接该动作的 triggered() 信号和信号映射程序的 map() 槽。这就意味着，用户无论何时从 Window 菜单中选择一个窗口，这个信号映射程序的 map() 槽都得到调用，值得注意的是，无论是信号还是槽，都没有带任何的参数；这将由信号映射程序来判定哪个动作会得到触发——例如，本应当是该用 sender() 的。在信号-槽连接之后，在信号映射程序内建立了一个映射，是从动作到相应的 QTextEdit。

在调用信号映射程序的 map() 槽时，信号映射程序将会找出是哪个动作调用了它，并会使用这个映射判定出该将哪个 QTextEdit 作为参数进行传递。接着，信号映射程序会用这个参数发射它自己的 mapped(QWidget \*) 信号。把 mapped() 信号连接到多文档界面工作区

的 `setActiveWindow()` 槽，因此就会依次调用这个槽，而作为参数传递的 `TextEdit` 就将变成激活窗口。

这样，就看完了 MDI 多文档界面文本编辑器的代码。这里省略了创建应用程序对象和主窗口的代码，因为这些代码都和先前看到的许多例子的代码一样。

## 小结

当对话框有许多选项时，通常可以通过使用 Tab 标签页窗口部件和堆叠窗口部件来让事情变得为用户可控。当希望让用户能够查看和编辑全部可用选项时，Tab 标签页窗口部件会显得特别有用。而当希望用户仅仅查看当前页面的相关选项时，堆叠窗口部件则比较适合。对于堆叠窗口部件，必须提供一种让用户能够选择当前页面的方式——例如，一个组合框窗口部件或者一个页面名称的窗口部件列表。当对话框中有一些“高级”或者不太常用的选项时，可以使用扩展对话框来隐藏那些额外的选项，除非用户要求查看它们。可选群组框可以用来启用或者禁用它们所包含的窗口部件；如果即使禁用了某些选项，也打算让用户能够看到这些可用的选项，这就很有用。对于某些对话框而言，可以将所有这些方法进行联合应用，尽管在这种复杂情况下，校验逻辑可能会变得相当难以理解。

窗口切分条特别适合用于创建具有多个窗口部件的主窗口，也适合创建让用户可以自由控制各个窗口部件相对尺寸大小的主窗口。一种可供替换的方法是设置一个中央窗口部件，而把其他的窗口部件都放进停靠窗口中。停靠窗口会自动在它们自己和停靠了的中央窗口部件之间放置一个窗口切分条，停靠窗口也可以从一个停靠区域拖动到另一个停靠区域或者自由悬浮着。

单文档界面使得用户可以轻松地用同一个应用程序打开多个文档。同样，单文档界面也使得跨窗口交互成为可能，比如具有“save all”和“quit”动作以及 Window 菜单，这些对于每个文档都要有一个应用程序实例的做法来说并不容易实现。单文档界面方法非常流行，尽管相较于多文档界面来说要更消耗资源，但单文档界面对于那些经验不足的用户来说则要比多文档界面更易理解和简单些。

除了多个文档窗口会限制在一个中央窗口部件内显示之外，多文档界面也提供了众多单文档界面所拥有的优点。这样就可以避免对菜单和工具条的重复复制，也可以在排布相互有关联的窗口时变得更为简单。多文档界面的一个缺点是，某些用户可能发现要比单文档界面更难理解些，至少在刚开始接触时是这样。多文档界面不会把窗口仅限制成一种类型的窗口部件，不过在主窗口中使用多种窗口部件类型的大多数现代多文档界面应用程序都是一种文档窗口类型，而将其他类型的窗口部件放到停靠窗口中。

在单文档界面和多文档界面应用程序的例子中，所有的动作总是有效的。由于所有的动作并没有做出什么有害的事情，貌似这样做并不会出现什么问题。然而，为了避免误导某些用户，最好还是能够根据应用程序的状态来启用和禁用一些动作；在第 13 章中将会说明如何做到这一点。

布局、窗口切分条、Tab 标签页窗口部件、堆叠窗口部件、停靠窗口、单文档界面和多文档界面一起为用户提供了大量的用户界面设计选择。另外，通过手写代码的形式创建自定义的布局，或者是创建自己的布局管理器，这些都是可能的，所以说创意无限。

## 练习题

修改多文档界面文本编辑器应用程序(在文件 `texteditor.pyw` 中)以便不要使用 MDI, 而是使用 `QTabWidget` 并使之成为一个 Tab 标签页式的编辑器。

这样也就不再需要 Window 菜单, 所以就可以将所有与之相关的代码都可以移除掉。将需要一个新的“file close tab”动作, 也需要一个对其进行处理的相应方法。不要使用 `QWorkspace.windowList()`, 而是使用一个从 0 开始到 `QTabWidget.Count()` 结束的迭代循环, 并用 `QTabWidget.widget()` 来依次访问每个窗口。

`closeEvent()` 将需要做些修改, 可能需要用些心思才能获得正确的结果。修改 `loadFiles()`, 使其能够限定命令行中载入文件的最大数为 10。`fileNew()` 将会像之前一样创建一个 `TextEdit`, 然后用 `QTabWidget.addTab()` 将其添加到 Tab 标签页窗口部件中, 用该窗口部件和它的窗口标题做参数。不要对窗口部件调用 `show()`, 而是使用 `QTabWidget.setCurrentWidget()`。`fileOpen()`、`loadFile()`、`fileSave()`、`fileSaveAs()` 和 `fileSaveAll()` 方法也将需要做些小改变。各个编辑方法只需要将它们代码的第一行用 `QWorkspace.activeWindow()` 代替 `QTabWidget.currentWidget()` 即可。

一切就绪后, 增加两个快捷键, `QKeySequence.PreviousChild` 和 `QKeySequence.NextChild`, 提供相应的 `prevTab()` 和 `nextTab()` 方法, 以便使其能够工作。

这些改变仅仅需要数十行代码, 外加大约 20 行用于快捷键和这些方法的代码; 与之前一样, 这里的重点仍旧是要放在思考和理解上, 而不是放在敲代码上。

本练习题的参考答案放在文件 `chap09/tabbededitor.pyw` 中。

# 第 10 章 事件、剪贴板和拖放

- 事件处理机制
- 重新实现事件处理程序
- 使用剪贴板
- 拖放

在这一小节中，会先从描述一些事件处理中的重要概念开始。在 10.2 节，将会在这些概念的基础上给出如何控制窗口部件行为的做法以及如何通过重新实现底层事件处理程序来控制其外观的做法。随后的一些章节，特别是第 11 章，将会以前面两节的内容为基础，给出如何创建自定义窗口部件的做法。

10.3 节将会给出如何使用剪贴板 (clipboard) 的做法，特别是如何从或者向系统全局剪贴板进行普通文本、HTML 和图片的传递与检索的做法。最后一节将会说明如何实现拖放 (drag and drop)，既包括使用 PyQt 的易于使用的内置功能函数，又包括使用我们自己对自定义数据拖放的处理。练习题会涵盖自定义数据的拖放操作，以允许用户在放下操作时选中内容是移动操作还是复制操作。

## 10.1 事件处理机制

PyQt 为事件处理提供了两种机制：高级的信号和槽机制，以及低级的事件处理程序 (event handler)。信号和槽方法非常适合用于对用户希望执行动作的关注，而不会在用户不同特殊需求的细节处理中越陷越深。信号和槽也可以用来对窗口部件的某些行为进行自定义。不过，当需要做出更为深入的研究时，特别是在创建不同的自定义窗口部件时，就需要使用低级的事件处理程序。

PyQt 提供了一系列的事件处理程序，某些关注于窗口部件的行为，比如那些处理键盘按下和鼠标事件的行为，而另一些则关注于窗口部件的外观，比如那些处理绘制事件 (paint event) 和重设尺寸大小事件 (resize event) 的行为等。

PyQt 的事件处理机制可以以我们所期望的逻辑方式进行工作。例如，如果用户在某个拥有键盘焦点的窗口部件上点击鼠标或者按下键盘，就可以给这个窗口部件一个事件。如果该窗口部件可以处理这个事件，那么事情就到此为止。不过，如果该窗口部件不能处理这个事件，事件就会传递到该窗口部件的父窗口部件中——是 PyQt 父-子结构的另一大好处。这样就可以把未处理事件从子窗口部件一直传递到父窗口部件直至顶层窗口部件，而如果还是没能处理掉这个事件，那么就会简单地予以扔掉。

PyQt 为拦截和处理事件提供了 5 种不同方式。头两种方法是应用最为频繁的，而剩下的其他方法则相较而言没有那么常用。

最为简单的方法就是重新实现特定的事件处理程序。不过截至目前，只见到过一个这样的例子：重新实现过窗口部件的 `closeEvent()`。正如在这一章和随后的章节中看到的，通过重新实现一些事件处理程序是可以控制窗口部件的行为的——例如，可以重新实现 `key-`

PressEvent()、mousePressEvent() 和 mouseReleaseEvent()。通过重新实现 resizeEvent() 和 paintEvent()，也可以控制一个窗口部件的外观。当重新实现这些事件时，通常不需要调用基类的事件，因为只是希望我们自己的代码能够仅作为调用事件处理程序的执行结果而已。

在任何特定事件处理程序调用之前，都会调用 event() 事件处理程序。重新实现这个方法可以允许我们处理那些不能在某一特定事件处理程序（特别是，对 Tab 键焦点转换行为的重新定义）中处理的事件，或者实现那些用于不存在明确事件处理程序的事件，比如 QEventToolBarChange。当需要重新实现这些处理程序时，可以对任何不是自己亲自处理的事件调用它们的基类实现。

第 3 种和第 4 种方法会用到事件过滤程序（event filter）。可以对任何 QObject 调用 installEventFilter()。这就意味着，对于 QObject 的全部事件来说，它们都会先传递到我们自己的事件过滤程序：在它们到达目标对象之前，可以抛弃或者修改这些事件中的任何一个。这一方法一种更为强大的版本是，在 QApplication 对象上安装一个事件过滤程序，尽管实践中这样做仅仅是用做程序的调试和处理那些发送给禁用窗口部件的鼠标事件。在一个对象或者 QApplication 上安装多个事件过滤程序也是可以的，此时，它们会按照安装时间从新到旧依次执行。

事件过滤程序提供了一种非常强大的事件处理方式，对于新的 PyQt 编程人员来说，通常需要对它们进行不断尝试。不过，还是建议尽量不要使用事件过滤程序，至少是在你具有丰富的 PyQt 编程经验之前要少使用它们。如果安装了非常多的事件过滤程序，应用程序的性能就会大打折扣；此外，与简单重新实现某一特定事件处理程序或者 event() 处理程序相比，这样做会大大增加代码的复杂性。这里不会看到任何事件过滤程序方面的例子，因为在常规 PyQt 编程中要尽量避免用到它们（而只有在创建自定义窗口部件时它们才会涉及），况且也确实很少有必要用到它们。

第 5 种方法是子类化 QApplication 并重新实现其 notify() 方法。这个方法会在任何事件过滤程序或者事件处理程序之前得到调用，因此它提供了终极控制能力。实践中，仅会在调试时才会用到这一方法，即使如此，事件过滤程序的用法是相当灵活的。

## 10.2 重新实现事件处理程序

在图 10.1 中给出了 QWidget 的一个子类，其中重新实现了一些事件处理程序。这个事件（Event）应用程序在文件 chap10/events.pyw 中，可以报告某些事件和状态，也会用它来看看在 PyQt 中是如何完成事件处理的。后续的章节会使用这里给出的同样技术来完成一些更为复杂和真实的事件处理。这里会先从这个应用程序的初始化程序中所提取的代码开始，看一看它的实例数据。

```
class Widget(QWidget):
    def __init__(self, parent=None):
        super(Widget, self).__init__(parent)
        self.justDoubleClicked = False
        self.key = QString()
        self.text = QString()
        self.message = QString()
```

这里将会说明最近按的键的名称保存在 key 中，而要把要绘制的文本——例如，“The mouse is at...”——放到 text 中，并把一条消息的内容放到 message 中。还会对用户是否执行过双击操作进行追踪。

第一个需要考虑的事件处理程序将是绘制事件。而有关绘制的内容会在第 11 章进行讨论。

```
def paintEvent(self, event):
    text = self.text
    i = text.indexOf("\n\n")
    if i >= 0:
        text = text.left(i)
    if not self.key.isEmpty():
        text += "\n\nYou pressed: %s" % self.key
    painter = QPainter(self)
    painter.setRenderHint(QPainter.TextAntialiasing)
    painter.drawText(self.rect(), Qt.AlignCenter, text)
    if self.message:
        painter.drawText(self.rect(),
                         Qt.AlignBottom|Qt.AlignHCenter, self.message)
    QTimer.singleShot(5000, self.message.clear)
    QTimer.singleShot(5000, self.update)
```

要显示的文本由两个部分构成。第一部分通常会包含着鼠标坐标，第二个部分（可能是空的）会包含用户按下的最后一个键。此外，消息文本或许会被绘制在窗口部件的最下方，因为在这个例子中，单触发计时器（single-shot timer）会在 5 秒后清空消息文本并调度一个绘制事件重绘这个没有消息文本的窗口部件。

在绘制事件中，像这里一样忽略事件的做法是相当常见的（绘制事件可以告诉我们需要重绘的精确区域，以便可以通过只绘制需要更新的区域来优化绘制过程，这一技术将会在第 16 章中看到）。rect() 方法会返回一个带有窗口部件尺寸大小的 QRect，因此只需把文本绘制到这个给定的矩形中心即可。这里没有调用基类的绘制事件；在 PyQt 绘制事件处理程序中，这是工程实践中的一种标准做法，不管怎样，QWidget 绘制事件什么也不做。

```
def resizeEvent(self, event):
    self.text = QString("Resized to QSize(%d, %d)" %
                        event.size().width(),
                        event.size().height())
    self.update()
```

无论何时对窗口部件进行尺寸大小的改变——例如，用户对一个角或者一条边进行了拖拽——都会产生一个尺寸大小改变的事件。会对实例文本进行设置以显示出新的尺寸大小，也会调用 update() 来调度一个绘制事件。尺寸大小事件还可以使用 QResizeEvent.oldSize() 方法找到之前的尺寸大小。这里不会调用基类的尺寸大小事件，因为它什么都不会做。

```
def keyPressEvent(self, event):
    self.key = QString()
    if event.key() == Qt.Key_Home:
        self.key = "Home"
    elif event.key() == Qt.Key_End:
        self.key = "End"
    elif event.key() == Qt.Key_PageUp:
```

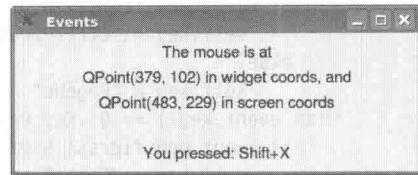


图 10.1 自定义事件处理程序的测试

```

if event.modifiers() & Qt.ControlModifier:
    self.key = "Ctrl+PageUp"
else:
    self.key = "PageUp"
elif event.key() == Qt.Key_PageDown:
    if event.modifiers() & Qt.ControlModifier:
        self.key = "Ctrl+PageDown"
    else:
        self.key = "PageDown"
elif Qt.Key_A <= event.key() <= Qt.Key_Z:
    if event.modifiers() & Qt.ShiftModifier:
        self.key = "Shift+"
        self.key += event.text()
    if self.key:
        self.key = QString(self.key)
        self.update()
else:
    QWidget.keyPressEvent(self, event)

```

如果用户按下键盘，就可以通过 `keyPressEvent()` 得到通知。还有对应的 `keyReleaseEvent()` 方法，不过很少对它进行重新实现。`QKeyEvent` 参数提供了数个有用的方法，包括 `key()` 方法，它会返回所按下的键的编码，还有 `modifiers()` 方法，它会返回一个位标识符，以反映 Shift、Ctrl 和 Alt 的状态。

这里会对 Home、End、PageUp、Ctrl + PageUp、PageDown 和 Ctrl + PageDown 的键序列进行处理，也可以对字母键序列 A…z 以及 Shift + A…Shift + z 进行处理。对于按下的键这样的描述性文本会保存在 `key` 变量中，也会调用 `update()` 来安排一个绘制事件。如果用户按下一个没有处理的键，就会让该键传递给基类的实现，这在对键事件的工程实践中也是常见的一种做法。

```

def contextMenuEvent(self, event):
    menu = QMenu(self)
    oneAction = menu.addAction("&One")
    twoAction = menu.addAction("&Two")
    self.connect(oneAction, SIGNAL("triggered()"), self.one)
    self.connect(twoAction, SIGNAL("triggered()"), self.two)
    if not self.message:
        menu.addSeparator()
        threeAction = menu.addAction("Thre&e")
        self.connect(threeAction, SIGNAL("triggered()"),
                     self.three)
    menu.exec_(event.globalPos())

```

创建上下文菜单最简单的方式就是使用 `QWidget.addAction()` 把动作添加到窗口部件中，再把窗口部件的上下文菜单策略设置成 `Qt.ActionsContextMenu`；在第 6 章中曾看到过是如何完成这一操作的。不过，如果打算要精细控制发生上下文菜单事件时的结果——例如，根据应用程序的状态不同而提供一些不同的选项——可以像这里所做的那样来重新实现上下文菜单事件处理程序。

在调用上下文菜单的时候 `globalPos()` 方法可以返回鼠标的位置；可以把该位置传递给 `QMenu.exec_()`，以确保该菜单可以在用户期望的地方弹出来。

```

def mouseDoubleClickEvent(self, event):
    self.justDoubleClicked = True
    self.text = QString("Double-clicked.")
    self.update()

```

如果用户双击了鼠标，就可以调用这个事件处理程序。在这个例子中，需要不断跟踪用户是否进行了双击操作，因为还会重新实现鼠标松开和鼠标移动事件。鼠标松开事件将会出现在鼠标双击的时候，而鼠标移动事件则几乎一定会在双击的时候出现，因为双击的时候，用户的手绝对不可能是恰好静止不动的。

在其他的事件处理程序中也会采用相同的方法：会对文本进行设置并安排一个重绘事件来显示其内容。对于我们自行处理的鼠标事件不调用其基类的事件处理程序是相当常见的做法。

```
def mouseReleaseEvent(self, event):
    if self.justDoubleClicked:
        self.justDoubleClicked = False
    else:
        self.setMouseTracking(not self.hasMouseTracking())
        if self.hasMouseTracking():
            self.text = QString("Mouse tracking is on.\n"
                                "Try moving the mouse!\n"
                                "Single click to switch it off")
        else:
            self.text = QString("Mouse tracking is off.\n"
                                "Single click to switch it on")
    self.update()
```

如果用户恰好松开了鼠标，而不仅是在双击之后松开了鼠标，就需要打开或者关闭鼠标的状态跟踪。在状态跟踪打开时，鼠标移动会产生所有的鼠标移动事件；而在状态跟踪关闭时，鼠标移动事件则只会在拖动鼠标时产生。默认情况下，鼠标状态跟踪会是关闭的。这里用的是鼠标单击来开关鼠标的状态跟踪功能。正如之前一样，会设置文本的内容，并安排一个绘制事件来显示它。

```
def mouseMoveEvent(self, event):
    if not self.justDoubleClicked:
        globalPos = self.mapToGlobal(event.pos())
        self.text = QString("The mouse is at\nQPoint(%d, %d) "
                            "in widget coords, and\n"
                            "QPoint(%d, %d) in screen coords" %
                            (event.pos().x(), event.pos().y(),
                             globalPos.x(), globalPos.y()))
    self.update()
```

表 10.1 QWidget 的部分事件处理方法

语法	说明
w.closeEvent (e)	重新实现本函数可以给用户一个保存那些未保存修改和用户设置的机会；w 是 QWidget 的一个子类，e 是 QEvent 的一个相关子类
w.contextMenuEvent (e)	重新实现本函数可以提供自定义的上下文菜单。一种更为简洁的替代方法是调用 setContextMenuPolicy(Qt.ActionsContextMenu) 并借助 QWidget.addAction() 向窗口部件添加一些动作
w.dragEnterEvent (e)	重新实现本函数可以用来说明窗口部件是否接受或者拒绝在 QDragEnterEvent e 中进行拖放操作
w.dragMoveEvent (e)	是对可接受的拖放动作的重新实现——例如，对 QDragMoveEvent e 不接受，或者接受一个或者多个移动、复制以及连接等操作事件
w.dropEvent (e)	是对在 QDropEvent e 中进行拖放操作的重新实现

(续表)

语法	说明
w.event(e)	是对没有特定事件处理程序的那些事件处理方法的重新实现——例如，对于 Tab 键的处理。这个事件是从 QObject 继承来的
w.keyPressEvent(e)	是对按键事件的重新实现
w.mouseDoubleClickEvent(e)	是对 QMouseEvent e 中给定的鼠标双击事件响应的重新实现
w.mouseMoveEvent(e)	重新实现来响应在 QMouseEvent e 中给定的鼠标移动事件。这个事件处理程序会受 QWidget.setMouseTracking() 的影响
w.mousePressEvent(e)	重新实现对鼠标按下事件的响应
w.mouseReleaseEvent(e)	重新实现对鼠标松开事件的响应
w.paintEvent(e)	窗口部件绘制事件的重新实现
w.resizeEvent(e)	窗口部件尺寸大小改变事件的重新实现

如果用户打开了鼠标状态跟踪(通过单击鼠标)，将会产生一些鼠标移动事件，并且会在每个鼠标移动事件中都调用这个方法。这里会提取鼠标在屏幕坐标系中的位置，也就是说，是一个相对于屏幕左上角的相对坐标值，而在窗口部件坐标系中，也就是说，将会是该窗口部件左上角的相对坐标值。这两个坐标系在左上角都是(0, 0)，y 坐标轴向下为正，x 坐标轴向右为正。

```
def event(self, event):
    if event.type() == QEvent.KeyPress and \
        event.key() == Qt.Key_Tab:
        self.key = QString("Tab captured in event()")
        self.update()
        return True
    return QWidget.event(self, event)
```

当向窗口部件传递一个事件时，就会先调用该窗口部件的 event() 方法。如果窗口部件可以处理掉该事件，这个方法就会返回 True，否则就会返回 False。在返回 False 的这种情况下，PyQt 就会把这个事件发送给窗口部件的父对象，然后是父对象的父对象，直到它们任何其一返回 True 为止，或者直到达到了顶层(再没有了父对象)，此时，就会扔掉而不再处理这个事件。event() 方法或许会自己处理掉事件，或者会把这项工作委托给诸如 paintEvent() 或者 mousePressEvent() 这样的特定事件处理程序。

当用户按下 Tab 键时，在绝大多数情况下，拥有键盘焦点的窗口部件的 event() 方法都会对 Tab 键次序中的下一个窗口部件调用 setFocus() 并返回 True，而不会将该事件传递给其他的任何一个按键处理程序(QTextEdit 类重新实现了事件处理程序，可以向文本中插入 Tab 键，不过也可以让它重新转回常见的焦点切换行为上)。

通过重新实现键盘事件处理程序，不能停止 Tab 键的键盘焦点切换功能，因为该按键就从来没有传送到它们那里。因此，就必须相应地重新实现 event() 方法并在那里处理 Tab 键的按下事件。

在这个例子中，如果用户按下了 Tab 键，只需更新那些要显示的文本即可。也可以通过返回 True 来说明该事件已经处理过。这样可以防止该事件再被进一步传播下去。对于其他的各类事件，可以调用基类的实现。

实际应用中的事件处理程序通常都要比这里给出的事件处理程序复杂得多，不过，在这里的目的就只是为了能够看到事件处理机制的工作方式是怎样的。在下一章以及后续的多个章节中，将会经常看到诸如 paintEvent() 和 resizeEvent() 事件处理程序的重新实现，也有

对 contextMenuEvent()、wheelEvent()、keyPressEvent() 和 mousePressEvent() 的重新实现，它们都是以实际上下文的形式进行的。在这一章的最后一节，将会重新实现一些与拖拽和放置相关的事件。

### 10.3 使用剪贴板

PyQt 通过剪贴板(clipboard)来支持 QTextEdit、QLineEdit、QTableWidget 以及其他可以编辑文本型数据的窗口部件。PyQt 的剪贴板和拖放系统使用 MIME(多用途网际邮件扩充服务，Multipurpose Internet Mail Extensions)格式的数据，这是一种可以用来存储任意数据的数据格式<sup>①</sup>。

偶然情况下，可以方便地把数据传送到剪贴板或者直接在代码中从剪贴板提取数据。PyQt 让这些变得很简单。 QApplication 类提供了可以返回一个 QClipboard 对象的静态方法，通过这个对象，就可以从剪贴板或者向剪贴板进行文本、图片或者其他数据的写入或者读取。

剪贴板中每次只能保存一个对象，因此，如果对其写入，比方说，一个字符串，然后再接着写入一幅图片，那么就只有图片是可用的，因为在写入图片的时候会删除该字符串。

这里给出了如何向剪贴板写入文本的用法：

```
clipboard = QApplication.clipboard()  
clipboard.setText("I've been clipped!")
```

这些文本会以普通文本的形式写入；很快就可以看到是如何处理 HTML 的了。

```
clipboard = QApplication.clipboard()  
clipboard.setPixmap(QPixmap(os.path.join(  
    os.path.dirname(__file__), "images/gvim.png")))
```

对于 QImage 型或者像素型图片数据，可以分别使用 setImage() 和 setPixmap() 写入剪贴板，就像这里所示的那样。无论是 QImage 还是 QPixmap 都可以用来处理一系列标准图片格式。

从剪贴板提取数据也相当简单：

```
clipboard = QApplication.clipboard()  
self.textLabel.setText(clipboard.text())
```

如果剪贴板中没有文本——例如，如果它有一幅图片，或者一些自定义数据类型——QClipboard.text() 将会返回一个空的字符串。

```
clipboard = QApplication.clipboard()  
self.imageLabel.setPixmap(clipboard.pixmap())
```

如果剪贴板没有图片——例如，如果它有的是文本，或者一些自定义数据类型——QClipboard.pixmap() 将会返回一幅空的图片。

除了可以处理普通文本和图片之外，还可以处理一些其他类型的数据。例如，这里给出的是如何把 HTML 文本复制到剪贴板中的做法：

<sup>①</sup> MIME 最早于 1992 年用在电子邮件系统中，但后来也应用到浏览器，用来告诉浏览器某种扩展名的文件应该用何种应用程序打开。因此，MIME 多用来指定一些客户端的自定义文件名及一些媒体文件的打开方式。例如，服务器会将它们发送的 MIME 型多媒体数据告诉浏览器，从而让浏览器知道接收到的信息是 MP3 文件还是 Shockwave 文件——译者注。

```

mimeData = QMimeData()
mimeData.setHtml(
    "<b>Bold and <font color=red>Red</font></b>")
clipboard = QApplication.clipboard()
clipboard.setMimeData(mimeData)

```

如果打算提取 HTML 文本，包括那些封装在 QMimeData 对象中的 HTML，都可以使用 QClipboard.text("html")。如果没有文本，或者其中的文本不是 HTML 格式（例如，仅是普通文本），这都将会返回一个空的字符串。这里给出一种从 QMimeData 对象封装数据中提取数据的一般方法：

```

clipboard = QApplication.clipboard()
mimeData = clipboard.mimeData()
if mimeData.hasHtml():
    self.textLabel.setText(mimeData.html())

```

在某些情况下，可能需要从或者向剪贴板写入或者读取那些自定义的数据。要做到这些就可以使用 QMimeData 类，就像在下一节中要看到的一样。

或者向剪贴板写入或者提取数据通常会对操作系统的全局剪贴板进行操作。此外，通过制定剪贴板模式，使用选择剪贴板（selection clipboard，一种存在于 Linux 或者其他基于 X 窗口系统上的附加剪贴板）也是有可能的，或者也可以在 Mac OS X 上找到粘贴板（pasteboard）。

## 10.4 拖放

许多 PyQt 窗口部件都支持对其他区域的拖放操作，而要做的只是打开支持模式并使其工作即可。例如，在图 10.2 所给出的应用程序中，就可以对左侧的 QListWidget 中的元素进行操作，而对中间的 QListWidget 或者右侧的 QTableWidget 则不会有什么影响。这个截图是应用程序已经拖放过几个元素之后的效果。

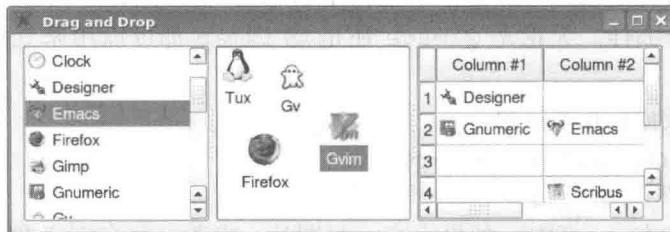


图 10.2 PyQt 的内置拖放功能

该应用程序的源代码在文件 chap10/draganddrop.pyw 中。

只需要通过对所用到的窗口部件进行设置就可以实现拖放功能。这里的代码就是用来创建左侧列表窗口部件的内容：

```

listWidget = QListWidget()
listWidget.setAcceptDrops(True)
listWidget.setDragEnabled(True)

```

中间列表窗口部件的实现与之类似，只是将其设置成了图标显示模式而不是列表显示模式：

```

iconListWidget = QListWidget()
iconListWidget.setAcceptDrops(True)
iconListWidget.setDragEnabled(True)

```

```
iconListWidget.setViewMode(QListWidget.IconMode)
```

让 QTableWidgetItem 获得支持拖放操作的方法也完全一样，只需要各调用一次 setAcceptDrops (True) 和一次 setDragEnabled (True) 即可。

再不需要其他代码了；这里给出的代码已经足可以让用户从一个列表窗口部件中拖拽图标和文本元素到另一个列表窗口部件中，也可以让用户在表格中的各个单元格之间相互拖放。

这样的内置拖放功能非常方便，通常也足够实用。不过，如果需要使其能够处理自定义数据，就必须重新实现一些事件处理程序，就像后面这一小节所讲的那样。

## 处理自定义数据

在图 10.3 中给出的应用程序可以支持自定义数据的拖放操作；特别是，对图标和文本的拖放操作（该应用程序的源代码放在文件 chap10/customdraganddrop.pyw 中）。尽管这一功能与提供的内置拖放功能一样，但该技术却更通用些，并且可以将该技术应用于我们想要应用的任何数据类型上。

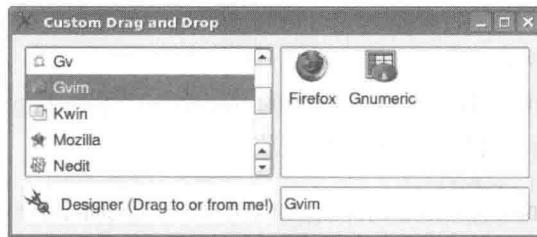


图 10.3 拖放自定义数据

可以把左侧列表窗口部件中的图标和文本拖放到右侧的列表窗口部件（为图标模式）中，或者将它们拖放到底部左侧的自定义窗口部件中，又或者将它们拖放到底部右侧的自定义行编辑窗口部件中——尽管对于最后一种情况来说只用到了文本。

对于放到剪贴板上或者有 PyQt 的拖放系统所用到的那些自定义数据，可以使用带自定义 MIME 数据类型的 QMimeData 对象。MIME 是一种用于处理具有多个组成部分的自定义数据的标准化格式。MIME 数据由一个数据类型和一个子类型构成——例如，text/plain、text/html 或者 image/png。要处理自定义 MIME 数据，就必须要选用一种自定义数据类型和一种子类型，然后将数据封装到 QMimeData 对象中。

在这个例子中，我们创建了一个 application/x-icon-and-text 型的新 MIME 数据。实际应用中，在子类型的开头部分使用 x- 是一种不错的习惯。这里已将数据保存在了 QByteArray 中，这是一个可变大小的字节数组，同时，在这个例子中，会用来保存一个 QString 和一个 QIcon，尽管它本来是可以保存任意数据的。

先从如何创建一个 QLineEdit 子类开始，这个子类可以接受 MIME 类型 application/x-icon-and-text 数据的拖放操作，它会用到文本并忽略图标。

```
class DropLineEdit(QLineEdit):
    def __init__(self, parent=None):
        super(DropLineEdit, self).__init__(parent)
        self.setAcceptDrops(True)
```

初始化程序会把行编辑窗口部件设置成只接受放下。

```

def dragEnterEvent(self, event):
    if event.mimeData().hasFormat("application/x-icon-and-text"):
        event.accept()
    else:
        event.ignore()

```

如果用户在行编辑窗口部件上方拖动的数据类型是 MIME 数据且可以进行处理，就希望能够显示一个图标；否则，行编辑窗口部件就会显示一个“不可拖放”的图标(通常会显示成 $\ominus$ 的形式)。通过对拖动输入事件的接受，就意味着可以接受所提供的 MIME 数据的放下操作；而通过忽略拖动输入事件，就意味着无法接受该种数据类型。在接下来的部分将会看到，一开始拖动操作，就会设置可接受数据的图标。

与拖拽相关的事件处理程序会自动在必要的时候被 PyQt 调用，因为在初始化程序中曾设置了可以接受放下操作。

```

def dragMoveEvent(self, event):
    if event.mimeData().hasFormat("application/x-icon-and-text"):
        event.setDropAction(Qt.CopyAction)
        event.accept()
    else:
        event.ignore()

```

当用户在窗口部件上拖动时，就会产生一系列的 dragMoveEvent()；这里希望能够将数据进行复制(而不仅仅是移动)，所以需要相应设置放下动作。

```

def dropEvent(self, event):
    if event.mimeData().hasFormat("application/x-icon-and-text"):
        data = event.mimeData().data("application/x-icon-and-text")
        stream = QDataStream(data, QIODevice.ReadOnly)
        text = QString()
        stream >> text
        self.setText(text)
        event.setDropAction(Qt.CopyAction)
        event.accept()
    else:
        event.ignore()

```

如果用户在窗口部件上放下数据就必须对其进行处理。通过提取数据(类型为 QByteArray)，然后再创建一个 QDataStream 来读取数据，可以实现这一点。QDataStream 类可以向或者从包括文件、网络 socket、外部进程以及字节数组在内的 QIODevice 读取和写入数据。这是因为，我们只对从字符串数组中提取的字符串感兴趣。值得注意的是，要想能够实现 QIcon 数据流或者从 QDataStream 数据流读取数据，就必须使用 PyQt 4.1 或者以后的版本。

由于 QLineEdit 只支持放下，因而会在下一个例子中，将创建一个 QListWidget 的子类，使其既可以支持拖动又可以支持放下。

```

class DnDListWidget(QListWidget):
    def __init__(self, parent=None):
        super(DnDListWidget, self).__init__(parent)
        self.setAcceptDrops(True)
        self.setDragEnabled(True)

```

这个初始化程序与之前用到的都很相似，只是这里既启用了拖动又启用了放下。

```

def dragMoveEvent(self, event):
    if event.mimeData().hasFormat("application/x-icon-and-text"):
        event.setDropAction(Qt.MoveAction)

```

```

        event.accept()
    else:
        event.ignore()

```

这与 DropLineEdit 的 dragMoveEvent() 的内容几乎完全一样；不同之处在于，这里是将放下动作设置到 Qt.MoveAction 上而不是 Qt.CopyAction 上。这里并未给出 dragEnterEvent() 的代码：因为它与 DropLineEdit 的代码完全一样。

```

def dropEvent(self, event):
    if event.mimeData().hasFormat("application/x-icon-and-text"):
        data = event.mimeData().data("application/x-icon-and-text")
        stream = QDataStream(data, QIODevice.ReadOnly)
        text = QString()
        icon = QIcon()
        stream >> text >> icon
        item = QListWidgetItem(text, self)
        item.setIcon(icon)
        event.setDropAction(Qt.MoveAction)
        event.accept()
    else:
        event.ignore()

```

这段代码再次与 DropLineEdit 的代码一样，只不过现在希望对图标的处理能够与对文本的处理方法一样。为了能够向 QListWidget 添加元素，就必须创建一个新的 QListWidgetItem 并把要传递的列表窗口部件 (self) 作为该元素的父对象。

```

def startDrag(self, dropActions):
    item = self.currentItem()
    icon = item.icon()
    data = QByteArray()
    stream = QDataStream(data, QIODevice.WriteOnly)
    stream << item.text() << icon
    mimeData = QMimeData()
    mimeData.setData("application/x-icon-and-text", data)
    drag = QDrag(self)
    drag.setMimeData(mimeData)
    pixmap = icon.pixmap(24, 24)
    drag.setHotSpot(QPoint(12, 12))
    drag.setPixmap(pixmap)
    if drag.start(Qt.MoveAction) == Qt.MoveAction:
        self.takeItem(self.row(item))

```

这是唯一一个没在 DropLineEdit 中的方法，它也正是那个使得从 DnDListView 中进行拖动成为可能的方法。不必检查 currentItem() 的返回值，因为只有元素才可以进行拖动，因而可以知道，如果 startDrag() 得到调用，那么就会有一个元素要被拖动了。startDrag() 方法会被 PyQt 在必要时自动调用，因为在初始化程序中就已经将其启用了。

这里创建了一个新的空字节数组，并使用 QDataStream 将其与 QListWidgetItem 的图标和文本进行组配。当只是使用 setVersion() 来处理那些仅在应用程序运行期间存在的内存数据时，这些数据不会与其他的应用程序进行数据交换，也就没有必要对 QDataStream 调用 setVersion()。一旦创建了字节数组，就可以将其封装到 QMimeData 对象中。然后，可以创建一个 QDrag 对象，并为其给定 MIME 数据。当时在这里曾经选择使用数据的图标作为拖拽操作的图标：如果当时没有这么做，PyQt 将会提供一个默认的图标。也曾经设置过拖拽的“热点” (hotspot) 作为图标的中心。鼠标的热点将总是会与图标的热点一致。

对 `QDrag.start()` 的调用会引发一个拖拽操作；可以把一个或者多个打算接受的动作作为参数给它。如果拖拽成功，也就是说，如果成功放下了数据，`start()` 方法就会返回发生了的动作——例如，复制动作或者移动动作。如果动作是移动的，就从这里的列表窗口部件中移除拖拽的 `QListWidgetItem`。从 Qt 4.3 开始，就应当使用 `QDrag.exec_()` 而不是使用 `QDrag.start()`。

`setAcceptDrops()` 方法从 `QWidget` 继承而来，不过 `setDragEnabled()` 则不是，因此在默认情况下，只有在部分窗口部件中它才是可用的。如果打算创建一个可以支持放下操作的自定义窗口部件，只需要简单调用 `setAcceptDrops(True)` 并重新实现 `dragEnterEvent()`、`dragMoveEvent()` 以及 `dropEvent()` 即可，正如在之前的例子中所做的那样。如果也希望这个自定义窗口部件能够支持拖动，而该窗口部件是从 `QWidget` 继承而来的，或者是从一些没有 `setDragEnabled()` 的 `QWidget` 子类继承而来的，就必须做两件事来让该窗口部件支持拖拽操作。一件事是提供一个 `startDrag()` 方法，以便可以创建一个 `QDrag` 对象，另一件事是要确保在适当的时间能够调用 `startDrag()` 方法。要确保 `startDrag()` 调用的最简单方法就是对 `mouseMoveEvent()` 的重新实现：

```
def mouseMoveEvent(self, event):
    self.startDrag()
    QWidget.mouseMoveEvent(self, event)
```

在本应用程序例子中底部左侧的窗口部件就是 `QWidget` 的一个直接子类，并且使用的就是这种技术。它的 `startDrag()` 方法与刚才看到的非常相似，只不过是会更简单些，因为它启用的是复制拖动操作而不是移动拖动操作，因此，不管是否成功放下了拖动，都不需要再做任何事情。

## 小结

在使用已有的窗口部件时，PyQt 的信号和槽机制往往就已经可以提供各种所需的行为。不过，当要创建一些自定义窗口部件时——例如，为了能够更好地控制一个窗口部件的外观和行为——就必须重新实现一些底层的事件处理程序。

对于外观来说，通常通过重新实现 `paintEvent()` 就足够了，尽管某些时候，或许还需要重新实现 `resizeEvent()`。对于这些事件通常并不需要调用基类实现。对于行为来说，通常可以通过重新实现 `keyPressEvent()` 以及一些诸如 `mousePressEvent()` 或者 `mouseMoveEvent()` 这样的鼠标事件来予以实现。通常不会调用基类中有关鼠标事件的实现，尽管这样做通常并无大害。如果通过重新实现 `QWidget.event()` 来练习底层控制，就必须给我们自己处理的那些事件返回 `True`，也必须为没有处理的那些事件在调用基类实现后的返回结果。

通常不必自行编写处理剪贴板的代码，因为大部分 PyQt 的文本编辑窗口部件会自动以我们所期待的方式与剪贴板进行交互。不过，如果确实希望在程序中能够与剪贴板一起工作，文本以及图片数据的写入和获取操作都很简单，只需要使用 `QApplication.clipboard()` 返回的 `QClipboard` 对象即可。对 HTML 数据的写入稍稍有些麻烦，因为必须把 HTML 封装到 `QMimeData` 对象中，尽管对 HTML 数据的提取会变得简单得多。在基于剪贴板来使用 MIME 数据时，并不会局限于 HTML；通过使用与处理拖放数据时一样的技术，可以对任意类型的数据进行保存和提取。

由标准 PyQt 窗口部件所提供的内置拖放操作支持功能非常易于搭建，也非常易于使用。尽管在某些情况下，需要拖放一些自定义数据类型。完成这一功能的代码写起来并不困难，而使用 `QByteArray` 可以确保能够对任意数量、任何 C++ 或者 PyQt 数据类型的数据进行拖放操作。然而，如果数据量非常巨大，或许会在传递代表数据的标记（比如说，在一个数据结构中的索引位置）而不是数据本身时，就只会在必要的时候才真正复制数据，这样可以实现更快的速度和更少的内存需求。

完全不借助 PyQt 的拖放功能也是可能的，通过重新实现一些鼠标事件处理程序，就可以实现一个我们自己的拖放系统。听起来这好像并不困难，不过，相较于使用 PyQt 已经提供的功能来说，它确实还是不大方便的。

PyQt 的事件处理系统非常强大，不过相当易于使用。在大多数情况下，使用更高级的信号和槽机制要容易得多，也要合适得多。不过，当需要更为精细地控制和自定义设置时，重新实现事件处理程序将可以得到所期望的精确外观和行为——在下一章实现一些自定义窗口部件时，将会看到这一方法的实际做法。

## 练习题

修改 `DnDListWidget` 类，以便能够在用户放下时可以在鼠标位置处为其提供一个具有两个选项的弹出菜单，分别是 `Copy` 和 `Move`。修改 `dragMoveEvent()`，使其具有一个移动的放下动作而不是复制动作。该菜单将需要放在 `dropEvent()` 中，并且要在创建新的列表元素之前。

`QMenu.exec_()` 方法会带一个 `QPoint` 参数，以用来说明该在哪里弹出菜单；`QCursor.pos()` 方法会提供当前鼠标的位置。根据用户的选择，必须把放下事件的放下动作设置成复制或者移动。

`startDrag()` 方法将需要做些轻微调整：`start()` 调用必须把移动和复制动作作为可接受的动作，并且只有在用户选择移动时才要移除该项。

决定该如何响应菜单动作的地方是最为巧妙的部分。例如，本来是可以使用 `functools.partial()` 或者 `lambda` 函数的。在参考答案中，只用到了一个保存放下动作的实例变量和两个方法，一个是把放下动作设置成移动，另一个是设置成复制，然后简单地将菜单动作与这些方法进行了连接。

也可以有一种更为巧妙的方法。不是使用弹出菜单，在拖拽移动事件中可以检查键盘的修饰符，也可以把放下动作设置成默认情况下的移动，或者如果在 `Ctrl` 键按下时可以将其设置成复制。相应地，可以根据 `Ctrl` 键的状态，在放下事件中设置放下的动作。这比弹出菜单要少些干扰，不过对于残障或者初级用户来说，这样做会显得少些直观性。

在答案中，创建了两个 `QListWidget` 的子类，一个是 `DnDMenuListWidget`，一个是 `DndCtrlListWidget`，以分别说明这两种方法。要实现这里所述的两个方法，只需要添加或者修改大约 25 行的代码（一旦复制/粘贴了 `DndListWidget` 并对这两个版本都进行了重命名）。

本练习题的参考答案放在了文件 `chap10/customdraganddrop_ans.pyw` 中。

# 第 11 章 自定义窗口部件

- 使用窗口部件样式表
- 创建复合窗口部件
- 子类化内置窗口部件
- 子类化 QWidget

PyQt 最强大也是由来已久的优势之一就是可以较方便地创建各类自定义窗口部件。用 PyQt 创建自定义窗口部件与内置标准窗口部件的创建方式一致，因此两者可以无缝集成，并在外观和行为上没有其他任何限制。在 PyQt 中创建自定义窗口部件的方式并不是“一劳永逸”的。相反，还可以从许多种方法中选出最能精细控制窗口部件行为和外观的方法来。

最最基本的自定义方法就是简单设置已有窗口部件的一些属性。这在前几章已经实现过多次。例如，在上一章，通过简单调用窗口部件的 `setAcceptDrops(True)` 和 `setDragEnabled(True)`，就启用了 PyQt 的默认拖拽行为。对于微调框(spinbox)来说，还可以限制它们的行为——比如通过调用 `setRange()`，就可以限制它们的最大值和最小值——而通过使用 `setPrefix()` 和 `setSuffix()`，则还可以改变微调框的外观。由于之前已经多次看到实际应用这一方法的例子，故而在本章将不会再做过多介绍。

如果设置现有窗口部件的各个属性仍无法满足要求，就可以借助样式表(style sheet)来自定义窗口部件的外观和行为的部分行为。可以对窗口部件应用样式的这种功能，是从 Qt 4.2 开始引入的，在这一章，将通过一个简单样例来初步说明这一功能。

有时，并不需要对某一特定窗口部件做那么多的自定义设置，要做的可能只是创建一个复合窗口部件，而这个复合窗口部件可能是由两个或者多个窗口部件合成一个而已。下面将用一个简单的例子来介绍这一方法的实现。

如果需要改变现有窗口部件的行为，而这个行为是无法通过属性设置来达到的，就可以对该窗口部件进行子类化并重新实现相关的事件处理程序来实现这一目的。

但是在某些情况下，如果需要的是一个不同于任何一个标准内置窗口部件的窗口部件，就需要通过直接子类化 `QWidget` 来实现自定义窗口部件的行为和外观了。下面的两个例子将对此进行介绍：第一个例子是一个“常规”窗口部件，它可能会应用在很多的地方和很多的应用程序中，第二个例子是一个与应用程序相关的窗口部件，它可能仅是针对某个程序而已。

## 11.1 使用窗口部件样式表

前面已经看到过很多通过改变窗口部件的属性来实现窗口部件自定义的例子。这些例子中的有些会影响窗口部件的行为，比如设置 `QSpinBox` 的有效范围，有一些会改变窗口部件的外观，比如设置 `QLabel` 的框架。Qt 4.2 引入了一个新的窗口部件属性，称为样式表(style sheet)。在这个属性中会保存一个 `QString` 并使用从 HTML 中借过来的 CSS(Cascading Style Sheets，级联样式表)语法<sup>①</sup>。

<sup>①</sup> 在 Mac OS X 上，官方尚未支持样式表，因此它们的行为可能尚无法预测。样式表有望在 Qt 4.4 及其后续版本中得到支持。

在图 11.1 中给出了一个使用了样式表的对话框。样式表会应用到所设置的窗口部件以及这些窗口部件的全部子窗口部件上。在这个例子中，会将组合框（combobox）的文字设置成深蓝色，将行编辑框（line edit）的文字设置成深绿色。同时，还将那些“必填”的全部行编辑框的背景都设置成了黄色。

尽管没有哪个 PyQt 窗口部件会有“必填”属性，但是自 Qt 4.2 以来，通过向 `QObject` 增加动态属性已成为可能。值得注意的是，Qt 的属性与 Python 的属性是不相同的——比如，它们分别可以使用 `property()` 和 `setProperty()` 进行访问。从 PyQt 4.2 开始，可以用 `QtCore.pyqtProperty()` 函数一次创建出既能作为 Python 又能作为 Qt 的属性。

```
self.lineedits = (self.forenameEdit, self.surnameEdit,
                  self.companyEdit, self.phoneEdit,
                  self.emailEdit)
for lineEdit in self.lineedits:
    lineEdit.setProperty("mandatory", QVariant(True))
self.connect(lineEdit, SIGNAL("textEdited(QString)"),
            self.updateUi)
```

以上代码来自表单的初始化程序。它给那些行编辑框都增加了“必填”属性，这样用户就不能再让这些行编辑框是空的了。所有的 Qt 属性都会保存成 `QVariants`。有关信号-槽连接的问题很快将在下面讨论到。

这里为窗口部件创建了一个样式表，并作为类的一个静态变量，然后在构造函数的最后将其应用于表单上。本来这个样式表是可以简单从文件读取的（因为样式表就是些简单的文本），又或者也可从 PyQt 资源中读取。

```
StyleSheet = """
QComboBox { color: darkblue; }
QLineEdit { color: darkgreen; }
QLineEdit[mandatory=true] {
    background-color: rgb(255, 255, 127);
    color: darkblue;
}
...
self.setStyleSheet(ContactDlg.StyleSheet)
```

样式表语法基本上由一些“选择器”和一些“属性名:值”对构成的。在上述代码中，第一行代码包含了一个 `QComboBox` 选择器，意味着这个选择器的属性值将会应用于任何 `QComboBox` 或者该窗口部件中设置了此样式表的所有 `QComboBox` 子类。在这个例子中，结果就是会把字体的颜色设置为深蓝色。第二个选择器是 `QLineEdit`，作用方式与第一个选择器类似。

第三个选择器的特别之处在于：它既给定了一个类，又给定了一个与这个类的状态必须匹配的属性。换句话说，这第三个选择器仅会应用到那些包含了“必填”属性并且属性值为 `True` 的 `QLineEdit` 及 `QLineEdit` 的子类上。在这个例子中，背景颜色设置为了黄色（由一个 RGB 三元组给定），字体颜色设置为了深蓝色。



图 11.1 用样式表自定义对话框

经过以上设置，对话框要比其最初出现时稍有不同。这是因为，只有在类别组合框设置成“Business”时，“company”行编辑框才会被设置成必填。为了达到这种效果，除了需要使用到之前看到的信号-槽连接之外，还需要有另一个连接：

```
self.connect(self.categoryComboBox, SIGNAL("activated(int)"),
             self.updateUi)
```

所有的连接都会连接到 `updateUi()` 方法上：

```
def updateUi(self):
    mandatory = self.companyEdit.property("mandatory").toBool()
    if self.categoryComboBox.currentText() == "Business":
        if not mandatory:
            self.companyEdit.setProperty("mandatory",
                                         QVariant(True))
    elif mandatory:
        self.companyEdit.setProperty("mandatory", QVariant(False))
    if mandatory != \
        self.companyEdit.property("mandatory").toBool():
        self.setStyleSheet(ContactDlg.StyleSheet)
    enable = True
    for lineEdit in self.lineEdits:
        if lineEdit.property("mandatory").toBool() and \
            lineEdit.text().isEmpty():
            enable = False
            break
    self.buttonBox.button(QDialogButtonBox.Ok).setEnabled(enable)
```

如果用户改变了类别，就必须通过重新应用样式表来强制窗口部件能够重新样式化。这样可以保证“company”行编辑框的背景能够得到正确设置，即可以根据是否为必填而将背景显示为白色或者黄色。遗憾的是，在运行较慢的机器上，当重新设置样式表时会有轻微的闪烁——考虑到这一原因，为保证样式表只在必要时才会再次应用，这里给出的代码看起来会显得稍微有点拖沓。对诸如启用、选中和悬浮这样的“伪”状态改变，并没有必要对样式表进行重新设置。

样式表的语法要比上面看到的这个简单例子丰富得多，功能也要强大得多。例如，如果在选择器的前面加上一个句点，比如`.QLineEdit`，则该选择器就会只应用于指定的类，而不会应用于这个类的子类。如果要求选择器仅应用于某一特定窗口部件，则可以对该窗口部件调用`setObjectName()`，然后用该名字作为选择器的一部分。譬如，如果有一个按钮，其对象名字是“`findButton`”，则应用于这个按钮上的选择器就应该是`QPushButton#findButton`。

有些窗口部件会有一些“子控件”。例如，`QComboBox`会有一个箭头子控件，用户通过点击这个箭头来看到下拉列表。子控件可以指定为选择器的一部分——例如，`QComboBox::drop-down`。伪状态可以用一个冒号指定——例如，`QCheckBox:checked`。

除了设置颜色外，样式表还可以用来设置字体、边框、边白、填充和背景。用来测试样式表的一个快速而又简单的方法是运行 Qt 设计师 (Qt Designer)，创建一个新表单，并将一些窗口部件拖放到该表单中，然后试着为该表单输入和编辑一个相应的样式表。

样式表可以用于对表单中的某个特定窗口部件进行设置，或者对表单 (`QDialog` 或者  `QMainWindow`) 自身进行设置。无论哪种情况，样式表都将会自动应用于所有的子窗口部件上。也可以(或者相当普遍)为整个应用程序单独设置一个样式表，在这种情况下，就需要把样式表设置应用到 `QApplication` 对象上。

## 11.2 创建复合窗口部件

复合窗口部件 (composite widget) 就是由两个或者多个其他窗口部件组合到一起的窗口部件。先前已经感受过复合窗口部件创建程序，例如，过去创建的每一个对话框都是一个复合窗口部件。而之所以会将一些书本空间浪费到一个之前已经涉及过的主题，这是因为，与前期创建的对话框（是 `QDialog` 的子类）有所不同，这一次要创建的这些复合窗口部件会都不是对话框，因而也就可以应用到一个对话框的内部（或者作为主窗口的中心窗口部件）。

这种类型的复合窗口部件与对话框非常相似：创建一些子窗口部件，对它们进行布局，创建一些需要的信号-槽连接。最主要的不同在于，创建的复合窗口部件继承自 `QWidget` 而不是 `QDialog`。

在图 11.2 中的截图展示了一个看似传统的对话框，但事实上，其中只明确创建了 6 个窗口部件而不是 12 个。这是因为，其中使用了 4 个自定义 `LabelledLineEdit`，一个自定义的 `LabelledTextEdit`，还有一个 `QDialogButtonBox`。

这个标签编辑器有两个比较特殊的地方。第一，它们会自动设置伙伴 (buddy) 关系，第二，它们可以将标签布局到待编辑窗口部件的左边或者上方。

```
self.zipcode = LabelledLineEdit("&Zipcode:")
self.notes = LabelledTextEdit("&Notes:", ABOVE)
```

就像这段来自表单构造函数中的代码所展示的那样，创建标签编辑器非常简单。`LabelledLineEdit` 会将它的窗口部件当成实例变量，因此可以通过 `LabelledLineEdit.label` 的形式访问它的 `QLabel`，也可以通过 `LabelledLineEdit.lineEdit` 访问它的 `QLineEdit`。`LabelledTextEdit` 也有同样的 `label` 变量，还有一个 `TextEdit` 变量可以用来访问它的 `QTextEdit`。`ABOVE` 只是一个模块常量；还有一个与之对应的 `LEFT` 也是。

```
class LabelledLineEdit(QWidget):
    def __init__(self, labelText=QString(), position=LEFT,
                 parent=None):
        super(LabelledLineEdit, self).__init__(parent)
        self.label = QLabel(labelText)
        self.lineEdit = QLineEdit()
        self.label.setBuddy(self.lineEdit)
        layout = QVBoxLayout(QVBoxLayout.LeftToRight \
            if position == LEFT else QVBoxLayout.TopToBottom)
        layout.addWidget(self.label)
        layout.addWidget(self.lineEdit)
        self.setLayout(layout)
```

以上代码就是完整的 `LabelledLineEdit` 类。如果需要把信号和槽连接到它的标签或者行编辑框上，由于它们已经被声明为 `public` 实例变量，故而可以通过直接访问它们来实现。使用的并不是 `QVBoxLayout` 或者 `QHBoxLayout`，而是使用了它们的基类，因而可以在创建行编



图 11.2 使用了标签窗口部件的对话框

辑框标签的时候就设置其布局的方向。这里将不会给出 `LabelledTextEdit` 的代码，因为它的不同之处仅仅在于创建的是 `QTextEdit` 而不是 `QLineEdit`，以及将其命名为 `TextEdit` 而不是 `lineEdit`。

虽然这里完全是使用代码的方式创建了这个复合标签编辑窗口部件，但使用 Qt 设计师创建这个复合窗口部件也是可行的，只需以“Widget”模板为基础即可。

在大的工程项目中创建可重复使用的复合窗口部件能够节约大量时间。另外，在需要创建一个主窗口风格的应用程序时，而这个应用程序的中心窗口部件至少包含两个或者多个窗口部件且 MDI 工作空间、窗口切分条或者停靠窗口都不是合适的解决方案，创建复合窗口部件这样做也会很有用。

### 11.3 子类化内置窗口部件

有时需要创建一个与现有窗口部件的外观和行为相似的窗口部件，不过，有许多的自定义需求是无法通过使用样式表或者设置窗口部件其他属性实现的。在这些情况下，就可以对类似的窗口部件进行子类化来满足我们的需要<sup>①</sup>。

为了说明如何对一个已存在的窗口部件进行子类化来做个假设，假定需要有一个罗马数字型而不是数字型的微调框，如图 11.3 所示。如果如何将整数转换为罗马数字或者做逆向转换的方法是已知的，那么，为实现这一目的，最直接的方法就是子类化 `QSpinBox`。

在对一个微调框进行子类化时，需要重新实现三个方法：

`validate()`，用于防止在微调框中输入无效的数据；`valueFromText()`，用于将用户输入的文本转换为整数；`textFromValue()`，用于将数字转换为其对应的文字表示。除此之外，还需要在构造函数中做些设置，以便可以知道是用哪一种方法开始的。

```
class RomanSpinBox(QSpinBox):
    def __init__(self, parent=None):
        super(RomanSpinBox, self).__init__(parent)
        regex = QRegExp(r"^\M{0,3}(CM|CD|D?C{0,3})\M{0,3}(IX|IV|V?I{1,3})$")
        regex.setCaseSensitivity(Qt.CaseInsensitive)
        self.validator = QRegExpValidator(regex, self)
        self.setRange(1, 3999)
        self.connect(self.lineEdit(), SIGNAL("textEdited(QString)"),
                     self.fixCase)
```



图 11.3 罗马数字的微调框

PyQt 提供有自己的正则表达式类，其语法与 Python 的 `re` 模块非常相似。主要的不同之处在于，`QRegExp` 虽然允许整个正则表达式非贪婪，但它并不支持非贪婪量词 (nongreedy quantifier)<sup>②</sup>。这个正则表达式列出了构成 1 ~ 3999 罗马字母的合法组合<sup>③</sup>。

① 附录 B 提供了部分 PyQt 窗口部件的截图和简要介绍，附录 C 给出了部分 PyQt 类的继承关系。

② 正则表达式的贪婪匹配是指，在整个表达式匹配成功的前提下，还继续尽可能多地匹配更多的内容；非贪婪匹配模式是指，在整个表达式匹配成功后就尽可能减少匹配的内容。匹配量词包括“?”、“\*”和“+”等，贪婪与非贪婪模式可以影响被量词修饰的子表达式的匹配行为。例如，字符串 `str = "abcaxc"`，匹配表达式模式 `p = "ab* c"`，那么贪婪匹配的结果为 `abcaxc`，而非贪婪匹配的结果为 `abc`——译者注。

③ 这个正则表达式节选自 Mark Pilgrim 的 *Dive into Python* 一书（参见 <http://www.diveintopython3.net/> 或者 <http://www.diveintopython.net/>）。

由于并不介意用户输入的是大写还是小写字母，因此会让正则表达式不区分大小写——但无论如何，这里还是会将输入字母全部转变为大写字母。验证器 (validator) 可以使用 PyQt 的预定义验证器（可用于整数和浮点数），也可以像上述一样使用基于正则表达式的验证器。要设置一个信号-槽连接，用于确保用户无论何时输入任何文本，都能够将其强制转换为大写。

```
def fixCase(self, text):
    self.lineEdit().setText(text.upper())
```

QSpinBox 有一个 QLineEdit 组件，它还提供了一个存取器方法来获得其中的值。可以使用这一方法来将用户的输入变为大写（用户不能输入诸如“A”、“B”、“1”或者“2”这样的非法字母，因为验证器将不会接受它们）。

```
def validate(self, text, pos):
    return self.validator.validate(text, pos)
```

提供 validate() 方法是必需的。这个方法会在用户修改文本时被自动调用，因为这一行为是 QSpinBox API 中的一部分。只需简单地将工作传递给在构造函数内创建的验证器对象即可。

```
def valueFromText(self, text):
    return intFromRoman(unicode(text))
```

如果用户输入文本，微调框就需要知道这些文本所代表的整数值。只需简单地将这些文本传给 intFromRoman() 方法即可，其中，这个方法节选自 *Python Cookbook* 一书中的“Roman Numerals”一节。

```
def textFromValue(self, value):
    return romanFromInt(value)
```

微调框必须能够将整数转换成它们的文字表示——例如，当调用 setValue() 时，或者是在用户使用微调框的按钮增加或者减小值时。这一次还是会将工作传递给 romanFromInt() 函数，这个函数节选自 *Python Cookbook* 一书“Decimal to Roman Numerals”一节中的部分内容。

RomanSpinBox 可应用于任何传统 QSpinBox 使用到的地方，仅有的局限性在于它所能够处理的数值范围。对已有窗口部件使用子类化方法的一个更为复杂的例子可以参阅第 13 章，在那里，会使用 QTextEdit 来创建一个 RichTextLineEdit。

## 11.4 子类化 QWidget

当无法通过设置属性、使用样式表或者对已存在的窗口部件进行子类化而获得想要的自定义窗口部件时，可以从头开始创建这个所需的窗口部件。在实际应用中，一般会通过子类化 QWidget 来创建自定义窗口部件，因为它提供了很多无须太多关注的后台技术便利，能够让我们专注于所需的工作：外观和自定义窗口部件的行为。

在这一节，将会看到两个不同的自定义窗口部件。第一个，FractionSlider，是一个普通的“范围控制”型窗口部件，可能会被多次用到。第二个，YPipeWidget，是一个可能只会应用于某个特定程序中的应用程序相关型窗口部件。

在了解这两个窗口部件的细节之前，首先会讨论 PyQt 中的绘图机制，尤其是 QPainter 所使用的坐标系。QPainter 有两个独立的坐标系：一个是设备（物理的）坐标系，可以用来

匹配窗口部件区域中的像素；另一个是逻辑坐标系。默认情况下，逻辑坐标系会与物理坐标系完全对应。

事实上，物理坐标没必要像素化，因为物理坐标取决于底层的绘图设备。绘图设备可以是 QGLPixelBuffer（用于 2D 与 3D 绘制），也可以是 QImage、QPicture、QPixmap、QPrinter（这种情况下，坐标是一些， $\frac{1}{72}$ "）、QSvgGenerator（在 Qt 4.3 中引入），又或者是 QWidget。

在 PyQt 术语中，物理坐标系会被称为“视口”（viewport），而令人困惑的是，逻辑坐标系会被称为“窗口”（window）。

如图 11.4 所示，有一个物理窗口部件的大小为  $800 \times 600$ 。通过调用 setWindow(-60, -60, 120, 120)，可以创建一个左上角坐标为 (-60, -60)、宽为 120、高为 120 并且中心点位于 (0, 0) 的“窗口”。这个窗口的坐标系是一个逻辑坐标系， QPainter 会自动将此坐标系映射到底层的物理设备上。在调用 setWindow() 后，所有的绘制工作都会根据逻辑（窗口）坐标系发生。

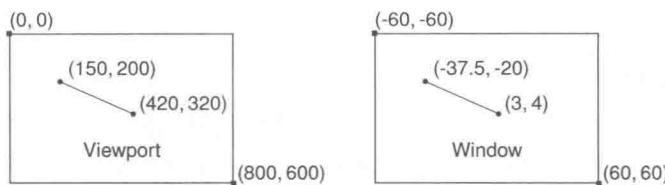


图 11.4 视口和窗口坐标系

在这个例子中，该窗口部件是矩形的，不过窗口也会有相同的宽度和高度。这就意味着，要绘制的这些元素都会在水平方向进行拉伸，因为在 y 轴的坐标会被 QPainter 按照  $120:600$ （即  $1:5$ ）的比例缩放，而在 x 轴的坐标会按  $120:800$ （即  $1:6\frac{2}{3}$ ）的比例缩放。

对于大多数窗口部件而言，矩形区域时都可以较好地工作，但是在某些情况下——譬如，当确实需要一个正方形逻辑窗口时——就应该对视口进行修改，以便可以对窗口部件的区域进行等比例变化。

```
side = min(self.width(), self.height())
painter.setViewport((self.width() - side) / 2,
                    (self.height() - side) / 2, side, side)
```

在窗口部件的 paintEvent() 内执行的这些代码，将窗口部件的视口改变为当前情况下所适用的最大正方形区域。在之前的例子中，将生成一个没有顶部边白和底部边白的  $600 \times 600$  像素的视口，但是在视口的左右两边，会各有 100 像素的边白。窗口现在将变成完全的正方形，而在该窗口中所绘制的所有东西的长宽比都仍将得以保留。

使用窗口的最大好处在于，可使用逻辑坐标系进行绘制。这将非常方便，因为它意味着，所有必要的缩放（譬如，当用户调整窗口部件的大小时），将会由 PyQt 自动完成。这个好处也有一个缺点：如果需要绘制的是文本，那么文本也将随着其他元素而被缩放。因此，最简单的方法是，对于绘制文本的自定义窗口部件，使用物理（视口）坐标，而其他的窗口部件，则使用逻辑（窗口）坐标。在后续的两个例子中，这两种方式都会用到：FractionSlider 会使用视口坐标系，而 YPipeWidget 则会使用窗口坐标系。

在表 11.1 中给出了 QWidget 的部分方法。

表 11.1 QWidget 的部分方法

语法	说明
w.addAction(a)	把 QAction a 添加到 QWidget w 中; 这对于上下文菜单很有用
w.close()	隐藏 QWidget w; 但如果设置了 Qt.WA_DeleteOnClose, 会将其删除
w.hasFocus()	如果 QWidget w 获得了键盘光标焦点, 返回 True
w.height()	返回 QWidget w 的高度
w.hide()	隐藏 QWidget w
w.move(x, y)	把最顶层的 QWidget w 移动到位置 (x, y) 处
w.raise_()	把 QWidget w 提升到父窗口部件的栈的顶部
w.rect()	以 QRect 的形式返回 QWidget w 的外形尺寸
w.restoreGeometry(ba)	用 QByteArray ba 中的编码存储值恢复顶层 QWidget w 的几何尺寸
w.saveGeometry()	返回一个 QByteArray, 其中会保存 QWidget w 的几何尺寸
w.setAcceptDrops(b)	根据 bool b 的值, 设置 QWidget w 可否接受拖放操作
w.setAttribute(wa, b)	根据 bool b 的值, 设置 Qt.WidgetAttribute wa 为 on 或者 off。最常用到的属性是 Qt.WA_DeleteOnClose
w.setContextMenuPolicy(p)	把 QWidget w 的上下文菜单策略设置成策略 p。这些策略包括 Qt.NoContextMenu 和 Qt.ActionsContextMenu
w.setCursor(c)	把 QWidget w 的光标设置成 c, c 可以是 QCursor 或者 Qt.CursorShape
w.setEnabled(b)	根据 bool b 的值, 把 QWidget w 设置成启用或者禁用
w.setFocus()	把键盘光标焦点设置到 QWidget w
w.setFont(f)	把 QWidget w 的字体设置成 QFont f
w.setLayout(l)	把 QWidget w 的布局设置成 QLayout l
w.setSizePolicy(hp, vp)	把 QWidget w 的水平和竖直策略 QSizePolicy 设置成 hp 和 vp
w.setStyleSheet(s)	把 QWidget w 的样式表设置成字符串 s 中的 CSS 文本
w.setWindowIcon(i)	把顶层 QWidget w 的图标设置成 QIcon i
w.setWindowTitle(s)	把顶层 QWidget w 的标题设置成字符串 s
w.show()	以非模态形式显示顶层 QWidget w。也可以使用 setWindowModality() 来以模态的形式显示它
w.update()	为 QWidget w 安排一个绘制事件
w.updateGeometry()	对于非顶层的那些窗口部件, 通报它们含有的布局可能会受 QWidget w 几何尺寸改变的影响
w.width()	返回 QWidget w 的宽度

#### 11.4.1 例：分数滑块

FractionSlider 是一个窗口部件, 它允许用户选择 0~1 之间的一个分数, 如图 11.5 所示。这个 FractionSlider 将会允许程序开发人员将分母区间设置为 3~60, 并且会在用户一旦改变该分数时, 就会发送一个 valueChanged (int, int) 信号(带的是分子和分母)。同时, 还会提供鼠标和键盘控制功能, 也会绘制整个窗口部件。此外, 还要确保窗口部件的最小尺寸大小能够总是与分母的大小成正比, 从而可以避免产生窗口部件被调整的太小而使得难以显示清楚分数文本的问题。

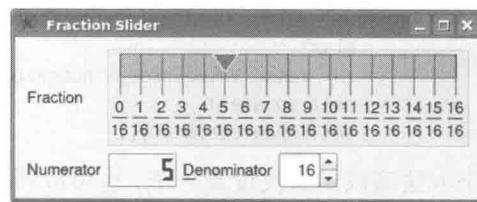


图 11.5 分数滑块对话框

下面先从静态数据和初始化程序看起：

```
class FractionSlider(QWidget):
    XMARGIN = 12.0
    YMARGIN = 5.0
    WSTRING = "999"

    def __init__(self, numerator=0, denominator=10, parent=None):
        super(FractionSlider, self).__init__(parent)
        self.__numerator = numerator
        self.__denominator = denominator
        self.setFocusPolicy(Qt.WheelFocus)
        self.setSizePolicy(QSizePolicy(QSizePolicy.MinimumExpanding,
                                      QSizePolicy.Fixed))
```

XMARGIN 和 YMARGIN 用来为窗口部件各边的周围留出一些水平方向和竖直方向的空白。WSTRING 是一个字符串，是可能需要用到的最长文本：2 位数字用于显示，剩余的数字用来提供一些边白。

程序会提供一个默认值，开始时窗口部件是关闭的，会显示成十分之零。焦点策略会选择 Qt.WheelFocus，因为这是“最强大的”一个策略，这就意味着，窗口部件将会在切换到、点击到或者是用户在其上面使用鼠标滚轮时，均可以获得焦点。要对水平与竖直方向的大小策略进行设置。这样可以确保窗口部件能与布局管理器正确协作。在这里，已经知道，在水平方向上，该窗口部件可以收缩至其最小尺寸大小但会可增，在竖直方向会有一个固定值，无论它的 sizeHint() 方法返回的高度是多少。

```
def decimal(self):
    return self.__numerator / float(self.__denominator)

def fraction(self):
    return self.__numerator, self.__denominator
```

对于返回值有两个简便方法，第一个方法可以返回一个浮点数，第二个方法则可以返回一对整数值。

```
def setFraction(self, numerator, denominator=None):
    if denominator is not None:
        if 3 <= denominator <= 60:
            self.__denominator = denominator
        else:
            raise ValueError, "denominator out of range"
    if 0 <= numerator <= self.__denominator:
        self.__numerator = numerator
    else:
        raise ValueError, "numerator out of range"
    self.update()
    self.updateGeometry()
```

这个方法可以用来仅设置分子，也可以同时用来设置分子和分母。一旦分数发生改变，就会调用 update() 来触发绘制事件，以便可以让标记当前分数的金色三角能够在正确的地方进行重新绘制。

还要调用 updateGeometry() 方法。这将通知所有该窗口部件相关的布局管理器，该窗口部件的形状可能已经发生了改变。这似乎很奇怪——毕竟，只是修改了分数。但是，如果修改的是分母，窗口部件的大小将会根据显示更多(或更少) 分数而改变。结果，如果有一个用

于该窗口部件的布局管理器，它就会重新计算布局，向该窗口部件询问其尺寸大小，并根据需要调整布局。

通过抛出异常可以用来处理非法的值。这是因为，`setFraction()`通常在编程中调用，所以应当不会在正常运行的过程中给出超出范围的值。另一种方法是，把分子和分母限制在合理的范围内：窗口部件的键盘和鼠标事件处理程序会使用到这一方法，可为窗口部件提供其行为，这正是现在需要的。

```
def mousePressEvent(self, event):
    if event.button() == Qt.LeftButton:
        self.moveSlider(event.x())
        event.accept()
    else:
        QWidget.mousePressEvent(self, event)
```

如果用户点击了窗口部件，希望能够将分子设置成最近的那个分数。本来是在鼠标按键事件内进行计算的，但是由于希望能够同时支持在点击金色三角时来拖动它，因此会把这一段代码单独剥离成一个`moveSlider()`方法，此方法会带有一个鼠标`x`坐标的参数。在修改了分数后，就可以接受此事件，因为已经完成了事件处理。如果没有处理点击操作（譬如，如果是右键单击），就可以调用基类的实现，虽然严格意义上这并不是必要的。

```
def moveSlider(self, x):
    span = self.width() - (FractionSlider.XMARGIN * 2)
    offset = span - x + FractionSlider.XMARGIN
    numerator = int(round(self._denominator * \
        (1.0 - (offset / span))))
    numerator = max(0, min(numerator, self._denominator))
    if numerator != self._numerator:
        self._numerator = numerator
        self.emit(SIGNAL("valueChanged(int,int)"),
                  self._numerator, self._denominator)
    self.update()
```

先从计算窗口部件的“长度范围”开始，不包括水平方向的边白。然后，计算出沿`x`轴方向鼠标点击（或拖动）的是多远，计算出分子作为窗口部件宽度的比例。如果用户的点击或者拖动是在左边的边白区域，则设置分子为0，如果点击或者拖动是在右侧的边白区域，则把分子设置成分母的值（因此分数将会是1）。如果分子之前已发生了改变，则相应地设置实例变量并发射一个信号，宣示值已改变。然后调用`update()`来触发一个绘制事件（移动金色三角）。

这样就会发射一个 Python 非短路信号；本来也是可以很容易地通过放置一个`(int, int)`而将其变为短路信号。也可以使用`_pyqtSignals_`定义信号，尽管仅仅是用在那些 PyQt 中编写的、可集成到 Qt 设计师中的自定义窗口部件时，这样做才确实会有用<sup>①</sup>。

```
def mouseMoveEvent(self, event):
    self.moveSlider(event.x())
```

以上这个简短的方法就是需要用来支持拖动金色三角改变分数值的全部内容。这个方法仅会在鼠标被拖动时调用，也就是，当鼠标左键按下并同时移动鼠标时调用该方法。这是`QWidget`的标准行为——如果愿意，可以通过调用`QWidget.setMouseTracking(True)`，可以生成所有鼠标移动而产生的鼠标移动事件，无论是否按下了鼠标按键。

<sup>①</sup> 可参阅 Python 文档的 `PyQt pyqt4ref.html`，位于“Writing Qt Designer Plugins”部分中。

```

def keyPressEvent(self, event):
    change = 0
    if event.key() == Qt.Key_Home:
        change = -self._denominator
    elif event.key() in (Qt.Key_Up, Qt.Key_Right):
        change = 1
    elif event.key() == Qt.Key_PageUp:
        change = (self._denominator // 10) + 1
    elif event.key() in (Qt.Key_Down, Qt.Key_Left):
        change = -1
    elif event.key() == Qt.Key_PageDown:
        change = -((self._denominator // 10) + 1)
    elif event.key() == Qt.Key_End:
        change = self._denominator
    if change:
        numerator = self._numerator
        numerator += change
        numerator = max(0, min(numerator, self._denominator))
        if numerator != self._numerator:
            self._numerator = numerator
            self.emit(SIGNAL("valueChanged(int,int)"),
                      self._numerator, self._denominator)
            self.update()
        event.accept()
    else:
        QWidget.keyPressEvent(self, event)

```

在键盘支持方面，希望 Home 键能够用来将分数设置成 0，End 键用来将分数设置成 1，向上或者向右箭头键能够将当前分数值向前移动到下一个分数值，而向下或者向左箭头键则能够将当前分数值向下移动到后一个分数。此外，还会把 PageUp 键设置成向前移动分母值的十分之一，而 PageDown 键则向下移动分母值的十分之一。

用来确保分子处于正确的范围内、用来设置实例变量等的代码与之前鼠标按下事件处理程序的代码完全相同。另外，还会将那些未做处理的按键事件传递给它的基类实现——基类不会做任何处理，就像基类的鼠标点击处理程序什么也不做一样。

```

def sizeHint(self):
    return self.minimumSizeHint()

```

将窗口部件的默认尺寸大小设置成它的最小尺寸大小。严格来说，没有必要再重新实现这个方法，但是这样做了会让意图更为明显些。由于在初始化程序中已经设置过尺寸策略，这个窗口部件是可以水平拉伸的，以便能够占据尽可能多的水平可用空间。

```

def minimumSizeHint(self):
    font = QFont(self.font())
    font.setPointSize(font.pointSize() - 1)
    fm = QFontMetricsF(font)
    return QSize(fm.width(FractionSlider.WSTRING) * \
                self._denominator,
                (fm.height() * 4) + FractionSlider.YMARGIN)

```

QFontMetricsF 会由 QFont 对象初始化，在这个例子中用做窗口部件的默认字体<sup>①</sup>。这个字体继承自窗口部件的父类，父类窗口部件又会再从自己的父类继承，以此类推，而顶级窗口部

<sup>①</sup> PyQt 还有一个 QFontMetrics 类，它可以给出整数值而不是浮点数值。相应地，PyQt 还有 QLine、QLineF、QPoint、QPointF、QPolygon、QPolygonF、QRect、QRectF 等其他的类。

件的字体(和配色方案以及其他用户设置)则会继承自 QApplication 对象, QApplication 对象则会从底层窗口系统中获取用户的偏好设置来给出这些属性值。显然,当然可以忽略用户的偏好设置,从而为任意的窗口部件明确设置其字体。

QFontMetricsF 对象会提供真实的度量值,也就是,当前真正用到的字体——这些字体可能会与预设的字体不同。举个例子,如果用到的是 Helvetica 字体,但这个字体一般在 Linux 或者 Mac OS X 系统上肯定能够找到,但在 Windows 系统上,可能找到的对应字体就是 Arial 字体了。代码中显示分数的字体大小要比用户给定的字体大小小一号,这就是为什么要调用 setPointSize() 方法的原因。

把确保能够显示全部分数的宽度设置成窗口部件的最小宽度,假设每一个分数的宽度都是三个数字宽,即两个数字加上每侧一点边白。整体的实际宽度要比此略小,因为这个值中还没有包括水平方向的边白。窗口部件的最小高度可以设置成一个字符高度的 4 倍,即要给分数“段”(表示每个分数的矩形)留够足够的竖直空间,这个“段”是指分数线、分子和分母。与宽度值一样,实际的高度比此值略小,因为其中考虑了竖直方向上的一半为边白。

键盘和鼠标事件的处理程序也已经实现了,设置过了尺寸大小策略,还实现了尺寸大小提示的方法,现在已经能够保证对用户的交互做出适当的行为,因而也就能够对布局该窗口部件的布局管理器进行相应的响应。目前只剩下了一件要做的事情:在需要绘制窗口部件时,必须能够对窗口部件进行绘制。paintEvent() 方法比较长,因此会将其分开一一细看。

```
def paintEvent(self, event=None):
    font = QFont(self.font())
    font.setPointSize(font.pointSize() - 1)
    fm = QFontMetricsF(font)
    fracWidth = fm.width(FractionSlider.WSTRING)
    indent = fm.boundingRect("9").width() / 2.0
    if not X11:
        fracWidth *= 1.5
    span = self.width() - (FractionSlider.XMARGIN * 2)
    value = self._numerator / float(self._denominator)
```

先从获取需要使用的字体和字体标准开始。然后,会对 fracWidth 进行计算,这是一个分数的宽度,也可以用做每一个分数分割线之间的对齐。if 语句用于补偿 X Window 系统与诸如 Windows 和 Mac OS X 之类的其他窗口系统之间的字体标准差异。span 变量是窗口部件不包括水平方向两个边白的宽度,而 value 变量是分数的浮点数值。

如果底层窗口系统是 X Window 系统,一般是指 Linux、BSD、Solaris 等类似的系统,x11 布尔变量的值会是 True,在其他系统下会是 False——譬如,在 Windows 和 Mac OS X 系统。会在文件的开头位置,在导入语句之后,使用下列语句设置 x11 布尔变量的值:

```
X11 = "qt_x11_wait_for_window_manager" in dir()
```

也可以像这样写得更清晰一些:

```
import PyQt4.QtGui
X11 = hasattr(PyQt4.QtGui, "qt_x11_wait_for_window_manager")
```

这样做是有效的,因为 PyQt4.QtGui.qt\_x11\_wait\_for\_window\_manager() 只存在于 X Window 系统上。在第 7 章,还用过一个检测 Mac OS X 系统的类似技术。

```
painter = QPainter(self)
painter.setRenderHint(QPainter.Antialiasing)
painter.setRenderHint(QPainter.TextAntialiasing)
```

```

painter.setPen(self.palette().color(QPalette.Mid))
painter.setBrush(self.palette().brush(QPalette.AlternateBase))
painter.drawRect(self.rect())

```

创建 QPainter 并把它的渲染提示(render hint)设置成抗锯齿绘制模式。然后，设置画笔(用于形状轮廓和文本绘制)和画刷(用于填充)，并在整个窗口部件上绘制一个矩形。因为对画笔和画刷使用了不同的明暗度，使得该窗口部件好像有边界的效果，观感上也只有轻微的锯齿感。

QApplication 对象有一个 QPalette 类，含有可用于不同目的的颜色，比如文本的前景色和背景色、按钮颜色，等等。这些颜色会与它们的角色，譬如 QPalette.Text 或者 QPalette.Highlight，保持一致，尽管在这个例子中使用了一些较为晦涩的角色。事实上，一共有三套颜色方案，可分别对应窗口部件的不同状态：“激活”、“禁用”和“非激活”。每个 QWidget 也有一个 QPalette，其颜色继承自 QApplication 的调色板——而它又会根据底层窗口系统的颜色进行初始化，从而可以较好地反映出用户的偏好设置。PyQt 会尽力确保让调色板中的各个颜色协同工作，例如，提供出良好的对比度效果。从程序开发人员的角度看，可以自由使用自己喜欢的颜色，但是，尤其是对于标准需求来说，比如文本的颜色和背景，最好还是使用调色板中的颜色方案。

```

segColor = QColor(Qt.green).dark(120)
segLineColor = segColor.dark()
painter.setPen(segLineColor)
painter.setBrush(segColor)
painter.drawRect(FractionSlider.XMARGIN,
                 FractionSlider.YMARGIN, span, fm.height())

```

对于分数段，创建了深绿色，并创建了更深的绿色来作为区分它们的竖线。然后，绘制了一个包围所有各个段的矩形。

```

textColor = self.palette().color(QPalette.Text)
segWidth = span / self._denominator
segHeight = fm.height() * 2
nRect = fm.boundingRect(FractionSlider.WSTRING)
x = FractionSlider.XMARGIN
yOffset = segHeight + fm.height()

```

在这里，会基于用户的调色板来设置字体的颜色。然后，算出每一个分数段的宽度和高度，并为 x 的位置和 yOffset 赋初值。nRect 是一个矩形，它的大小足以容纳一个带有左右边白空间的数字。

```

for i in range(self._denominator + 1):
    painter.setPen(segLineColor)
    painter.drawLine(x, FractionSlider.YMARGIN, x, segHeight)
    painter.setPen(textColor)
    y = segHeight
    rect = QRectF(nRect)
    rect.moveCenter(QPointF(x, y + fm.height() / 2.0))
    painter.drawText(rect, Qt.AlignCenter, QString.number(i))
    y = yOffset
    rect.moveCenter(QPointF(x, y + fm.height() / 2.0))
    painter.drawText(rect, Qt.AlignCenter,
                    QString.number(self._denominator))
    painter.drawLine(QPointF(rect.left() + indent, y),
                    QPointF(rect.right() - indent, y))
    x += segWidth

```

在这个循环中，会绘制分割每一个分数段的竖线，分子位于每一个分数段的下方，而分母位于每一个分子的下方，它们的中间是分数线。对于 `drawText()` 方法的调用，会提供一个用来说明应该在哪里绘制文字的矩形，同时通过使用 `Qt.AlignCenter`，来确保文字可以在这个特殊的矩形内水平、竖直居中。使用同一个矩形，但是会在左右两端进行对齐，从而计算出分数线的两个端点，然后将其绘制出来。对于每个分数线、分子和分母， $y$  偏移量都是固定的，但是，在绘制完每一个分数后， $x$  偏移量则会每次增加一个分数段的宽度。

```
span = int(span)
y = FractionSlider.YMARGIN - 0.5
triangle = [QPointF(value * span, y),
            QPointF((value * span) + \
                     (2 * FractionSlider.XMARGIN), y),
            QPointF((value * span) + \
                     FractionSlider.XMARGIN, fm.height())]
painter.setPen(Qt.yellow)
painter.setBrush(Qt.darkYellow)
painter.drawPolygon(QPolygonF(triangle))
```

最后，绘制金色三角，用来说明用户选择的是哪一个分数。通过提供一个点的列表来说明这个多边形。不必在列表的最后再次重复第一个点，因为如果使用 `drawPolygon()` 的话，PyQt 会自动把起点和终点连接起来，然后再填充绘制这个封闭的区域。在下一章，将会看到更多高级绘制技术，包括功能非常丰富的 `QPainterPath` 类的用法。

现在，已经完成了这个通用的 `FractionSlider` 窗口部件。它既支持键盘也支持鼠标，在所有平台看起来都比较合理，也能够很好地与布局管理器予以合作。

相较于这个窗口部件所需的功能，`QPainter` 类能够提供更多的功能，而在下一小节，也会给出更多能做的功能，包括绘制非填充多边形和渐变填充的多边形等。还将会看到如何将其他的窗口部件嵌入到自定义窗口部件中。

#### 11.4.2 例：流体混合窗口部件

有时需要为特殊的应用程序创建一个自定义窗口部件。对于这个例子来说，假设有一个应用程序，可以用来模拟流体通过一个“Y”形管道混合的效果，如图 11.6 所示。

这个窗口部件必须绘制 3 个渐变填充的多边形和 3 个黑色轮廓（为管道形状提供一个清晰的边框轮廓），还必须允许用户设置左右的流体，也要能够显示合并后的流体。这个例子中，会选择微调框来为用户提供流体设置功能，并会用标签来显示得到最终的流体。使用这些内置窗口部件的好处在于，能够让我们只需关心它们的位置和连接即可；可以让 PyQt 来提供鼠标和键盘的交互功能，以便可以将它们予以正确显示。另一个好处是，可以使用一个“窗口”来绘制这个自定义窗口部件，也就是，使用逻辑坐标而不是设备（视口）坐标，也不必担心文本的缩放，因为仅仅是在覆盖自定义窗口部件的窗口部件中，文本才会出现，并且也不会受到窗口设置的影响。

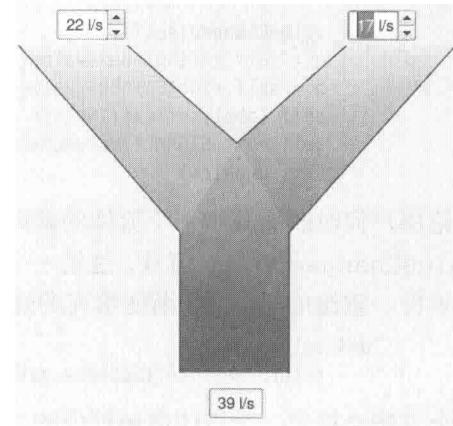


图 11.6 “Y”形管道窗口部件

先从初始化程序开始看起，会将其分为两个部分。

```
class YPipeWidget(QWidget):
    def __init__(self, leftFlow=0, rightFlow=0, maxFlow=100,
                 parent=None):
        super(YPipeWidget, self).__init__(parent)
        self.leftSpinBox = QSpinBox(self)
        self.leftSpinBox.setRange(0, maxFlow)
        self.leftSpinBox.setValue(leftFlow)
        self.leftSpinBox.setSuffix(" l/s")
        self.leftSpinBox.setAlignment(Qt.AlignRight|Qt.AlignVCenter)
        self.connect(self.leftSpinBox, SIGNAL("valueChanged(int)"),
                     self.valueChanged)
```

在调用 `super()` 之后，会创建左侧微调框，设置它的一些参数，并会将它与 `valueChanged()` 方法相连接，这个 `valueChanged()` 方法将会稍后看到。值得注意的是，这里会将微调框的父对象设置为 `self`（是 `YPipeWidget` 的实例）；这是因为，由于不会布局这个微调框，所以不会有布局管理器将其重新布局到它的父窗口部件中。右侧微调框的创建和设置代码不再给出，因为那些代码几乎与左侧微调框的代码一模一样。

```
self.label = QLabel(self)
self.label.setStyleSheet(QFrame.StyledPanel|QFrame.Sunken)
self.label.setAlignment(Qt.AlignCenter)
fm = QFontMetricsF(self.font())
self.label.setMinimumWidth(fm.width(" 999 l/s "))
self.setSizePolicy(QSizePolicy(QSizePolicy.Expanding,
                             QSizePolicy.Expanding))
self.setMinimumSize(self.minimumSizeHint())
self.valueChanged()
```

创建一个用来显示流体合并后的标签，为该标签设置若干属性。要为标签设置最小宽度值，以便在调整其大小时不会出现问题——例如，如果流量要在 9 和 10，或者 99 和 100 之间变化。要把 `YPipeWidget` 的尺寸大小策略设置为拉伸，这就意味着这个窗口部件可以在两个方向上尽可能增大。同时，还要把窗口部件的最小尺寸大小设置成最小尺寸大小提示，可以通过调用 `valueChanged()` 为该标签设置一个初始值。

```
def valueChanged(self):
    a = self.leftSpinBox.value()
    b = self.rightSpinBox.value()
    self.label.setText("%d l/s" % (a + b))
    self.emit(SIGNAL("valueChanged"), a, b)
    self.update()
```

无论用户何时改变其中一个流体的微调框，都会调用这个方法。它会更新标签，发射它自己的 `valueChanged` Python 信号，这是一个可以被任何外部窗口部件连接的信号，再触发一个重绘事件。重绘的原因是，渐变填充的颜色是与微调框的值成比例的。

```
def values(self):
    return self.leftSpinBox.value(), self.rightSpinBox.value()
```

这个方法会提供一个流体微调框值的二元元组。

```
def minimumSizeHint(self):
    return QSize(self.leftSpinBox.width() * 3,
                self.leftSpinBox.height() * 5)
```

已经让该窗口部件的最小宽度和最小高度与各个微调框的最小宽高比成比例。这样就可以确保“Y”形状不会变的太小以至于无法看清。

```
def resizeEvent(self, event=None):
    fm = QFontMetricsF(self.font())
    x = (self.width() - self.label.width()) / 2
    y = self.height() - (fm.height() * 1.5)
    self.label.move(x, y)
    y = self.height() / 60.0
    x = (self.width() / 4.0) - self.leftSpinBox.width()
    self.leftSpinBox.move(x, y)
    x = self.width() - (self.width() / 4.0)
    self.rightSpinBox.move(x, y)
```

尺寸大小调整事件对于那些窗口部件中含有其他不带布局的窗口部件特别重要。这是因为，可以使用这个事件来确定这些子窗口部件的位置。尺寸调整事件通常会在窗口部件首次显示之前调用，因此窗口部件在被用户第一次看到之前，就需要能够自动获得定位子窗口部件的机会。

这个标签是横向居中的，绘制于接近窗口部件的底端( $y$ 坐标的值是向下变大的，因此 `self.height()` 可以返回最大值——即最底端的—— $y$  值)。两个微调框绘制于窗口部件的顶端附近，分别距离最小  $y$  坐标值——即最顶端的—— $y$  值下方的  $1/60$  和窗口部件左右边缘宽度的  $1/4$ 。

由于使用了 `QSpinBox` 和 `QLabel` 以及若干的信号-槽连接，可以对全部的用户交互进行响应，因此我们只需要关心尺寸大小的调整和绘制。虽然微调框和标签是由 PyQt 绘制的，也大大简化了绘制工作，但仍有不少的东西需要处理，因此会将这个绘制事件分成不同的部分。

```
def paintEvent(self, event=None):
    LogicalSize = 100.0

    def logicalFromPhysical(length, side):
        return (length / side) * LogicalSize

    fm = QFontMetricsF(self.font())
    ymargin = (LogicalSize / 30.0) + \
              logicalFromPhysical(self.leftSpinBox.height(),
                                   self.height())

    ymax = LogicalSize - \
           logicalFromPhysical(fm.height() * 2, self.height())
    width = LogicalSize / 4.0
    cx, cy = LogicalSize / 2.0, LogicalSize / 3.0
    ax, ay = cx - (2 * width), ymargin
    bx, by = cx - width, ay
    dx, dy = cx + width, ay
    ex, ey = cx + (2 * width), ymargin
    fx, fy = cx + (width / 2), cx + (LogicalSize / 24.0)
    gx, gy = fx, ymax
    hx, hy = cx - (width / 2), ymax
    ix, iy = hx, fy
```

与其在使用设备(物理)坐标时需要手动缩放所有坐标，还不如创建一个逻辑坐标系，设置其左上角为  $(0,0)$ ，宽度和高度都设置成  $100$  (`LogicalSize`)。这里会定义一个小型辅助函数 (helper function)，用来计算绘制微调框时上方  $y$  向的边白，以及绘制的标签下方的最大  $y$  值。

如图 11.7 所示，根据绘制“Y”形状所需的全部点实施绘制工作。对于图中的每一个点，

分别计算它们的  $x$  坐标值和  $y$  坐标值。例如，对于左上角的点  $a$ ，可以吧它的坐标在代码中用  $ax$  和  $ay$  表示。大部分的坐标计算工作会根据点  $c$ , ( $cx$ ,  $cy$ ) 来完成。

```
painter = QPainter(self)
painter.setRenderHint(QPainter.Antialiasing)
side = min(self.width(), self.height())
painter.setViewport((self.width() - side) / 2,
                   (self.height() - side) / 2, side, side)
painter.setWindow(0, 0, LogicalSize, LogicalSize)
```

这里会创建一个绘图器，把它的视口设置成最大的居中正方形区域，从而能够匹配其内部的矩形。然后，设置一个窗口，即放上我们自己的逻辑坐标系，让 PyQt 来完成逻辑坐标到物理坐标之间的转换。

```
painter.setPen(Qt.NoPen)
```

不使用画笔是因为，不希望构成管道的每一个多边形都有轮廓。相反，将会在绘制事件的最后，根据需要绘制相应的那些线条。

```
gradient = QLinearGradient(QPointF(0, 0), QPointF(0, 100))
gradient.setColorAt(0, Qt.white)
a = self.leftSpinBox.value()
gradient.setColorAt(1, Qt.red if a != 0 else Qt.white)
painter.setBrush(QBrush(gradient))
painter.drawPolygon(
    QPolygon([ax, ay, bx, by, cx, cy, ix, iy]))
```

在表 11.2 和表 11.3 中，分别给出 QPainter 的部分方法。

表 11.2 QPainter 的部分方法(不包括与绘制有关的方法)

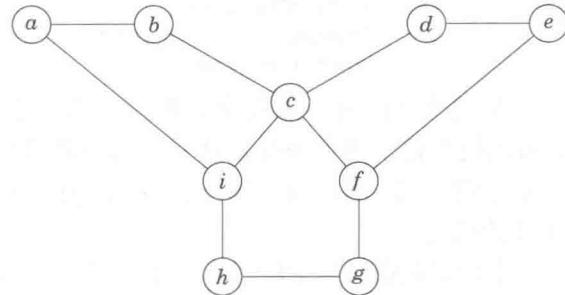


图 11.7 “Y”形管道的坐标点

语法	说明
p.restore()	把 QPainter p 的状态恢复到上次保存的状态
p.rotate(a)	根据 int a° 的值旋转 QPainter p
p.save()	保存 QPainter p 的状态，包括它的变换矩阵、画笔和画刷
p.scale(x, y)	根据 float x 和 float y 的值，分别在水平和竖直方向缩放 QPainter p；1.0 不缩放，0.5 缩小一半，3.0 放大 3 倍
p.setMatrix(m)	把 QPainter p 的变换矩阵设置成 QMatrix m
p.setRenderHint(h)	打开 QPainter.RenderHint h。渲染提示包括 QPainter.Antialiasing、QPainter.TextAntialiasing 和 QPainter.SmoothPixmapTransform
p.setViewport(x, y, w, h)	把 QPainter p 的视口(物理坐标系)限定到矩形中，矩形左上角的坐标是 (x, y)，宽度是 w，高度是 h；所有参数都是 int
p.setWindow(x, y, w, h)	把 QPainter p 的逻辑坐标系设置成矩形，矩形左上角的坐标是 (x, y)，宽度是 w，高度是 h；所有参数都是 int
p.shear(x, y)	对 QPainter p 的坐标系进行错切变换，水平错切量为 float x，竖直错切量为 float y
p.translate(dx, dy)	对 QPainter p 的坐标系进行水平 int dx 和竖直 int dy 的平移

表 11.3 QPainter 绘制相关的部分方法

语法	说明
p.drawArc(r, a, s)	根据 QPainter p 上 QRect r 的内接圆绘制一段圆弧, 起始角度为 int a°, 跨度为 s°/16
p.drawChord(r, a, s)	根据 QPainter p 上 QRect r 的内接圆绘制一根弦, 起始角度为 int a°, 跨度为 s°/16
p.drawConvexPolygon(pl)	在 QPainter p 上绘制一个凸多边形, 其顶点在 QPoint 型列表 list pl 中, 且最后一个点会回到第一个点处
p.drawEllipse(r)	在 QPainter p 上 QRect r 的内部绘制一个椭圆; 如果 r 是正方形, 则绘制一个圆
p.drawImage(pt, i)	在 QPainter p 上 QPoint pt 处绘制图片 QImage i; 不同的参数可以只绘制出图片的一部分
p.drawLine(p1, p2)	在 QPainter p 上的 QPoint p1 和 QPoint p2 之间绘制一条直线; 也可以有许多不同的参数; 此外还有 drawLines() 方法
p.drawPath(pp)	在 QPainter p 上绘制 QPainterPath pp
p.drawPie(r, a, s)	在由 QRect r 所包围的圆内绘制一个饼图, 起始角度为 int a°, 跨度为 s°/16
p.drawPixmap(pt, px)	在 QPainter p 上 QPoint pt 处绘制像素图 QPixmap px; 不同参数可以允许只绘制像素图的一部分
p.drawPoint(pt)	在 QPainter p 上绘制一个点 QPoint pt; 还有一个 drawPoints() 方法
p.drawPolygon(pl)	在 QPainter p 上绘制一个多边形, 其顶点在 QPoint 型列表 list pl 中, 且最后一个点会回到第一个点处
p.drawPolyline(pl)	在 QPainter p 上绘制一个多项线, 其顶点在 QPoint 型列表 list pl 中, 且最后一个顶点不会回到第一个顶点处
p.drawRect(r)	在 QPainter p 上绘制一个矩形 QRect r
p.drawRoundRect(r, x, y)	在 QPainter p 上绘制一个由矩形 QRect r 倒圆角而来的圆角矩形, 圆角系数分别为 int x 和 int y
p.drawText(r, s, o)	在 QPainter p 上绘制一个由矩形 QRect r 界定的字符串, 可以带一个选项 QTextOption o
p.drawText(x, y, s)	在 QPainter p 上的点 (x, y) 处绘制一个字符串
p.fillPath(pp, b)	在 QPainter p 上用 QBrush b 填充 QPainterPath pp
p.fillRect(r, b)	在 QPainter p 上用 QBrush b 填充矩形区域 QRect r
p.setBrush(b)	把待填充形状的画刷设置成 QBrush b
p.setPen(pn)	把绘制直线和轮廓线的画笔设置成 QPen pn
p.setFont(f)	把 QPainter p 的文本字体设置成 QFont f

左侧微调框可以表示“Y”形状的左侧部分——由(*a*, *b*, *c*, *i*)确定的形状——可以使用从白色到红色的线性颜色渐变。

```
gradient = QLinearGradient(QPointF(0, 0), QPointF(0, 100))
gradient.setColorAt(0, Qt.white)
b = self.rightSpinBox.value()
gradient.setColorAt(1, Qt.blue if b != 0 else Qt.white)
painter.setBrush(QBrush(gradient))
painter.drawPolygon(
    QPolygon([cx, cy, dx, dy, ex, ey, fx, fy]))
```

右侧微调框可以表示“Y”形状的右侧部分——由(*d*, *e*, *f*, *c*)确定的形状——与左侧部分的代码非常相似, 只是使用从白色到蓝色的线性颜色渐变。

```
if (a + b) == 0:
    color = QColor(Qt.white)
else:
    ashare = (a / (a + b)) * 255.0
```

```

bshare = 255.0 - ashare
color = QColor(ashare, 0, bshare)
gradient = QLinearGradient(QPointF(0, 0), QPointF(0, 100))
gradient.setColorAt(0, Qt.white)
gradient.setColorAt(1, color)
painter.setBrush(QBrush(gradient))
painter.drawPolygon(
    QPolygon([cx, cy, fx, fy, gx, gy, hx, hy, ix, iy]))

```

“Y”形状的底座部分——由( $c, f, g, h, i$ )确定的形状——使用从白色到红色/蓝色的线性颜色渐变，该渐变颜色可以与左侧/右侧的流量成正比。

```

painter.setPen(Qt.black)
painter.drawPolyline(QPolygon([ax, ay, ix, iy, hx, hy]))
painter.drawPolyline(QPolygon([gx, gy, fx, fy, ex, ey]))
painter.drawPolyline(QPolygon([bx, by, cx, cy, dx, dy]))

```

最后，绘制表示管道边界的线条。第一条从 $a$ 经 $i$ 到 $h$ ，绘制出管道的左边，第二条从 $g$ 经 $f$ 到 $e$ ，绘制出管道的右边，第三条从 $b$ 经 $c$ 到 $d$ ，标出“V”形状的顶部。

就像各个 PyQt 的内置窗口部件一样，无论是 YPipeWidget 还是 FractionSlider，都可以用做顶级窗口部件，这在开发和测试自定义窗口部件时都非常有用。chap11/fractionslider.py 和 chap11/ypipewidget.py 都可以作为独立程序运行，因为它们在 QWidget 子类的后面都有一个 `if __name__ == "__main__":` 语句，利用这些代码可以创建 QApplication，也可以创建和显示自定义的窗口部件。

## 小结

PyQt 提供了多个自定义窗口部件外观和行为的方法，这些方法也各不相同。最简单且最常用的方法是，把现有的窗口部件属性设置成我们想要的值。从 Qt 4.2 开始，可以使用样式表属性，这就允许我们只是简单使用一些 CSS 语法纯文本就能够巨大影响窗口部件的外观。样式表一个常见而又非常简单的用法就是对托管的窗口部件背景颜色进行设置。

复合窗口部件允许我们将两个或者两个以上的窗口部件布局到一起，并把得到的最终窗口部件看成一个单独的窗口部件。如果这个复合窗口部件会被多次使用时，就能够节省大量时间，同时这也提供了一种方式，使得在主窗口风格应用程序的中心区域可以出现多个窗口部件。一些程序开发人员会把这些组成的窗口部件私有化并转发它们的信号和槽，不过在很多情况下，最简单的方法就是把这些组成窗口部件作为公有实例变量，这样就可以直接访问它们，或者直接连接它们。

子类化现有的窗口部件来复用它们的外观和行为要比创建一个 QWidget 子类并自己完成所有工作要简单得多。这一方法几乎适用于每一个 PyQt 窗口部件，因为它们中的大多数都是设计用于子类化的。这个方法的唯一限制是，只能用于那些与我们想要的窗口部件足够相似的窗口部件中，才可以让复用变得可行。

如果需要创建一个不同于其他任何窗口部件的窗口部件，或者如果想完全控制自定义窗口部件的外观和行为，可以子类化 QWidget。新得到的子类必须重新实现 `paintEvent()`、`sizeHint()` 和 `minimumSizeHint()`，并且也几乎总是会重新实现 `keyPressEvent()` 和一些鼠标事件处理程序。绝大多数的内置窗口部件都是以这种方式创建的，而其余的内置窗口部件则是其他窗口部件的子类。

在这一章中自定义和创建的所有窗口部件，包括贯穿全书的各个窗口部件，它们的外观和行为也都非常传统。PyQt 并没有强制要求一定要这么保守，可以自由创建所能想到的具有任意外观和任意行为的各种窗口部件。

## 练习题

创建如图 11.8 所示的 Counters 自定义窗口部件。这个窗口部件应当显示一个  $3 \times 3$  的网格，其中，每个正方形要么空白(只显示背景色)，要么是红色或者黄色的椭圆。任一方格的状态都应当是从空白色变为黄色再变为红色，然后再变为空白色，这样无限循环。当用户点击一个方格或者是在这个方格上按下空格键，它的状态就应该发生改变。拥有键盘焦点的方格要用较粗的蓝色画笔重新绘制方框，而不像那些其他的方格，会使用较细的黑色绘制。用户应该可以通过点击方格或者使用上、下、左、右箭头键移动焦点来改变和移动拥有焦点的方格。

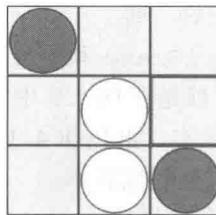


图 11.8 Counters 自定义窗口部件

要确保提供了尺寸大小提示和最小尺寸大小提示方法，以便让窗口部件能够具有正确的尺寸大小调整行为，也不会被压缩得太小。绘制事件可以相当简短，但会稍微有些复杂；可能还会需要用到 `QPainter.save()` 和 `QPainter.restore()`，以便保存和恢复绘制器的状态，从而使得用于某个方格的画笔和画刷的颜色不会传播到其他方格中。

在文件的最后要包含一个 `if __name__ == "__main__":` 语句，还要创建一个 `QApplication` 对象和一个 `Counters` 窗口部件的实例，以便可以测试它们。所有的代码不应超过 130 行。

本练习题的参考答案在文件 `chap11/counters.py` 中。

## 第 12 章 基于项的图形

- 图形项的自定义和交互
- 动画和复杂形状

如果创建一个自定义窗口部件并重新实现它的绘制事件，就可以得到任何想要的图形。在前一章，曾经给出过这一方法，并且这一方法对于绘制自定义窗口部件、图形的绘制和少量项的绘制来说都很理想。但如果需要绘制大量的单个项，如从几十个到数以万计的项，或者是需要绘制用户能够进行单独交互的项——例如，单击、拖动或者对其予以选择——又或者是如果需要对项进行动画处理，使用 PyQt 的图形视图类 (graphics view class) 而不是去重新实现一个自定义窗口部件的绘图事件来说要更好一些。

诸如 `QGraphicsView`、`QGraphicsScene` 和 `QGraphicsItem` 这样的图形视图类——还包括 `QGraphicsItem` 的各个子类，都是在 Qt 4.2 中引入的，所以在这一章，各个有关 PyQt 的例子将只能用于 Qt 4.2 及其后续的版本，如 PyQt 4.1 版。不过，对基于图形视图的那些应用程序来说，强烈推荐使用 PyQt 4.2 或其后续版本。

要使用图形视图类就必须创建一个场景 (scene)，这个场景是通过 `QGraphicsScene` 对象来表示的。场景都纯粹是数据，而只有将它们与一个或多个 `QGraphicsView` 对象相关联时才可以实现可视化。在场景中绘制的项都是 `QGraphicsItem` 的子类。PyQt 提供了数个预定义的子类，包括 `QGraphicsLineItem`、`QGraphicsPixmapItem`、`QGraphicsSimpleTextItem`(纯文本) 和 `QGraphicsTextItem`(HTML)。正如将在本章后续内容中看到的那样，创建一些自定义图形项子类也是可能的。

场景一旦创建，并且向其添加了项，就可以使用 `QGraphicsView` 来进行场景的可视化。图形视图的一个重要而强大的功能就是可以对项应用变换，例如缩放和旋转，这些变化可以影响场景的呈现方式，但不会改变场景中任何项的自身内容。另外，将多个图形视图与某个特定的场景予以关联也是可以的，这样就可以允许查看场景中的不同部分，还可以让每个视图都能够使用各自的变换。

各个图形视图类基本上都是二维的；然而，每个项都有一个 *z* 值，那些 *z* 值较高的项就会绘制在那些 *z* 值较低的项之上。碰撞检测 (collision detection) 是基于项的一些位置 (*x*, *y*)。除了有关碰撞的信息之外，场景可以说明哪些项会包含一个特定的点或者这些项是在哪个特定的区域，以及选中了哪些项。场景有一个有用的前景层，例如，可以为场景中的所有项绘制一个网格；场景也可以有一个背景层，是在所有项的下面进行绘制的，在提供一个背景图像或者背景色时很有用。

项既可以是场景的一些子对象(很像 PyQt 中常见的父-子窗口部件之间的那种关系)，又可以是另一个项的子对象。当对一个项应用变换时，这些变换就会自动应用于该项的所有子对象上，并逐层递归到最下面的子孙。这就意味着，如果移动一个项——例如，由用户进行拖动操作——就会移动它的所有子对象。存在多个同等项群组也是可能的，也就是说，对群组中的一个项应用变换就只会影响该项的子对象，而不会影响群组中的其他成员。

图形视图类可以使用三种不同的坐标系，尽管实际中通常只会使用其中的两种。视图使

用的是物理坐标系 (physical coordinate system)。场景使用逻辑坐标系 (logical coordinate system)，这个坐标系是在创建场景时选择的。PyQt 会自动把场景坐标系映射视图坐标系。从本质上来说，场景使用“窗口”(window)坐标系而视图使用“视口”(viewport)坐标系。所以，在对项进行定位时，会用场景坐标系的方式来放置它们。第三种坐标系是由项所使用的坐标系。这是一个特别方便的坐标系，因为它是以点(0, 0)为中心的逻辑坐标系。每个项的(0, 0)点实际上就是该项在场景中的位置。这就意味着，在实践应用中，总是可以用项的中心点的形式来绘制它们——并且也不需要关心那些经由父辈项而应用过来的任何变换，因为场景会自动替我们处理好这些关系。图 12.1 说明了场景和项的坐标系之间的关系。

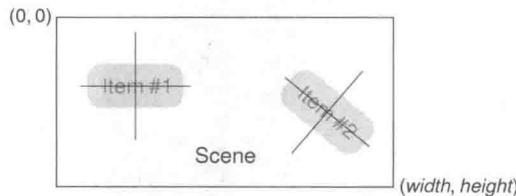


图 12.1 基于局部逻辑坐标系的图形项

在这一章，我们将会看到两个例子，并从这两个例子中可以看到图形视图类很多的不同方面。第一个例子是一个典型的应用程序，用户会在其中一个接一个地创建各个项，也会单独或者群组性地操纵各个项。这个应用程序还会给出一些用户交互的说明，包括选择、移动和调整项的大小。第二个例子会给出用一些复杂形状来模拟的动画复合项。它还会演示如何根据场景的细节级别 (Levels of Detail, LOD, 如何对项进行放大或缩小) 来尽量减少项绘制的工作量<sup>①</sup>。

## 12.1 图形项的自定义和交互

那些预定义图形项可以借助适当的常数来对它们调用 `setFlags()` 而使其可移动、可选择以及可获得焦点。用户可以用鼠标拖动那些可移动的项，还可以通过单击它们来选择那些可选的项，并且使用 `Ctrl` 键加单击鼠标的方式来选择多个项。可获得焦点的项将能够接收键事件，不过会忽略这些事件，除非是用带有键事件处理程序而创建的项的子类。同样，通过子类化并实现一些适当的鼠标事件处理程序可以让项来响应鼠标事件。

在这一节中，将会用到预定义的两个图形项，并创建两个自定义图形项子类来说明是如何使用图形项的，还会说明该如何控制它们的行为和外观。同时，也将会看到该如何加载和保存场景，以及该如何来打印它们。有关这些事情的做法可以查看图 12.2 中所示的页面设计器 (Page Designer) 应用程序。这个程序允许用户创建一个包含有多个文本、图片和框的页面。用户还可以创建一些行——它们只是一些 1 像素宽或高的框。由用户创建的图片可以保存和加载.`.pgd` 格式的文件，这是一种针对这个应用程序而自定义的特殊文件格式，也可以通过打印对话框来打印它们 (或保存成 PDF 文件)。

① 细节级别又叫 LOD 技术，即 Levels of Detail 的简称。LOD 技术是指，根据物体模型的节点在显示环境中所处的位置和重要度，决定渲染物体模型所分配的内存、CPU 等计算资源，降低非重要物体的面数和细节度，从而获得较高的渲染运算效率——译者注。

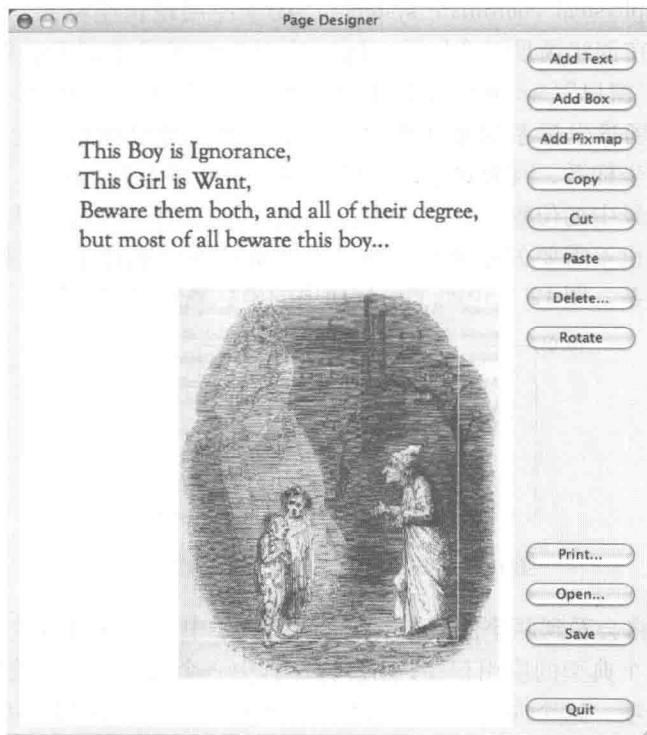


图 12.2 页面设计师(Page Designer)应用程序

对于文本项(text item)来说,会使用 `QGraphicsTextItem` 的子类,对它们进行扩展以允许用户通过双击设置项的字体和文本。对于那些框式(和行式)项,可以使用 `QGraphic-  
sItem` 的子类。这里会有一个上下文菜单,外加调整大小的键盘支持,它就可以处理它的所有图形绘制工作了。pixmap位图项只需使用内置的 `QGraphicsPixmapItem` 类,页和页边空白则可以使用内置的 `QGraphicsRectItem` 类。显示场景的视图会使用能够支持橡皮筋框选方法和鼠标滚轮缩放的 `QGraphicsView` 子类。

下面会首先看看 `QGraphicsView` 子类。然后,会回顾其主体形式,最后,再来看看自定义 `QGraphicsItem` 的子类。

```
class GraphicsView(QGraphicsView):
    def __init__(self, parent=None):
        super(GraphicsView, self).__init__(parent)
        self.setDragMode(QGraphicsView.RubberBandDrag)
        self.setRenderHint(QPainter.Antialiasing)
        self.setRenderHint(QPainter.TextAntialiasing)

    def wheelEvent(self, event):
        factor = 1.41 ** (-event.delta() / 240.0)
        self.scale(factor, factor)
```

上面的代码是 `GraphicsView` 子类的完整代码。在初始化程序中设置了拖拽模式:这就意味着,在视图中拖拽将导致 PyQt 给出一个橡皮筋选择框,并通过橡皮筋选择框的缩放来选中每个项。渲染提示(render hint)会传递到视图中所有需要绘制的项,所以并不需要为每一个单独的视图项进行渲染提示设置。

每当用户滚动鼠标滚轮并且根据滚轮的方式导致视图缩放的较小或较大时，就会调用滚轮事件。这一操作的效果就是会更改页面的外观尺寸——而底层的场景却并不会发生改变。在这个事件处理程序中，所使用的数学思想还是比较巧妙的，但这并不是什么问题，因为该方法可以“按原样”进行复制和粘贴。

在文件 chap12/pagedesigner.pyw 中靠近顶部的地方，会有一些全局声明。

```
PageSize = (612, 792)
PointSize = 10

MagicNumber = 0x70616765
FileVersion = 1

Dirty = False
```

页面的尺寸大小为以磅为单位的美国信纸的大小（源代码中也有 A4 页面大小的纸张，但把这部分注释掉了）。QDataStream 会使用幻数（magic number）和文件的版本信息，正如在第 8 章和其他地方所看到的那样。这里还包含了一个全局的 dirty 标志。

在这之前还没有给出导入的内容，但它们会包含 functools。这是需要的，因为在上下文菜单中需要使用 functools.partial() 函数来封装可以用适当参数调用的那些方法。

主窗体的初始化程序相当长，所以这里会分开来看这些代码，并且会省略那些之前在其他地方已经见到过的类似代码——例如，在创建和布置窗体各个按钮的地方就见过。

```
class MainForm(QDialog):
    def __init__(self, parent=None):
        super(MainForm, self).__init__(parent)

        self.filename = QString()
        self.copiedItem = QByteArray()
        self.pasteOffset = 5
        self.prevPoint = QPoint()
        self.addOffset = 5
        self.borders = []

        self.printer = QPrinter(QPrinter.HighResolution)
        self.printer.setPageSize(QPrinter.Letter)
```

所复制的项本质上是一大块描述要剪切或复制的最新项的二进制数据。会在应用程序内存储这些数据，而不是放到剪贴板中，因为这些内容不会用于任何其他应用程序的。粘贴补偿（paste offset）会用在以下情况中：用户反复粘贴相同的项，就像之前说明的那些情况以及用户多次添加相同的项类型时，都会用到粘贴补偿技术。在两种情况下，新添加的项添加到补偿位置，而不是正好位于前一项。这样一来，就更容易使用户能够看到那些项是在哪个位置了。

边框列表中将包含两个图形项，两个均为黄色的矩形：一个用做页面轮廓，一个用做页面轮廓内允许用做边白的空间。它们都只是作为轮廓，是不会被保存或打印的。

尽管在需要有 QPrinter 对象时再来创建它是可以的，就是通过创建一个该对象并使其作为实例变量，可以确保用户的各项设置，例如页面的尺寸大小等，能够将它们保留在同一个任务中的不同用处。

```
self.view = QGraphicsView()
self.scene = QGraphicsScene(self)
self.scene.setSceneRect(0, 0, PageSize[0], PageSize[1])
self.addBorders()
self.view.setScene(self.scene)
```

表 12.1 QGraphicsScene 的部分方法

语法	说明
s.addEllipse(r, pn, b)	在 QGraphicsScene s 中添加一个由 QRectF r 所限定的椭圆，其轮廓由 QPen pn 确定并可用 QBrush b 进行填充
s.addItem(g)	向 QGraphicsScene s 中添加一个 QGraphicsItem g。还有其他一些内置的图形项 add*() 创建和添加方法
s.addLine(l, pn)	向 s 中添加 QLineF l，绘制的画笔是 QPen pn
s.addPath(pp, pn, b)	向 QGraphicsScene s 中添加一个 QPainterPath pp，其轮廓由 QPen pn 确定并可用 QBrush b 进行填充
s.addPixmap(px)	把 QPixmap px 添加到 QGraphicsScene s 中
s.addPolygon(pg, pn, b)	向 QGraphicsScene s 中添加一个 QPolygon pg，其轮廓由 QPen pn 确定并可用 QBrush b 进行填充
s.addRect(r, pn, b)	向 QGraphicsScene s 中添加一个 QRect r，其轮廓由 QPen pn 确定并可用 QBrush b 进行填充
s.addText(t, f)	向 QGraphicsScene s 中使用 QFont f 字体添加文本 t
s.collidingItems(g)	返回一个与 QGraphicsItem g 相冲突的 QGraphicsItem 列表(也可能是空的)
s.items()	返回 QGraphicsScene s 中的全部 QGraphicsItem；使用不同参数，位于特定点或者是在给定的矩形、多边形或者绘制路径中的那些项都可以返回
s.removeItem(g)	从 QGraphicsScene s 移除 QGraphicsItem g；并将拥有权传给调用程序
s.render(p)	渲染程序在 QPainter p 上渲染 QGraphicsScene s；多余的参数可以用来控制源矩形和目标矩形
s.setBackgroundBrush(b)	将 QGraphicsScene s 的背景设置成 QBrush b
s.setSceneRect(x, y, w, h)	把 QGraphicsScene s 所在的矩形设置到位置 (x, y)，矩形的宽度和高度分别为 w 和 h；各个参数类型都是 float 的
s.update()	为 QGraphicsScene s 安排一个绘制事件
s.views()	返回一个(可能是空的) QGraphicsView 列表来显示 QGraphicsScene s

这里创建自定义 GraphicsView 类的一个实例，还会创建一个标准的 QGraphicsScene。设置用在场景上的矩形是“窗口”，也就是说，场景将会使用逻辑坐标系——在本例中，矩形左上角点为(0, 0)，而其宽度和高度对应页面的大小，单位是磅。

初始化程序的其余部分会创建并连接各个按钮，还会对按钮和视图进行布局设置。

```
def addBorders(self):
    self.borders = []
    rect = QRectF(0, 0, PageSize[0], PageSize[1])
    self.borders.append(self.scene.addRect(rect, Qt.yellow))
    margin = 5.25 * PointSize
    self.borders.append(self.scene.addRect(
        rect.adjusted(margin, margin, -margin, -margin),
        Qt.yellow))
```

这一方法会创建两个 QGraphicsRectItem，第一个对应页的大小，第二个(说明页边距)在第一个的内部。QRect.adjusted()方法会返回一个矩形，该矩形的左上角点和右下角点可通过 dx 和 dy 这两个数据集来进行调整。在这里，会把左上角朝右和下移动(通过增加这里每一个 margin 数值来实现)，也会把右下角向左和向上移动(通过减小每个 margin 数值来实现)。

```
def removeBorders(self):
    while self.borders:
        item = self.borders.pop()
        self.scene.removeItem(item)
        del item
```

在打印或保存时，通常不希望包含各条边。这个方法就会从 `self.borders` 列表中解析出每个项（以随机的顺序），并从场景中移除这些项。当从场景中移除一个项时，场景就会自动通知它的视图，以便可以重新绘制那些未被覆盖的区域。删除操作还有一种替代方法，就是通过调用 `setVisible(False)` 来隐藏这些边框的。

调用 `QGraphicsScene.removeItem()` 可以从场景中移除一个项（以及它的子项），但不会删除这个项，而是将其所有权传递给它的调用者。所以，在 `removeItem()` 调用后，这个项仍然会存在。本来也是可以在每个项引用超出范围后，就会自然删除那些项，不过，我们大多人还是更喜欢采用显式删除项的方式，以便可以让我们清楚地知道已经获得了这些项的主动权并且确实是正在删除它们。

```
def addPixmap(self):
    path = QFileInfo(self.filename).path() \
        if not self.filename.isEmpty() else "."
    fname = QFileDialog.getOpenFileName(self,
                                        "Page Designer - AddPixmap", path,
                                        "Pixmap Files (*.bmp *.jpg *.png *.xpm)")
    if fname.isEmpty():
        return
    self.createPixmapItem(QPixmap(fname), self.position())
```

当用户点击 `AddPixmap` 按钮时，就会调用这个方法。如果只是为了获得用户打算添加到页面的图像文件的名字，那么就可以把工作传递给 `createPixmapItem()` 方法即可。之所以没有在同一个方法中处理所有全部的事情是因为，将各个功能分开处理会更为方便些——例如，对于从页面设计师应用程序 `Page Designer` 的 `.pgd` 文件中加载一些像素图时，分开处理就会比较方便些。`position()` 方法用来获得一个项的位置，就是该项应该添加到的位置；很快就会看到这一做法的。

```
def createPixmapItem(self, pixmap, position, matrix=QMatrix()):
    item = QGraphicsPixmapItem(pixmap)
    item.setFlags(QGraphicsItem.ItemIsSelectable |
                  QGraphicsItem.ItemIsMovable)
    item.setPos(position)
    item.setMatrix(matrix)
    self.scene.clearSelection()
    self.scene.addItem(item)
    item.setSelected(True)
    global Dirty
    Dirty = True
```

这些图形视图类中所包括的 `QGraphicsPixmapItem` 非常适合在场景中显示图像。在 Qt 4.2 中，`QGraphicsItem` 有三个标志，分别是 `ItemIsMovable`、`ItemIsSelectable` 以及 `ItemIsFocusable`（在 Qt 4.3 中，分别又添加了 `ItemClipsToShape`、`ItemClipsChildrenToShape` 和 `ItemIgnoresTransformations` 这三个标志，其中，最后一个特别适用于显示那些不打算进行视图转换的文本）。

项创建后，就可以在场景中设置它的位置了。`setPos()` 方法是唯一的、可直接使用场景坐标的项方法；而所有的其他方法都仅能使用于项的局部逻辑坐标系中。无须设置变换矩阵

(因为由 `QMatrix()` 所返回的变换矩阵就是一个单位矩阵)，不过，还是希望能够得到一个显式的矩阵，以便在遇到需要保存和加载(或复制以及粘贴)场景的各个项时能够使用这个矩阵<sup>①</sup>。

`QMatrix` 类会带有一个  $3 \times 3$  矩阵，它也是专为图形变换而设计的，因而它并不是一个普通的矩阵类。因此，它是 Qt 各个类中少有的命名名称较差的一个个例。不过，从 Qt 4.3 开始，`QMatrix` 就已经被更为明智地命名为 `QTransform` 类，这个类会使用一个  $4 \times 4$  矩阵，所以也就可以获得更为强大的变换功能。

项一旦设置完毕，就可以清除那些现有的选择并把项添加到场景中。然后选择这个项，为用户和它的交互做好准备。

```
def position(self):
    point = self.mapFromGlobal(QCursor.pos())
    if not self.view.geometry().contains(point):
        coord = random.randint(36, 144)
        point = QPoint(coord, coord)
    else:
        if point == self.prevPoint:
            point += QPoint(self.addOffset, self.addOffset)
            self.addOffset += 5
        else:
            self.addOffset = 5
            self.prevPoint = point
    return self.view.mapToScene(point)
```

这个方法用来在场景中为新添加的项提供它所在的位置。如果鼠标位于视图上方，就使用由 `QCursor.pos()` 所提供的鼠标位置来作为项所在的位置——在这里所说的“光标”意思是鼠标的光标——不过，如果在同一个地方还添加过另外一个项，那么就会外加一个偏移量。这就意味着，如果用户多次按下添加按钮 Add，后续添加的每个项都会与之前所添加的每个项有一个位置偏移量，这样也就可以让用户更为轻松地看到它们并与它们进行交互。如果鼠标是在视图之外，那么就把这个项放在场景靠近左上角的半随机位置。

`MapFromGlobal()` 方法可将屏幕坐标转换成物理窗口部件的坐标，就像在视图中所用的方法那样。但场景会使用它们自己的逻辑坐标系，所以必须使用 `QGraphicsView.mapToScene()` 把物理坐标转换成场景的坐标。

```
def addText(self):
    dialog = TextItemDlg(position=self.position(),
                          scene=self.scene, parent=self)
    dialog.exec_()
```

当用户单击添加文本按钮 Add Text 时，就会调用这个方法。它会弹出一个智能的添加/编辑项对话框，如图 12.3 所示。如果用户单击确定 OK 按钮，就会添加一个使用了其中文本的项，并会用到用户所选定的字体。这里将不会讨论这个对话框，因为它与图形编程的内容无关；在 `chap12/pagedesigner.pyw` 中可以看到它的源代码。

并不需要保留对所添加的项的引用，因为会将它的所有权传递给智能对话中的场景上。

```
def addBox(self):
    BoxItem(self.position(), self.scene)
```

<sup>①</sup> 这里所说的单位矩阵 (identity matrix) 是指，在设置变换的时候，而不会产生任何变换的矩阵。

当用户单击添加框按钮 Add Box 时，将会调用这个方法。用户可以调整框的尺寸大小，甚至可以通过箭头键把它变成一条线(通过把宽度或高度减少为 1 像素)，正如之后将要看到的那样。

同样，还是不需要对添加进来的框项保持引用，因为它的所有权还是会传递给场景的。

在页面设计器(Page Designer)应用程序中，还是希望能够给用户提供剪切、复制和粘贴项的功能的，不过，由于这些项对于其他应用程序来说没有什么意义，所以这里不会使用剪贴板。

```
def copy(self):
    item = self.selectedItem()
    if item is None:
        return
    self.copiedItem.clear()
    self.pasteOffset = 5
    stream = QDataStream(self.copiedItem, QIODevice.WriteOnly)
    self.writeItemToStream(stream, item)
```

如果用户用到了复制动作，再开始看看是否恰好有项是选中的。如果确实有选中的项，那么就清空复制项的字节数组，并再创建一个数据流来写入字节数组。这里没必要使用 `QDataStream.setVersion()` 是因为，这里的数据流仅用于该应用程序运行期间的剪切、复制和粘贴动作，所以只要是在当前版本中，无论怎么使用都会工作良好。随后将会看到 `writeItemToStream()` 以及相应的 `readItemFromStream()` 方法。

```
def selectedItem(self):
    items = self.scene.selectedItems()
    if len(items) == 1:
        return items[0]
    return None
```

这个方法会返回一个选择项，或者在没有选中项的时候返回 `None`，又或者在有两个或者两个以上的选中项时什么也不返回。`QGraphicsScene.selectedItems()` 方法可以返回所有选中的项的清单。还有一些 `items()` 方面的方法，可以返回项的一些列表，这些项可以是某个相交点的项，也可以是在某个特定长方形或者多边形内的项，还有一个 `collidingItems()` 方法，可以用来报告碰撞的情况。

```
def cut(self):
    item = self.selectedItem()
    if item is None:
        return
    self.copy()
    self.scene.removeItem(item)
    del item
```

这个方法会使用 `copy()` 来复制所选定的项，然后将该项从场景中移除。正如之前在讨论移除边框矩形时所述，`removeItem()` 只是将一个项从场景中移除；它是不会删除这个项的。本来在项引用超出范围时，是可以删除这个项的，不过，我们还是更喜欢显式删除的方法来明确说明我们是拥有它的所有权的，也确实是真正删除了这个项的。

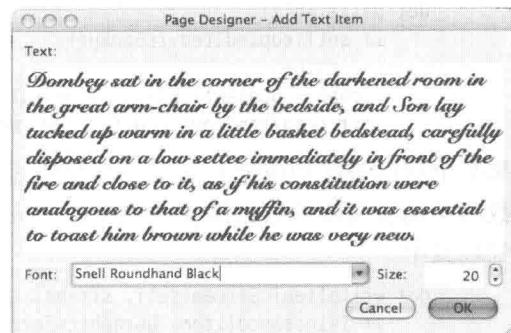


图 12.3 添加一个新的文本项

```

def paste(self):
    if self.copiedItem.isEmpty():
        return
    stream = QDataStream(self.copiedItem, QIODevice.ReadOnly)
    self.readItemFromStream(stream, self.pasteOffset)

```

如果某个项已被剪切或复制到了一个复制过的项中，就只需创建一个数据流并从复制项的字节数组中读取该项的数据即可。`readItemFromStream()`方法会处理项的创建事宜并将其添加到场景中。

```

def writeItemToStream(self, stream, item):
    if isinstance(item, QGraphicsTextItem):
        stream << QString("Text") << item.pos() << item.matrix() \
            << item.toPlainText() << item.font()
    elif isinstance(item, QGraphicsPixmapItem):
        stream << QString("Pixmap") << item.pos() \
            << item.matrix() << item.pixmap()
    elif isinstance(item, BoxItem):
        stream << QString("Box") << item.pos() << item.matrix() \
            << item.rect
    stream.writeInt16(item.style)

```

该方法会在复制、剪切(间接地)和保存中用到。对于每一个项，都会写入一个字符串来描述该项的类型，然后是该项的位置和变换矩阵，以及那些与项相关的数据。对于文本项来说，还会额外有一些数据，分别是文本项的文本内容和字体；对于像素映像项来说，其额外数据是像素映像内容自身——这就意味着，.pgd 文件可能会相当冗长；而对于边框来说，额外的数据则是框的矩形和线的样式。

```

def readItemFromStream(self, stream, offset=0):
    type = QString()
    position = QPointF()
    matrix = QMatrix()
    stream >> type >> position >> matrix
    if offset:
        position += QPointF(offset, offset)
    if type == "Text":
        text = QString()
        font = QFont()
        stream >> text >> font
        TextItem(text, position, self.scene, font, matrix)
    elif type == "Box":
        rect = QRectF()
        stream >> rect
        style = Qt.PenStyle(stream.readInt16())
        BoxItem(position, self.scene, style, rect, matrix)
    elif type == "Pixmap":
        pixmap = QPixmap()
        stream >> pixmap
        self.createPixmapItem(pixmap, position, matrix)

```

这个方法既用于 `paste()` 方法中，又用于 `open()` 方法中(用来加载一个.pgд文件)。一开始，它会读取项的类型、位置以及每个类型的项的变换矩阵。然后，它会通过偏移量来调整每个项的位置——这一点仅会在粘贴一个项的时候才会用到。接下来，就会读取那些与项相关的数据，并使用那些收集的数据创建出适当的项来。

TextItem 和 BoxItem 的初始化程序以及 createPixmapItem() 方法，它们都会创建出相应的图形项并将它们的所有权传递给场景。

```
def rotate(self):
    for item in self.scene.selectedItems():
        item.rotate(30)
```

如果用户点击旋转按钮 Rotate，所有的选定项都会旋转 30°。在这个应用程序中没有用到子项，但如果那些转动的任何项有子项的话，这些子项也都会跟着进行旋转。

```
def delete(self):
    items = self.scene.selectedItems()
    if len(items) and QMessageBox.question(self,
        "Page Designer - Delete",
        "Delete %d item%s?" % (len(items),
        "s" if len(items) != 1 else ""),
        QMessageBox.Yes|QMessageBox.No) == QMessageBox.Yes:
        while items:
            item = items.pop()
            self.scene.removeItem(item)
            del item
        global Dirty
        Dirty = True
```

如果用户点击删除 Delete 按钮并至少有一个选中的项，那么就会询问用户，要删除的选中项是否就是打算删除的，如果选择是，就会删除这个选中项。

```
def print_(self):
    dialog = QPrintDialog(self.printer)
    if dialog.exec_():
        painter = QPainter(self.printer)
        painter.setRenderHint(QPainter.Antialiasing)
        painter.setRenderHint(QPainter.TextAntialiasing)
        self.scene.clearSelection()
        self.removeBorders()
        self.scene.render(painter)
        self.addBorders()
```

QPrinter 是一个绘图设备，就像 QWidget 或 QImage 一样，所以可以很容易地对打印机进行绘制。在这里，由于已经利用过 QGraphicsScene.render() 便捷方法，它可以将整个场景（或者是它的某个裁剪区域）轻松绘制到绘图设备上。在绘制之前，可以先移除边框（就是那些黄色的矩形），在绘制之后，再恢复这些边框即可。在绘图前还需要清空选择的内容，因为如果有些选择内容是选中的，渲染的结果可能会有所不同。类似的 QGraphicsView.render() 方法也可以用来渲染场景（或者是选定的某些部分），正如之前所见的那样。

对于保存和加载.pgd 文件的代码这里会略去不谈，因为它们与之前看到的二进制文件的操作非常类似。为简便起见，这里会创建一个 QDataStream，对其调用 setVersion()，然后再写入一个幻数和一个文件版本号。接下来，就会对场景中的所有项进行遍历，由数据流调用参数化的 writeItemToStream()，并以项为单位进行依次调用。对于加载操作，还会再同样创建一个 QDataStream。然后，读取幻数和文件版本号，如果两者正确，则删除所有现有项。只要文件中有数据，就要数据流依次调用 readItemFromStream()。这一方法会依次读取项的全部数据并依次创建出所有的项，同时将这些项逐一添加到场景中。

现在，从总体上已经看完了应用程序是如何工作的，也看到应用程序是如何创建和使用这

两个预定义的图形项类，即 `QGraphicsRectItem` 和 `QGraphicsPixmapItem`。现在，我们就可以把注意力转向自定义图形视图项(custom graphics view item)了。这里先从 `TextItem` 子类开始；这个子类会用一些附加行为来对 `QGraphicsTextItem` 类进行扩展，不过还是会将全部的绘制操作功能留给基类处理的。然后，将会看看 `BoxItem` 类；在这个类中，则会给出行为和绘图两个方面的代码。

```
class TextItem(QGraphicsTextItem):
    def __init__(self, text, position, scene,
                 font=QFont("Times", PointSize), matrix=QMatrix()):
        super(TextItem, self).__init__(text)
        self.setFlags(QGraphicsItem.ItemIsSelectable |
                      QGraphicsItem.ItemIsMovable)
        self.setFont(font)
        self.setPos(position)
        self.setMatrix(matrix)
        scene.clearSelection()
        scene.addItem(self)
        self.setSelected(True)
        global Dirty
        Dirty = True
```

`TextItem` 的初始化函数与用来创建和初始化 `QGraphicsPixmapItems` 的 `createPixmapItem()` 方法非常相似。如果没有给该初始化函数提供默认的字体和默认的变换矩阵(单位矩阵)，那么就会给它们提供这些内容。

```
def parentWidget(self):
    return self.scene().views()[0]
```

一个项的父项要么是另外的一个项，要么是一个场景。不过，有的时候需要知道那个让项出现的可见窗口部件，也就是说，该项的视图。对于项来说，场景是可做这一目的的，也可以返回一个正在呈现场景的视图列表。为方便起见，假设总有至少一个视图需要用来显示场景，并且假设第一个视图就是“父”视图。

```
def itemChange(self, change, variant):
    if change != QGraphicsItem.ItemSelectedChange:
        global Dirty
        Dirty = True
    return QGraphicsTextItem.itemChange(self, change, variant)
```

如果用户与项进行交互(例如，移动或选择它)，那么就会调用这个方法。如果所做的交互不只是选择状态的变化，还可以设置一个全局的 dirty 标志。

对于 `itemChange()` 的重新实现来说，有两点需要予以重点说明。第一点，必须始终返回基类调用执行后的结果，第二点是，决不能在这个方法里面做任何操作，否则将导致另一个(递归)`itemChange()` 调用的发生。尤其要说明的是，决不能在 `itemChange()` 内调用 `setPos()` 方法。

```
def mouseDoubleClickEvent(self, event):
    dialog = TextItemDlg(self, self.parentWidget())
    dialog.exec_()
```

如果用户双击该项，就会弹出一个智能对话框，用户可以通过它来更改项的文本和字体。这个对话框与用来添加一个文本项的那个对话框是一样的。

## 图像绘制

图像绘制 (printing image) 通常来说是与场景绘制一样简单的。假设这里给出的 printer 是 QPrinter 时, printImage() 方法就将可以把任意的 QImage 或 QPixmap (两者都可以加载 .bmp、.png、.jpg 以及其他各种图形文件类型) 绘制到单一页面上:

```
def printImage(image, printer, scaleToFillPage=False):
    dialog = QPrintDialog(printer)
    if dialog.exec_():
        painter = QPainter(printer)
        painter.setRenderHint(QPainter.Antialiasing)
        rect = painter.viewport()
        size = image.size()
        size.scale(rect.size(), Qt.KeepAspectRatio)
        painter.setViewport(rect.x(), rect.y(),
                            size.width(), size.height())
        if scaleToFillPage:
            painter.setWindow(image.rect())
        if isinstance(image, QPixmap):
            painter.drawPixmap(0, 0, image)
        else:
            painter.drawImage(0, 0, image)
```

与 QPicture 的绘制很相似, 而不同之处在于, 图像尺寸的大小必须基于图片边框尺寸的大小靠自己来计算, 并且需要调用 QPainter.drawPicture() 来绘制图形。

也可以绘制 SVG 图像。做法与用于绘制 QGraphicsScenes 的方法非常类似。QSvgRenderer 类可以加载一幅 SVG 图像, 还有一个 render() 方法, 可以用这个方法在任何绘图设备上进行图像的绘制, 包括在 QPrinter 上。对于 Qt 4.3 用户来说, 绘制时通过使用 QSvgGenerator 类, 现在已经可以创建一幅 SVG 图像了, 这里的 QSvgGenerator 类可以认为是一个绘图设备<sup>①</sup>。

这样文本项类就完成了。由于这里关注的只是其行为的改变, 所以这里的内 容就显得比较少。对于接下来将会讲到的 BoxItem 类, 则会提供其行为和外观方面的所有代码。

```
class BoxItem(QGraphicsItem):
    def __init__(self, position, scene, style=Qt.SolidLine,
                 rect=None, matrix=QMatrix()):
        super(BoxItem, self).__init__()
        self.setFlags(QGraphicsItem.ItemIsSelectable |
                      QGraphicsItem.ItemIsMovable |
                      QGraphicsItem.ItemIsFocusable)
        if rect is None:
            rect = QRectF(-10 * PointSize, -PointSize,
                          20 * PointSize, 2 * PointSize)
        self.rect = rect
        self.style = style
        self.setPos(position)
        self.setMatrix(matrix)
        scene.clearSelection()
```

<sup>①</sup> 有关文档的绘制, 包括图片的绘制, 都会在下一章中讲述。

```

scene.addItem(self)
self.setSelected(True)
self.setFocus()
global Dirty
Dirty = True

```

边框项必须要有接收键盘焦点的能力，因为希望用户能够使用箭头键来调整框的尺寸大小。如果没有明确给出框的尺寸大小——比如当用户点击添加框按钮 Add Box 时，而不是从文件中重新创建一个框或者是从复制的项中粘贴过来的一个框——都需要能够提供一个默认的尺寸大小，以此作为信箱框的形状尺寸。还会提供一个默认的实线线型和一个默认的单位矩阵。初始化程序的其余部分则与之前在那些文本项中所用到的都是一样的。

对于 `parentWidget()` 和 `itemChange()` 的代码这里会予以省略，因为它们的实现代码与用于 `TextItem` 类中的两者的代码是相同的。

```

def setStyle(self, style):
    self.style = style
    self.update()
    global Dirty
    Dirty = True

```

这个方法用来设置框的线型。它会告诉场景哪些项需要重新绘制，并且会设置这些项的 `dirty` 标志，因为在保存到 `.pgd` 文件时，会记录边框项的线型。

```

def contextMenuEvent(self, event):
    wrapped = []
    menu = QMenu(self.parentWidget())
    for text, param in (
        ("&Solid", Qt.SolidLine),
        ("&Dashed", Qt.DashLine),
        ("D&otted", Qt.DotLine),
        ("D&ashDotted", Qt.DashDotLine),
        ("DashDo&tDotted", Qt.DashDotDotLine)):
        wrapper = functools.partial(self.setStyle, param)
        wrapped.append(wrapper)
        menu.addAction(text, wrapper)
    menu.exec_(event.screenPos())

```

如果用户点击了上下文菜单中的项（例如，在某些系统平台上，可以通过单击鼠标右键来调出上下文菜单），就会调用这个方法<sup>①</sup>。这个上下文菜单会使用偏函数应用程序（partial function application）来封装所调用的方法，即 `setStyle()` 方法，并且在调用它时会带一个参数，这是一个 PyQt 内置线型类的参数。所封装的函数必须要有足够长的存续时间，即要一直存续到菜单交互完成为止，因为只有在那时，其中的一个函数才会得到调用。为此，就需要有一个当地列表来存放那些封装函数；只有当菜单的 `exec_()` 完成之后，这个列表才会失效，此时，`contextMenuEvent()` 自身也会完全结束。

```

def keyPressEvent(self, event):
    factor = PointSize / 4
    changed = False
    if event.modifiers() & Qt.ShiftModifier:
        if event.key() == Qt.Key_Left:

```

<sup>①</sup> 需要注意的是，如果已经指定了视图的上下文菜单事件处理方法，那么就不会再来调用这个方法——例如，通过将它的上下文菜单策略设置成 `Qt.ActionsContextMenu` 就不会再调用这个方法了。

```

        self.rect.setRight(self.rect.right() - factor)
        changed = True
    elif event.key() == Qt.Key_Right:
        self.rect.setRight(self.rect.right() + factor)
        changed = True
    elif event.key() == Qt.Key_Up:
        self.rect.setBottom(self.rect.bottom() - factor)
        changed = True
    elif event.key() == Qt.Key_Down:
        self.rect.setBottom(self.rect.bottom() + factor)
        changed = True
    if changed:
        self.update()
        global Dirty
        Dirty = True
    else:
        QGraphicsItem.keyPressEvent(self, event)

```

如果用户按下箭头键且视图中有滚动条，视图就会做出适当的滚动。这个方法会拦截箭头键的按压情况，当某项拥有键盘焦点时，按下 Shift 键就会发生这种拦截作用，这样就可以给用户提供一种调整边框尺寸大小的手段。所述的 `QRect.setRight()` 和 `QRect.setBottom()` 方法会改变矩形的尺寸大小，因为它们可以修改矩形的宽度和高度。如果由我们自己处理按键事件，就需要调用 `update()` 来安排一个绘制事件，并需要设置 dirty 标志；否则，就可以由基类的实现代码来予以处理。

现在已看完了框的行为是如何实现的，下面是该把注意力转向框是如何绘制的了。在子类化 `QGraphicsItem` 时，必须至少要提供 `boundingRect()` 和 `paint()` 两个方法的实现。这一点，对于 `shape()` 的重新实现也是比较常见的，不过，这些示例会推迟到下一节中。

```

def boundingRect(self):
    return self.rect.adjusted(-2, -2, 2, 2)

```

表 12.2 中给出了 `QGraphicsItem` 的部分方法。

表 12.2 `QGraphicsItem` 的部分方法

语法	说明
<code>g.boundingRect()</code>	返回 <code>QGraphicsItem g</code> 的包围盒矩形 <code>QRect F</code> ；它的各个子类都应该重新实现这一方法
<code>g.collidesWithPath(pp)</code>	如果 <code>QGraphicsItem g</code> 与 <code>QPainterPath pp</code> 相碰撞，返回 <code>True</code>
<code>g.collidingItems()</code>	返回一个（也可能是空的） <code>QGraphicsItems</code> 的列表，其中是那些与 <code>QGraphicsItem g</code> 相碰撞的项
<code>g.contains(pt)</code>	如果 <code>QGraphicsItem g</code> 中包含 <code>QPointF pt</code> 就返回 <code>True</code>
<code>g.isObscured()</code>	如果 <code>QGraphicsItem g</code> 被它的碰撞项所遮盖，则返回 <code>True</code> ，会假设这些项的 z 值更大一些
<code>g.isSelected()</code>	如果选中了 <code>QGraphicsItem g</code> ，则返回 <code>True</code>
<code>g.itemChange(c, v)</code>	什么都不做；其子类可以重新实现这个方法，以便用来发现是否有变化发生过——比如，选择的状态或者是位置的改变。不要直接调用 <code>QGraphicsItem.setPos()</code> 或者间接调用这个方法
<code>g.moveBy(dx, dy)</code>	在场景坐标系的水平方向上把 <code>QGraphicsItem g</code> 移动 <code>dx</code> 的大小，在垂直方向上移动 <code>dy</code> 的大小

(续表)

语法	说明
g.paint(p, o)	什么都不做；其子类应当先重新实现这一方法，以便可以在 QPainter p 上重新绘制自己，可以带一个可选项参数 QStyleOptionGraphicsItem o；绘制是基于逻辑坐标系的，默认的中心点是(0,0)
g.pos()	返回 QGraphicsItem g 的位置，其类型是 QPointF。如果 g 是另一个 QGraphicsItem 的子项，该点就会表示成父项的局部逻辑坐标系；否则，就会采用场景坐标系。
g.resetMatrix()	将 QGraphicsItem g 的变换矩阵重置为单位矩阵；对于 PyQt 4.3 的用户来说，可将本方法替换成 resetTransform() 方法
g.rotate(a)	把 QGraphicsItem g 旋转 float a°
g.scale(x, y)	在水平方向和竖直方向上分别把 QGraphicsItem g 缩放 float x 和 float y；1.0 表示不缩放，0.5 表示大小缩为一半，3.0 表示大小为原来的三倍
g.scene()	返回 QGraphicsItem g 的 QGraphicsScene，或者在它尚未添加到场景中时返回 None
g.sceneBoundingRect()	以场景坐标系的形式返回 QGraphicsItem g 的 QRectF 型包围盒矩形——这个函数会处理相应的各类变换
g.setCursor(c)	把 g 的光标设置成 c、QCursor 或者 Qt.CursorShape
g.setEnabled(b)	根据 b 的值，把 g 设置起作用或者不起作用
g.setFocus()	将键盘的焦点设置到 QGraphicsItem g 上
g.setFlag(f)	将 g 设置成给定的 QGraphicsItem.ItemFlag f
g.setMatrix(m)	把 QGraphicsItem g 的矩阵设置成 QMatrix m；对于 PyQt 4.3 的用户来说，可以使带有 QTransform 参数的 setTransform()
g.setPos(x, y)	设置 QGraphicsItem g 的位置。如果 g 是另一个 QGraphicsItem 的子项，其位置就会基于父项的逻辑坐标系；否则，它就会采用场景的坐标系
g.setSelected(b)	根据 bool b 的值，把 QGraphicsItem g 设置成选中或者不选中
g.setZValue(z)	把 QGraphicsItem g 的 z 值设置成 float z。默认值是 0；那些具有较大 z 值的项会显示在那些 z 值较小的项之前。
g.shape()	以 QPainterPath 的形式返回 QGraphicsItem g 的形状。默认实现会调用 boundingRect 来确定要返回的绘制路径；对该方法进行重新实现通常会创建和返回确切的形状。默认情况下，碰撞检测是基于形状的
g.shear(x, y)	对 QGraphicsItem g 的坐标系分别在水平方向上错切 float x、在竖直方向错切 float y
g.translate(dx, dy)	对 QGraphicsItem g 的坐标系分别在水平方向上移动 int dx、在竖直方向上移动 int dy
g.update()	在可见的场景视图中给 QGraphicsItem g 安排一个绘制事件
g.zValue()	返回 QGraphicsItem g 的 z 值

这个方法本来应该返回该项的包围盒矩形，如果该项有轮廓线的话，还应该另外加上笔宽的一半。这里会做个弊，并让包围盒矩形稍稍大一些。这样做可以让用户更为轻松地点中该框，即使它们的高度或宽度已经减少成了 1 像素的直线了。

shape() 方法目前还没有得以实现，所以将会使用基类的 shape() 方法，从而可以基于包围盒矩形来生成形状。由于所给定的矩形要比实际情况下的矩形大一些，因而该形状也将更大一些。在做碰撞检测的时候就会用到这个形状，但这里它并不重要，因为在这个应用程序中并不会用到碰撞检测；尽管在下一章中会用到。

```
def paint(self, painter, option, widget):
    pen = QPen(self.style)
    pen.setColor(Qt.black)
    pen.setWidth(1)
    if option.state & QStyle.State_Selected:
        pen.setColor(Qt.blue)
    painter.setPen(pen)
    painter.drawRect(self.rect)
```

绘制边框非常简单。先用用户设置的线型创建一个画笔，笔的宽度固定为 1 个逻辑单元宽。如果选中了矩形，则更改画笔的颜色，然后设置画笔并绘制矩形。

使用图形视图类和绘制图形项往往要比重新实现绘制事件更容易些。这是因为每个项都有自己的 `paint()` 方法，并且因为这些项会使用中心为  $(0, 0)$  的局部逻辑坐标系，对于旋转来说，这一点就显得尤为方便了。

在这一节中，我们已经看到了预定义图形项和自定义项的用法，两者都提供了自定义行为，而且后一个 `BoxItem`，还做了一些自定义的绘制操作。在下一节的例子中，将会看到一些更为复杂的项绘制操作，还会看到碰撞检测和简单形式的动画。

## 12.2 动画和复杂形状

在上一节中，看到在图形视图应用程序中，是以用户交互为中心的。在本节中，将会看到一个完全不同类型的程序，会在其中模拟一种生物——“千足虫”的繁殖过程，并会基于一个图形项集来可视化这个生物种群中的每一个成员，如图 12.4 所示。每条千足虫都会有一个内部定时器(timer)。在每段时间间隙中，这些千足虫都在不断移动中，并且如果它们相互产生了碰撞，就会慢慢改变它们的颜色，直至最终消失。

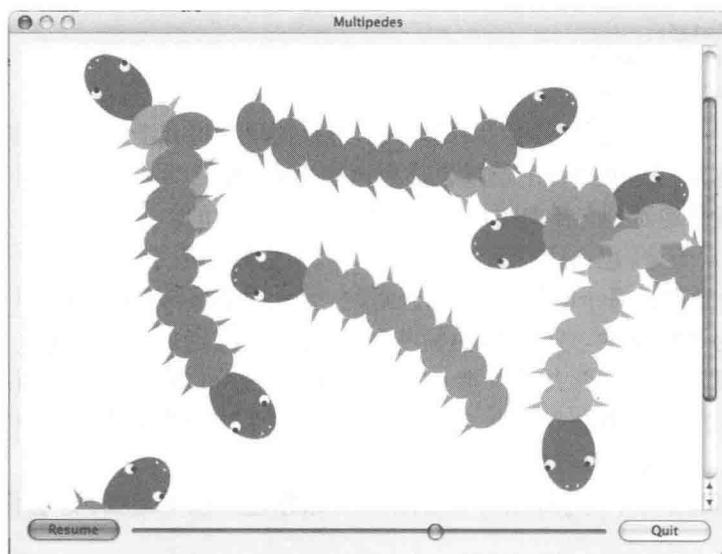


图 12.4 千足虫应用程序

这里会先从主窗体初始化程序中截取的部分代码开始。然后，将会查看窗体的 `populate()` 方法，这个方法用于千足虫的创建和定位。接下来，将会着眼于 Pause/Resume 按钮

的动作实现和缩放滑动条的实现。再接着，将会查看窗体的计时器事件，这是一个以前还从来没有使用过的事件处理器。一旦大致明白这个应用程序是如何工作的，就会看看那些用于可视化千足虫的图形子项的实现方法。

```
class MainForm(QDialog):
    def __init__(self, parent=None):
        super(MainForm, self).__init__(parent)
        self.scene = QGraphicsScene(self)
        self.scene.setSceneRect(0, 0, SCENESIZE, SCENESIZE)
        self.view = QGraphicsView()
        self.view.setRenderHint(QPainter.Antialiasing)
        self.view.setScene(self.scene)
        self.view.setFocusPolicy(Qt.NoFocus)
        zoomSlider = QSlider(Qt.Horizontal)
        zoomSlider.setRange(5, 200)
        zoomSlider.setValue(100)
        self.pauseButton = QPushButton("Pa&use")
        quitButton = QPushButton("&Quit")
```

这个窗体的一开始会创建一个图形场景。通常而言，对于那些非可见的 `QObject` 子类，可以给场景一个父项。`SCENESIZE` 是一个全局的、值为 500 的整数。对于视图的设置与在之前的例子中所看到的做法相似。缩放滑动条可以对场景进行放大或缩小。这里会将滑动条的初始值设置为 100(当 100% 时)，并将其范围设置为 5% 至 200%。暂停按钮 `Pause` 用于动画的暂停和恢复。

```
self.connect(zoomSlider, SIGNAL("valueChanged(int)"),
            self.zoom)
self.connect(self.pauseButton, SIGNAL("clicked()"),
            self.pauseOrResume)
self.connect(quitButton, SIGNAL("clicked()"), self.accept)
self.populate()
self.startTimer(INTERVAL)
self.setWindowTitle("Multipedes")
```

由于之前已经不止一次看到过布局方面的应用，所以这里会将其予以省略，加之这一次也并没有什么不一样的地方。这里的各个连接也都在情理之中，但之所以要在再次予以展示，主要是为了能够让我们清楚地看到，各个用户交互是如何进行处理的。

每个 `QObject` 子类(包括所有的 `QWidget`，因为它们都是 `QObject` 的子类)都可以触发一个计时器，进而导致在每个时间间隔内会触发一个计时器事件(timer event)。在这里的时间间隔 `INTERVAL` 为 200 毫秒。定时器的精准程度取决于底层的操作系统，但它应该至少为 20 毫秒，除非该机器已经负荷太过严重。在 `startTimer()` 方法中，会返回一个定时器 ID，如果打算要不止一次地调用这个方法，那么可以设置多个定时器，此时这个 ID 就会非常有了；这里会忽略它，因为只希望有一个定时器。

在初始化程序的最后，将会调用 `populate()` 来创建一些千足虫，还会像往常一样设置应用序的窗口标题。

```
def pauseOrResume(self):
    global Running
    Running = not Running
    self.pauseButton.setText("Pa&use" if Running else "Res&ume")
```

如果用户点击暂停按钮 `Pause`，那么就会将全局布尔变量 `Running` 设置成它原来状态的相反

值，并且随之更改按钮的标题内容。窗体的计时器和千足虫的定时器会引用这个变量的值，而如果它的值是 False，则什么都不做。

```
def zoom(self, value):
    factor = value / 100.0
    matrix = self.view.matrix()
    matrix.reset()
    matrix.scale(factor, factor)
    self.view.setMatrix(matrix)
```

要放大场景，只需要改变显示场景的视图的比例即可。通过获取该视图的当前变换矩阵，清空所有可能正在起作用的那些变换（也就是，缩放变换），然后再将它重新缩放到一个与滑动条的设置值成比例的因子数就可以了。

缩放操作对千足虫的绘制形式具有显著的影响。这是因为，在 `QGraphicsItem.paint()` 方法中，可以发现场景是如何放大和缩小的，也可以使用这一信息来确定到底该绘制多少细节。这就意味着，例如，如果场景缩小到用户无法分辨的细节时，就可以用更快、更简便的方式来绘制，而随着用户不断放大，则还可以不断增加绘制的细节。缩放的效果图如图 12.5 所示。

```
def populate(self):
    red, green, blue = 0, 150, 0
    for i in range(random.randint(6, 10)):
        angle = random.randint(0, 360)
        offset = random.randint(0, SCENESIZE // 2)
        half = SCENESIZE / 2
        x = half + (offset * math.sin(math.radians(angle)))
        y = half + (offset * math.cos(math.radians(angle)))
        color = QColor(red, green, blue)
        head = Head(color, angle, QPointF(x, y))
        color = QColor(red, green + random.randint(10, 60), blue)
        offset = 25
        segment = Segment(color, offset, head)
        for j in range(random.randint(3, 7)):
            offset += 25
            segment = Segment(color, offset, segment)
        head.rotate(random.randint(0, 360))
        self.scene.addItem(head)
global Running
Running = True
```

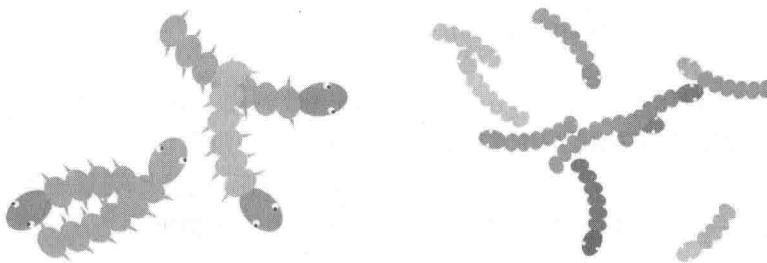


图 12.5 以两种不同缩放级别的形式来显示的千足虫应用程序

这一方法用来随机产生一个具有 6 ~ 10 条千足虫的种群。每条千足虫都由一个头、4 ~ 8 个身体节段组成。对于每一条千足虫来说，会先创建它的头部，颜色为半随机色，朝向也是随机的，位置则位于场景中心的圆内的某个随机位置上。然后，会创建千足虫的第一节段，并

以头部作为其父项。这就意味着，无论对头部应用何种类型的变换，例如，移动或旋转变换，这第一个节段都会随之应用到。接下来，再额外创建 3~7 个节段。每一个节段都会是其前一个节段的子项。这样做的效果是，如果头部被施加了变换，第一个节段也会被相应地施以变换，接下来还会对第一个节段的子段施以变换，以此类推，就会依次应用于千足虫的所有节段。

一旦创建了头和各个节段，就可以旋转头部并将其添加到场景中。向场景中自动添加一个图形项，就会递归地依次添加它的所有子项，所以只需在场景中添加千足虫的头，那么整条千足虫就被添加到场景中了。

最后，把全局 Boolean 变量 `Running` 设置为 `True`。此外，对于窗体的计时器，千足虫每个节段都有一个计时器，只要 `Running` 是 `True` 值，这个节段都会在每一个时间间隔内不停移动。

这里所用到的红色对于头项来说具有特殊意义。在起初创建的时候，所有千足虫的红色组件都会设置成 0。如果一条千足虫的头部的红色组件值达到了最大值（即 255）——就是用来作为碰撞的结果——该条千足虫将会“死亡”，也就是说，它就会被移除。这一剔除操作是在定时器事件中完成的。

```
def timerEvent(self, event):
    if not Running:
        return
    dead = set()
    items = self.scene.items()
    if len(items) == 0:
        self.populate()
        return
    heads = set()
    for item in items:
        if isinstance(item, Head):
            heads.add(item)
            if item.color.red() == 255:
                dead.add(item)
    if len(heads) == 1:
        dead = heads
    del heads
    while dead:
        item = dead.pop()
        self.scene.removeItem(item)
        del item
```

在每个时间间隔中，窗体的 `timerEvent()` 方法都会被调用。如果 Boolean 变量 `Running` 是 `False`，该方法则不执行任何操作并立即返回。如果场景中没有任何项（所有的千足虫都死亡了），就可以调用 `populate()` 来重新开始一个新局。否则，就遍历场景中的所有项，对两个数据集进行装配：一个是头项的红色组件值为 255 的那些项的数据集，另一个是场景中所有头项的数据集。

如果只有一个头项数据集，那么就用含有剩余头项的数据集来重写死亡数据集。这样可以确保如果只剩下一条千足虫，也可以将其杀死。然后，删除头数据集，以便可以不会剩下仍旧有效的任何引用。最后，遍历死亡项数据集，从场景中随机移除每一个项，并且因为所有权已经转移给了我们，所以会在移除的同时删除每一个项。归功于这种父子关系，当删除千足虫的头项时，头项的子项（第一个节段）也会被删除，并依次删除第一个节段的子项（第二个节

段), 等等, 直到最后一个孙子项, 这样一来, 只需删除千足虫的头项, 就可以删除千足虫的所有节段。

现在已经明白了应用程序是如何工作的, 所以就可以把注意力转向千足虫自身的代码实现了。正如在 `population()` 方法中所给出的那样, 千足虫是由一个头项以及至少四个节段组成的一——这两个类都是 `QGraphicsItem` 的子类, 也都可以足够智能地来绘制那些对于当前缩放细节下的绘制内容。这里将会先看一下头项 `Head`, 然后会看一下节段 `Segment`。

```
class Head(QGraphicsItem):
    Rect = QRectF(-30, -20, 60, 40)

    def __init__(self, color, angle, position):
        super(Head, self).__init__()
        self.color = color
        self.angle = angle
        self.setPos(position)
        self.timer = QTimer()
        QObject.connect(self.timer, SIGNAL("timeout()"), self.timeout)
        self.timer.start(INTERVAL)
```

所有的头项都有相同的形状: 一个放在 `static Rect` 中的椭圆。当对头项进行初始化时, 都会在各个实例变量中记录它们的颜色值和角度值, 并且会将其移动到场景中给定的位置处。

`QGraphicsItem` 类不是 `QObject` 子类, 也没有提供内置定时器。这并不是什么大问题, 因为可以像之前所做的那样, 简单地用一个 `QTimer` 来解决这一问题<sup>①</sup>。`QTimer` 的超时 (`timeout`) 不会触发定时器事件, 不过, 定时器还是会发射一个 `timeout()` 信号的。这里会创建一个计时器, 这个计时器的时间间隔 `INTERVAL` 是 200 毫秒, 也就是说, 每秒触发 5 次。这里已经把定时器的 `timeout()` 信号连接到了我们自己的 `timeout()` 方法上; 后面很快就会看到这个方法。

```
def boundingRect(self):
    return Head.Rect
```

这个包围盒矩形很简单——这只是一个静态的矩形, `static Rect`, 由其作为所有千足虫头项的基本形状。

```
def shape(self):
    path = QPainterPath()
    path.addEllipse(Head.Rect)
    return path
```

这个方法是用于碰撞检测的默认方法, 除非为该包围盒矩形指定了并不成熟的其他方法。绘图器路径是一系列的矩形、椭圆、圆弧以及其他各种形状(包括绘图器路径), 由它们合在一起完整地描述项的形状。在这个例子中, 路径 `path` 就只是一个椭圆。

为图形项的形状使用绘图器路径可以确保碰撞能够被准确地检测出来。例如, 两条千足虫的头可能会在包围盒矩形的角部相互穿插而却没有碰撞到一起, 因为构成它们头部的椭圆根本就不会占用矩形的角。

```
def paint(self, painter, option, widget=None):
    painter.setPen(Qt.NoPen)
    painter.setBrush(QBrush(self.color))
    painter.drawEllipse(Head.Rect)
```

<sup>①</sup> 对于 C++/Qt 编程开发人员来说, 他们可能会试着从 `QGraphicsItem` 和 `QObject` 做多重继承, 但 PyQt 则只允许从单一的 Qt 类进行间接继承。

```

if option.levelOfDetail > 0.5:
    painter.setBrush(QBrush(Qt.yellow)) # Outer eyes
    painter.drawEllipse(-12, -19, 8, 8)
    painter.drawEllipse(-12, 11, 8, 8)
    if option.levelOfDetail > 0.9:
        painter.setBrush(QBrush(Qt.darkBlue)) # Inner eyes
        painter.drawEllipse(-12, -19, 4, 4)
        painter.drawEllipse(-12, 11, 4, 4)
        if option.levelOfDetail > 1.3:
            painter.setBrush(QBrush(Qt.white)) # Nostrils
            painter.drawEllipse(-27, -5, 2, 2)
            painter.drawEllipse(-27, 3, 2, 2)

```

更进一步地说，头部其实就是一个椭圆，对于两只眼睛，每个眼睛又都是两个椭圆，一个在另一个的内部，并且还有两个微小的鼻孔，它们也都是椭圆。`paint()`方法一开始会去掉画笔并设置实线画刷来作为千足虫的颜色。接下来会绘制头部的基本形状。

这里的 `option` 变量的类型是 `QStyleOptionGraphicsItem`，它会包含各种有用信息，包括项的变换矩阵、字体规格、调色板和状态(选中与否、“开”、“关”以及其他许多状态)等。它也包含了“细节等级”(Level of Detail, LOD)，这是一种确定如何缩放场景的措施。如果场景根本就没有放大，细节等级就是 1.0；如果放大为原尺寸的两倍，那么细节等级就是 2.0，而如果缩小到原尺寸的一半大小，那么细节等级将是 0.5。

如果场景是以原有大小的 50% 或者是更大的尺寸，那么就把千足虫的眼睛黄色外圈画出来。可以通过手工硬编码的方式来编写坐标系，因为图形项会使用它们各自所在的逻辑坐标系，而所有的任何外部变换都会替我们自动处理。如果场景以原有大小的 90% 或者是更大的尺寸进行显示，则会绘制内嵌的眼睛，而如果场景继续放大到原有尺寸的 130% 或者更大尺寸，就还会画出千足虫的微小鼻孔。

必须考虑的最后一个方法是 `timeout()` 方法，这个方法在每一个 INTERVAL 间隔都会被计时器调用。这里将会分成两个部分来查看这个方法，因为它做的事情可以看成是两个方面。

```

def timeout(self):
    if not Running:
        return
    angle = self.angle
    while True:
        angle += random.randint(-9, 9)
        offset = random.randint(3, 15)
        x = self.x() + (offset * math.sin(math.radians(angle)))
        y = self.y() + (offset * math.cos(math.radians(angle)))
        if 0 <= x <= SCENESIZE and 0 <= y <= SCENESIZE:
            break
    self.angle = angle
    self.rotate(random.randint(-5, 5))
    self.setPos(QPointF(x, y))

```

如果全局的 Boolean 变量 `Running` 是 `False`，就什么也不做并立即返回。否则，可以对方向做一个小的随机变化量( $\pm 9^\circ$ )来计算出头的新位置，也可以为运动一个小的运动量(3 ~ 15 个逻辑单位)。为了避免千足虫游荡到场景之外，还要保证新的移动坐标位置( $x, y$ )始终是在场景的范围之内。

一旦有了新的坐标值，就要记录用过的角度，微微地旋转一下头部，再设置一下头部的位置。此时，因为移动的缘故，就可能会发生碰撞了。

```

for item in self.scene().collidingItems(self):
    if isinstance(item, Head):
        self.color.setRed(min(255, self.color.red() + 1))
    else:
        item.color.setBlue(min(255, item.color.blue() + 1))

```

要求场景中的所有项都要有检测碰撞的功能。如果一个头击中了另一个头，就让这个头更红一些，如果它撞到了一个节段，那么就让被撞的节段更蓝一些。如果一个头的红色组件的值达到了 255，就从场景中移除这个头部（并因此会导致整条千足虫，包括该千足虫的所有节段）。移除操作会在窗体的计时器事件中发生，正如前面所看到的那样。

现在，我们来看看节段的实现方法。它的初始化程序要比头的初始化程序的代码长一点，但也可以让 `boundingRect()`、`shape()` 以及 `paint()` 方法更简单一些。

```

class Segment(QGraphicsItem):
    def __init__(self, color, offset, parent):
        super(Segment, self).__init__(parent)
        self.color = color
        self.rect = QRectF(offset, -20, 30, 40)
        self.path = QPainterPath()
        self.path.addEllipse(self.rect)
    class Segment(QGraphicsItem):
        x = offset
        y = -20
        def __init__(self, color, offset, parent):
            self.path.addPolygon(QPolygonF([QPointF(x, y),
                QPointF(x - 5, y - 12), QPointF(x - 5, y)]))
            self.path.closeSubpath()
        y = 20
        self.path.addPolygon(QPolygonF([QPointF(x, y),
            QPointF(x - 5, y + 12), QPointF(x - 5, y)]))
        self.path.closeSubpath()
        self.change = 1
        self.angle = 0
        self.timer = QTimer()
        QObject.connect(self.timer, SIGNAL("timeout()"), self.timeout)
        self.timer.start(INTERVAL)

```

接受父项的参数并传给基类是需要注意的第一件事情。对于头项 `Head` 来说并没有这样做是因为，当一个项添加到场景中时，场景就会自动拥有该项的所有权，所以也就没有必要。但对于各个节段来说，之所以没有显式添加到场景中是因为它们都是其他项的子项。第一个节段的父项是千足虫的头，第二个节段的父项是第一个节段，第三个节段的父项是第二个节段，以此类推。当头部项添加到场景中时，则各个节段项也会添加进来；不过，场景却只有头部项的所有权。尽管本来也可以让各个节段都没有父项并将它们直接添加到场景中，但当前这种做法则会使对它们子项的管理更方便一些。尤其是，图形项之间的父-子关系是可以将变换从父项逐一扩展到子项的。

偏移量 `offset` 是一个相对于头项的偏移量 `x`，无论是哪个节段正在进行初始化。矩形用来绘制节段的椭圆，但与头项不一样的是，它不能包围整个形状，因为每个节段都有伸出来的腿。因为节段的形状并不简单，就可以使用绘制路径来创建它们。先从节段的“身体”开始，这是一个简单的椭圆。然后，绘制一条腿（这是一个非常平坦的三角形），之后再绘制另一条腿。`addPolygon()` 方法可以带一个 `QPolygonF()` 参数，它本身也可以是一个 `QPointF` 对象的列表。添加完每一条腿之后，调用 `closeSubpath()`；或者也可以在结束时简单地添加

一个额外的点，这是第一个点的一个副本。各个 change 和 angle 实例变量都是用于运动的；会在超时事件 `timeout()` 中讲述它们。

```
def boundingRect(self):
    return self.path.boundingRect()
```

包围盒矩形必须要考虑整个形状的大小，包括所有的腿，但使用 `QPainterPath.boundingRect()` 则可以很容易地获取到它的尺寸大小。

```
def shape(self):
    return self.path
```

尽管形状不是那么简单，但归功于在初始化程序中所计算出的路径，这个方法就很简单了。

```
def paint(self, painter, option, widget=None):
    painter.setPen(Qt.NoPen)
    painter.setBrush(QBrush(self.color))
    if option.levelOfDetail < 0.9:
        painter.drawEllipse(self.rect)
    else:
        painter.drawPath(self.path)
```

由于之前已经计算出了矩形和绘制器路径，`paint()` 方法会比之前所有用过的其他方法都要容易和快速。如果场景缩放到 90% 及以下，就只需要画一个椭圆；否则，就可以使用绘制器路径来绘制出形状的全部细节。

```
def timeout(self):
    if not Running:
        return
    matrix = self.matrix()
    matrix.reset()
    self.setMatrix(matrix)
    self.angle += self.change
    if self.angle > 5:
        self.change = -1
        self.angle -= 1
    elif self.angle < -5:
        self.change = 1
        self.angle += 1
    self.rotate(self.angle)
```

当千足虫的头部移动时，它的第一个（子项）节段就会随之而移动，而该节段的子节段则也会随之移动，等等。这样就很好了，但这意味着千足虫的形状是刚性的。要想让千足虫的各个节段能够慢慢地从一边摆动到另一边，于是，就需要让各个节段要拥有自己的计时器。

检索各个节段的变换矩阵，清空全部有效的变换矩阵（旋转变换等），然后再旋转该节段。`change` 变量从 1 开始，而旋转角 `angle` 则从 0° 开始。在每一个时间间隔内，都要把 `change` 加到角度 `angle` 上。如果角度 `angle` 达到了 6（或者 -6），就将其设为 5（或者 -5）并将值变为负值。这就意味着，`angle` 所具有的序列是 1, 2, 3, 4, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -4, -3, -2, -1, 0, 1, 2，以此类推，这样就可以产生出一种非常好的摆动效果。

这样就回顾完了复杂形状动画的内容。使用绘制器路径，可以创建出任意复杂的形状，通过把路径存储成静态路径或者是作为实例变量，可以一次性完成大量的计算而不需要每次都调用绘制方法。这里所用到动画实现方法可能并不是唯一的方法。例如，本来也是可以联合使用 `QGraphicsItemAnimation` 项和 `QTimeLine` 的。另外还有一种方法是用项自身的定时器超时而不是对它们进行引用。然后，可能会用到窗体中的计时器，并在每个间隔中，每个

项都应当是可以移动并且可以检测碰撞的，这都可以通过该窗体的超时处理程序来解决。没有唯一的解决方法，只有正确的解决方法；当然，最好的方法仍将取决于应用程序本身的需求。

## 小结

图形视图类对于要绘制很多单独的项的情况非常理想，这些项既可以是几十个，也可以是几十万个。当要允许用户与项进行交互时，图形视图类也很不错（例如，点击、拖动以及选择它们），对于绘制动画来说，也是非常理想的。

场景会使用自己的逻辑坐标系，也会包含一些图形项。场景的可视化可以使用 `QGraphicsView`，而如果打算让用户能够通过使用两个或多个不同的变换（例如，以不同的缩放等级或者旋转角度）来查看场景的话，就可以让多个视图与一个场景相关联。

图形视图类包括许多有用的“所见即所得”的预定义项。也可以对 `QGraphicsItem` 或者它的子类进行子类化操作来提供所需的自定义行为（例如，上下文菜单和按键事件处理等），也可以提供出自定义的绘制操作，以便可以绘制任何想要的形状。

如果想要把场景保存到文件或是从文件中加载场景，一种简单的方法是确保每个项都有一个变换矩阵，还要保存一个项的描述，该项在场景中的位置，项的矩阵，以及额外的那些与项相关的数据等可能都是必要的。使用 `QDataStream` 来实现这些要求则非常容易。

任何场景都可以用任意绘制设备绘出来，包括打印机、PDF 文件或者 `QImage`（例如，保存成.png 文件），这只需使用场景和视图类所提供的 `render()` 方法即可。而且从 Qt 4.3 开始，借助 `QSvgGenerator` 绘制设备类还可以将场景绘制成 SVG 格式。

图形视图项的绘制可以简单实现，因为其简便的逻辑坐标系使我们可以忽略外部的任何变换——比如，来自父项的那些变换。`QPainter` 类提供了很多便捷的绘制方法来绘制诸如圆弧、弦、椭圆、多边形、直线、折线、矩形、图像和文字等的事物。此外，还可以使用 `QPainterPath` 对象来创建出各种复杂的形状，并且这些东西都还可以直接使用 `QPainter.drawPath()` 来予以完成。

即使是更为复杂的形状，也可以通过组合两个或者多个项并使用相互的父-子关系而创建出来。这种父-子关系可以确保应用于父项的变换会自动应用到子项上，一直到最远的子孙项。利用这一点，并结合使用局部逻辑坐标系，使得做出复杂形状的动画也要比自己手动协调所有各类变换要容易得多。

可以使用图形视图类来完成模拟、游戏和时间序列数据的可视化。动画的一种简单方法是让每个项都有自己的定时器，并且在定时器一超时就可以移动它，尽管还可能会有一些其他方法。对于绘制来说，使用预先计算出的形状可以在绘制方法中节约大量时间，就像利用视图的细节等级（LOD）来决定要绘制多少细节一样。

对于作图，创建一个自定义窗口部件并重新实现它的绘制事件，就像在前一章中所描述的那样，可能会是最好的办法。对于一般科学和工程应用程序来说，PyQwt 库（用于科学技术应用程序的 Qt 窗口部件）会提供许多意料之外的功能，包括 2D 绘图，而 PyQwt3D 绑定则将 PyQwt 扩展到了 3D 绘图领域。要获得这些极好的附加组件，以用于进行快速数字处理，还会建议你安装 NumPy 包。更多信息可以参阅 <http://pyqwt.sourceforge.net> 和 <http://numpy.scipy.org>。

PyQt 的图形视图类的功能要比在这里所叙述的功能多得多。此外，还有许多由图形视图类所提供的功能，也可以使用 `QtOpenGL` 模块来实现 2D 和 3D 图形的绘制。同时，这一模块能够在与 PyQt 的其他图形类结合使用。例如，绘制可以使用 `QGLWidget` 而不是使用 `QWidget` 并调用 `QGraphicsView.setViewport(QGLWidget())`，也可以利用 `QPainter` 来重绘 `QGLWidget` 的绘制区域。

## 练习题

可以为页面设计师应用程序添加一个新的按钮来增强其功能，对齐(`Align`)，它可以有一个弹出菜单，其中含有左对齐(`Align Left`)、右对齐(`Align Right`)、顶部对齐(`Align Top`)和底端对齐(`Align Bottom`)。只用一个方法，`setAlignment()`，其中会带一个对齐参数。一定要确保实例变量与封装程序之间的关联，以免它们会被作为垃圾处理掉。要执行对齐操作，就必须至少要有两个项是选中的(因为每个项都会相对于彼此而相互对齐)。

答案中的算法分为两个阶段：第一个阶段是，找到要相对于其他项而对齐的项；例如，如果要执行左对齐，就需要先找到最左边的项。要注意的是，必须要以 `sceneBoundingRect()` 的形式，而不是 `boundingRect()` 的形式(如果该项旋转过，则会有所不同了)来实现这一功能。第二个阶段是，计算出要应用到其他项上的 `x` 或 `y` 的差值，然后将其应用一下。额外添加这个按钮及其菜单所需的代码不会超过 15 行代码，而 `setAlignment()` 应当可以在 45 行以内完成，所以整个过程大约可以用 60 行完成。

本练习题的参考答案在文件 `chap12/pagedesigner_ans.pyw` 中。

# 第 13 章 Rich 文本和打印

- Rich 文本的编辑
- 文档打印

PyQt 支持 Rich 文本(Rich text)。Rich 文本实际上就是 HTML 的一个子集，因而也可以对 CSS(层叠样式表，cascading style sheets) 提供部分支持<sup>①</sup>。这就意味着在工程实践中，就可以将含有 HTML 标记符(tag)的字符串传递给 PyQt 的许多文本处理类，并可以借助 PyQt 来在用户界面中进行 HTML 的适当渲染。

之前的章节中已经见过一些例子，就是把 HTML 传递给了 QLabel。QGraphicsTextItem 图形元素类也可以接受 HTML 内容。类 QTextBrowser 支持基本的 HTML 显示样式，包括超链接，尽管它还无法与完整的网页浏览器相提并论，但还是有很多开发人员都发现，对于显示一些帮助文本来说，这已经足够用了。对于 HTML 的编辑，PyQt 还提供了类 QTextEdit。尽管这个类可以对 PyQt 所支持的全部 HTML 标记符进行渲染，但它却没有为用户提供创建或者编辑某些标记符的方法——例如，它可以显示 HTML 表格，但却不支持对 HTML 表格的插入或者编辑操作。当然，这些缺陷也是可以通过子类化和提供自己的功能实现来予以修补的。

QTextEdit 的另一个用处是提供源代码的编辑功能，这可以通过自定义的 QSyntaxHighlighter 子类提供语法高亮功能。也可以借用一个专门用于源代码编辑的开源组件(component)。这个组件的名字叫做 Scintilla(参见 <http://www.scintilla.org>)，它里面有一个叫做 QScintilla 的 Qt 接口，也可以用于 PyQt 中。

可以处理 Rich 文本的所有窗口部件都会把文本存储在 QTextDocument 里面。也可以在实际应用中直接使用这个数据类。

在这一章，将会探索 QTextEdit 的一些特性，包括借助 QSyntaxHighlighter 的功能来将其做成一个源代码编辑器。尽管 PyQt 提供了一系列窗口部件来满足这里所需的大部分情况，既包括简单的标签和框架，又包括复杂的 Tab 标签页窗口部件、树形视图等，却并没有包含单行的 HTML 文本编辑窗口部件。为此，这里将会通过子类化 QTextEdit 来创建一个新的自定义窗口部件；这样将有利于加深对 QTextEdit 和 QTextDocument 的理解，并且在后续章节中，当需要在数据库型应用程序中为用户提供编辑单行 HTML 文本的功能时，它也会是一个非常有用的窗口部件。

13.2 节将会讲述有关打印的知识。之前已经看过如何将图片和图形场景(graphics scenes)打印到一个页面上，但在接下来的这一节将会看到如何打印多页面文档，包括嵌套了多张图片的多个文档。在文档打印方面，PyQt 提供了三种不同的技术：第一种是基于 HTML 的排版和打印，另一种是基于 QTextDocument 的创建和打印，还有一种是基于 QPainter 在页面上直接打印的技术。这里将会对这三种方法均加以讨论。

<sup>①</sup> 在 <http://doc.qt.io/qt-4.8/richtext-html-subset.html> 可以找到所支持的 HTML 标记符和 CSS 属性的列表。

## 13.1 Rich 文本的编辑

在这一节，将会从两个不同的视角来学习 Rich 文本的编辑。在 13.1.1 节，将会创建一个纯文本的编辑器，它使用 `QSyntaxHighlighter` 类来提供语法高亮功能。在 13.1.2 节，将会创建一个单行的 Rich 文本编辑器，它与 `QLineEdit` 类似，但会有右键弹出菜单，也可以支持多个格式化快捷键，如使用快捷键 `Ctrl + B` 和 `Ctrl + I` 分别来切换字体的加粗和斜体效果。在这两个例子中，都将基于 `QTextEdit` 来构建程序。

### 13.1.1 使用 `QSyntaxHighlighter`

如果打算让一个文本编辑器能够理解 Python 并对其进行语法高亮显示，并不需要由我们自己从头创建它。基于 Tkinter 的 IDLE 应用程序既提供了测试 Python 代码的“沙箱”，又可以认为是一个完美的 Python 代码编辑器。而如果需要更为强大的功能，还可以使用 Eric4，它本身就是用 PyQt 写成的，并且还使用 `QScintilla` 来提供文本编辑功能。然而，所有这些现成的编辑器还都无法完美提供我们所需的全部功能，并且在创建一个自己的 Python 编辑器的过程中将会较好地启发和揭示其中所蕴含的东西，所以这里会写一个简单的 Python 代码编辑器，以便能够学到如何使用 `QTextEdit` 和 `QSyntaxHighlighter`。如图 13.1 所示，这是一个简单的主窗口风格的应用程序，带有两个菜单和两个工具栏。由于之前已经看到过创建这种应用程序的类似结构，所以这里将会重点关注与 Rich 文本编辑相关的那些部分，并会先从主窗口的初始化程序开始。

```
class MainWindow(QMainWindow):
    def __init__(self, filename=None, parent=None):
        super(MainWindow, self).__init__(parent)
        font = QFont("Courier", 11)
        font.setFixedPitch(True)
        self.editor = QTextEdit()
        self.editor.setFont(font)
        self.highlighter = PythonHighlighter(self.editor.document())
        self.setCentralWidget(self.editor)
```

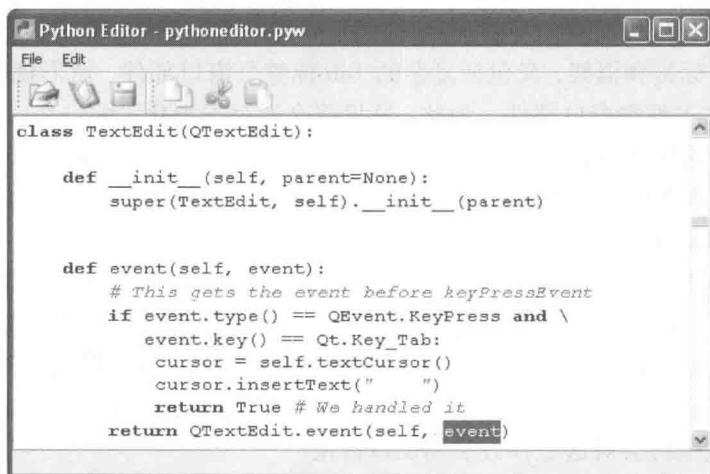


图 13.1 用 Python 编辑器编辑自身

先来创建一个固定宽度的字体(这种字体里的每个字符都具有相同的宽度)。然后, 创建一个 `TextEdit` 类, 这是 `QTextEdit` 的一个子类, 但不同的是, 它会把 Tab 键转化为四个空格。接下来, 创建一个 `PythonHighlighter` 类, 这是 `QSyntaxHighlighter` 的一个子类, 会将文本编辑器的 `QTextDocument` 传递给它——编辑器的文本内容和格式实际都是存储在这个 `QTextDocument` 对象里的。最后把该编辑器设置成主窗口的中央窗口部件。

初始化程序的剩余部分主要用来创建动作、菜单和工具栏, 这些内容都已经比较熟悉了, 所以这里会简单跳过。唯一是要看的那些与文件处理相关的方法, 因为它们中含有文本编辑器的相关内容。

```
def fileNew(self):
    if not self.okToContinue():
        return
    document = self.editor.document()
    document.clear()
    document.setModified(False)
    self.filename = None
    self.setWindowTitle("Python Editor - Unnamed")
    self.updateUi()
```

这个方法只是用来获取并清空 `QTextDocument` 对象里的文本内容, 并且会把改动标记设置为 `False`。结果是一个空白的、不再有未保存修改的 `QTextEdit`。`updateUi()` 方法会根据应用程序的状态来启用或者禁用某些动作; 有关详细内容可以参阅下页中“动作的启用与禁用”的内容。

```
def loadFile(self):
    fh = None
    try:
        fh = QFile(self.filename)
        if not fh.open(QIODevice.ReadOnly):
            raise IOError, unicode(fh.errorString())
        stream = QTextStream(fh)
        stream.setCodec("UTF-8")
        self.editor.setPlainText(stream.readAll())
        self.editor.document().setModified(False)
        self.setWindowTitle("Python Editor - %s" % \
            QFileInfo(self.filename).fileName())
    except (IOError, OSError), e:
        QMessageBox.warning(self, "Python Editor -- Load Error",
            "Failed to load %s: %s" % (self.filename, e))
    finally:
        if fh is not None:
            fh.close()
```

如果已经加载文件(例如, 用户通过调用 `File→Open` 动作)就会调用这个方法。与文件处理相关的代码和之前看到过的代码一样; 唯一的区别就是, 会将 `QTextDocument` 的修改标记设置为 `False`。

用于保存的代码(这里没有给出来)与之前用来加载的代码很相似。即得到文件句柄, 创建 `QTextStream` 对象, 将编码格式设置成 UTF-8, 再用 `QTextEdit.toPlainText()` 方法把所有文本都输出来。同时, 还同样会将 `QTextDocument` 的改动标记设置为 `False`。

```
class TextEdit(QTextEdit):
    def __init__(self, parent=None):
```

```

super(TextEdit, self).__init__(parent)

def event(self, event):
    if event.type() == QEvent.KeyPress and \
        event.key() == Qt.Key_Tab:
        cursor = self.textCursor()
        cursor.insertText("    ")
        return True
    return QTextEdit.event(self, event)

```

上面给出了 `TextEdit` 子类的完整代码。每一个 `QTextDocument` 都可以通过 `QTextCursor` 对象进行修改，这在程序语义上与用户通过键盘和鼠标修改文档具有相同的效果。

### 动作的启用与禁用

有时，一些特定的动作只适用于某些特定的条件。例如，如果没有修改过文档，就不应该允许对文档实施“File→Save”的动作，还有，对于一个空白文档就需要根据情况来决定是否可以对其应用“File→Save as”的动作。类似地，如果没有选中任何文本，那么某种意义上就既不能让复制“Edit→Copy”起作用，也不能让剪切“Edit→Cut”起作用。一种处理方法是一次性地把所有这些动作都设为可用，但对于那些不起作用的情况，这些动作就什么都不做；例如，如果任何文本都没有选中，那么对 `QTextEdit.cut()` 的调用，从安全的角度来说就什么也别做。

另一种解决方法是，根据应用程序的状态来启用或者禁用某些动作。这可以通过三步来实现这一点：首先，让这些将要被启用或者禁用的动作都成为实例变量，以便可以在初始化程序的外面仍然可以访问这些动作；其次，创建一个可以根据应用程序的运行状态来启用或者禁用这些动作的方法（例如，`updateUi()`）；最后，让应用程序的某些的信号-槽连接能够加到 `updateUI()` 里，以便可以在应用程序的状态一发生变化就可以调用它们。

以 Python 编辑器为例，其中需要的连接有这些：

```

self.connect(self.editor,
             SIGNAL("selectionChanged()"), self.updateUi)
self.connect(self.editor.document(),
             SIGNAL("modificationChanged(bool)"), self.updateUi)
self.connect(QApplication.clipboard(),
             SIGNAL("dataChanged()"), self.updateUi)

```

这些连接也就意味着，如果编辑器的选取状态发生变化，或者文档修改过，或者剪贴板里的数据发生了变化，都可以禁用或者启用相应的那些动作。

```

def updateUi(self, arg=None):
    self.fileSaveAction.setEnabled(
        self.editor.document().isModified())
    self.fileSaveAsAction.setEnabled(
        not self.editor.document().isEmpty())
    enable = self.editor.textCursor().hasSelection()
    self.editCopyAction.setEnabled(enable)
    self.editCutAction.setEnabled(enable)
    self.editPasteAction.setEnabled(self.editor.canPaste())

```

这个方法会对之前的那些信号-槽连接进行响应。它也会在初始化程序的最后被明确地调用，以便可以对用户界面的初始状态进行设置，同时还会在 `fileNew()` 方法的结尾处也有调用。

QTextEdit.canPaste() 方法是在 Qt 4.2 中引入的；对于之前的版本，可以使用 not QApplication.clipboard().text().isEmpty()。在这一章的后续部分，将会介绍有关文本光标和文本文档的一些类。

event() 处理程序会在任何与鼠标和键盘相关的事件处理程序之前得到调用，这也是唯一一个可以对 Tab 键按下进行拦截和处理的地方。如果用户按下 Tab 键，就可以从 QTextEdit 那里获得一个 QTextCursor；这就允许以编程的方式对保存着文本的底层 QTextDocument 进行交互。默认情况下，文本编辑器所返回的文本光标会位于当前的文本插入点（也称为光标位置），所以只需要简单插入 4 个空格就行了。之后，返回 True 来通知事件处理系统，这个 Tab 键按下事件已经得到了处理，也就没有其他更多动作（比如改变焦点等）了。

为了能够提供语法高亮功能就必须创建一个 QSyntaxHighlighter 子类，重新实现 highlightBlock() 方法，用 QTextDocument 作为参数，为这个语法高亮创建一个实例。最后一部分会在 MainWindow 的初始化程序中完成，所以现在会先来看一看 QSyntaxHighlighter 子类。

QSyntaxHighlighter 是以单行的方式工作的，也为跟踪多行文本的状态提供了一种简单方式。对于这里的 Python 编辑器来说，将会使用正则表达式来识别 Python 的关键字、注释和字符串，包括含有多行的三引号字符串，因此就可以对这些内容加上高亮效果。在这个子类的初始化程序中，先会设置这些正则表达式，这些内容将会从两个部分来进行查看。

```
class PythonHighlighter(QSyntaxHighlighter):  
    Rules = []  
  
    def __init__(self, parent=None):  
        super(PythonHighlighter, self).__init__(parent)  
        keywordFormat = QTextCharFormat()  
        keywordFormat.setForeground(Qt.darkBlue)  
        keywordFormat.setFontWeight(QFont.Bold)  
        for pattern in ((r"\band\b", r"\bas\b", r"\bassert\b",  
                         ...  
                         r"\byield\b")):  
            PythonHighlighter.Rules.append((QRegExp(pattern),  
                                         keywordFormat))
```

Rules 静态列表保存着一些数据对。每对数据的第一个元素是一个正则表达式（就是一个 QRegExp），用来匹配一个只占一行文本的 Python 语法结构（如一个 Python 关键字）。第二个元素是一个 QTextCharFormat，这是一个保存着用来说明文本格式化方式信息内容的对象，比如这些文本字体以及用来绘制这些文本时所用到的画笔等。

这里创建了用于 Python 关键字的规则，假定每个规则都会使用相同的 keywordFormat 格式（大部分的关键字都没有在这些程序片段中给出，而是用省略号…来表示它们的）。每个关键字都有一个 \b 作为其前缀和后缀；这是一个正则表达式符号，它不会匹配任何的文本，但只会用来匹配单词的边界。这就意味着，例如，在表达式 a and b 中，and 就会被识别为关键字，而在表达式 a = sand 中，sand 中的 and 就不会被（正确地）识别成关键字。

```
commentFormat = QTextCharFormat()  
commentFormat.setForeground(QColor(0, 127, 0))  
commentFormat.setFontItalic(True)
```

```

PythonHighlighter.Rules.append((QRegExp(r"#+.*"),
                                commentFormat))
self.stringFormat = QTextCharFormat()
self.stringFormat.setForeground(Qt.darkYellow)
stringRe = QRegExp(r"""(?:(^|[^']*'|[^"]*")|(?!""))""")
stringRe.setMinimal(True)
PythonHighlighter.Rules.append((stringRe, self.stringFormat))
self.stringRe = QRegExp(r"""(?:"".*""|''.*'')""")
self.stringRe.setMinimal(True)
PythonHighlighter.Rules.append((self.stringRe,
                                self.stringFormat))
self.tripleSingleRe = QRegExp(r"""\n+(?!"")""")
self.tripleDoubleRe = QRegExp(r'''\n+(?!"')''')

```

在这些关键字之后，会为 Python 的注释创建一个格式和一个正则表达式。这里的正则表达式并不完美；比如，它就不能正确地处理引用到的#字符。

对于字符串，会让字符串格式保存成一个实例变量，因为在 `highlightBlock()` 方法中，当处理多行字符串时，将会需要用到它。对于单行字符串的处理就可以使用在初始化程序中所构建的简单(且快速)的正则表达式，这些表达式也会被加入到 `Rules` 列表中。最后，还会再创建两个正则表达式。这两个表达式都使用了负向先行断言(negative lookahead)；例如，`(?!")`的意思是，“后面不要再有””。它们会在 `highlightBlock()` 方法中使用到，在那里将会分成两个部分来查看接下来的代码。

```

def highlightBlock(self, text):
    NORMAL, TRIPLESINGLE, TRIPLEDOUBLE = range(3)

    for regex, format in PythonHighlighter.Rules:
        i = text.indexOf(regex)
        while i >= 0:
            length = regex.matchedLength()
            self.setFormat(i, length, format)
            i = text.indexOf(regex, i + length)

```

对于 `QString text` 参数中要显示的每一行文本，都会调用 `highlightBlock()` 方法。

一开始会设置三个可能的状态：正常、三单引号引导的字符串、三双引号引导的字符串。然后，将会遍历每一条规则，并且一旦发现与该规则的正则表达式相匹配，就把文本的格式修改为与正则表达式所匹配的长度相对应的格式。正则表达式与格式对所构成的列表和之前的代码中所给出的 `for` 循环相结合，足以对每个语法成分只占用一行且每个都可以用一个正则表达式来表示的所有语法进行高亮显示。

```

self.setCurrentBlockState(NORMAL)
if text.indexOf(self.stringRe) != -1:
    return
for i, state in ((text.indexOf(self.tripleSingleRe),
                  TRIPLESINGLE),
                 (text.indexOf(self.tripleDoubleRe),
                  TRIPLEDOUBLE)):
    if self.previousBlockState() == state:
        if i == -1:
            i = text.length()
        self.setCurrentBlockState(state)
        self.setFormat(0, i + 3, self.stringFormat)
    elif i > -1:
        self.setCurrentBlockState(state)
        self.setFormat(i, text.length(), self.stringFormat)

```

接下来，会将当前块的状态设置为正常状态。这个状态是一个 `QSyntaxHighlighter` 与当前行关联起来的选择的整数值。然后，会测试是否有一个完整的三引号括起的单行字符串，也就是，会在该行的开头和结尾都有一个三引号；如果有，就对其进行格式化，这样就可以完成并返回了。

在 PyQt 的各个文本处理类中，比如 `QTextDocument`、`QTextEdit` 和 `QSyntaxHighlighter`，一个文本片段通常可以认为是由换行符所分隔开的连续字符串。对于字处理器类型的文档来说，这就相当于是一个段落（因为文本处理类就可以自动换行），不过对于手工插入换行符的源代码来说，此时的每个段落都完全相当于代码中的一行。

现在还有三种情况需要处理。要么是在一个三引号引起的字符串中，它开始于前一行而结尾部分又不在这一行；或者，三引号的开始或者结尾部分在这一行。

如果前面一行是在一个三引号引导的字符串中且在这一行又没有三引号，那么整个这一行就仍然是在这个三引号字符串中，因而只需要把当前块的状态设置成与上一行的状态相同、把整个这一行的格式都设置成三引号的格式。如果上一行的状态是三引号引导的状态且这里有一个三引号，那么这个三引号必定是一个封闭引号，因而会更新三引号的状态并包含这个三引号。在这个例子中，这里会将状态仍旧保留为正常状态，因为将会从三引号引导字符串的结尾开始向前应用该状态。另一方面，如果前一行的状态不是三引号且找到了多个三引号，就可以把状态设置成三引号状态并从这些三引号开始一直格式化到这一行的结尾。

这样，到这里就结束了语法高亮的例子。显然，还可以使用更为复杂的正则表达式，甚至也可以完全不用正则表达式而是用有限状态机（finite state automaton）或者是用语法分析程序（parser）来识别每行源代码各个部分所需的特定格式。对于有复杂语法的大文本文档来说，语法高亮很消耗计算资源，不过 `QSyntaxHighlighter` 可以简化这些工作，只需要很短的一些格式化语句就能为文本提供正确的语法高亮效果。

### 13.1.2 Rich 文本的行编辑

在某些应用程序中，要求用户能够输入单行的 Rich 文本内容。比如，一个数据库程序中可能有一个“描述”（description）域，用户可以在这里根据需要把一些特定的字词格式化为加粗、斜体或者带上颜色。在第 14 章和第 16 章，将会见到一些这样的例子。但遗憾的是，PyQt 并没有提供这样的窗口部件。在这一小节，将会创建一个 `RichTextLineEdit`，这是 `QTextEdit` 的一个子类，如图 13.2 所示，它可以提供之前所需的功能。在接下来的过程中，将会学习到该如何通过程序方式来格式化 `QTextEdit` 中的文本，以及如何对 `QTextEdit` 中的 `QTextDocument` 进行遍历，来提取出里面的文本及其格式。

Rich 文本行编辑将能够支持大部分常见类型且用于单行文本上的文本格式化指令：加粗、斜体、下画线、删除线、上标和下标。除此之外，还可以支持三种字体样式——等宽的 monospaced、有衬线的 sans serif 和无衬线的 serif 字体——还具有将文本设置成一定范围颜色的功能。这里将先从带有一些静态常量的初始化程序开始。

```
class RichTextLineEdit(QTextEdit):
    (Bold, Italic, Underline, StrikeOut, Monospaced, Sans, Serif,
     NoSuperOrSubscript, Subscript, Superscript) = range(10)

    def __init__(self, parent=None):
```

The  $e = mc^2$  formula for dummies

图 13.2

```

super(RichTextLineEdit, self).__init__(parent)
self.monofamily = QString("courier")
self.sansfamily = QString("helvetica")
self.seriffamily = QString("times")
self.setLineWrapMode(QTextEdit.NoWrap)
self.setTabChangesFocus(True)
self.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
self.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
fm = QFontMetrics(self.font())
h = int(fm.height()) * (1.4 if platform.system() == "Windows" \
else 1.2))
self.setMinimumHeight(h)
self.setMaximumHeight(int(h * 1.2))
self.setToolTip("Press <b>Ctrl+M</b> for the text effects " \
"menu and <b>Ctrl+K</b> for the color menu")

```

先从设置一些默认字体类型开始。如今，每个平台都可以提供 Courier、Helvetica 和 Times 字体(或者提供这些字体的其他别名)。还要确保 Tab 可以造成焦点的改变而不是只用来插入一个 Tab 字符。通过计算文字的最小和最大高度将有助于实现那些尺寸大小提示的方法，不过，对于跨平台的字体系统来说会显得稍微有些麻烦。窗口部件也可以给出一些提示来让用户知道还有哪些特殊键是可以使用的。

值得注意的是，条件表达式会用括号括起来。在这个例子中，这一点很重要，因为在非 Windows 平台上，如果没有这些括号，h 的值就总会设置成 1 而不是所期待的 20 或 30。

```

def sizeHint(self):
    return QSize(self.document().idealWidth() + 5,
                self.maximumHeight())

```

最佳尺寸应当是文本的“理想”宽度(可以把字体的尺寸大小和诸如粗体及斜体这样的属性都考虑进去)再加上一点外边框和文本的最大高度。QTextDocument.idealWidth() 方法是在 Qt 4.2 中引入的。

```

def minimumSizeHint(self):
    fm = QFontMetrics(self.font())
    return QSize(fm.width("WWWW"), self.minimumHeight())

```

对于最小尺寸大小来说，这里用 4 个大写 W 字符的宽度来作为该窗口部件的默认字体宽度。当然，也可以只用一个任意值，比如说，40 像素。

RichTextLineEdit 与 QLineEdit 相比的不同之处在于，用户具有改变文字和字符的格式与颜色的能力。为了支持文字格式和颜色设置功能，必须提供一些方式让用户可以应用这些修改。这可以通过提供一个文字效果菜单和颜色菜单来实现，也可以通过一些按键顺序来实现格式化。这两个菜单都可以通过一些特定组合键来调用，并且在有上下文菜单事件触发时，也可以弹出文本效果菜单。

```

def contextMenuEvent(self, event):
    self.textEffectMenu()

```

在某些平台上，当用户点击了鼠标右键或者按下了某个特定的键或者是其他的组合键时，就会出现上下文菜单事件。在这个例子中，只是调用 textEffectMenu() 方法，它会弹出一个适当的菜单。默认情况下，QTextEdit 有它自己的上下文菜单，但通过重新实现这个上下文菜单事件处理程序，可以先让自己的代码起作用。

```
def keyPressEvent(self, event):
    if event.modifiers() & Qt.ControlModifier:
        handled = False
        if event.key() == Qt.Key_B:
            self.toggleBold()
            handled = True
        elif event.key() == Qt.Key_I:
            self.toggleItalic()
            handled = True
        elif event.key() == Qt.Key_K:
            self.colorMenu()
            handled = True
        elif event.key() == Qt.Key_M:
            self.textEffectMenu()
            handled = True
        elif event.key() == Qt.Key_U:
            self.toggleUnderline()
            handled = True
        if handled:
            event.accept()
            return
    if event.key() in (Qt.Key_Enter, Qt.Key_Return):
        self.emit(SIGNAL("returnPressed()"))
        event.accept()
    else:
        QTextEdit.keyPressEvent(self, event)
```

由于用户正在录入文字，所以就有必要提供一个修改文本格式的键盘接口。这里将  $\text{Ctrl} + \text{B}$  设置为粗体字的切换开关，将  $\text{Ctrl} + \text{I}$  设置成斜体字的切换开关，并将  $\text{Ctrl} + \text{U}$  设置成下画线字体的切换开关。另外， $\text{Ctrl} + \text{K}$  可以调出颜色菜单，而  $\text{Ctrl} + \text{M}$  则可以调出文字特效菜单(除此之外，还可以用上下文菜单事件进行调用)。通过对自己要处理的按键事件调用 `accept()` 方法，可以通知事件处理系统：这个按键事件已被处理无须再有其他更多按键事件了。

如果用户按回车键，就可以发出一个 `returnPressed()` 信号，因为这很有用；也不会在文本中插入换行符。任何其他键盘事件都会传给它的基类进行处理。`QTextEdit` 类还支持它自己的快捷键——比如， $\text{Ctrl} + \text{左箭头}$  每次可以向左移动一个单词， $\text{Ctrl} + \text{Del}$  每次可以向右删除一个单词，而  $\text{Ctrl} + \text{C}$  可以将任何已经选中的文本复制到剪贴板上，当然，那些简单的字符，比如，`a`、`b`、`Shift + A` 和 `Shift + B` 等，也都可以逐字符插入进来。

```
def toggleBold(self):
    self.setFontWeight(QFont.Normal \
        if self.fontWeight() > QFont.Normal else QFont.Bold)
```

PyQt 可以支持数种字体加粗方式，不过这里只是选用了一种简单的打开、关闭的方法。`QTextEdit.fontWeight()` 方法可以返回当前插入点字体的高度，与之相似，`setFontWeight()` 方法可用来对当前插入点或者选中文本的字体粗细进行设置。至少在 Linux 平台上，`QTextEdit` 在格式化时是很智能的。例如，当有选中的文本要进行格式化时，就只去格式化这些选中的文本；如果没有选中任何文本，且插入点处在了文本的最后，那么就会将格式从该点开始向前应用，而如果没有选中文本且插入点在一个文字的中间，那么就会将格式应用于整个单词。

```
def toggleItalic(self):
    self.setFontItalic(not self.fontItalic())

def toggleUnderline(self):
    self.setFontUnderline(not self.fontUnderline())
```

无论是斜体还是下画线都只是简单的开启(on)/关闭(off)设置方式，而在 Linux 平台上，这些工作的切换方式与字体的加粗或者其他格式化方法是一样的。

```
def colorMenu(self):
    pixmap = QPixmap(22, 22)
    menu = QMenu("Colour")
    for text, color in (("&Black", Qt.black), ("B&lue", Qt.blue),
                        ("Dark Bl&ue", Qt.darkBlue), ("&Cyan", Qt.cyan),
                        ("Dar&k Cyan", Qt.darkCyan), ("&Green", Qt.green),
                        ("Dark Gr&een", Qt.darkGreen),
                        ("M&agenta", Qt.magenta),
                        ("Dark Mage&nta", Qt.darkMagenta),
                        ("&Red", Qt.red), ("&Dark Red", Qt.darkRed)):
        color = QColor(color)
        pixmap.fill(color)
        action = menu.addAction(QIcon(pixmap), text, self.setColor)
        action.setData(QVariant(color))
    self.ensureCursorVisible()
    menu.exec_(self.viewport().mapToGlobal(
        self.cursorRect().center())))

```

通过 `Ctrl + K` 可以调用颜色菜单。要创建这样的菜单，可以对文本和颜色常量进行遍历，然后将各个值都依次加入到菜单中。将每个动作的数据依次设置成它所对应的颜色；然后就可以在 `setColor()` 中使用这个颜色了。

希望在光标的当前插入位置能够弹出该菜单，也就是说，就是在文本光标的位置。这并不容易，因为 `RichTextLineEdit` 的宽度可能并足以显示它所包含的全部文本，这样一来，当前插入点的位置可能就不会在当前的可见区域。

要解决这一问题需要做两件事。第一，调用基类的 `QTextEdit.ensureCursorVisible()` 方法；这样它就会根据需要水平滚动编辑器，从而把光标置于可见区域中——而如果当前插入点已经可见，那么就什么也不做。第二，在插入点所在的矩形位置中心处弹出菜单。`cursorRect()` 方法可以返回一个相对于窗口部件坐标系的 `QRect`，所以就必须相应地把从 `QRect.center()` 得到的 `QPoint` 的坐标进行坐标转换。通过调用 `viewport()` 可以完成这一工作，它可以有效地返回一个具有可见区域的窗口部件的精确尺寸大小，这样也就知道要显示的是编辑器的那个区域。然后，可以使用该视口窗口部件的 `mapToGlobal()` 方法把该点的坐标从窗口部件坐标系转换成全局(相对于计算机屏幕)坐标系，这个全局坐标系可由 `QMenu.exec_()` 使用以用来定位自己。

```
def setColor(self):
    action = self.sender()
    if action is not None and isinstance(action, QAction):
        color = QColor(action.data())
        if color.isValid():
            self.setTextColor(color)
```

如果用户选中一个颜色就会调用 `setColor()` 方法。可以从调用动作的用户数据里获取这个颜色值并将其应用到选中的文本上。同理，也会将这相同的处理逻辑用于选中文字的加粗、斜体及下画线中，或者是当前文字，又或者是从文字的最后向前应用。

```
def textEffectMenu(self):
    format = self.currentCharFormat()
    menu = QMenu("Text Effect")
    for text, shortcut, data, checked in (
```

```

        ("&Bold", "Ctrl+B", RichTextEdit.Bold,
         self.fontWeight() > QFont.Normal),
        ("&Italic", "Ctrl+I", RichTextEdit.Italic,
         self.fontItalic()),
        ...
        ("Subs&cript", None, RichTextEdit.Subscript,
         format.verticalAlignment() == \
         QTextCharFormat.AlignSubScript)):
    action = menu.addAction(text, self.setTextEffect)
    if shortcut is not None:
        action.setShortcut(QKeySequence(shortcut))
    action.setData(QVariant(data))
    action.setCheckable(True)
    action.setChecked(checked)
    self.ensureCursorVisible()
    menu.exec_(self.viewport().mapToGlobal(
        self.cursorRect().center())))

```

文字效果菜单方法可由  $\text{Ctrl} + \text{M}$  或者上下文菜单事件触发，它在结构上与之前看到的颜色菜单方法很相似，但貌似要稍微复杂些。先从获取当前要强制应用的文本格式开始，因为需要根据这些信息来决定是否选中菜单中的那些不同选项。然后，利用四元值对（文本、快捷键、常量、选中状态）生成一个菜单。该菜单的最终效果如图 13.3 所示。

由于对某些动作已经设置了快捷键——例如， $\text{Ctrl} + \text{B}$  是加粗的快捷键。这里没有使用那些标准的组合键是因为在之前所看到的通过手工编码方式实现的键盘键击事件处理程序中已经用过了这些标准化的组合件，所以必须确保这些按键要能够让相应的处理程序对其予以匹配。

实际上，这些快捷键还没有起到相应的效果，因为它们只存在于它们所关联的那些动作的有效期内，且仅仅只有该菜单存在时才会存在这些动作，也就是说，只存在于 `textEffectMenu()` 方法的存在期。但这并没有什么关系，因为这里已经重新实现了键盘事件处理程序并是由我们自己来提供这些快捷键的。菜单中的这些快捷键只对诸如主窗口菜单栏中的那些常规菜单才有效。那么，为什么还要费尽周折地对菜单中的那些快捷键进行处理？因为把这些快捷键添加到菜单中会让它们更清晰，也会方便用户了解到这些快捷键。这样做并没有解决如何才能让用户首先找到  $\text{Ctrl} + \text{M}$  和  $\text{Ctrl} + \text{K}$  菜单的问题，但只能是希望用户能够看到工具栏提示或者是在应用程序的用户手册中发现它们。

这个菜单的创建和弹出方式与颜色菜单都几乎完全相同。如果用户点击了任何一个文本效果选项，都会调用 `setTextEffect()` 方法。这里将会用两个部分来查看这个方法的内容。

```

def setTextEffect(self):
    action = self.sender()
    if action is not None and isinstance(action, QAction):
        what = action.data().toInt()[0]
        if what == RichTextEdit.Bold:
            self.toggleBold()
            return

```

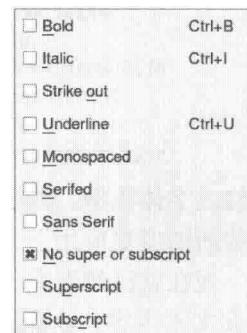


图 13.3 文字效果菜单

```

if what == RichTextLineEdit.Italic:
    self.toggleItalic()
    return
if what == RichTextLineEdit.Underline:
    self.toggleUnderline()
    return

format = self.currentCharFormat()
if what == RichTextLineEdit.Monospaced:
    format.setFontFamily(self.monofamily)
elif what == RichTextLineEdit.Serif:
    format.setFontFamily(self.seriffamily)
elif what == RichTextLineEdit.Sans:
    format.setFontFamily(self.sansfamily)
if what == RichTextLineEdit.StrikeOut:
    format.setFontStrikeOut(not format.fontStrikeOut())
if what == RichTextLineEdit.NoSuperOrSubscript:
    format.setVerticalAlignment(
        QTextCharFormat.AlignNormal)
elif what == RichTextLineEdit.Superscript:
    format.setVerticalAlignment(
        QTextCharFormat.AlignSuperScript)
elif what == RichTextLineEdit.Subscript:
    format.setVerticalAlignment(
        QTextCharFormat.AlignSubScript)
self.mergeCurrentCharFormat(format)

```

要修改字体类型、删除线或者是竖直方向的对齐方式，都必须要先获得当前的格式，然后再将这些新的变更应用上去，最后再把更新后的格式与当前格式相合并使其生效。

现在就已经看完了 Rich 文本行编辑器所支持的全部格式化选项。用户一旦输入他们的 Rich 文本内容，毋庸置疑，肯定会先获取到这些内容，以便可以对它们进行存储、搜索或者做出相应的处理。本来也是可以使用 `QTextEdit.toPlainText()` 方法的——但它会清除所有的 HTML 标记，即使功能再好，最终效果应该也不会比 `QLineEdit` 更好了。一种更为合理的做法是使用 `QTextEdit.toHtml()`，但这个方法所返回的 HTML 内容会相当繁琐，因为它必须很通用，以便能够照顾到 PyQt 所支持的全部 HTML 标记。

为了更好地说明这一点，举一个实际的例子，如果有“*The bold cat.*”（共 13 个字符）这样的文本，其中“bold”这个单词会用粗体，而“cat”这个单词则是红色，`toHtml()` 方法会返回 503 个字符：

```

<html><head><meta name="qrichtext" content="1" />
<style type="text/css"> p, li { white-space: pre-wrap; } </style>
</head>
<body style=" font-family:'Nimbus Sans L'; font-size:11pt;
font-weight:400; font-style:normal; text-decoration:none;">
<p style=" margin-top:0px; margin-bottom:0px; margin-left:0px;
margin-right:0px; -qt-block-indent:0; text-indent:0px;">The
<span style=" font-weight:600;">bold</span>
<span style=" color:#ff0000;">cat</span>
<span style=" color:#000000;">. </span></p></body></html>

```

为了能够更好地适合书本显示方式，这里添加了一些换行符。在不同的操作系统平台和 Qt 版

本里，生成的这些文本内容可能会稍有差异，但都在 500 个字符左右。由于这个 Rich 文本行编辑器只需要支持有限的部分标记，所以会使用一种更为简洁的 HTML 代码。例如：

```
The <b>bold </b><font color="#ff0000">cat</font>.
```

这样就只需要 49 个字符。为了能够获得这种较为简化的 TML 格式，这里会提供一个 `toSimpleHtml()`；它的代码有点长，所以会分成三个部分来看。

```
def toSimpleHtml(self):
    html = QString()
    black = QColor(Qt.black)
    block = self.document().begin()
```

这里会先创建一个空的 `QString` 目标，并会假定文本的颜色是黑色。`QTextEdit.document()` 所返回的 `QTextDocument` 类会提供一种遍历它所包含的文本和格式化的方法。从本质上而言，每一个主要的文本组成成分，比如一个段落或者一个表格，都会包含在一个“块”中，可以使用 `QTextDocument.begin()` 来对这些块进行遍历以便能够获取第一个块，然后会调用 `QTextBlock.next()` 来依次获取后续的各个块。对于一个空文档，将可以得到一个无效的块。

每个文本块都至少会包含一个或者多个文本“片段”(fragment)，每个文本片段都有其自己的独立格式属性。实际上，`QTextDocument` 的结构要比这复杂得多，但这里将会忽略这些额外的细节，比如表格、列表和图片等，因为它们在这个 Rich 文本行编辑器中是用不到的。

```
while block.isValid():
    iterator = block.begin()
    while iterator != block.end():
        fragment = iterator.fragment()
        if fragment.isValid():
            format = fragment.charFormat()
            family = format.fontFamily()
            color = format.foreground().color()
            text = Qt.escape(fragment.text())
            if format.verticalAlignment() == \
                QTextCharFormat.AlignSubScript:
                text = QString("<sub>%1</sub>").arg(text)
            elif format.verticalAlignment() == \
                QTextCharFormat.AlignSuperScript:
                text = QString("<sup>%1</sup>").arg(text)
            if format.fontUnderline():
                text = QString("<u>%1</u>").arg(text)
            if format.fontItalic():
                text = QString("<i>%1</i>").arg(text)
            if format.fontWeight() > QFont.Normal:
                text = QString("<b>%1</b>").arg(text)
            if format.fontStrikeOut():
                text = QString("<s>%1</s>").arg(text)
```

对于当前块中的每一个文本片段，都可以提取到它的格式、字体类型和文本颜色。然后，通过调用 `Qt.escape()` 函数就可以将那些 HTML 字符(“&”、“>”和“<”)转换成适当的其他条目(“&amp;”、“&lt;”和“&gt;”)。接下来，会检查该文本片段是下标还是上标，如有必要，会用适当的 HTML 标记包围这些文本。再接着，还会对其他格式进行类似的检测——特别是下画线、斜体、加粗和删除线等——对于这些情况中的每一种，都会为它们的文本加入相应的 HTML 标记。

```

if color != black or not family.isEmpty():
    attribs = ""
    if color != black:
        attribs += ' color="%s"' % color.name()
    if not family.isEmpty():
        attribs += ' face="%s"' % family
    text = QString("<font%1>%2</font>")\
        .arg(attribs).arg(text)
    html += text
    iterator += 1
block = block.next()
return html

```

如果字体类型不为空或者文字颜色不是黑色，就必须要使用带有 `face` 或者 `color`(或者两个都要)属性的 `<font>` 标记。在文本片段的最后，将表示这个文本片段的那些文本内容追加到 `html` 字符串中，这个字符串会保存一整行的 Rich 文本。由于每个文本块都可能会包含一个或者多个文本片段，就需要对遍历器进行累加，直到遍历器的值等于 `QTextBlock.end()` 时才会跳出内部的 `while` 循环，也就是说，处理完了块中的最后一个文本片段。然后，调用 `QTextBlock.next()` 来处理下一个文本块中的文本片段，一直到用来表达所有块都已经处理完毕的无效文本块那里为止才跳出外部的 `while` 循环。到达真正结束的地方才会返回那个包含着有效(但却是最小的)HTML 的 `html` 字符串，这些 HTML 都是用来表达该 Rich 文本行文本内容所必需的东西。

这些就是 `RichTextLineEdit` 类的所有内容。在后续几章中还会使用到它。尽管这一小节只实现了一个单行的 HTML 编辑器，但这里所看到的各项技术则可以轻松应用于 `QTextEdit` 子类上，而该子类则是设计用于整个 HTML 文档编辑的。在上述这些例子中，还应当需要再加入一些快捷键，例如用于文本加粗的 `Ctrl + B` 和用于斜体文字的 `Ctrl + I`，甚至还可以加一个文字效果上下文菜单。但对于那些其他的文本效果、颜色和格式，可能会更适合用于诸如列表、表格这样的大文档，也可以通过菜单选项和工具栏按钮来提供这些功能，就像 HTML 编辑器或者一些字处理软件一样。

## 13.2 文档打印

从 PyQt 程序中获得内容并予以打印的方法有好几种。一种方法是输出一个可以被其他程序打印的形式，例如生成可用于浏览器打印的 HTML，或者生成可被 SVG 绘图程序理解并打印出来的 SVG。

从 Qt 4.1 起，用户可以借助打印对话框 `Print` 并选中其“打印到文件”的选项生成 PDF 格式的文档。当然，也可以通过程序代码的形式生成 PDF 文档。例如，假定要打印的文档是 `QTextDocument` 型文档，则可以：

```

printer = QPrinter()
printer.setPageSize(QPrinter.Letter)
printer.setOutputFormat(QPrinter.PdfFormat)
printer.setOutputFileName(filename)
document.print_(printer)

```

对用户来说，要让应用程序自身能够具有打印功能，这些方法操作起来就显得不大方便了，为此，PyQt 提供了三种主要方法可供选择：

1. 生成 HTML，把它传给 QTextDocument，然后用 QTextDocument.print\_() 引入 QPrinter，或者调用 QTextDocument.drawContents()，以引入 QPainter 来渲染绘制该文档。
2. 创建 QTextDocument 并从它那里获取 QTextCursor，再用 print\_() 或者 drawContents() 方法来绘制它。
3. 创建 QPainter 来直接在 QPrinter 上进行绘制，也就是说，将内容绘制到页面上。这是最为繁琐的方法，但却提供了最佳的操控能力。

在这一节所用到的示例应用程序将会展示所有这三种方法。它可以打印客户声明，在每个 Statement 对象中都保存着公司名、联系人姓名、住址和交易清单，每条交易清单都是一个 (QDate, float) 元组。Statement 类还提供了一个 balance() 方法，它可以返回交易记录的总和。希望能够在右上角位置处打印发行商的商标和地址，紧接着下面是日期，而在左侧则是客户的地址，之后是可以根据客户使用的是信用卡还是贷记卡而套用内容的格式有所不同，然后是一个交易表格和结尾段落。自然，也希望每份声明都能够新起一页。这里将会假设所有的声明都会保存在名为 self.statement 的列表中，这样，借助保存这些声明的窗体的一些方法就可以实现对这些声明的打印。打印的页面效果如图 13.4 所示。



图 13.4 使用 QPainter 的页面打印

但是，在学习这三种通用的打印技术之前，先看看是如何打印图片的，特别是会给出第 6 章图片转换 (Image Changer) 应用程序中 filePrint() 方法的代码实现。

### 13.2.1 图片的打印

重新回到第6章，打印图片用到的是 `MainWindow.filePrint()` 方法，但当时并没有看到它的具体实现代码，因为当时还没介绍 `QPainter`。现在，在第11章已经看到过 `QPainter`，也已经看到过一个通用的“打印图片”函数，所以现在就可以来看一下第6章图片转换应用程序中 `filePrint()` 方法的实现过程(其源代码在文件 `chap06/imagechanger.pyw` 中)。

```
def filePrint(self):
    if self.image.isNull():
        return
    if self.printer is None:
        self.printer = QPrinter(QPrinter.HighResolution)
        self.printer.setPageSize(QPrinter.Letter)
    form = QPrintDialog(self.printer, self)
    if form.exec_():
        painter = QPainter(self.printer)
        rect = painter.viewport()
        size = self.image.size()
        size.scale(rect.size(), Qt.KeepAspectRatio)
        painter.setViewport(rect.x(), rect.y(), size.width(),
                           size.height())
        painter.drawImage(0, 0, self.image)
```

如果这是用户第一次使用这个方法来尝试打印一幅图片，`printer` 实例变量将会是 `None`，所以会先初始化它并同时提供一个适当的页面尺寸默认值(默认页面尺寸大小是 `QPrinter.A4`)。一旦创建了 `printer` 对象，就可以创建并弹出一个模态对话框 `QPrintDialog`；用户就可以用它来选择想要使用的打印机，也可以做一些与打印相关的其他设置选项。由于 PyQt 会尽可能使用系统的本地打印对话框，如果可用的话，所以这个对话框会根据系统的不同而有所不同。如果用户点击了 `Print`，就能够将该图片打印出来。

PyQt 还有一个“绘图设备”的概念，可以在它上面打印一些诸如线条、文本、形状及图片的东西。正如在之前的章节中所看到的那样，一个窗口部件就是一个绘图设备——它的外观就是通过使用阴影和高光线来绘制出具有立体感的深度效果。一个 `QImage` 也是一个绘图设备，`QPrinter` 同样也是。所有的绘图设备都可以使用 `QPainter` 进行绘制，所以这里会创建一个新的 `QPainter` 来用其在 `QPrinter` 上进行绘制。

这里会得到一个表示绘图器的视口(`viewport`)的矩形。视口就是绘图器的绘制区域，而当把绘图器与一个打印机绑定在一起时，就意味着页面的区域就是实际能够在其上打印的地方。(许多打印机不能向右一直绘制到页面的边界)。然后，可以将获取的图片尺寸大小变成 `QSize` 对象，然后就可以把该对象缩放到打印机的视口矩形框内，以保证图片的长宽比固定不变。接下来，修改打印机的视口矩形来匹配经过变形了的图片矩形，保证它的起始点不变，只是对宽度和高度进行缩放。最后，在绘图器的原点处绘制图片就可以了。

如果用户需要的是一个 PDF 格式的文件，可以调用“打印”动作并选择“打印到文件”选项。在某些平台上，通过将“打印到文件”的文件扩展名修改为 `.ps`，还可以得到 PostScript 格式的文档输出。

### 13.2.2 使用 HTML 和 QTextDocument 打印文档

要看到的第一种方法将会用到一个含有 HTML 的字符串，并会使用 `QTextDocument` 把

HTML 渲染绘制到 QPrinter 上。下面的 printViaHtml() 方法相当长，所以会从 3 个部分来对其进行介绍。

```
def printViaHtml(self):
    html = u""
    for statement in self.statements:
        date = QDate.currentDate().toString(DATE_FORMAT)
        address = Qt.escape(statement.address).replace(", ", "<br>")
        contact = Qt.escape(statement.contact)
        balance = statement.balance()
        html += ("<p align=right><img src=':/logo.png'></p>"
                  "<p align=right>Greasy Hands Ltd."
                  "<br>New Lombard Street"
                  "<br>London<br>WC13 4PX<br>%s</p>" 
                  "<p>%s</p><p>Dear %s,</p>" 
                  "<p>The balance of your account is %s." ) % (
                  date, address, contact,
                  QString("$ %L1").arg(float(balance), 0, "f", 2))
    if balance < 0:
        html += ("<p><font color=red><b>Please remit the "
                  "amount owing immediately.</b></font>"")
    else:
        html += (" We are delighted to have done business "
                  "with you.")
    html += ("</p><p>&nbsp;</p><p>"
              "<table border=1 cellpadding=2 "
              "cellspacing=2><tr><td colspan=3>"
              "Transactions</td></tr>"")
```

这里会创建一个空的 unicode 变量 html。然后，会遍历所有的声明。联系人和地址中都会含有文本，所以对任何 HTML 字符都进行预防性转义。地址会在每个部分之间都用逗号进行分割，保存成单行的形式；用换行符替换各个逗号。商标会放在一个资源文件(resource file) 中，以:/作为前缀；这里的资源文件本来也可以使用文件系统中的任何文件，也可以是 PyQt 所支持的任何类型的图片格式。

截至目前使用 Python 的 % 运算符就完成了字符串的格式化。不过在某些情况下，使用 PyQt 的字符串格式化具有一定的优越性。QString 类有一个可以传递对象的 arg() 方法，通常是字符串或者数字，还可以传递一些可选参数。每个 arg() 调用都会用适当的文本来替换 QString 中最左侧的 %n 元素。例如：

```
QString("Copied %1 bytes to %2").arg(5387).arg("log.txt")
```

结果字符串会是：

```
Copied 5387 bytes to log.txt
```

这些 %n 元素都不具有像 Python 的 % 那样的运算符语法格式，不过可以通过向 arg() 方法传递一些额外的参数来获得这样的格式。也可以通过使用 %Ln 来获得本地化的格式。例如：

```
QString("Copied %L1 bytes to %2").arg(5387).arg("log.txt")
```

结果字符串会是：

```
Copied 5,387 bytes to log.txt
```

如果在美国和英国，后面的数字是 5,387，而在其他一些国家中会是 5.387。

在这个例子中，希望能够用两位小数来打印出数字，而对于整个数字来说会将其位数分成 3 个部分。这可以通过带有 4 个参数的 arg() 来实现——浮点型的数字、数字占用的最小字符

数、输出格式(“f”表示一般浮点数，“E”或者“e”表示科学计数法)以及小数点后的位数。也可以给出第 5 个参数，这是一个填充字符(padding character)，如果需要给定最小字符数的话。

在用到的这个例子中：

```
QString("$ %L1").arg(float(balance), 0, "f", 2))
```

在美国的话，余额值 64 325.852 将会输出成字符串 \$ 64 325.85。

继续回到代码，可以根据客户使用的是贷记卡还是信用卡来添加一些不同的文字。然后，创建带有三列的 HTML 表头，第一行会合并所有的列并含有标题“交易记录”。这里的 &nbsp; 是一个 HTML 元素，表示一个非间断空格(nonbreaking space)。

```
for date, amount in statement.transactions:
    color, status = "black", "Credit"
    if amount < 0:
        color, status = "red", "Debit"
    html += ("<tr><td align=right>%s</td>" +
             "<td>%s</td><td align=right>" +
             "<font color=%s>%s</font></td></tr>" % (
                 date.toString(DATE_FORMAT), status, color,
                 QString("$ %L1").arg(
                     float(abs(amount)), 0, "f", 2)))
    html += ("</table></p><p style='page-break-after=always;'>" +
             "We hope to continue doing "
             "business with you,<br>Yours sincerely,"
             "<br><br>K.&nbsp;Longrey, Manager</p>")
```

遍历每一条交易记录，并在表格中为每条记录添加一个新行。然后，添加表格结束的标记和最后段落的标记。如果希望能够在最后一段的位置处添加一个分页符，就可以通过将 page-break-after 样式选项设置成 always 来做到这一点。这个样式选项是在 Qt 4.2 中引入的，而在早前的 Qt 版本中则会忽略这个选项。

```
dialog = QPrintDialog(self.printer, self)
if dialog.exec_():
    document = QTextDocument()
    document.setHtml(html)
    document.print_(self.printer)
```

最后只需要弹出打印对话框即可，并且如果用户点击了 Print，就创建一个新的 QTextDocument，将其 text 设置成在 html 字符串中生成的 HTML 内容，并让该文档在打印机上打印出来即可。

创建一个 HTML 字符串并且使用 QTextDocument 打印它可能会是 PyQt 中最快、最简单的打印输出方式了。唯一不足可能是，若要实现精细控制，就需要稍稍用些技巧，尽管可以使用样式属性或者是设置一个样式表。

### 13.2.3 使用 QTextCursor 和 QTextDocument 打印文档

现在来看看该如何通过编程创建 QTextDocument 而不是通过创建 HTML 字符串 string 再使用 setHtml() 来实现同样的工作。这些代码可能会比刚才的代码长两倍(就是使用 QPainter 之后的那些代码)，不过请不要仅从这个特殊的例子就认为，这些技术的实现通常都必须要使用到更多的代码。

```

def printViaQCursor(self):
    dialog = QPrintDialog(self.printer, self)
    if not dialog.exec_():
        return
    logo = QPixmap(":/logo.png")
    headFormat = QTextBlockFormat()
    headFormat.setAlignment(Qt.AlignLeft)
    headFormat.setTextIndent(
        self.printer.pageRect().width() - logo.width() - 216)
    bodyFormat = QTextBlockFormat()
    bodyFormat.setAlignment(Qt.AlignJustify)
    lastParaBodyFormat = QTextBlockFormat(bodyFormat)
    lastParaBodyFormat.setPageBreakPolicy(
        QTextFormat.PageBreak_AlwaysAfter)
    rightBodyFormat = QTextBlockFormat()
    rightBodyFormat.setAlignment(Qt.AlignRight)
    headCharFormat = QTextCharFormat()
    headCharFormat.setFont(QFont("Helvetica", 10))
    bodyCharFormat = QTextCharFormat()
    bodyCharFormat.setFont(QFont("Times", 11))
    redBodyCharFormat = QTextCharFormat(bodyCharFormat)
    redBodyCharFormat.setForeground(Qt.red)
    tableFormat = QTextTableFormat()
    tableFormat.setBorder(1)
    tableFormat.setCellPadding(2)

```

这里使用的方法是，只有在用户点击了 Print 对话框中的 Print 之后才会创建该文档，而不是就像之前所做的那样先创建它，然后只是在最后才调用它。这里会创建一个文本格式集——一些是带有属性的 QTextBlockFormat，其属性可用于整个段落，还有一些是带有属性的 QTextCharFormat，它的属性则可以用于段落片段，比如词组、单词以及单个的字符。可以使用段落格式来设置文本对齐方式，而字符格式则可以用来设置字体和颜色。这个值中的 216 表示偏移的点数， $\frac{216}{72}$ ，也就是 3 英寸，商标和地址文本将会据此对齐。

与 HTML 的 <p> 标记中设置分页符 page-break-after 格式选项相当的程序语句是对一个段落使用 QTextBlockFormat.setPageBreakPolicy() 方法，不过这只能在 Qt 4.2 之后的版本中使用。除了在这个例子中使用到的文本和表格格式外，还有一些可用于列表、框架和图片的格式。

```

document = QTextDocument()
cursor = QTextCursor(document)
mainFrame = cursor.currentFrame()
page = 1

```

一旦把格式准备好，就可以创建 QTextDocument。接着，会为该文档创建一个 QTextCursor，它可以让能够像在 QTextEdit 中一样获得一个类似的用户插入位置点。

之前曾经提到过，QTextDocument 是由一系列的块构成的；实际上，它们是由一个根框架 (root frame) 或者是由几个蕴含递归结构的框架构成的，这里的根框架自身中的各个元素可能就是一些块（比如，文本块和表格块）。在我们这个例子中的文档就只有一个根框架，其中包含了一系列的文本块和表格。在这个表格中的每个单元格都含有一个文本块，当在表格中完成单元格的插入操作后，就需要把文档的结构形式备份到跟随着表格的位置处（但不是在表格的里面），以便可以在随后的每个表格中插入文本。为此，需要保存对 currentFrame() 的引用，而正在使用的那个框架，会保存在 mainFrame 变量中。

```

for statement in self.statements:
    cursor.insertBlock(headFormat, headCharFormat)
    cursor.insertImage(":/logo.png")
    for text in ("Greasy Hands Ltd.", "New Lombard Street",
                 "London", "WC13 4PX",
                 QDate.currentDate().toString(DATE_FORMAT)):
        cursor.insertBlock(headFormat, headCharFormat)
        cursor.insertText(text)
    for line in statement.address.split(", "):
        cursor.insertBlock(bodyFormat, bodyCharFormat)
        cursor.insertText(line)
    cursor.insertBlock(bodyFormat)
    cursor.insertBlock(bodyFormat, bodyCharFormat)
    cursor.insertText("Dear %s," % statement.contact)
    cursor.insertBlock(bodyFormat)
    cursor.insertBlock(bodyFormat, bodyCharFormat)
    balance = statement.balance()
    cursor.insertText(QString(
        "The balance of your account is $ %L1.").arg(
        float(balance), 0, "f", 2))
    if balance < 0:
        cursor.insertBlock(bodyFormat, redBodyCharFormat)
        cursor.insertText("Please remit the amount owing "
                         "immediately.")
    else:
        cursor.insertBlock(bodyFormat, bodyCharFormat)
        cursor.insertText("We are delighted to have done "
                         "business with you.")
    cursor.insertBlock(bodyFormat, bodyCharFormat)
    cursor.insertText("Transactions:")
    table = cursor.insertTable(len(statement.transactions), 3,
                               tableFormat)

```

文档一旦构建起来就可以通过 `QTextCursor` 向文档插入各个元素，现在就为遍历每份客户声明做好了准备。

对于打算插入的每一段，可以插入一个带有段落和字符格式的新块。然后，插入段落中打算包含的各个文本或者图片。通过插入一个什么内容都没插入的块可以实现一些空段的插入（以便可以占用一些纵向空间）。

要插入一个表格就必须说明该表格应该有多少行、多少列，还要说明它的格式是什么。

```

row = 0
for date, amount in statement.transactions:
    cellCursor = table.cellAt(row, 0).firstCursorPosition()
    cellCursor.setBlockFormat(rightBodyFormat)
    cellCursor.insertText(date.toString(DATE_FORMAT),
                          bodyCharFormat)
    cellCursor = table.cellAt(row, 1).firstCursorPosition()
    if amount > 0:
        cellCursor.insertText("Credit", bodyCharFormat)
    else:
        cellCursor.insertText("Debit", bodyCharFormat)
    cellCursor = table.cellAt(row, 2).firstCursorPosition()
    cellCursor.setBlockFormat(rightBodyFormat)
    format = bodyCharFormat
    if amount < 0:
        format = redBodyCharFormat

```

```
cellCursor.insertText(QString("$ %L1").arg(  
    float(amount), 0, "f", 2), format)  
row += 1
```

表格的每一行都表示一条交易记录，带有日期、一些文字[ 贷记卡(Debit)或者信用卡( Credit ) ]、数量等信息，如果是贷记卡( Debit )会用红色表示。要向表格插入这些元素必须得到一个 QTextCursor，它可以对给定的行和列进行访问。不必向一个单元格插入一个新块(除非一个单元格中会有多个段落)，因此只需在设置单元格的段落格式后插入所需的文本内容即可。

```
cursor.setPosition(mainFrame.lastPosition())  
cursor.insertBlock(bodyFormat, bodyCharFormat)  
cursor.insertText("We hope to continue doing business "  
    "with you,")  
cursor.insertBlock(bodyFormat, bodyCharFormat)  
cursor.insertText("Yours sincerely")  
cursor.insertBlock(bodyFormat)  
if page == len(self.statements):  
    cursor.insertBlock(bodyFormat, bodyCharFormat)  
else:  
    cursor.insertBlock(lastParaBodyFormat, bodyCharFormat)  
cursor.insertText("K. Longrey, Manager")  
page += 1
```

一旦完成了表格的构建并在其后添加了各个元素，就必须把文本光标的位置重置到紧跟表格之后。如果不这样做，光标就会插入到表格中并且会在表格中第一页剩下的页面处结束，而第二页会在第一页中结束，以此类推循环不止！要避免这一问题，可以把文本光标设置到文档的最后位置处，那个位置也是插入的最后一个东西的位置，也就是说，正好是紧跟在表格之后。

完成页面只是采用适当的格式来插入一些块而已，下面还需要插入相应的文本。对于除了最后一页之外的所有页面，还需要将最后的块的格式设置成 lastParaBodyFormat，这将可以确保后面跟随的内容会在一个全新的页面上(使用 Qt 4.2)。

```
document.print_(self.printer)
```

最后要说的是，记得把文档在打印机上打印出来。由于此时的文档是空的，所以如果愿意，也可以对其调用 toHtml() 来得到 HTML 的形式。这也意味着，如果愿意，可以用编程的方式联合使用 QTextCursor 和 QTextDocument 来创建 HTML 页面。

使用 QTextDocument 的好处是，无论给它的是 HTML 字符串还是直接使用 QTextCursor 来创建文档，要找到该在页面的什么位置放置文本，都不必进行大量的计算。不足之处在于，无论是否需要，PyQt 都会在文档上放置页码，并且也无法精细地控制其位置。如果使用 QPainter 则这些问题都不会发生。

### 13.2.4 使用 QPainter 打印文档

有关如何使用 QPainter 进行打印的问题将会在这一节做个了断。使用这一方法，也就意味着不仅需要我们自己计算所有的位置，而且还将具有在页面任意位置进行绘制的好处，而不必受限于 HTML 或者 QTextDocument 到底能够表达什么以及不能表达什么。此外，有关绘制中的相关方法和技术都已在前面两章中见过，因为无论是对窗口部件、图片还是对页面， PyQt 有一个统一的方法。

```

def printViaQPainter(self):
    dialog = QPrintDialog(self.printer, self)
    if not dialog.exec_():
        return
    LeftMargin = 72
    sansFont = QFont("Helvetica", 10)
    sansLineHeight = QFontMetrics(sansFont).height()
    serifFont = QFont("Times", 11)
    fm = QFontMetrics(serifFont)
    DateWidth = fm.width(" September 99, 2999 ")
    CreditWidth = fm.width(" Credit ")
    AmountWidth = fm.width(" W999999.99 ")
    serifLineHeight = fm.height()
    logo = QPixmap(":/logo.png")
    painter = QPainter(self.printer)
    pageRect = self.printer.pageRect()
    page = 1

```

先从显示给用户的 Print 对话框开始，而如果用户点击了 Cancel 按钮，对话框就退出打印操作。如果继续打印，就可以设置字体、宽度和线高，并且会创建一个 QPainter 来在打印机上进行直接绘制。如果给定的字体不可用，PyQt 将会使用它所能找到的最为接近的匹配字体。

```

for statement in self.statements:
    painter.save()
    y = 0
    x = pageRect.width() - logo.width() - LeftMargin
    painter.drawPixmap(x, 0, logo)
    y += logo.height() + sansLineHeight
    painter.setFont(sansFont)
    painter.drawText(x, y, "Greasy Hands Ltd.")
    y += sansLineHeight
    painter.drawText(x, y, "New Lombard Street")
    y += sansLineHeight
    painter.drawText(x, y, "London")
    y += sansLineHeight
    painter.drawText(x, y, "WC13 4PX")
    y += sansLineHeight
    painter.drawText(x, y,
                    QDate.currentDate().toString(DATE_FORMAT))
    y += sansLineHeight
    painter.setFont(serifFont)
    x = LeftMargin
    for line in statement.address.split(", "):
        painter.drawText(x, y, line)
        y += serifLineHeight
    y += serifLineHeight

```

对于每份客户声明，会打印商标、地址、日期以及客户的地址。在每份客户声明的开头，还会保存绘制器的状态，包括它的字体、画笔、画刷以及变换矩阵，而在每份客户声明的最后则会恢复这些状态。这样就可以确保每次都能够从零开始每一份客户声明。

```

painter.drawText(x, y, "Dear %s," % statement.contact)
y += serifLineHeight
balance = statement.balance()
painter.drawText(x, y, QString("The balance of your "
                             "account is $ %L1").arg(float(balance), 0, "f", 2))
y += serifLineHeight

```

```

if balance < 0:
    painter.setPen(Qt.red)
    text = "Please remit the amount owing immediately."
else:
    text = ("We are delighted to have done business "
            "with you.")
painter.drawText(x, y, text)

```

在地址之后会采用常规形式并根据账户的状态，套用格式打印文本内容。

```

painter.setPen(Qt.black)
y += int(serifLineHeight * 1.5)
painter.drawText(x, y, "Transactions:")
y += serifLineHeight
option = QTextOption(Qt.AlignRight|Qt.AlignVCenter)

```

对于交易记录表格先从写入表题开始，然后会创建一个 QTextOption 对象。这些对象可以用来说明各类文本格式选项，包括对齐和自动换行等。

```

for date, amount in statement.transactions:
    x = LeftMargin
    h = int(fm.height() * 1.3)
    painter.drawRect(x, y, DateWidth, h)
    painter.drawText(
        QRectF(x + 3, y + 3, DateWidth - 6, h - 6),
        date.toString(DATE_FORMAT), option)
    x += DateWidth
    painter.drawRect(x, y, CreditWidth, h)
    text = "Credit"
    if amount < 0:
        text = "Debit"
    painter.drawText(
        QRectF(x + 3, y + 3, CreditWidth - 6, h - 6),
        text, option)
    x += CreditWidth
    painter.drawRect(x, y, AmountWidth, h)
    if amount < 0:
        painter.setPen(Qt.red)
    painter.drawText(
        QRectF(x + 3, y + 3, AmountWidth - 6, h - 6),
        QString("$ %L1").arg(float(amount), 0, "f", 2),
        option)
    painter.setPen(Qt.black)
    y += h

```

要绘制交易记录表，必须由我们自己绘制各个文本和各条直线。这里会通过为表格的每个单元绘制一个矩形而不是通过绘制一些直线来分割各个单元格的形式来做到这一点。这就意味着各个矩形会共享这些公共的直线——例如，某个矩形的右侧会与该矩形旁边的矩形的左侧相互共享——这样就会出现重复打印的情况，不过，看起来并不算太过明显。

```

y += serifLineHeight
x = LeftMargin
painter.drawText(x, y, "We hope to continue doing "
                "business with you,")

y += serifLineHeight
painter.drawText(x, y, "Yours sincerely")
y += serifLineHeight * 3
painter.drawText(x, y, "K. Longrey, Manager")

```

最后一段与之前的两个方法一样，不过这一次，将会在页面的下方添加一条免责声明。

```
x = LeftMargin
y = pageRect.height() - 72
painter.drawLine(x, y, pageRect.width() - LeftMargin, y)
y += 2
font = QFont("Helvetica", 9)
font.setItalic(True)
painter.setFont(font)
option = QTextOption(Qt.AlignCenter)
option.setWrapMode(QTextOption.WordWrap)
painter.drawText(
    QRectF(x, y,
           pageRect.width() - 2 * LeftMargin, 31),
    "The contents of this letter are for information "
    "only and do not form part of any contract.",
    option)
```

当使用 `QPainter` 时，在页面添加页脚非常简单，因为在已经准确知道页面的各个尺寸大小后，就可以在任何喜欢的  $(x, y)$  位置处进行绘制。

```
page += 1
if page <= len(self.statements):
    self.printer.newPage()
    painter.restore()
```

最后，在除最后一条客户声明之外的每一条客户声明之后都会切换一个新的页面。在所有的 Qt 4 各个版本中这都可以正常工作，而不像之前的两个方法，它们只能在 Qt 4.2 或者更高的版本上才能标注适当的页码。

尽管对于打印来说，使用 `QPainter` 要比使用 `QTextDocument` 需要更操心和更复杂的计算，但 `QPainter` 确实能够对输出赋予完全控制。

## 小结

使用 `QSyntaxHighlighter` 为诸如源代码这样的常规语法普通文本提供语法高亮功能相当简单。对于多行结构的处理也可以非常简单。最为困难的地方在于对那些含混不清和特殊情形的处理，比如引号中又含有引号字符串，或者是在引号中的字符串以注释符开头，又或者是能够取消或者结束前述符号语义的其他情形。一种替换方法是使用 `QScintilla` 编辑器。

`QTextEdit` 类非常强大，用途也非常广泛。除常规用法之外，它还可以既用来编辑普通文本，又可以用来编辑 HTML。通过键盘和上下文菜单事件处理程序为用户提供一些基本的格式化选项的 `QTextEdit` 子类的创建并不困难，并且所用到的技术也可轻易地扩展到菜单和工具栏上，而用户可以通过它们实现列表、表格和图片的添加、编辑与删除，还可以将格式应用到诸如下画线和删除线这样的字符级层次上，也可以将格式应用到诸如左对齐、右对齐、中间对其或者两端对齐这样的段落级层次上。

由 `QTextEdit.toHtml()` 所返回的 HTML 相当繁琐，因为它必须支持相当大范围内的 HTML 标记。可以提供一些自己的方法来遍历 `QTextDocument` 的结构并输出自定义的格式。在该例子中，可以输出相当简单并且相当短的 HTML，不过同样是这一方法，应当也是可以用来输出 XML 或者其他类型的标记的。

可以相当简单地将大部分的简易格式应用到由 QTextEdit、QTextBrowser、QLabel 和 QGraphicsTextItem 底层使用的 QTextDocument 上。而要应用一些高级的格式化，比如表格，则要有些技巧，因为必须要小心避免出现段落块之间的相互嵌套。

间接通过输出 HTML 或者 SVG，或者直接通过使用 QPrinter 打印到物理打印机上，又或者从 Qt 4.1 后输出 PDF 文件，都可以实现文档的打印。通过创建 HTML 字符串并将其传给 QTextDocument，或者通过编程方式把元素插入到空白的 QTextDocument 中，都可以实现文档的打印。在这两种情况下，QTextDocument 都可以在打印机上打印自己，或者是在 QPainter 上绘制自己。

对那些熟悉 HTML 标记的用户来说，使用 HTML 是最简单的方法了，而且通过使用样式属性和样式表，可以获得相当大的控制权。用 QTextCursor 在 QTextDocument 中进行插入操作可以相当简单地获得精细控制，特别是对于那些不熟悉样式表的用户来说。通过直接使用 QPainter 可以对页面外观获得相当的控制力。对于那些不习惯使用 QPainter API 或者打算在绘制处理和打印处理都重复使用同一份代码的人来说，这也是最为简单的方法。使用 QTextDocument 也可以获得那些能够重用的代码，因为可以在 QLabel 或者使用了 QTextDocument 的其他窗口部件中绘制它们。也可以将它们绘制到任意的绘制设备上，比如窗口部件上，通过使用 QPainter 而实现它们的打印。

## 练习题

用快捷键 `Ctrl + ]` 和 `Ctrl + [` 来添加两个动作，缩进和取消缩进。在 `images` 子目录中提供了一些适当的图标，并已经将其放到了资源文件 `.qrc` 中。这两个动作都应当可以添加到 Edit 菜单和编辑工具栏上。实现 `editIndent()` 和 `editUnindent()` 方法的代码。应当可以通过在行首位置处插入或者移除 4 个空格来实现当前行的缩进或者取消缩进，而不必考虑插入点是在行中间还是在行内的其他地方。如果是在行尾，那么在缩进或者取消缩进后，插入点的位置应该仍旧在相应的相对位置处。这些动作应当是实例变量，并且仅应当在文档非空时才能够启用。

要确保用到了 `QTextCursor.beginEditBlock()` 和 `QTextCursor.endEditBlock()`，以便能够将缩进和取消缩进作为一个单独的动作而允许撤销——`QTextEdit` 可以支持撤销的 `Ctrl + Z` 快捷键。大约总共需要使用 20 行的代码就可以完成这两个方法。

如果打算更多些挑战，还可以试着扩展下这两个方法，以便在有选中的文本时，只会对选中的那些行进行缩进或者取消缩进。这样另外大约需要 40 行的代码，并且稍稍需要些技巧。要确保在操作后，仍然要能够选中一开始入选中的那些内容。这样就可能需要阅读有关 `QTextCursor`，特别是 `anchor()`、`position()`、`setPosition()` 和 `movePosition()`，这样一些方法的文档说明。

本练习题的参考答案在文件 `chap13/Pythoneditor_ans.pyw` 中。

## 第 14 章 模型/视图编程

- 使用简便项窗口部件
- 创建自定义模型
- 创建自定义委托

模型/视图编程 (Model/View programming) 是一种有关将数据与其可视化表达相互分离的技术。这一技术起初流行于 Smalltalk 编程语言中的 MVC (Model/View/Controller, 模型/视图/控制器) 编程模式。

所谓模型 (model)，是一个为访问数据元素项 (data item) 提供统一接口的类。所谓视图 (view)，是一个将模型中的数据项在屏幕上展示给用户的类。所谓控制器 (controller)，是一个位于用户接口 (如鼠标事件和键盘按键) 和数据元素项操作之间起到媒介作用的类。

MVC 方法可以提供多种好处。例如，可以对巨量数据集进行处理，因为只有那些真正显示或者编辑的数据才会从数据源中读取或者向数据源中写入。这在使用两种或者更多种不同方式查看同一数据集的不同视图，或者是在对大数据集的不同部分进行查看时，都非常有用。同样，如果改变了数据集的存储方式，例如，从二进制文件改为数据库，只需要对模型做些相应的改变即可——在视图和控制器中的所有逻辑关系仍将可以正常工作，因为模型会将它们与数据完全隔绝开来。

PyQt 对 MVC 的使用略有不同，可称之为模型/视图/委托 (Model/View/Delegate)，可以提供与 MVC 相同的全部优秀特征。在图 14.1 中给出了两者的示意图。其中，最为主要的一个不同之处在于，经典 MVC 保留用于控制器的一些函数功能既可以通过委托实现，也可以通过模型/视图/委托方法中的模型来进行实现。

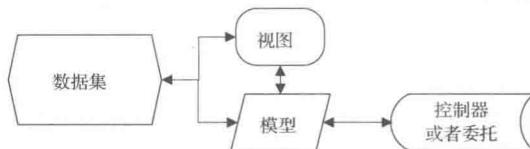


图 14.1 模型/视图/控制器和模型/视图/委托

从概念上而言，模型/视图/委托方法的工作方式类似于这样：数据源中数据的读写是通过模型完成的。视图会向模型请求视图打算显示的那些数据项，也就是说，那些项在用户界面中就是可见的。对于每一个视图所显示的项，它都会把项和绘图器 (painter) 传给委托，并要求委托绘制该项。相应地，如果用户开始编辑操作，视图就要要求委托提供一个适当的编辑器，而如果用户接受了他们的编辑 (也就是，如果用户没有按下 Esc 或者 Cancel 键)，更新后的数据就会回传到模型。这里的编辑器可以是任何窗口部件，可以紧紧绘制在项的上方，让应用程序的用户产生一种错觉，认为该项已经是可编辑的。

模型中的每个数据项 (因而也就意味着数据项在数据集中) 都可以通过唯一的 QModelIndex 加以识别。每个模型索引值 (model index) 都有三个重要的属性：行、列和父对象。

1. 对于一维模型[比如，列表(list)]只会用到行。
2. 对于二维模型[比如，表格(table)，包括数据库表格]只会用到行和列。
3. 对于分层模型[比如，树(tree)]则会用到全部三个属性。

尽管 QModelIndex 可以指向任何模型中的任何数据项，但仍需时刻记住的是，从概念上而言，实际上有两种类型的模型。第一种是表格，包括列表，因为列表只不过是一个仅用到了列的表格。当用到表格时，实际上是用到了行和列。第二种是树。对于树来说，实际上是用到了各个父对象和各个子对象(树中也是可以采用行-列法的，不过这样做会事与愿违并将导致代码效率低下且维护困难)。

无论底层数据集是什么，也无论数据是在内存中、数据库中还是在文件中，PyQt 模型都会提供统一的数据访问接口——特别是会提供 QAbstractItemModel.data() 和 QAbstractItemModel.setData() 方法。创建自定义模型并在其中包含数据集也是可以的，比如对字典或者列表进行封装的封装器。

在模型中的所有数据都会存储成 QVariant。这并不意味着所有的数据集的数据项都必须是 QVariant——模型仅是一个数据集的接口，在任何给定的情况下，模型或许只会访问整个数据集中很少的一部分数据项，因此，只有那些真实用到的数据才会存储成 QVariant，并且也只是在模型中才如此。模型会负责模型所用到的数据集中数据项类型在从 QVariant 或者向 QVariant 类型的转换。

一些 PyQt 的窗口部件，包括 QListWidget、QTableWidget 和 QTreeWidget，都是带有模型和委托的视图。这些窗口部件都是简便项视图窗口部件(convenience item view widget)，对于小型或者特种数据集它们尤其有用。在 4.1 节中，将会看到这些窗口部件的用法。

PyQt 也提供了一些纯视图窗口部件，包括 QListView、QTableView 和 QTreeView。这些窗口部件必须与外部的窗口部件一起使用，这些外部窗口部件要么是我们自己创建的，要么是些内置的模型，如 QStringListModel、QDirModel 或者 QSqlTableModel。在 14.2 节，将会看到如何创建一个自定义模型。

所有的简便视图和纯视图都会用到默认委托，它用来控制数据项是如何展示和编辑的。在这一章的最后一节，将会看到如何创建自定义委托，以便能够训练对数据项的编辑和展示功能进行完整控制。自定义委托可用于任何视图中，无论它是简便视图还是纯视图。

本章将为 PyQt 各个模型/视图类的使用奠定一定的基础。在第 15 章，将会看到该如何将这些模型/视图类用于数据库，而在第 16 章，还会涉及一些更高级的用法，包括自定义视图的创建、委托中重用代码的优化以及树中表型数据的展示等。

本章将会在所有的例子中都使用同一个数据集，以便可以更为容易地对所用到的各种技术进行比较和对照。该数据集的项会在 14.1 节中予以介绍。

## 14.1 使用简便项窗口部件

简便项窗口部件(convenience item widget)是一些具有内置模型的视图窗口部件。它们会为数据的展示和编辑使用默认委托，但如果愿意，亦可以用自定义委托来替换这里的默认委托。

如图 14.2 所示，这是同一个数据集在三个不同的简便视图窗口部件中的显示效果。这就意味着，这些数据会分别被复制到每一个窗口部件中，因此，会存在相当可观的数据重复率。另外一个问题是，如果允许用户编辑这些数据，就必须编写一些代码来确保所有这三个视图能

够保持同步。如果使用自定义模型，所有这些问题都将不复存在，很快就会在下一节看到这一做法。

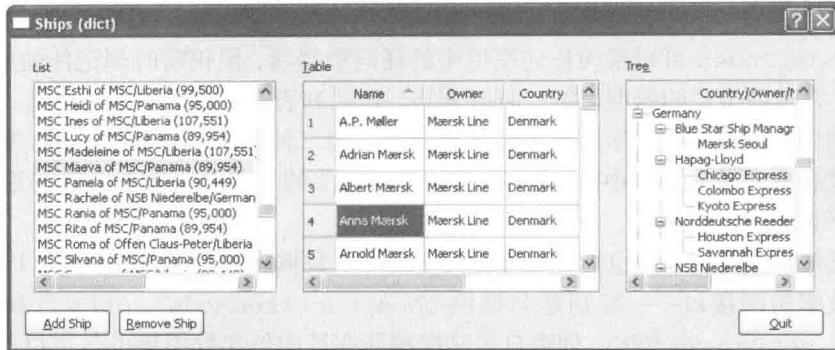


图 14.2 使用中的 QListWidget、QTableWidget 和 QTreeWidget

所正在使用的数据集是一个有关集装箱货船信息的集。每条船都可以通过 `Ship` 对象进行表示，其定义在 `chap14/ships.py` 模块中。

```
class Ship(object):
    def __init__(self, name, owner, country, teu=0, description=""):
        self.name = QString(name)
        self.owner = QString(owner)
        self.country = QString(country)
        self.teu = teu
        self.description = QString(description)

    def __cmp__(self, other):
        return QString.localeAwareCompare(self.name.toLower(),
                                           other.name.toLower())
```

前面这些代码是完整的 `Ship` 类。其中的整数属性 `teu` 表示“20 英尺标准集装箱”(twenty-foot equivalent units)，也就是说，该条船可以放下多少个 20 英尺长的集装箱<sup>①</sup>(当今的绝大部分集装箱都是 40 英尺长，所以每个集装箱就相当于两个 TEU)。这里的 `name`、`owner` 和 `country` 属性都是纯文本，不过 `description` 属性可以保存一行 HTML 文本。

特殊的 `__cmp__()` 方法提供了一种用于排序的比较方法。`QString.localeAwareCompare()` 方法可基于本地化方式实现字符串的比较——例如，可以正确处理重音符号。

由于这里使用的是不带自定义委托的简便视图，所以只具有数据项编辑的有限控制能力。例如，不能在编辑 `owner` 和 `country` 时提供下拉式组合框(combobox)，也不能为 TEU 的编辑使用微调框(spinbox)。同时，`description` 文本的显示很简陋，而不能将其显示成 HTML 的形式。当然，在本章的后续内容中将会解决所有这些问题，但在这里，仍将主要关注于简便视图的用法。

对于列表、表格和树中的各个项，都会用到简便视图窗口部件，设置它们的字体、文本对齐方式、文本颜色以及背景色也是可能的，甚至还可以给它们设置图标或者使其成为复选框。对于那些纯视图窗口部件来说，可以训练通过自定义模型来近似控制数据项的外观；或者也可以训练通过使用自定义委托来完美控制数据项的外观和编辑功能。

<sup>①</sup> TEU 是以长度为 20 英尺的集装箱为国际计量单位，也称国际标准箱单位。通常用来表示船舶装载集装箱的能力，也是集装箱和港口吞吐量的重要统计和换算单位——译者注。

本节例子的代码在文件 `chap14/ships-dict.pyw` 中。存放在 Python 字典中的数据自身封装在 `ships.ShipContainer` 类中。这里仅会讨论那些与模型/视图编程相关的代码——其他部分的代码所用到的思想和方式都与本书中前面所看到的代码一样(例如,类似于第 8 章中的例子),也就不难模仿。

```
class MainForm(QDialog):
    def __init__(self, parent=None):
        super(MainForm, self).__init__(parent)
        listLabel = QLabel("&List")
        self.listView = QListWidget()
        listLabel.setBuddy(self.listView)

        tableLabel = QLabel("&Table")
        self.tableView = QTableWidget()
        tableLabel.setBuddy(self.tableView)

        treeLabel = QLabel("Tre&e")
        self.treeView = QTreeWidget()
        treeLabel.setBuddy(self.treeView)
```

对于这里的每一个简便视图都会创建一个标签并为其设置一个伙伴,从而可以方便地使用键盘进行导览操作。有关布局的代码与之前看到的相似,所以这里会忽略它们,而主要关注于各个连接以及数据结构的创建过程。

```
self.connect(self.tableView,
             SIGNAL("itemChanged(QTableWidgetItem*)"),
             self.tableItemChanged)
self.connect(addShipButton, SIGNAL("clicked()"), self.addShip)
self.connect(removeShipButton, SIGNAL("clicked()"),
            self.removeShip)
self.connect(quitButton, SIGNAL("clicked()"), self.accept)
self.ships = ships.ShipContainer(QString("ships.dat"))
self.setWindowTitle("Ships (dict)")
```

默认情况下,列表窗口部件是不可编辑的,因而所有用户能做到的都仅是选择一个项。这一点对于树窗口部件来说也是如此。不过,表格窗口部件在默认情况下是可编辑的,用户也就可以通过 F2 键或者是双击一个单元格开始编辑。借助 `QAbstractItemView.setEditable()`,可以完美控制一个视图窗口部件是否是可编辑的;所以,例如,可以让表格只读或者让列表可编辑。

这个应用程序允许用户编辑表格中的船只数据,也可以添加和移除船只数据。通过在载入数据后对这三个视图进行组装,还可以对它们进行全部更新,无论变化是何时发生的。

```
def populateList(self, selectedShip=None):
    selected = None
    self.listView.clear()
    for ship in self.ships.inOrder():
        item = QListWidgetItem(QString("%1 of %2/%3 (%L4)") \
                               .arg(ship.name).arg(ship.owner).arg(ship.country) \
                               .arg(ship.teu))
        self.listView.addItem(item)
        if selectedShip is not None and selectedShip == id(ship):
            selected = item
    if selected is not None:
        selected.setSelected(True)
        self.listView.setCurrentItem(selected)
```

与其他的组装方法类似，这个方法用来把窗口部件与选择项相对应的 `selectedShip` 组装到一起——`Ship` 的 `id()`——如果传入有 `id` 的话。

先从窗口部件的清空开始。然后会对船只集装箱中的每条船进行遍历。这里的 `inOrder()` 方法是由自定义类 `ShipContainer` 提供的。对于每条船，都会创建一个单列表窗口部件项，其中保存一个单字符串。这里会使用 `QString.arg()`，以便可以借助适当的数值运算（比如，逗号）用 `%L` 来显示 TEU 的数量。

如果到达用来显示选中船只的列表窗口部件，就可以用一个引用指向该选中项，并在该列表窗口部件组装之后，让选中的项成为当前项和选中项。

```
def populateTable(self, selectedShip=None):
    selected = None
    self.tableWidget.clear()
    self.tableWidget.setSortingEnabled(False)
    self.tableWidget.setRowCount(len(self.ships))
    headers = ["Name", "Owner", "Country", "Description", "TEU"]
    self.tableWidget.setColumnCount(len(headers))
    self.tableWidget.setHorizontalHeaderLabels(headers)
```

组装表格的方法与组装列表的方法相当相似。先从表格的清空开始——这样做既会清空各个单元格，还会清空竖直方向和水平方向的表头（行数和列标题）。然后，设置行和列的数字，同时也对各个列标题进行设置。

希望用户能够在点击一列后，表格能够对那一列的内容进行排序。这个功能内置于 `QTableWidget` 中，不过在组装表格之前必须先将其关闭<sup>①</sup>。在表格一旦组装完毕，就必须重新将其打开。

```
for row, ship in enumerate(self.ships):
    item = QTableWidgetItem(ship.name)
    item.setData(Qt.UserRole, QVariant(long(id(ship))))
    if selectedShip is not None and selectedShip == id(ship):
        selected = item
    self.tableWidget.setItem(row, ships.NAME, item)
    self.tableWidget.setItem(row, ships.OWNER,
                           QTableWidgetItem(ship.owner))
    self.tableWidget.setItem(row, ships.COUNTRY,
                           QTableWidgetItem(ship.country))
    self.tableWidget.setItem(row, ships.DESCRIPTION,
                           QTableWidgetItem(ship.description))
    item = QTableWidgetItem(QString("%L1") \
                           .arg(ship.teu, 8, 10, QChar(" ")))
    item.setTextAlignment(Qt.AlignRight|Qt.AlignVCenter)
    self.tableWidget.setItem(row, ships.TEU, item)
self.tableWidget.setSortingEnabled(True)
self.tableWidget.resizeColumnsToContents()
if selected is not None:
    selected.setSelected(True)
    self.tableWidget.setCurrentItem(selected)
```

对于每条船只，都必须创建一个单独的表格项，以便在行里的每个单元格中显示它的数据。列索引值，`NAME`、`OWNER`，等等，这些都是来自 `ships` 模块中的整型数。

在每行的第一个项中会设置文本（船只的名字）和船只的 ID，如同用户数据一样。ID 的存

<sup>①</sup> 在 Qt 4.0 和 Qt 4.1 中，在组装一个表格之前而忘记关闭排序功能无甚大碍，但从 Qt 4.2 之后就必须要那么做了。

储为我们提供了借助表格项来表示该项所在行的船只的一种方式。这之所以能够工作是因为 `ShipContainer` 是一个字典，其键是船只的 ID，而其值则是 `ships`。

对于简单文本项，通常可以用一个语句来创建该项并将其插入到表格中：这里对 `owner`、`country` 和 `description` 属性就是这样做的。不过，如果打算格式化该项或者在其中存储用户数据，就必须单独创建该项，然后再调用它的方法，最后再用 `setItem()` 将其放到表格中。这里会使用第二种方法来把船只的 ID 存储成用户数据，并向右对齐存储 TEU 的值。

TEU 的值是整数，而 `QString.arg()` 方法会带四个参数：一个整数、一个最小域宽度、一个数进制和一个填补字符，该字符可以在必要的时候填满最小域的宽度。

一旦表格组装完成，就可以重新打开排序功能了，将每列的宽度值重置为单元格中的最大宽度值，并让选中的项（如果有的话）成为当前项并将其选中。

列表和表格的组装非常相似，因为它们使用的都是基于行和基于列的方法。对于树的组装则相当不同，因为必须使用基于父子对象的方法。有关船只数据的视图是两个列。第一列是用国家（country）作为根项的树，下一级别的项是拥有者（owner），底层项是船只（ships）。第二列只是给出 TEU。本来还应该再增加一个第三列来给出一个备注，不过，即使不增加备注列也不会对理解树窗口部件的工作方式有什么不同，所以就没有再增加这一列。

```
def populateTree(self, selectedShip=None):
    selected = None
    self.treeWidget.clear()
    self.treeWidget.setColumnCount(2)
    self.treeWidget.setHeaderLabels(["Country/Owner/Name", "TEU"])
    self.treeWidget.setItemsExpandable(True)
    parentFromCountry = {}
    parentFromCountryOwner = {}
```

这里开始方式与之前的类似，清空树，设置树的各个列和各列的标题。还会把树的各个项设置成可扩展型。随后将会对这里的两个字典进行说明。

```
for ship in self.ships.inCountryOwnerOrder():
    ancestor = parentFromCountry.get(ship.country)
    if ancestor is None:
        ancestor = QTreeWidgetItem(self.treeWidget,
                                   [ship.country])
    parentFromCountry[ship.country] = ancestor
    countryowner = ship.country + "/" + ship.owner
    parent = parentFromCountryOwner.get(countryowner)
    if parent is None:
        parent = QTreeWidgetItem(ancestor, [ship.owner])
        parentFromCountryOwner[countryowner] = parent
    item = QTreeWidgetItem(parent, [ship.name,
                                    QString("%L1").arg(ship.teu)])
    item.setTextAlignment(1, Qt.AlignRight|Qt.AlignVCenter)
    if selectedShip is not None and selectedShip == id(ship):
        selected = item
    self.treeWidget.expandItem(parent)
    self.treeWidget.expandItem(ancestor)
```

每个船只（ship）都必须在树中有一个 owner 父对象，每个 owner 都必须在树中有一个 country 父对象。

对于每个船只（ship）都要做个检查，看看树中是否还有用于某个项的 country。通过查询 `parentFromCountry` 字典就可以做到这一点。如果没有，就用树窗口部件为其父对象来创

建一个新的 country 项，并在字典中存储一个指向该项的引用。此时，要么就获得了，要么就创建了一个 country(被继承的)项。

接下来，看看树中是否还有用于 ship 的 owner 项。为此，会查询 parentFromCountryOwner 字典。同样，如果没有，就创建一个新的 owner 项，其父对象就是刚才找到或者创建了的那个 country(被继承的)项，并同样在字典中存储一个指向该项的引用。此时，要么就获得了，要么就创建了一个 owner(父对象)项。现在就可以创建一个用于该 ship 的新项，其父对象就是 owner。

这样就可以有一个 parentFromCountryOwner 字典，而不是 parentFromOwner 字典，因为某一个 owner 可能会在多个国家中有业务往来。

树窗口部件项可以有多个列，这就是为什么在创建它们的时候，除了它们的父对象之外，还要给它们传递一个列表的原因。对于船只(ship)来说，还会用到一些额外的列，实际上只有一个额外列，来保存该船只的TEU。通过在调用 QTreeWidgetItem.setTextAlignment() 时用列数字作为其第一个参数，可以右对齐 TEU 的数字。

在向简易视图窗口部件中添加项时，既可以不带父对象创建这些项并自己添加它们，例如，可以使用 QTableWidgetItem.setItem()，也可以带上父对象来创建这些项，此时，PyQt 将会自动添加它们。在组装树的时候这里会选用第二种方法。

这里选择对每个项都进行了扩展，以便该树能够在一开始就可以得到完全扩展。对于相对较小的树来说这样做还是可以的，不过，并不推荐将其用于大型树。

```
self.treeWidget.resizeColumnToContents(0)
self.treeWidget.resizeColumnToContents(1)
if selected is not None:
    selected.setSelected(True)
    self.treeWidget.setCurrentItem(selected)
```

结束的时候，会改变两列的尺寸大小并让选中的项(如果有的话)变为当前项和选中项。

列表和树视图会仍旧保存为只读状态。这就意味着，只是在用户编辑表格中各个项，或者只是在添加或者移除 ships 的时候，这些数据才可以改变；因而在这些情况下，就必须确保各个视图的同步性。以数据编辑为例，无论编辑操作是在何时完成的，都会调用 tableItemChanged() 方法。通过改变焦点，用户就可以完成一个编辑操作，例如，在该项外点击鼠标，或者按下 Tab 键，又或者是按下 Enter 回车键；通过按下 Esc，就可以取消编辑操作。

```
def tableItemChanged(self, item):
    ship = self.currentTableShip()
    if ship is None:
        return
    column = self.tableWidget.currentColumn()
    if column == ships.NAME:
        ship.name = item.text().trimmed()
    elif column == ships.OWNER:
        ship.owner = item.text().trimmed()
    elif column == ships.COUNTRY:
        ship.country = item.text().trimmed()
    elif column == ships.DESCRIPTION:
        ship.description = item.text().trimmed()
    elif column == ships.TEU:
        ship.teu = item.text().toInt()[0]
    self.ships.dirty = True
    self.populateList()
    self.populateTree()
```

如果用户在表格中编辑某项，就可以获得相应的 ship 并更新对应的属性值。使用 `QString.trimmed()`，可以去除开头和结尾处的任何空白字符<sup>①</sup>。由于编辑操作已经完成了表格的更新，所以再不需要对表格做任何操作，故而只需要对列表和树进行重新组装即可。对于小型数据集（顶多就是数百项）来说，这样重新组装的效果是不错的，不过，对于大型数据集来说，这样做就会显得非常慢。针对这一问题的解决方案是，只有当发生了改变的那些项在窗口部件中可见时，才进行更新操作。如果使用的是一个带有视图窗口部件的自定义模型，就可以自动完成前述要求，这在下一节中将可以看到。

```
def currentTableShip(self):
    item = self.tableWidget.item(self.tableWidget.currentRow(), 0)
    if item is None:
        return None
    return self.ships.ship(item.data(Qt.UserRole).toLongLong()[0])
```

`QTableWidget.item()`方法可以根据给定的行和列的值返回一个表格项。我们总是希望能够获得当前行和第一列的项，因为在这些项中，会保存每一列所对应的 ship 的 ID。

然后会使用 `ShipContainer.ship()`方法来获取给定 ID 的 ship。这会非常快，因为这些 ship 是保存在字典中的，而字典的键就是它们的 ID。

```
def addShip(self):
    ship = ships.Ship(" Unknown", " Unknown", " Unknown")
    self.ships.addShip(ship)
    self.populateList()
    self.populateTree()
    self.populateTable(id(ship))
    self.tableWidget.setFocus()
    self.tableWidget.editItem(self.tableWidget.currentItem())
```

新增一个 ship 是比较简单的，在某种程度上是因为这里没有做任何验证。只是用一些“未知”的值（开头的空格字符会让这些值格外凸显）来创建一个新的 ship，并将该 ship 添加到了 ships 字典中，包括刚刚创建的那个。将这个新 ship 的 ID 传给组装表格的方法来确保表格的第一列是当前的、选中的表格项，并将键盘焦点给它。`editItem()`的调用从程序语义上来说相当于按下了 F2 键或者是相当于鼠标双击来开始编辑操作，这样就可以让第一个域，即 ship 的名字 name 成为可编辑状态。用户只需按下 Tab 键就可以编辑那些剩余的域，因为在离开该行或者按下 Enter 回车键（或者按下 Esc 键取消）之前，都会保留为可编辑的状态。

```
def removeShip(self):
    ship = self.currentTableShip()
    if ship is None:
        return
    if QMessageBox.question(self, "Ships - Remove",
                           QString("Remove %1 of %2/%3?").arg(ship.name) \
                               .arg(ship.owner).arg(ship.country),
                           QMessageBox.Yes|QMessageBox.No) == QMessageBox.No:
        return
    self.ships.removeShip(ship)
    self.populateList()
    self.populateTree()
    self.populateTable()
```

<sup>①</sup> `QString.simplified()`方法用起来也非常方便。它可以移除最后的空格并将内部的一个或者多个空白字符简化成一个空格。

对 ship 的移除甚至比添加它们还要简单。在获得当前的 ship 后会弹出一个消息框，询问用户来让他们确信自己的确是希望能够移除该 ship 的。如果用户点击了 Yes 按钮，就从 Ship-Container 中移除该 ship 并重新组装该视图窗口部件。

尽管像这里所做的那样使用了三种不同的视图，显得有些不同寻常，但这里所用到的这些技术，特别是 QTableWidgetItem 的使用，则是相当常见的。

对于小型和临时性数据集来说，这些简便窗口部件都非常有用，也可以用在非单独数据集中——数据自身的显示、编辑和存储。在这个例子中，之所以选择将数据分离是为了能够对模型/视图技术，特别是下一节将要讲到的自定义模型技术，做一些铺垫。

## 14.2 创建自定义模型

在这一节，将会创建一个自定义模型来保存船只的数据，并会用两种不同的表格视图形式来显示同样的模型。在图 14.3 中，给出的是一个充分使用了该模型的应用程序。用户可以分别滚动该表格，也可以分别编辑各自的数据，安全起见，所做的任何修改都会在两个视图中自动予以反映。

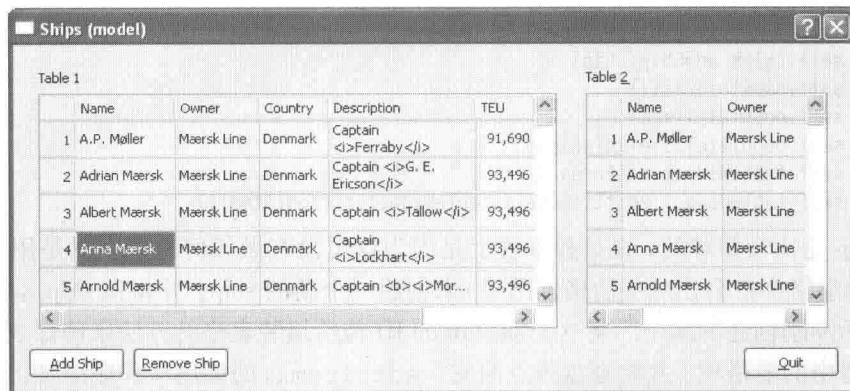


图 14.3 用于两个 QTableView 中的自定义表格模型

这里会从该应用程序主窗口提取的代码开始。这里将会给出一些模型/视图 API 的用法。然后，将会看到该模型的实现方法。PyQt 的模型/视图架构的一个非常重要的好处是，同样的编码模式可以一次次地多次使用，因此，一旦知道了要该如何创建一个表格模型，就可以知道该如何创建其他的任何表格(或者列表)模型了。

这个模型是由文件 chap14/ships.py 中的类 ShipTableModel 提供的，应用程序放在文件 chap14/ships-model.pyw 中。通过对前景色和背景色的设置，可以对视图中对数据的外在表现形式进行优化，不过，通过对各个表格项调用适当的方法，这些工作本来是可以在简便视图中完成的。在上一个例子中存在的一些问题，尤其是，对于 owner 和 country 没有使用下拉组合框，对于 TEU 没有使用微调框，而且对于备注说明中的 HTML 文本的显示还相当粗糙。而只需要通过使用将在下一节中介绍的委托(delegate)，这些问题都可以得到解决。

### 14.2.1 实现视图逻辑

表面上看，使用了简便视图内置模型的简便视图与使用了单独模型的纯视图，两者的最终

实现看起来好像并没有什么区别。在前一个例子中，就曾经用三个视图来表达同一份底层数据，并由我们自己来保持它们三个的同步。在这个例子中，将会对同一份数据使用两种视图，并且会将同步工作留给 PyQt，因为这两个视图使用的是同一个模型。另一个好处是，这些视图只会对那些实际看到的或者编辑过的数据进行获取和存储，在使用大数据集的时候，这样做就可以获得相当可观的性能表现。

先从该窗口初始化程序中提取的代码开始。

```
class MainForm(QDialog):
    def __init__(self, parent=None):
        super(MainForm, self).__init__(parent)
        self.model = ships.ShipTableModel(QString("ships.dat"))
        tableLabel1 = QLabel("Table &1")
        self.tableView1 = QTableView()
        tableLabel1.setBuddy(self.tableView1)
        self.tableView1.setModel(self.model)
        tableLabel2 = QLabel("Table &2")
        self.tableView2 = QTableView()
        tableLabel2.setBuddy(self.tableView2)
        self.tableView2.setModel(self.model)
```

首先，会创建一个新的模型。然后，会创建两个带有标签的表格视图，带上标签可以使导航切换方便些。每个表格视图所应用的模型都一样。这里忽略了那些关于布局的代码，因为它们与主题不大相关。

```
for tableView in (self.tableView1, self.tableView2):
    header = tableView.horizontalHeader()
    self.connect(header, SIGNAL("sectionClicked(int)"),
                self.sortTable)
self.connect(addShipButton, SIGNAL("clicked()"), self.addShip)
self.connect(removeShipButton, SIGNAL("clicked()"),
            self.removeShip)
self.connect(quitButton, SIGNAL("clicked()"), self.accept)
self.setWindowTitle("Ships (model)")
```

在使用自定义模型时，必须靠自己处理排序。可以把每个表格视图的水平（列）标题连接到 `sortTable()` 方法上。其他各个连接与之前的连接类似。不过，值得注意的是，在编辑一个表格项时，还没有相应的连接：这没什么必要，因为该视图将会替我们处理编辑操作，自动向模型反映这些编辑操作，这样就可以让各个视图能够时刻保持为最新状态。

```
def accept(self):
    if self.model.dirty and \
        QMessageBox.question(self, "Ships - Save?",
                             "Save unsaved changes?",
                             QMessageBox.Yes|QMessageBox.No) == QMessageBox.Yes:
        try:
            self.model.save()
        except IOError, e:
            QMessageBox.warning(self, "Ships - Error",
                               "Failed to save: %s" % e)
    QDialog.accept(self)
```

如果用户终止应用程序并且存在有尚未保存的修改，在退出应用程序之前可以给用户一个保存的机会。该模型的 `dirty` 属性以及它的 `save()` 方法都是我们自己对 `QAbstractTableModel` 的 API 的扩展，以便该模型可以从文件加载或者向文件保存它的数据。

用于模型的基类是 `QAbstractItemModel`，不过，基于行/列模型的基类通常会是 `QAbstractTableModel`，这是 `QAbstractItemModel` 的一个子类。

```
def sortTable(self, section):
    if section in (ships.OWNER, ships.COUNTRY):
        self.model.sortByCountryOwner()
    else:
        self.model.sortByName()
    self.resizeColumns()
```

这里只提供了两种排序方法，不过也没有必要知道为何不支持更多的排序。再说，`sortBy*` () 方法都是些已经添加到标准 API 中的扩展方法。在用户排序时，将会有机会来改变各个列的尺寸大小。之所以这么做是因为，编辑操作可能会改变各列所需的宽度，并且由于排序会一定程度上改变视图，不打搅用户地做些尺寸大小的改变貌似也是合理的。

```
def resizeColumns(self):
    for tableView in (self.tableView1, self.tableView2):
        for column in (ships.NAME, ships.OWNER, ships.COUNTRY,
                       ships.TEU):
            tableView.resizeColumnToContents(column)
```

这里选择对这两个表格视图中除备注说明列 `description` 之外的每一列都改变其尺寸大小。

```
def addShip(self):
    row = self.model.rowCount()
    self.model.insertRows(row)
    index = self.model.index(row, 0)
    tableView = self.tableView1
    if self.tableView2.hasFocus():
        tableView = self.tableView2
    tableView.setFocus()
    tableView.setCurrentIndex(index)
    tableView.edit(index)
```

新增一条船的操作与上一节类似，不过会更简洁些。在模型中的最后一行的后面，增加一个新行。然后，获得指向新行第一列的模型索引值。接着，找出是哪一个表格视图拥有（或者最后曾拥有）键盘焦点的，再把焦点设置到该视图上。设置视图的索引值，使其指向获得的模型索引值并为其开启编辑操作。

这里的 `rowCount()`、`insertRows()` 以及 `index()` 方法都是标准 `QAbstractTableModel` API 的一部分。

```
def removeShip(self):
    tableView = self.tableView1
    if self.tableView2.hasFocus():
        tableView = self.tableView2
    index = tableView.currentIndex()
    if not index.isValid():
        return
    row = index.row()
    name = self.model.data(
        self.model.index(row, ships.NAME)).toString()
    owner = self.model.data(
        self.model.index(row, ships.OWNER)).toString()
    country = self.model.data(
        self.model.index(row, ships.COUNTRY)).toString()
    if QMessageBox.question(self, "Ships - Remove",
                           QString("Remove %1 of %2/%3?").arg(name).arg(owner) \
```

```

    .arg(country),
    QMessageBox.Yes | QMessageBox.No) == QMessageBox.No:
    return
self.model.removeRows(row)
self.resizeColumns()

```

如果用户点击了 Remove 按钮，获得用于当前表格视图的当前项的模型索引值。从这个模型索引值中提取该行，并将其用于 `QAbstractTableModel.data()` 方法来获得船只的名字、拥有者和国家名。`data()` 方法会以模型索引值为强制参数并可以返回一个 `QVariant`。使用 `QAbstractTableModel.index()` 可以创建想要的行/列组合的模型索引值，而使用 `QVariant.toString()` 可以将返回的各个值转换成 `QString`。

如果用户确认了删除操作，只需从模型中移除相关行。模型将会自动通知各视图，从而会自动自己。之前曾经添加过一个对 `resizeColumns()` 的调用，因为删除操作后，最大列宽或许已经发生了改变。

### 14.2.2 实现自定义模型

之前曾看到过 `QAbstractTableModel` API 的用法，也看到过一些我们自己的扩展。在一个模型子类中的这些方法可以分成三类：

- 对实现只读模型很有必要的方法
- 对实现可编辑模型很有必要的方法
- 对某些特定情况需要扩展 API 的方法。

用于只读表格模型的必要方法有 `data()`、`rowCount()` 以及 `columnCount()`，尽管也经常会出现 `headerData()`。

可编辑模型也需要重新实现那些用于只读模型中的同样方法，此外，还有 `flags()` 和 `setData()`。如果模型要支持行的添加和移除，也支持对已有数据的编辑，还需要实现 `insertRows()` 和 `removeRows()`。

还可以实现一些其他方法，不过在之前的两个段落中所给出的这些方法，都是必须得实现的方法。

对于 ship 模型，要能够把 ship 保存在内存中的一个列表中，以及将其保存在磁盘上的二进制文件中。为了支持这一功能，这里通过增加 `sortByNome()`、`sortByCountryOwner()`、`load()` 以及 `save()` 来扩展模型的 API。

这里的 `ShipTableModel` 在文件 `chap14/ships.py` 中。

```

class ShipTableModel(QAbstractTableModel):
    def __init__(self, filename=QString()):
        super(ShipTableModel, self).__init__()
        self.filename = filename
        self.dirty = False
        self.ships = []
        self.owners = set()
        self.countries = set()

```

希望能够从二进制文件加载或者保存模型的数据，因而会让一个实例变量与该文件名相连。保存在列表中的 ship 数据自身起初是无须的。也会用到两个集，一个集用于 owner，另一个集用于 country：在下一节，当创建一个自定义委托时，会用这两个集来构成组合框。

```

def rowCount(self, index=QModelIndex()):
    return len(self.ships)

def columnCount(self, index=QModelIndex()):
    return 5

```

行数和列数的提供很简单。这对具有固定列数的表格模型来说很常见。

```

def data(self, index, role=Qt.DisplayRole):
    if not index.isValid() or \
        not (0 <= index.row() < len(self.ships)):
        return QVariant()
    ship = self.ships[index.row()]
    column = index.column()
    if role == Qt.DisplayRole:
        if column == NAME:
            return QVariant(ship.name)
        elif column == OWNER:
            return QVariant(ship.owner)
        elif column == COUNTRY:
            return QVariant(ship.country)
        elif column == DESCRIPTION:
            return QVariant(ship.description)
        elif column == TEU:
            return QVariant(QString("%L1").arg(ship.teu))

```

`data()`方法有一个强制参数——关于该项的模型索引值——还有一个可选参数——“角色”(`role`)。角色用来说明所需的信息类型。默认角色`Qt.DisplayRole`意味着该数据会以期望的方式进行显示。

如果模型索引值非法或者如果行超出范围，会返回一个无效的`QVariant`。PyQt 的模型/视图架构不会抛出异常或者给出错误消息；它只会使用无效的`QVariant`。如果该索引值有效，获得船只(`ship`)所在行所对应的索引值的行。如果角色是`Qt.DisplayRole`，就给所需列返回`QVariant`型数据。以`TEU`为例，返回的不是整数，而是以本地化的字符串形式返回该数字。

```

    elif role == Qt.TextAlignmentRole:
        if column == TEU:
            return QVariant(int(Qt.AlignRight|Qt.AlignVCenter))
        return QVariant(int(Qt.AlignLeft|Qt.AlignVCenter))
    elif role == Qt.TextColorRole and column == TEU:
        if ship.teu < 80000:
            return QVariant(QColor(Qt.black))
        elif ship.teu < 100000:
            return QVariant(QColor(Qt.darkBlue))
        elif ship.teu < 120000:
            return QVariant(QColor(Qt.blue))
        else:
            return QVariant(QColor(Qt.red))
    elif role == Qt.BackgroundColorRole:
        if ship.country in (u"Bahamas", u"Cyprus", u"Denmark",
                            u"France", u"Germany", u"Greece"):
            return QVariant(QColor(250, 230, 250))
        elif ship.country in (u"Hong Kong", u"Japan", u"Taiwan"):
            return QVariant(QColor(250, 250, 230))
        elif ship.country in (u"Marshall Islands",):
            return QVariant(QColor(230, 250, 250))
        else:
            return QVariant(QColor(210, 230, 230))
    return QVariant()

```

data()会以 Qt.TextAlignmentRole 的形式进行调用，返回一个右对齐的 TEU，而返回的其他列则会是左对齐。 QVariant 不能接受对齐形式，所以必须将这些对齐形式转换成整数值。

对于 Qt.TextColorRole，会给 TEU 列返回一种颜色，忽略其他各列。这就意味着非 TEU 列将会使用默认的文本颜色——通常是黑色。对于 Qt.BackgroundColorRole 来说，会根据船只所属的国家组别的不同而提供不同的背景颜色。

如果愿意，还可以处理其他几种角色，包括 Qt.DecorationRole(项的图标)、Qt.ToolTipRole、Qt.StatusTipRole 以及 Qt.WhatsThisRole。同时，为了控制外观，除了对齐和之前讨论过的颜色角色之外，还有 Qt.FontRole 和 Qt.CheckStateRole。

对于没有选择处理的其他所有情况，都返回一个无效的 QVariant。这就会告诉模型/视图架构，对于这些情况使用默认值即可。

一些程序开发人员不喜欢将外观与数据的信息相互混合，就像这里在 data() 的实现中所做的那样。PyQt 对于这一情况持中立态度：为混合外观提供了足够的灵活性，但如果更喜欢 data() 仅仅是用做数据的处理也是可以的，就会将和外观相关的所有问题都留给委托。

```
def headerData(self, section, orientation, role=Qt.DisplayRole):
    if role == Qt.TextAlignmentRole:
        if orientation == Qt.Horizontal:
            return QVariant(int(Qt.AlignLeft|Qt.AlignVCenter))
        return QVariant(int(Qt.AlignRight|Qt.AlignVCenter))
    if role != Qt.DisplayRole:
        return QVariant()
    if orientation == Qt.Horizontal:
        if section == NAME:
            return QVariant("Name")
        elif section == OWNER:
            return QVariant("Owner")
        elif section == COUNTRY:
            return QVariant("Country")
        elif section == DESCRIPTION:
            return QVariant("Description")
        elif section == TEU:
            return QVariant("TEU")
    return QVariant(int(section + 1))
```

尽管并非必要，提供 headerData() 的实现还是非常不错的工程实践做法。当 orientation 是 Qt.Vertical 时，section 会是行偏移，而当 orientation 是 Qt.Horizontal 时，则会是列偏移。在这里，提供了列标题，而把行从 1 开始进行编号。

与 data()一样，这个方法也会接受一个角色，可以使用这个角色来让行编号右对齐并让列标题左对齐。

这个方法截至目前所看到的内容已经足以实现只读表格模型的处理。现在，将会看到一些额外的方法，要让一个模型可编辑，还是必须要实现这些方法的。

```
def flags(self, index):
    if not index.isValid():
        return Qt.ItemIsEnabled
    return Qt.ItemFlags(QAbstractTableModel.flags(self, index) |
        Qt.ItemIsEditable)
```

如果有有效的模型索引值，就返回一个 Qt.ItemFlag，会将已有的项标识符与 Qt.ItemIsEditable 标识符进行结合。只在模型索引值是用于打算让行和列为可编辑时，通过应用 Qt.ItemIsEditable 标识符，才可以使用这个方法让这些项只读。

```

def setData(self, index, value, role=Qt.EditRole):
    if index.isValid() and 0 <= index.row() < len(self.ships):
        ship = self.ships[index.row()]
        column = index.column()
        if column == NAME:
            ship.name = value.toString()
        elif column == OWNER:
            ship.owner = value.toString()
        elif column == COUNTRY:
            ship.country = value.toString()
        elif column == DESCRIPTION:
            ship.description = value.toString()
        elif column == TEU:
            value, ok = value.toInt()
            if ok:
                ship.teu = value
        self.dirty = True
        self.emit(SIGNAL("dataChanged(QModelIndex,QModelIndex)"),
                  index, index)
    return True
return False

```

当用户完成编辑时，才会调用这个方法。这个例子中忽略了角色，尽管拥有单独的数据显示和编辑也是可能的（例如，电子数据表的结果及其蕴含的公式）。如果索引值有效且是在有效范围内的行，可以获得相应的船只并更新已经编辑过的列。以 TEU 为例，只有当用户键入的内容成功转换成整数时才会应用所做的修改。

如果发生了修改，就必须发射 `dataChanged()` 信号。这一模型/视图架构能够根据这个信号来确保所有的视图都保持为最新的状态。必须传递两次修改项的模型索引值，因为这个信号可以用来自说明有许多变化，第一个索引值用做左上角的项，第二个索引值用来自说明右下角的项。如果接受或者应用了修改，就必须返回 `True`，否则，就必须返回 `False`。

实现 `flags()` 和 `setData()`（对于只读模型来说，除了必要的方法之外）就足以让一个模型成为一个可编辑的模型。不过，为了使用户有可能添加或者删除多个整行，还需要额外实现两个方法。

```

def insertRows(self, position, rows=1, index=QModelIndex()):
    self.beginInsertRows(QModelIndex(), position,
                        position + rows - 1)
    for row in range(rows):
        self.ships.insert(position + row,
                          Ship("Unknown", "Unknown", "Unknown"))
    self.endInsertRows()
    self.dirty = True
    return True

```

在打算向一个模型中插入一行或者多行时，对 `beginInsertRows()` 的调用是必要的。这里的 `position` 就是打算插入的行的位置。对 `beginInsertRows()` 的调用是直接从 PyQt 文档中摘取的，对于任意表格模型的 `insertRows()` 实现都应当是无须修改的。插入后，就必须调用 `endInsertRows()`。模型将会自动通知各个视图已经发生了修改，如果对用户有可见的相关行，视图就会请求获得新的数据。

```

def removeRows(self, position, rows=1, index=QModelIndex()):
    self.beginRemoveRows(QModelIndex(), position,
                        position + rows - 1)

```

```

self.ships = self.ships[:position] + \
            self.ships[position + rows:]
self.endRemoveRows()
self.dirty = True
return True

```

这个方法与前面的方法类似。对 `beginRemoveRows()` 的调用是直接从 PyQt 文档中摘取的，是用于表格模型的一个标准实现。在移除相关行之后，就必须调用 `endRemoveRows()` 了。模型将会把这些改变自动通知给各个视图。

现在就已实现了用于可编辑表格模型的必要方法。一些模型只是作为诸如数据库表格（参见下一章）这样的外部数据资源的接口，也可能是作为外部文件或者进程的接口。在这个例子中，由于是在模型自身中保存的数据，基于此，必须额外提供一些方法，特别是 `load()` 和 `save()`。还提供了一系列排序方法来为用户提供方便。对于大型数据集来说，排序的代价非常大，对于此类情况，可以使用有序型数据结构，比如 `OrderedDict`，或者是采用列表与 `bisect` 模块中的函数相结合的方法，这样可能会更有效些。

```

def sortByName(self):
    self.ships = sorted(self.ships)
    self.reset()

```

在对列表应用 `sort()` 调用时，对于比较，`sort()` 会使用该项的 `__lt__()` 特殊方法，但如果还没有实现 `__lt__()` 方法时，可以重新使用 `__cmp__()` 特殊方法。这里提供了 `Ship.__cmp__()` 方法来对 `ship` 的名字进行本地化方式的比较。

表 14.1 QAbstractItemModel 的部分方法

语法	说明
<code>m.beginInsertRows(p, f, l)</code>	要在 <code>insertRows()</code> 的实现代码中插入数据之前调用。各参数分别是父对象 <code>QModelIndex p</code> 、新行中各行所要占用的第一行和最后一行的行号； <code>m</code> 是 <code>QAbstractItemModel</code> 的一个子类
<code>m.beginRemoveRows(p, f, l)</code>	要在 <code>removeRows()</code> 的实现代码中移除数据之前调用。各参数分别是父对象 <code>QModelIndex p</code> 、要移除的第一行和最后一行的行号； <code>m</code> 是 <code>QAbstractItemModel</code> 的一个子类
<code>m.columnCount(p)</code>	各个子类都必须重新实现这个方法；父对象 <code>QModelIndex p</code> 必须是树模型
<code>m.createIndex(r, c, p)</code>	各个子类都必须使用这个方法来创建带有行为 <code>int r</code> 、列为 <code>int c</code> 且父对象为 <code>QModelIndex p</code> 的 <code>QModelIndex</code>
<code>m.data(i, rl)</code>	返回 <code>QVariant</code> 型数据，可用于 <code>QModelIndex i</code> 和 <code>Qt.ItemDataRole rl</code> ；各个子类都必须重新实现这个方法
<code>m.endInsertRows()</code>	要在 <code>insertRows()</code> 的实现代码中插入新的数据之后调用； <code>m</code> 是 <code>QAbstractItemModel</code> 的一个子类
<code>m.endRemoveRows()</code>	要在 <code>removeRows()</code> 的实现代码中移除数据之后调用； <code>m</code> 是 <code>QAbstractItemModel</code> 的一个子类
<code>m.flags(i)</code>	为 <code>QModelIndex i</code> 返回 <code>Qt.ItemFlags</code> ；这些状态标识可以管理项的可选性、可编辑性，等等。各个可编辑型模型子类都必须重新实现这个方法
<code>m.hasChildren(p)</code>	如果父对象 <code>QModelIndex p</code> 有子对象，返回 <code>True</code> ；只有对树模型，这个方法才会有意义
<code>m.headerData(s, o, rl)</code>	为“片段”（行或者列） <code>int s</code> 返回一个 <code>QVariant</code> 参数，用 <code>Qt.Orientation o</code> 说明行或者列方向，用 <code>Qt.ItemDataRole rl</code> 说明行或者列的角色。各个子类通常都会重新实现这个方法； <code>m</code> 是 <code>QAbstractItemModel</code> 的一个子类

(续表)

语法	说明
m.index(r, c, p)	在给定行 int r、列 int c 以及父对象 QModelIndex p 后，返回 QModelIndex；各个子类都必须重新实现这个方法并且必须使用 createIndex()
m.insertRow(r, p)	在行 int r 之前插入一个新行。在树模型中，插入行会成为父对象 QModelIndex p 的一个子对象
m.insertRows(r, n, p)	在行 int r 之前插入 int n 个行。在树模型中，插入的这些行都会成为父对象 QModelIndex p 的子对象。各个可编辑型子类通常都会重新实现这个方法——重新实现过程中都要调用 beginInsertRows() 和 endInsertRows()
m.parent(i)	返回 QModelIndex i 的父对象 QModelIndex。各个树模型子类都必须重新实现这个方法
m.removeRow(r, p)	移除行 int r。父对象 QModelIndex p 只与树模型相关；m 是 QAbstractItemModel 的一个子类
m.removeRows(r, n, p)	从行 int r 开始移除 int n 个行。父对象 QModelIndex p 只与树模型相关。各个可编辑型模型子类通常都会重新实现这个方法——重新实现过程中都必须调用 beginRemoveRows() 和 endRemoveRows()
m.reset()	通知所有相关的视图，模型的数据已经彻底发生了改变——这就可以强制各视图重新获取它们自己可见的数据
m.rowCount(p)	各个子类都必须重新实现这个方法；父对象 QModelIndex p 只与树模型相关
m.setData(i, v, rl)	把 QModelIndex i 用于 Qt.ItemDataRole rl 的数据设置成 QVariant v。各个可编辑型模型子类都必须重新实现这个方法——重新实现过程中，如果数据确实发生了改变，都必须发射 dataChanged() 信号
m.setHeaderData(s, o, v, rl)	把“片段”int s 的标题数据设置为 Qt.Orientation o(也就是说，行或者列)，把 Qt.ItemDataRole rl 设置成 QVariant v

对数据进行排序会让所有模型索引值无效，也就是说，各个视图现在都会显示出错误的数据。模型必须通知各个视图去获取新的数据，以便可以更新自己。为做到这一点，一种方法是发射一个 dataChanged() 信号，不过对于存在大量修改的情况，使用 QAbstractTableModel.reset() 会更有效率些；这就会告诉所有相关视图的内容不是最新的，需要强制更新了。

```
def sortByCountryOwner(self):
    def compare(a, b):
        if a.country != b.country:
            return QString.localeAwareCompare(a.country, b.country)
        if a.owner != b.owner:
            return QString.localeAwareCompare(a.owner, b.owner)
        return QString.localeAwareCompare(a.name, b.name)
    self.ships = sorted(self.ships, compare)
    self.reset()
```

这里提供了一个自定义排序方法，可以根据 country、owner 以及 ship 的 name 进行排序。对于大型数据集使用 DSU (Decorate-Sort-Undecorate，封装-排序-解封) 可能会更有效率些<sup>①</sup>。例如：

```
def sortByCountryOwner(self):
    ships = []
    for ship in self.ships:
```

<sup>①</sup> DSU 方法不会创建自定义的排序方法，而是创建一个辅助的排序列表，然后对这个列表进行默认排序。需要说明的是，DSU 是一种比较老的方法，现在已经基本上不使用了——译者注。

```

    ships.append((ship.country, ship.owner, ship.name, ship))
    ships.sort()
    self.ships = [ship for country, owner, name, ship in ships]
    self.reset()

```

这里会使用标准的 `QString.compare()`, 所以最好还是能够使用 `unicode(ship.country)`、`unicode(ship.owner)` 和 `unicode(ship.name)`。当然, 对于特别大的数据集来说, 或许最好是能够避免总是进行排序, 相反, 可以使用有序容器。

`save()` 和 `load()` 方法与之前使用 `QDataStream` 处理二进制数据时所使用的这两个同名方法非常相似, 所以这里只会摘取两者各自最为核心的内容, 先从 `save()` 方法开始。

```

for ship in self.ships:
    stream << ship.name << ship.owner << ship.country \
        << ship.description
    stream.writeInt32(ship.teu)

```

正是使用了 `QDataStream` 而不必让我们再担心字符串的长度是有多长或者有关编码方面的问题了。

会载入相应的 `ship`: 这里是从 `load()` 方法中提取的部分代码:

```

self.ships = []
while not stream.atEnd():
    name = QString()
    owner = QString()
    country = QString()
    description = QString()
    stream >> name >> owner >> country >> description
    teu = stream.readInt32()
    self.ships.append(Ship(name, owner, country, teu,
                          description))
    self.owners.add(unicode(owner))
    self.countries.add(unicode(country))

```

正如之前提到过的那样, 保留 `owner` 和 `country` 的集可以在添加自定义委托的时候使两者能够用于组合框中。

实现自定义模型, 尤其是列表和表格模型, 都相当简单。对于只读型模型, 只需要实现三个方法, 尽管通常实际上会实现 4 种方法。对于可编辑行模型, 通常一共会实现 8 种方法。一旦创建了一些模型, 再创建其他模型就很简单了, 因为所有的列表和表格模型都会遵循同样的实现模式。树模型的实现要稍稍具有一定的挑战性; 有关这一方面的内容会在第 16 章的最后一节介绍。

### 14.3 创建自定义委托

如果希望能够练习完全控制数据项的表达和编辑, 就必须创建自定义委托。委托(`delegate`)可以纯粹用来控制外观(例如, 对于那些只读型视图)或者通过提供自定义编辑器用来完全控制编辑操作, 又或者用于这两个方面。

图 14.4 与之前的几个截屏效果图类似, 只有一个显著不同在于, 说明文本(`Description`)得到了适当的格式化而不再是粗糙的 HTML 文本。然而, 不同之处远非如此。例如, 如果编辑 `owner` 或者 `country` 域, 将会可以看到组合框已经与当前的 `owner` 和 `country` 域组合到了一起, 且若编辑 `TEU` 将会用到一个微调框。通过使用委托, 所有这些对外观和编辑的控制都可以实

现出来——委托也可以用于简便视图或者纯视图中，尽管在这个例子中，只是将委托用在了几个纯视图中。

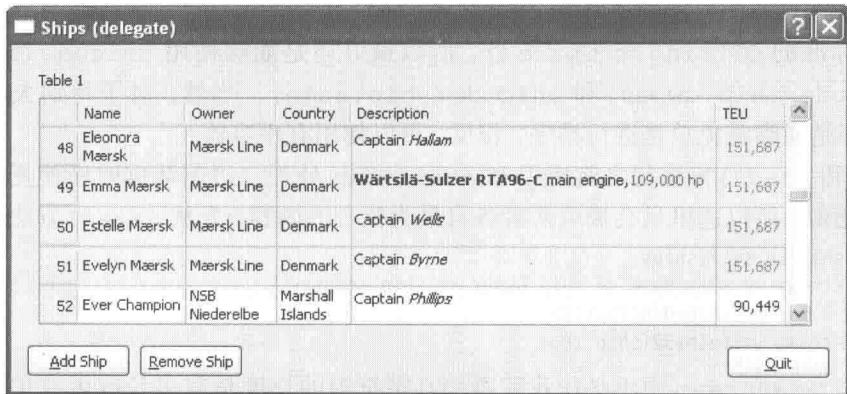


图 14.4 使用中的自定义委托

在这一节，要用到的应用程序放在了文件 `chap14/ships-delegate.pyw` 中。这个应用程序与 `ships-model.pyw` 一样，只是在窗口标题上存在些差异，这是因为实际上是重新调整了所有各列的尺寸大小而不是仅仅简单跳过说明 (Description) 列，并且实际上也使用了自定义委托。委托类 `ShipDelegate` 放在文件 `chap14/ships.py` 中。值得注意的是，这个类需要基于 PyQt 4.1 或者后续的版本。

与模型的一些子类一样，委托也遵循一定的模式。以用于只读型模型的委托为例，唯一必须重新实现的方法是 `paint()`。对于可编辑型模型，必须重新实现 `createEditor()`、`setEditorData()` 以及 `setModelData()` 方法。如果在编辑过程中用到了 `QLineEdit` 或者 `QTextEdit`，通常也会重新实现 `commitAndCloseEditor()`。最后，某些时候也有必要重新实现 `sizeHint()`，很快就会看到这一点。

创建仅能处理某些列的委托的做法相当常见，特别是当将其用做处理绘制时，这样做就足以不使用基类的默认行为来处理这些列。

先从主窗体构造函数中抽取的一小段代码开始，看看是如何创建第一个表格的：

```
class MainForm(QDialog):
    def __init__(self, parent=None):
        super(MainForm, self).__init__(parent)
        self.model = ships.ShipTableModel(QString("ships.dat"))
        tableLabel1 = QLabel("Table &1")
        self.tableView1 = QTableView()
        tableLabel1.setBuddy(self.tableView1)
        self.tableView1.setModel(self.model)
        self.tableView1.setItemDelegate(ships.ShipDelegate(self))
```

与上一节的唯一不同之处在于，这里调用的是 `setItemDelegate()`，并为其传递了一个新创建的 `ship.ShipDelegate`。该委托必须以给定的窗体作为父对象，以便在使用到该窗体时能够使其保持激活状态。对于第二个表格的代码也是一样的，使用同样的模型设置，只不过它会使用自己的 `ship.ShipDelegate`。这也是唯一需要变更的地方——现在，数据所有的展示和编辑操作工作都可以由委托来完成。

```
class ShipDelegate(QItemDelegate):  
    def __init__(self, parent=None):  
        super(ShipDelegate, self).__init__(parent)  
  
    def paint(self, painter, option, index):  
        if index.column() == DESCRIPTION:  
            text = index.model().data(index).toString()  
            palette = QApplication.palette()  
            document = QTextDocument()  
            document.setDefaultFont(option.font)  
            if option.state & QStyle.State_Selected:  
                document.setHtml(QString("<font color=%1>%2</font>") \  
                                .arg(palette.highlightedText().color().name()) \  
                                .arg(text))  
            else:  
                document.setHtml(text)  
            color = palette.highlight().color() \  
            if option.state & QStyle.State_Selected \  
            else QColor(index.model().data(index,  
                                         Qt.BackgroundColorRole))  
            painter.save()  
            painter.fillRect(option.rect, color)  
            painter.translate(option.rect.x(), option.rect.y())  
            document.drawContents(painter)  
            painter.restore()  
        else:  
            QItemDelegate.paint(self, painter, option, index)
```

对于纯文本字符串来说，数字、日期，等等，基类的 `QItemDelegate.paint()` 方法都可以相当好地予以处理，所以很多时候都根本不会重新实现它。然而，在这个例子中，`Description` 列中含有 HTML，是需要由我们自己进行处理的。

对 `paint()` 方法的调用会使用一个用来绘制的绘制器，也会使用一个包含了信息不同片段的 `QStyleOptionViewItem`，其中包括一个在其中进行绘制的矩形，还会使用一个要绘制的项的模型索引值。

这里先从使用模型的 `data()` 方法来获取 HTML 文本开始，其中会用模型中设置的 `Qt.DisplayRole` 默认值作为第二个参数。值得注意的是，一个模型索引值可以提供一个用 `QModelIndex.model()` 方法引用的模型索引。

接下来，会获取应用程序的调色板——可基于用户主题颜色的偏好。如果该项是选中的，对 HTML 应用调色板的高亮文本色；否则，使用 HTML“原来的颜色”。`QColor.name()` 方法可以用十六进制字符串的形式返回颜色；例如，红色可能会返回成字符串“#FF0000”，这一格式与 HTML 颜色规范的格式一致。相应地，如果项是选中的，就会使用该调色板的高亮背景色；否则，通过调用带有 `Qt.BackgroundColorRole` 的 `data()` 方法，可以使用模型所给定的背景色。

`QTextDocument.drawContents()` 方法的绘制会从相对于绘制器的左上角坐标(0, 0)开始。基于此，可以把绘制器的左上角移动（或者变换）到风格选项矩形的位置( $x, y$ )处，然后将文档绘制在绘制器上。

很多情况下，在不同的绘制事件之间，不必担心绘制器状态的保存和恢复，不过在这个例

子中则是需要的。一些 Qt 编程开发人员认为总是保存和恢复绘制器的状态是一种良好的做法，然而也有一些人更倾向于认为，仅在必要时才那么做，也就是说，只有当他们是对绘制器的状态进行持续性修改时才有必要，比如对绘制器的状态使用了诸如平移这样的几何变换。

遗憾的是，HTML 的绘制操作并非如此简单。当视图要求一个 HTML 列提供尺寸大小提示时，默认的行为会返回一个基于该视图字体和字符数的尺寸大小提示。

因为 HTML 相当繁琐，用于上述计算过程中的字符数可能要远大于实际显示出来的字符数。

针对这一问题，有两种解决方法，但两者都需要由我们自己计算 HTML 文本的尺寸大小提示。一种方法是，修改 `QAbstractTableModel.data()` 方法，在带有 `Qt.SizeHintRole` 的 `data()` 调用时返回一个适当的尺寸大小提示。另一种解决方法是，重新实现 `QItemDelegate.sizeHint()` 方法。这里会倾向于 `sizeHint()` 的重新实现，因为这可以让问题和问题的解决方法都在同一个类中。

```
def sizeHint(self, option, index):
    fm = option.fontMetrics
    if index.column() == TEU:
        return QSize(fm.width("9,999,999"), fm.height())
    if index.column() == DESCRIPTION:
        text = index.model().data(index).toString()
        document = QTextDocument()
        document.setDefaultFont(option.font)
        document.setHtml(text)
        return QSize(document.idealWidth() + 5, fm.height())
    return QItemDelegate.sizeHint(self, option, index)
```

可选参数的类型是 `QStyleOptionViewItem`，这是 `QStyleOption` 的一个子类，拥有多个有用的属性。在这个方法中，实际会负责两个列的尺寸大小提示。对于 TEU 来说，会返回一个尺寸大小提示，其宽度要能够满足所期望处理的最大 TEU。对于 Description 来说，会根据文本的字体和字体属性，外加 5 个像素的边白，使用 `QTextDocument()` 计算出文本的“理想”宽度值。对于其他各列来说，会将这一工作传给基类。

很多时候，委托根本无法重新实现 `paint()` 方法，绘制操作就需要依靠相当不错的默认行为来完成，相反会为数据项的编辑提供一些自定义方法。

```
def createEditor(self, parent, option, index):
    if index.column() == TEU:
        spinbox = QSpinBox(parent)
        spinbox.setRange(0, 200000)
        spinbox.setSingleStep(1000)
        spinbox.setAlignment(Qt.AlignRight|Qt.AlignVCenter)
        return spinbox
    elif index.column() == OWNER:
        combobox = QComboBox(parent)
        combobox.addItems(sorted(index.model().owners))
        combobox.setEditable(True)
        return combobox
    elif index.column() == COUNTRY:
        combobox = QComboBox(parent)
        combobox.addItems(sorted(index.model().countries))
        combobox.setEditable(True)
        return combobox
    elif index.column() == NAME:
        editor = QLineEdit(parent)
```

```

        self.connect(editor, SIGNAL("returnPressed()"),
                     self.commitAndCloseEditor)
    return editor
elif index.column() == DESCRIPTION:
    editor = richtextlineedit.RichTextLineEdit(parent)
    self.connect(editor, SIGNAL("returnPressed()"),
                 self.commitAndCloseEditor)
    return editor
else:
    return QItemDelegate.createEditor(self, parent, option,
                                      index)

```

当用户开始对数据项编辑时，一般都会通过按下 F2 键或者双击来实现，视图就会要求委托为该项提供一个编辑器。对于任何不想或者不需要自行处理的项来说，只需把编辑工作传给基类即可，不过在这个委托中，更倾向于由我们自己来处理每一列。

对于 TEU 列来说，会创建和返回一个微调框。可以使用任何窗口部件，既可以是一个 QSpinBox，也可以是一个自定义编辑器，比如是上一章中创建的 RichTextLineEdit。无论是哪种情况，处理过程都是一样的：用给定的父对象创建编辑器，然后对其进行设置，再将其返回。

这里将各个组合框与排序后的列表进行了组装，并让它们能够编辑，以便让用户可以添加新的条目。如果打算让用户能够只从给定的列表选择东西，将只需要忽略 setEditable (True) 调用即可。

在 QLineEdit、QTextEdit 和其他一些类的示例中，曾用 returnPressed () 信号来说明编辑操作已经完成，这里会把该信号连接到重新实现的 commitAndCloseEditor () 方法上。

```

def commitAndCloseEditor(self):
    editor = self.sender()
    if isinstance(editor, (QTextEdit, QLineEdit)):
        self.emit(SIGNAL("commitData(QWidget*)"), editor)
        self.emit(SIGNAL("closeEditor(QWidget*)"), editor)

```

在以前，曾经总是使用内置的 isinstance () 函数来考虑某个对象与某个类相关，不过这里会提供一个两个类的元组。在用户按下回车键 (Enter) 时会调用这个方法，并且，进而会向该编辑器发射信号来告诉它要将数据保存到模型中并关闭自己。

```

def setEditorData(self, editor, index):
    text = index.model().data(index, Qt.DisplayRole).toString()
    if index.column() == TEU:
        value = text.replace(QRegExp("[., ]"), "").toInt()[0]
        editor.setValue(value)
    elif index.column() in (OWNER, COUNTRY):
        i = editor.findText(text)
        if i == -1:
            i = 0
        editor.setCurrentIndex(i)
    elif index.column() == NAME:
        editor.setText(text)
    elif index.column() == DESCRIPTION:
        editor.setHtml(text)
    else:
        QItemDelegate.setEditorData(self, editor, index)

```

一旦编辑器创建完成并传给视图，视图就会调用 setEditorData ()。这就给委托一个把编

辑器和当前数据进行组装的机会，以便为用户的编辑操作做好准备。在 TEU 这个例子中，会为用户显示文本，通常会包含一些空格、逗号或者句号。对于这些情况，就需要去除这些不想要的字符，把值转换成整数值，以及把微调框的值设置成相应的值。还有一种替换方法是，对这一列可以分别使用 `Qt.DisplayRole` 和 `Qt.EditRole` 的值。

如果编辑器是组合框，可以把它的当前索引值设置成与数据值项相匹配的项。如果没有匹配项，只需让第一个项成为当前项即可。对于用于船只的 `name` 的行编辑器可以使用 `setText()`，而对于 Rich 文本行编辑器可以使用 `setHtml()`（从 `QTextEdit` 集成而来）。像往常一样，可以将未做处理的情况传给基类，尽管这里的做法比较正式，因为所有列都是由我们自己处理的。

```
def setModelData(self, editor, model, index):
    if index.column() == TEU:
        model.setData(index, QVariant(editor.value()))
    elif index.column() in (OWNER, COUNTRY):
        model.setData(index, QVariant(editor.currentText()))
    elif index.column() == NAME:
        model.setData(index, QVariant(editor.text()))
    elif index.column() == DESCRIPTION:
        model.setData(index, QVariant(editor.toSimpleHtml()))
    else:
        QItemDelegate.setModelData(self, editor, model, index)
```

如果用户确认了他们的编辑结果，编辑器的数据就必须写回模型中。模型将会通知视图，该项已经发生改变，而显示该项的这些视图就会要求对显示的数据进行更新。

在每种情况下，只是从适当的编辑器中获取值并调用 `setData()`，并把这些值以 `QVariant` 的形式进行传递。

这样就完成了委托的内容。在下一章还会用到两个委托，它们都会给特定的域提供相应的编辑器，并且两者都只会实现一下 `createEditor()`、`setEditorData()` 和 `setModelData()`。在这一章和下一章中所用到的各自定义委托，都仅用于特定的模型。不过，在第 16 章，会有专门的一节内容，“范型委托”（generic delegates），而范型委托就可以用来创建一些任意的、不一定与模型相关的自定义委托——这样就可以降低代码的冗余量并让之后的代码维护更为轻松些。

## 小结

PyQt 的简便项视图窗口部件，如 `QListWidget`、`QTableWidget` 和 `QTreeWidget`，对于小型或者特定数据集的查看和编辑都非常有用。它们可以与 14.1 节中所看到的外部数据集联用，也可以在某种意义上用做自己的数据容器。添加、编辑和移除项的操作比较简单，不过如果使用一个以上的视图来显示数据集，就必须要能够负责视图和数据集的同步操作。如果对一个自定义模型使用模型/视图方法，这就不会再是什么问题了。

简便视图不提供处理项的任何编辑控制功能。这一不足很容易得到克服，无论是对简便视图还是对纯视图来说，只需使用自定义项委托即可。

纯视图提供了与简便视图相类似的功能，不过不提供排序或者直接控制数据项的外观。这些视图必须与模型联用，这里的模型要么是一个由 PyQt 提供的预定义模型，要么是，通常也更为常见，一个自定义模型。

为实现自定义表格模型，无论是只读型模型还是可编辑型模型，都必须重新实现 rowCount()、columnCount() 和 data()；通常也会重新实现 headerData()。此外，必须实现 flags() 和 setData()，以使各项可编辑，还要实现 insertRows() 和 removeRows() 来允许用户插入或者移除数据行。如果打算让用户能够对数据进行排序，可以额外添加一些有关排序的方法，尽管在数据库表格的例子中，只是添加了些 ORDER BY 语句。带模型/视图架构的数据库的用法会在下一章讲到。

创建自定义将允许我们实现对外观和数据项编辑的完全控制。在模型和委托之间共同负责数据外观，或者是将责任赋给两者之一，也都是可能的。不过，只能用自定义委托来提供编辑操作的控制功能。对于只读型委托，以及对于那些只关心数据外在表现的委托来说，通常只需要重新实现 paint() 方法即可，尽管某些情况下，还必须重新实现 sizeHint()(或者在模型的 data() 重新实现代码中处理 Qt.SizeHintRole)。对大部分委托来说，根本不需要重新实现 paint() 或者 sizeHint()，而只需要重新实现 createEditor()、setEditorData() 和 setModelData()。

在下一章将会看到一些更高级的模型/视图架构的示例，会用到纯视图、自定义委托和内置的 SQL 数据库模型等。

## 练习题

向 ShipTableModel 中添加一个新的方法 sortByTEU()。可以使用任何想用的技术；之前曾用过 DSU 技术。然后，在 MainForm.sortTable() 中使用这个方法。所有这些内容应该不会超过十来行代码。

扩展 ShipTableModel.data() 方法来提供工具栏提示。提示内容应当只是数据项的文本内容，除了用于 TEU 的提示内容之外，它的文本内容应当是(本地化的)数字后跟上“相当于 20 英尺”。值得注意的是，HTML 在工具栏提示中要能够得到正确格式。这也比较容易做到，大约需要十来行代码。

修改 ShipTableDelegate.setModelData()，以便可以用其修改 name、owner 或者 country，只是新的文本至少要有三个字符。为那些带有“( minimum of 3 characters)”文本的列扩展其工具栏提示。做到这些应该不会超过十来行代码。

添加一个 Export 按钮，在按下它时，提示用户输入.txt 后缀的文件名，使用 UTF-8 的编码形式保存这些数据，每行一条船，类似如下的形式：

name|owner|country|teu|description

要以竖杠“|”(管道符)作为分隔符。通过模型的 data() 方法应当可以访问这些数据，它们也可用 country/owner 的形式输出，在 Description 中不要有 HTML 标记，在 TEU 的输出中只能用数字(不能有逗号、句号或者空格)。最后弹出一个消息框，要么用来报告一个错误，要么用来说明成功了。使用 Python 或者 PyQt 来实现文件的写操作；之前曾使用过 PyQt。如果编程时使用的是 PyQt 4.1 之前的版本，在将 TEU 写入文本流之前需要把 TEU 转换成 QString。export() 方法可以用不超过 50 行的代码写成。

本练习题的参考答案在文件 chap14/ships\_ans.py 和 chap14/ships-delegate\_ans.py 中。

# 第 15 章 数据库

- 连接数据库
- 执行 SQL 查询
- 使用数据库窗体视图
- 使用数据库表视图

PyQt 提供了一贯的跨平台 API 来用于数据库的访问，既可使用 `QtSql` 模块，也可基于 PyQt 的模型/视图架构 (model/view architecture)<sup>①</sup>。Python 也有自己完全不同的数据库 API，称为 DB-API，但它并不需要与 PyQt 绑定，所以本章不会涉及这部分的内容。商业版的 Qt 会附带许多数据库驱动程序，而 GPL 版本的 Qt 则因许可限制等问题而相对较少。可用的数据库驱动程序包括诸如用于 IBM 的 DB2、Borland 的 Interbase、MySQL、Oracle、ODBC（用于 Microsoft 的 SQL Server）、PostgreSQL、SQLite 以及 Sybase 等。然而，正如 PyQt 的其他方面一样，如果我们所需的数据驱动程序不可用时，也可以利用 PyQt 创建出额外的数据库驱动程序来。

在利用源码包来构建 Qt 时，是可以对 Qt 进行配置来包含进 SQLite 的，这是一个得到了广泛使用的数据库。对于二进制的 Qt 包用户来说，比如那些适用于 Windows 和 Mac OS X 的 GPL 包用户们来说，SQLite 会内置在里面。在本章中的各个例子就会使用 SQLite，但除了与数据库的初始化连接这部分，是涉及 SQL 原始语法的许多方面，这些内容应该是可通用于其他的任意 SQL 数据库的。

PyQt 会从两个层面上提供对数据库的访问。高层次的数据库访问会涉及 `QSqlTableModel` 或 `QSqlRelationalTableModel` 的使用。这些模型会提供一些处理数据库表的抽象方法，它们与 `QAbstractItemModel` 的其他子类具有相同的 API，还提供了一些与数据库相关的扩展。SQL 模型可用于诸如 `QTableView` 的视图中，在这一章的最后一节中将会看到这一点，对于窗体视图来说，也可以借助 `QDataWidgetMapper`，这就是 15.2 节的主要论题。

低层次的数据库访问方法，也是最通用的方法，是基于对 `QSqlQuery` 的使用上。这个类可以接受任何形式的 DDL (data definition language，数据定义语言) 或 DML (data manipulation language，数据控制语言) SQL 语句并在数据库中予以执行。例如，可以使用 `QSqlQuery` 来创建表，以及插入、更新和删除表中的各条记录等。在 15.1 节，将会看到 `QSqlQuery` 的应用实例。

## 15.1 连接数据库

不过，在对数据库做出具体操作之前，必须要先建立与数据库之间的连接。在许多数据库应用程序中，这一步都是在创建了 `QApplication` 对象之后完成的，但要在创建或显示主窗

<sup>①</sup> 本章假定读者了解 PyQt 的模型/视图架构，在上一章讲过这部分的内容，同时它也是 SQL 的基础知识。

体的前面。其他一些应用程序会在随后建立与数据库之间的连接——例如，只有在需要用到它们的时候才去创建连接。

要使用 PyQt 的 SQL 类，就必须像下面这样先导入 QSql 模块：

```
from PyQt4.QtSql import *
```

数据库的连接是通过调用静态方法 QSqlDatabase.addDatabase() 建立的，该方法需要带有拟使用的驱动程序名称。然后，必须对一些属性进行设置，比如数据库名、用户名、密码等。最后，必须调用 open() 来完成连接操作。

```
db = QSqlDatabase.addDatabase("QSQLITE")
db.setDatabaseName(filename)
if not db.open():
    QMessageBox.warning(None, "Phone Log",
                        QString("Database Error: %1").arg(db.lastError().text()))
    sys.exit(1)
```

对于 SQLite 来说，只需要给定数据库名称即可。这个名称通常会是一个文件名，但有可能会是一个特殊的名字，如内存型数据库的“:memory:”。在调用 QSqlDatabase.open() 时会用到 SQLite 的驱动程序，如果上述文件不存在就会创建一个该名字的文件，在这种情况下，该文件不会有任何的表(table)或记录(record)。

值得注意的是，这里是以 None 来作为消息对话框的父对象的：这是因为，在创建主窗口之前就已经在试着建立连接了，所以是不可能有任何父对象存在的。由于这个应用程序要基于数据库，如果连接没有建立起来，那么就只能用错误消息来告知用户并终止应用程序了。

如果数据库连接成功打开，那么从现在开始，数据库的所有方法都将可以用于这一连接了。如果需要有两个或者更多个单独连接，而这些连接无论是用于同一个数据库的还是用于不同数据库的，都必须给 addDatabase() 传递第二个参数，以便为这一连接赋予一个名字，方便随后可以把不同的连接区分开来。

## 15.2 执行 SQL 查询

假设现在已经建立过一个连接，就可以对其执行一些 SQL 语句了。

```
query = QSqlQuery()
query.exec_("""CREATE TABLE outcomes (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
    name VARCHAR(40) NOT NULL)""")
query.exec_("""CREATE TABLE calls (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
    caller VARCHAR(40) NOT NULL,
    starttime DATETIME NOT NULL,
    endtime DATETIME NOT NULL,
    topic VARCHAR(80) NOT NULL,
    outcomeid INTEGER NOT NULL,
    FOREIGN KEY (outcomeid) REFERENCES outcomes)""")
```

由于没有说明要使用的是哪一个特定的数据库连接，所以 PyQt 将会使用默认的(未命名)连接，即早前建立的那个。在图 15.1 中，给出了由 SQL 调用语句所创建的表的形式。

AUTOINCREMENT 语句会让 SQLite 自动用后一个比前一个更大的 ID 来填充每一个 id 域，而第一个 id 域的值是 1。同样，FOREIGN KEY 语句会告诉 SQLite 这是一个与外键(foreign key)相关的关系。SQLite 3 不会强制使用外键关系，而只是用其作为文档辅助说明的一种手

段。对于自动 ID 和外键的语句形式在其他数据库中或许会有所不同。

许多数据库都有自己的数据类型集。例如，SQLite 3 称之为所谓的“存储类”(storage class)，包括 INTEGER、REAL 和 TEXT。PyQt 可以支持各种 SQL 标准数据类型，包括 VARCHAR、NUMBER、DATE 和 DATETIME，并可以在后台自动显式完成数据库各本体数据类型之间的转换。对于文本数据，PyQt 使用的是 Unicode，除非是数据库不支持 Unicode，在这种情况下，PyQt 会用数据库本体编码模式进行数据的转换。

现在就创建好了表格，可以向表里填充数据了。

```
for name in ("Resolved", "Unresolved", "Calling back", "Escalate",
             "Wrong number"):
    query.exec_("INSERT INTO outcomes (name) VALUES ('%s')" % name)
```

这里并不需要提供 ID 是因为已经要求由数据库替我们自动生成了它们。遗憾的是，前面的代码并不够完善：例如，如果在名称中含有任何一个单引号，代码运行就会失败。解决这一问题的办法是，要确保能够把那些无法接受的字符要么移除掉，要么转义成可接受的字符，不过，PyQt 提供了一种更好的替代解决方案：预查询(prepared query)技术。

对于预查询技术来说，有两种广泛使用的语法形式，一种是基于 ODBC 的占位符法(ODBC place holder)，而另一种是基于 Oracle 的命名变量法(Oracle named variable)。这两种方法 PyQt 都可以支持，如有需要，可以在后台自动完成从一种方法到另一种方法的转换，所以无论数据库底层采用的是哪一种方法，数据库都可以正常工作。

```
query.prepare("INSERT INTO calls (caller, starttime, endtime,
                                    "topic, outcomeid) VALUES (?, ?, ?, ?, ?)")
for name, start, end, topic, outcomeid in data:
    query.addBindValue(QVariant(QString(name)))
    query.addBindValue(QVariant(start))      # QDateTime
    query.addBindValue(QVariant(end))        # QDateTime
    query.addBindValue(QVariant(QString(topic)))
    query.addBindValue(QVariant(outcomeid)) # int
    query.exec_()
```

本例中使用的是 ODBC 语法。使用占位符法的一个好处是，PyQt 可以自行处理引号的问题，所以不必担心数据中所包含的内容是什么，只要传送的数据类型合适，它们都可以填充到数据域中。

```
query.prepare("INSERT INTO calls (caller, starttime, endtime,
                                    "topic, outcomeid) VALUES (:caller, :starttime,
                                    ":endtime, :topic, :outcomeid")
for name, start, end, topic, outcomeid in data:
    query.bindValue(":caller", QVariant(QString(name)))
    query.bindValue(":starttime", QVariant(start))
    query.bindValue(":endtime", QVariant(end))
    query.bindValue(":topic", QVariant(QString(topic)))
    query.bindValue(":outcomeid", QVariant(outcomeid))
    query.exec_()
```

第二个示例与第一个示例执行的是同样的工作，但使用了 Oracle 风格的命名变量法。PyQt 还可以支持其他不同类型的预查询语法格式，不过，除了用这里给出的这两种形式的语法所显示

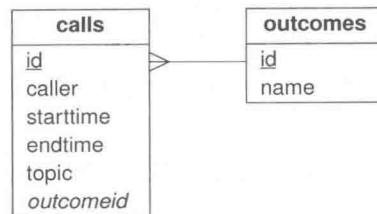


图 15.1 电话日志(Phone Log)数据库的设计

据，PyQt 会用数据库本体编码模式进行数据的转换。

的内容外，它们不会再添加任何新的东西。预查询技术可以提高它所支持的数据库的性能，而对于它所不支持的数据库则没有任何影响。

预查询也可以用来调用存储过程(stored procedure)，但这里并不打算讲述这些内容，因为这种做法既不普遍也不完全统一。比如，并不是所有的数据库都支持存储过程的，而不同的数据库和数据库之间对于调用和检索 OUT 值的语法也是不同的。此外，有返回值的存储过程也并未得到所有数据库的支持。

可以使用 QSqlQuery 来执行任何 SQL 语句。比如：

```
query.exec_("DELETE FROM calls WHERE id = 12")
```

在一条查询操作执行之后，可以通过调用 QSqlQuery.isActive() 来检查是否有错误发生；如果返回的是 False，就说明有错误发生了，通过调用 QSqlQuery.lastError().text() 就可以获得一个 QString 型的该错误信息的内容。

如果执行一次查询可能会影响到多个行，比如，DELETE 或者 UPDATE 中的 WHERE 从句就可能会选中多于一条的记录，此时可以调用 QSqlQuery.numRowsAffected(); 语句，如果它返回的是 -1，就表示出现了错误。

通过访问数据库驱动程序并调用 hasFeature()，可以看出数据库底层是否支持诸如事务(transaction)和 BLOB(Binary Large Object，二进制大对象)这样的功能。例如：

```
driver = QSqlDatabase.database().driver()
if driver.hasFeature(QSqlDriver.Transactions):
    print "Can commit and rollback"
```

在使用 QSqlQuery 时，可以通过调用 QSqlDatabase.database().transaction() 来触发一项事务操作，然后就可以要么使用 QSqlDatabase.database().commit()，要么使用 QSqlDatabase.database().rollback() 了。

通过查看 QSqlQuery 是如何执行 SELECT 语句以及如何遍历所产生的记录来对其做个总结。

```
DATETIME_FORMAT = "yyyy-MM-dd hh:mm"
ID, CALLER, STARTTIME, ENDTIME, TOPIC, OUTCOMEID = range(6)

query.exec_("SELECT id, caller, starttime, endtime, topic, "
           "outcomeid FROM calls ORDER by starttime")
while query.next():
    id = query.value(ID).toInt()[0]
    caller = unicode(query.value(CALLER).toString())
    starttime = unicode(query.value(STARTTIME).dateTime() \
                           .toString(DATETIME_FORMAT))
    endtime = unicode(query.value(ENDTIME).dateTime() \
                           .toString(DATETIME_FORMAT))
    topic = unicode(query.value(TOPIC).toString())
    outcomeid = query.value(OUTCOMEID).toInt()[0]
    subquery = QSqlQuery("SELECT name FROM outcomes "
                         "WHERE id = %d" % outcomeid)
    outcome = "invalid foreign key"
    if subquery.next():
        outcome = unicode(subquery.value(0).toString())
    print "%02d: %s %s - %s %s [%s]" % (id, caller, starttime,
                                           endtime, topic, outcome)
```

当执行 SELECT 语句时，可以使用诸如 QSqlQuery.next()、QSqlQuery.previous() 以及 QSqlQuery.seek() 这样的方法来遍历结果集。SELECT 操作一正确完成，isActive() 就会返回 True，但数据库的内部记录指针则不会再指向任何有效记录。如果查询的内部记录指针

成功地移动到有效记录的话，则任何一种导航方法都会返回 `True`；这就是为什么在访问第一条记录之前要调用 `QSqlQuery.next()` 的原因。如果发生错误或者是移动到了最后一条（或者第一条）记录，它们则会返回 `False`。

表 15.1 `QSqlQuery` 的部分方法

语法	说明
<code>q.addValue(v)</code>	在 <code>QSqlQuery q</code> 中使用绑定位置的值时，把 <code>QVariant v</code> 作为下一个变量添加进来
<code>q.bindValue(p, v)</code>	当在 <code>QSqlQuery q</code> 中使用占位符的值时，把占位符字符串 <code>p</code> 替换成 <code>QVariant v</code>
<code>q.boundValue(p)</code>	返回一个 <code>QVariant</code> 变量，作为 <code>QSqlQuery q</code> 中的占位符字符串 <code>p</code> 的值
<code>q.driver()</code>	返回与 <code>QSqlQuery q</code> 有关的 <code>QSqlDriver</code> 。 <code>QSqlDriver</code> 类会提供 <code>hasFeature()</code> 功能，以便用来说明底层数据库就可以支持哪些特性
<code>q.exec_(s)</code>	对 <code>QSqlQuery q</code> 执行字符串 <code>s</code> 中保存的那些 SQL 查询语句
<code>q.first()</code>	在 <code>SELECT</code> 查询执行完成之后，回到 <code>QSqlQuery q</code> 结果集合中的第一条记录处
<code>q.isActive()</code>	如果查询“有效”，就返回 <code>True</code> ——比如，在执行完 <code>SELECT</code> 查询后可以检查查询是否仍旧有效
<code>q.isValid()</code>	如果查询定位到了一条有效记录，就返回 <code>True</code> ；对于执行一条 <code>SELECT</code> 查询后，只有当 <code>isActive()</code> 返回的是 <code>True</code> 并且确实可以定位到一条记录时，它才会返回 <code>True</code>
<code>q.last()</code>	在执行完一次 <code>SELECT</code> 查询之后，回到 <code>QSqlQuery q</code> 结果集合中的最后一条记录处
<code>q.lastError()</code>	返回一个 <code>QSqlError</code> 对象；这样就可以提供一种 <code>errorString()</code> 式的方法
<code>q.next()</code>	在执行完一次 <code>SELECT</code> 查询之后，回到 <code>QSqlQuery q</code> 结果集合中的下一个记录处。这是对一个结果集合向前遍历的唯一方法。
<code>q.numRowsAffected()</code>	返回一个执行过 SQL 查询的行数，假设执行的不是 <code>SELECT</code> 查询，且假设底层数据库支持这项操作的话
<code>q.prepare(s)</code>	<code>q</code> 查询要执行的查询语句保存在字符串 <code>s</code> 中，为 <code>q</code> 提前做好准备
<code>q.previous()</code>	在执行完一次 <code>SELECT</code> 查询后，回到 <code>QSqlQuery q</code> 结果集合中的前一条记录处
<code>q.record()</code>	如果说有的话，返回一个 <code>QSqlRecord</code> 对象，其中会包含着 <code>QSqlQuery q</code> 的当前记录；在 <code>QSqlQuery.value()</code> 中带一个域索引参数通常会更方便些
<code>q.size()</code>	返回 <code>SELECT</code> 结果集合中的行数，如果 <code>SELECT</code> 没有得到执行，或是底层数据库不支持此项操作，则返回 <code>-1</code>
<code>q.value(i)</code>	如果当前记录中有域指针 <code>int i</code> 的话，返回其 <code>QVariant</code> 型的值

当对大型结果集进行操作时，假设只使用 `next()`，或只是向前使用 `seek()`，则可以调用 `QSqlQuery.setForwardOnly(True)`。对于某些数据库来说，这样做可以有力地提高性能或减少内存开销，或者同时满足这两种效果。

`QSqlQuery.value()` 方法会带一个索引位置参数，该参数是基于 `SELECT` 语句中域名称的顺序的。为此，不推荐使用 `SELECT *`，因为在这种情况下，并不知道各个域的顺序到底会是什么样的。每个域都是以 `QVariant` 型返回的，因此必须将其转换为适当的类型。对于日期/时间型域值，可以先将它们从 `QVariant` 转换成 `QDateTime`，再转换成 `QString`，最后转换成 `unicode`，这样就可以准备在控制台上打印了。

这里额外使用一个查询语句来查找源自其 ID 结果的名称，如果数据库中没有与这个完整名称相关的名称，就会给出错误的信息。对于一个大型的数据集合来说，在子查询中使用预查询技术往往更有效。

对于打算对数据库做的所有工作，都可以使用 `QSqlQuery` 来完成，不过，对于 GUI 编程来说，利用 PyQt 的那些 SQL 模型来完成会更简单些，并且当需要使用 `QSqlQuery` 时，这样做也不会妨碍到它的使用。

### 15.3 使用数据库窗体视图

能够为数据库数据提供的最简单的用户界面之一就是窗体 (form)，窗体可以一次性呈现出来自同一记录的各个域。在这一节中，将会开发一个使用了这种窗体的应用程序，该程序一开始只能认为是上一章节所介绍过的电话日志数据库的一个简化版，随后在其完整版本中将会包括一些外键域。

在这一节中介绍的各个例子都会用到 Qt 4.2 中所引入的 QDataWidgetMapper 类。下一节中的例子则会用到 SQL 表模型和 QTableView，也可以用于 Qt 4.1 或者更高的版本中。

在图 15.2 中给出的是该应用程序的简化版，它的源代码在 chap15/phonelog.pyw 中；而完整版的代码则放在 chap15/phonelog-fk.pyw 中。当这些应用程序第一次运行时，会创建其在后续运行时所要用到的伪数据记录。在 Linux 上使用 Qt 的内置 SQLite 来生成这些记录，往往比较快，但在某些 Windows 机器上，这一过程则会非常缓慢 [ 此时可以用启动画面闪屏 (splash screen) 的方式来掩饰这一缓慢的过程 ]。

简化版的应用程序只有一个简单的表，call 和一些调用程序，并且也没有外键域。窗体是通过 PhoneLogDlg 类表示的。初始化程序的代码很长，所以这里将会先看到其中的一部分，因为这一章的重点是对数据库编程技术的讨论，所以会暂时先跳过有关布局的那部分内容。

```
class PhoneLogDlg(QDialog):
    FIRST, PREV, NEXT, LAST = range(4)

    def __init__(self, parent=None):
        super(PhoneLogDlg, self).__init__(parent)

        callerLabel = QLabel("&Caller:")
        self.callerEdit = QLineEdit()
        callerLabel.setBuddy(self.callerEdit)
        today = QDate.currentDate()
        startLabel = QLabel("&Start:")
        self.startDateTime = QDateTimeEdit()
        startLabel.setBuddy(self.startDateTime)
        self.startDateTime.setDateRange(today, today)
        self.startDateTime.setDisplayFormat(DATETIME_FORMAT)
        endLabel = QLabel("&End:")
        self.endDateTime = QDateTimeEdit()
        endLabel.setBuddy(self.endDateTime)
        self.endDateTime.setDateRange(today, today)
        self.endDateTime.setDisplayFormat(DATETIME_FORMAT)
        topicLabel = QLabel("&Topic:")
        topicEdit = QLineEdit()
        topicLabel.setBuddy(topicEdit)
        firstButton = QPushButton()
        firstButton.setIcon(QIcon(":/first.png"))
```

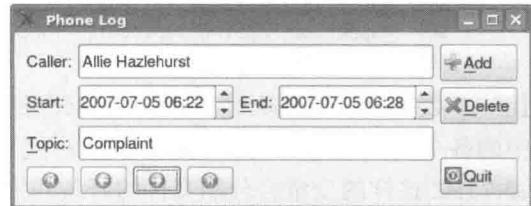


图 15.2 简化版的电话日志应用程序 Phone Log

为每个域都创建一个标签和一个适当的编辑用窗口部件。还要创建窗体的所有按钮，虽然这里只给出了第一个窗体按钮的创建过程。把字符串传递给添加按钮 (Add)、删除按钮 (Delete)

和退出按钮(Quit)的构造函数，以便可以为它们各自赋写标题，此外还要为它们赋予相应的图标。

```
self.model = QSqlTableModel(self)
self.model.setTable("calls")
self.model.setSort(STARTTIME, Qt.AscendingOrder)
self.model.select()
```

所有的窗口部件放置安排妥当后，就可以创建 QSqlTableModel 了。由于并没有指定用哪一种具体的数据库连接，所以它会使用一个默认的数据库连接。要告诉该模型要用的是哪个表，还要调用 `select()` 来给自己填充数据。此外，还要选择给表应用一下排序命令。

现在就具备了合适的窗口部件和模型，接下来就必须用某种方法把它们联系起来。用 QDataWidgetMapper 就可以实现这一点。

```
self.mapper = QDataWidgetMapper(self)
self.mapper.setSubmitPolicy(QDataWidgetMapper.ManualSubmit)
self.mapper.setModel(self.model)
self.mapper.addMapping(self.callerEdit, CALLER)
self.mapper.addMapping(self.startDateTime, STARTTIME)
self.mapper.addMapping(self.endDateTime, ENDTIME)
self.mapper.addMapping(topicEdit, TOPIC)
self.mapper.toFirst()
```

要让一个数据窗口部件映射器工作，就必须给它一个数据模型和一个映射集，该映射集是指窗体中的各个窗口部件和数据模型中所对应的列之间的映射关系(各个诸如 ID、CALLER、STARTTIME 这样的变量会分别文件的开头并依次设置成 0、1、2 等值)。可以设置让映射器来提供操作自动更改的功能，也可以设置成仅在告知更改时才提供更改的操作。当然，后一种方法会更好些，因为它可以给出更为精细的控制，并且意味着，当用户要对另一个记录进行操作时，可以确保那些尚未保存的更改可以被保存起来。一旦构建了映射，就需要让映射器把各条记录填充到各个窗口部件内；可以之前调用的 `toFirst()` 完成这一操作，这就意味着在应用程序启动时，就可以显示第一批记录了。

```
self.connect(firstButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(PhoneLogDlg.FIRST))
self.connect(prevButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(PhoneLogDlg.PREV))
self.connect(nextButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(PhoneLogDlg.NEXT))
self.connect(lastButton, SIGNAL("clicked()"),
            lambda: self.saveRecord(PhoneLogDlg.LAST))
self.connect(addButton, SIGNAL("clicked()"), self.addRecord)
self.connect(deleteButton, SIGNAL("clicked()"),
            self.deleteRecord)
self.connect(quitButton, SIGNAL("clicked()"), self.accept)
self.setWindowTitle("Phone Log")
```

前四个连接用来提供导航功能。在每个连接中，这里所谓的 `saveRecord()`，可以用来保存那些尚未保存的全部更改，然后，就可以根据那些已经封装进去的 `lambda` 语句中的参数进行导航。这意味着，只需要使用同一个方法，即 `saveRecord()` 就可以了，而不是每个导航按钮都分别要用一个自己的方法。不过，这个连接只适用于 PyQt 4.1.1 或更高的版本中。对于较早的版本，就必须要有实例变量(例如，列表)，其中要含有可用于防止被垃圾收集的 `lambda` 函数的引用。

```
def accept(self):
    self.mapper.submit()
    QDialog.accept(self)
```

如果用户单击退出(Quit)按钮，就需要调用 `QDataWidgetMapper.submit()`，它会将当前的记录写回到底层数据模型中，然后，再来关闭该窗口。

```
def saveRecord(self, where):
    row = self.mapper.currentIndex()
    self.mapper.submit()
    if where == PhoneLogDlg.FIRST:
        row = 0
    elif where == PhoneLogDlg.PREV:
        row = 0 if row <= 1 else row - 1
    elif where == PhoneLogDlg.NEXT:
        row += 1
        if row >= self.model.rowCount():
            row = self.model.rowCount() - 1
    elif where == PhoneLogDlg.LAST:
        row = self.model.rowCount() - 1
    self.mapper.setCurrentIndex(row)
```

如果用户在进行的是导航，就必须记住当前所在行，因为在调用 `submit()` 后会忘记它的值。然后，在保存了当前记录后，就要对行进行设置，使其成为用户想要的(但仍要保持在有效范围内)那个适当的行，接着使用 `setCurrentIndex()`，将该行移动到适当的记录处。

```
def addRecord(self):
    row = self.model.rowCount()
    self.mapper.submit()
    self.model.insertRow(row)
    self.mapper.setCurrentIndex(row)
    now = QDateTime.currentDateTime()
    self.startTime.setDateTime(now)
    self.endTime.setDateTime(now)
    self.callerEdit.setFocus()
```

通常总是会在末尾处添加那些新的记录。为此，就需要先找到最后一条记录之后的行的位置，然后再来保存当前记录，之后在模型中的最后一行，再插入新的记录。接下来，将映射器的当前索引值设置成新的行，初始化若干个域，并为调用者域赋予焦点，这些都是为用户开始键入字符所做的准备。

```
def deleteRecord(self):
    caller = self.callerEdit.text()
    starttime = self.startTime.dateTime().toString(
        DATETIME_FORMAT)
    if QMessageBox.question(self,
        QString("Delete"),
        QString("Delete call made by<br>%1 on %2?") \
        .arg(caller).arg(starttime),
        QMessageBox.Yes|QMessageBox.No) == QMessageBox.No:
        return
    row = self.mapper.currentIndex()
    self.model.removeRow(row)
    self.model.submitAll()
    if row + 1 >= self.model.rowCount():
        row = self.model.rowCount() - 1
    self.mapper.setCurrentIndex(row)
```

如果用户从当前记录中挑出了一些信息后单击了删除(Delete)按钮, 就需要询问用户是否确定要执行删除操作。如果用户确定删除, 就检索到当前行, 从数据模型中删除那些行, 再调用 `submitAll()` 来迫使数据模型把修改写回底层数据源(在本例中就是数据库)中。然后, 通过导航到下一条记录来结束当前操作。

之前曾使用过 `submitAll()`, 因为已经看到是如何对模型进行删除操作的, 但没有对映射器进行过删除操作, 而对于数据库, 则必须通过调用这一方法来确认是对模型进行了相应的更改, 除非数据库视图(或数据窗口部件映射器)已提前设置为自动提交了。数据窗口部件映射器的 API 不允许我们添加或者删除记录, 而只能编辑现有的数据记录, 正是因为这样, 就必须要使用底层模型来进行记录的添加或者删除。

到目前为止, 使用到的所有技术都可应用于任何数据库表或者可编辑型数据库视图, 以便为用户进行记录的导航、添加、更新和删除操作提供相应的手段。然而, 大多数情况下, 还有一些外键需要加以考虑, 这里在回顾电话日志应用程序时, 将会就这一问题做些讨论, 如图 15.3 所示。

`calls` 表有一个外键域 `outcomeid`。为了能够让这个域在窗体中显示成组合框的形式, 以便将 `outcomes` 表中的 `name` 域来对应于相应的 ID。要做到这一点, 可以用常见的方式先创建一个组合框, 但先不要为之填充内容。

由于现在使用的表有一个外键, 所以就必须使用 `QSqlRelationalTableModel` 而不是 `QSqlTableModel`。

```
self.model = QSqlRelationalTableModel(self)
self.model.setTable("calls")
self.model.setRelation(OUTCOMEID,
    QSqlRelation("outcomes", "id", "name"))
self.model.setSort(STARTTIME, Qt.AscendingOrder)
self.model.select()
```

`QSqlRelationalTableModel` 与 `QSqlTableModel` 非常类似, 只是它额外提供了几个方法来处理域间的关系。`setRelation()` 方法会带一个模型中的域指针和一个 `QSqlRelation` 对象。关系对象是采用外键的表、要用来存储的域以及要用来显示的域的名字创建的。

还必须对数据窗口部件映射器的代码做些修改。特别是, 必须要使用 `QSqlRelationalDelegate` 而不是用标准的内置委托, 还必须对那些用于外键的组合框进行设置。

```
self.mapper = QDataWidgetMapper(self)
self.mapper.setSubmitPolicy(QDataWidgetMapper.ManualSubmit)
self.mapper.setModel(self.model)
self.mapper.setItemDelegate(QSqlRelationalDelegate(self))
self.mapper.addMapping(self.callerEdit, CALLER)
self.mapper.addMapping(self.startTime, STARTTIME)
self.mapper.addMapping(self.endTime, ENDTIME)
self.mapper.addMapping(topicEdit, TOPIC)
relationModel = self.model.relationModel(OUTCOMEID)
self.outcomeComboBox.setModel(relationModel)
self.outcomeComboBox.setModelColumn(
```

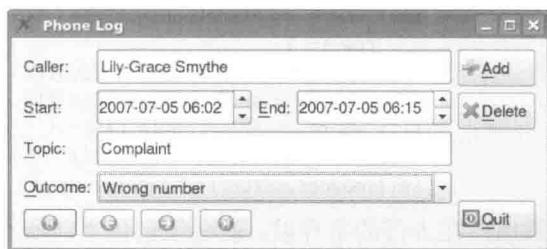


图 15.3 电话日志应用程序 Phone Log

```
relationModel.fieldIndex("name"))
self.mapper.addMapping(self.outcomeComboBox, OUTCOMEID)
self.mapper.toFirst()
```

这些代码与之前看过的那些代码类似。关系委托的设置很容易，但组合框的设置却略有些许麻烦。首先，必须检索在数据模型(`calls` 表)中所使用的关系模型(`outcomes` 表)来处理外键。`QComboBox` 实际上是一个带有内置模型的便捷视图窗口部件，就像一个 `QListWidget` 一样；不过，也有可能用它来取代我们自己的数据模型，这就是我们在这里所做的事情的实情。然而，一个组合框只能显示单一的一个列，而我们的关系模型却有两个列(`id, name`)，所以必须说明所要显示的是哪一个。被关系模型所用到的是哪个列索引值尚无法确定(因为它是为我们而创建的，而不是由我们自己创建的)，所以可以使用带一个域值名字的 `fieldIndex()` 方法来得到那个正确的列索引。组合框一旦构建完毕，就可以将其添加到诸如普通窗口部件那样的映射器中了。

这样就完成了处理外键这一工作所需做的更改。此外，这里还适时地对应用程序做了其他几个方面的小改变。

在应用程序的简化版中，把退出(`Quit`)按钮连接到了自定义的 `accept()` 方法上，并且会显得相当直观，就是从 `reject()` 方法中调用了 `accept()` 方法。这样做是为了确保应用程序始终能够在终止之前保存当前记录的更改情况。在外键版应用程序中，则采取了不同的办法，是将退出(`Quit`)按钮与 `done()` 方法相连接的。

```
def done(self, result=None):
    self.mapper.submit()
    QDialog.done(self, True)
```

这个方法会作为退出按钮的连接而被调用，或者，如果用户单击了窗口上的“X”关闭按钮或按下了“Esc 键”，就会关闭窗口。此时，就会保存当前记录并调用基类的 `done()` 方法。第二个参数是一个强制参数，但在此种情况之下，无论该参数保存的是什么值都无关紧要：`True` 值可以代表 `accept()`，而 `False` 值则代表 `reject()`，但无论是哪一种方式，都将会关闭该窗口。

还完成了一个细小的变化，可以把以下两行添加到 `addRecord()` 方法中：

```
self.outcomeComboBox.setCurrentIndex(
    self.outcomeComboBox.findText("Unresolved"))
```

这么做就可以确保当用户单击添加(`Add`)按钮来添加新的记录时，结果组合框 `outcome` 将会产生一个有意义的默认值，除了已经设置过的日期/时间默认值之外。

对于拥有较多域的表来说，窗体通常是非常有用的，特别是当有许多验证内容需要基于跨域的联系时更是如此。不过，对于那些只有较少的域的表，或者是对于用户希望看到多条记录的那些表来说，就需要使用表格化的视图了。这正是下一节的主题。

## 15.4 使用数据库表视图

也许呈现数据库中数据最为自然、最为便捷的方式就是在 GUI 表中显示数据库的各个表和各个视图。这样就可以允许用户能够一次看到多条记录，同时它对于显示主从关系的细节内容也会特别方便。

在这一节中，我们将会看看资产管理器应用程序 Asset Manager。它的代码在 `chap15/`

assetmanager.pyw 文件中。这个应用程序一共有四个表，是通过下面的 SQL 语句创建的：

```
query = QSqlQuery()
query.exec_("""CREATE TABLE actions (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
    name VARCHAR(20) NOT NULL,
    description VARCHAR(40) NOT NULL)""")
query.exec_("""CREATE TABLE categories (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
    name VARCHAR(20) NOT NULL,
    description VARCHAR(40) NOT NULL)""")
query.exec_("""CREATE TABLE assets (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
    name VARCHAR(40) NOT NULL,
    categoryid INTEGER NOT NULL,
    room VARCHAR(4) NOT NULL,
    FOREIGN KEY (categoryid) REFERENCES categories)""")
query.exec_("""CREATE TABLE logs (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
    assetid INTEGER NOT NULL,
    date DATE NOT NULL,
    actionid INTEGER NOT NULL,
    FOREIGN KEY (assetid) REFERENCES assets,
    FOREIGN KEY (actionid) REFERENCES actions)""")
```

动作表 actions 和类别表 categories 都是典型的引用数据表，分别有一个 ID、一个简要描述(name)和一个长描述(description)。主表是资产表 assets；这个表中由名称 name、类别 category 以及建筑中每个资产的位置 location 构成。日志表 logs 用于跟踪记录某个资产在其生命周期内所做的变动情况。这些表格的示意图如图 15.4 所示。

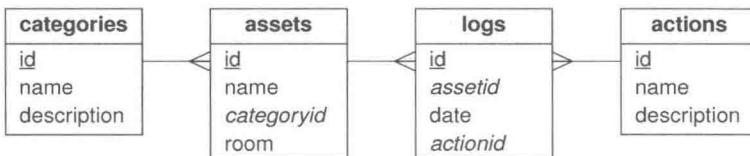


图 15.4 资产管理器应用程序 Asset Manager 的数据库设计

资产管理器应用程序 Asset Manager 是一个对话框式的主窗口，其中带有两个主从关系型的 QTableView。该应用程序的运行效果如图 15.5 所示。顶部的那个表格中显示的是资产表 assets 的内容，底下的那个表格显示的是从 logs 表中读取的当前资产记录的信息。用户可以添加和删除资产表 assets 和日志记录表 logs 中的记录，也可以就地编辑这两个表格。用户还可以通过弹出一个适当的对话框来添加、删除和编辑类别引用表 categories 和动作引用表 actions。这个对话框也会用到 QTableView，尽管本可以轻松用 QDataWidgetMapper 来代替 QTableView。

这里将会从数据库的创建和连接操作开始，然后是主窗体，再之后将会查看那个基于引用数据的对话框。正如在手机登录应用程序 Phone Log 中所做的那样，资产管理器应用程序 Asset Manager 也会在其第一次运行时生成一系列的伪记录。正如在上一节中所述的那样，在 Linux 中使用 SQLite 这样做的运行速度会非常快，而在一些 Windows 机器上却会非常慢。

```

app = QApplication(sys.argv)
db = QSqlDatabase.addDatabase("QSQLITE")
db.setDatabaseName(filename)
if not db.open():
    QMessageBox.warning(None, "Asset Manager",
        QString("Database Error: %1").arg(db.lastError().text()))
    sys.exit(1)
form = MainForm()
form.show()
app.exec_()

```

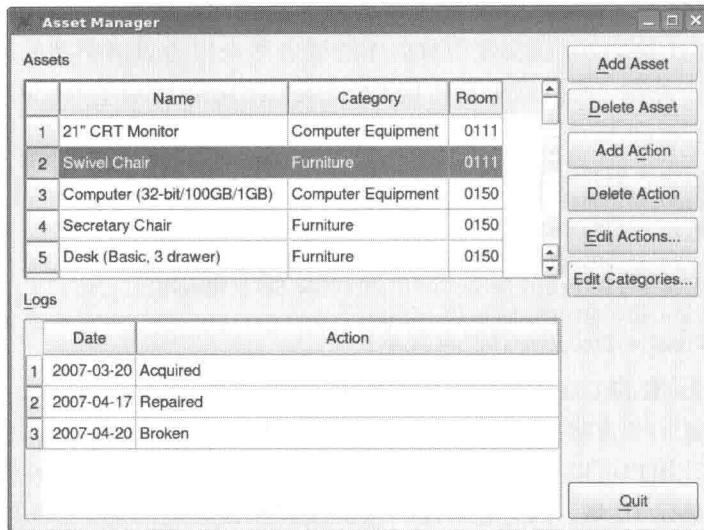


图 15.5 资产管理器应用程序 Asset Manager

像往常一样，还是先从创建一个 `QApplication` 对象开始。接下来，会创建各个连接；如果数据库文件不存在的话，SQLite 将会创建一个空的数据库文件。然后，来创建其主窗体，并对其调用一下 `show()` 来为其安排一个绘制事件，这样就可以开始事件的循环了。

这里并没有给出应用程序第一次运行时用来生成伪数据的那些代码，也没有给出闪屏的代码。这些代码，当然，也会放到源文件中的，可以在 `chap15 / assetmanager.pyw` 文件中找到它们。

正如在上一节中所做的一样，这里将会跳过有关窗体布局的代码，以精力集中在创建各个窗口部件和模型上。同时，还会跳过创建各个按钮的代码，虽然之后会给出第一批中几个有关信号-槽连接的代码。

```

class MainForm(QDialog):
    def __init__(self):
        super(MainForm, self).__init__()
        self.assetModel = QSqlRelationalTableModel(self)
        self.assetModel.setTable("assets")
        self.assetModel.setRelation(CATEGORYID,
            QSqlRelation("categories", "id", "name"))
        self.assetModel.setSort(ROOM, Qt.AscendingOrder)
        self.assetModel.setHeaderData(ID, Qt.Horizontal,
            QVariant("ID"))

```

```

        self.assetModel.setHeaderData(NAME, Qt.Horizontal,
                                      QVariant("Name"))
        self.assetModel.setHeaderData(CATEGORYID, Qt.Horizontal,
                                      QVariant("Category"))
        self.assetModel.setHeaderData(ROOM, Qt.Horizontal,
                                      QVariant("Room"))
        self.assetModel.select()
    
```

正如前面章节中所看到的那样，该模型是以几乎相同的方式创建的。ID、NAME 和其他参数这样的整数型列索引都是在 `assetmanager.pyw` 文件中提前设置好的。和以往使用 `QDataWidgetMapper` 不同的是，我们已经设置过作为列标题的标题数据；如果还没有这样设置过，用来呈现模型的 `QTableView` 就会使用数据库的域名称来作为表格中列的标题。由于目录域 `categoryid` 是一个外键，所以这里会使用 `QSqlRelationalTableModel` 并以适当的方式来调用 `setRelation()`。

```

    self.assetView = QTableView()
    self.assetView.setModel(self.assetModel)
    self.assetView.setItemDelegate(AssetDelegate(self))
    self.assetView.setSelectionMode(QTableView.SingleSelection)
    self.assetView.setSelectionBehavior(QTableView.SelectRows)
    self.assetView.setColumnHidden(ID, True)
    self.assetView.resizeColumnsToContents()

```

这里的视图是一个标准的 `QTableView`，而没有对 `QSqlRelationalDelegate` 进行设置，因为我们已经设置过一个自定义委托。下面很快就会先来看看这部分的内容。对选择模式进行设置是为了能够让用户可以导航至单个域；选择的行为是，拥有焦点的行都会进行高亮显示。这里并不打算显示 ID 域，因为它对用户来讲没有什么意义，所以会将其予以隐藏。

目前还没有用到标准 `QSqlRelationalDelegate` 的原因是希望能够控制房间号码的编辑功能，因为房间号的验证一般不是那么简单。现在先会兜一个简单的圈子，先来看看 `AssetDelegate` 类。

```

class AssetDelegate(QSqlRelationalDelegate):
    def __init__(self, parent=None):
        super(AssetDelegate, self).__init__(parent)

```

初始化程序是大多数委托子类的一种典型形式，只是对基类进行了调用。

```

def paint(self, painter, option, index):
    myoption = QStyleOptionViewItem(option)
    if index.column() == ROOM:
        myoption.displayAlignment |= Qt.AlignRight|Qt.AlignVCenter
    QSqlRelationalDelegate.paint(self, painter, myoption, index)

```

这里对 `paint()` 方法的重新实现只是为了把房间号码进行右对齐。要实现这一目的，可以改变 `QstyleOptionViewItem` 的值，并且将绘制事件自身的实现交给了基类来完成。

```

def createEditor(self, parent, option, index):
    if index.column() == ROOM:
        editor = QLineEdit(parent)
        regex = QRegExp(r"(?:0[1-9]|1[0124-9]|2[0-7])"
                       r"(?:0[1-9]|1[5-9]|6[012])")
        validator = QRegExpValidator(regex, parent)
        editor.setValidator(validator)
        editor.setInputMask("9999")
        editor.setAlignment(Qt.AlignRight|Qt.AlignVCenter)

```

```

        return editor
    else:
        return QSqlRelationalDelegate.createEditor(self, parent,
                                                option, index)

```

`createEditor()`方法的核心那些设置 `QLineEdit` 以实现房间号输入的那些代码。房间号的长度是四个数字，其中之一是楼层数，其范围为 01 ~ 27（但不包括 13），还有一部分是楼层上的房间号，范围为 01 ~ 62。例如，0231 是二层、房间号为 31 的房间，但 0364 无效。这里的正则表达对于给定有效的房间号是非常重要的，但它不能设置房间号数字中的最小数，因为一个、两个或三个数字或许都可以用来表示一个有效的四位数房间号。通过输入掩码的方式，即必须键入四个数字的方法来成功解决了这一问题。对于其他的域，则会把工作的重点交给基类来完成。

```

def setEditorData(self, editor, index):
    if index.column() == ROOM:
        text = index.model().data(index, Qt.DisplayRole).toString()
        editor.setText(text)
    else:
        QSqlRelationalDelegate.setEditorData(self, editor, index)

```

编辑器一旦创建完成，视图就会调用 `setEditorData()`，以便可以用当前的值来填充它。在这个例子中，只关心的是房间这一列，但对其他域的处理工作则会转交给它的基类完成。

```

def setModelData(self, editor, model, index):
    if index.column() == ROOM:
        model.setData(index, QVariant(editor.text()))
    else:
        QSqlRelationalDelegate.setModelData(self, editor, model,
                                            index)

```

这里使用了与之前那个方法相类似的处理方法，来实现房间域的处理并将其余各个域的处理交给基类来进行处理。实际上，本来是可以不必重新实现此方法的，因为 PyQt 还是比较智能的，它可以从 `QLineEdit` 中自动检索出所需的值来。然而，对我们自己的自定义实现要负有完全的责任，这始终是一个不错的实践需要。

到此为止，就算是结束了提前先学习的内容，可以返回到 `MainForm.__init__()` 方法中了，那就先从下面的那个表格开始看起吧，其中显示着适用于当前资产的各条日志记录。

```

self.logModel = QSqlRelationalTableModel(self)
self.logModel.setTable("logs")
self.logModel.setRelation(ACTIONID,
                        QSqlRelation("actions", "id", "name"))
self.logModel.setSort(DATE, Qt.AscendingOrder)
self.logModel.setHeaderData(DATE, Qt.Horizontal,
                           QVariant("Date"))
self.logModel.setHeaderData(ACTIONID, Qt.Horizontal,
                           QVariant("Action"))
self.logModel.select()

```

用于创建日志模型的代码与用于创建资产模型的代码几乎完全相同。因为有外键存在的缘故，这里使用的是 `QSqlRelationalTableModel`，同时还提供了我们自己的各个列的标题。

```

self.logView = QTableView()
self.logView.setModel(self.logModel)
self.logView.setItemDelegate(LogDelegate(self))
self.logView.setSelectionMode(QTableView.SingleSelection)

```

```

self.logView.setSelectionBehavior(QTableView.SelectRows)
self.logView.setColumnHidden(ID, True)
self.logView.setColumnHidden(ASSETID, True)
self.logView.resizeColumnsToContents()
self.logView.horizontalHeader().setStretchLastSection(True)

```

这段代码与在资产表 assets 中的代码也十分类似，但有三个不同之处。在这里使用的是自定义的 LogDelegate 类——这里之所以没有再次对其进行回顾是因为它的结构与 AssetDelegate 类的代码非常类似。它能实现日期域的自定义编辑。这里还同时隐藏了日志记录的 ID 域和 assetid 外键——这里之所以没有必要显示这些资产日志记录是属于哪一条资产的，是因为这里所正在使用的是主-从关系，所以对用户可见的那些日志记录就一定会是当前资产中的部分记录（后面很快将会看到编写主-从关系的代码是如何完成的）。最后一个不同之处在于，这里对最后的一列进行了设置，使其可以进行扩展，以便能够填充该列中所有可用的空间。QTableView.horizontalHeader() 方法会返回一个 QHeaderView 结果，而正好可以用它来控制表格视图中的某些列，包括对这些列的宽度的控制等。

```

self.connect(self.assetView.selectionModel(),
             SIGNAL("currentRowChanged(QModelIndex,QModelIndex)"),
             self.assetChanged)
self.connect(addAssetButton, SIGNAL("clicked()"),
             self.addAsset)

```

如果用户导航到了另外一行，那就必须更新日志视图，以便能够为正确的资产显示出正确的日志记录来。可以将 assetChanged() 方法与第一个连接相互配合的形式来实现这一点，该函数将会在后面马上看到。

每个视图都至少有一种选择模式，以用来跟踪选择的是视图中的哪些模型（如果有的话）。这里与视图的选择模型的 currentRowChanged() 信号相连接，是为了可以根据当前的资产值来更新日志的视图。

所有其他的连接都是按钮点击型连接，就像这里给出的第二个连接的代码一样。随着这一节内容的不断向前推进，这里将会慢慢涵盖所有与按钮相连的各个方法。

```

self.assetChanged(self.assetView.currentIndex())
self.setMinimumWidth(650)
self.setWindowTitle("Asset Manager")

```

初始化程序的最后是对资产视图当前模型索引调用 assetChanged() 方法——这将导致在日志视图中会显示出相关资产的记录。

```

def assetChanged(self, index):
    if index.isValid():
        record = self.assetModel.record(index.row())
        id = record.value("id").toInt()[0]
        self.logModel.setFilter(QString("assetid = %1").arg(id))
    else:
        self.logModel.setFilter("assetid = -1")
    self.logModel.select()
    self.logView.horizontalHeader().setVisible(
        self.logModel.rowCount() > 0)

```

这个方法仅被窗体的初始化程序调用过一次，然后，无论用户何时导航到一个不同的资产，即导航到资产表视图中的任何一个不同的行，都会调用这个方法一次。

如果视图中新位置的模型索引是有效的，就从模型中提取所有的记录并且对记录模型设置一个过滤器，这个过滤器只选择那些与当前行中资产 ID 相对应的 assetid 的日志记录（这

与 `SELECT * FROM logs WHERE asstid=id` 的语法是等效的)。然后, 调用 `select()` 来刷新那些选中日志记录的日志视图。如果模型索引无效, 就把 ID 设置成一个并不存在的 ID 值, 从而可以保证检索到的任何行和日志视图都将为空。最后, 隐藏日志视图的列标题, 如果没有日志记录要显示的话。

`Record()` 方法是由 `QSqlTableModel` 和 `QSqlRelationalTableModel` 类提供的除了它们的 `QAbstractItemModel` 基类之外的一个扩展方法, 以便让它们可以更轻松地使用数据库。其他的扩展方法还有 `setQuery()`, 它可以支持利用 SQL 语法和 `insertRecord()` 来写入我们自己的 SELECT 语句, 比如记录的添加。

对于 `assetChanged()` 方法的连接以及对于该方法的实现, 都会在两个模型之间建立主-从关系(因此, 还会在它们两者的视图之间)。

```
def done(self, result=1):
    query = QSqlQuery()
    query.exec_("DELETE FROM logs WHERE logs.assetid NOT IN"
               "(SELECT id FROM assets)")
    QDialog.done(self, 1)
```

当应用程序终止时会执行最后一个查询来删除那些所有已不存在的(比如已经删除了的)资产的全部日志记录。理论上来说, 这并非是一定非做不可的, 因而这样做了也不应该有什么。这是因为, 对于支持事务操作的那些数据库来说, 可以使用事务来确保一个资产的删除操作, 当然也可以对其日志记录使用事务操作来保证它们也会被删除掉。

```
def addAction(self):
    index = self.assetView.currentIndex()
    if not index.isValid():
        return
    QSqlDatabase.database().transaction()
    record = self.assetModel.record(index.row())
    assetid = record.value(ID).toInt()[0]

    row = self.logModel.rowCount()
    self.logModel.insertRow(row)
    self.logModel.setData(self.logModel.index(row, ASSETID),
                         QVariant(assetid))
    self.logModel.setData(self.logModel.index(row, DATE),
                         QVariant(QDate.currentDate()))
    QSqlDatabase.database().commit()
    index = self.logModel.index(row, ACTIONID)
    self.logView.setCurrentIndex(index)
    self.logView.edit(index)
```

如果用户要求添加一个动作(比如一个新的日志记录)时, 就会调用这个方法。首先会检索当前资产的 `assetid`, 然后会在日志表 `logs` 中的最后插入一个新的日志记录。接下来, 会将记录的 `assetid` 外键设置到刚才检索的那个记录上, 同时为其提供一个初始默认日期。最后, 把模型索引查找至新日志记录的动作组合框中, 并且为用户初始化好编辑功能, 以便用户可以选择想要的功能。

在检索 `assetid` 之前, 可以先开始一项事务操作。这是为了防止理论上出现的可能性, 即在检索 `assetid` 时, 在创建新的日志记录之前, 资产却已经被删除了。如果发生了这种情况, 日志记录就会引用一个不存在的资产, 这就可能在随后的操作中会导致崩溃或莫名其妙的问题。所以, 一旦调用了 `commit()`, 就可以知道, 资产和新的日志记录已经存在了。如果现

在有人想删除资产，就可以这样做——但资产的各条日志记录，包括当前这条记录，都将会正确地得到删除。

实际上，从防范性的角度来说，要从根本上防止此类问题的出现，或许可以像下面这样来结构化事务操作的代码：

```
class DatabaseError(Exception): pass

rollback = False
try:
    if not QSqlDatabase.database().transaction():
        raise DatabaseError
    rollback = True
    # execute commands that affect the database
    if not QSqlDatabase.database().commit():
        raise DatabaseError
    rollback = False
finally:
    if rollback:
        if not QSqlDatabase.database().rollback():
            raise DatabaseError
```

这样做可以尽量保证在问题发生时，避免提交事务，也可以避免在调用提交操作时它不能得到正确执行，这样就可以回滚到之前正确的位置处，并由此来保证了数据库的关系完整性。不过，倘若回滚失败，那么所有的操作尝试都将会被停止。相关的错误信息可以从 `QSqlDatabase.database().lastError().text()` 中得到，该函数可以返回一个 `QString` 型信息文本。

一个事务的作用域是从调用 `transaction()` 开始直到该事务操作得到提交或者是回滚。无论是通过 `QSqlDatabase` 还是通过模型来访问数据库的。事务操作的上下文适用于所有的 SQL 语句，包括那些独立查询执行语句以及那些由不同模型所执行的语句，只要它们在同一个事务操作的上下文作用域内，都会应用到同一个数据库内。

倘若正在使用的是 Python 2.6，或者正在使用的是 Python 2.5 中的 `from __future__ import with_statement`，通过创建并利用上下文管理器都可以用来简化之前的那些代码。

面向事务的处理方法可以自动处理一些操作，所以不会出现什么问题。还有另外一种处理方法是，假定一切都能工作，可以依靠数据库来保留外键关系和有关数据完整性的其他内容。这样做的话，就不会用到 SQLite 3，因为它并不会强调关系的完整性，但是它的确是与一些其他数据库共同完成工作的。使用这种方法后，在编写代码时通常就无须再来使用事务方面的代码了。大多数时候这样做都可以正常工作，而对于极少数情况下所出现的那几种情况，则可以依靠数据库来拒绝执行那些不遵守数据库规则的动作，同时还可以向用户提供报告错误的消息文本。

需要注意的是，所有事务都是在数据库上设置的，都是通过静态的 `QSqlDatabase.database()` 方法进行访问的。也可以通过对模型调用 `database()` 方法来访问数据库。每个数据库连接都可以一次处理一个事务，所以如果需要同时处理多个事务的话，就必须为打算使用的每个额外事务再额外单独建立一个连接。

```
def deleteAction(self):
    index = self.logView.currentIndex()
    if not index.isValid():
        return
```

```

record = self.logModel.record(index.row())
action = record.value(ACTIONID).toString()
if action == "Acquired":
    QMessageBox.information(self, "Delete Log",
        "The 'Acquired' log record cannot be deleted.<br>"
        "You could delete the entire asset instead.")
    return
when = unicode(record.value(DATE).toString())
if QMessageBox.question(self, "Delete Log",
    "Delete log<br>%s %s?" % (when, action),
    QMessageBox.Yes|QMessageBox.No) == QMessageBox.No:
    return
self.logModel.removeRow(index.row())
self.logModel.submitAll()

```

对于删除操作，已经实现过的逻辑是，用户不能删除“Acquired”日志记录，就是说，不可删除第一条日志记录(但用户可以删除一个资产，也可以删除资产所带的全部日志记录，这点很快就会看到)。如果该日志记录是用户可以删除并且是用户确定要删除的记录，那么就只需在日志模型上调用 `removeRow()`，然后再调用 `submitAll()` 即可更新底层数据库了。

```

def editActions(self):
    form = ReferenceDataDlg("actions", "Action", self)
    form.exec_()
def editCategories(self):
    form = ReferenceDataDlg("categories", "Category", self)
    form.exec_()

```

由于 `actions` 表和 `categories` 表都具有相同的结构，因而当想要向下深入挖掘它们的记录时，如添加、编辑和删除它们的记录，都可以使用同一个智能对话框。可以将数据库中表的名字赋给对话框，也可以向用户界面(例如，在对话框的标题栏中)给定要显示的引用数据。

由于并没有什么新的需要学习的东西，这里将不会给出图 15.6 中所示的 `ReferenceDataDlg` 的代码。它使用了带有 `QSqlTableModel` 集的 `QTableView`，并将其设置到构造函数中的表上。当前位置的编辑功能可以由表格视图和表模型自动处理。添加一条记录就像向模型中插入一个新行并设置一个指向它的视图一样简单。

对于引用数据的删除操作，可以执行一个查询来查看那些特定的引用数据记录是否正被其他某个表所使用，也就是说，日志表 `logs` 中是否有任何记录在使用着某个动作，或者是资产表 `assets` 中是否有任何记录在使用着某个类别。如果有记录正在使用中，就会弹出详细的错误消息并阻止删除操作。否则，可以向模型中的相关行调用 `removeRow()`，然后，调用 `submitAll()` 来向数据库提交所做的修改，就像我们当时在删除动作中所做的那样。

与引用数据不同，添加和删除资产是由主窗体的方法处理的。

```

def addAsset(self):
    row = self.assetView.currentIndex().row() \

```

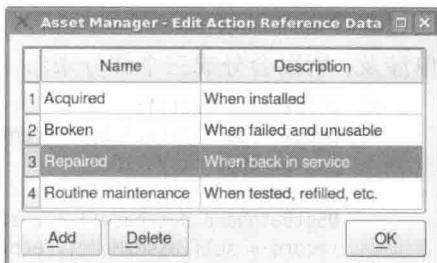


图 15.6 资产管理器应用程序的引用数据窗体

```

if self.assetView.currentIndex().isValid() else 0

QSqlDatabase.database().transaction()
self.assetModel.insertRow(row)
index = self.assetModel.index(row, NAME)
self.assetView.setCurrentIndex(index)

assetid = 1
query = QSqlQuery()
query.exec_("SELECT MAX(id) FROM assets")
if query.next():
    assetid = query.value(0).toInt()[0]
query.prepare("INSERT INTO logs (assetid, date, actionid) "
             "VALUES (:assetid, :date, :actionid)")
query.bindValue(":assetid", QVariant(assetid + 1))
query.bindValue(":date", QVariant(QDate.currentDate()))
query.bindValue(":actionid", QVariant(ACQUIRED))
query.exec_()
QSqlDatabase.database().commit()
self.assetView.edit(index)

```

当用户添加一个新的资产时，我们还是会希望能够为该资产创建一个新的日志记录，并将该资产动作设置为“Acquired”。通常来讲，我们要么希望创建这些记录，亦或是，要是在出了什么问题时，这些记录一个都不创建，要做到这一点就必须使用事务操作。

先从开启一个事务开始。然后，插入一个新行并让它成为资产视图中的当前行。如果这确实是第一个资产，其 ID 就为 1，但如果还有其他的资产，其 ID 就会比最大资产 ID 的值大一。执行一个查询来查找当前最大的资产 ID 号，然后使用预查询（因此不必担心引用方面的问题），以便将新记录插入到日志表 logs 中。一旦新的记录插入到了日志表 logs 中，就可以提交事务了。现在，对于新的资产来说，就会有一条日志记录，且其动作是“Acquired”，还会有一个新的空白资产记录。最后，为新资产的名称域初始化好编辑功能。

通过查看 deleteAsset() 方法后，将会完成主窗体的学习。在这里，仅会对这个方法进行简单涉及，所以会分成三个部分来看。

```

def deleteAsset(self):
    index = self.assetView.currentIndex()
    if not index.isValid():
        return
    QSqlDatabase.database().transaction()
    record = self.assetModel.record(index.row())
    assetid = record.value(ID).toInt()[0]
    logrecords = 1
    query = QSqlQuery(QString("SELECT COUNT(*) FROM logs "
                             "WHERE assetid = %1").arg(assetid))

    if query.next():
        logrecords = query.value(0).toInt()[0]

```

先从启动一个事务开始。这是因为，如果要删除一个资产，那么它的所有日志记录也应该予以删除，而这样的两件事要么一起都发生，要么两者都不发生，以此来确保数据库的关系完整性。

众所周知，至少必须要有一个日志记录，即“Acquired”的那条记录，不过，还是会执行一下查询，以便能够知道到底有多少条日志记录。

```
msg = QString("<font color=red>Delete</font><br><b>%1</b>"  
             "<br>from room %2") \  
             .arg(record.value(NAME).toString()) \  
             .arg(record.value(ROOM).toString())  
if logrecords > 1:  
    msg += QString(", along with %1 log records") \  
             .arg(logrecords)  
msg += "?"  
if QMessageBox.question(self, "Delete Asset", msg,  
                         QMessageBox.Yes | QMessageBox.No) == QMessageBox.No:  
    QSqlDatabase.database().rollback()  
    return
```

在这里，会给用户一个确定删除操作或者是取消删除操作的机会。如果用户选择了取消，就回滚事务并返回。

```
query.exec_(QString("DELETE FROM logs WHERE assetid = %1") \  
             .arg(assetid))  
self.assetModel.removeRow(index.row())  
self.assetModel.submitAll()  
QSqlDatabase.database().commit()  
self.assetChanged(self.assetView.currentIndex())
```

通过使用 SQL 查询语句，已将该日志记录删除了，再使用模型 API，可以删除资产记录。删除之后，就可以提交事务并调用 `assetChanged()` 了，以便确保主-从视图能够显示出正确的日志记录。

对以上两者的删除，本来均可以采用模型 API 的。比如：

```
self.logModel.setFilter(QString("assetid = %1").arg(assetid))  
self.logModel.select()  
if self.logModel.rowCount() > 0:  
    self.logModel.removeRows(0, self.logModel.rowCount())  
    self.logModel.submitAll()
```

这样就完成了资产管理器应用程序 Asset Manager 的查看。创建各个表之间的主-从关系是相当简单的，而使用数据窗口部件映射器则也可以在表和窗体间创建主-从关系。

SQL 表模型非常容易使用并且可以与各个 `QTableView` 之间较好地协同工作。除此之外，可以创建自定义附属关系来实现对外表和域编辑的控制，此方法必要之处在于我们可以利用附属关系来提供记录的级别验证。

还有一个一直没有涉及的问题，就是新记录中唯一键 (unique key) 的创建。在这里，我们是通过在表中对 ID 域自动增量的方法，解决了这个问题。但有的时候，采用自动增量的方法是不合适的——例如，当键是一个比整数键复杂得多时，就不合适了。此时，就可以通过连接 `QSqlTableModel.beforeInsert()` 信号来处理这种情况了。这个信号会给出一种连接至待插入记录 (在用户对其编辑完成之后) 的引用方法，所以可以对任何域都进行填充或更改，只要是在数据实际插入到数据库之前就行。

还有其他一些可以连接的与 SQL 相关的信号——例如，`beforeDelete()` 和 `beforeUpdate()`；如果打算在一个单独的表中记录删除操作或者记录各个更改操作时，它们都可能会比较有用。最后，还有一个 `primeInsert()` 信号——当创建一条新的记录时，就会发射这种信号，但要在用户对其进行编辑之前。这就是这里所说的，可以用那些有益的默认值来填充各条记录。然而，这一章中的所有例子中，当用户单击添加 (Add) 按钮前，我们都已经为其设置

过了默认值。还需注意的是，由于 `QSqlRelationalTableModel` 是 `QSqlTableModel` 的一个子类，所以它也有这些信号。

## 小结

PyQt 凭借由 `QtSql` 模块所提供的统一的 API 为 SQL 数据库提供了强有力的支持。对于绝大多数广泛使用的数据库来说，PyQt 都为它们提供了数据库驱动程序，尽管对于某些数据库来说，可能受版权因素的限制而只能在商业版本中才会提供数据库驱动程序。

如果只是创建一个数据库连接，那么所有后续的数据库访问操作将都会使用该默认连接。但如果需要有多个数据库连接的话，则可以分别命名每个连接，然后在通过连接名称的方式来访问它们，以便可以说明用的是哪一个连接来完成某一特定动作的。

可以访问数据库的驱动程序，并通过访问来发现该数据库是否支持某些功能，比如 BLOB (Binary Large OBject，二进制大对象) 和事务操作这样的功能。不管底层数据库是哪种类型的数据库，PyQt 都允许我们使用基于 ODBC 和 Oracle 语法的预查询，也可以自动处理任何必要的变换和引用。PyQt 支持所有标准的 SQL 数据类型，并且在数据库自身不支持某个数据类型时，可以自动执行相应的必要转换。

`QSqlQuery` 类允许使用其 `exec_()` 方法执行任意的 SQL 语句。这就意味着，例如，可以使用它来创建和删除表，也可以使用它来插入、更新和删除记录。当执行一条 `SELECT` 语句时， `QSqlQuery` 对象可以为该语句生成的结果集提供一些导航方法，它们也可以为查询操作所涉及的若干行提供必要的信息——例如，有多少行被删除或者是得到了更新。

利用 `QDataWidgetMapper` 来创建 GUI 窗体以显示数据库表(或者是可编辑的视图)相当简单。通常会为每个外键域使用一个 `QComboBox`，为其内部模型赋予适当的关系模型。虽然它也可以通过设置 `QDataWidgetMapper` 来自动提交更改，但当用户导航时，这样做则可能会导致数据丢失，所以，倘若我们提供了自己的导航手段，就最好交由我们自己来提交所有的更改。

将 `QSqlTableModel` 或 `QSqlRelationalTableModel` 与 `QTableView` 联合使用来显示数据库表和视图，是比较容易的。这些类可以组合使用，以提供域数据的就地编辑功能。通过从模型中插入或删除行可以轻松实现记录的添加和删除，而当需要使用原子操作 (atomic action) 时，则可以使用事务操作。

PyQt 的模型/视图体系架构的所有功能对于数据库程序员来说都是可用的。此外，SQL 表模型中的那些 API 已为轻松实施数据库编程提供了大量拓展。当需要执行原始的 SQL 语句时，可以借助  `QSqlQuery` 类来轻松地实现这一目的。

现在，应当已经达到了创建任何想要的 GUI 应用程序的地步了，而能够制约的，只有想象力和可用的时间了。在第四部分，将会看到一些很难处理的其他主题，这些主题的实现一般需要模型/视图编程方面更高级的某些资源，然后会是有关国际化、互联网等方面的内容，而最后会以多线程为结束。

## 练习题

在一个引用表中创建一个对话框型应用程序来实现记录的添加、编辑和删除操作，就像图 15.7 中所示的效果。该应用程序应当在第一次运行的时候创建一个 `reference.db` 数据库，其中会带一个如下所示的简单空表：

```
CREATE TABLE reference (
    id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
    category VARCHAR(30) NOT NULL,
    shortdesc VARCHAR(20) NOT NULL,
    longdesc VARCHAR(80))
```

除了提供添加(Add)按钮和删除(Delete)按钮之外，还要提供一个排序(Sort)按钮，它带一个提供三种排序方法的弹出式菜单：通过 ID、通过类别和通过简短描述。这三个参数都可以通过 lambda 或 functools.partial 与同一个方法相连接。若要使任何新的排序方法(或过滤)生效，就必须在模型上调用 select()。对于所有的按钮都可以使用 QDialogButtonBox 来实现。

	Category	Short Desc.	Long Desc.
1	Actions	Wait	Wait for Information
2	Actions	Progress	Progress to Next Stage
3	Actions	Escalate	Send to Supervisor

Buttons at the bottom: Add, Delete, Sort, Close.

图 15.7 引用数据对话框

如果用户单击删除(Delete)按钮，就会弹出一个是/否(yes/no)的消息框，并且只有在用户单击 Yes 按钮时，才会执行真正的删除操作。这个应用程序与资产管理器应用程序 Asset Manager 中的 ReferenceDataDlg 十分类似，应该可以用大约 130 行的代码完成这一功能。

本章练习题的参考答案在文件 chap15/referencedata.pyw 中。



# 第四部分

## 高级 GUI 编程

---

- 第 16 章 高级模型/视图编程
- 第 17 章 在线帮助和国际化
- 第 18 章 网络应用
- 第 19 章 多线程

# 第 16 章 高级模型/视图编程

- 自定义视图
- 泛型委托
- 树中表达表格数据

在之前的两章中，讨论了 PyQt 模型/视图编程的基础知识<sup>①</sup>，了解了如何创建自定义模型，学习了如何使用预定义的 SQL 表格模型，还看到了如何通过创建自定义委托来控制数据项的外观和编辑。在这一章，将更加深入地学习 PyQt 的模型/视图编程的知识。

在这一章中所涉及的全部主题，以及这一章之后的相关内容，通常都会比之前所看到的内容更为高级，至少在概念上会如此。然而，对于大多数情况来说，从代码的复杂程度上来看，并没有比之前所见过的困难多少。

在 16.1 节将会看到如何实现一个自定义视图，以便可以看到是如何用我们想要的方式来可视化数据的。这一节对于理解视图的工作方式非常有帮助，同时还将看到一个简单的方法来实现一个自定义视图。

16.2 节将会重新回到自定义委托这一主题，从中将会看到如何减少代码的重用度以及如何轻松为视图创建一些任意委托。这一节对于那些想要创建大量委托的程序开发人员来说应该会大有帮助，尤其是对诸如 SQL 表格中的每列都是一种特殊类型的数据集来说更是如此。

在最后一节将会学习如何把表格类型的数据反映到树视图中。一种用法是，当把表格表示成树而一开始的一些列通常会包含同样值的时候——这样做就可以显著减少用户找到想要数据项时必须导航的行数。另一种用法是，可以让用户通过选择一些特定值来构成“路径”。例如，不是提供两个、三个或者更多的组合框，而是每个组合框的值都会取决于它的父对象的当前值，进而只会为用户提供一个用做导览和选择的树。

## 16.1 自定义视图

PyQt 提供的几个视图类都可以较好工作，包括 `QListView`、`QTableView` 和 `QTreeView`。这些视图所具有的共同点之一是，它们通常都用来以文本的方式来展示数据项——尽管如若需要的话，它们也都可以显示图标和复选框。对以文本方式展示数据项的替代方案是图形化表示法，这可以使用第 12 章中介绍的图形视图类来实现。不过，有时，确实难以使用类中现有的方式来呈现数据。这时候，就可以通过创建我们自己的视图子类并将其用做模型数据的可视化来解决这一问题。

在图 16.1 中用两种不同的视图来展示同一个数据集。左边的视图是一个标准的 `QTableView`，而右边的视图是一个自定义的 `WaterQualityView`。这两个视图都可以用文本的形式显示同一时刻读取的水质数据，只不过是，`WaterQualityView` 会用三个彩色圆圈来显示三

<sup>①</sup> 这一章会认为已经对 PyQt 的模型/视图架构有了基本认识，相关知识可以参阅第 14 章。

个关键指标，并且会用 Unicode 箭头符号来表示特殊的流动状况，显然，表格视图对数据的表达清晰而直观，可以更为简单地看清事实，而且 WaterQualityView 仅仅通过颜色就可以让人轻松获得重要趋势的直观印象。

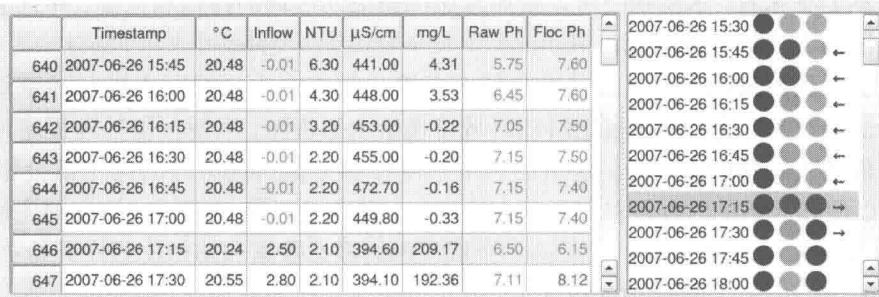


图 16.1 水质数据的两种视图

这里的水质数据集是一个小型污水净化处理厂半年内的水质数据——不过，这些数据每 15 分钟获取一次，这样累加起来就有超过 17 500 条的数据量。这就意味着，每个视图都需要有一个垂直滚动条。PyQt 提供了三种方式来获取滚动条。第一种方式是，创建一个继承自 QAbstractScrollArea 的窗口部件；这一方法通常由 QGraphicsView 和 QTextEdit 之类的窗口部件使用。第二种方法是，创建一个含有多个 QScrollBar 的复合窗口部件。不过，PyQt 文档建议使用更为简单的第三种方法——用 QScrollArea 来代替。使用 QScrollArea 的一个缺点是，这是 PyQt 各个类中为数不多的并非设计用于子类化的一个类。因此，要想带一些滚动条，就必须创建一个实例并将其添加到窗口部件中。回到这里的例子上，就是将其放到水质数据应用程序 (Water Quality Data application) 的初始化程序中：

```
class MainForm(QDialog):
    def __init__(self, parent=None):
        super(MainForm, self).__init__(parent)
        self.model = WaterQualityModel(os.path.join(
            os.path.dirname(__file__), "waterdata.csv.gz"))
        self.tableView = QTableView()
        self.tableView.setAlternatingRowColors(True)
        self.tableView.setModel(self.model)
        self.waterView = WaterQualityView()
        self.waterView.setModel(self.model)
        scrollArea = QScrollArea()
        scrollArea.setBackgroundRole(QPalette.Light)
        scrollArea.setWidget(self.waterView)
        self.waterView.scrollarea = scrollArea
        splitter = QSplitter(Qt.Horizontal)
        splitter.addWidget(self.tableView)
        splitter.addWidget(scrollArea)
        splitter.setSizes([600, 250])
        layout = QHBoxLayout()
        layout.addWidget(splitter)
        self.setLayout(layout)
        self.setWindowTitle("Water Quality Data")
        QTimer.singleShot(0, self.initialLoad)
```

以上代码就是整个程序中的全部内容。WaterQualityModel 是 QAbstractTableModel 的一个子类，可为水质数据文件提供只读接口。WaterQualityView 是一个在这一节中将要开发的类。这里所做的一件特殊事情是，创建一个 QScrollArea 窗口部件并向其添加水质视图——这基本上意味着，水质视图可以按照意愿调整滚动区域的高度和宽度，并且滚动区会自行处理有关滚动方面的各类问题。

很快就会看到键盘用户可以通过键盘的上下箭头键来在水质视图中进行导览切换，要确保选中的行总是可见，就必须通过把滚动区域传递给水质视图，以便按键处理程序可以与其交互。另外一件比较特殊的事情是，这里已为水平窗口切分条(splitter)的两个部分给定了初始尺寸大小，以便在开始的时候，它们可以大致与它们所包含的窗口部件成恰当的比例。

现在先来查看 WaterQualityView，会从程序的静态数据和初始化程序开始：

```
class WaterQualityView(QWidget):
    FLOWCHARS = (unichr(0x21DC), unichr(0x21DD), unichr(0x21C9))

    def __init__(self, parent=None):
        super(WaterQualityView, self).__init__(parent)
        self.scrollarea = None
        self.model = None
        self.setFocusPolicy(Qt.StrongFocus)
        self.selectedRow = -1
        self.flowfont = self.font()
        size = self.font().pointSize()
        if platform.system() == "Windows":
            fontDb = QFontDatabase()
            for face in [face.toLower() for face in fontDb.families()]:
                if face.contains("unicode"):
                    self.flowfont = QFont(face, size)
                    break
        else:
            self.flowfont = QFont("symbol", size)
        WaterQualityView.FLOWCHARS = (
            chr(0xAC), chr(0xAE), chr(0xDE))
```

把焦点策略设置成任意(除 Qt.NoFocus 之外)意味着窗口部件可以接受键盘焦点。接下来将会讨论为什么要这么做，也会在这一节的最后讨论 selectedRow 实例变量。

当水流方向错误，或者水流太慢，或者水流太快，希望能够通过适当的字符来显示这些状态——例如，→、←以及⇒。这些字符在 Unicode 中都是可用的，不过由 Windows 提供的大部分默认字体中似乎并没有完全包含整个 Unicode 字符集，所以所有的箭头都会显示成字符。(在 Linux 中，如果一个 Unicode 字符在当前字体中不可用，PyQt 通常可以在另外一个字体中找到相应的字符，此时，Linux 就会使用找到的字体来代替那个字符)。

为了解决 Windows 上的这个问题，将会遍历所有可用字体的列表，直到找到一个名字中带有“Unicode”的字体(例如，“Lucida Sans Unicode”)。如果能够找到一个这种字体，就将它作为水流字符的字体；否则，将返回到标准的(但非 Unicode)Symbol 字体，并使用该字体中最为接近的相应字符。

```
def setModel(self, model):
    self.model = model
    self.connect(self.model,
                SIGNAL("dataChanged(QModelIndex, QModelIndex)"),
                self.setNewSize)
```

```
self.connect(self.model, SIGNAL("modelReset()"),
             self.setNewSize)
self.setNewSize()
```

一旦把一个模型作用于一个视图上，就需要连接该视图的数据改变信号，以便可以改变视图的尺寸大小来匹配那些可用的数据。

```
def setNewSize(self):
    self.resize(self.sizeHint())
    self.update()
    self.updateGeometry()
```

这个方法会把视图的尺寸大小调整为较佳的尺寸大小，并且会调用 `update()`，通过安排重绘和 `updateGeometry()` 来告知管理该视图的布局管理器，视图的尺寸大小已经发生了改变。因为视图是放在 `QScrollArea` 中的，滚动区域将会通过调整它所提供的滚动条来响应尺寸大小所发生的改变。

```
def minimumSizeHint(self):
    size = self.sizeHint()
    fm = QFontMetrics(self.font())
    size.setHeight(fm.height() * 3)
    return size
```

这里会计算视图的最小尺寸大小，使其在宽度上使用首选尺寸大小的宽度值，在高度上使用三个字符的高度。在布局中这是有道理的，不过由于使用的是 `QScrollArea`，实际上，最小尺寸大小将由滚动区域决定。

```
def sizeHint(self):
    fm = QFontMetrics(self.font())
    size = fm.height()
    return QSize(fm.width("9999-99-99 99:99 ") + (size * 4),
                (size / 4) + (size * self.model.rowCount()))
```

这里用一个字符的高度(包括它的行间距)作为垂直和水平度量体系的尺寸大小单位。视图的首选尺寸大小的宽度足以显示一个时间戳外加四个尺寸大小单位，以便能够容纳彩色圆圈和流向字符，它的高度足以容纳模型中的所有行外加四分之一的尺寸大小单位，以便可以留出来一点空白。

绘制事件不是太困难，不过这里会将其分成三个部分，而且只会给出一个彩色圆圈的代码，因为这三个圆圈的代码几乎是完全一样的。

```
def paintEvent(self, event):
    if self.model is None:
        return
    fm = QFontMetrics(self.font())
    timestampWidth = fm.width("9999-99-99 99:99 ")
    size = fm.height()
    indicatorSize = int(size * 0.8)
    offset = int(1.5 * (size - indicatorSize))
    minY = event.rect().y()
    maxY = minY + event.rect().height() + size
    minY -= size
    painter = QPainter(self)
    painter.setRenderHint(QPainter.Antialiasing)
    painter.setRenderHint(QPainter.TextAntialiasing)
```

如果没有模型，就什么也不做并直接返回。否则，需要对一些尺寸大小进行计算。就像 `sizeHint()` 一样，会用一个字符的高度作为尺寸大小的单位，把 `indicatorSize`(彩色圆圈的直

径)设置成这个值的 80%。偏移量是垂直空间上的一个小值,设计用来让各个圆圈能够在垂直方向上与时间戳文本相对齐。

考虑到视图可能会用来显示大型数据集,只去绘制那些用户完全或者部分可见的项看起来会显得比较明智。出于这一原因,会将最小  $y$  坐标设置成绘制事件矩形的  $y$  坐标(但会减去一个尺寸大小单位),并且将最大  $y$  坐标设置成最小值加上绘制事件的高度再加一个尺寸大小单位。这就意味着,将会从所有可见的[即范围中最小  $y$  坐标的那个项,因为点(0, 0)位于左上角]最顶端项上方的项开始绘制,向下绘制到所有可见的(即范围中最大  $y$  坐标的那个项)最底部项的下方项。

绘制事件的事件参数含有需要重绘区域的尺寸大小。通常会忽略这些信息,只是简单绘制整个窗口部件,不过有的时候,会使用这些信息来让绘制事件更有效率些。

```

y = 0
for row in range(self.model.rowCount()):
    x = 0
    if minY <= y <= maxY:
        painter.save()
        painter.setPen(self.palette().color(QPalette.Text))
        if row == self.selectedRow:
            painter.fillRect(x, y + (offset * 0.8),
                            self.width(), size,
                            self.palette().highlight())
        painter.setPen(self.palette().color(
            QPalette.HighlightedText))
        timestamp = self.model.data(
            self.model.index(row, TIMESTAMP)).toDateTime()
        painter.drawText(x, y + size,
                        timestamp.toString(TIMESTAMPFORMAT))
        x += timestampWidth
        temperature = self.model.data(
            self.model.index(row, TEMPERATURE))
        temperature = temperature.toDouble()[0]
        if temperature < 20:
            color = QColor(0, 0,
                           int(255 * (20 - temperature) / 20))
        elif temperature > 25:
            color = QColor(int(255 * temperature / 100), 0, 0)
        else:
            color = QColor(0, int(255 * temperature / 100), 0)
        painter.setPen(Qt.NoPen)
        painter.setBrush(color)
        painter.drawEllipse(x, y + offset, indicatorSize,
                            indicatorSize)
        x += size

```

这里会遍历模型中的每一行,但只对范围内带有  $y$  坐标的项进行绘制。一旦有一行需要绘制,就会把画笔(用来绘制文本)设置成调色板中文本的颜色。如果该行是选中的(在介绍完绘制事件后将会做一些说明),就会用调色板中的高亮色来绘制背景,并且会把画笔设置成调色板的高亮文本的颜色。

设置了文本的颜色,可能也会绘制背景色,然后就会获取并绘制该行的时间戳。对于每一行,会通过一个  $x$  坐标来记录已经经过的距离,据此通过字体度量值来增加之前已经计算过的时间戳的宽度。

第一个彩色圆圈用来表明水的温度,单位是摄氏度(°C)。如果水太凉,会表示成色调为

蓝色的颜色；如果太热，会表示成色调为红色的颜色；否则，使用绿色色调。然后，关掉画笔并且把画刷的颜色设置成前面计算的值，并且在时间戳的右边画一个椭圆。`drawEllipse()`方法将会画一个圆，因为被画的椭圆里的宽度值和高度值与矩形的宽度值和高度值一样。

然后，增加  $x$  坐标的值。现在，对于其他两个彩色圆圈指示符的绘制，可以继续重复这一过程，并采用了与温度中一样的色调处理方法。这里省略了这些代码，因为结构上这些代码与用于温度圆圈的代码完全相同。

```
flow = self.model.data(  
    self.model.index(row, INLETFLOW))  
flow = flow.toDouble()[0]  
char = None  
if flow <= 0:  
    char = WaterQualityView.FLOWCHARS[0]  
elif flow < 3:  
    char = WaterQualityView.FLOWCHARS[1]  
elif flow > 5.5:  
    char = WaterQualityView.FLOWCHARS[2]  
if char is not None:  
    painter.setFont(self.flowfont)  
    painter.drawText(x, y + size, char)  
    painter.restore()  
y += size  
if y > maxY:  
    break
```

如果水流的方向是错误的，或者如果它太快或者太慢，将会使用在初始化程序中设置的字体和字符来绘制一个适当的字符。

最后，增加  $y$  坐标，以便为下一行的数据做好准备，但是，如果绘制的已经是视图中的最后一行，那么只需停止即可。

到目前为止所编写的代码已经足以以为数据集提供一个只读视图。不过用户经常希望能够高亮一个项。对于此问题最简单的方法是，添加一个鼠标按下事件处理程序。

```
def mousePressEvent(self, event):  
    fm = QFontMetrics(self.font())  
    self.selectedRow = event.y() // fm.height()  
    self.update()  
    self.emit(SIGNAL("clicked(QModelIndex)"),  
              self.model.index(self.selectedRow, 0))
```

字符的高度是计算的尺寸大小单位。鼠标位置的  $y$  坐标(相对于窗口部件的左上角)就是根据尺寸大小单位来划分的，以用来找到用户点击的是哪一行。使用整数除法的原因是，行数都是整数。接着，调用 `update()` 来安排一个绘制事件。在 `paintEvent()` 中，可以看到，选中行会用背景颜色和文字高亮颜色进行绘制。还会发送一个带有点选行第一列模型索引值的 `clicked()` 信号。在这个应用程序中并没有使用到该信号，不过在实现自定义视图时，提供它是一种很不错的做法。

滚动区域也会照顾到键盘用户：他们可以通过 `Page Up` 键和 `Page Down` 键来进行滚动。不过仍然应当提供一种方法来使键盘用户能够选中某一项。为了做到这一点，必须确保该窗口部件具有适当的焦点策略——这是在初始化程序中完成的——并且必须提供一个按键事件处理程序。

```

def keyPressEvent(self, event):
    if self.model is None:
        return
    row = -1
    if event.key() == Qt.Key_Up:
        row = max(0, self.selectedRow - 1)
    elif event.key() == Qt.Key_Down:
        row = min(self.selectedRow + 1, self.model.rowCount() - 1)
    if row != -1 and row != self.selectedRow:
        self.selectedRow = row
        if self.scrollarea is not None:
            fm = QFontMetrics(self.font())
            y = fm.height() * self.selectedRow
            self.scrollarea.ensureVisible(0, y)
        self.update()
        self.emit(SIGNAL("clicked(QModelIndex)"),
                  self.model.index(self.selectedRow, 0))
    else:
        QWidget.keyPressEvent(self, event)

```

这里选择只支持两个按键：向上箭头(Up Arrow)和向下箭头(Down Arrow)。如果用户按下了这两个键中的任何一个，在确保选中的行是在有效范围之内后，将会向上或者向下选中行，并且接着会安排一个绘制事件。如果用户导航切换选中行的时候，向上超过了最上面的一行，或者向下超过了最下面的一行，将会通知滚动区域要确保所选中的行可以显示出来——如果有必要，滚动区域将会通过滚动来实现这一点。同时，还会发射一个带有最新选中行的模型索引值的 clicked() 信号。在这种情况下，使用 clicked() 信号是相当常见的方法，因为实际上，用户是通过键盘实现“点击”操作的——毕竟，信号和槽机制所关心的主要是用户想要做什么，而不是要求用户是怎么做到的，而在这里，用户只是希望能够选中一行。

如果不是自己处理键盘按下事件的，也就是，对于所有其他键盘按下事件，都会将按键事件传递给基类。

水质视图窗口部件在外观上看起来确实与它里面显示的表格视图有很多不同，然而它的实现并不需要很多代码，编程也不算困难。通过减少不必要的绘制，已经可以较大程度地提高了该窗口部件的效率。同时，通过窗口部件总是能够确保显示整个数据集的尺寸大小，还可以让绘制的代码尽可能简单。这种方法的缺点是，会将责任推给使用这个窗口部件的程序开发人员，他需要借助 QScrollArea，尽管这可以让我们不用再靠自己去实现滚动操作。

水质视图可以将数据与模型中的数据可视化地一一对应起来，但是并不局限于这样做。创建可显示聚合数据的自定义视图也是可以的。在这个例子中，例如，本来也可以是每天或者每小时显示一条数据，通过对每天或者每小时的读数进行平均就可以做到这一点。

## 16.2 泛型委托

正如在之前的各章中所看到的那样，自定义委托可以让我们对视图中出现的数据项的外观和行为进行完全控制。虽然很明显，如果有很多模型，可能会希望不是全部的大多数模型能够仅用一个自定义委托，如果不能这么做，那么对于这些自定义委托，将很有可能存在大量的重复代码<sup>①</sup>。

<sup>①</sup> 这一节的部分思想源自“Qt 4 的模型/视图型委托”一文作者的白皮书，详情可见 <http://www.ics.com/developers/papers/>。

假设只有四个模型，每一个模型都有一个整数 ID 列、一些含有纯文本的字符串列以及一个含有 HTML 文本的说明列，并且对于某些模型来说，会有一到两个浮点数列。所有模型都用 ID 作为它们的第一列，但其他各列则并不一致，所以每个模型都需要有自己的自定义委托。提供这些自定义委托并不会成为什么大的负担，但是，这些模型中用于处理整数 ID 的代码，则对于所有的模型来说可能都是相同的；同样，用于处理字符串、HTML 字符串以及浮点数的代码应该也是一样的。

现在，假设还必须为另外数十个新的模型编写它们的自定义委托，那么大部分的代码将会被再次重复一遍——这很有可能会使维护工作变得更为困难。

那么对于模型，特别是对于那些每行都有一种数据类型的模型来说，如数据库表，更好的方法就是不要为每个模型创建一个自定义委托，而是用一系列的通用组件来共同构成一个委托。这就意味着，维护工作就会仅局限于对通用组件的维护，而一个错误修复将会自动使任何使用了它的视图都受益。

在代码中，实现代码可能会是如下的样子：

```
self.table1 = QTableView()
self.table1.setModel(self.model1)
delegate1 = GenericDelegate(self)
delegate1.insertColumnDelegate(1, PlainTextColumnDelegate())
delegate1.insertColumnDelegate(2, PlainTextColumnDelegate())
delegate1.insertColumnDelegate(3, RichTextColumnDelegate())
delegate1.insertColumnDelegate(4, IntegerColumnDelegate())
self.table1.setItemDelegate(delegate1)

self.table2 = QTableView()
self.table2.setModel(self.model2)
delegate2 = GenericDelegate(self)
delegate2.insertColumnDelegate(1, PlainTextColumnDelegate())
delegate2.insertColumnDelegate(2, IntegerColumnDelegate())
delegate2.insertColumnDelegate(3, FloatColumnDelegate())
delegate2.insertColumnDelegate(4, FloatColumnDelegate())
delegate2.insertColumnDelegate(5, RichTextColumnDelegate())
self.table2.setItemDelegate(delegate2)
```

这里有两个单独的模型，但两者都使用了由预定义列委托构成的泛型委托(generic delegate)，其中的列委托才会与数据类型相关。

利用这一方法，对于每种打算处理的数据类型，如整数、浮点数、日期、时间和日期/时间，曾经都需要分别创建纯文本列委托、Rich 文本列委托，等等。除此以外，或许会为处理自定义类型而需要创建一些具体的列委托，但对于任何给定数据类型都只有一个列委托，这样就大大减少了代码的重复度，并且只需使用添加适当列委托的泛型委托，就可以确保任何模型都会有一个“自定义”委托。

在这一节中，将会看到如何创建一个 GenericDelegate 类，还会看到几个列委托的例子。然后，将会学习如何在如图 16.2 所示的应用程序的上下文中使用它们。

GenericDelegate 类非常简单，因为它基本上把所有的工作都转给其他类了。

```
class GenericDelegate(QItemDelegate):

    def __init__(self, parent=None):
        super(GenericDelegate, self).__init__(parent)
        self.delegates = {}
```

初始化程序会像往常一样调用 super() 并创建一个空的字典。字典的键将会是列的索引值，而值则是 QItemDelegate 子类的实例。

```
def insertColumnDelegate(self, column, delegate):
    delegate.setParent(self)
    self.delegates[column] = delegate
```

	License	Customer	Hired	Mileage #1	Returned	Mileage #2	Notes	Miles	Days
37	FT56 LNC	Ms Inglis	2006-06-06	13199	2006-06-22	17055		3856	16
38	NN07 AUP	Mr Berrett	2006-06-07	18032	2006-06-26	20901	Customer complained about the gears	2869	19
39	YR56 JRK	Ms Boston	2006-06-09	25000	2006-06-13	25540		540	4
40	BQ07 WRM	Mr Terhune	2006-06-10	17015	2006-07-01	22769		5754	21
41	UN06 DUE	Ms Harvill	2006-06-11	14374		Sold			
42	DV58 QEJ	Ms Jolly	2006-06-11	17142	2006-06-21	19012		1870	10
43	JJ57 TDP	Ms Clemy	2006-06-12	23996	2006-06-20	26364	Returned damaged	2368	8
44	QD06 XBQ	Ms Courtice	2006-06-12	17063	2006-06-15	17600		537	3
45	SJ06 IAQ	Ms Garraway	2006-06-12	12966	2006-06-14	13508	Returned with empty fuel tank	542	2
46	JY56 RVW	Ms Verney	2006-06-14	22085	2006-06-28	26257	Returned dirty	4172	14

图 16.2 使用了泛型委托的表格视图

当一个新的列委托插入到泛型委托中时，该泛型委托就会拥有它并把它插入到字典中。

```
def removeColumnDelegate(self, column):
    if column in self.delegates:
        del self.delegates[column]
```

包含这个方法是出于完整性考虑的，但貌似不会用到它。如果删除一个列委托，泛型委托将会只对该列使用 QItemDelegate 基类。

```
def paint(self, painter, option, index):
    delegate = self.delegates.get(index.column())
    if delegate is not None:
        delegate.paint(painter, option, index)
    else:
        QItemDelegate.paint(self, painter, option, index)
```

这个方法的结构是 GenericDelegate 类工作方式的关键。先从获得给定列的列委托开始。如果得到了列委托，就将工作交给它；否则，就将工作传递给它的基类。

```
def createEditor(self, parent, option, index):
    delegate = self.delegates.get(index.column())
    if delegate is not None:
        return delegate.createEditor(parent, option, index)
    else:
        return QItemDelegate.createEditor(self, parent, option,
                                         index)
```

这个方法遵循与 paint() 方法相同的模式，只是除了它会返回一个值之外（会为它创建一个编辑器）。

```
def setEditorData(self, editor, index):
    delegate = self.delegates.get(index.column())
    if delegate is not None:
        delegate.setEditorData(editor, index)
    else:
        QItemDelegate.setEditorData(self, editor, index)
```

```
def setModelData(self, editor, model, index):
    delegate = self.delegates.get(index.column())
    if delegate is not None:
        delegate.setModelData(editor, model, index)
    else:
        QItemDelegate.setModelData(self, editor, model, index)
```

最后的这两个 GenericDelegate 方法会遵循与 paint() 和 createEditor() 方法相同的模式，如果其中任何一个已经设置用于给定的列，就使用该列委托，否则，就使用 QItemDelegate 基类。

现在已经看完了 GenericDelegate 的实现，可以将注意力转向插入其中的列委托上了。在文件 chap16/genericdelegates.py 中会提供 IntegerColumnDelegate、DateColumnDelegate、PlainTextColumnDelegate 和 RichTextColumnDelegate 这几个类。它们全部都有着相似的结构，所以这里只会查看它们中的两个类，一个是 DateColumnDelegate，另一个是 RichTextColumnDelegate。一旦理解了它们的实现方式（这比较容易，至少对于日期列委托来说会是这样的），再来创建其他的列委托，比如创建一个用于浮点数的列委托，就很简单了。

```
class DateColumnDelegate(QItemDelegate):
    def __init__(self, minimum=QDate(), maximum=QDate.currentDate(),
                 format="yyyy-MM-dd", parent=None):
        super(DateColumnDelegate, self).__init__(parent)
        self.minimum = minimum
        self.maximum = maximum
        self.format = QString(format)
```

对于日期，希望能够提供最小值和最大值，还要有显示格式。

```
def createEditor(self, parent, option, index):
    dateedit = QDateEdit(parent)
    dateedit.setDateRange(self.minimum, self.maximum)
    dateedit.setAlignment(Qt.AlignRight|Qt.AlignVCenter)
    dateedit.setDisplayFormat(self.format)
    dateedit.setCalendarPopup(True)
    return dateedit
```

依据一般模式创建编辑器的代码可以回看第 14 章：使用给定的父对象创建编辑器，对其进行设置，然后再将其返回。在这里，用到了传送给初始化程序的最大值、最小值和带格式的值。

```
def setEditorData(self, editor, index):
    value = index.model().data(index, Qt.DisplayRole).toDate()
    editor.setDate(value)
```

编辑器的值会设置成数据项的值，该数据项由给定的模型索引值确定。不需要进行列检查，因为这个列委托仅会用于用户指定的列的 GenericDelegate 调用。

```
def setModelData(self, editor, model, index):
    model.setData(index, QVariant(editor.date()))
```

当把编辑器的数据写回给模型时，仍不需要进行列检查，因为 GenericDelegate 会处理这件事。

这就是完整的 DateColumnDelegate。并不需要重新实现 paint() 方法，因为 QItemDelegatebase 类可以相当完美地绘制出这些数据。IntegerColumnDelegate 和 PlainTextColumnDelegate 都与 DateColumnDelegate 非常相似。RichTextColumnDelegate 也类似，不过，它也需要重新实现 paint() 和 sizeHint() 方法。

```
class RichTextColumnDelegate(QItemDelegate):
    def __init__(self, parent=None):
        super(RichTextColumnDelegate, self).__init__(parent)
```

构造函数甚至比其他列委托中所用到构造函数还要简单。本来是可以直接忽略的，但希望能够更清楚些，才将其放到这里。

```
def createEditor(self, parent, option, index):
    lineedit = richtextlineedit.RichTextLineEdit(parent)
    return lineedit
```

这里使用了第 13 章中创建的 RichTextLineEdit 类。从结构上来看，这个方法与其他列委托完全相同，只是不再需要特定的方式对编辑器进行设置而已。

```
def setEditorData(self, editor, index):
    value = index.model().data(index, Qt.DisplayRole).toString()
    editor.setHtml(value)
```

如果使用它的 setHtml() 方法的话，RichTextLineEdit 可以接受 HTML 文本（它还有一个 setPlainText() 方法）。

```
def setModelData(self, editor, model, index):
    model.setData(index, QVariant(editor.toSimpleHtml()))
```

RichTextLineEdit 有一个 toHtml() 方法，不过这里会使用在第 13 章中开发的 toSimpleHtml() 方法。这样就可以确保在存储完美表达文本时尽可能简化 HTML 的代码。这一点很重要，因为在 paint() 方法中，对于高亮显示的（比如，选中的）项，将可以通过前后封装一个 <font> 标记来设置文本的颜色——对于使用了简单 HTML 格式的文本来说，这是可以正常工作的，因为要处理的仅仅是一个 HTML 片段而已，不过，却不能用于一般 HTML 格式，因为那是一个完整的 HTML 文档。

```
def paint(self, painter, option, index):
    text = index.model().data(index, Qt.DisplayRole).toString()
    palette = QApplication.palette()
    document = QTextDocument()
    document.setDefaultFont(option.font)
    if option.state & QStyle.State_Selected:
        document.setHtml(QString("<font color=%1>%2</font>") \
            .arg(palette.highlightedText().color().name()) \
            .arg(text))
    else:
        document.setHtml(text)
    painter.save()
    color = palette.highlight().color() \
        if option.state & QStyle.State_Selected \
        else QColor(index.model().data(index,
            Qt.BackgroundColorRole))
    painter.fillRect(option.rect, color)
    painter.translate(option.rect.x(), option.rect.y())
    document.drawContents(painter)
    painter.restore()
```

paint() 方法和第 14 章中给出的 ShipDelegate 的使用方式几乎一样。唯一不同的是，无需再去检查特定的列，因为列委托仅会在列的使用者给定时才会得到调用。

这一方法的一个局限之处在于，只能对 HTML 片段进行高亮显示。如果要想让这部分代码既可作用于 HTML 片段又可用于整个 HTML 文档，则可以像这样来使用代码：

```
if option.state & QStyle.State_Selected:
    if text.startsWith("<html>"):
        text = QString(text).replace("<body ",
            QString("<body bgcolor=%1 ")
            .arg(palette.highlightedText().color().name()))
    else:
        text = QString("<font color=%1>%2</font>")\
```

```
.arg(palette.highlightedText().color().name())\n    .arg(text))\ndocument.setHtml(text)
```

另外一种方法是，提取文本文档的样式表，更新背景颜色，并且将更新后的样式表重新作用于文档上。

```
def sizeHint(self, option, index):\n    text = index.model().data(index).toString()\n    document = QTextDocument()\n    document.setDefaultFont(option.font)\n    document.setHtml(text)\n    return QSize(document.idealWidth() + 5,\n                option.fontMetrics.height())
```

Rich 文本列尺寸大小提示的计算必须靠自己，因为默认的计算过程是基于窗口部件的字体尺寸大小和字符数的，通常会给出一个过宽的宽度值。这是因为，HTML 文本通常包含的字符数（如 HTML 标记和字符实体等）要比实际显示的字符数多得多。通过使用 `QTextDocument` 可轻松解决这一问题。这些代码几乎和用于 `ShipDelegate` 中尺寸大小提示的代码一样。

其他列委托的创建比较容易，本来是可以让任何列委托都提供比这里所给出的例子更多功能的。例如，对于 `IntegerColumnDelegate` 可以提供最大值和最小值，不过再额外提供一些其他功能选项也是非常容易的，比如文本的前缀和后缀。

现在已经知道了 `GenericDelegate` 是如何工作的以及列委托是如何创建的，还可以来看看它们是如何在实践中使用的。图 16.2 给出了一个表格视图，使用了一个带有几个列委托的泛型委托，可以对其列的编辑和外观实现完全控制。该窗体放在文件 `chap16/carhire-log.pyw` 中；这里是其初始化程序的开头部分：

```
class MainForm(QMainWindow):\n\n    def __init__(self, parent=None):\n        super(MainForm, self).__init__(parent)\n\n        model = CarHireModel(self)\n\n        self.view = QTableView()\n        self.view.setModel(model)\n        self.view.resizeColumnsToContents()\n\n        delegate = genericdelegates.GenericDelegate(self)\n        delegate.insertColumnDelegate(CUSTOMER,\n            genericdelegates.PlainTextColumnDelegate())\n        earliest = QDate.currentDate().addYears(-3)\n        delegate.insertColumnDelegate(HIRED,\n            HireDateColumnDelegate(earliest))\n        delegate.insertColumnDelegate(MILEAGEOUT,\n            MileageOutColumnDelegate(0, 1000000))\n        delegate.insertColumnDelegate(RETURNED,\n            ReturnDateColumnDelegate(earliest))\n        delegate.insertColumnDelegate(MILEAGEBACK,\n            MileageBackColumnDelegate(0, 1000000))\n        delegate.insertColumnDelegate(NOTES,\n            genericdelegates.RichTextColumnDelegate())\n\n        self.view.setItemDelegate(delegate)\n        self.setCentralWidget(self.view)
```

这里的模型是一个自定义模型，类似于在第 14 章中创建的自定义模型。这里的视图是一个标准的 `QTableView`。

这个模型一共有 9 列：一个纯文本的驾驶证号码、纯文本的客户名、租赁日期（是一个 `QDate`）、表示起始里程的整数（去程）、归还日期（是一个 `QDate`）、表示归还里程的整数（归程）、Rich 文本型的备注，还有两个生成的而不是存储的列——一个列是里程数（就是去程和回程之差），一个列是天数（就是归还日期和租赁日期之差）。

这个模型以及底层的数据结构会处理驾驶证号码列和生成的两个列（后面很快就会讨论到），所以只需要为可编辑的各列提供一些列委托即可。对于客户名称会使用 `PlainTextColumnDelegate`，而对于备注会使用 `RichTextLineEdit`。不过，对于日期和里程则会使用几个自定义列委托，它们都是 `IntegerColumnDelegate` 和 `DateColumnDelegate` 的子类。需要使用这些子类来提供列之间的交叉验证。例如，不允许归还日期早于租赁日期，或者归还里程数小于起始里程数。

```
class HireDateColumnDelegate(genericdelegates.DateColumnDelegate):  
    def createEditor(self, parent, option, index):  
        i = index.sibling(index.row(), RETURNED)  
        self.maximum = i.model().data(i, Qt.DisplayRole) \  
                      .toDate().addDays(-1)  
        return genericdelegates.DateColumnDelegate.createEditor(  
            self, parent, option, index)
```

这就是 `HireDateColumnDelegate` 子类的全部代码。需要做的只是重新实现 `createEditor()` 方法。获取归还日期并且把最大租赁日期设置为汽车归还的前一天，因为最小值是汽车一日租赁。实际上，这里将编辑器的创建过程留给基类处理。当创建列委托时，还不能设置一个有意义的最大日期，因为用户有可能会在任何时候编辑归还日期的，因此，必须在用户开始编辑它的时候计算最大租赁日期。

使用 `sibling()` 方法可以为我们调用 `index.model().index(index.row(), RETURNED)` 提供更为方便的替代方法。

`ReturnDateColumnDelegate` 几乎完全一样，只是在获取租赁日期并把最小归还日期设置成汽车租赁后的第二天这两个地方有所不同。`MileageOutColumnDelegate` 和 `MileageBackColumnDelegate` 很相似；它们两个都只是重新实现了 `createEditor()` 方法，并且也都是根据其他的里程值来设置最大值（或者最小值）。

这个模型的 `setData()` 方法不允许编辑牌照号，或者编辑生成的两列。通过让这些列简单返回一个 `False` 就可以做到这一点，因为这会表明它们都没有被更新过。对于其他各列，传给 `setData()` 方法的值会在底层数据结构中进行设置。

模型的 `data()` 方法会如实地返回要求列的值，只需通过底层数据结构就可以提供该功能。该数据结构可以为大多数的列返回存储的值，不过，对于里程数列 `MILEAGE` 和天数列 `DAYS`，它可以返回通过相关值计算后的值。

创建通用的与数据类型相关的列委托相当容易，当必须考虑整行（记录）的验证功能时，对它们进行子类化也并不困难。不过，由于为每个模型创建一个自定义委托并不困难，为什么还要使用泛型委托方法呢？从上下文来看，这里主要有两个泛型委托没有说清楚的地方：对于简单的应用程序来说，需要的可能只是少数几个委托，而且只有几个会含有异构数据类型的

列模型。不过，对于需要大量委托单的应用程序来说，其中的列很多都会含有异构数据类型，比如用于数据库中的数据类型，泛型委托就可以提供三个关键的好处。

- 可以轻松修改用于某一特定列中的委托，或者，如果需要修改模型使其含有更多列时，就可以轻松添加那些额外的列委托。
- 使用列委托就意味着可以避免代码重复，而如果创建许多与模型相关的自定义委托，代码重复将是不可避免的——例如，之前的例子中就只是编写了一个 Rich 文本行编辑委托、一个日期/时间编辑委托等。
- 一旦创建了一个与数据类型相关的列委托，就可以将它用于使用了该数据类型的任何列上，或者用在带有任意数量模型的任何泛型委托上。这就意味着，任何的错误修复和性能增强只需应用于每个数据类型的列委托上即可。

### 16.3 树中表达表格数据

假设希望用户要选择一个数据项，但要选择的这个项需要取决于之前选中的项，而这些项又会再次需要取决于一些之前的选中项。具体而言，假设想让用户选择某个特定的机场——首先，必须先选择国家，然后选择城市，最后才能够选择机场。通过提供三个组合框应当可以实现这一功能，其中，第一个组成的是国家名，第二个组成的是当前国家中的城市名，而第三个组成的是当前城市的机场。这并不难编程，但是用户必须使用三个独立的窗口部件来给定他们的选择，并且用户也不能轻易看到他们的可选范围中的各个选项。

关联数据项选择的一个解决方案是使用树视图(tree view)。继续以上面这个例子为例，树根将会是国家，每个国家都会有一些作为树枝的城市，而每个城市树枝都会有作为叶子的机场。对于用户来说，这样做就可以更为容易地形成一条通路(他们也就可以只去顺着那些有效的路径)，对我们来说，也更容易获得用户完整的国家/城市/机场的选择结果了。

以使用表格视图为例，使用树视图相比它的另一个好处是，树视图会更紧凑且更易于浏览。例如，如果有 100 个国家，每个国家平均有 4 个城市，每个城市平均有 2 个机场，使用表格就需要  $100 \times 4 \times 2 = 800$  行——但树只需要 100 行(一个国家一行)，其中，每行都可以进行展开，以显示其中的城市和机场。

在这一节中，将会说明是如何在树中表达一个表格数据的，还会说明是如何提取用户选择完整“路径”的。这里用到的示例应用程序叫服务器信息(Server Info)。该应用程序会读取一个 6 列的数据集——国家、州(只对美国有意义)、城市、供应商、服务器和 IP 地址——还可以允许用户给定一个特殊的 6 元组。样本数据集有 163 行，但只会指向不重复的 33 个国家，因此，用户只需要在 33 个顶级项中进行导航切换即可，而不需要在近 5 倍的行之间进行滚动切换。

这个应用程序的主要功能是由 TreeOfTableModel 类提供的，这是 QAbstractItemModel 的一个子类，可以在树中表达任意表格数据。这里会用到这个模型的一个自定义子类，并会用 QTreeView 子类来表达数据。这个应用程序本身可以使用不同的嵌套级别来创建这个树，只需通过在控制台中用命令行参数 1、2、3 或者 4 来运行它即可。图 16.3 给出了使用默认嵌套级别为 3 的树的运行效果(嵌套级别不会包括一系列树枝最末端之后的叶子)。



图 16.3 显示成树的表格数据

这里将先从主窗体开始看起，因为它非常短。然后，将会看到表视图子类以及 TreeOfTableModel 子类。接下来，将会查看 treeoftable 模块，包括 BranchNode 和 LeafNode 类，而在最后，是 TreeOfTableModel 类本身。

```
class MainForm(QMainWindow):
    def __init__(self, filename, nesting, separator, parent=None):
        super(MainForm, self).__init__(parent)
        headers = ["Country/State (US)/City/Provider", "Server", "IP"]
        self.treeWidget = TreeOfTableWidget(filename, nesting,
                                           separator)
        self.treeWidget.model().headers = headers
        self.setCentralWidget(self.treeWidget)
        self.connect(self.treeWidget, SIGNAL("activated"),
                     self.activated)

    self.setWindowTitle("Server Info")
```

TreeOfTableWidget 类似于一个简便视图，因为它里面会包含一个模型。这个模型是 ServerModel，这是 TreeOfTableModel 的一个小子类，只是添加了一个显示图标的功能。

这里的 filename 是指一个含有可用于 TreeOfTableModel 的数据的文件名。实际上，它的每行都必须有一条记录，且每一列(域)都必须通过给定的分隔符进行分隔。

这里的嵌套值是指从根中可以分离出来的树枝的最大数，其值并不包含末端的叶子。在这个例子中，由嵌套参数传递过来嵌套值是 3(除非在命令行中改变了它的值)，这就意味着，将会有 3 个层级的分支(国家、州、城市)和 1 个层级的叶子(供应商)。由于有 6 个域，这就意味着，头 4 个域将会在树窗口部件中的树部分显示出来，而剩余的 2 个域则会显示成单独的列，这些列仅存在于带有叶子的行。最终的树视图将会有 3 个列，一个列中含有的是树，而另外 2 个列会显示其他域。这里通过直接访问模型中自定义的 TreeOfTableWidget，对模型的标题进行了设置。

当用户双击鼠标或者在树窗口部件的行上按回车(Enter)键时，就会调用 activated() 方法。

```
def activated(self, fields):
    self.statusBar().showMessage("*".join(fields), 60000)
```

这里的“路径”(path)，即 6 元组(国家、州、城市、供应商、服务器、IP 地址)，也是用户选择的内容，无论何时，只要选中了适当的行，都会在状态栏中以“\*”分割的形式将内容显示一分钟(60 000 毫秒)。在这里所说的适当的行是指，包含有一个叶子的行，因为只有这些行才是拥有全部 6 个域的行。

TreeOfTableWidget 是 QTreeView 的一个子类，其中含有要显示的模型。它还提供了一些简单的便捷方法，并且创建了一些有用的信号-槽连接。

```
class TreeOfTableWidget(QTreeView):  
    def __init__(self, filename, nesting, separator, parent=None):  
        super(TreeOfTableWidget, self).__init__(parent)  
        self.setSelectionBehavior(QTreeView.SelectItems)  
        self.setUniformRowHeights(True)  
        model = ServerModel(self)  
        self.setModel(model)  
        try:  
            model.load(filename, nesting, separator)  
        except IOError, e:  
            QMessageBox.warning(self, "Server Info - Error",  
                                unicode(e))  
        self.connect(self, SIGNAL("activated(QModelIndex)"),  
                    self.activated)  
        self.connect(self, SIGNAL("expanded(QModelIndex)"),  
                    self.expanded)  
        self.expanded()
```

ServerModel 是 TreeOfTableModel 的一个子类。它的唯一目的是重写 data() 方法，以便它可以提供适当的图标(国旗和州旗)。在从文件中加载完模型数据并且完成了信号-槽连接之后，就可以调用 expanded() 方法来给各列一个合适的宽度了。

```
def expanded(self):  
    for column in range(self.model().columnCount(QModelIndex())):  
        self.resizeColumnToContents(column)
```

用户无论何时展开一个分支——例如，通过点击树的“+”符号或者是通过箭头进行导航并且按下右箭头——就会调用这个方法。它可以确保显示扩展项文本的那些列能够有足够的宽度来显示可读文本。在树模型中，每个项都要么是另一个项的子项(因此会有一个父对象)，要么是一个顶级(根)项，此时它会没有父对象，意指一个无效模型索引值。因此，当调用带有 QModelIndex() (即用一个无效模型索引值) 的 columnCount() 时，就可以得到顶级项的列数。

```
def activated(self, index):  
    self.emit(SIGNAL("activated"), self.model().asRecord(index))
```

如果用户通过双击鼠标或按下回车(Enter)键来激活一个项，就会调用这个方法，而它也会发射自己的 activated() 信号。对当前模型索引值来说，它的参数就是整条路径(记录)，这是一个由各个域值构成的列表。

```
def currentFields(self):  
    return self.model().asRecord(self.currentIndex())
```

这个方法会为 activated() 信号提供相同的信息，但可以在任何时候调用它以得到当前的记录值；同样，这也是一个由各个域值构成的列表。

ServerModel 是 TreeOfTableModel 的一个子类，重新实现了一个方法 data()。它之所以这么做，是为了能够在靠近国家名和美国各州的地方显示相应的旗帜。

```
class ServerModel(treeoftable.TreeOfTableModel):  
    def __init__(self, parent=None):  
        super(ServerModel, self).__init__(parent)  
  
    def data(self, index, role):  
        if role == Qt.DecorationRole:  
            node = self.nodeFromIndex(index)  
            if node is None:  
                return QVariant()  
            if isinstance(node, treeoftable.BranchNode):  
                if index.column() != 0:  
                    return QVariant()  
                filename = node.toString().replace(" ", "_")  
                parent = node.parent.toString()  
                if parent and parent != "USA":  
                    return QVariant()  
                if parent == "USA":  
                    filename = "USA_" + filename  
                filename = os.path.join(os.path.dirname(__file__),  
                                       "flags", filename + ".png")  
                pixmap = QPixmap(filename)  
                if pixmap.isNull():  
                    return QVariant()  
                return QVariant(pixmap)  
        return treeoftable.TreeOfTableModel.data(self, index, role)
```

这里重新实现 `data()` 方法只是为了处理角色为 `Qt.DecorationRole` 的各项请求，并且会把任何其他请求都传送给 `TreeOfTableModel` 基类。在列表、表格和树视图中，用修饰角色来设置或获取数据项的图标。

树模型采用父子对象的形式进行工作。在 `TreeOfTableModel` 基类中曾提供了一个方法，`nodeFromIndex()`，它可以返回特定模型索引值的节点(项)。这里有两种节点，即分支节点和叶子节点。每个节点都可以有任意数量的列，尽管在这个例子中的分支节点只有一个列并且叶子节点至少有一个列。这里将只为分支节点的第一个(并且只有一个)列提供图标，并且也只会为国家和美国的各个州的分支提供图标。

各个国旗图标都保存在国旗子目录中，国旗名用的是下画线而不是空格，并且美国各州的名字均以“USA\_”开头。所有的国旗图标都是.png 图片。这里没有使用.qrc 资源文件，而是直接从文件系统中检索这些图片。`os.path.dirname()` 函数会返回一个完整文件名中的路径部分，而 `os.path.join()` 函数则会把两个或者多个字符串与适当的路径分隔符连接成一个单一的路径字符串。如果所需的图片不存在或不可读，`QPixmap.isNull()` 就会返回 `True`；在这个例子中，将会通过返回一个无效 `QVariant` 来说明没有可用的图标。否则，就把图片的像素值封装在 `QVariant` 中返回。

目前为止所看到的各个类都非常简单。这是因为，真正为树模型提供工作的事情都是由 `TreeOfTableModel` 完成的。这个模型可以读取一个表格数据集并将其行/列数据转换成一棵树。这棵树用一个分支节点作为根，再在根上挂载任意数量的分支节点，每个分支节点都可以再有自己的分支。在每个分支的末尾都是一个或者多个叶子节点。

挂在一个分支上的节点就是该分支的子节点。子节点既可以是分支也可以是叶子，它们都保存在一个列表中。每个子节点在它的父节点列表中的位置就是该子节点的行号。列数指的是一个子节点(分支或者叶子)中项(域)的个数。一个完整的记录(或者“路径”)就是根分

支上所有域的级联体，所有的中间分支以及最后的叶子节点。分支和叶子节点之间的关系如图 16.4 所示。

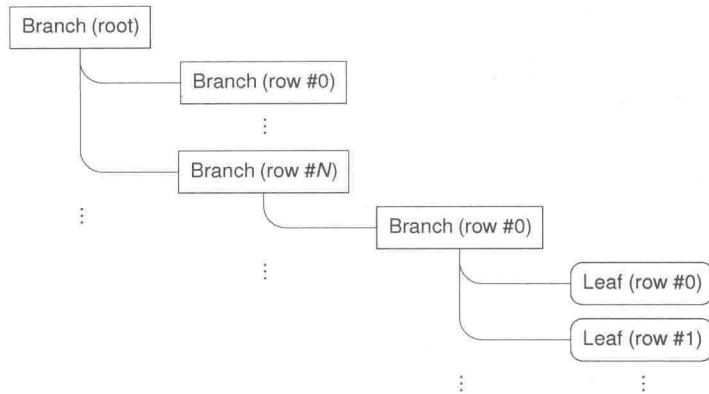


图 16.4 树模型中的分支和叶子

这里选用表格模型树来让每个分支的子节点可以按照字母顺序进行排列。为了能够让这一功能尽可能快速和容易实现，每个分支的子节点列表实际上都会是一个双元素项列表，第一个元素项会是顺序键，而第二个元素项则是子节点。访问这个双元素项列表中的各个项可以使用常量 KEY 和 NODE，而不是用文字数字 0 和 1。

现在来看一看分支节点和叶子节点的实现，然后再来看看表格模型树的实现。

分支和叶子节点有很多共同的方法，因为在某些情况下，它们是可以交替使用的[要感谢 Python 的鸭子类型(duck typing)，详见第 7 章最后一节]。

```

class BranchNode(object):
    def __init__(self, name, parent=None):
        super(BranchNode, self).__init__(parent)
        self.name = name
        self.parent = parent
        self.children = []
  
```

一个分支节点的名字就是显示在第一(且仅是)列的文本。在服务器信息应用程序示例中，分支的名字可能就是国家、州或城市的名字，这取决于该分支在树中的层次。

```

def orderKey(self):
    return self.name.lower()

def toString(self):
    return self.name

def __len__(self):
    return len(self.children)
  
```

顺序键是一个字符串，由节点的父节点来确定该分支在节点之父节点列表中的正确位置。`toString()`方法会返回该分支的一个字符串型的域。这些方法可用来为叶子节点提供兼容性，以便基于鸭子类型来使其更易用于节点的各种类型。`__len__()`方法可以返回一个分支的子节点个数。

```

def childAtRow(self, row):
    assert 0 <= row < len(self.children)
    return self.children[row][NODE]
  
```

这个方法可以返回一个用于给定行的节点。这里用到了一个断言语句，并且也会用在表格模型树的其他许多地方的代码中。要想正确实现这些代码还是需要用些技巧的，不过，通过断言至少可以清楚在代码的某些特定地方，到底期望什么才是正确的。

```
def rowOfChild(self, child):
    for i, item in enumerate(self.children):
        if item[NODE] == child:
            return i
    return -1
```

在这里，会返回某个特定子节点的行索引值，如果该子节点不是这个节点的子节点之一，就返回 -1。

```
def childWithKey(self, key):
    if not self.children:
        return None
    i = bisect.bisect_left(self.children, (key, None))
    if i < 0 or i >= len(self.children):
        return None
    if self.children[i][KEY] == key:
        return self.children[i][NODE]
    return None
```

有时希望找到给定顺序键的第一个子节点。一种方法或许就是在 `rowOfChild()` 方法中所做的那样，遍历全部子节点列表来找到要找的那个子节点。这里会采用一种更为有效的方法。先用给定的键来找到该位置，这里应该是由一个节点占用着的，如果这个位置在有效范围内并且也正是那个给定的键，那就返回该子节点。

```
def insertChild(self, child):
    child.parent = self
    bisect.insort(self.children, (child.orderKey(), child))
```

列表中插入一个新的子节点，并让这个分支成为该子节点的父节点。通过将 `bisect.insort()` 与该子节点的顺序键联用，可以确保该子节点能够尽可能快速有效地放到正确的位置。`insort()` 函数与 `insort_right()` 一样。

```
def hasLeaves(self):
    if not self.children:
        return False
    return isinstance(self.children[0], LeafNode)
```

在表格模型树中，一个有子节点的分支要么有其他分支，要么有叶子节点，但不可能两者兼有。基于这一原因，如果一个分支根本就没有孩子，显然它也就没有叶子；同样，如果一个分支中有子节点并且第一个子节点就是个叶子节点，那么所有的子节点都应该是叶子节点。

现在就看完了整个分支节点类。接下来，将会看看更短小的叶子节点类。

```
class LeafNode(object):
    def __init__(self, fields, parent=None):
        super(LeafNode, self).__init__(parent)
        self.parent = parent
        self.fields = fields
```

叶子节点中的域(field)就是该节点的各个列。

```
def orderKey(self):
    return u"\t".join(self.fields).lower()

def toString(self, separator="\t"):
```

```

        return separator.join(self.fields)

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

```

一个叶子节点的顺序键就是它的各个域通过 Tab 制表符连接起来的级联体。同样，它的 `toString()` 方法会返回一个它的各个域的级联体。`__len__()` 方法会返回域的个数；对于分支来说，这个方法会返回子节点的个数。

```

    def field(self, column):
        assert 0 <= column <= len(self.fields)
        return self.fields[column]

```

这个方法让单个域值的提取变得很轻松，而借助断言语句，可以确保域的列在有效范围内。

```

    def asRecord(self):
        record = []
        branch = self.parent
        while branch is not None:
            record.insert(0, branch.toString())
            branch = branch.parent
        assert record and not record[0]
        record = record[1:]
        return record + self.fields

```

所谓的由表格模型树使用的记录就是从根节点到叶子节点的父节点的所有分支的级联体，外加叶子节点自身——换句话说，就是用户的完整选择“路径”。从服务器信息应用程序这个例子来说，这就是国家、州、城市、供应商、服务器、IP 地址，其中的国家、州和城市为分支，而每个叶子都会包含三个域：供应商、服务器和 IP 地址。

要建立一个记录（一个域的列表），可以先从叶子节点的父分支开始，并沿着树的各个分支走下去。把每个分支的字符串都添加到这个记录列表中。根分支没有字符串，所以可将其从列表中移除掉。返回的列表就是所有分支字符串加上叶子的字符串所构成的级联体。

现在就完成了对节点的回顾。表格模型树是 `QAbstractItemModel` 的一个子类，该子类重新实现了许多想要的方法，如 `data()`、`headerData()`、`rowCount()` 和 `columnCount()`。另外，它还提供了树模型通常需要重新实现的 `index()`、`parent()` 和 `nodeFromIndex()` 方法。它还有一些额外的方法，也就是，`load()` 和 `addRecord()`；这些方法都会用来加载表格型数据并将其转换成一个带有分支和叶子的树。接下来将会查看初始化程序，然后是用来加载数据的方法，再后是一些标准的模型/视图方法。

```

class TreeOfTableModel(QAbstractItemModel):
    def __init__(self, parent=None):
        super(TreeOfTableModel, self).__init__(parent)
        self.columns = 0
        self.root = BranchNode("")
        self.headers = []

```

列数取决于加载进来的数据的列数，也取决于嵌套的层次。总会有一个根分支节点，它不包含任何文本，纯粹用做所有其他分支的父节点。这里的 `headers` 是用做列标题的文本。

```

    def load(self, filename, nesting, separator):
        assert nesting > 0
        self.nesting = nesting
        self.root = BranchNode("")
        exception = None

```

```

fh = None
try:
    for line in codecs.open(unicode(filename), "rU", "utf8"):
        if not line:
            continue
        self.addRecord(line.split(separator), False)
except IOError, e:
    exception = e
finally:
    if fh is not None:
        fh.close()
    self.reset()
    for i in range(self.columns):
        self.headers.append("Column #%d" % i)
    if exception is not None:
        raise exception

```

要加载的文件必须是每行一条记录的文本文件，且每个域都要通过给定的分隔符进行分割。文件编码必须是 UTF-8 的 Unicode(或者是 ASCII, 因为它是 UTF-8 的一个子集)。空白行都忽略掉；其他的行都会视为一个记录并被添加到树中。

加载一旦完成(成功与否)，都将会调用 `reset()` 来通知所有视图，模型已经发生了显著变化，需要创建一些初始列标题。如果加载失败，接着就会抛出异常并由调用者进行处理。列变量在初始化程序中会设置为 0，同时，在 `addRecord()` 中会将其设置成一个有意义的值。

```

def addRecord(self, fields, callReset=True):
    assert len(fields) > self.nesting
    root = self.root
    branch = None
    for i in range(self.nesting):
        key = fields[i].lower()
        branch = root.childWithKey(key)
        if branch is not None:
            root = branch
        else:
            branch = BranchNode(fields[i])
            root.insertChild(branch)
            root = branch
    assert branch is not None
    items = fields[self.nesting:]
    self.columns = max(self.columns, len(items))
    branch.insertChild(LeafNode(items, branch))
    if callReset:
        self.reset()

```

要添加一个记录，就必须要有比嵌套层次更多的域。这里用到的逻辑类似于在第 14 章中当组装 `QTreeWidget` 的内部模型时所看到的那样。对于每一个即将成为分支的域，都会用同样的键来查找已有的分支。如果能够找到一个，就使其成为当前的根分支；否则，将创建一个新的分支，并将其作为子节点插入到当前的根分支中，再让该新分支作为当前的根分支。随着循环的进展，就可以沿着树向下搜寻，在此过程中可能会创建一些需要的分支，直到达到最底层的分支。

循环一旦走遍所需的全部分支，创建一些先前不存在的分支，就可以创建一个非嵌套域的列表，并且可以将它们作为子叶子节点添加到当前的(最低层的)分支中。

具体来说，这里以服务器信息应用程序为例，来看看到底发生了什么。当读到第一个记录

时，就可以得到一个新的国家、新的城市、新的供应商等，所以也就不会有合适的分支存在。首先，将会创建一个新的国家分支，然后会创建一个州分支，再然后会创建一个城市分支，最后创建的是一个包含了供应商、服务器、IP 地址等域在内的叶子节点。如果下一个要读取的记录是同一个国家，但却是一个新的州，就可以找到已经存在的国家节点并将其作为新州的父节点。同样，如果一个记录中的国家和州与该分支中已经存在的国家和州相同，那么就会使用这些节点作为新读取节点的父节点。但无论何时，只要创建一个新的分支，在循环体中的代码就会创建它。

当新的记录添加到某一基础上时，就可以调用 `reset()` 来通知所有视图，已经发生了较为显著的改变；不过，当从文件中加载时，可以传递 `False` 并在调用代码中调用 `reset()`，那么就可以一次读取所有的记录。

```
def asRecord(self, index):
    leaf = self.nodeFromIndex(index)
    if leaf is not None and isinstance(leaf, LeafNode):
        return leaf.asRecord()
    return []
```

这个方法可以提供一个用户选择“路径”的列表。它只对叶子节点有意义，因为只有叶子节点才可以表达一个完整的路径。对于非叶子节点会返回 `None`，这应该是一个不错的设计选项。值得注意的是，`nodeFromIndex()` 方法可以用来获取给定模型索引值的节点：不久就会讨论它是如何工作的。

```
def rowCount(self, parent):
    node = self.nodeFromIndex(parent)
    if node is None or isinstance(node, LeafNode):
        return 0
    return len(node)
```

对于树模型来说，行数是某个特定节点所拥有的子节点数。这里的实现方法只允许分支节点拥有子节点，所以当对叶子节点调用时总会返回 0。`len()` 函数会调用 `BranchNode.__len__()`，它会返回分支的子节点数。

```
def columnCount(self, parent):
    return self.columns
```

列数是非嵌套域的最大数。将其设置成 1 看起来可能会显得太少了，但这却是正确的，因为第一个非嵌套域会显示在第一（树）列。

```
def data(self, index, role):
    if role == Qt.TextAlignmentRole:
        return QVariant(int(Qt.AlignTop|Qt.AlignLeft))
    if role != Qt.DisplayRole:
        return QVariant()
    node = self.nodeFromIndex(index)
    assert node is not None
    if isinstance(node, BranchNode):
        return QVariant(node.toString()) \
            if index.column() == 0 else QVariant(QString(""))
    return QVariant(node.field(index.column()))
```

如果一个分支节点要求显示数据，就可以为列 0 返回节点的文本，而对其他列则返回一个空字符串。对于一个叶子节点来说，可以返回与要求的列相对应的域。

在 Qt 4.2 之前，默认文本的对齐功能做得很好并且不需要给定，但从 Qt 4.2 开始，就必须由我们自己明确返回一个合理的文本对齐方式。

```
def headerData(self, section, orientation, role):
    if orientation == Qt.Horizontal and \
        role == Qt.DisplayRole:
        assert 0 <= section <= len(self.headers)
        return QVariant(self.headers[section])
    return QVariant()
```

树视图只有水平(列)标题。它们没有行标题(比如, 行号), 这是因为这些标题并没有多大的实际意义, 因为每个分支都有自己从 0 开始的子节点(行)列表。

```
def index(self, row, column, parent):
    assert self.root
    branch = self.nodeFromIndex(parent)
    assert branch is not None
    return self.createIndex(row, column,
                           branch.childAtRow(row))
```

`index()`方法必须为给定行和列的数据项返回器模型索引值, 该数据项也是给定父节点的子节点。在一个具有多个分支和叶子节点的树模型中, 这就意味着, 必须返回父节点元素的第 `row` 行子节点的模型索引值。

先通过找到给定父模型索引值的分支节点开始, 可以返回给定行和列的模型索引值, 并可以返回该(分支)节点的第 `row` 行子节点的父节点。

```
def parent(self, child):
    node = self.nodeFromIndex(child)
    if node is None:
        return QModelIndex()
    parent = node.parent
    if parent is None:
        return QModelIndex()
    grandparent = parent.parent
    if grandparent is None:
        return QModelIndex()
    row = grandparent.rowOfChild(parent)
    assert row != -1
    return self.createIndex(row, 0, parent)
```

`parent()`方法必须返回在具有多个分支和叶子节点的树模型中给定子节点的父节点的模型索引值, 这就是该子节点祖父节点的行子节点。

这里先从找到子节点父节点的父节点(即子节点的祖父)开始。然后, 返回一个模型索引值, 它具有的行的父节点会占用祖父列表的子节点, 即列 0(因为所有父节点都是分支而且这些分支都只有一个第 0 列), 还会返回一个父节点, 它是子节点的父节点。

这里给出的 `index()` 和 `parent()` 方法的重新实现十分巧妙。然而, 它们采用的都是一个分支和一个叶子的方法, 对于树模型来说都是标准做法, 所以在大多数情况下它们的代码都可以“基本不动”地简单复制即可。

```
def nodeFromIndex(self, index):
    return index.internalPointer() \
        if index.isValid() else self.root
```

当调用 `QAbstractItemModel.createIndex()` 时, 第三个参数是一个节点引用。这个引用可以从一个模型索引值中获得并且会被 `internalPointer()` 方法返回。对于任何给定的模型索引值, 都会返回一个分支或这一个叶子节点, 又或者返回该分支的根节点。

对于树模型的理解可能要比对表格模型(或者列表模型, 只是些具有单列的表格)的理解

更具挑战性。然而，在多数情况下，通过构建或调整这里所给出的代码，能够一定程度地减少这些困难。

## 小结

PyQt 的内置视图窗口部件和图形视图窗口部件，在两者之间还为数据集可视化提供了相当大的空白地带。但是，当我们的需求无法由这些类所提供的功能满足时，就可以创建一些自定义视图，以便可以用我们最喜好的方式来精确呈现数据。

由于自定义视图可能用来显示一个大数据集的一部分，通常最好是能够对其绘制事件处理程序进行优化，以便获得并只显示那些实际可见的数据项。如果需要滚动条，可以要求视图类用户使用 `QScrollArea`，或者也可以创建一个带有几个 `QScrollBar` 的复合窗口部件，再或者可以创建一个继承自 `QAbstractScrollArea` 的窗口部件。所有这些方法中只有第一种方法需要在用户的代码中添加几行代码，这可以使视图的实现更为容易些。

使用带有与数据类型相关的列委托的泛型委托可以很容易地为视图创建“自定义”委托。列委托的创建比较容易，并且可以大幅度减少代码的重复度，因为对于打算用到的每种数据类型只需要用一个列委托。泛型委托法对数据库表这样的每列数据都有一个数据类型的数据集非常理想。

树模型的创建可能会稍稍有些困难，因为不得不从父节点和子节点的角度思考问题，其中的子节点也可能是父节点，不断递归，直至到任意的深度。这绝不能简单地认为就是树和列模型中所需的行和列。尽管在这一章中给出的表格模型树是一个特殊的例子，但其中一些提供树功能的方法，如 `index()`、`parent()`、`nodeFromIndex()` 等，还是应当可以“直接”或者稍稍做些修改使用的，而其他一些方法，如 `addRecord()`，也说明具有较好的适用性。

## 练习题

这个练习题将会用到这一章和前几章中用到的模型/视图的许多特性。

创建一个可显示两个窗口部件的应用程序：一个是 `QListView`，另一个是自定义的 `BarGraphView`。数据应当保存在自定义的 `BarGraphModel` 中。用户应当能够通过 `QListView` 编辑数据，也可以使用自定义的 `BarGraphDelegate` 来控制列表视图中数据项的表达和编辑操作。该应用程序如图 16.5 所示。

这里的模型应该是 `QAbstractListModel` 的一个子类，它应当保存一个数据值（许多整数）列表和一个颜色字典（键由“行”确定；也就是说，键 6 的颜色对应于第 7 个数据值，等等）。该模型应当重新实现 `rowCount()`、`insertRows()`——应当在适当的地方调用 `beginInsertRows()` 和 `endInsertRows()`，`flags()` 可使模型变成可编辑，`setData()` 可以允许设置值（`Qt.DisplayRole`）和颜色值（`Qt.UserRole`），它们也应该可以发射信号来告知数据已经发生了改变——而 `data()`，则应当可以返回值、颜色和一个用于 `Qt.DecorationRole` 的  $20 \times 20$  的填色位图。如果没有给某个特定行设置颜色，就使用默认的红色。

这里的委托非常简单，它与这一章之前所提到的 `IntegerColumnDelegate` 非常类似。关键的不同在于，必须重新实现 `paint()` 方法，不过只是将对齐方式设置成 `Qt.AlignRight`；基类也可以很好地完成这一绘制工作。

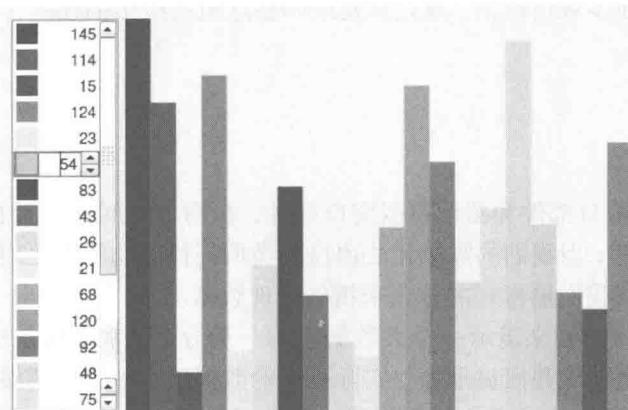


图 16.5 柱状图应用程序的各个窗口部件

这里的自定义视图需要重新实现 `setModel()`，其中应当连接到基类的 `update()` 方法，以便可以在模型的数据发生改变时重绘，还有 `minimumSizeHint()`、`sizeHint()`——可以简单调用 `minimumSizeHint()` 和 `paintEvent()` 重新实现。绘制事件可以大约用 12 行的代码完成——确保使用了 `QPainter.setWindow()`，以便图总是可以填满所有可用的空间。即使没有设置任何模型，所有方法也应该可以正常工作——例如，在没有设置模型时，绘制事件应该什么也不去绘制。

这里给出的代码是 `MainForm` 的，以便让你可以初步感受到类的用法：

```
class MainForm(QDialog):
    def __init__(self, parent=None):
        super(MainForm, self).__init__(parent)
        self.model = BarGraphModel()
        self.barGraphView = BarGraphView()
        self.barGraphView.setModel(self.model)
        self.listView = QListWidget()
        self.listView.setModel(self.model)
        self.listView.setItemDelegate(BarGraphDelegate(0, 1000, self))
        self.listView.setMaximumWidth(100)
        self.listView.setEditTriggers(QListView.DoubleClicked |
                                     QListView.EditKeyPressed)
        layout = QHBoxLayout()
        layout.addWidget(self.listView)
        layout.addWidget(self.barGraphView, 1)
        self.setLayout(layout)
        self.setWindowTitle("Bar Grapher")
```

在这个模型的答案中，添加了一些额外代码，以创建 20 个随机项来构建一个初始条形图。整件事情应当可以在 200 行内完成。

本练习题的参考答案在文件 `chap16/bargrapher.pyw` 中。

# 第 17 章 在线帮助和国际化

- 在线帮助
- 国际化

只通过阅读应用程序的菜单项和按钮上的文本，用户可以使用非常简单的应用程序了。其他一些应用程序就需要有更多的信息才能使用，在这种情况下，工具栏提示和状态栏提示可能就是最简单易行的解决方案了。不过，一些应用程序会相当复杂或者难以理解，用户或许就需要更多的帮助才能理解有哪些功能是可用的，以及如何才能使用这个应用程序。

给出足够信息的一种方法就是提供一本打印版手册；另一种方法是提供一个帮助系统。还有几种方法可以用来创建一个适当在线帮助系统；这些方法都会在这一章提及，并且会给出其中的一种方法。这里将会返回到第 6 章中介绍的图片变换(Image Changer)应用程序，在 17.1 节将会给出这个应用程序 `MainWindow.helpHelp()` 方法的实现过程，并介绍提供在线帮助系统的方法。

在整本书中给出的所有应用程序都以英语的形式为用户提供了菜单、按钮、标签、提示信息等内容。对于这个世界上能够读懂英语的少数人来说，这样做很不错，但是，对于世界上应用最为广泛、说的人最多的语言，汉语，或者是对于那些说的比较多的其他主要语言，比如西班牙语、阿拉伯语、印度语、葡萄牙语、孟加拉语、俄语或者日语等，都比较糟糕。

要让一个应用程序尽可能广泛地得到应用，就必须要让非英语人士能够使用。PyQt 提供了一种工具链，可以用来识别那些用户可见的字符串并让翻译人员在易于使用的 Qt Linguist GUI 应用程序中访问这些字符串进而提供出合适的翻译文字。在 17.2 节，将会对翻译工具做些讨论并给出该如何使用好它们。还会使用翻译工具给出图片转换应用程序一个适当的翻译新版本。

## 17.1 在线帮助

提供在线帮助系统的方法通常有三种。一种方法是，以 HTML 文件的形式提供帮助，把其中的相应页面通过 Web 浏览器予以打开。另一种方法是，使用由 Qt 提供的 Qt Assistant 应用程序。第三种方法是，提供一个帮助文档(help form)，其中会再次用到 HTML，不过是把图片和 HTML 文件作为帮助表单的资源。

第一种方法可通过单独进程的形式打开 Web 浏览器实现，其中，要么会用到 Python 的 `subprocess` 模块，要么会用到 PyQt 的 `QProcess` 类。Qt 4.2 中引入了一个新类，`QDesktopServices`，利用其 `openUrl()` 静态便捷方法，可以真正轻松实现浏览器的平台独立打开。

第二种方法要稍微棘手些，因为它需要创建一个特殊格式的 XML 文件，并使用我们的应用程序来启动 Qt Assistant。使用 Qt Assistant 的好处是，它可以提供自动检索功能。

第三种方法，会使用一种自定义帮助文档并会把 HTML 文件和图片做成资源，这也是我们将会采用的方法。回顾第 6 章可以发现，在资源文件中应当是可以包含包括 HTML 文件在内

的任意文件的，当时就把一些演示(demo)帮助文件放到了 resources.qrc 文件中。下面给出的是图片转换应用程序 MainWindow.helpHelp() 方法中的代码：

```
def helpHelp(self):
    form = helpform.HelpForm("index.html", self)
    form.show()
```

帮助文档的使用还是比较简单的：只需给它一些 HTML 文件和一个 self(帮助文档会在 self 上居中)。值得注意的是，这里使用的是 show() 而不是 exec\_(); 这通常意味着，这个帮助文档的显示将会对“关闭即删除属性”( delete on close attribute)进行设置<sup>①</sup>。

在图 17.1 所示的截图中，好像会造成一种误导性印象，即难以满足键盘用户的需要，然而实际上，用于显示 HTML 文件的类，QTextBrowser，是可以提供键盘支持的。例如，用户可以按 Tab 键来在不同的超链接之间移动光标焦点，也可以按 Enter 键进入一个超链接。通过按下 Alt + 左箭头 (Alt + Left Arrow) 可以回退，也可以通过按下 Home 键回到第一页。同时，由于这个帮助文档是 QDialog 的一个子类，通过按下 Esc 键也可以关闭该窗口。

现在，我们已经比较熟悉 PyQt 对话框的创建，所以这里仅给出与创建在线系统相关的那些代码——特别是从 HelpForm 的初始化程序以及从它的方法中的一个而提取的那些代码(全部代码在文件 chap17/helpform.py 中)。

```
class HelpForm(QDialog):
    def __init__(self, page, parent=None):
        super(HelpForm, self).__init__(parent)
        self.setAttribute(Qt.WA_DeleteOnClose)
        self.setAttribute(Qt.WA_GroupLeader)
```

属性 Qt.WA\_GroupLeader 可以确保，如果帮助文档是从模态对话框调用的，用户将能够拥有同时和模态对话框及帮助文档进行交互的能力，不过，某些功能可能会失效。如果帮助文档是从非模态对话框或者主窗口调用的，该属性就会失效，用户可以像往常一样与两者进行交互。

```
    self.textBrowser.setSearchPaths([":/"])
    self.textBrowser.setSource(QUrl(page))
```

QTextBrowser 类是 QTextEdit 的一个子类，可以用来显示 HTML 标记的一个大的子集，包括图片、列表和表格。这里已经将其搜索路径设置到资源文件的根目录，并且把它的初始页设置成传进来的页面。因为已经设置了搜索路径，所以可传送不带路径的页面(比如，只是一个 index.html 或者一个 filenew.html)。QTextBrowser 可以理解资源所在路径，因而也就可以找到 <img> 标记中的图片资源，比如 <img src = ":/filenew.png">。



图 17.1 图片转换应用程序的帮助文档

<sup>①</sup> 只要文件一关闭掉，系统就会删除这个文件，从而可以避免在某些时候忘记删除而产生临时文件的问题——译者注。

```
self.connect(backAction, SIGNAL("triggered()"),
            self.textBrowser, SLOT("backward()"))
self.connect(homeAction, SIGNAL("triggered()"),
            self.textBrowser, SLOT("home()"))
self.connect(self.textBrowser, SIGNAL("sourceChanged(QUrl)"),
            self.updatePageTitle)
```

从一页导航到另一页可以由 `QTextBrowser` 自动处理。尽管如此，还是提供了两个工具栏按钮，`Back` 按钮和 `Home` 按钮，并将两者连接到 `QTextBrowser` 适当的槽，以便可以获得我们想要的行为。如果 HTML 文档发生了改变——例如，由于用户点击了一个超链接——就可以调用 `updatePageTitle()` 自定义槽。

```
def updatePageTitle(self):
    self.pageLabel.setText(self.textBrowser.documentTitle())
```

这个槽仅会在工具栏中的 `QLabel` 上放置 HTML 页面的 `<title>` 文本，这个 `QLabel` 在工具栏按钮的右侧。

一旦有了 `HelpForm` 类，就可以完全用 HTML 来实现在线帮助系统了，要么把这些文件作为资源的形式，要么将其安装到 filesystem 中，然后就可以使用如下代码找到它们：

```
helppath = os.path.join(os.path.dirname(__file__), "help")
```

这里会假定这些帮助文件都是在一个 `help` 目录中的，而且这个目录也是 `.pyw` 文件所在的目录。

编写提供在线帮助系统的代码非常简单；不过，设计一个易于导航且易于理解的系统，则可能会相当具有挑战性。

## 17.2 国际化

要让应用程序能够适用于某些用户，他们的说话语言与应用程序起初采用的语言不同，就需要考虑几个问题。最大也是最显而易见的问题是，所有的用户可见的字符串必须要能够翻译成目标语言——这不仅包括那些用在菜单选项和对话框按钮中的字符串，还包括工具栏提示、状态栏提示以及其他在线帮助字符串。此外，还要执行一些本地化操作，比如要确保数字能够使用合适的小数点和千位符，时间和日期的格式是正确的，纸张尺寸大小和测量系统是对的。例如，大多数的美国人和英国人说的都是英语，但两者不同的文化差异造成他们具有不同的日期格式习惯、不同的汇率、不同的标准纸张尺寸大小以及不同的测量系统。

幸好有 Unicode 可用，使得任何语言所用到的任何字符都可以显示出来。在本书的一开始就看到了，只要使用 `unicode` 转义字符和目标字符的十六进制代码点，又或者使用 `unichr()` 函数，任何 `unicode` 字符都可以包含到 `unicode` 字符串或者 `QString` 字符串中。在读写含有 Unicode 的文本文件时，可以使用 Python 的 `codecs.open()` 函数，或者像在上一章中那样使用 PyQt 的 `QTextStream`。

在实施本地化的某些东西时，可以使用 `QString`、`QDate` 和 `QDateTime`。例如，假设 `n` 是一个数字，`QString("% L1").arg(n)` 将会生成一个可用于本地的带有千位和小数分隔符的 `QString`。无论是 `QDate` 还是 `QDateTime`，都有一些 `toString()` 方法，要么可以接收某种自定义格式，要么可以接收一种预定义格式，比如 `Qt.SystemLocaleDate`（老代码中是 `Qt.LocalDate`），要么可以接收“通用的”`Qt.ISODate`。此外，`QLocale` 还提供了返回本地化 `QString` 的许多方法，也提供了一些从本地化的 `QString` 中提取数字的方法。还会有一些方法可以返回与本地相关的字符，比如某个字符需要使用的负号、百分号，等等。

有关应用程序国际化的大部分工作就是翻译，因此，在本节剩下的内容中，就主要关注这一主题。

为了能够帮助翻译应用程序，PyQt 提供了三个工具的工具链：pylupdate4、lrelease 和 Qt Linguist。要让这些工具发挥效用，必须对每个用户可见的字符串进行特殊化标记。使用 `QObject.tr()` 方法可以轻松实现这一点，该方法会被 `QWidget` 的所有子类所继承，包括全部的对话框和主窗口。例如，无须写成 `QString("&Save")`，可以直接写成 `self.tr("&Save")`。传送给 `tr()` 的文本应当是 ASCII；如果需要使用一些超出 ASCII 字符范围，可以使用 `trUtf8()` 来代替之。

对于每一个待翻译的标记字符串，翻译工具都会提供一对字符串：一个是“上下文”字符串（类名），一个是标记的字符串自身。上下文字符串的目的是，帮助翻译人员确认要翻译的字符串是在哪个窗口中显示的，因为在某些语言中，或许在不同的窗口中会需要不同的译文。

对于那些需要翻译但又不在类里面的字符串，则必须使用 `QApplication.translate()` 方法，并由我们自己提供上下文字符串。例如，在 `main()` 函数中或许需要翻译应用程序的名字，可以像 `QApplication.translate("main", "Gradgrind")` 这样。在这里，上下文字符串就是“`main`”，要翻译的字符串就是“`Gradgrind`”。

遗憾的是，用在 `self.tr()` 中的上下文字符串可以与 C++/Qt 的 `tr()` 方法中使用的上下文字符串不同，因为 PyQt 会动态确定上下文，而 C++ 在编译时却无法支持它<sup>①</sup>。如果翻译后的文件是在 C++/Qt 和 PyQt 应用程序共享的，这就可能会带来一些麻烦。如果这些帮助文档（help form）是子类化而来的，也可能会是一个问题。如果这确实造成了问题，那么解决方案就是只需用双参数的 `QApplication.translate()` 调用替换每一个单参数的 `self.tr()` 调用，明确、正确地给出作为第一个参数的上下文字符串，以及作为第二个参数的待翻译字符串。

一旦应用程序的全部用户可见字符串得到了适当标记，就必须小心改变应用程序的启动方式，以便它可以在其运行的地方读取出那些翻译后的字符串而实现本地化。

这里给出的是如何创建一个国际化应用程序。

1. 创建应用程序时，对所有的用户可见字符串使用 `QObject.tr()` 或者 `QApplication.translate()`。
2. 修改应用程序，如果本地化配置文件.qm（Qt 消息）可用，在应用程序启动时读取它们。
3. 创建一个.pro 文件，其中会列示应用程序将要使用到的.ui（Qt 设计师）文件、.py 和.pyw 源文件以及.ts（翻译来源，translation source）文件。
4. 运行 pylupdate4，创建.ts 文件。
5. 请翻译人员使用 Qt Linguist 翻译.ts 文件中的字符串。
6. 运行 lrelease，把更新后的.ts 文件（其中含有译文）转换成.qm 文件。

这里还给出了如何来维护应用程序。

1. 更新应用程序，确保所有的用户可见字符串都使用了 `QObject.tr()` 或者 `QApplication.translate()`。
2. 如果有必要，更新.pro 文件——例如，添加了任何新的.ui 文件，或者之前添加过.py 文件。

---

<sup>①</sup> 参见 PyQt 文档中的 `pyqt4ref.html`，它在“Differences Between PyQt and Qt”一节中。

3. 运行 pylupdate4，更新带有新字符串的.ts 文件。
4. 请翻译人员使用翻译.ts 文件中的新字符串。
5. 运行 lrelease，把.ts 文件转换成.qm 文件。

我们将会使用上述的全部步骤，在应用程序中使用 `QObject.tr()`，并在第 17 章的图片转换应用程序中提取出可供翻译的文件版本。

```
fileNewAction = self.createAction(self.tr("&New..."),
    self.fileNew, QKeySequence.New, "filenew",
    self.tr("Create an image file"))
```

标记的第一个待翻译字符串就是菜单项字符串，新建(New...)，第二个待翻译字符串是用于工具栏和状态栏提示的("filenew"字符串是不带.png 的图标文件的名字)。

```
self.fileMenu = self.menuBar().addMenu(self.tr("&File"))
```

菜单项字符串与动作字符串一样，也需要翻译。

```
self.statusBar().showMessage(self.tr("Ready"), 5000)
```

这里会给用户提供一个初始状态消息，同样也必须得使用 `tr()`。

通常不会对 `QSettings` 键中的字符串进行翻译，特别是因为，这些字符串通常对用户来说，都是不可见的。

```
reply = QMessageBox.question(self,
    self.tr("Image Changer - Unsaved Changes"),
    self.tr("Save unsaved changes?"),
    QMessageBox.Yes | QMessageBox.No |
    QMessageBox.Cancel)
```

对于这个消息框，这里已经将窗口标题和消息文本标记为待翻译内容。在这个例子中，不必担心按钮的翻译，因为仅仅使用了标准按钮，Qt 会自动完成它们的翻译<sup>①</sup>。如果对它们使用了自己的文本，就务必要使用 `tr()`，就像对待其他的可见字符串一样。

```
self.tr("Saved %1 in file %2").arg(self.dataname).arg(self.filename)
```

提供上述字符串的一种方式可以这样写：

```
self.tr("Saved %s in file %s" % (self.dataname, self.filename)) # BAD
```

并不推荐使用这种方式。要经常使用 `QString`，也可以经常使用 `QString.arg()`；这样会给翻译人员带来不少方便(`tr()`方法会返回一个 `QString`，因此可以在它的返回值上调用任何 `QString` 方法，比如 `arg()`)。例如，在某些语言中，译文可能是“Saved in file %2 the data %1”的形式。使用带有多个 `arg()` 的 `QString` 不是什么问题，因为翻译人员可以修改这些%*n* 在字符串中的次序，而 `arg()` 方法则会响应这些改变。不过，试图交换 Python 字符串中两个%*s* 的位置，将不会发生任何改变。

对于手工编码的.pyw 和.py 文件中的每一个用户可见字符串，都必须使用 `tr()`。不过，对于由.ui 文件通过 `pyuic4` 而生成的.py 文件，则什么也不需要做，因为 `pyuic4` 会对每个字符串自动使用 `QApplication.translate()`。这一工作机制甚至会应用于未翻译的应用程序上，因为，如果没有合适的翻译文件，就会用原来的语言予以顶替——例如，英语。

PyQt 应用程序总是会使用 PyQt 的内置对话框；例如，文件打开对话框，或者文件打印对

<sup>①</sup> Trolltech 为一些语言提供了译文，比如法语和德语，还对许多非官方支持的不同语言提供了译文。这些翻译文件在 Qt 的（不是 PyQt 的）translations 目录中；可以在你的文件系统中搜索 `qt_fr.qm` 文件，比如，要找到法语翻译文件的话。

话框。这些对话框也必须进行翻译，尽管 Trolltech 已经早就在其.qm 文件中提供了数种译文<sup>①</sup>。

通篇应用 `tr()`，确定适当的 Qt 译文，是修改应用程序的启动代码来加载这些翻译文件的时候了，如果这些翻译文件存在的话。

```
app = QApplication(sys.argv)
locale = QLocale.system().name()
qtTranslator = QTranslator()
if qtTranslator.load("qt_" + locale,(":/"):
    app.installTranslator(qtTranslator)
appTranslator = QTranslator()
if appTranslator.load("imagechanger_" + locale,(":/"):
    app.installTranslator(appTranslator)
app.setOrganizationName("Qtrac Ltd.")
app.setOrganizationDomain("qtrac.eu")
app.setApplicationName(app.translate("main", "Image Changer"))
app.setWindowIcon(QIcon(":/icon.png"))
form = MainWindow()
form.show()
app.exec_()
```

调用 `QLocale.system().name()` 将会返回一个字符串，比如“en\_US”（英语，美国），或者“fr\_CA”（法语，加拿大），等等。`QTranslator.load()` 方法会带一个文件句柄和一个路径。在这个例子中，所给定的路径是`:`，其实就是应用程序的资源文件的路径。如果本地系统是“fr\_CA”，该文件的句柄就将是 `qt_fr_CA` 和 `imagechanger_fr_CA`。假如说就是这些内容，那么 PyQt 将会搜索 `qt_fr_CA.qm`，而如果失败，就接着搜索 `qt_fr.qm`，与之相似，也会搜索 `imagechanger_fr_CA.qm`，如果失败，就接着搜索 `imagechanger_fr.qm`。如果本地系统是“en\_US”，就不会找到任何的.qm 文件，因而什么也就不会安装——这样是比较妥当的，因为应用程序将会返回并使用原始字符串，在这个例子中，就是英语。

值得注意的是，将不得不使用 `QApplication.translate()`（可以写成 `app.translate()`），因为这个代码不在 `QObject` 子类的方法中。不带类名，这里选择使用文本“main”作为上下文字符串；一些编程开发人员或许更喜欢使用“global”。完全可以自由使用任意名字——因为上下文字符串的目的仅仅是为了帮助翻译人员确认位置而已。

一个 `QTranslator` 对象只能加载一个译文，不过可以向 `QApplication` 对象添加任意数量的译文。如果没有冲突，也就是说，如果同一字符串却有不同的译文，那么就会使用最近刚加载的译文。

尽管这里选择使用把翻译文件包含到资源文件中，其实并没有义务必须这么做；本来就是可以非常简单地从文件系统中访问到它们的。

这里是已经使用过的 `resource.qrc` 文件中提取的一段代码：

```
<qresource>
<file>qt_fr.qm</file>
<file>imagechanger_fr.qm</file>
</qresource>
<qresource>
<file alias="editmenu.html">help/editmenu.html</file>
```

<sup>①</sup> Trolltech 是 Qt 的初创公司——译者注。

```

<file alias="filemenu.html">help/filemenu.html</file>
<file alias="index.html">help/index.html</file>
</qresource>
<qresource lang="fr">
<file alias="editmenu.html">help/editmenu_fr.html</file>
<file alias="filemenu.html">help/filemenu_fr.html</file>
<file alias="index.html">help/index_fr.html</file>
</qresource>

```

资源文件可以有任意数量的 `<qresource>` 标记，尽管至今还没有真正用到过一个。如果当前的本地系统是“en\_US”，那么帮助文件主页将会是：/index.html；但如果当前的本地系统是“fr\_CA”或者“fr”或者任何其他的“fr\_\*”，在代码中试图访问文件：/index.html 时，那么实际得到的文件将是：/index\_fr.html。

用来创建和更新.ts 文件的工具是 pylupdate4。这个程序可以从命令行终端运行，会带一个作为参数的.pro 文件。这里给出了完整的 imagechanger.pro 文件内容：

```

FORMS      += newimagedlg.ui
SOURCES    += helpform.py
SOURCES    += imagechanger.pyw
SOURCES    += newimagedlg.py
SOURCES    += resizedlg.py
TRANSLATIONS += imagechanger_fr.ts

```

.pro 文件格式主要用于C++/Qt 程序开发人员，但如果将其用于 PyQt 工程的话，它将可以让 pylupdate4 和 lrelease 的使用变得更加容易。只需要关心三类选项即可：FORMS 用于.ui 文件，SOURCES 用于.py 和.pyw 文件，而 TRANSLATIONS 用于.ts 文件。值得注意的是，这里并没有列出.qm 文件（比如 qt\_fr.qm）；这是因为，这里并没有生成过 qt\_fr.qm 文件，而仅仅是将其从 translations 目录中复制过来而已。

这里并不需要使用任何文件中的任何一行代码；相反，还可以对文件进行分组。例如：

```

FORMS      = newimagedlg.ui
SOURCES    = helpform.py imagechanger.pyw newimagedlg.py resizedlg.py
TRANSLATIONS = imagechanger_fr.ts

```

一旦在源代码中用过 `tr()` 和 `translate()` 并创建过.pro 文件，就可以运行 pylupdate4：

```

C:\>cd c:\pyqt\chap17
C:\pyqt\chap17>pylupdate4 -verbose imagechanger.pro
Updating 'imagechanger_fr.ts'...
    Found 96 source texts (96 new and 0 already existing)

```

当然，`-verbose` 选项的使用是可选的。如果在.pro 中列示的.ts 文件不存在，pylupdate4 程序就会创建该.ts 文件，并且会将.pro 文件中 FORMS 和 SOURCES 选项内给出文件中使用 `tr()` 和 `translate()` 标记的所有上下文和字符串都放到.ts 文件中。如果.ts 文件已经存在，pylupdate4 会添加一些必要的新的上下文和字符串，而让之前已经添加过的那些内容的译文保持不变。因为 pylupdate4 比较智能，因而可以随时运行，即它在翻译人员添加或者修改译文并更新.ts 文件之后也可以运行，而不会造成任何数据的丢失。

当准备发布（或者只是做发布测试）翻译后的应用程序时，可以通过运行 lrelease，为.ts 文件生成一个.qm 文件

```
C:\pyqt\chap17>lrelease -verbose imagechanger.pro
Updating 'C:/pyqt/chap17/imagechanger_fr.qm'...
Generated 85 translations (81 finished and 4 unfinished)
Ignored 11 untranslated source texts
```

就像 pylupdate4 一样，只要愿意，可以随时随地运行 lrelease。我们这里并不需要生成 qt\_fr.qm 文件，因为我们已经将其复制过来了。

完全不使用.pro 文件也是可能的，而只需依靠 mkpyqt.py 或者 Make PyQt 构建工具。要做到这一点，就必须在命令行终端中运行过一次 pylupdate4。例如：

```
C:\>cd c:\pyqt\chap17
C:\pyqt\chap17>pylupdate4 *.py *.pyw -ts imagechanger_fr.ts
```

从现在开始，只需要运行 mkpyqt.py，带一个 -t(翻译, translate)选项，或者运行 Make PyQt 并且选中 Translate 复选框即可。随着翻译选项的打开，这两个工具都会在运行 pylupdate4 之后再运行一下 lrelease。

剩下要做的主要工作就是翻译这件事了。为此，可以让翻译人员使用 Qt Linguist 应用程序——它由C++/Qt 写成，可以运行于 Windows、Mac OS X 和 Linux——使用.ts 文件，同时要求他们输入各个字符串的译文。Qt Linguist 应用程序(如图 17.2 所示)使用起来非常简单，可以根据前面已经做出的翻译词汇给出相似的翻译建议，从而尽可能多地降低重复输入内容的次数。它可以根据上下文(通常就是窗口类的名字)对译文字符串进行分组。当某个字符串需要根据出现地方的不同而需要分别译为不同的含义时，这个分组功能非常有用。

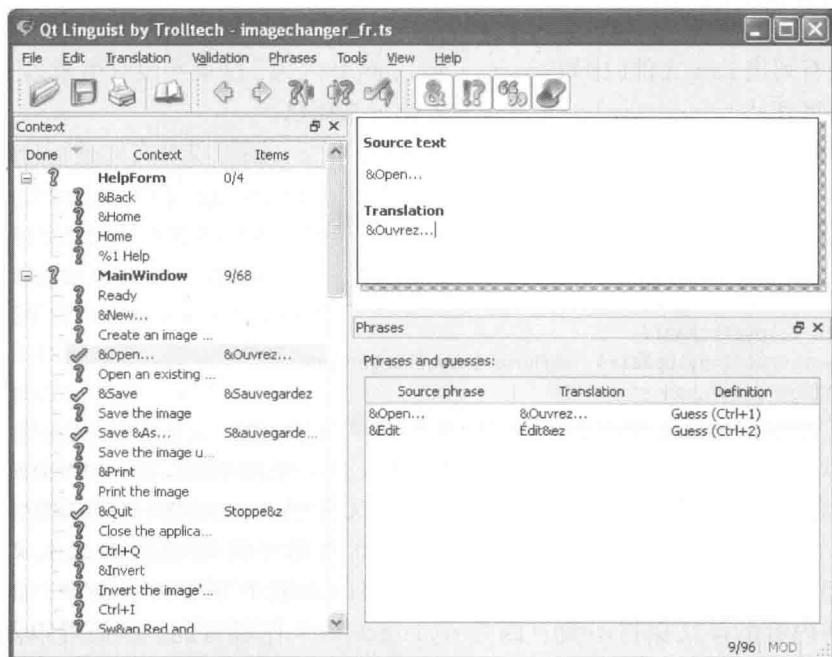


图 17.2 Qt Linguist

要开始使用 Qt Linguist，可以先运行它，点击文件(File)→打开(Open)，会打开一个.ts 文件。现在，单击 Context 停靠窗口中左侧的任意一个“田”符号，显示这些字符串的上下文，然后再点击其中一个字符串，这个字符串就会出现在右上角的面板中，位于“源文本”

(Source text) 标题的下方。单击“译文”(Translation) 标题并键入翻译的内容。要确认这个字符串翻译完成，可以点击 Context 停靠窗口中相应字符串旁边的问号图标，单击该图标会让它在问号标记或者对勾号两个之间选中一个。翻译后的会用对勾号表示“已经完成”(done)，就会由 lrelease 将其放到.qm 文件中。

## 小结

使用 QTextBrowser 或者 QDesktopServices.openUrl() 创建一个基于 HTML 的在线帮助系统都很简单，而创建一个使用 Qt Assistant 的帮助系统则相对要难一些。但无论采用哪一种方法来提供可访问的在线帮助，真正的挑战是在线帮助文档自身的内容和设计。

对待翻译的应用程序进行设置相当简单。.pro 文件通常用来列示.ts 文件和含有许多用户可见字符串的.ui、.py 和.pyw 文件，必须使用 pylupdate4 和 lrelease 让.ts 文件保持更新并生成.qm 文件。通过生成.ts 文件并使用 mkpyqt.py 或者 Make PyQt 可以避免用到.pro 文件。

在手写代码中，务必确保每一个用户可见的字符串都使用了 QObject.tr() 或者 QApplication.translate()。拥有可替换参数的字符串总会使用带有数字% n 的 QString.arg() 而不是 Python 的% 运算符。

对于数字，可能需要使用% Ln 来获得正确的千分位和小数点分隔符。可以像下面这样使用一个可用于现金符号的技巧：

```
currency = QApplication.translate("Currency", "$")
```

可以将“\$”翻译成“€”、“£”、“¥”或者其他相应的符号。对于日期，可以使用 QDateTime.toString(Qt.SystemLocaleDate) 或者 QDateTime.toString(Qt.ISODate)。对于测量单位，最好是要么提供合理的默认值，让用户可以通过配置对话框做出修改，要么是使用一个“首次运行”(first run) 对话框，请用户选择他们的单位、默认的纸张尺寸大小，等等。

## 练习题

如果能够使用多种语言，可以从全部例子或者练习题中任选其一，或者也可以是对自己的 PyQt 应用程序，将其翻译成第二语言。

如果不能使用多种语言，可以从全部例子或者练习题中任选其一，或者也可以是对自己的 PyQt 应用程序，为其添加在线帮助，包括工具栏提示和状态栏提示，与 HTML 帮助文件一样。

本章无须提供答案。

## 第 18 章 网络应用

- 创建 TCP 客户端
- 创建 TCP 服务器

Python 标准库有许多可以提供网络应用功能的模块。在第 4 章，我们曾看到过标准库在网络应用上的函数，当时曾用 `urllib2.urlopen()` 为 Internet 上的文件提供“文件句柄”，并使用 `for line in fh:` 这种形式一行一行地读取了该文件的内容。甚至也可以利用它，只从 Internet 上“抓取”整个文件：

```
source = "http://www.amk.ca/files/python/crypto/" + \
         "pycrypto-2.0.1.tar.gz"
target = source[source.rfind("/") + 1:]
name, message = urllib.urlretrieve(source, target)
```

这里的 `name` 中包含着保存资源时所使用的名字；这个例子中，它与 `target` 是一样的，不过，如果没有给出 `target`，将会为其生成一个名字——例如，`/tmp/tmpX-R8z3.tar.gz`。对于 HTTP 型的下载，`message` 会是 `httplib.HTTPMessage` 的一个实例，其中含有相应的 HTTP 报头。

Python 的 `urllib` 和 `urllib2` 标准库模块的功能非常强大。它们可以使用 FTP 和 HTTP 协议，在后面的例子中将可以看到 GET 或者 POST，也可以使用 HTTP 代理。`urllib2` 模块支持基本的认证功能，可用做 HTTP 报头。并且，如果 Python 安装了 SSL 支持功能，`urllib2` 也可以使用 HTTPS 协议。Python 标准库还包括对许多其他网络协议的支持，包括用于 `email` 的 IMAP4、POP3 和 SMTP，以及用于网络新闻的 NNTP，还包括可用于处理 cookies、XML-RPC、CGI 和用来创建服务器的库。绝大多数 Python 的网络应用支持可基于 `socket` 模块，该模块可以直接用于底层的网络编程。

除 Python 的标准库之外，PyQt4 也提供了自身的一系列网络应用类，包括用于客户端 FTP 支持的 `QFtp` 和用于 HTTP 支持的 `QHttp`。底层网络应用功能的实现可以使用 `QAbstractSocket` 的各个子类，包括 `QTcpSocket`、`QTcpServer` 和 `QUdpSocket`，以及从 Qt 4.3 之后的 `QSslSocket`。

对 Python 的网络应用支持也可以在其他的第三方库中找到，其中最为知名的可能就是 Twisted 网络应用框架；更多详情可以查阅 <http://twistedmatrix.com>。

在这一章，将只会集中精力创建一个简单的 client/server 应用程序，并且也只会使用两个 PyQt 的网络应用类来创建客户端和服务器：`QTcpSocket` 和 `QTcpServer`。在下一章，将会看到该服务器的多线程版本，具有处理多个并发请求而不会产生任何阻塞的能力。

client/server 应用程序通常会实现成两个单独的程序：服务器会对请求进行等待和响应，而一个或者多个客户端则会向服务器发送请求并读取回服务器的响应。要使其工作，客户端必须知道从何处连接到服务器上，也就是说，要知道服务器的 IP 地址和端口号。另外，无论是客户端还是服务器都必须能够使用相互认可的 `socket` 协议来发送和接收数据，也必须使用都能够理解的数据格式。

PyQt 提供两种不同类型的 `socket`。UDP ( 用户数据报协议，User Datagram Protocol ) 由

QUDpSocket 类提供支持。UDP 属轻量级协议，但不大可靠——对于待接收数据没有什么保障措施。UDP 也是无连接型协议，因而数据会以离散项的形式进行发送或者接收。TCP(传输控制协议, Transmission Control Protocol)由 QTcpSocket 类提供支持。TCP 是一种可靠连接型、面向数据流的协议；任何数量的数据都可以发送和接收——socket 负责将数据打散成足够小的数据块进行发送，也可以在另一端负责数据的重构。

UDP 通常用来监控那些持续读取数据的设备，其中的奇漏读取(odd missed reading)现象不大明显。client/server 应用程序一般会使用 TCP，因为它们要的是可靠；这也是要在这一章中使用的协议。

另一个必须要做出的决定是，是以行文本的形式还是以二进制数据段的形式来发送和接收数据。PyQt 的 TCP 套接字(socket)也是可以用在方法中的，不过还是会将其用于二进制数据中，因为这样做是最为全面且最简单的处理方式。

这里将要用到的例子是建筑设施服务应用程序(Building Services application)。其中的服务器会保存这栋建筑物中各个房间以及这些房间被预订日期的详细信息。客户端用来对某些特定日期的特定房间进行预订和退订操作。可以使用任意数量的终端，不过如果有两个终端做出的请求是在几乎完全相同的时间到达的，那么就会将其中的一个进行阻塞，直至另外一个请求处理完毕为止。通过使用多线程服务器可以大大缓解这一问题的出现，在下一章中将会看到这一点。

由于这只是一个例子，所以会在同一台计算机上运行服务器和客户端；这就意味着，可以用“localhost”作为 IP 地址。在图 18.1 中，给出了具有一个服务器和两个客户端的结构示意图。这里选择的端口号是 9407——这只是一个任意数字。端口号应当比 1023 大，并且通常会在 5001 ~ 32 767 之间，尽管端口号高达 65 535 也是有效的。服务器可以接受两种类型的请求，“BOOK”和“UNBOOK”，也可以做出三种类型的响应，“BOOK”、“UNBOOK”和“ERROR”。所有类型的请求和响应都可以以二进制数据的形式进行发送和接收；在随后的几节中，将会看到这些形式的具体格式。



图 18.1 一个服务器带两个客户端

除了在 PORT 变量中保存的端口号之外，还可以创建一个 SIZEOF\_UINT16 变量并将其设置成 2(意思是 2 个字节)。除了导入那些常规的模块之外，还需要导入 QtNetwork 模块：

```
from PyQt4.QtNetwork import *
```

对于这里的 PORT 和 SIZEOF\_UINT16 变量，以及所导入的 QtNetwork 模块，都会在客户端和服务器端应用程序中用到。在 18.1 节中，将会看到客户端的实现代码，而在 18.2 节中，将会看到服务器端的实现代码。

## 18.1 创建 TCP 客户端

建筑设施服务应用程序的客户端代码在 `chap18/buildingservicesclient.pyw` 文件中。它允许用户输入房间号(只能够接受有效的房间号)和日期, 以及发出在那一天预订(或者退订)该房间的请求。服务器端可以对该请求进行响应, 客户端会显示相应的响应, 以便用户可以读取相应的响应标签。

这里先从初始化程序开始, 不过会忽略有关创建窗口部件和布局的那些代码。将从三个部分来进行代码分析, 然后再来查看客户端的方法。

```
class BuildingServicesClient(QWidget):
    def __init__(self, parent=None):
        super(BuildingServicesClient, self).__init__(parent)
        self.socket = QTcpSocket()
        self.nextBlockSize = 0
        self.request = None
```

这里会从 `QWidget` 而不是从 `QDialog` 或者  `QMainWindow` 进行子类化。这样做最为显著的不同是, 加入是从 `QDialog` 子类化的, 按下 `Esc` 键就将终止整个应用程序。

有三个对象要保存。第一个是客户端用来与服务器端进行通信的套接字(`socket`)。第二个是“下一段的大小”;这是一个变量, 可以用来判定在处理该响应时, 是否已经接收到了足够的响应数据。第三个是一个请求对象;这是一个含有 `QByteArray` 型的请求数据, 或者若没有要发送的数据则会是 `None`。

这里会忽略有关窗口部件创建、设置以及布局的相关代码, 因为现在对于这些内容应该都已经非常熟悉了, 不过在看完 `socket` 连接之后还是会有各个窗口部件进行相互连接的内容。

```
self.connect(self.socket, SIGNAL("connected()"),
            self.sendRequest)
self.connect(self.socket, SIGNAL("readyRead()"),
            self.readResponse)
self.connect(self.socket, SIGNAL("disconnected()"),
            self.serverHasStopped)
self.connect(self.socket,
            SIGNAL("error(QAbstractSocket::SocketError)"),
            self.serverHasError)
```

前面这四个信号是关于 `socket` 的信号。需要知道是在何时建立这些连接的, 因为那时是可以发出请求数据的。也需要知道是否有需要读取的 `socket`, 因为若需要有读取的数据的话, 将可以得到服务器的响应, 这也正是希望读取到的内容。如果终止连接(例如, 由于服务器的关闭或者出现错误)就希望能够知道这些情况, 以便可以告知用户。

```
self.connect(self.roomEdit, SIGNAL("textEdited(QString)"),
            self.updateUi)
self.connect(self.dateEdit, SIGNAL("dateChanged(QDate)"),
            self.updateUi)
self.connect(self.bookButton, SIGNAL("clicked()"),
            self.book)
self.connect(self.unBookButton, SIGNAL("clicked()"),
            self.unBook)
self.connect(quitButton, SIGNAL("clicked()"), self.close)
```

其他的连接都是有关用户界面的。像通常一样, 可以使用 `updateUi()` 方法来进行验证以及

启用/禁用那些相应的按钮。也可以让那些连接进行房间的预订和退订，也可以关闭应用程序。

```
def updateUi(self):
    enabled = False
    if not self.roomEdit.text().isEmpty() and \
       self.dateEdit.date() > QDate.currentDate():
        enabled = True
    if self.request is not None:
        enabled = False
    self.bookButton.setEnabled(enabled)
    self.unBookButton.setEnabled(enabled)
```

如果在房间编辑区填有房间号，又或者日期编辑区的日期比当天要晚，那么就启用预订和退订按钮——不过，如果要是还有待处理请求（比如，`self.request` 不是 `None`）需要处理的话，就会暂时先禁用它们。

```
def closeEvent(self, event):
    self.socket.close()
    event.accept()
```

如果可以确保 `socket` 已经关闭，就可以接受 `close` 事件，终止应用程序。实际上没有必要这样做，不过，通过这样做，确实可以在应用程序关闭时起到确认的作用。

```
def book(self):
    self.issueRequest(QString("BOOK"), self.roomEdit.text(),
                      self.dateEdit.date())

def unBook(self):
    self.issueRequest(QString("UNBOOK"), self.roomEdit.text(),
                      self.dateEdit.date())
```

如果用户点击 `Book` 按钮，就会调用 `book()` 方法。这个方法只是用带有 `QString` 房间号和日期的请求动作“BOOK”来调用 `issueRequest()`。`unBook()` 方法几乎与之一样，不同的只是请求动作是“UNBOOK”而已。

```
def issueRequest(self, action, room, date):
    self.request = QByteArray()
    stream = QDataStream(self.request, QIODevice.WriteOnly)
    stream.setVersion(QDataStream.Qt_4_2)
    stream.writeUInt16(0)
    stream << action << room << date
    stream.device().seek(0)
    stream.writeUInt16(self.request.size() - SIZEOF_UINT16)
    self.updateUi()
    if self.socket.isOpen():
        self.socket.close()
    self.responseLabel.setText("Connecting to server...")
    self.socket.connectToHost("localhost", PORT)
```

这个方法用来为请求 `QByteArray` 做好准备，也用来对那个将请求发送给服务器的进程进行初始化。

图 18.2 展示了那些可写入 `QByteArray` 请求的数据。`QByteArray` 能够像其他的 `QIODevice` 一样进行读取和写入。头两个字节中含有一个无符号整数，初始值是 0。这个整数用来保存请求占用的字节数（不包括整数自身的大小数量），也就是说，是该整数之后的字节数。一开始的时候，必须要将其设置为 0，因为并不知道其中会有多少个字节。

在整数值之后，就可以写入数据了。这里的动作字符串就是请求动作（可以是“BOOK”或者“UNBOOK”），房间字符串中会保存房间号（如“213”），日期字符串会包含一个 QDate 数据。

一旦将这些数据写入到了字节数组 Byte Array 中，就可以使用 seek() 把写入位置指针移动到开头了，以便能够用下一个数据覆盖 QByteArray 的开头（实际是对 QDataStream 上的 QIODevice 进行的搜索，这只需通过调用 QDataStream.device() 即可）。要写入的 16 位无符号整数的值等于 QByteArray 的长度值减去初始整数长度大小值之后的差值。这样，要发送的请求字节数组就准备好了。

对用户界面做一下刷新——这样做将会禁用 Book 和 Unbook 按钮，因为此时的请求对象不是 None；这样可以防止用户在收到响应之前，再次做出其他请求。然后要确保 socket 已经关闭，因为 socket 可能会在处理之前的请求时被打开过，并且要让响应标签告知用户，现在打算要建立连接了。

最后，调用 connectToHost()。可以用点阵型字符串（比如，“82.94.237.218”）、主机名称（比如，“www.python.org”）或者 QHostAddress 对象的形式给出 IP 地址。借助在初始化程序中创建的那些信号-槽连接，一旦创建该连接，就会调用 sendRequest() 方法，直至该连接失效，此时，可以用 serverHasStopped() 或者 serverHasError() 方法进行代替。

```
def sendRequest(self):
    self.responseLabel.setText("Sending request...")
    self.nextBlockSize = 0
    self.socket.write(self.request)
    self.request = None
```

一旦这个连接建立，就会调用这个方法。这样会更新响应标签，告诉用户，请求已经发送，并会将下一段的大小值设置成 0。这样做，就可以对希望返回的响应进行关注；在 readResponse() 方法的用法中，将会看到这一做法。然后，会将请求字节数组写入到 socket 中。一旦写入这些数据，请求就会设置成 None，以便为新的请求做好准备。

如果没有出现错误，并且假设服务器没有关闭，服务器将会给出响应，此时就会调用 readResponse() 方法（否则，要么调用 serverHasStopped() 方法，要么调用 serverHasError() 方法）。

服务器有两种不同的响应格式，如图 18.3 所示。当接收到响应时，必须先从无符号整数中读取响应的大小值开始。然后，一旦知道了至少要有多少个字节是可以读取的，就会先去读取第一个 QString。如果其中包含的是“ERROR”文本，就可以知道，此时是一个错误响应，那么就只需读取第二个字符串，即错误信息文本即可；否则，该文本要么是“BOOK”，要么是“UNBOOK”，也就是说，这个请求动作会带有请求的详细内容，即第二个 QString 是房间号，在 QDate 中是日期，以便可以用来确认这个请求的动作成功了。

```
def readResponse(self):
    stream = QDataStream(self.socket)
    stream.setVersion(QDataStream.Qt_4_2)

    while True:
        if self.nextBlockSize == 0:
            if self.socket.bytesAvailable() < SIZEOF_UINT16:
                break
```



图 18.2 请求的格式

```

        self.nextBlockSize = stream.readUInt16()
        if self.socket.bytesAvailable() < self.nextBlockSize:
            break
        action = QString()
        room = QString()
        date = QDate()
        stream >> action >> room
        if action != "ERROR":
            stream >> date
        if action == "ERROR":
            msg = QString("Error: %1").arg(room)
        elif action == "BOOK":
            msg = QString("Booked room %1 for %2").arg(room) \
                .arg(date.toString(Qt.ISODate))
        elif action == "UNBOOK":
            msg = QString("Unbooked room %1 for %2").arg(room) \
                .arg(date.toString(Qt.ISODate))
        self.responseLabel.setText(msg)
        self.updateUi()
        self.nextBlockSize = 0
    
```

服务器的响应也有可能以碎片的形式返回。为此，可以使用一个无限循环，先接收字节数，然后再用来确保至少读取了相应字节数的内容。这样就可以让服务器来负责缓冲，也就意味着在读取的时候，可以一次读取到一个完整的响应。

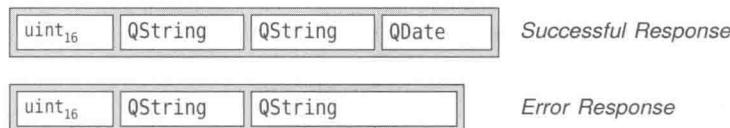


图 18.3 响应的格式

如果至少有两个字节可用，就可以以无符号 16 位整数的形式读取它们：这样就可以知道后面还有多少个字节。然后就可以进行测试，以便查明是否有要读取的足够字节：如果没有，就退出循环并等候另一个因调用 `readResponse()` 方法而生成的 `readyRead()` 信号。如果有足够的字节，就可以读取该动作以及其后的字符串——这可能是房间号也可能是错误消息。如果这个动作不是“ERROR”，还可以读取到相应的日期。然后，根据接收的是哪种动作，可以为其准备相应的消息字符串并将其在响应标签中予以显示出来。

一读取到整个响应就把下一段大小值进行重置是因为已经读取到许多字节。现在在循环时，要么等候另一个响应，本例中 `QTcpSocket.bytesAvailable()` 会返回一个比零大的值，并不停重复这一读取和显示的过程；要么没有其他响应，那么就只需跳出循环并结束即可。

```

def serverHasStopped(self):
    self.responseLabel.setText(
        "Error: Connection closed by server")
    self.socket.close()

def serverHasError(self, error):
    self.responseLabel.setText(QString("Error: %1") \
        .arg(self.socket.errorString()))
    self.socket.close()
    
```

如果关闭了服务器或者服务器以网络错误的形式进行响应（而不是用我们自己的“ERROR”响

应），就会调用相应的 `serverHas*` () 方法。在这两个例子中，都会在响应标签中向用户显示该错误消息并关闭 socket。

现在就完成了 `BuildingServicesClient` 类。用户可以输入他们的预订和退订请求并只需通过点击 `Book` 和 `Unbook` 按钮就可以将其发送给服务器，也可以在响应标签中看到请求处理的结果。通过将 `QByteArray` 写入适当的 socket 配置可以发送这些请求。通过 `QDataStream` 可以从 socket 读回这些响应；这样就可以直接把 `QString`、`QDate` 以及其他任何数据流所支持的数据类型读取到本地变量中。

现在已经看到客户端是如何创建的，就可以把注意力转移到服务器端了。

## 18.2 创建 TCP 服务器

建筑设施服务应用程序 TCP 服务器的代码放在 `chap18/buildingservicesserver.pyw` 中。它有三个组成部分：GUI 中保存着一个 TCP 服务器实例并提供了一种让用户可以关闭服务器的简单途径，`QTcpServer` 子类是经实例化后可以提供服务器实例，而 `QTcpSocket` 子类则用来处理连入的那些连接。这里先从前两个开始，因为它们都很短，然后再来关注 `QTcpSocket` 子类，大部分的工作都是在其中完成的。

```
class BuildingServicesDlg(QPushButton):
    def __init__(self, parent=None):
        super(BuildingServicesDlg, self).__init__(
            "&Close Server", parent)
        self.setWindowFlags(Qt.WindowStaysOnTopHint)

        self.loadBookings()
        self.tcpServer = TcpServer(self)
        if not self.tcpServer.listen(QHostAddress("0.0.0.0"), PORT):
            QMessageBox.critical(self, "Building Services Server",
                QString("Failed to start server: %1") \
                    .arg(self.tcpServer.errorString()))
        self.close()
        return

        self.connect(self, SIGNAL("clicked()"), self.close)
```

为了能够有些变化，也为了再次提醒大家，任何 PyQt 窗口部件都是可以作为顶层窗口的，这里让该对话框成了 `QPushButton` 的一个子类。也对 `Qt.WindowStaysOnTopHint` 进行了设置；绝大多数的窗口系统会遵从这一提示并让该窗口部件保持在其他窗口的上面。

这里没有涉及 `loadBookings()` 方法的内容；它用来管理内存中的数据结构，这里的默认字典 `Bookings`，其中就保存着房间预订数据。字典的各个键都保存成 `datetime.date` 对象的日期，各个值则都是保存成 `unicode` 字符串的房间号顺序排列而成的列表。默认字典是在 Python 2.5 中引入的。它们与普通字典类似，只是使在用了不在字典中的键时，会给它插入一个默认值。至于插入的是哪种默认值，则取决于字典的创建方式。在这个建筑设施服务应用程序的服务器代码中就这样创建过字典：

```
Bookings = collections.defaultdict(list)
```

这里要说的是，对于任何新建的默认值都会是一个空的列表；在其他情况下，或许会选用空集。总可以用普通字典替换默认字典——例如，如果使用的 Python 版本老于 2.5——在访问那些可能并不存在的键值时，可以使用 `dict.setdefault()`，而随后就会看到这一点。

这里仅会对 `TcpSever` 类进行简单查看。一旦创建了服务器，就会让它对给定的 IP 地址和端口号上连入的连接进行监听。IP 地址会以 `QHostAddress` 的形式给定，而特殊地址“`0.0.0.0`”，意思是“所有的网络接口”；端口号也是与客户端一样任意的 `9407`。

如果用户点击该按钮，这一连接就会确保关闭窗口。对于 TCP 服务器不需要做任何特殊的清空操作；在将其销毁时，会对那些连接了的任意客户端进行通知，也就会发射它们的 socket 的 `disconnected()` 信号。

对该对话框再不需要什么了，因此现在就可以来查看从 `QTcpServer` 继承来的简易 `TcpServer` 类了。

```
class TcpServer(QTcpServer):  
    def __init__(self, parent=None):  
        super(TcpServer, self).__init__(parent)  
  
    def incomingConnection(self, socketId):  
        socket = Socket(self)  
        socket.setSocketDescriptor(socketId)
```

这就是 TCP 服务器的全部代码。无论何时，只要有连入的连接请求，都会调用 `incomingConnection()` 方法，该函数会带一个 `socket` 描述符 `socketId`。只需创建一个新的 `Socket`（是 `QTcpSocket` 的一个子类，接下来很快就会看到），并让它使用服务器所提供的那个 `socket` 描述符。

这个 TCP 服务器需要根据 PyQt 的事件循环。如果打算创建一个基于 `QTcpServer` 且不带 GUI 的服务器，可以有两种方法。一种方法是使用 `QEventLoop`，以提供一个无须 GUI 的事件循环，就像这里所给出的那样来撰写代码。另一种方法是不使用事件循环，不过在这个例子中可能会稍稍有些不同。特别是，应当使用 `QTcpServer.waitForNewConnection()` 方法的阻塞而不是对 `incomingConnection()` 进行重新实现。当然，如果该服务器不带 GUI，则应当完全可以使用 Python 标准库来完成而根本不需要使用 `QtNetwork` 模块的任何东西。不过，该服务器也可以用 Twisted 编写。

一旦建立连接，全部工作都会传送给 `Socket` 类，一个马上就要看到的 `QTcpSocket` 子类。

```
class Socket(QTcpSocket):  
    def __init__(self, parent=None):  
        super(Socket, self).__init__(parent)  
        self.connect(self, SIGNAL("readyRead()"), self.readRequest)  
        self.connect(self, SIGNAL("disconnected()"), self.deleteLater)  
        self.nextBlockSize = 0
```

`socket` 会把它的 `readyRead()` 信号连接到我们的自定义 `readRequest()` 方法上，也会把它 的 `disconnected()` 信号与它的 `deleteLater()` 槽进行连接——这样就可以确保在该连接终止时，可以干净删除 `socket`。下一段大小的变量与客户端中的目的相同，使用方式也相同，以确保只有在与请求中要读取的字节至少一样多时才读取需求的内容。

一旦 `socket` 创建完毕且建立了连接，只需等候直到它的 `readRequest()` 方法得到调用。这个方法会有点长，因此会分成两个部分对其进行查看。

```
def readRequest(self):  
    stream = QDataStream(self)
```

```

stream.setVersion(QDataStream.Qt_4_2)

if self.nextBlockSize == 0:
    if self.bytesAvailable() < SIZEOF_UINT16:
        return
    self.nextBlockSize = stream.readUInt16()
if self.bytesAvailable() < self.nextBlockSize:
    return

action = QString()
room = QString()
date = QDate()

```

先从是否至少有两个需要读取的字节开始：如果有，读入下一段的大小值。如果要读取的不是两个字节，或者如果没有足够的字节可供读取整个请求，返回并在有更多字节抵达时等待 `readRequest()` 的再次调用。

一旦有了足够的字节，可以创建空的动作和房间号字符串，以及一个空的 `QDate`，以用来为连入的请求数据而做好准备。

```

stream >> action
if action in ("BOOK", "UNBOOK"):
    stream >> room >> date
    bookings = Bookings.get(date.toPyDate())
    uroom = unicode(room)
if action == "BOOK":
    if bookings is None:
        bookings = Bookings[date.toPyDate()]
    if len(bookings) < MAX_BOOKINGS_PER_DAY:
        if uroom in bookings:
            self.sendError("Cannot accept duplicate booking")
        else:
            bisect.insort(bookings, uroom)
            self.sendReply(action, room, date)
        else:
            self.sendError(QString("%1 is fully booked") \
                .arg(date.toString(Qt.ISODate)))
    elif action == "UNBOOK":
        if bookings is None or uroom not in bookings:
            self.sendError("Cannot unbook nonexistent booking")
        else:
            bookings.remove(uroom)
            self.sendReply(action, room, date)
    else:
        self.sendError("Unrecognized request")

```

服务器只认识两个请求动作，“BOOK”和“UNBOOK”；如果服务器获得了其中之一，就会读取房间号和日期，并获得（也可能是空的）给定日期的预订清单。它还会保存一个 `QString` 型房间号的 `unicode` 副本，因为服务器使用的 `Bookings` 字典会使用 Python 类型而不是 PyQt 类型来保存全部数据。

接下来，服务器会尝试预订或者退订给定日期的特定房间。在预订时，如果给定的日期没有预订操作，那么就为当天日期创建一个空的预订列表。之所以可以这样工作是因为这里使用到了默认字典，因此在使用它所没有的键来访问该字典时，它就会为给定的键自动插入一个默认值的新项；在本例中，就是一个空的预订列表。这些代码看起来稍显巧妙，因为它是从调用 `get()` 开始的。如果动作是退订，这样做就可以避免生成一个空的预订列表。仅在知道该

动作是要打算确认的预订动作时，才会为给定的日期生成一个列表。

如果使用的是普通字典，就应使用 `dict.setdefault()` 在列表中检索所给定的日期，用给定的日期创建一个新的项和作为其值的一个空的列表，如果该键还没有，例如：

```
bookings = Bookings.setdefault(date.toPyDate(), [])
```

`QDate.toPyDate()` 方法是在 PyQt 4.1 中引入的；对于之前的 PyQt 版本，可以借助 `dateTime.date()`（如 `date.year()`、`date.month()` 以及 `date.day()` 等）来执行这种转换。

一旦有了自己的（可能是空的）预订列表，并且假设预订数小于所允许的最大预订量（`MAX_BOOKINGS_PER_DAY`，其值是 5），就把房间号字符串依次插入到列表中，再给客户端发送一个回复来作为该请求数据的回应。如果该房间在给定的日期已经预订出去了，或者如果该日期已经是最大预订数了，就需要给客户端发送一个错误消息作为回应。

如果该动作是退订，就需要从给定日期的预订列表中移除该房间号并在客户端中回应一下该动作；或者如果该预订并不存在，就给出一个错误响应。由于各个房间号是依次存储的，只需使用 `not in` 和 `list.remove()`，两者都可以执行一次线性搜索；对于比较长的列表，应当使用基于二分法的 `bisect.bisect_left()` 来查找房间号，不过，在这个例子中则显得有些“杀鸡用牛刀”。

如果无法识别该请求动作，只需用一条错误消息进行回应即可。

```
def sendReply(self, action, room, date):
    reply = QByteArray()
    stream = QDataStream(reply, QIODevice.WriteOnly)
    stream.setVersion(QDataStream.Qt_4_2)
    stream.writeUInt16(0)
    stream << action << room << date
    stream.device().seek(0)
    stream.writeUInt16(reply.size() - SIZEOF_UINT16)
    self.write(reply)
```

发送给客户端的响应的创建方式与客户端请求的创建方式相同。使用 `QDataStream` 写入 `QByteArray`，先从写入一个无符号 16 位整数开始并用响应的大小值来覆盖该整数值为结尾，然后再把该响应写入 `socket` 中。

```
def sendError(self, msg):
    reply = QByteArray()
    stream = QDataStream(reply, QIODevice.WriteOnly)
    stream.setVersion(QDataStream.Qt_4_2)
    stream.writeUInt16(0)
    stream << QString("ERROR") << QString(msg)
    stream.device().seek(0)
    stream.writeUInt16(reply.size() - SIZEOF_UINT16)
    self.write(reply)
```

发送一条错误响应的代码几乎与发送一条成功响应的代码完全一样，因而这里会为两者使用同样的方法。

只需要向 `readRequest()` 添加更多的 `if` 语句，该服务器就可以轻松地以扩展的方法来处理更多的请求类型。例如，客户端或许就希望知道在某个特定的日期是哪间客房已经预订过，又或者希望知道某间客房是在哪些日期都已经预订过了。

尽管是使用字典来保存服务器的数据，但没有理由不想到为何不把服务器中的数据处理放到 SQLite 数据库、非过程型数据库或者几个文件中。也没有必要认为服务器端就不需要

GUI；当然，它也可以没有 QWidget，只需要像 Linux 的守护进程或者 Windows 的服务一样在后台运行即可。

## 小结

Python 标准库、Twisted 网络引擎和 PyQt QtNetwork 模块为网络应用提供了相当不错的支持，无论是从底层的 socket 还是不同的高级协议，包括 FTP 和 HTTP。

为编写客户端/服务器端应用程序，必须确保客户端和服务器端程序能够相互通信。这就意味着，服务器必须在已知的 IP 地址上运行并且可以监听给定的端口地址。无论是客户端还是服务器端都必须使用一致的协议进行通信，比如 UDP 或者更常见的 TCP 等。它们也必须使用一致的数据传输方式，是以文本行的方式，还是以二进制数据块的形式——无论是哪种形式，都必须知道每个请求采用的格式和每个响应带的形式。

本章中给出的例子是一个非常常见的例子：服务器坐等请求，客户端发送请求并读取服务器端的响应。在客户端可以通信之前必须先建立一个连接，然后，一旦建立了该连接，就可以发送它的数据了。服务器或许会用数据进行响应，也或许会出现一些问题。如果接收到了数据，就必须确保再不要试图读取比可用的更多的字节（或者文本行）。

PyQt 的 QTcpServer 和 QTcpSocket 类会让服务器的实现变得非常简单。尽管采用文本型数据进行读取和写入是可行的，而使用二进制数据也是可以的，从而可以允许对任意类型的数据进行发送和接收而无须再编写解析程序。

之前已经完成的 TCP 服务器存在一个理论性问题，这是一个单线程程序。这就意味着，如果一次有多个请求同时到达，就不得不阻塞一个请求了。使用线程化服务器可以有效解决这一问题，这将会在下一章中看到。

## 练习题

对本章建筑设施服务（Building Services）应用程序的服务器进行修改，以便使它可以接受新的请求动作 BOOKINGSONDATE。在接收到这种请求时，它应当忽略房间号，相反，而是找到给定日期的全部预订信息。如果没有预订，服务器应当发出一条错误响应。否则，它应当发送一个字符串，其中不应只有一个房间号，而是一个使用逗号分隔各个房间号的字符串，如图 18.4 所示。

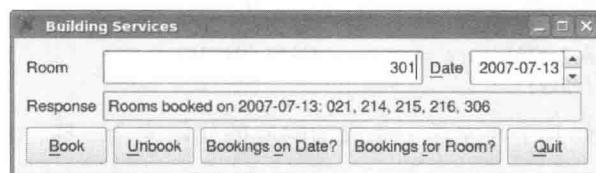


图 18.4 建筑设施服务应用程序——特定日期的预订信息

修改建筑设施服务应用程序的客户端，以便使它可以有一个 Bookings on Date? 按钮，并使该按钮能够连接到生成适当请求的方法上。客户端的 readResponse() 方法将需要做些小修改，以便它可以读取服务器对新请求的响应。

对提供“bookings on date”功能所做的必要修改相当简单。为了多些挑战，还可以修改建筑设施服务应用程序的服务器端，以便可以接受其他的新的请求动作，如 BOOKINGSFORROOM。在接收到这些请求中的任何一个时，都应该忽略日期，然后会遍历全部的预订信息，把给定房间的预订日期整合成一个日期列表。如果没有预订信息，就应当返回一条错误响应。否则，不要使用 sendReply() 方法，它应当发送自己的字节数组，其中含有它的长度、动作、房间字符串和一个表明列表中含有日期天数的 32 位整数，整数之后接着就是各个日期。由于这些日期是存储成 QDateTime.date 对象的，它们必须把流中的 QDate 转换到 QByteArray 中。

建筑设施服务应用程序的客户端也必须要做些修改，以便可以提供一个 Bookings for Room? 按钮，从而能够将其连接到发出适当请求的方法上。客户端的 readResponse() 方法将需要做些修改，以便在接收到 BOOKINGSFORROOM 响应时，可以读取日期，创建一个适当的字符串并在客户端用户界面中予以显示，如图 18.5 所示。

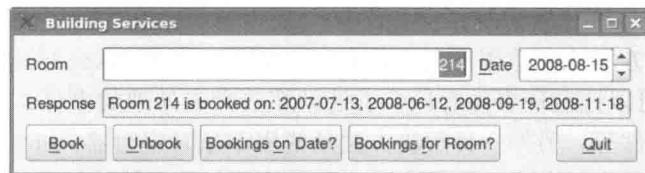


图 18.5 建筑设施服务应用程序——房间预订情况

大约添加 30 行代码是可以完成对服务器端修改的，而对于客户端的修改则可能需要添加 40 行。不过，BOOKINGSFORROOM 请求/响应确实需要稍加留意。

本练习题的参考答案在文件 chap18/buildingservicesserver\_ans.pyw 和 chap18/buildingservicesclient\_ans.pyw 中。

## 第 19 章 多线程

- 创建线程服务器
- 创建和管理次线程
- 实现次线程

一般来说，应用程序都只有一个单一的执行线程且一次只执行一种操作。对于 GUI 程序来说，这样做有时就会出现问题，例如，如果用户执行的是一个相当耗时的操作，那么当此操作正处在运行时，整个应用程序的用户界面就会冻结不动。有好多种解决方案可以解决这一问题。

一种简单的解决方案特别适用于长时间运行的循环，调用 `QApplication.processEvents()`。这一方法可以让循环事件有机会去处理那些尚未处理的事件，比如绘制事件，又比如鼠标和键的按下事件等。另外一种解决方法是使用零超时定时器(zero-timeout timer)。在很多例子中，会将以上两种方法结合起来使用，通常用于大量文件的加载过程中，例如，在第 9 章的文本编辑器应用程序中的 `MainWindow.loadFiles()` 方法中。

第三种解决方案则是将当前工作完全转交给另一个程序进行处理。使用 Python 标准库的子进程模块(`subprocess module`)可以实现这一点，也可以使用 PyQt 的 `QProcess` 类。在 `makepyqt.pyw` 应用程序中就提供了使用 `QProcess` 来执行诸如 `pyuic4` 和 `pyrcc4` 那样的 PyQt 命令行工具的示例。

在某些情况下，真正需要的可能是在应用程序本身中执行一个与执行线程相分离的单独线程。这种具有多于一个执行线程的应用程序就是所谓的多线程<sup>①</sup>。比如，可能会打算创建一个服务器，它能够像硬件一样来同时处理很多的并联连接，如果每个连接都能够像多线程那样处理事情，那么很多事情实现起来就会容易得多。在某些情况下，可能会有一个 GUI 应用程序，希望该程序能够在用户启动一个长时间运行的进程后，继续与应用程序保持交互的能力；在这种情况下，最好是把运行进程传给能够单独运行的次线程，以便让主(GUI)线程能够空闲下来，从而及时响应用户的其他需求。

本章将会给出一些用在多线程编程中的常见技术。这些技术对于初学者来说足以入门，但所涉及的内容尚不够全面，因为很多技术都超出了本书的范围，而且如要对它们做详细介绍的话，恐怕另外再需要一本书了。

因为多个线程可能会同时访问同一份数据，多线程应用程序在编写、维护和调试上相比单线程应用程序来说也要困难得多。在单处理器的机器上，多线程应用程序的运行速度有时要比单线程应用程序慢一些(由于还有一些额外的线程操作)，但用户也通常会觉得多线程应用程序运行得更快，因为这些应用程序不会冻结用户界面，并且也因为它们会通过运行进程来不断地向用户提供反馈。

使用适当数量的线程会显著影响应用程序的性能。例如，在本章后面要讲述到的页面索

<sup>①</sup> 这一章假定你对线程化的概念稍有了解。为全面考虑，可以参阅 *Foundations of Multithreaded, Parallel, and Distributed Programming* 一书中“引言”的内容。

引应用程序 Page Indexer 中，就有一个主(GUI)线程和一个次线程(secondary thread)。在这一章的练习题中，就会要求将这个应用程序改变成使用次线程的形式。如果用到的次线程太多，应用程序的运行速度就会比仅使用主线程的应用程序还要慢，但若线程的数量合适，就可以先将应用程序单个次线程的版本启动，然后再将应用程序多个次线程的版本启动，看看多个次线程版本的应用程序是如何在单个次线程版本的应用程序结束之前赶上、超过和结束的。那么，到底应该使用多少个次线程呢？这个问题的答案取决于所必须完成的操作进程类型，应用程序所要运行的特定机器，以及机器上该应用程序运行时所基于的操作系统。可以用真实的数据集来试验出合适的固定数字，或者也可以视具体情况而定，以便让代码能够使用更多或者更少数量的次线程。

Python 的标准库提供了低层次的线程模块和高层次的线程模块，但对于 PyQt 编程来说，则建议使用 PyQt 的线程类。PyQt 的各个线程类提供了高层次的 API，但是在这些类的底层的一些基本操作中，则为了使它们的运行能够尽可能快和细粒度而采用了汇编语言进行实现，所以，有部分操作并没有在 Python 的线程模块中完成。

PyQt 的应用程序总是至少有一个执行线程，即主(初始)线程。此外，应用程序可根据需要创建所需的一些次线程。然而，如果应用程序有 GUI，则所述的 GUI 操作，比如执行事件循环等，就可能只发生在主线程中。通过重新实现 `QThread.run()` 方法来实例化 `QThread` 的一些子类，可以创建一些新的线程。

创建一个不具有 GUI 的 PyQt 应用程序也是可以的，这就要使用 `QCoreApplication` 来代替 `QApplication`。就像一些 GUI 型 PyQt 应用程序一样，它们有一个主线程，或许还有一定数量的次线程。

在次线程和主线程之间进行通信通常也是用户希望的——例如，使用户了解进程的执行情况，以允许用户干预进程的处理过程，或者是让主线程知道次线程可以在何时处理完成等。一般来说，会使用共享变量和一定的资源保护机制相结合的方式来实现这种通信。

PyQt 会有一些类来支持这种做法，其中包括 `QMutex`、`QReadWriteLock` 和 `QSemaphore` 等。此外，PyQt 应用程序可以使用信号-槽机制在线程之间进行通信；这点非常方便和实用。

在 19.1 节中，将会看到一个线程化的 TCP 服务器；它的工作方式与上一章最后一节中描述的服务器一样，但不同的是，由于它采用了多线程，所以它可以同时服务于多个客户端。在 19.2 节和 19.3 节，将会看到一个 GUI 应用程序，它会处理一个非常耗时的进程，但它会把该处理进程传给次线程加以处理。该应用程序利用信号和槽来使用户界面能够随进程而实时更新，并让用户可以对次线程加以控制。这个例子还使用了一些资源保护类，使得用户界面能够访问正在处理的工作。

## 19.1 创建线程服务器

与其他一些 GUI 库不同，PyQt 的网络套接字类会和事件循环整合到一起。这就意味着，在网络处理过程中用户界面仍旧可以保持相应的响应，即使是在单线程的 PyQt 应用程序中也是如此。但如果想要能够处理多个传入的并发连接，则可能会更希望使用多线程服务器。

创建一个多线程服务器并不比创建一个单线程服务器复杂多少——二者的差异并不在于

说是创建了一个单独的套接字来处理传入的那些连接，一个多线程服务器会为每一个新的连接都分别创建一个新的线程，并且会为每个新的线程创建一个新的套接字。例如，下面就是一个完整的线程服务器：

```
class TcpServer(QTcpServer):
    def __init__(self, parent=None):
        super(TcpServer, self).__init__(parent)

    def incomingConnection(self, socketId):
        thread = Thread(socketId, self)
        self.connect(thread, SIGNAL("finished()"),
                     thread, SLOT("deleteLater()"))
        thread.start()
```

这里的 `incomingConnection()` 方法是对 `QTcpServer` 基类中该方法的重新实现。无论何时，只要和服务器之间创建了一个新的网络连接，就会调用这个方法。

信号-槽连接是必要的，可以用来确保删除那些不再需要的线程，从而可以使服务器的内存占用量尽可能小一些。虽然必须重新实现 `QThread` 子类中的 `QThread.run()`，但开启线程还总是要通过调用 `QThread.start()` 的（而永远不要直接调用 `run()`）。

`Thread` 线程子类有一个静态变量和四个方法。`sendReply()` 和 `sendError()` 方法与在上一章中给出的一样，所以这里忽略它们。

```
class Thread(QThread):
    lock = QReadWriteLock()

    def __init__(self, socketId, parent):
        super(Thread, self).__init__(parent)
        self.socketId = socketId
```

`Thread.lock` 是一个静态变量，因此所有 `Thread` 实例都会共享此变量。初始化程序只是简单地关注那些准备用于线程启动的套接字描述符。`run()` 方法的代码很长，所以这里分成不同的部分来查看它的代码。

```
def run(self):
    socket = QTcpSocket()
    if not socket.setSocketDescriptor(self.socketId):
        self.emit(SIGNAL("error(int)"), socket.error())
        return
    while socket.state() == QAbstractSocket.ConnectedState:
        nextBlockSize = 0
        stream = QDataStream(socket)
        stream.setVersion(QDataStream.Qt_4_2)
        while True:
            socket.waitForReadyRead(-1)
            if socket.bytesAvailable() >= SIZEOF_UINT16:
                nextBlockSize = stream.readUInt16()
                break
            if socket.bytesAvailable() < nextBlockSize:
                while True:
                    socket.waitForReadyRead(-1)
                    if socket.bytesAvailable() >= nextBlockSize:
                        break
```

先从创建一个新的套接字并将其套接字描述符设置成已经给出过的套接字描述开始。这里采取一种比之前所用方法稍微更为保险的方法，即检查 `QTcpSocket.setSocketDescriptor`

() 调用的返回值，而如果失败的话则返回一条错误消息。一旦 run() 方法完成，就会发射 finished() 信号，借助之前的信号-槽连接机制，也就可以确保该线程的删除。

只要连接了套接字，就可以用它接收请求并发送响应。与前一章所创建的 TCP 服务器不同，它不是以异步方式运行或者是等待事情的发生，如数据的可用性，借助信号-槽连接机制，这里则用 waitReadyRead() 进行阻塞并一直等到有数据为止（这里的参数 -1 意味着“永远等待”）。至于用 waitReadyRead() 进行阻塞不会有什么问题，因为它是在一个单独的执行线程内进行的，所以对于应用程序其余部分来说，其主线程和任何其他连接都可以不受阻碍地继续对次线程进行处理。

一旦有两个字节可用，就可以读取它的 16 位无符号字节计数值，并且一次读取的字节数至少要满足读取需要，这样就可以继续处理了。

```
action = QString()
room = QString()
date = QDate()
stream >> action
if action in ("BOOK", "UNBOOK"):
    stream >> room >> date
try:
    Thread.lock.lockForRead()
    bookings = Bookings.get(date.toPyDate())
finally:
    Thread.lock.unlock()
uroom = unicode(room)
```

按照要求动作进行读取时，要求动作应该是“BOOK”或者是“UNBOOK”，而如果确实是二者之一，那么就可以接着读取房间号码字符串和日期。Bookings 预订的默认字典会保存全部的预订数据，还会保存能够同时访问的线程数量。为此，就必须保护每一个访问。在这里，我们只是想读取数据，所以可以调用 lockForRead()，提取想要的数据，然后再打开阻塞锁即可。这里使用 try...finally 块来保证该阻塞锁可以在访问共享数据时是处于解锁状态的。

Python 2.6（以及 Python 2.5 中与之相适应的 from\_future\_ 语法）中提供了更好和更为紧凑的语法来代替 try...finally，这一点可以参见“用于解锁的上下文管理器”中的内容。

互斥体[ mutex，也称为二进制信号量（binary semaphore）] 是一种知名的锁机制，是由 PyQt 的 QMutex 类提供的。互斥体只允许对其锁定的线程访问受保护的资源。PyQt 还提供了粒度更细的机制，即读/写锁（read/write lock），是由之前用过的 QReadWriteLock 类提供的。每当锁对一个线程进行有效锁定时，可能会阻塞其他线程，以等待访问。为缓解这一问题，可以从两个方面着手。第一，尽可能多地使用读取锁——如果有效锁全部都是读取锁，那么也就不会阻塞任何线程，其原因是，如果没有线程正在写操作，那么所有线程的读取都会是安全的。第二，有效锁定时，将所做的处理工作量压缩至最少。这两种技术都用在了对 run() 的重新实现中；其缺点是，代码量或许要比预期的长一些。

QMutex、QReadWriteLock 和其他保护机制都可以正常工作，因为它们都是“线程安全”（thread-safe）的。任意数量的线程都可以同时调用线程安全对象的方法，并可以借助底层系统，即 PyQt，对任何可能出现的共享数据的访问进行自动序列化。这就意味着，例如，如果有任何两个或者更多的线程试图锁定 QReadWriteLock 以进行写操作，那最后必然只有一个线程可以成功锁定，而其他的线程则都会被阻塞掉。这种方法就可以允许那个获得了锁的线程来执行它对共享数据的更新，而当它释放了对该线程的锁定后，其他想要进行写操作的线程就

说是创建了一个单独的套接字来处理传入的那些连接，一个多线程服务器会为每一个新的连接都分别创建一个新的线程，并且会为每个新的线程创建一个新的套接字。例如，下面就是一个完整的线程服务器：

```
class TcpServer(QTcpServer):
    def __init__(self, parent=None):
        super(TcpServer, self).__init__(parent)

    def incomingConnection(self, socketId):
        thread = Thread(socketId, self)
        self.connect(thread, SIGNAL("finished()"),
                    thread, SLOT("deleteLater()"))
        thread.start()
```

这里的 `incomingConnection()` 方法是对 `QTcpServer` 基类中该方法的重新实现。无论何时，只要和服务器之间创建了一个新的网络连接，就会调用这个方法。

信号-槽连接是必要的，可以用来确保删除那些不再需要的线程，从而可以使服务器的内存占用量尽可能小一些。虽然必须重新实现 `QThread` 子类中的 `QThread.run()`，但开启线程还总是要通过调用 `QThread.start()` 的（而永远不要直接调用 `run()`）。

`Thread` 线程子类有一个静态变量和四个方法。`sendReply()` 和 `sendError()` 方法与在上一章中给出的一样，所以这里忽略它们。

```
class Thread(QThread):
    lock = QReadWriteLock()

    def __init__(self, socketId, parent):
        super(Thread, self).__init__(parent)
        self.socketId = socketId
```

`Thread.lock` 是一个静态变量，因此所有 `Thread` 实例都会共享此变量。初始化程序只是简单地关注那些准备用于线程启动的套接字描述符。`run()` 方法的代码很长，所以这里分成不同的部分来查看它的代码。

```
def run(self):
    socket = QTcpSocket()
    if not socket.setSocketDescriptor(self.socketId):
        self.emit(SIGNAL("error(int)"), socket.error())
        return
    while socket.state() == QAbstractSocket.ConnectedState:
        nextBlockSize = 0
        stream = QDataStream(socket)
        stream.setVersion(QDataStream.Qt_4_2)
        while True:
            socket.waitForReadyRead(-1)
            if socket.bytesAvailable() >= SIZEOF_UINT16:
                nextBlockSize = stream.readUInt16()
                break
            if socket.bytesAvailable() < nextBlockSize:
                while True:
                    socket.waitForReadyRead(-1)
                    if socket.bytesAvailable() >= nextBlockSize:
                        break
```

先从创建一个新的套接字并将其套接字描述符设置成已经给出过的套接字描述开始。这里采取一种比之前所用方法稍微更为保险的方法，即检查 `QTcpSocket.setSocketDescriptor`

```

        bisect.insort(bookings, uroom)
    finally:
        Thread.lock.unlock()
        self.sendReply(socket, action, room, date)
else:
    self.sendError(socket, error)

```

如果在给定的日期该房间已经预订出去，则不应重复预订该房间，相反，应当将一条错误响应信息发送到客户端。在非线程化的服务器中，只有在这种情况下才会调用 `sendError()`，但在这里，则只是赋值一下错误消息文本。这样做可以使得进程锁的上下文能够在尽可能小的范围内及时完成处理操作。

如果可以实现该项预订，就可以用一个写操作锁，将该房间插入到预订列表中，同时发送一个响应来说明预订成功了。否则，就发送一个错误响应。这两个响应都是在写操作锁的上下文范围内发送出去的，从而可以尽可能降低写操作锁的影响范围。

```

elif action == "UNBOOK":
    error = None
    remove = False
    try:
        Thread.lock.lockForRead()
        if bookings is None or uroom not in bookings:
            error = "Cannot unbook nonexistent booking"
        else:
            remove = True
    finally:
        Thread.lock.unlock()
    if remove:
        try:
            Thread.lock.lockForWrite()
            bookings.remove(uroom)
        finally:
            Thread.lock.unlock()
        self.sendReply(socket, action, room, date)
    else:
        self.sendError(socket, error)

```

这里的 unbooking `if` 分支与 booking 分支有着相同的处理模式。这里先从检查是否可以进行预订开始，会使用读取锁，如果有必要的话，来存储一条错误消息，而不是在锁定生效时去执行一个非常耗时的发送操作。然后，要么以给定的日期来从预定字典 `Bookings` 的列表中删除房间，取消预订，发送处理成功的响应，要么就直接发送一个出错的响应。再次重申，只有在没有锁定生效的情况下才会发送这些响应的内容。

```

else:
    self.sendError(socket, "Unrecognized request")
    socket.waitForDisconnected()

```

如果服务器接收到的是一个无法识别的请求，就只是发送一条错误信息。最后，调用 `QTcpSocket.waitForDisconnected()`；这个锁定会一直有效到连接关闭，也就是说，直到相应的响应信息发送成功才会解除锁定。并不需要或是要求连接一直处于开放状态，因为客户端/服务器型应用程序的运行采用成对独立的请求-响应处理模式。一旦连接关闭，`run()` 方法也就完成了，还要归功于 `deleteLater()` 这一信号-槽连接，也会将该线程删除掉。

## 19.2 创建和管理次线程

在 GUI 应用程序中，线程的一个常见用法就是将处理任务传给一个次线程，以便让用户界面可以保持响应状态并且可以显示次线程的处理进度。在这一节中，我们将会看到一个页面索引器 (Page Indexer) 应用程序，如图 19.1 所示，该应用程序可在特殊的地址目录及其所有子目录中进行 HTML 文件的索引。索引编制的工作会传递给一个与主线程通信的次线程，从而可以告诉主线程索引完成处理工作的实时完成进度。



图 19.1 页面索引器 (Page Indexer) 应用程序

对于索引工作所要使用的算法是：对于遇到的每个 HTML 文件，读取其文本内容，将各个实体都转换成等效的 Unicode 字符，同时去除 HTML 标记。然后，将文本拆分成多个单词，每个单词的长度是 3~25 个字符，再把文本中那些不常用的词都添加到 `filenamesForWords` 默认字典中。字典中的每个键都是一个独特的词，并且每个相关的值都是这些词所出现的文件名的集合。而如果有任何一个单词同时出现在超过 250 个文件中，就会将其从 `filenamesForWords` 字典中删除并添加到常用词集合中。这样就可以确保把字典始终保持在一个合理的大小，也同时意味着搜索诸如“and”和“the”这样的单词时，算法根本就不必工作——这是一件好事，因为这样的单词有可能会出现在数以万计的文件中，这些数量太多的单词一般不会有什么用。

下面将先从应用程序中主窗体部分提取的两段代码看起，这些代码都放在本书文件 `chap19/pageindexer.pyw` 中。

```
class Form(QDialog):
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)
        self.fileCount = 0
        self.filenamesForWords = collections.defaultdict(set)
        self.commonWords = set()
        self.lock = QReadWriteLock()
        self.path = QDir.homePath()
```

`fileCount` 变量用来跟踪到目前为止到底索引了多少个文件。`filenamesForWords` 默认字典的键是单词，而其值是文件名 (`filenames`) 的集合。`commonWords` 集会把那些至少在 250 个

文件中出现过的单词保存下来。读/写锁用于确保对 filenamesForWords 字典和 commonWords 集的访问进行保护，因为会在主线程中读取它们并在次线程中进行写操作。 QDir.homePath() 方法可以返回用户的主目录；使用它可以设置一个初始的搜索路径。

```
self.walker = walker.Walker(self.lock, self)
self.connect(self.walker, SIGNAL("indexed(QString)"),
            self.indexed)
self.connect(self.walker, SIGNAL("finished(bool)"),
            self.finished)
self.connect(self.pathButton, SIGNAL("clicked()"),
            self.setPath)
self.connect(self.findEdit, SIGNAL("returnPressed()"),
            self.find)
```

次线程在 walker 模块(之所以这样命名是因为它就像是在文件系统中漫步一样)中，而 QThread 的子类叫做 Walker。每当线程索引到一个新的文件，它就会发射一个带有该文件名的信号。当在一开始所给定的路径中的全部文件都被索引完成后，它还会发射一个 finished() 信号。

在一个线程中发射的信号会认为是另一个异步工作，也就是，它们并不会被阻塞。但只有在接收端恰有一个事件循环时它们才会起作用。这就意味着，次线程可以利用这些信号将信息传递给主线程，但不会采用其他方式——除非是在次线程中运行了一个独立的事件循环(这是可能的)。实际上，当发射跨线程的各个信号时，不像在同一线程中处理发射信号和接收信号时直接调用相关方法的处理方式那样，PyQt 会在带有传递数据的接收线程事件队列上放一个事件。当接收线程的事件循环读取该事件时，它就会通过调用带有所传数据的相关方法对其进行响应。

如图 19.2 所示，主线程通常会使用方法调用(method call)的形式来将信息传递给次线程，而次线程则通过信号来将信息传递给主线程。还有一种通信机制，同时由主线程和次线程所使用，是使用共享数据。这种数据必须具有访问保护——例如，通过互斥体或通过读/写锁定来进行保护。

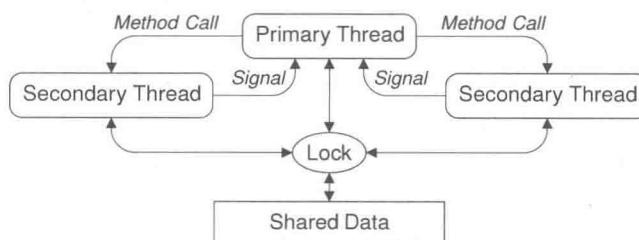


图 19.2 典型的 PyQt 线程间通信模式示意图

如果用户单击设置路径(Set Path)按钮，就会调用这里的 setPath() 方法，并且如果用户在查找行编辑框上按下了 Enter 键，就会调用 find() 方法。

窗体类 Form 实质上就是一个 QDialog，但在这里却对其进行设计的原因是，如果在进行索引的过程中用户按下了 Esc 键，那么索引就应当予以停止，而如果索引完成(或者是被停止)时用户按下了 Esc 键，那么应用程序就应当予以终止。下面就来看看在对 accept() 和 reject() 进行重新实现时是如何实现这一点的。

```

def setPath(self):
    self.pathButton.setEnabled(False)
    if self.walker.isRunning():
        self.walker.stop()
        self.walker.wait()
    path = QFileDialog.getExistingDirectory(self,
                                            "Choose a Path to Index", self.path)
    if path.isEmpty():
        self.statusLabel.setText("Click the 'Set Path' "
                                "button to start indexing")
        self.pathButton.setEnabled(True)
        return
    self.path = QDir.toNativeSeparators(path)
    self.findEdit.setFocus()
    self.pathLabel.setText(self.path)
    self.statusLabel.clear()
    self.filesListWidget.clear()
    self.fileCount = 0
    self.filenamesForWords = collections.defaultdict(set)
    self.commonWords = set()
    self.walker.initialize(unicode(self.path),
                          self.filenamesForWords, self.commonWords)
    self.walker.start()

```

当用户点击设置路径(Set Path)按钮时，一开始会先禁用该按钮，然后才会停止那个正在运行的线程。这里的 stop() 方法是一个自定义方法。wait() 方法则继承自 QThread；它会一直锁定到线程结束运行为止，那就是说，直到 run() 方法有了返回才会解锁。在 stop() 方法中，会间接确保 run() 方法可以在 stop() 方法调用一结束就完成运行，在下一节将会看到这一点。

接下来将会得到用户选定的路径(或者如果用户取消操作，则直接返回)。由于在 PyQt 中，总是使用“/”来分隔路径的各个部分，而在 Windows 操作系统上却会显示成“\”，所以这里会使用 QDir.toNativeSeparators()。不过，这个 toNativeSeparators() 方法是在 Qt 4.2 之后才引入的；对于更早期版本的用户来说，则可以使用 QDir.convertSeparators() 来代替它。默认情况下，getExistingDirectory() 只会显示目录，因为它的第四个可选参数会带一个默认值 QFileDialog.ShowDirsOnly；如果想让文件名可见，就可以通过传递 QFileDialog.Options() 的值来清除该标志。

用户界面已经做过设置，可以通过键盘将焦点移动到查找行编辑框上，从而将路径标签的内容设置成所选定的路径，并同时清空那个用来使用户了解处理进度的状态标签。文件列表窗口部件会列出含有查找行编辑框内输入单词的文件的清单。这里并不需要对 filenamesForWords 默认字典或者对 commonWords 集进行访问保护，因为此时唯一正在运行的线程只有主线程一个线程。

通过用该路径和对打算填充的数据结构的引用来初始化 walker 线程后，就可以作为结束了，接下来，将会调用 start() 来开始执行它了。

```

def indexed(self, fname):
    self.statusLabel.setText(fname)
    self.fileCount += 1
    if self.fileCount % 25 == 0:
        self.filesIndexedLCD.display(self.fileCount)
    try:

```

```

        self.lock.lockForRead()
        indexedWordCount = len(self.filenamesForWords)
        commonWordCount = len(self.commonWords)
    finally:
        self.lock.unlock()
        self.wordsIndexedLCD.display(indexedWordCount)
        self.commonWordsLCD.display(commonWordCount)
    elif self.fileCount % 101 == 0:
        self.commonWordsListWidget.clear()
    try:
        self.lock.lockForRead()
        words = self.commonWords.copy()
    finally:
        self.lock.unlock()
        self.commonWordsListWidget.addItems(sorted(words))

```

只要 walker 线程一完成对文件的索引，它就会发射一个带有文件名的 `indexed()` 信号；而这个信号就会连接到前面给出的 `Form.indexed()` 方法上。这里会更新状态标签，以用其显示刚才索引过的文件名称，并且会在每 25 个文件后就更新一次文件计数值、索引单词和常用词 LCD 窗口部件。这里会使用一个读取锁来确保从那些共享数据结构中读取时的安全性，并且会最小化该读取锁上下文中所包含的工作量，同时，只有在该锁释放后才会更新 LCD 窗口部件。

每处理到第 101 个文件时，都会更新常用单词列表窗口部件。这里还会再次用到一个读取锁，并且会用 `set.copy()` 来确保在释放锁的同时不会引用这部分共享数据。

```

def finished(self, completed):
    self.statusLabel.setText("Indexing complete" \
        if completed else "Stopped")
    self.finishedIndexing()

```

当线程一停止或者一完成，就会发射一个 `finished()` 信号，该信号会与这里给出的方法相连接并传递一个 Boolean 来说明操作是否完成了。这里会更新状态标签并调用 `finishedIndexing()` 方法来更新用户界面。

```

def finishedIndexing(self):
    self.walker.wait()
    self.filesIndexedLCD.display(self.fileCount)
    self.wordsIndexedLCD.display(len(self.filenamesForWords))
    self.commonWordsLCD.display(len(self.commonWords))
    self.pathButton.setEnabled(True)

```

完成索引之后就可以调用 `QThread.wait()` 来确保该线程的 `run()` 方法也已经完成了。然后，就可以基于共享数据结构的当前值来更新用户界面了。这里不需要对字典或者数据集进行访问保护，因为 walker 线程并没有运行。

### 用于解锁的上下文管理器

在这一章，我们使用 `try...finally` 块来确保用完锁之后的解锁。Python 2.6 利用新的关键字 `with` 与上下文管理器相结合的方式给出了一种替代方法。有关上下文管理器 (context manager) 的说明可以参阅 <http://www.python.org/dev/peps/pep-0343>；可以肯定的是，通过创建一个带有两个特殊方法的类，就可以创建一个上下文管理器，这两个特殊方法分别是 `_enter_()` 和 `_exit_()`。然后，就像下面这样的代码：

```

try:
    self.lock.lockForRead()
    found = word in self.commonWords
finally:
    self.lock.unlock()

```

利用这一方法，就可以将上述内容写得更为简洁、更为精炼

```

with ReadLocker(self.lock):
    found = word in self.commonWords

```

之所以可以这样做，是因为赋给 with 语法的(以其最为简单的)对象的语义是

```

ContextManager.__enter__()
try:
    # statements, e.g., found = word in self.commonWords
finally:
    ContextManager.__exit__()

```

如下所示的 ReadLocker 上下文管理器类自身也很容易实现，假设它可以传递一个 QReadWriteLock 对象

```

class ReadLocker:
    def __init__(self, lock):
        self.lock = lock
    def __enter__(self):
        self.lock.lockForRead()
    def __exit__(self, type, value, tb):
        self.lock.unlock()

```

如果实际允许，由于使用的是 PyQt 4.1，QReadLocker 和 QWriteLocker 都可以用上下文管理器，所以，使用的是 Python 2.6(或者是在 Python 2.5 版本中使用 from \_\_future\_\_ import with\_statement)的话，就可以不用 try...finally 来保证解锁，并可以像这样来写代码：

```

with QReadLocker(self.lock):
    found = word in self.commonWords

```

本章在 chap19 文件夹下的 pageindexer\_26.pyw 和 walker\_26.py 文件中，都用到了这一方法。

在索引进行的过程中，用户可以与用户界面进行交互而不会感觉到冻结或者性能衰退的迹象。尤其是，他们仍旧可以在查找行编辑框中输入要查找的文本并按下 Enter 键，从而可以用含有已键入单词的文件来填充文件列表窗口部件。如果不止一次地连续按下 Enter 键，就可能会改变文件列表，因为在不同的按键间隔中，可能会有更多的文件得到索引。find() 方法的代码比较长，所以这里会将它分成两个部分来看。

```

def find(self):
    word = unicode(self.findEdit.text())
    if not word:
        self.statusLabel.setText("Enter a word to find in files")
        return
    self.statusLabel.clear()
    self.filesListWidget.clear()
    word = word.lower()
    if " " in word:
        word = word.split()[0]

```

```

try:
    self.lock.lockForRead()
    found = word in self.commonWords
finally:
    self.lock.unlock()
if found:
    self.statusLabel.setText(
        "Common words like '%s' are not indexed" % word)
    return

```

如果用户在查找行编辑框中输入了一个单词，就会清除状态标签和文件列表窗口部件并在常用词集合中寻找该单词。如果可以找到这个单词，就表明这个单词太容易索引到了，所以只会给出一条信息性消息并予以返回。

```

try:
    self.lock.lockForRead()
    files = self.filenamesForWords.get(word, set()).copy()
finally:
    self.lock.unlock()
if not files:
    self.statusLabel.setText(
        "No indexed file contains the word '%s'" % word)
    return
files = [QDir.toNativeSeparators(name) for name in \
         sorted(files, key=unicode.lower)]
self.filesListWidget.addItems(files)
self.statusLabel.setText(
    "%d indexed files contain the word '%s'" % (
        len(files), word))

```

如果用户输入的单词不在常用词集合中，那么它有可能是在索引中。此时在访问 filenamesForWords 默认字典的时候就会使用一个读取锁，并予以复制那些与该单词相匹配的文件集。如果所有文件中都没有这个单词，则该文件集将会为空，不过，在两种情况下，我们所拥有的文件集都会是一个副本，所以在读取锁的上下文之外访问该共享数据也就不会有风险了。如果可以找到匹配的文件，就将它们添加到文件列表窗口部件中，对其进行排序并使用与操作系统平台相对应的路径分隔符来存放它们。

`sorted()` 函数会以排序后的顺序返回它的第一个参数（例如，一个列表或者是一个集合）。可以用一个比较函数来作为第二个参数，但在这里会明确给定一个“键”。这样就可以起到 DSU(Decorate, Sort, Undecorate, 即封装-排序-解封) 排序的作用，也就具有如下相同的排序效果：

```

templist = [(fname.lower(), fname) for fname in files]
templist.sort()
files = [fname for key, fname in templist]

```

这样做会比比较函数更为高效，因为每个项都只是小写一次而不是像在比较函数中的每次比较都需要小写一次。

```

def reject(self):
    if self.walker.isRunning():
        self.walker.stop()
        self.finishedIndexing()
    else:
        self.accept()

```

如果用户按下了 Esc 键，就会调用 `reject()` 方法。如果索引正在进行中，就会对该线程调

用 `stop()`，然后再调用 `finishedIndexing()`；而这个 `finishedIndexing()` 方法调用则会进一步调用 `wait()`。否则，索引就会在上次按下 `Esc` 键或者是在索引完成时停止；无论是哪一种方式，都会调用 `accept()` 来终止应用程序。

```
def closeEvent(self, event=None):
    self.walker.stop()
    self.walker.wait()
```

当应用程序终止的时候，要么会调用 `reject()` 方法中出现的 `accept()`，要么会通过其他手段，比如用户单击了关闭按钮 `X` 后会调用关闭 (`close`) 事件等。在这里，会确保索引已经停止且线程也已经完成，以便可以干净地终止应用程序。

所有的索引工作都已经通过 `walker` 次线程完成了。主线程通过自己的方法（例如，`start()` 和 `stop()`）对次线程进行了控制，并通过 PyQt 的信号 - 槽机制来告知主线程该次线程的当前状态（索引到的文件、索引的完成等）。对于共享数据的访问——例如，当用户问有哪些文件包含有某个特定词时，或者是当数据已经被 `walker` 线程进行了更新时，都会用到读 / 写锁的保护。在接下来的一节中将会看到，`Walker` 线程是如何实现的，如何发出信号的，还有是如何填充所给定的数据结构的。

### 19.3 实现次线程

页面索引 (Page Indexer) 应用程序的次线程的实现，放在文件 `chap19/walker.py` 的 `Walker` 类中。这个类是 `QThread` 的一个子类，会使用 `QMutex` 来把要保护的访问当成是自己的私有数据，也会使用传递给它的 `QReadWriteLock` 来保护和主线程共享的数据。

```
class Walker(QThread):
    COMMON_WORDS_THRESHOLD = 250
    MIN_WORD_LEN = 3
    MAX_WORD_LEN = 25
    INVALID_FIRST_OR_LAST = frozenset("0123456789_")
    STRIPHTML_RE = re.compile(r"<[^>]*?>", re.IGNORECASE|re.MULTILINE)
    ENTITY_RE = re.compile(r"&(\w+?);|\#(\d+?);")
    SPLIT_RE = re.compile(r"\W+", re.IGNORECASE|re.MULTILINE)
```

这个类的开头是一些静态变量，这些静态变量用来控制某个词在被认定为常用词之前，会在多少个文件中出现，还会用来控制一个单词的最大和最小长度值，以及有哪些字符是不可能用于一个单词的开头或者结尾的。这里给出的“strip HTML”正则表达式用来去掉各类 HTML 标签，而实体正则表达式 `entity` 则用来挑出那些要转换的 Unicode 字符实体，而拆分正则表达式 `SPLIT` 则用于将文件中的文本内容拆分成构成各个连续的单词。而更接近真实的应用程序则可能会使用 HTML 解析器，而不是用正则表达式。

```
def __init__(self, lock, parent=None):
    super(Walker, self).__init__(parent)
    self.lock = lock
    self.stopped = False
    self.mutex = QMutex()
    self.path = None
    self.completed = False
```

该应用程序会创建一个 `walker` 线程对象，但并不是一创建完就会启动它。这里的锁与主线程中所使用的 `QReadWriteLock` 相同——`walker` 线程使用它来保护对共享的默认字典

filenamesForWords 和 commonWords 集的所有访问。在类内部的 stopped 变量用来确定该线程是否已经要求停止(通过对 stop() 方法的调用)。mutex 互斥体由 walker 线程自己用来保护对 stopped 变量的访问。这一点很有必要, 因为就是在 run() 方法执行的时候, 恰好有可能会调用了另外一个方法, 比如 stop()。completed 变量用以说明当线程被停止时, 索引是否完成了。

```
def initialize(self, path, filenamesForWords, commonWords):
    self.stopped = False
    self.path = path
    self.filenamesForWords = filenamesForWords
    self.commonWords = commonWords
    self.completed = False
```

这个方法设计用于只有在 QThread.start() 调用之前才会得到调用, 以便将线程设置成开始索引。而在线程正在运行时, 则不应对其予以调用(但如果执意要调用的话, 可以在开头的地方放上一个 if not self.isStopped(): return)。

虽然使用互斥体和锁也不会有什么害处, 但都没有必要使用两者。当调用这个方法时, walker 线程并没有运行, 所以对 stopped 变量赋值不会有什么问题, 而对于出入的字典和集合来说, 则只需要带上对它们的引用即可, 都不会以任何的方式来改变它们。

```
def run(self):
    self.processFiles(self.path)
    self.stop()
    self.emit(SIGNAL("finished(bool)"), self.completed)
```

当调用方调用 start() 时, 该线程会随之调用 run() 方法——就像有些事我们永远不会去做一样。这个方法只有三种语法, 但 processFiles() 会执行很长的时间, 因为它会递归地将路径中的全部 HTML 文件都挨个检索一遍。不过, 这不是什么问题, 因为当前的处理进程会代替正在执行的 walker 线程, 所以用户界面会一直保持响应, 并且主线程依然可以调用 walker 线程的方法来响应 walker 线程的信号, 就像我们在前面章节中看到的那样。最后, run() 方法会发射一个 finished() 信号, 并用 Boolean 标志来说明索引是否已经完成; 如果还没有完成, 用户则必须通过用户界面将其予以停止。

```
def stop(self):
    try:
        self.mutex.lock()
        self.stopped = True
    finally:
        self.mutex.unlock()
```

这个锁会一直持续到获取该块, 并且正是由于有了 try...finally 块, 才可以保证互斥体能够在最后得到解锁。

如果使用的是 Python 2.6 或者是使用了 from \_\_future\_\_ 语句的 Python 2.5, 则可以像下面这样来重写这个方法:

```
def stop(self):
    with QMutexLocker(self.mutex):
        self.stopped = True
```

自 PyQt 4.1 开始, QMutexLocker 类就可以用做上下文管理器。它会将给定的 QMutex 锁用作参数(一直阻塞到获得锁为止), 也会在超出控制流的作用域时解除互斥体的锁定(即使是因为异常而跳出了作用域)。

```

def isStopped(self):
    try:
        self.mutex.lock()
        return self.stopped
    finally:
        self.mutex.unlock()

```

需要注意的是，`return` 语句是在 `try...finally` 块中的。当到达 `return` 时，该方法就会尝试返回一个值，但这就会迫使它进入 `finally` 块，之后，该方法就会返回 `return` 语句的值。

如果正在使用的是 Python 2.6(或者是使用了适当的 `from __future__` 语句的 Python 2.5)，则可能会完全省略这一方法，而在其他一些方法中，则会写成如下这样：

```

if self.isStopped():
    return

```

而我们则可能会写成这样：

```

with QMutexLocker(self.mutex):
    if self.stopped:
        return

```

这些方法不应该有什么大的区别，虽然使用 `QMutexLocker` 似乎是最为简洁和最为明确的方法。

`ProcessFiles()` 方法的代码比较长，所以会把它分成三个部分来看。

```

def processFiles(self, path):
    def unichrFromEntity(match):
        text = match.group(match.lastindex)
        if text.isdigit():
            return unichr(int(text))
        u = htmlentitydefs.name2codepoint.get(text)
        return unichr(u) if u is not None else ""

```

方法一开始是一个嵌套的函数定义。它用来连接 entity 实体正则表达式。这个表达式中有两个相互匹配的群组，但只有二者中的其一可以在任何时间得到匹配。假如正则表达式可以与一个可匹配的对象相匹配，那么该函数只会与后一个相匹配，那就是说，仅仅会与相匹配的群组进行匹配，而如果它全部是数字，则会返回相应代码点的 Unicode 字符。否则，该函数就返回与实体名称相匹配的 Unicode 字符名称，或者，若该名称没有在 `htmlentitydefs.name2codepoint` 字典中，则返回一个空字符串。

```

for root, dirs, files in os.walk(path):
    if self.isStopped():
        return
    for name in [name for name in files \
                if name.endswith((".htm", ".html"))]:
        fname = os.path.join(root, name)
        if self.isStopped():
            return
        words = set()
        fh = None
        try:
            fh = codecs.open(fname, "r", "UTF8", "ignore")
            text = fh.read()
        except (IOError, OSError), e:
            sys.stderr.write("Error: %s\n" % e)

```

```

        continue
    finally:
        if fh is not None:
            fh.close()
    if self.isStopped():
        return
    text = self.STRIPIHTML_RE.sub("", text)
    text = self.ENTITY_RE.sub(unichrFromEntity, text)
    text = text.lower()

```

`os.walk()`方法会从给定的路径开始进行目录树的递归遍历。对于它所找到的每个目录，它都会返回一个根路径的三元组，一个子目录的列表以及目录中的一个文件列表。

这里会对目录中以`.htm`或者`.html`作为后缀的文件进行遍历。`unicode.endswith()`和`str.endswith()`方法要么接受单个字符串，要么接受可匹配的一系列字符串元组。对于每个要匹配的文件来说，都会为其创建一个带有全路径的文件名，还会创建一个局部的空集合来包含之后在文件中找到的那些特定单词。

实际上，本应该要检查这些 HTML 文件所使用的编码，但相反，这里仅假设它们使用的是 UTF-8 Unicode 或者是 ASCII(这是 UTF-8 的一个严格子集)。还会传递一个额外的参数来说明该如何处理那些解码的错误(即本应该忽略它们)。

一旦有了文件的文本后，就可以将它当成是一个大的字符串来进行读取，这样做会剔除掉它的 HTML 标签，把所有的实体都转换成对应的 Unicode 实体，并会将剩下的文本全部写成小写字母。`re.sub()`(即“substitute”)方法会将要处理的文本作为其第二个参数；而它的第一个参数则要么是一个打算替换的匹配项字符串，要么是一个要调用的函数。如果是一个要调用的函数，对于传递给函数的每个相匹配的匹配对象，都会将函数的返回值(这应该是一个字符串)用做替换字符串。

这里有好几个地方会检查`stopped`变量的值是否是`True`，如果调用了`stop()`方法，就会发生这种情况。如果发生了，就无须再进一步进行索引了，而只是简单地返回即可。如果检查的次数太少，用户可能会遇到要求索引终止时与线程的实际停止之间存在延迟的问题。但另一方面，如果有太多的检查投入，也会减慢线程的运行速度。这样一来，检查的频率，以及检查的地点等，这些也都都需要不断地进行尝试，才可以找到合适的结果。

```

for word in self.SPLIT_RE.split(text):
    if self.MIN_WORD_LEN <= len(word) <= \
        self.MAX_WORD_LEN and \
        word[0] not in self.INVALID_FIRST_OR_LAST and \
        word[-1] not in self.INVALID_FIRST_OR_LAST:
        try:
            self.lock.lockForRead()
            new = word not in self.commonWords
        finally:
            self.lock.unlock()
        if new:
            words.add(word)
    if self.isStopped():
        return
    for word in words:
        try:
            self.lock.lockForWrite()
            files = self.filenamesForWords[word]

```

```

        if len(files) > self.COMMON_WORDS_THRESHOLD:
            del self.filenamesForWords[word]
            self.commonWords.add(word)
        else:
            files.add(unicode(fname))
    finally:
        self.lock.unlock()
    self.emit(SIGNAL("indexed(QString)'), fname)
self.completed = True

```

对于文件文本中的每个不太长也不太短且开头不含不可识别字符的单词来说，可以先查看它是否在常用词汇集中，如果不在，那么就将其添加到本地词集中。

一旦将文件中的所有生僻字都集齐到了词集中，就可以开始遍历此集了。新的单词会添加到 `filenamesForWords` 默认字典中。如果该字典的文件名集合对于这个单词来说太大了，就会删除字典中的条目，并将该单词添加到常用词集中；否则，就将该文件名添加到当前单词的字典集中。当然，必须使用一个写锁来确保没有其他的线程（例如，主线程）正在访问该字典或者是正在更新常用词的时候对它们进行访问。

文件索引后，就会发出一个以文件名为参数的 `indexed()` 信号。主线程会连接到这个信号并会在标签中显示出该文件的名称，以便用户可以看到索引的是哪个文件。

一旦 `os.walk()` 循环完成，`completed` 变量就会设置成 `True`，该方法的调用到此结束，并将控制权限返回给调用程序，即 `run()`。最后一个语句也可能永远不会执行到，因为如果用户停止索引操作（通过按下 `Esc` 键，就会造成 `stop()` 的调用，也同时会使得 `stopped` 设置为 `True`，也同时意味着 `isStopped()` 将会返回 `True`），`if isStopped():` 语句的一种情况是造成 `processFiles()` 方法即刻返回。在这种情况下，`completed` 变量（正确地）将会是 `False`。

至此，我们就完成了 `walker` 线程的学习。使用 `with` 语句和上下文管理器而不是使用 `try...finally` 块可以让代码更短、更容易理解，这一点可以从 `pageindexer_26.pyw`、`pageindexer.pyw` 和 `walker.py` 以及 `walker_26.py` 中看出来。`stop()` 方法和 `stopped` 变量对于为主线程服务的次线程来说比较常见，所以 `Walker` 类，虽然运行时稍显特殊，但其结构还是相当普遍的。

## 小结

使用 PyQt 的线程和网络类编写线程服务器相对是较为简单的方法。对于不带 GUI 的服务器来说，或许可以使用 `QCoreApplication` 而不是 `QApplication`，或者根本就不需要使用 PyQt 的任何类，而是完全依赖 Python 的标准库线程和网络类，又或者是直接使用 Twisted<sup>①</sup>。

<sup>①</sup> Twisted 是用 Python 实现的基于事件驱动的网络引擎框架。Twisted 诞生于 2000 年初，当时的网络游戏开发者发现无论使用何种语言，都鲜有可兼顾扩展性及跨平台的网络库，即缺少一个满足可扩展性高、基于事件驱动、跨平台的网络开发框架。Twisted 支持许多常见的传输及应用层协议，包括 TCP、UDP、SSL/TLS、HTTP、IMAP、SSH、IRC 以及 FTP。像 Python 一样，Twisted 也具有“内置电池”（batteries-included）的特点。Twisted 对于其支持的所有协议都带有客户端和服务器实现，同时附带有基于命令行的工具，使得配置和部署产品级的 Twisted 应用程序变得非常方便。Twisted 的官方网站是 <http://twistedmatrix.com/trac/>——译者注。

理论上来看，对于次线程的处理并不是什么难事，但在实践中，对其处理一定要非常小心，以确保那些可由多个线程访问的任意数据都能够使用诸如 QMutex、QReadWriteLock 或者 QSemaphore 的保护机制。在上下文保护机制中，必须确保做的工作量是最少的，以便压缩对其他线程造成的可能阻塞时间。在读取数据时，尤其是数据量不算太大的情况下，先将其进行复制往往是最好的办法，这样可以避免数据访问时超出保护范围的风险。

通过调用次线程的方法来实现主线程和次线程之间的通信，这样的做法非常常见——例如，start() 可以用来启动它们的线程，而 stop() 则可以用来终止它们的线程。次线程可以通过发射与主线程相连接的信号来实现与主线程的通信。主线程与次线程都可以使用由 QMutex、QReadWriteLock 或者 QSemaphore 所保护的共享数据结构——一种常用的方案是，对于共享数据来说，主线程会读取它们，而次线程则会读取和写入它们。各个线程可能需要对它们自己的数据予以访问保护——例如，传递给次线程的 stopped 变量——因为在同一时刻(例如，在同一个次线程中的 run() 和 stop())，可能会有远不止一个的方法是处于活动状态的。

多线程编程比单线程编程来说，线程的编程和维护要难得多，所以找寻是否还有更为简单的可替代方法往往还是很有必要的，比如，对于 QApplication::processEvents() 的调用或者是通过 QProcess 来调用外部进程，都可以用来替代原有的方法。

## 练习题

对页面索引器(Page Indexer)应用程序进行修改，以便它可以使用多个次线程而不是只用一个次线程。通过获取适当的次线程数，该应用程序应当会比单线程版本的应用程序运行得更快一些。虽然在这个练习题中涉及的编程或者修改的代码只有大约 100 行，但却会需要相当的技巧且颇具挑战性。

在练习题答案中所采取的办法是。将 os.walk() 移动到了主线程中，并创建了一个文件名列表。每当该列表中有 1000 个文件后，就会创建一个次线程来处理这些文件。在最后，会再创建一个次线程来处理那些剩下的文件。Walker.initialize() 方法不再是必需的，因为可以将所有的参数都传递给构造函数。而对于 Walker.run() 和 Walker.processFiles() 的更改都很少。大部分的更改工作都必须在 pageindexer.pyw 文件中完成。

setPath() 方法中用来搜集各个文件名，并会创建一些处理它们的次线程。在练习题答案中使用了一个单独的方法来创建这些次线程。因为可能会有很多的次线程会添加到 stopWalkers() 方法中并进而修改 finished()、accept()、reject() 和 finishedIndexing() 方法。由于用户界面中的一些窗口部件可能会因为同时响应多个线程中的信号而被访问到，所以可以利用互斥体来对它们施加访问保护。

请确保那些不再需要用到的线程能够得到及时删除，以便避免因在每次调用 setPath() 的时候都会创建的线程会变得越来越多。

新版本的 walker 模块的代码相比原有版本的代码会有所短小，但在新版本的页面索引器应用程序中的代码则会比原有版本多出 90 行左右的代码。本章练习题的参考答案放在文件 chap19/pageindexer\_ans.pyw 和 chap19/walker\_ans.py 中。

## 这并非结束

至此就到了本书的最后，但这绝非就意味着已经掌握了 Python 或者 PyQt 的所有内容。Python 的标准库非常大，但由于我们仅仅关注于 PyQt 而几乎很少用到其中的万分之一。还有很多其他库也可用做 Python 和 PyQt 的插件和辅助功能，所以在很多时候，编程的时候都是尽可能多地使用已有的组件而不是完全靠我们自己的一己之力来完成所有的工作。查找这些相关扩展，可以先从一个不错的地方开始，即 Python 的程序包索引 (Python Package Index)，<http://pypi.python.org/pypi><sup>①</sup>。而如果要查找有关 Python 和 PyQt 的使用技巧、小窍门和思路，可以查阅 <http://aspn.activestate.com/ASPN/Python/Cookbook> 中的 *Python Cookbook* 一书。

这里对于 PyQt 的涉及内容已经相当宽泛，并且对其最主要的特征都做了详尽的阐释。不过，PyQt 拥有数以百计的类，所以也难以一一涵盖，或者更严格地来说，也难以全部提到它们。例如，PyQt 包括的窗口部件要远多于本书中所用到的那些，包括 QCalendarWidget、QGroupBox、QProgressBar 和 QToolBox 等。还有很多的非窗口部件类，比如 QCompleter(文本补齐)、QFileSystemWatcher(监控文件系统中文件和目录所发生的变化)以及 QSystemTrayIcon(在系统栏上会放一个带有弹出菜单的图标)。PyQt 还支持检索访问和撤销(undo)/重复(redo)框架。此外，PyQt 还有一些与系统平台相关的特性，包括对 ActiveX 的支持以及 Mac OS X 的抽屉。所有这些内容都可以在在线文档中找到。PyQt 还提供了它自己的示例集——那些例子中会有比较感兴趣的领域，因而也就非常值得阅读了。

这本书有望为 Python 和 PyQt 的 GUI 编程奠定坚实基础。这本书中所给出的各个原理和实践练习有望为学习未来 PyQt 文档中给出的新的 PyQt 类和例子变得简单而直接，从而也就能够成功地创建出自己的类。利用 PyQt 编程不仅高效而且令人享受，它可以让我们将精力集中在开发那些看起来漂亮舒服且运行良好的各类应用程序了。

---

<sup>①</sup> 截至今天，该网址所列出的程序包已达 79251 个——译者注。

# 附录 A 安装

- 在 Windows 系统上安装
- 在 Mac OS X 系统上安装
- 在 Linux 和 UNIX 系统上安装

本书中所讲述到的全部工具都可以在网上免费下载。不过，值得注意的是，某些安装包则相当大(Qt 大约有 50 MB, Python 大约有 12 MB, PyQt 大约有 6 MB, 而 SIP 则大约有 0.5 MB<sup>①</sup>)，所以下载它们确实需要有非常高的带宽连接。在这个附录中，既会涉及下载方面的内容，又会涵盖安装方面的内容，包括在 Windows、Mac OS X 和基于 X11 的 UNIX 与诸如 Linux 和 BSD 这样的 UNIX 衍生系统上的安装。

所有的软件包都会带有自己的安装操作说明，它们会比这里给出的内容更新、更全面，因而在安装时最好能够按照那些自带的安装说明进行操作。然而，许多时候，在这个附录中给出的信息也足以进行这些工具的安装并使其正常工作。使用本附录的一种办法是，可以用它来确定需要获得的软件包有哪些(以及应该按照何种顺序下载它们，对于 Windows 用户来说这可能会是个问题)，还有它们的安装顺序(对所有系统平台来说都存在这一问题)。之后，一旦这些工具下载完成，就可以按照官方说明文档依次进行每个工具的安装，同时快速浏览本附录说明，以便能够明确下一步需要做什么，也可以从中学到一些技巧提示，这一方法无论是对 Windows 用户来说，还是对 Mac OS X 用户来说，都很有用。

## A.1 在 Windows 系统上安装

对于 Windows 系统来说，需要安装四个工具：C++ 编译器、Qt C++ 应用程序开发框架、Python 解释器及其绑定库、PyQt4(其中会含有 Windows 二进制包形式的 SIP)。假定使用的是 GPL 版本，在这种情况下，能够与 Qt 一起工作的 C++ 编译器只有 MinGW (Visual C++ 只可以用于商业版的 Qt 和 PyQt，当购买它们后，自然会附带其安装说明)。

就在本书中写作这些内容的时候，一个涵盖所有内容的软件包正在开发中。这个集成包预计将包含 PyQt 的所有模块(除 Qt Designer 模块之外)、QScintilla、翻译与 Qt 设计师支持工具、文档和示例、SQLite 数据库以及对.png、.svg、.gif 和.jpeg 图像格式的支持。这个集成包将会除 Python 自身之外，再不需要安装其他软件，是一个完整、自包含的软件包。不过，这个集成包也将无法支持扩展。如果是第一次学习或者试用 PyQt，用该软件包作为入门起点可能将是最为简单的方法。在有了一定的经验后，就可以再来对那些所需的特定组件进行卸载和安装。该软件包一旦可用，就会出现在网站 <http://riverbankcomputing.com/> 上<sup>②</sup>。在安装好 Python 之后，只需下载并执行这个集成软件包就可以完成所有其他组件的安装了。

<sup>①</sup> SIP 是一种可用来创建“绑定”的工具，允许从 Python 中访问各个 C++ 类。SIP 的下载安装网址见：<https://www.riverbankcomputing.com/software/sip/download>。

<sup>②</sup> 该集成安装包目前已经开发完成，可以在这个网站上直接下载，并且能够支持最新版的 Qt 集成开发框架——译者注。

在接下来的操作说明中，将会基于 Windows XP 家庭版并一一安装这些组件。与其他版本的 Windows 可能会存在一些不同，不过，也不至于会出现使用了这里给出的方法后，各个组件不能正常工作的情况。

安装所需的文件有 `MingW-3.4.2.exe`、`qt-win-opensource-4.2.3-mingw.exe`、`python-2.5.1.msi` 和 `PyQt-gpl-4.2-Py2.5-Qt4.2.3.exe`。这本书中的各个例子都放在文件 `pyqtbook.zip` 中。

要获取的第一项是 PyQt。这是因为，所需的 Python 和 Qt 的版本要取决于 PyQt 的版本。到网站 <https://www.riverbankcomputing.com/>，下载二进制包 `PyQt-gpl-4.2-Py2.5-Qt4.2.3.exe`<sup>①</sup>。文件名中嵌入的版本号可能会与这里给出的有所不同：第一个数字表示的 PyQt 版本，必须至少是 4.2 以上才能最有效地利用本书；第二个数字表示必须安装的 Python 的版本；第三个数字表示 Qt 的版本号——必须严格下载它给定的版本。

现在来获取 Qt。到网站 <http://www.trolltech.com/developer/downloads/qt/index>，然后单击 `Qt/Windows Open Source Edition` 链接；在页面的底部，下载 `qt-win-opensource-4.2.3-mingw.exe`<sup>②</sup>。这里的版本号一定要与 PyQt 程序包名称中给出的数字准确匹配，要是这样的话，比如，前面下载的是 `PyQt-gpl-4.3-Py2.5-Qt4.3.1.exe`，这里就需要下载 `qt-win-opensource-4.3.1-mingw.exe`。

也可以从 Trolltech 的 Web 站点上获得 MinGW C++ 编译器，只不过是用一个完全不同的 URL 而已。登录 <ftp://ftp.trolltech.com/misc/>，下载 `MingW-3.4.2.exe`（也可以跳过这一步，而让 Qt 安装程序下载该编译器，只是如果自己下载的话，就可以随时使用该安装包，这样在其他机器上安装或者是当自己的 Windows 安装设备运行出错时需要恢复时，都会比较方便些）<sup>③</sup>。

现在轮到 Python 了。登录 <http://www.python.org/downloads/>，下载一个 Windows 安装程序（在页面顶部的那些不会包含源代码；这还是不错的，因为只有在需要修改 Python 自己时才会需要用到源代码）。可能会有多个 Windows 安装程序；如果有能够与计算机处理器相匹配的 Python 版本，可以下载那些与硬件相关的安装程序，如 AMD64 或 Itanium；否则，只需单击第一个即可——例如，“`Python 2.5.1 Windows installer`”。先把这个安装程序保存到磁盘上；这是一个 Microsoft Installer（微软安装程序）文件——例如，`python-2.5.1.msi`。值得注意的是，版本号中的前两个部分必须与 PyQt 版本号中的那部分数字一致；所以，对于 `PyQt-gpl-4.2-Py2.5-Qt4.2.3.exe` 来说，Python 2.5 的任何版本都是可以接受的，如 Python 2.5，或 Python 2.5.1。

如果打算运行本书中的示例程序，或是想看看练习题参考答案的运行效果，可以解压 <http://www.qtrac.eu/pyqtbook.html> 中的 `pyqtbook.zip` 文件。

---

① 最新的 PyQt4 版本下载地址为 <https://www.riverbankcomputing.com/software/pyqt/download>；而最新的 PyQt5 版本下载地址为 <https://www.riverbankcomputing.com/software/pyqt/download5>；或者，之前老版本的 PyQt 可以在这里下载 <http://sourceforge.net/projects/pyqt/files/PyQt4/>。考虑到译文应尽可能尊重原书内容，这里仅修正了那些已经失效的链接——译者注。

② 最新的 Qt 各个版本下载地址为 <http://download.qt.io/archive/qt/>——译者注。

③ 目前，Trolltech 的 Ftp 服务器已经关闭，故而无法在此下载 MinGW。不过，如果确实需要下载的话，可以到这里 <http://sourceforge.net/projects/mingw/>，根据提示下载不同版本的 MinGW 编译器。实际上，在安装 Qt 应用程序开发框架时，安装程序会给出是否安装相应版本 MinGW 编译器的选项——译者注。

现在，所有的组件都已经准备齐全了，可以开始安装了。安装的顺序很重要，与下载的顺序有所不同(需要先下载 PyQt，以确保可以获得正确版本的 Python 和 Qt；但安装时，必须先从 C++ 编译器开始，而最后再来安装 PyQt)。假设这里要安装的各个版本就是上面提到的那些，不过，显然，不管下载的是哪个版本，都要清楚知道它们的版本号，以便可以做出相应的调整。

如果没有下载 MinGW 安装程序，要么是因为之前已经安装过该编译器，要么是因为想要让 Qt 安装程序自动下载或者自动安装该程序，所以就可以跳到下一步了。否则，可以运行 MinGW 安装程序(例如，双击 MinGW-3.4.2.exe)，再按照安装程序的说明即可。该安装程序首先弹出的效果如图 A.1 的左图所示。唯一需要决定的是 MinGW 的安装位置。这里假设接受其默认安装地址，即 c:\MinGW；当然，也可以将其放到任何位置。如果没有使用默认位置，那么，就要记下其路径，因为当安装 Qt 时将需要用到此路径。

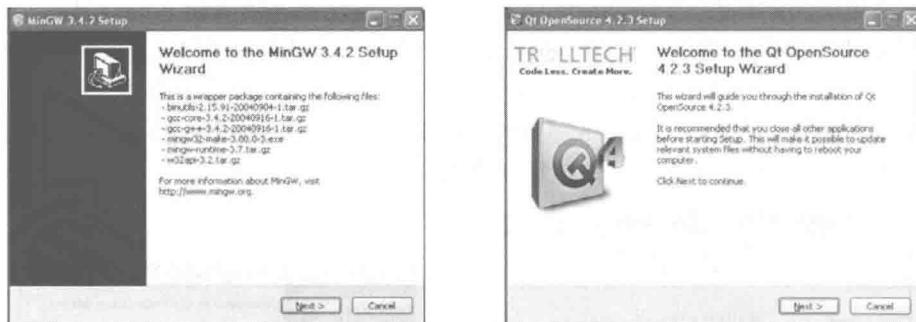


图 A.1 Windows 系统上的 MinGW 和 Qt 安装程序

若要安装 Qt，可以通过双击 qt-win-opensource-4.2.3-mingw.exe(或者所下载的那个版本)来启动其安装程序。安装程序的第一页如图 A.1 右图所示。安装操作的说明比较容易上手，再次说明的是，仍旧假设接受默认目录(比如，c:\Qt\4.2.3)。当出现“MinGW Installation”提示时，如果之前把 MinGW 放在了默认位置，那么“Previously installed MinGW”的路径就应该是正确的。如果不正确，或者之前如果是在非标准位置安装的 MinGW，那么就必须键入该路径，或者使用浏览按钮“(…)"找到它。如果还没有安装过 MinGW，那么就可以选中复选框“Download and install minimal MinGW”，以便让 Qt 安装程序下载并安装它。

遗憾的是，GPL 版 Qt 安装程序没有将 Qt 添加到 Windows 的系统环境变量 path 中；这就意味着，需要依赖诸如 QtCore4.dll、QtGui4.dll、QtXml4.dll 等 Qt DLL 的应用程序，或者需要依赖诸如 mingwm10.dll 这样的 MinGW DLL 应用程序，运行时就无法找到它们。由于 PyQt 应用程序会依赖这些库，所以就必须得手动把它们添加到 path 中，以便能够在双击一个 PyQt.pyw 应用程序时可以使其正常工作。如果 path 中没有这条路径，任何 PyQt 应用程序都将无法正常工作，而会总是弹出错误消息对话框，比如，“pythonw.exe-Unable To Locate Component”，意思就是说，它无法找到 mingwm10.dll。

单击开始(Start)→控制面板(Control Panel)，然后再单击系统(System)，就会弹出系统属性对话框(System Properties)。单击高级选项卡(Advanced)，然后单击环境变量按钮(Environment Variables)(大致在该对话框的底部)。单击“系统变量 System variables”中的路径变量 Path(大约在对话框靠下一半的位置处)，然后再单击编辑(Edit)按钮。

如图 A.2 所示的编辑系统变量对话框 Edit System Variable 中会有 Windows 路径 Path 这个

变量。千万要小心，不要把已有的路径给删除了！如果不小心误删了，可以单击取消(Cancel)按钮，然后试着再次编辑路径。按下小键盘上的 End 键可以不选中路径内容而直接将文本光标放到行编辑上的最右边，然后添加文本";C:\Qt\4.2.3\bin"。这里在最前面给出的分号是必不可少的；很明显，版本号要使用实际安装的 Qt 版本，如果它与这里所给出的版本号不同的话。这一路径可用于所有的 Qt DLL 和 MinGW DLL(因为 Qt 安装程序会将其复制到 Qt 的 bin 目录中)。

现在为安装 Python 就准备好了。双击 python-2.5.1.msi 或者其他任何下载过.msi 文件，启动 Python 安装程序。安装程序的第一屏如图 A.3 中的左图所示。安装程序的使用较为简单；如果不使用默认的 C:\Python25 路径的话，这里唯一需要键入的信息就是 Python 的路径。如果要使用非标准路径，就要一直记着它，因为将来还需要在 PyQt 安装程序中键入它的内容。如果担心磁盘空间不足，可以不用安装测试套件或工具程序的脚本，但这里会假设安装了所有全部的组件，包括 Tcl/Tk。一旦 Python 安装完毕，安装程序就可能会要求重新启动计算机——在安装 PyQt 之前，还是应该重新启动一下的。

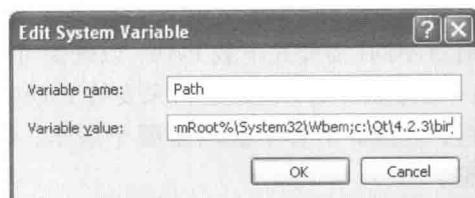


图 A.2 Windows 的系统环境变量 path 的设置

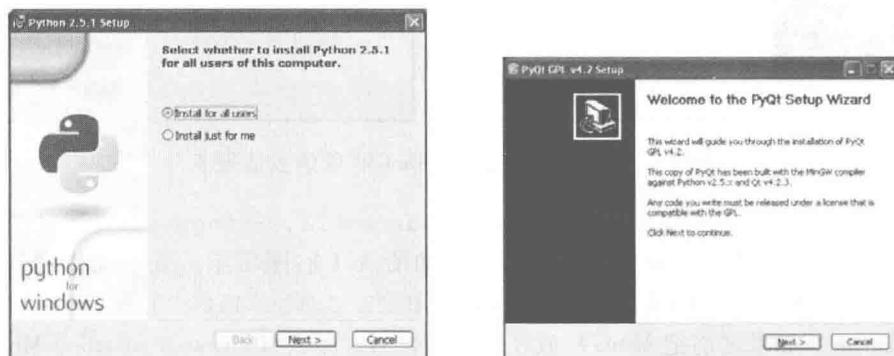


图 A.3 Windows 系统上 Python 和 PyQt 的安装程序

现在可以开始安装 PyQt4 了。双击 PyQt-gpl-4.2-Py2.5-Qt4.2.3.exe 文件；或者是双击所下载的那个 PyQt 的任意版本，都可以启动 PyQt 安装程序。安装程序的第一屏如图 A.3 中的右图所示。如果 Python 是在非标准位置安装的，就必须在选择安装位置页面 Choose Install Location 中输入其正确的安装位置——PyQt 将安装成 Python 的扩展，因此它的库会放在 Python 的目录内(就是由于这个原因，如果想要卸载 Python，就应该先卸载 PyQt，然后再卸载 Python)。

PyQt 是完成所有安装并让它们正常工作的最后一个必须安装的工具。要想进行测试，可以单击开始(Start)→所有程序(All Programs)→PyQt GPL v4.2→Examples and Demos)。这会启动 PyQt 版的标准 Qt 演示程序。在这个应用程序中可以运行很多演示应用程序，它们都是用 PyQt 完成的。这些演示程序和许多其他 PyQt 例子的源代码，通常都会安装在 C:\Program Files\ PyQt4\examples 中。

如果下载了本书的例子，或许可能会在 C 盘将 pyqtbook.zip 解压缩到 C:\pyqt 目录，以获取书中所有的例子和练习题的参考答案，这些内容都已经按章进行了分类。在 C:\pyqt

目录中将可以找到 `mkpyqt.py` 和 `makepyqt.pyw`；这些工具在本书第 7 章做过相关解释。如果打算在阅读本书前，先试试运行一下书中的示例，则要确保先运行一下 `makepyqt.pyw` [在运行 `makepyqt.pyw` 时，要将其路径设置到 `C:\ PyQt`，选中它的递归(Recurse)复选框，然后再单击生成(Build)按钮。这样，所有例子才能够为运行做好准备]。

这就完成了 Windows 的安装，已经足以用来进行 PyQt GUI 的编程了。但如果还想写一些控制台应用程序，或者偶尔想在控制台中运行一些 PyQt 应用程序（在调试时这样做会很有用），再做几个步骤就可以使这一点变得更为方便些。

单击开始(Start)→所有程序(All Programs)→附件(Accessories)→Windows 浏览器(Windows Explorer)。一旦 Windows 浏览器(Windows Explorer)开始运行，就可以找到目录 `My Computer\Local Disk (C:)\Documents and Settings`，然后再找到含有你的用户名的目录，在那里，找到开始\应用程序\附件(即 Start Menu\Programs\Accessories)。复制并粘贴控制台(Console)(或者是 MS-DOS 提示符)的快捷方式，并把新粘贴的快捷方式重命名为"Console (PyQt)"。在这个新的 Console (PyQt) 上右键单击，编辑其属性。在常规(General)页上，将其目标(Target)更改到 `cmd.exe/k C:\ PyQt\pyqt.bat`。现在，如果想使用一个类 PyQt 的控制台时，就可以单击开始(Start)→所有程序(All Programs)→附件(Accessories)→Console(Python)，之后出现的控制台就会自动运行 `C:\ PyQt\pyqt.bat`。这个批处理文件只包含了两行代码：

```
set QMAKESPEC=win32-g++  
path=C:\ PyQt;C:\ MinGW\bin;c:\ Python25;c:\ Python25\lib\idlelib;%path%
```

或许还想编辑这个文件（可以使用纯文本编辑器）来添加含有 `cd` 命令的第三条行代码——例如，`cd C:\ PyQt`——以便让控制台可以从更为方便的目录中启动。如果 MinGW 或者 Python 是在非标准位置安装的，那么无论如何都需要编辑此文件，以将它们放置到正确的路径中。

现在就可以在自己的 Windows 计算机上编写并运行 PyQt 应用程序了——这些代码在 Mac OS X 和 Linux 上将无须更改而可以直接运行！

## A.2 在 Mac OS X 系统上安装

要在 Mac OS X 上安装 PyQt，就必须要先安装过 Xcode 工具。这是因为，安装 PyQt 时需要有一个编译器和一个生成工具。Xcode 是一个非常大的安装包，通常是随计算机一起提供的，会单独放在那个用于开发人员的 CD 上；也可从网站 <http://developer.apple.com/tools/xcode> 在线获得它。下面的安装说明将会假定已经安装过 Xcode 了<sup>①</sup>。

尽管 Mac 通常都会预装某一版本的 Python，但它可能是一个早前的版本，而在那里则会建议安装一个最新版的 PyQt。要检查 Python 的版本，可以启动终端，并键入 `python -V`；如果显示的是“Python 2.5”，或者是一个更高的版本号，那么就没有必要再去安装新版本的 Python 了。

安装 PyQt 所需用到的文件是 `qt-mac-opensource-4.2.3.dmg`、`python-2.5.1-macosx.dmg`（除非已经安装过 Python 2.5 或者更高的版本）、`sip-4.6.tar.gz` 以及 `PyQt-mac`

<sup>①</sup> 中文版的 Xcode 见 <https://developer.apple.com/xcode/cn/>，英文版见 <https://developer.apple.com/xcode/download/>——译者注。

gpl-4.2.tar.gz。这本书的各个例子会放在文件 `pyqtbook.tar.gz` 中。

先从获取 Qt 开始。登录 <http://download.qt.io/archive/qt/4.2/> 并选择相应版本的 Mac Open Source Edition 的链接，下载 `qt-mac-opensource-4.2.3.dmg`。更高的版本号，比如 4.3.1，应该会更好些<sup>①</sup>。

如果需要安装最新版的 Python，可以登录 <http://www.python.org/downloads> 并下载 Macintosh OS X 版本的 Python 2.5.1——例如，`python-2.5.1-macosx.dmg`。而 2.x 系列的后续版本，如 2.5.2 或 2.6.0，应该也不会有什么问题，假定它们都是发行版（而不是 alpha、beta 或待发布版）。

最后两个必须应有的工具分别是 SIP 和 PyQt。登录 <http://www.riverbankcomputing.co.uk/sip/download.php> 并下载源文件包 `sip-4.6.tar.gz`，然后再登录 <http://www.riverbankcomputing.co.uk/pyqt/download.php> 并下载源文件包 `PyQt-mac-gpl-4.2.tar.gz`。再次说明的是，对于 PyQt 的版本号可能会更高——例如，4.3——都是可行的<sup>②</sup>。

如果打算运行书中所给出的示例程序或者是查看一下练习题的参考答案，可以从网站 <http://www.qtrac.eu/pyqtbook.html> 中下载文件 `pyqtbook.tar.gz` 并解压缩，以获得相应的内容。

现在，所需的全部工具就已经都准备好了，并且也假设安装过 Xcode，所以就到了执行 PyQt 安装程序的时候了。首先必须要安装的是 Qt 和 Python，然后是 SIP，最后才是 PyQt。这里将会假设这些版本都是之前提到的那些版本，但很明显，不管使用的是哪一个下载版本，都要在安装过程中做一些相应的调整。假定所有下载的这些文件都放在了桌面上，并且也知道管理员密码（这通常就是自己的密码）。

首先双击 `qt-mac-opensource-4.2.3.dmg` 来安装 Qt，或者是双击下载的那个任意版本的 Qt 安装包，然后按照说明进行安装即可。安装程序的第一屏如图 A.4 所示。假设接受所有的默认设置并将软件安装到了标准位置上。旧的 Qt 版本会使用未经优化的生成工具，而这就意味着，安装程序会需要相当长的安装时间。而最新的版本则已经使用了优化过的生成工具，从而使安装工作要快得多。

Qt 一旦安装成功，就该安装 Python 了，如果需要安装的话。双击 `python-2.5.1-macosx.dmg` 或者所下载的那个软件包。安装程序的第一屏如图 A.5 所示。这时可能会弹出一个带有 `MacPython.mpkg` 文件的新窗口——只需双击它来启动安装程序并按照说明进行操作即可。正如安装 Qt 时所做的那样，假设也会接受各项默认设置并将其安装到标准位置上。如果已经有一个或者多个 Python 版本，就会发现，它们与之前的那些版本并没有什么关系，也就只是在 `/usr/local/bin` 后面添加了两个新的 Python 可执行文件名而已，这些名字中也会包含它们的版本号——例如，`python2.5` 和 `pythonw2.5`。第一个可执行文件会用在终端

---

① Trolltech 的下载服务器已经关闭，最新的 Qt 各个版本下载地址为 <http://download.qt.io/archive/qt/>——译者注。  
② 最新的 SIP 文件的下载地址为 <https://www.riverbankcomputing.com/software/sip/download>；最新的 PyQt4 版本下载地址为 <https://www.riverbankcomputing.com/software/pyqt/download>；而最新的 PyQt5 版本下载地址为 <https://www.riverbankcomputing.com/software/pyqt/download5>；或者，之前老版本的 PyQt 可以在这里下载到 <http://sourceforge.net/projects/pyqt/files/PyQt4/>。考虑到整本书中的译述文字能够最大程度上尊重原书的行文内容，这里仅会修正那些已经失效的链接，而不会对所有详细内容进行重新撰写或描述——译者注。

(Terminal)窗口中，而第二个则会用来启动 GUI 应用程序并避免在应用程序之后再弹出一个不必要的终端(Terminal)。

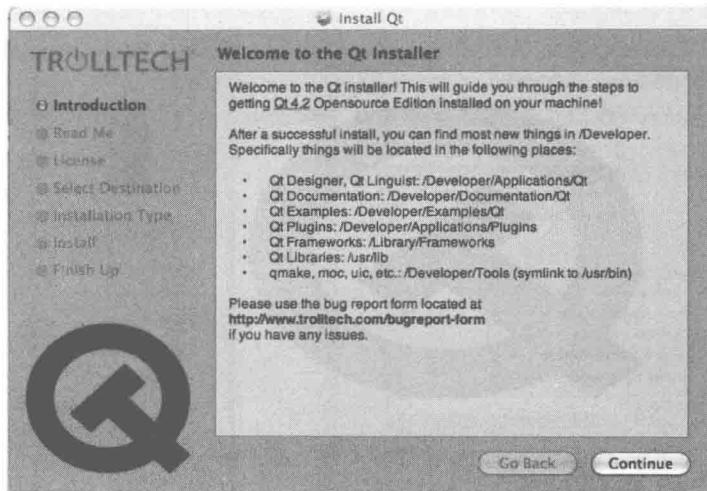


图 A.4 在 Mac OS X 系统上安装 Qt

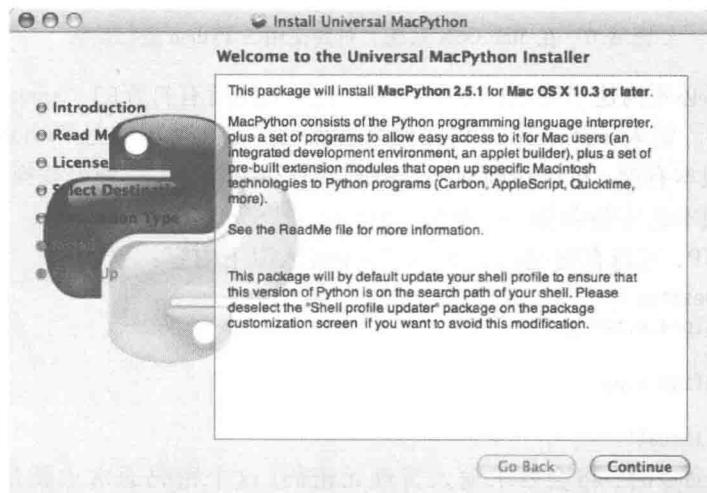


图 A.5 在 Mac OS X 系统上安装 Python

这一安装过程应当让 Python 只安装默认的版本。要选中、关闭任何已有的终端(Terminal)窗口，然后再启动一个新的终端窗口，可以键入 `python -V`。如果启动的这个版本并不是默认安装的那个版本，将会需要手动更改这些设置。关闭终端，然后进入 Finder 中，再到 Applications→MacPython 2.5，就可以启动 Python 的启动程序了。打开首选项对话框(如图 A.6 所示)，并对用于文件类型组合框中的 Settings 的每一项——比如用于 Python Scripts、Python GUI Scripts 和 Python Bytecode Documents——将它们的版本都改成当前这个 Python 版本。对于 Python Scripts 和各个 Python Bytecode Documents 项目来说，可以改成 `/usr/local/bin/python2.5`，而对于 Python GUI Scripts 项目来说，可以改成 `/usr/local/bin/pythonw2.5`(注意可执行文件名称中的“w”)。或许这些值在下拉列表中是不可用的，

在这种情况下，就必须手动键入这些值。对于每个条目来说，还要确保没有选中 Run in a Terminal 复选框。

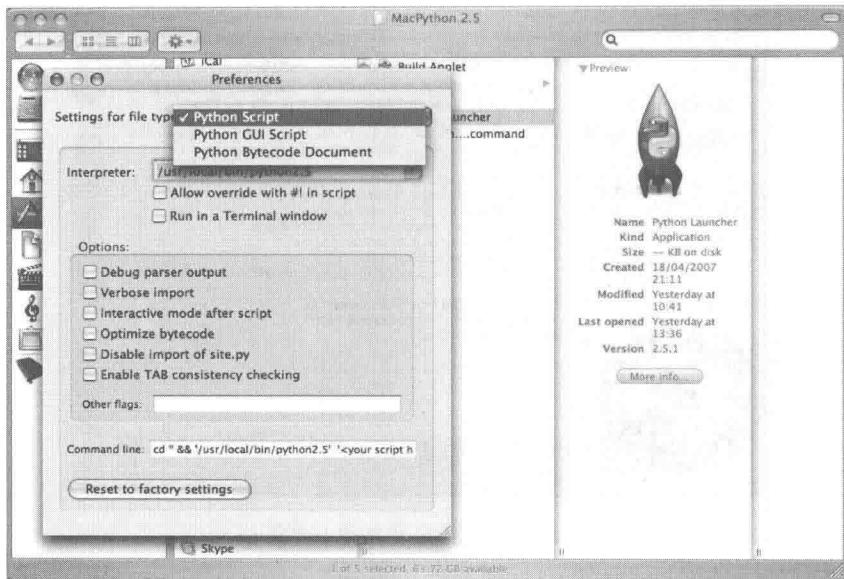


图 A.6 在 Mac OS X 系统上对要使用的 Python 进行设置

SIP 和 PyQt 都必须构建于终端(Terminal)上。关闭所有打开的 Terminal，然后再重新启动一个新的终端。键入 `python -V` 可以用来确保当前正在使用的是 Python 的正确版本。如果 Python 的当前版本有误，可以在前面一段中找到解决方法；另一种可替换的方法是，输入打算使用的 Python 版本的完整名称——例如，`python2.5 configure.py`。

必须先构建 SIP，可以在终端(Terminal)中键入以下内容：

```
cd $HOME/Desktop
tar xvfz sip-4.6.tar.gz
cd sip-4.6
python configure.py
make
sudo make install
```

当最后执行 `sudo` 命令时，将会要求输入管理元密码(这个密码通常也就是你自己的密码)。现在，可以开始安装 PyQt 了。

```
cd $HOME/Desktop
tar xvfz PyQt-mac-gpl-4.2.tar.gz
cd PyQt-mac-gpl-4.2
python configure.py
make
sudo make install
```

再次说明一下，当执行 `sudo` 命令时，将会提示输入密码。构建 PyQt 会花很长的时间，所以还是需要些耐心的。

借助 Qt Assistant 可以得到 Qt 文档，而 Qt Assistant 可以从 Finder 中运行。PyQt 文档是在 `$HOME/Desktop/PyQt-mac-gpl-4.2/doc` 目录中的，格式是 HTML 格式。将其移动到适当的永久位置并在自己的浏览器中添加一个合适的书签，这样做还是非常值得的。它还涵盖了大量例子；至少一点，对于看看 PyQt 的例子并运行它的演示程序还是很有用的(例如，可以

把目录更改为 \$ HOME/Desktop/PyQt-mac-gpl-4.2/examples/tools/qtdemo，然后就可以运行 ./qtdemo.py。)

如果已经下载了本书中的例子，可能希望将 pyqtbook.tar.gz 文件解压缩到 \$ HOME 中，就可以得到一个涵盖书中所有例子和全部练习题参考答案的 \$ HOME/pyqt 目录，这些内容都已经按章分类了。mkpyqt.py 和 makepyqt.pyw 都可以在 \$ HOME/pyqt 目录中找到；或许还希望把这些内容移动(或软链接)到 \$ PATH 变量中——例如，\$ HOME/bin 中——以便可以更为方便地使用它们。其中的一些例子会依赖于 Qt Designer 的.ui 文件或者依赖于.qrc 资源文件。有关如何将这些内容转换成 Python 模块的方法，可以参阅本书的第 7 章，不过，这里只是执行一下转换，可能会比较方便：

```
cd $HOME/pyqt  
./mkpyqt.py -r
```

这样就可以将 pyqt 目录及其子目录中找到的全部.ui 和.qrc 文件进行转换<sup>①</sup>。如果更喜欢使用 GUI 中的 makepyqt.pyw 工具，则可能需要单击它的 More→Tool paths 选项并将该路径设置为 pyuic4。或许还有必要把该路径设置用到其他工具上。

至此，就完成了 Mac OS X 上的安装。如果之前解压缩了包含有例子的文件，则可以转到桌面并单击 pyqt 目录，然后再回到 chap12 目录中，单击 multipedes.pyw 就可以看到一个图形应用程序。如果弹出了一个不需要的终端(Terminal)窗口，就可以右键单击 multipedes.pyw，然后单击信息(Info)对话框；把 Open with 的 Python Launcher 设置成 Python 的正确版本，并应用一下这个更改，以便可以作用于所有以.pyw 为后缀的文件上。

现在就可以在 Mac OS X 设备上编写并运行 PyQt 应用程序了——这些代码在 Windows 和 Linux 上也一样可以直接运行！

### A.3 在 Linux 和 UNIX 系统上安装

如果正在运行的是 Kubuntu(7.04 Fiesty Fawn 或是后续版本)，就表明已安装好了 PyQt4！所以，只需要安装本书的例子、文档包 python-doc 以及 python-qt4-doc 就可以了。

对于 Linux 和其他没有提前安装 PyQt4 的大多数 UNIX 系统来说，有四个工具需要安装：Qt C++ 应用程序开发框架、Python 解释器及其附带的库、SIP 绑定工具和 PyQt4 自身。要为正在使用的 Linux 或 UNIX 发行版备考一切事情并使其能够运行最为简便的方法就是，使用标准安装包来安装这些工具。

对于 ArchLinux、Debian、Fedora、Gentoo、Kubuntu、Pardus、Ubuntu 以及许多其他系统来说，那些必要的组件都可以以程序包的形式获得。可以利用操作系统所使用的诸如 Adept、Pirut、apt-get、yum 或者其他程序包管理程序进行安装。对于 PyQt4 来说，程序包通常会称为 pyqt4 或 pyqt4-dev-tools。PyQt4 的文档程序包则通常会称为 pyqt4-doc 或 python-qt4-doc 或 PyQt4-examples。Python 的文档程序包通常都位于 Python-doc 或是 python-docs 的程序包中。IDLE 通常可以在称为 idle 或是 python-tools 中的程序包中分别获得。如果想要拥有一个更为强大的 IDE，本身就是由 PyQt 写成的 Eric4 则可以在很多的发行版本中都能够找到。程序包管理程序应当能够找出组件之间的依赖关系，但如果不能实现这一点的

<sup>①</sup> 如果 mkpyqt.py 不能正常运行，则需要编辑 mkpyqt.py 文件并至少要将该路径硬编码添加到 pyuic4 中。

话，可能就需要再来安装 Python，甚至 Qt 和 g++ 编译器了。Qt 设计师可视化设计工具和翻译支持工具通常会分成不同的程序包——例如，它们分别在 qt4-designer 和 qt4-dev-tools 的程序包中。

如果足够幸运，可以使用标准程序包来执行安装工作，一旦真的要这么弄，那就意味着已经为编写 PyQt 程序做好了准备，那么就可以直接跳到安装本书例子的那一步。

对于那些使用老版本的用户来说，以及对于那些没有得到合适程序包的用户和那些只在程序包中获取了部分可用组件的用户来说，如果想手动获得最新的版本，再来构建、安装它们，其实也是相当简单的。然而，如果打算利用源代码进行构建，就需要做两个假设——第一，假设已经安装有诸如 make 这样的 C++ 编译器和工具链，并且是可以使用的；第二，假设所安装是以 root 权限安装它们的（可以使用 su 或者 sudo），或已经知道该如何使用 configure 的--prefix 选项来进行本地化安装。

安装所需的文件是 qt-x11-opensource-src-4.2.3.tar.gz、Python-2.5.1.tgz、sip-4.6.tar.gz 和 PyQt-x11-gpl-4.2.tar.gz。这本书的例子放在了文件 pyqtbook.tar.gz 中。

先来获取 Qt。登录 <http://www.trolltech.com/developer/downloads/qt/index> 并单击 Qt/X11 的 Open Source Edition 链接，在页面的底部附近可以找到并下载 qt-x11-opensource-src-4.2.3.tar.gz。对于更高的后续版本，比如说 4.3.1 也应该是可以这样做的<sup>①</sup>。

现在该获取 Python 了。登录 <http://www.python.org/download>，然后单击当前产品版本的链接，就可以下载其他平台的开源版本之一了——例如，Python-2.5.1.tgz 或 Python-2.5.1.tar.bz2。假定获得的是.tgz 版本——后续的 2.x 系列版本，诸如 2.5.2 或 2.6.0，应该都不会有什么问题，只要给定它们的产品发行版本号就可以了（而不是 alpha、beta 或是待发布的版本）。

最后两个必须获得的工具分别是 SIP 和 PyQt。登录 <http://www.riverbankcomputing.co.uk/sip/download.php> 并下载源程序包 sip-4.6.tar.gz，然后转到 <http://www.riverbankcomputing.co.uk/pyqt/download.php> 并下载源程序包 PyQt-x11-gpl-4.2.tar.gz。再次说明一下，这些版本号可能会更高——例如，4.3——对于 PyQt 来说，都是可以的。

如果打算运行书中所给出的示例或看看练习题的参考答案，可以从网站 <http://www.qtrac.eu/pyqtbook.html> 中获得的文件 pyqtbook.tar.gz 解压缩，就可以得到相关的内容了。

现在所有东西都已备齐了，是时候执行 PyQt 安装程序了。Qt 和 Python 必须要先安装，然后是 SIP，最后才是 PyQt 自己。这里将会假设这些版本就是之前提到的那些版本，但很明显，不管使用的是哪个下载版本，都要在安装过程中进行相应的调整。假定下载的各个压缩包都位于 \$HOME/packages 目录中，要么可以像超级用户那样利用 su 来完成后续的所有事情，要么也可以像超级用户利用 sudo 那样来完成每个组件的安装。

首先需要构建 Qt。如果使用的是 sudo 的话，最后一行代码就应该改成 sudo make install。

---

<sup>①</sup> 最新的 Qt 各个版本下载地址为 <http://download.qt.io/archive/qt/>——译者注。

```
cd $HOME/packages  
tar xvfz qt-x11-opensource-src-4.2.3.tar.gz  
cd qt-x11-opensource-src-4.2.3  
.configure -fast -qt-sql-sqlite -no-qt3support  
make  
make install
```

-qt-sql-sqlite 选项将生成 SQLite 进程间数据库；这会在第 15 章中用到，但如果想省略的话也是可以的。-fast 和 -no-qt3support 选项应该可以稍微减少其生成时间，但将两者都忽略也并没有什么不妥。如果想要看看还有其他什么选项是可用的，包括安装的那些数据库驱动程序，可以运行 ./configure -help。构建 Qt 会花很长的时间（根据处理器的不同，时间可以从半小时到三个多小时不等），因为它有超过 60 万行的 C++ 代码。

Python 和 SIP 的代码长度差不多。接下来应该构建 Python，如果正在使用的是 sudo 的话，可以再来调用代码 sudo make install（从现在起，会默认使用 sudo）。

```
cd $HOME/packages  
tar xvfz Python-2.5.1.tgz  
cd Python-2.5.1  
.configure  
make  
make install
```

这会比创建 Qt 快得多。一旦完成，就可以执行最后两步了，即创建 SIP 和 PyQt，这两步的实现方式会比创建 Qt 和 Python 复杂一点。

```
cd $HOME/packages  
tar xvfz sip-4.6.tar.gz  
cd sip-4.6  
python configure.py  
make  
make install
```

这里假设（正确版本的）Python 已经在 \$PATH 路径中了。如果事实并非如此（即因为已经安装了两个或多个版本的 Python），那就得为 Python 提供完整的路径来使其可执行——例如，\$HOME/opt/python25/bin/python configure.py。一旦安装了 SIP，就可以开始安装 PyQt 了。

```
cd $HOME/packages  
tar xvfz PyQt-x11-gpl-4.2.tar.gz  
cd PyQt-x11-gpl-4.2  
python configure.py  
make  
make install
```

与 SIP 的安装相似，这里会假设（正确版本的）Python 已经在 \$PATH 路径中了。再次说明一下，如果事实并非如此，那就应在 configure.py 运行时，给 Python 提供完整的路径来使其可执行。这一步会花掉很长的时间（但比构建 Qt 的时间要短）。

Qt、Python 和 PyQt 的文档都会以 HTML 的形式提供。将该文档的路径修改某个永久位置并在浏览器中添加一个适当的书签还是很值得的。这三者的文件中也都涵盖了大量的示例；至少对于看看 PyQt 的例子和运行演示来说，这样做还是很有用的（例如，将目录更改为 \$HOME/PyQt-X11-gpl-4.2/examples/tools/qtdemo，就可以运行 ./qtdemo.py 了）。

如果已经下载了本书中的例子，可能会想把 pyqtbook.tar.gz 解压缩到 \$HOME 中，可以获得一个 \$HOME/pyqt 目录，其中会涵盖书中所有的例子、练习题参考答案，这些内

容都是按章分类的。`mkpyqt.py` 和 `makepyqt.pyw` 都在 `$HOME/pyqt` 目录中；可能会想把这些目录移动（或软链接）到 `$PATH` 中——例如，`$HOME/bin`——可以更为方便地来使用它们。一些例子会依赖于 Qt 设计师的 `.ui` 文件或是 `.qrc` 的资源文件。如何将它们转换成 Python 模块的方法可以参阅本书的第 7 章，但使用以下代码会使转换的执行更为方便<sup>①</sup>：

```
cd $HOME/pyqt  
./mkpyqt.py -r
```

至此，就完成了在基于 X11 形式的 UNIX 和类 UNIX 衍生系统上的安装，包括 Linux 和 BSD。现在就可以在 UNIX 和类 UNIX 衍生平台上编写代码并运行 PyQt 应用程序了——这些代码在 Windows 和 Linux 上也是一样可以直接运行的！

---

<sup>①</sup> 如果 `mkpyqt.py` 不能正常运行，则需要编辑 `mkpyqt.py` 文件并至少要将路径硬编码添加到 `pyuic4` 中。

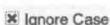
## 附录 B PyQt 的部分窗口部件

这里所给出的截屏图都是在 Linux 系统下使用 KDE 截取的，以提供视觉效果上的一致性。而对于书中的主体内容，各副截屏图分别用于 Windows、Linux 和 Mac OS X，一般会根据不同的章节而有所不同。



QCalendarWidget

这个窗口部件可以作为一个显示窗口部件，尽管设计之初是作为输入窗口部件的，通过它用户可以选择某个特定的日期。这个窗口部件的显示可做高度设置；例如，周数字可以也可以不显示，日期的名字可以表示为单一的字母，或者用简写或者全称的形式表示，也可以设置选用的颜色和字体，所以也就可以设置哪天作为本周的第一天。还可以设置最小和最大允许日期。对 QDateTimeEdit 或者 QDateTimeEdit 调用 setCalendarPopup (True) 将会使得它们的微调按钮被箭头按钮所代替。如果用户点击箭头按钮，就会弹出一个 QCalendarWidget



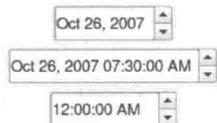
QCheckBox

复选框可向用户表示一个简单的 yes/no 选项。如果调用了 QCheckBox.setTristate (True)，复选框就会有三个状态：用户选中、未选中或者用户就没有改变它原有的状态。这种三状态方法可能对于表示允许出现 IS NULL 的布尔型数据库域非常有用



QComboBox

该截图给出的是打开了下拉项的 QComboBox。组合框会用一个元素项清单来表示用户的选择，其中能够提供比 QListView 所需的更大空间。调用 QComboBox.setEditable (True) 会让用户要么选择清单中的一项，要么使用用户自己键入的文本。每个组合框中的元素项都会有相应的文本，一个可选的图标，还有可选的数据。可以使用 QComboBox.addItem () 或者 QComboBox.addItems () 来构建组合框，也可以使用自定义的或者 QComboBox.setModel () 来使用内置的 QAbstractItemModel 子类来构建组合框



QDateEdit,  
QDateTimeEdit, and  
QTimeEdit

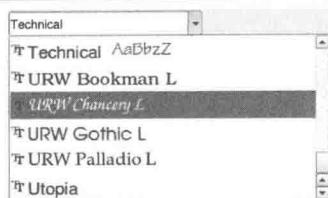
QDateEdit 用来显示和输入日期，QDateTimeEdit 用于日期和时间，QTimeEdit 则用于时间。默认情况下，这些窗口部件使用本地化的日期和时间格式——这里是使用美国本地化格式显示的。可以改变这些格式，也可以设置最小和最大允许日期和时间



QDialogButtonBox

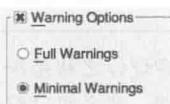
这个窗口部件可用来创建一个行或者列的按钮。这些按钮可以是预定义了角色和文本的标准按钮，也可以是添加进来的带有角色和选项文本的按钮。这个窗口部件会根据这些按钮的角色和所处窗口系统的用户界面规范来自动排列它们

(续表)



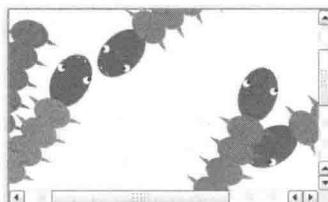
QFontComboBox

PyQt 提供了一种弹出式字体对话框，会使用可用的本地字体对话框。如果打算提供我们自己的字体选项——例如，在工具栏中——可以用 QFontComboBox，像这里向下弹出的那样。对于 Qt 4.0 和 Qt 4.1 来说，最接近的相当效果(但不带字体预览)是使用普通的 QComboBox，可以使用 QFontDatabase.families()返回的清单来构建它



QGroupBox and QRadioButton

群组框可单纯用做可视化的设备群组，或者可以像这里给出的那样用做复选框。如果是复选框，在群组框中的这些窗口部件只有在选中时才可以与用户进行交互。如果只需要框架而不需要标题，就可以用 QFrame 代替。在把一些 QRadioButton 放到群组框中时，它们可以自动获得正确的行为，即用户只能选择其中之一。QComboBox 和 QListWidget 通常要比 QRadioButton 更便捷些



QGraphicsView

这个窗口部件用来查看 QGraphicsScene 中的 QGraphicsItem。在一个场景中可以查看任意数量的 QGraphicsView，每个 QGraphicsView 都可以有起作用的自己的几何变换(例如，缩放和旋转)。如果需要滚动条，它会自动出现。每个 QGraphicsView 都可以提供自己的背景和前景，从而覆盖由场景所提供的背景和前景

A multi-line QLabel,  
showing HTML text  
with *font effects*.

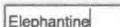
QLabel

QLabel 窗口部件是一个显示窗口部件，可以用来显示图片、普通文本字符串、QTextDocument 或者 HTML。带有加速键(带有下画线的单一字母)的标签(label)可以成为一个窗口部件的“伙伴”(buddy)，从而在按下加速键的时候把光标传给其伙伴



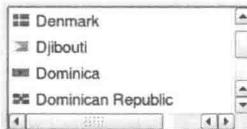
QLCDNumber

这是一个显示窗口部件，能以 7 字段的 LCD 形式显示数字



QLineEdit

这个窗口部件可以接受用户输入的单行文本。可以使用验证程序(例如，QIntValidator 或者 QRegExpValidator)对这些文本进行输入限制，或者通过设置输入掩码保护输入文本，或者两者都用。响应模式可以设置成显示一些“\*”(或者什么也不显示)来代替所输入的文本



QListView and QListWidget

通过这些窗口部件用户可以选择一个元素，或者用一种适当的选择模式，选择多个元素。这些窗口部件可处于清单模式(如图所示)，或者处于图标模式，此时的图标会比在图标下方显示的文本大很多。在使用 QListWidget.setModel() 时，QListWidget 必须要与自定义的或者内置的 QAbstractItemModel 子类联合使用。QListWidget 具有内置模式，所以可以使用 QListWidget.addItem() 和 QListWidget.addItems() 直接添加各个元素。在纵向空间具有较高优先级的地方，也可以用 QComboBox 代替

(续表)



QProgressBar

这个窗口部件可用来向用户显示一项长时间运行操作的进度。通常会使用 `QStatusBar.addWidget()` 或者 `addPermanentWidget()`，把它放进 `QMainWindow` 的状态栏。另外一种方法是弹出 `QProgressDialog`



QPushButton

这些按钮用做动作调用。如果一点击按钮就会弹出一个对话框，那么就通常会在按钮的文本最后添加一个省略号(…). 这些按钮也可以用来弹出菜单(在这种情况下，PyQt 会添加一个小的三角符号)，或者可以设置成单选按钮，在奇数次点击的时候会呈现按下状态。因为在 Qt 4.2 下，大多数应用程序都会使用 `QDialogButtonBox` 而不是仅仅使用 `QPushButton`



QSlider

滑动条通常用来显示比例，也通常与用来显示精确数值的 `QLabel` 或者 `QLCDNumber` 一起联用。滑动条可以纵向或者水平对齐。`QScrollBar` 可以用做类似的目的



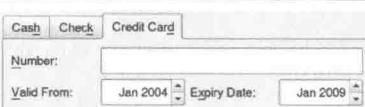
QDoubleSpinBox and QSpinBox

这些窗口部件用来接受和显示数字。显示的数字可以带有前缀或者后缀，也可以具有特定的对齐方式(这里给出的 `QDoubleSpinBox` 就带有一个“\$”前缀)。可以设置数字的最小和最大值，对于 `QDoubleSpinBox` 来说，也可以设置小数点后数字的位数。另外一种方法是，让 `QLineEdit` 与 `QIntValidator` 或者 `QDoubleValidator` 联用

Category	Short Desc.	Long Desc.
1 Actions	Wait	Wait for Information
2 Actions	Progress	Progress to Next Stage
3 Actions	Escalate	Send to Supervisor

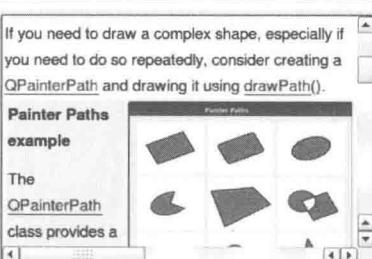
QTableView and QTableWidget

这些窗口部件用来显示表格型数据。`QTableView` 必须与自定义或者内置的 `QAbstractItemModel` 子类联用，比如，`QSqlTableModel`，会使用 `QTableView.setModel()`。`QTableWidget` 有一个内置方法，所以可以直接向其添加元素——例如，可以使用 `QTableWidget.setItem()`。所有这些窗口部件都可以在每个单元格中显示图标和文本，包括在单元格标题中



QTabWidget

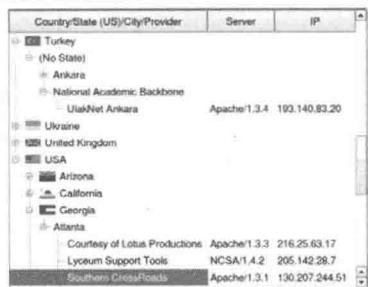
当空间优先，或者只是逻辑化地群组各个窗口部件时，可以使用这个窗口部件。这些 Tab 页有两个设置区，可以出现在上、左、右、下侧，而出现在左侧和右侧时，文本也会予以旋转



QTextEdit and QTextBrowser

这些窗口部件可以显示 HTML，包括清单、表格和图片。`QTextEdit` 也可以用做编辑窗口部件，既可以用做普通文本的编辑，也可以用做 PyQt“Rich 文本”(特别是 HTML，尽管可能还需要一个自定义子类来提供表格和图片的编辑)的编辑。`QTextBrowser` 支持可点击的链接，所以可用来作为简单的 Web 浏览器。这些窗口部件都可以支持 CSS(层叠样式表，Cascading Style Sheets)

(续表)



这些窗口部件可用来显示层次化的数据。QTreeView 使用时必须与自定义或者内置的 QAbstractItemModel 子类的 QTreeView.setModel()一起使用。与所有窗口部件使用模型一样，只有那些对用户可见的数据才可以检索，所以即使是大型数据集合，速度也会非常快。QTreeWidget 有一个内置模型，所以使用 QTreeWidget.insertTopLevelItem() 和 insertTopLevelItems() 可以直接向它添加各个元素，或者通过创建一些 QTreeWidgetItem 来作为其他元素的孩子。

QTreeView and QTreeWidget

## 附录 C 部分 PyQt 类的层次

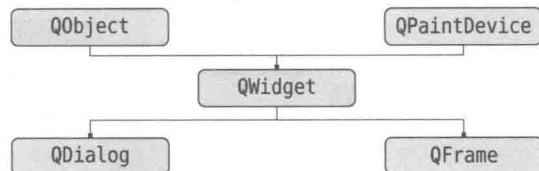


图 C.1 部分基类

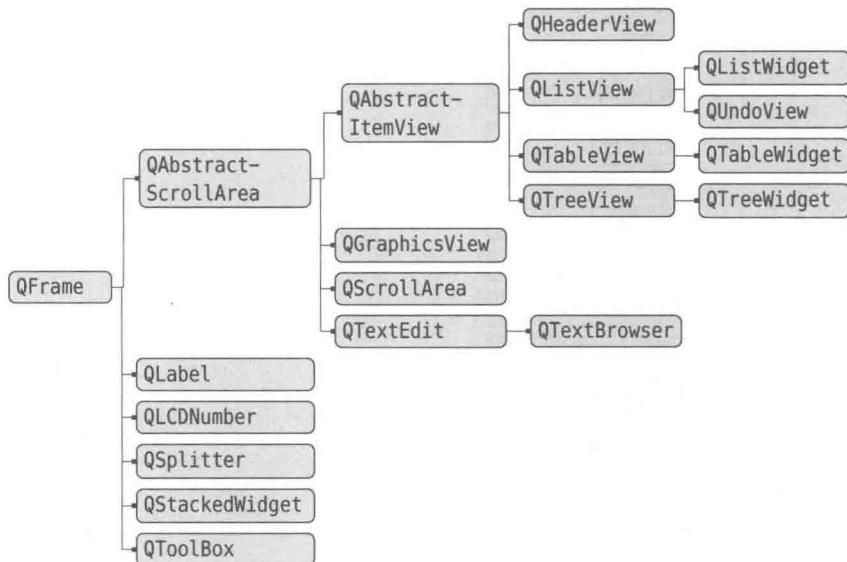


图 C.2 QFrame 中部分类的层次

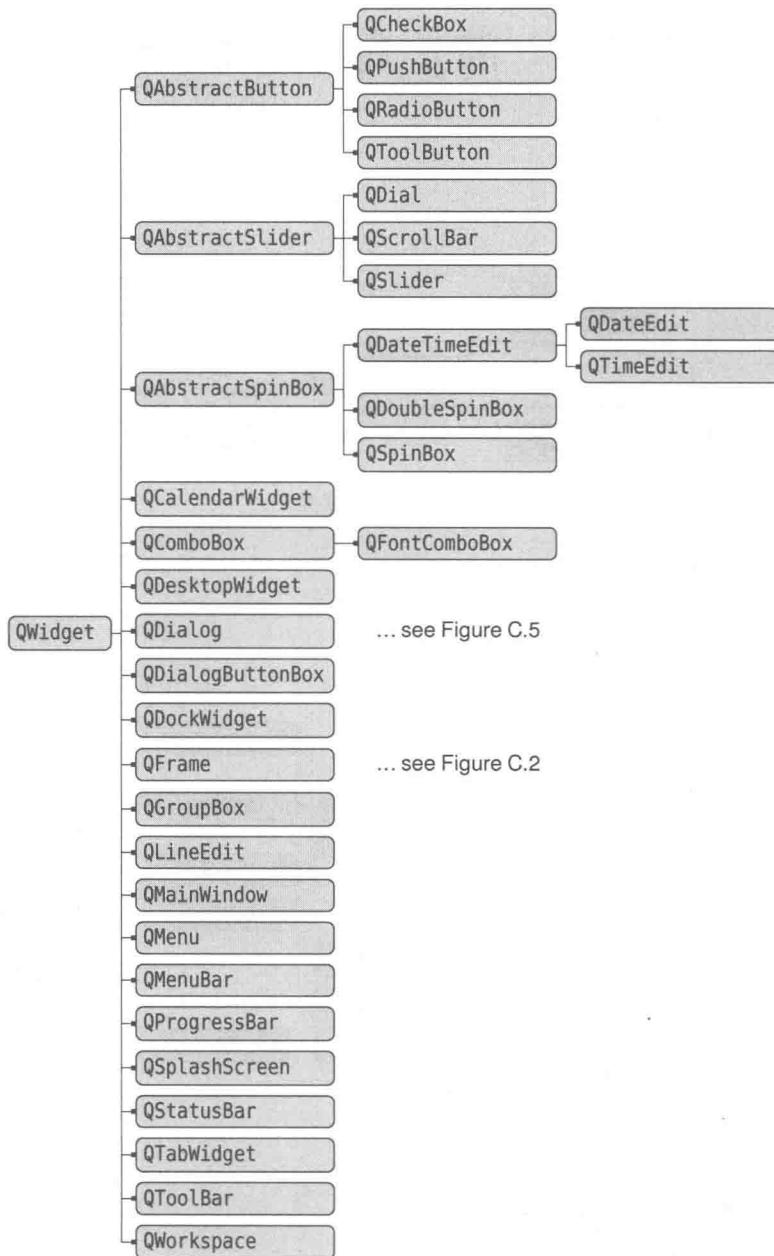


图 C.3 QWidget 中部分类的层次

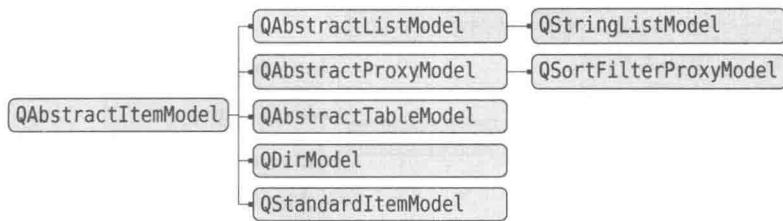


图 C.4 QAbstractItemModel 中部部分类的层次



图 C.5 QDialog 中部部分类的层次

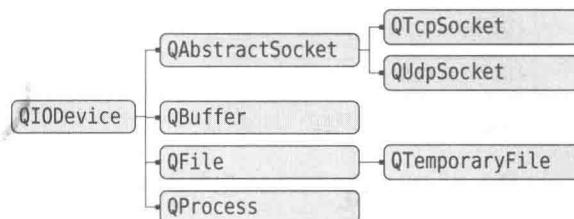


图 C.6 QIODevice 中部部分类的层次

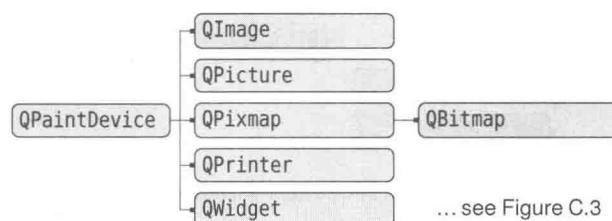


图 C.7 QPaintDevice 中部部分类的层次

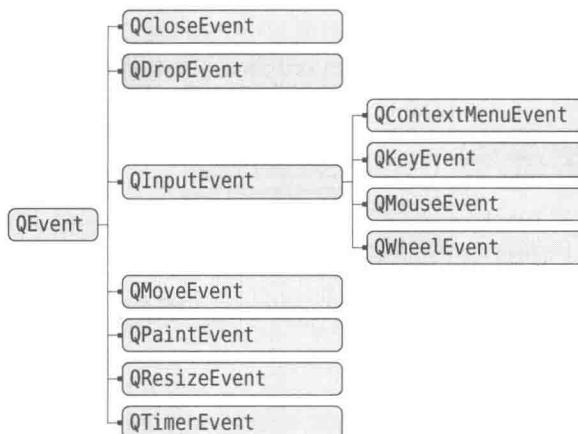


图 C.8 QEvent 中部分类的层次

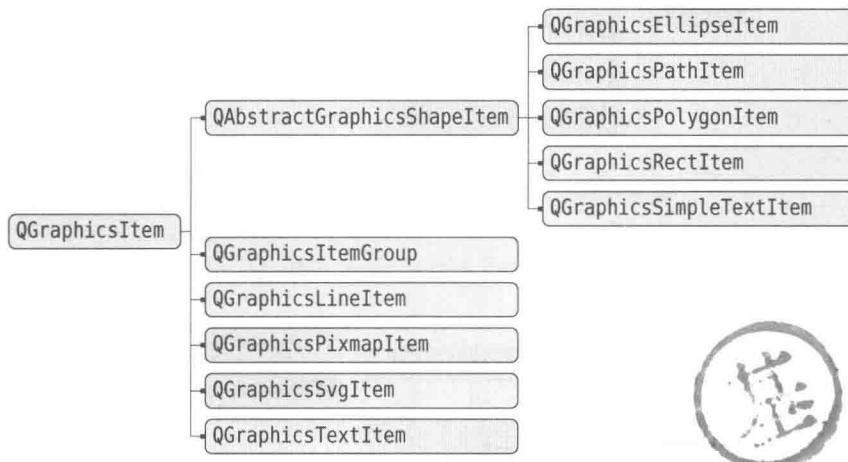


图 C.9 QGraphicsItem 中部分类的层次

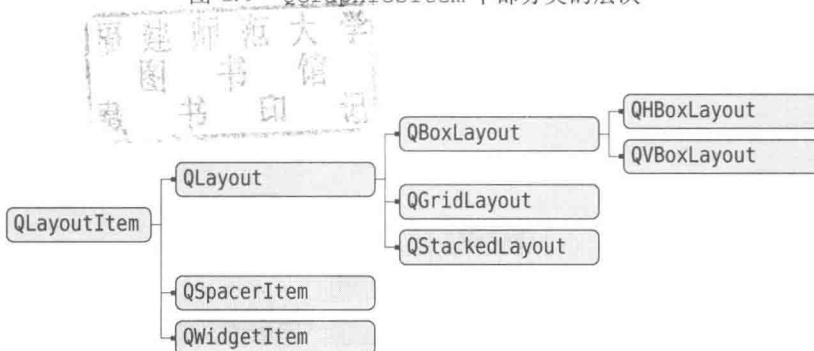


图 C.10 QLayoutItem 中部分类的层次

1480289



T1480289

# Python Qt GUI 快速编程

## — PyQt编程指南

Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming

本书是受公众认可的PyQt编程学习用书之一。作者Mark Summerfield在Qt公司初创时期就任职于此，近年来也一直是Qt和Python开发的知名代码贡献人员，在两者中均拥有良好的经验。Mark在撰写本书的过程中，不断跟踪Python和Qt的实时发展动态，因而无论是书中的例子和风格，还是课后习题的设置，始终秉承传授编程思想和原理方法为主、解决和分析技术难点为辅的写作风格。Mark撰写的多部作品都获得了有软件业界奥斯卡之称的“震撼奖”（Jolt Award）。本书与获奖作品《C++ GUI Qt 4编程》一书的写作风格类似，案例设置通俗易懂，因而是学习PyQt的不可多得的一本好书。

本书主要讲述如何利用Python和Qt开发GUI应用程序的原理、方法和关键技术。

### 本书特点

- 讲解Python基础知识。
- 通过三个例子讲解PyQt GUI应用程序。
- 深入讲述窗口部件布局、事件处理、窗口部件子类化、Qt图形架构和Qt的模型/视图。
- 介绍国际化、网络化和多线程化。

书中所用到的示例程序的源代码可从原书站点www.qtrac.eu（英文）下载，也可直接从站点www.qtcn.org/pyqtbook（中文）下载。



本书相关网址

可扫描二维码



策划编辑：冯小贝  
责任编辑：李秦华  
责任美编：孙焱津



Pearson  
[www.pearson.com](http://www.pearson.com)



9 787121 129806 6 >

定价：79.00 元